# DISEÑO E IMPLEMENTACIÓN DE UN SISTEMA PARA CONTROL DE VUELO Y NAVEGACIÓN GPS DE UN CUADRICÓPTERO - OUTDOOR

Christian Mamani-Mamani

Piura, abril de 2017

FACULTAD DE INGENIERÍA

Máster en Ingeniería Mecánico-Eléctrica con Mención en Automática y Optimización

Mamani, C. (2017). *Diseño e implementación de un sistema para control de vuelo y navegación GPS de un cuadricóptero - Outdoor* (Tesis de Máster en Ingeniería Mecánico-Eléctrica con mención en Automática y Optimización). Universidad de Piura. Facultad de Ingeniería. Piura, Perú.

# U N I V E R S I D A D  D E  P I U R A

FACULTAD DE INGENIERÍA


**“Diseño e implementación de un sistema para control de vuelo y navegación GPS de un cuadricóptero - *Outdoor*”**

Tesis para optar el Título de
Master en Ingeniería Mecánico – Eléctrico con mención en Automática y Optimización

**Christian Yeymi Mamani Mamani**

ASESOR: Dr. Ing. Cesár Alberto Chinguel Arrese

Piura, abril 2017

***"A mi familia, madre, hermanas y a mi padre Clemente por ser mi apoyo y modelo a seguir".***

# Prólogo

El presente proyecto se enmarca dentro de las líneas de investigación de la Facultad de Ingeniería de la "Universidad de Piura", en la cual se viene desarrollando algoritmos para el control de cuadricópteros dentro del grupo de investigación de Laboratorio de Sistemas Automáticos de Control (SAC), centrándose este trabajo en el control de posición para el seguimiento de trayectorias. Para ello se utilizará la plataforma robótica *Robotic Operating System (ROS)* para la creación de este sistema de control, sirviendo como base para futuros trabajos los cuales podrán utilizar esta misma plataforma.

Aunque el proyecto se encuentra dentro de dichas líneas de investigación, se hace énfasis por ser un trabajo pionero en esta universidad, siendo la primera tesis que desarrolla un prototipo basado en la plataforma ROS, la cual brinda en sus repositorios una cantidad de paquetes desarrollados por la comunidad de software libre en aplicaciones de robótica.

Agradecer al CONCYTEC por haberme dado la oportunidad en realizar está maestría, a mi asesor Dr. Ing. Cesar Chinguel Arrese y a las personas que conforman Laboratorio SAC por haberme brindado su apoyo para la realización del presente trabajo.

# Resumen

Este proyecto propone un algoritmo de control para resolver el problema de seguimiento de trayectoria para cuadricópteros para ambientes abiertos utilizando el concepto perfil de velocidad trapezoidal, navegación *GPS*, controladores PID, el sistema embebido comercial *ArduCopter* y el *framework ROS*.

En el capitulo 1 se presenta una revisión del modelo matemático de un cuadricóptero desarrollado por otros investigadores así como el sistema de navegación para la utilización de coordenadas rectangulares. Una vez conocido el comportamiento del cuadricóptero se explicará lo fundamental acerca de *ROS* en el capitulo 2 para utilizar esta herramienta en nuestro sistema de control, también se utiliza paquetes y programas ya desarrollados por otros investigadores tanto para la simulación del cuadricóptero así como para la comunicación entre la estación base y el cuadricóptero construido. En el capitulo 3 se realiza una revisión del controlador PID y el enfoque de nuestro sistema de control en el cuadricóptero. Para el capitulo 4 se realiza las pruebas tanto en simulación como en campo.

Obteniéndose los resultados en simulación y pruebas de campo, se determinó que mejor desempeño tuvo uno de los métodos de perfil de velocidad trapezoidal desarrollados, con esto se planteará posibles mejoras para trabajos futuros y también las enormes ventajas de utilizar este magnífico software como lo es *ROS*.

# Índice

# Introducción

Los avances tecnológicos han permitido un gran desarrollo en el campo de la aeronáutica donde encontramos aplicaciones en los sectores: militares, comerciales y demás. Una de las áreas de intenso desarrollo son los Vehículos Aéreos no Tripulados (VANT) en especial los cuadricópteros.

Los cuadricópteros son uno de los VANT más difundidos y estudiados en la actualidad [1], [2], [3] debido a ciertas ventajas como: maniobrar en lugares reducidos o ingresar en lugares no seguros para cualquiera. De esta forma se le vienen dando diferentes usos, lo que conlleva a que se desarrollen y mejoren los algoritmos de control, interfaz, etc. por parte de los desarrolladores, pero muchas de las aplicaciones y herramientas desarrolladas necesitan de un operador por lo que no es totalmente autónomo.

Otro ámbito que fue desarrollándose a la par, fue el software robótico, los cuales fueron varios creados pero no existía un consenso sobre esto. Así nació la plataforma para robots *Robotic Operating System* (*ROS*), el cual es un marco de código abierto que proporciona toda una serie de servicios y librerías que simplifican considerablemente la creación de aplicaciones complejas para robots, bajo el cual se han desarrollado muchas aplicaciones no solo en cuadricópteros sino para varios sistemas robóticos.

Si bien es cierto que existen muchas otras plataformas robóticas desarrolladas, algunas de estas no son de código abierto lo que limita mejorarlo o revisar las deficiencias del mismo por parte de los investigadores, no obstante existen otras plataformas que si tienen estás característica pero no tienen soporte constante.

Frente a lo mencionado en el laboratorio SAC existe un grupo de investigadores realizando pruebas con cuadricópteros para aplicaciones en agricultura porque vienen siendo utilizados para el monitoreo y recolección de datos de cultivos, plagas, etc. Siendo una de las

necesidades la creación de un algoritmo de control para el seguimiento de trayectoria en entornos abiertos (*outdoor*).

En la presente tesis se realiza la lectura de posición del cuadricóptero utilizando un *Global Positioning System* (*GPS*), se utiliza el sistema embebido *ArduCopter* [4] y *ROS* [5] para entornos abiertos, con esto se desarrollará un sistema de control de trayectoria para cuadricóptero como en [6]. Este sistema de control utilizará el concepto de perfil de velocidad trapezoidal, este perfil calcula varios vectores de datos y un vector de tiempo utilizando trayectos rectos, diferenciándose claramente 3 etapas, las cuales constan de: una aceleración constante (recta con pendiente positiva), una velocidad constante (recta constante) y una desaceleración constante (recta con pendiente negativa) que juntos describen un trapecio.

Al trabajar en un plataforma de software de código abierto como lo es *ROS*, crea la posibilidad de que otros desarrolladores puedan utilizarlo, mejorarlo y servir de base para proyectos más ambiciosos, con esto en cuenta familiarizarse con *ROS* será importante, ya que se aprovecha paquetes y librerías disponibles en los repositorios ya probados, lo que permitirá centrarnos sólo en el algoritmo para el sistema de control de trayectoria.

# Capítulo 1

# Base conceptual

## 1.1. Descripción y modelado de un cuadricóptero

El cuadricóptero es un sistema robótico formado por cuatro motores en formación coplanaria[1] cuadrada, en cuyos extremos del cuadrado se encuentran los motores, cada motor está a la misma distancia de su centro masa del cuadricóptero. Para poder controlarlo, lo que se hace es variar las velocidades angulares de los 4 motores eléctricos, la configuración de estos motores es tal que se evita el par de arrastre. Los 2 motores de una misma línea cruzada girarán en un mismo sentido mientras los otros 2 lo harán en sentido opuesto, de esta forma se puede lograr trasladar, rotar o permanecer suspendido el cuadricóptero tal y como se observa en la figura 1.1.

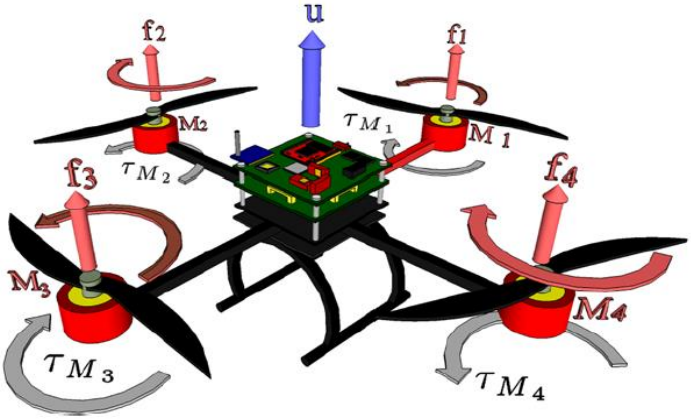



Figura 1.1 Fuerzas presentes en el cuadricóptero para su funcionamiento [2].

[1] Puntos incluidos en un mismo plano.

Para las simulaciones de nuestro sistema de control, se debe contar con un buen modelo del sistema debido a las siguientes razones:

- Comprender la dinámica del sistema frente a perturbaciones, la acción de control aplicado, sensores necesarios, las limitaciones del sistema (saturación de los actuadores por poner ejemplo), etc.
- Llevar el sistema del mundo real a un entorno de simulación, por obvias razones primero se experimentará en simuladores robóticos más adelante serán presentados, para que luego pueda ser implementado físicamente.

Otro de los puntos a tener en cuenta que al ser un sistema muy complejo, se tiene que partir de varias suposiciones, esto nos ayudará a obtener un modelo adecuado, trabajos anteriores como de Schermuk [7] muestran que tales simplificaciones no afecta en mucho a la hora del diseño del control, tales suposiciones son:

- El centro de masa y el marco de referencia intrínseco del cuadricóptero coinciden espacialmente.
- Se desprecia el efecto suelo, es decir no se considera el aumento de empuje producido por proximidad a la superficie de la Tierra.
- El cuerpo del cuadricóptero es completamente rígido.
- Tanto las fuerzas de empuje y arrastre se consideran proporcionales al cuadrado de las velocidades angulares de los motores.

Para comprender más este sistema robótico vamos a ver los movimientos que realiza y así familiarizarnos con él, para más adelante revisar la parte matemática del mismo y nos ayudará a comprender su comportamiento en un entorno de simulación.

### 1.1.1. Descripción del cuadricóptero

#### 1.1.1.1. Movimientos básicos

El cuadricóptero cuenta con los siguientes movimientos:

***a.*** ***Roll.*** Conocido también como el movimiento de alabeo, movimiento de rotación sobre el eje $x$, una variación o diferencial en las velocidades angulares de los motores de este eje permite desplazar el cuadricóptero hacia la derecha o bien a la izquierda.

***b.*** ***Pitch***. O cabeceo, movimiento de rotación sobre el eje $y$, cuyo movimiento se logra con el diferencial de velocidad angular de los motores en el eje mencionado, logrando desplazar hacia adelante o hacia atrás.

**c.** ***Yaw***. Denominado alternativamente como guiñada, movimiento de rotación sobre el eje $z$, este movimiento se logra teniendo dos motores del mismo eje cruzado a la misma velocidad angular respectivamente, el sentido de giro dependerá de que pareja de motores tiene mayor velocidad angular, con esto logrará girar en sentido horario o anti-horario el cuadricóptero.

d. **Empuje**. Si los 4 motores a la vez ejercen la misma fuerza hacia arriba, entonces este ascenderá o caso contrario descenderá a lo largo del eje $z$, con esto se logra controlar la altura del cuadricóptero.

#### 1.1.1.2. Modos de vuelo

Los *UAVs*[2] tienen diferentes configuraciones o modos de vuelo dependiendo de la necesidad del usuario, entre los cuales tenemos:

- **Manual:** Es de tipo lazo abierto, donde el piloto autónomo calcula los valores de las velocidades angulares para los motores a partir de los controles de roll, pitch, yaw y el empuje enviados por el usuario.
- **Autónomo:** Se controla mediante la navegación *GPS*, en la mayoría de casos se utiliza una serie de puntos (*waypoints*) que son previamente definidos y enviados al cuadricóptero desde una estación de control (base).
- **Estabilizado:** Es parecido al modo manual pero este modo tiene un lazo de control cerrado para la orientación es decir los controles se manejan de forma manual pero el *UAV* se estabiliza automáticamente en el aire.

### 1.1.2. Modelado del cuadricóptero

Existen varias aproximaciones sobre el modelamiento del cuadricóptero, uno de los primeros trabajos es de Bouabdallah [1] el cual presenta dos de las metodologías más usadas hoy en día para el modelamiento, las metodologías usadas son:

- **Modelamiento Newton-Euler:** Basada en la Ley de Newton.
- **Modelamiento de Euler-Lagrange:** Basado en la conservación de la energía.

Ambos métodos trabajan con un marco de referencia inercial y otra en el cuadricóptero. En el presente trabajo me baso en el Modelo de Newton-Euler desarrollado en el trabajo de Vianna [3].

### 1.1.3. Orientación del cuadricóptero

Para describir el movimiento del cuadricóptero se presentará como estimar la posición y orientación del vehículo con respecto a un sistema de referencia inercial.

#### 1.1.3.1. Sistema de ejes

El cuadricóptero, como sistema rígido, está caracterizado por un sistema de coordenadas ligado a él y con origen en su centro de masa [3] como se muestra en la figura 1.2.

[2] *Unmanned Aerial Vehicle*, vehículo aéreo no tripulado lo que en español seria VANT.

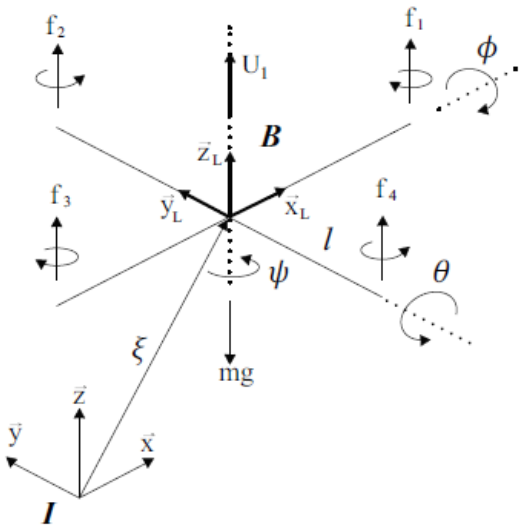



Figura 1.2 Posición y orientación del cuadricóptero respecto a un sistema de referencia [3].

Este sistema se define considerando $B=\{\vec{x}_L,\vec{y}_L,\vec{z}_L\}$ como el sistema de coordenadas fijo al cuadricóptero, donde el eje $\vec{x}_L$ es la dirección normal de ataque del cuadricóptero, el eje $\vec{y}_L$ es ortogonal a $\vec{x}_L$ y es positivo hacia la derecha en el plano horizontal, mientras que $\vec{z}_L$ está orientado en sentido ascendente y ortogonal al plano formado por los ejes $\vec{x}_L$ y $\vec{y}_L$. El sistema de coordenadas inercial $I=\{\vec{x},\vec{y},\vec{z}\}$ se considerará fijo con respecto a la Tierra.

En el presente trabajo se designará el vector $\xi=\{x,y,z\}$, como la posición del centro de masa del cuadricóptero con respecto al sistema inercial $I$. Así mismo, la orientación del vehículo se supondrá dad por una matriz de rotación $R_I = B \rightarrow I$, donde $R_I$ es una matriz de rotación ortonormal[3].

### 1.1.3.2. Ángulos de *Tait - Bryan* y matrices de cambio de fase

La rotación de un cuerpo rígido puede ser representado utilizando varios métodos: ángulos de Euler, cuaterniones, etc. A través de 12 definiciones independientes de los ángulos de Euler se puede representar la orientación relativa de dos sistemas de coordenadas. La convención $x,y,z$ (giro alrededor de $x,y,z$) se utiliza en aplicaciones de ingeniería aeroespacial y son conocidos como los ángulos de *Tait – Bryan* o *Ángulos de Cardano* [8].

[3] Ortonormal se da cuando un conjunto de vectores es ortogonal y la norma de cada uno de sus vectores es igual a la unidad.

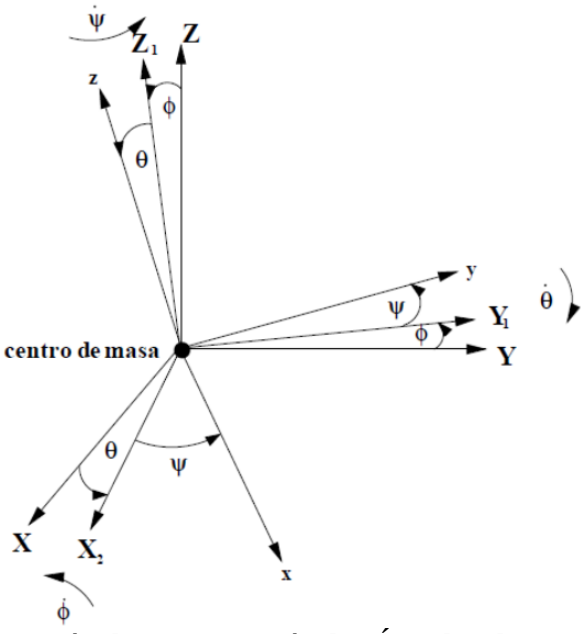



Figura 1.3. Rotación de un cuerpo rígido (Ángulos de Tait-Bryan) [3].

Estos ángulos de Tait-Bryan son tres, usados para describir una rotación general en el espacio euclidiano tridimensional a través de tres rotaciones sucesivas entorno de ejes del sistema móvil en el cual están definidos como en la figura 1.3.

- **Rotación según $\vec{x}$ de $\phi$ :** El primer giro corresponde al ángulo *roll* o de balanceo $\phi$ y se realiza alrededor del eje $\vec{x}$.

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -sen\phi \\ 0 & sen\phi & \cos\phi \end{bmatrix} \begin{bmatrix} x_L \\ y_L \\ z_L \end{bmatrix} \tag{1.1}$$

Donde se define:

$$R(x,\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -sen\phi \\ 0 & sen\phi & \cos\phi \end{bmatrix} \tag{1.2}$$

- **Rotación según $\vec{y}$ de $\theta$ :** Segundo giro corresponde al ángulo *pitch* o de cabeceo $\theta$ y se realiza alrededor del eje $\vec{y}$, para dejar el eje $\vec{z}$.

$$\begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & sen\theta \\ 0 & 1 & 0 \\ -sen\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \tag{1.3}$$

Donde se define:

$$R(y,\theta)=\begin{bmatrix}\cos\theta & 0 & sen\theta\\ 0 & 1 & 0\\ -sen\theta & 0 & \cos\theta\end{bmatrix} \tag{1.4}$$

- **Rotación según $\vec{z}$ de $\psi$ :** El último giro corresponde al ángulo *yaw* o de guiñada $\psi$ y se realiza alrededor del eje $\vec{z}$, a partir del nuevo eje $\vec{z}_L$ para llevar el cuadricóptero a su posición final.

$$\begin{bmatrix}x\\ y\\ z\end{bmatrix}=\begin{bmatrix}\cos\psi & -sen\psi & 0\\ sen\psi & \cos\psi & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x_2\\ y_2\\ z_2\end{bmatrix} \tag{1.5}$$

Donde se define:

$$R(z,\psi)=\begin{bmatrix}\cos\psi & -sen\psi & 0\\ sen\psi & \cos\psi & 0\\ 0 & 0 & 1\end{bmatrix} \tag{1.6}$$

### 1.1.3.3. Matriz de rotación de transformación de velocidades angulares

A partir de las rotaciones presentadas en las ecuaciones 1.1, 1.3, 1.5, se definen las matrices de rotación 1.2, 1.4, 1.6 que representan la orientación del cuerpo rígido rotando alrededor de cada eje: $R(x,\phi)$, $R(y,\theta)$, $R(z,\psi)$.

La matriz de rotación completa de $B$ respecto a $I$ como se ve en la figura 1.2, llamada Matriz de Coseno Directa [3], está dado por:

$$R_I = R(x,\phi)\bullet R(y,\theta)\bullet R(z,\psi) \tag{1.7}$$

$$R_I=\begin{bmatrix}cos\psi cos\theta & cos\psi sen\theta sen\phi - sen\psi cos\phi & \cos\psi sen\theta cos\phi + sen\psi sen\phi\\ sen\psi cos\theta & sen\psi sen\theta sen\phi + cos\psi cos\phi & sen\psi sen\theta cos\phi - cos\psi sen\phi\\ -sen\theta & cos\theta sen\phi & cos\theta cos\phi\end{bmatrix} \tag{1.8}$$

La aplicación más útil de la transformación de cantidades angulares relaciona las velocidades angulares $p$, $q$ y $r$ en los ejes del sistema de referencia móvil y las componentes de velocidad angular $\dot{\phi}$, $\dot{\theta}$ y $\dot{\psi}$ respecto al sistema de referencia fijo.

Las velocidades angulares $p$, $q$ y $r$ son en referencia al cuadricóptero $B=\{\vec{x}_L,\vec{y}_L,\vec{z}_L\}$, por tanto considerando cada rotación necesaria para hacer que los ejes de este marco de referencia coincidan con los ejes del sistema de referencia fijo $I=\{\vec{x},\vec{y},\vec{z}\}$. Primero se rota alrededor a $\vec{x}_L$ un ángulo $\phi$ con una velocidad angular $\dot{\phi}$, después se gira alrededor de $\vec{y}_L$

con un ángulo $\theta$ con velocidad angular $\dot{\theta}$, para que al último rotar alrededor a $\vec{z}_L$ un ángulo $\psi$ con velocidad angular $\dot{\psi}$. Todo esto se puede representar de la siguiente manera:

- $p$ es igual a la suma de las componentes *de* $\dot{\phi}$, $\dot{\theta}$ y $\dot{\psi}$ proyectadas sobre $\vec{x}_L$.

$$p = \dot{\phi} - \dot{\psi}\, sen\theta \tag{1.9}$$

- $q$ es igual a la suma de las componentes *de* $\dot{\phi}$, $\dot{\theta}$ y $\dot{\psi}$ proyectadas sobre $\vec{y}_L$.

$$q = \dot{\theta} cos\phi - \dot{\psi}\, sen\phi cos\theta \tag{1.10}$$

- $r$ es igual a la suma de las componentes *de* $\dot{\phi}$, $\dot{\theta}$ y $\dot{\psi}$ proyectadas sobre $\vec{z}_L$.

$$r = \dot{\psi} cos\phi cos\theta - \dot{\theta}\, sen\phi \tag{1.11}$$

En notación matricial se tiene:

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} 1 & 0 & -sen\theta \\ 0 & cos\phi & sen\phi cos\theta \\ 0 & -sen\phi & cos\phi \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{1.12}$$

Para ángulos pequeños se puede realizar la siguiente aproximación:

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{1.13}$$

### 1.1.4. Modelo aerodinámico de acuerdo formulación Newton – Euler

Las ecuaciones de movimiento del cuadricóptero son obtenidos mediante la formulación Newton-Euler. Este modelo es presentado como una aproximación alternativa. Además esta aproximación provee una clara comprensión de las fuerzas y momentos aplicados al cuadricóptero [3].

Las ecuaciones dinámicas de un cuerpo rígido sujeto a fuerzas externas aplicadas al centro de masa y expresado en el cuerpo del sistema pueden obtenerse a través de Newton-Euler como:

$$\begin{bmatrix} mI_{3x3} & 0 \\ 0 & J \end{bmatrix} \begin{bmatrix} \dot{V} \\ \dot{\omega} \end{bmatrix} + \begin{bmatrix} \omega \times mV \\ \omega \times J\omega \end{bmatrix} = \begin{bmatrix} F_B \\ \tau_B \end{bmatrix} \tag{1.14}$$

Donde:

$m$ = masa total del cuadricóptero.
$I_{3x3}$ = matriz de identidad de orden 3.
$J$ = matriz de inercia de orden 3.
$V$ = vector de velocidad traslacional en el sistema B.
$\omega$ = vector de velocidad angular en el sistema B.

Dado las suposiciones la matriz de inercia queda definida como:

$$J = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \tag{1.15}$$

El término $\omega \times mV = 0$ de la ecuación 1.14 se hace cero, por que hicimos la suposición de que el sistema de coordenadas coincida con el centro de masa de nuestro sistema, lo cual resulta:

$$\begin{bmatrix} mI_{3x3} & 0 \\ 0 & J \end{bmatrix} \begin{bmatrix} \dot{V} \\ \dot{\omega} \end{bmatrix} + \begin{bmatrix} 0 \\ \omega \times J\omega \end{bmatrix} = \begin{bmatrix} F_B \\ \tau_B \end{bmatrix} \tag{1.16}$$

Considerando los siguientes vectores:

- Vector de posición: $\xi = \begin{bmatrix} x & y & z \end{bmatrix}^T$
- Vector de velocidad angular: $\omega = \begin{bmatrix} p & q & r \end{bmatrix}^T$

Las ecuaciones de movimiento de un cuerpo rígido se pueden escribir de la siguiente forma:

$$\dot{\xi} = v \tag{1.17}$$
$$m\dot{v} = R_I F_B \tag{1.18}$$
$$J\dot{\omega} = -\omega \times J\omega + \tau_B \tag{1.19}$$

En la cual se tiene los vectores de velocidad:

$$\dot{\xi} = \begin{bmatrix} \dot{x} & \dot{y} & \dot{z} \end{bmatrix}^T \tag{1.20}$$
$$\dot{\omega} = \begin{bmatrix} \dot{p} & \dot{q} & \dot{r} \end{bmatrix}^T \tag{1.21}$$

Se tiene, $F_B \in B$ y $\tau_B \in B$, son las fuerzas y pares externos correspondientes aplicados al cuerpo del cuadricóptero, que resultan en su propio peso, vector de fuerzas aerodinámicas, en el empuje y en los pares desarrollados por los cuatro motores. Estas fuerzas y pares generados pueden expresarse como:

$$R_I F_B = -mg \bullet E_3 + R_{I_{E_3}} \left( \sum_{i=1}^{4} b\Omega_i^2 \right) \tag{1.22}$$

$$\tau_B = -\left( \sum_{i=1}^{4} J_R \left( \omega \times E_3 \right) \bullet \Omega_i \right) + \tau_a \tag{1.23}$$

Donde:

$g$ = gravedad.
$J_R$ = momento de inercia rotacional del rotor alrededor de su eje
$b$ = coeficiente de empuje de los rotores.
$\Omega_i$ = velocidad angular del i – ésimo rotor.

La sumatoria de fuerzas traslacionales que actúan sobre el cuadricóptero, está compuesto por el empuje total generado por la suma de los cuatro motores, la fuerza gravitacional y la fuerza aerodinámica. La fuerza principal $U_1$ o también conocida como la entrada principal de control aplicada al cuadricóptero es:

$$U_1 = \left( \sum_{i=1}^{4} f_i \right) = \left( \sum_{i=1}^{4} b\Omega_i^2 \right) \tag{1.24}$$

Donde:

$f_i$ = fuerza generada por el rotor i – ésimo rotor.

Los pares generados son debido a la diferencia de los empujes entre los motores, es decir, el movimiento de cabeceo (*pitch*) se debe al empuje diferencial existente en los motores frontal y trasero, el movimiento de balanceo (*roll*) es producto al empuje diferencial de los motores de la izquierda y la derecha, el movimiento de guiñada se debe a la diferencia de los pares entre los dos motores que giran en sentido horario y los otros 2 que giran en sentido anti-horario, matemáticamente lo expuesto se expresa como:

$$\tau_a = \begin{bmatrix} (f_2 - f_4)l \\ (f_3 - f_1)l \\ \sum_{i=1}^{4} \tau_{M_i} \end{bmatrix} = \begin{bmatrix} lb\left(\Omega_4^2 - \Omega_2^2\right) \\ lb\left(\Omega_1^2 - \Omega_3^2\right) \\ d\left(\Omega_2^2 + \Omega_4^2 - \Omega_1^2 - \Omega_3^2\right) \end{bmatrix} = \begin{bmatrix} lU_2 \\ lU_3 \\ U_4 \end{bmatrix} \tag{1.25}$$

Reemplazando las ecuaciones 1.22, 1.23, 1.24, 1.25 en las ecuaciones 1.17, 1.18, 1.19, logramos obtener las ecuaciones que describen la dinámica del cuadricóptero:

$$\ddot{x} = \left(cos\psi.sen\theta.cos\phi + sen\psi.sen\phi\right)\frac{U_1}{m}$$

$$\ddot{y} = \left(sen\psi.sen\theta.cos\phi - cos\psi.sen\phi\right)\frac{U_1}{m}$$

$$\ddot{z} = -g + \left(cos\theta.cos\phi\right)\frac{U_1}{m}$$

$$\dot{\phi} = p + q.sen\phi.tg\theta + r.cos\phi.tg\theta$$

$$\dot{\theta} = q.cos\phi - r.sen\phi$$

$$\dot{\psi} = q.sen\phi.sec\theta + r.cos\phi.sec\theta$$

$$\dot{p} = \frac{I_{yy} - I_{zz}}{I_{xx}}.q.r - \frac{J_R}{I_{xx}}.\Omega.q + \frac{l.U_2}{I_{xx}}$$

$$\dot{q} = \frac{I_{zz} - I_{xx}}{I_{yy}}.p.r + \frac{J_R}{I_{yy}}.\Omega.p + \frac{l.U_3}{I_{yy}}$$

$$\dot{r} = \frac{I_{xx} - I_{yy}}{I_{zz}}.p.q + \frac{U_4}{I_{zz}} \qquad (1.26)$$

Como se tiene motores girando en sentido horario y anti-horario, se debe tener en cuenta velocidades angulares con signo negativo para la ecuación:

$$\Omega = \Omega_1 + \Omega_2 + \Omega_3 + \Omega_4 \qquad (1.27)$$

Donde:

$\Omega_i$ = velocidad angular general.

De la ecuación 1.25, se tiene la relación proporcional entre las fuerzas y las velocidades angulares de los motores, donde las constantes de proporción son el factor de empuje y el factor de arrastre:

$$\begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix} = \begin{bmatrix} b\left(\Omega_1^2 + \Omega_2^2 + \Omega_3^2 + \Omega_4^2\right) \\ b\left(\Omega_4^2 - \Omega_2^2\right) \\ b\left(\Omega_3^2 - \Omega_1^2\right) \\ d\left(\Omega_2^2 + \Omega_4^2 - \Omega_1^2 - \Omega_3^2\right) \end{bmatrix} \qquad (1.28)$$

## 1.2. Sistemas de navegación

El poder ubicar en papel un lugar específico señalando ciertas características del lugar así como de su entorno, motivó al hombre a conocer casi en totalidad los diferentes espacios geográficos de la Tierra y más aún en la actualidad se realizan mapas del universo.

Para interpretar un mapa es imprescindible conocer algunos elementos, como son: contenido, su naturaleza y fuente de información. Además de conocer el sistema de coordenadas

utilizado para poder ubicar cualquier punto en el mapa. Las cartas[4] referencian el sistema de coordenadas utilizado, con el objetivo de mostrar la ubicación geográfica y con ella la de todos los puntos y detalles contenidos en el mismo además de características métricas.

### 1.2.1. Sistemas de coordenadas

Todo mapa está referido por lo menos a un sistema de coordenadas universal, cuyo objetivo es el de brindar su ubicación geográfica y con ella la de todos los puntos y detalles contenidos en la misma, también la de facilitar más detalles de las características métricas del mapa. En relación a la cartografía formal lo que se discute en este apartado está relacionado con la ubicación espacial en un marco geográfico de referencia y en este sentido se tratarán los sistemas de coordenadas geodésicas o curvilíneas y el sistema de coordenadas rectangulares.

#### 1.2.1.1. Sistema de coordenadas geodésicas

Es un sistema curvilíneo debido a que los círculos máximos que lo definen son líneas curvas, un circulo máximo en una esfera es cualquier circulo cuyo plano contiene el centro de la esfera y por lo tanto puede haber un número infinito de círculos máximos.


Figura 1.4: Sistema de coordenadas geodésicas.
Fuente: *Google Earth*.

El mapa está limitada por un formato constituido por líneas que representan paralelos de latitud (norte - sur) y meridianos de longitud (este -oeste), las que aparentemente forman rectángulos. Los paralelos de latitud son círculos menores paralelos al Ecuador. A este conjunto de líneas se le llama comúnmente canevá[5] y siendo rigurosos las líneas no son paralelas como en un rectángulo, sino que constituyen lo que se llama un cuadrángulo[6]. En efecto, se puede reconocer que si bien los paralelos son como su nombre lo indica, los meridianos son convergentes en los polos como se observa en la figura 1.4.

[4] Documento extendido que describe gráficamente el planeta en su conjunto o partes de la misma.
[5] Gradícula o intersección de paralelos y meridianos que resulta de un sistema de proyección y constituye la base de una carta geográfica. http://espacio-geografico.over-blog.es/article-caneva-108482168.html.
[6] Aplicado a cualquiera de las figuras geométricas que tiene cuatro ángulos que pueden ser homogéneos o aplicado a un polígono de distintas formas que cumple con estas cualidades.

### 1.2.1.2. Sistema de coordenadas *Universal Transversal of Mercator* (*UTM*)

Es una proyección cilíndrica transversal al eje de la tierra y conforme (conserva las formas o contornos pero no las áreas). Su representación se realiza en husos[7] de 6° de anchura (668 km.), de forma que el globo terráqueo queda cubierto por 60 husos enumerados desde el 1 hasta el 60, de oeste a este desde el anti-meridiano de Greenwich (180°) como se muestra en la figura 1.5.

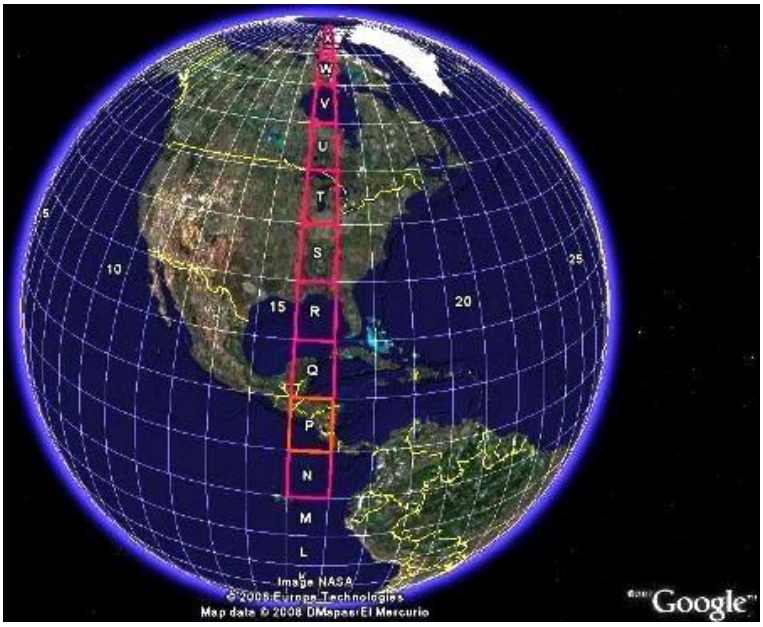



Figura 1.5: Sistema de coordenadas *UTM*.
Fuente: *Google Earth*.

A su vez cada huso está dividido en cuadrados de 100 km. de lado, designados por dos nuevas letras mayúsculas y cada cuadrado en una retícula kilométrica, cuya referencia numérica aparece expresada en los márgenes del mapa.

### 1.2.1.3. Conversión de coordenadas geodésicas a coordenadas *UTM*

Para el presente trabajo, el algoritmo de control utilizará coordenadas *UTM*, pero debido a que los *GPS* proporcionan coordenadas geodésicas, se realizará la conversión de estas coordenadas geodésicas a *UTM*, para ello nos basaremos en las fórmulas y trabajo realizado por Coticchia-Surace [9] y Ortiz [10], por ello es que se implementará un algoritmo para llevar a cabo dicha tarea, utilizando variables de punto flotante y doble precisión, en este apartado se muestra como se deducen las fórmulas.

Tomaremos un punto de referencia en coordenadas geográficas como ejemplo, para poder obtener sus coordenadas en *UTM*, el punto de ejemplo será el área de ingeniería Mecánica Eléctrica (IME) de la Universidad de Piura (UDEP-Perú), con las siguientes coordenadas geográficas:

[7] Posiciones geográficas que ocupan todos los puntos comprendidos entre dos meridianos.

- Longitud: $\lambda = 80.637°$ Oeste
- Latitud: $\varphi = 5.172°$ Sur

Primero necesitaremos los datos básicos de la geometría del elipsoide de Hayford [8], es decir los semiejes mayor $a$ y menor $b$, partiendo de estos datos deduciremos los demás:

- $a = 6378388$
- $b = 6356911.946$

**Cálculos Previos**

- **Geometría del elipsoide:** Calcularemos la excentricidad[8], la segunda excentricidad, el radio polar de curvatura y el aplanamiento:

  Excentricidad:

  $$e = \frac{\sqrt{a^2 - b^2}}{a} \rightarrow e = \frac{\sqrt{6378388^2 - 6356911.94613^2}}{6378388} = 0.081991889$$

  Segunda Excentricidad:

  $$e' = \frac{\sqrt{a^2 - b^2}}{b} \rightarrow e' = \frac{\sqrt{6378388^2 - 6356911.94613^2}}{6356911.94613} = 0.082268889$$

  Para cálculos posteriores utilizaremos la segunda excentricidad al cuadrado:

  $$e'^2 = 0.00676817019657$$

  Ahora calculamos el radio polar de curvatura y el aplanamiento:

  Radio polar de curvatura:

  $$c = \frac{a^2}{b} \rightarrow c = \frac{6378388^2}{6356911.94613} = 6399936.60811$$

  Aplanamiento:

  $$\alpha = \frac{a - b}{a} \rightarrow \alpha = \frac{6378388 - 6356911.94613}{6378388} = 0.003367003 \approx \frac{1}{127}$$

  Para las ecuaciones de Cottiacchia-Surace no son necesarias el aplanamiento y la primera excentricidad, pero se las incluyó debido a que habitualmente los parámetros del elipsoide se dan como el semieje mayor $a$ y el aplanamiento $\alpha$, o bien como el semieje mayor $a$ y la excentricidad $e$, conociendo estas fórmulas también podríamos calcular el semieje menor $b$.

[8] Parámetro que determina el grado de desviación de una sección cónica con respecto a una circunferencia.

- **Longitud y latitud:** Lo primero que haremos es convertir todos los datos de sexagesimales a radianes, de acuerdo a las fórmulas:

  Grados sexagesimales:

  $$(^{o}) = grados + \frac{minutos}{60} + \frac{segundos}{3600}$$

  Radianes:

  $$rad = \frac{(^{o}).\pi}{180}$$

  Con esto se obtiene:

  Longitud=-80.637 y en radianes como -1.40738114893
  Latitud=-5.172 y en radianes como -0.0902684289131

  Lo siguiente es calcular el signo de la longitud, para ello se tiene en cuenta que si la longitud está referida al Oeste del meridiano de *Greenwich*, entonces la longitud es negativa (-), caso contrario al situarse al Este será positiva (+).

- **Huso:** Calculamos el huso donde caen las coordenadas a convertir:

  Huso= Parte entera de $\left[\frac{(^{o}\text{longitud})}{6} + 31\right] \rightarrow huso = \left[\frac{\text{-80.637}}{6} + 31^{o}\right] = 17^{o}$

  Ahora tenemos que obtener el meridiano central del huso donde se encuentra las coordenadas:

  $$\lambda_0 = 6 \times huso - 183^{o} \rightarrow \lambda_0 = 6 \times 17 - 183^{o} = -81^{o}$$

  Procedemos a calcular la distancia angular que hay entre la longitud del punto con el que operamos y el meridiano central del huso. Los datos procesados deben estar en radianes:

  $$\Delta\lambda = \lambda - \lambda_0 \rightarrow \Delta\lambda = \text{-1.40738114893} - \left(\frac{-81^{o} \times \pi}{180^{o}}\right) = 0.00633554518474$$

**Ecuaciones de Cottiacchia-Surace**

Calcularemos una serie de parámetros que van en continuación uno tras otro y que son parámetros que necesitaremos para las ecuaciones de Cottiacchia-Surace, para más detalle y explicación se puede consultar su trabajo [9].

- **Cálculo de parámetros:**

$$A = cos\varphi \times sen\Delta\lambda = cos\left(\text{-}0.090268\right) \times sen\left(0.006335\right)$$

$$A = 0.006309$$

$$\xi = \frac{1}{2} ln\left[\frac{1+A}{1-A}\right] = \frac{1}{2} ln\left[\frac{1+\left(0.006309\right)}{1-\left(0.006309\right)}\right]$$

$$\xi = 0.006309$$

$$\eta = arctan\left(\frac{tan\varphi}{cos\Delta\lambda}\right) - \varphi = arctan\left(\frac{tan\left(\text{-}0.090268\right)}{cos\left(0.0063355\right)}\right) - \left(\text{-}0.090268\right)$$

$$\eta = \text{-}1.8018 \times 10^{-6}$$

$$\nu = \frac{0.9996 \times c}{\sqrt{\left(1 + e'^2 \times cos^2\varphi\right)}} = \frac{0.9996 \times 6399936.60811}{\sqrt{\left(1 + 0.006768 \times cos^2\left(\text{-}0.090268\right)\right)}}$$

$$\nu = 6376010.80889$$

$$\zeta = \frac{e'^2}{2} \times \xi^2 \times cos^2\varphi = \frac{0.006768}{2} \times 0.0063095^2 \times cos^2\left(\text{-}0.090268\right)$$

$$\zeta = 1.3363 \times 10^{-7}$$

$$A_1 = sen\left(2 \times \varphi\right) = sen\left(2 \times \text{-}0.090268\right)$$

$$A_1 = \text{–}0.179557$$

$$A_2 = A_1 \times cos^2\varphi \rightarrow A_2 = \text{–}0.179557 \times cos^2\left(\text{–}0.090268\right)$$

$$A_2 = \text{–}0.178098$$

$$J_2 = \varphi + \frac{A_1}{2} = \text{–}0.090268 + \frac{\text{–}0.179557731787}{2}$$

$$J_2 = \text{–}0.180047$$

$$J_4 = \frac{3 \times J_2 + A_2}{4} = \frac{3 \times \left(\text{–}0.180047\right) + \left(\text{–}0.178098\right)}{4}$$

$$J_4 = \text{–}0.179560$$

$$J_6 = \frac{5 \times J_4 + A_2 \times cos^2\varphi}{3} = \frac{5 \times \left(\text{–}0.179560\right) + \left(\text{–}0.178098\right) \times cos^2\left(\text{–}0.090268\right)}{3}$$

$$J_6 = \text{–}0.358150$$

$$\alpha = \frac{3}{4} \times e'^2 = \frac{3}{4} \times 0.006768$$

$$\alpha = 0.005076$$

$$\beta = \frac{5}{3} \times \alpha^2 \rightarrow \beta = \frac{5}{3} \times 0.005076128^2$$

$$\beta = 4.29 \times 10^{-5}$$

$$\gamma = \frac{35}{27} \times \alpha^3 = \frac{35}{27} \times 0.005076128^3$$

$$\gamma = 1.695 \times 10^{-7}$$

$$B_\Phi = 0.9996 \times c \times \left(\varphi - \alpha \times J_2 + \beta \times J_4 - \gamma \times J_6\right)$$

$$B_\Phi = \text{–}571683.243117$$

- **Cálculo final:** Con los datos obtenidos anteriormente ya estamos listos para hallar las coordenadas *UTM* con los cuales podemos trabajar para el diseño de nuestro controlador:

$$x = \xi \times \nu \times \left(1 + \frac{\zeta}{3}\right) + 5 \times 10^5 \rightarrow x = 540231.303715$$

$$y = \eta \times \nu \times (1 + \zeta) + B_{\Phi} + 1 \times 10^7 \rightarrow y = 9428305.2653$$

Hay que tener en cuenta para el eje $y$, si se trabaja al norte o al sur del Ecuador (latitud), si este se encuentra al sur de debe sumar el valor de $10^7$, como el ejemplo está al sur del Ecuador, se realiza está suma.

### 1.2.2. Planificación de trayectorias

El control de un sistema robótico se divide en dos áreas principales:

- La planificación del movimiento (o trayectoria).
- Control de movimiento.

Las cuales en conjunto permiten que un sistema robótico siga una trayectoria establecida como la trayectoria circular seguida por un brazo robotico como se muestra en la figura 1.6, para ello en esta parte nos enfocaremos en el primer punto, mencionando algunas posibilidades y consideraciones que se deben tener en cuenta para la planificación de trayectorias.

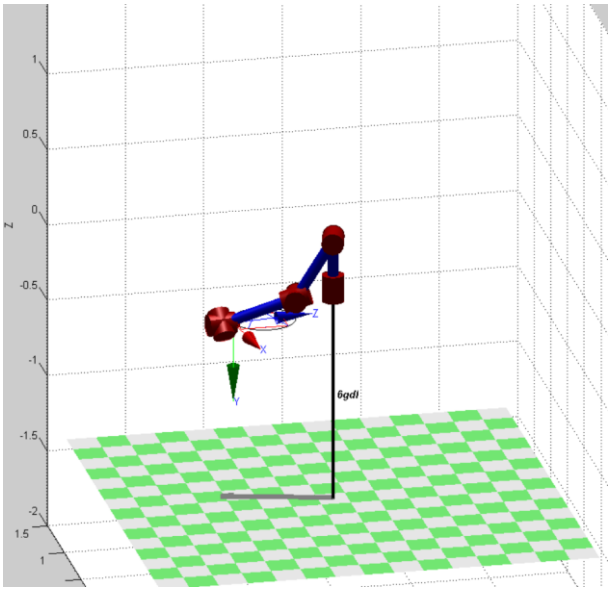



Figura 1.6: Brazo robótico de 6 grados de libertad siguiendo una trayectoria.
Fuente: Elaboración propia.

La planificación de trayectoria involucra varios puntos de las cuales conoceremos varios conceptos y enfoques que se utilizarán para el desarrollo de la planificación de trayectorias.

Dado cualquier sistema robótico es importante conocer donde se encuentra cada elemento del mismo en un espacio de trabajo con el objetivo de establecer un movimiento libre de colisiones. Esto significa planificar las trayectorias de tal manera que se eviten posibles obstáculos, por lo tanto se denominará este problema como: Planificación de trayectorias con restricciones por obstáculos de camino. Al mencionar un sistema robótico como un manipulador el cual involucra el control de la orientación y trayectoria espacial seguida por su efector final, este problema se denominará como: Planificación de trayectorias ligadas a una trayectoria [11], [12].

La forma más sencilla de resolver los dos problemas mencionados es no involucrar los efectos inducidos por las fuerzas que intervienen en el movimiento, hacer esto puede traer errores en el seguimiento de las trayectorias deseadas. Para contrarrestar estos efectos es necesario tener en cuenta la dinámica del sistema robótico en el cálculo de las trayectorias.

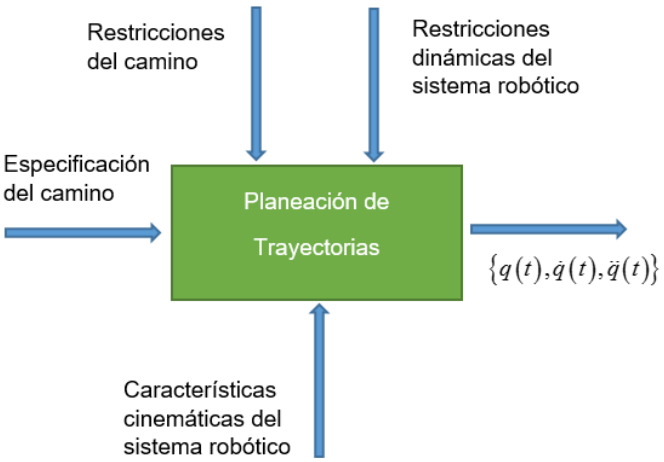



Figura 1.7: Representación sistemática del problema de planeación de trayectorias [13].

La figura 1.7, muestra en términos generales, dados los parámetros cinemáticos y dinámicos de un manipulador y la especificación de la trayectoria a seguir, se deben hallar los valores de posición, velocidad y aceleración de cada articulación en el tiempo para interpolar adecuadamente la trayectoria deseada, entendiendo como interpolar al hecho de unir varios puntos, ya sea que pertenezcan al espacio cartesiano, con una función en el dominio del tiempo.

En general se puede decir que la planificación de trayectorias se puede realizar tanto en el espacio de las variables de articulación como en el espacio cartesiano de referencia. Al planificar la trayectoria en términos de las variables directamente controladas (articulaciones), se logra reducir el tiempo de cálculo necesitado y la complejidad que involucra. Sin embargo, es difícil determinar la posición de los diferentes elementos críticos del manipulador durante el movimiento, para garantizar recorridos libres de obstáculos [13]. Por otro lado, calcular las trayectorias en coordenadas cartesianas tiene la ventaja de ser conceptualmente más directo y más preciso, pero con la desventaja de incrementar la complejidad en los cálculos implicados. Esto se debe principalmente al hecho de tener que convertir al espacio de articulación los puntos especificados por el usuario debido a que la parte de los algoritmos de control se basan en coordenadas de articulación ya que no existen sensores que midan la posición y orientación del efector final[9] directa y confiable.

[9] El efector final es el dispositivo en el extremo de un brazo robótico, diseñado para interactuar con su entorno.

#### 1.2.2.1. Tipos de trayectorias

Existen infinitas posibilidades para interpolar el movimiento entre dos puntos diferentes en el espacio. Sin embargo, en el movimiento de un robot desde un punto inicial a un punto final, existen algunas que por su sencillez de implementación y compatibilidad con las características cinemáticas de un manipulador mecánico tienden a ser más utilizadas a la hora de implementarlos [13].

**Trayectorias Articulares**

Las trayectorias en el espacio de articulación son uno de los campos más desarrollados en la planificación y especificación de tareas, puesto que las variables directamente a controlar en el movimiento de un manipulador son la posición, la velocidad angular y el torque en cada una de las articulaciones.

**Trayectorias Punto a Punto**

Para este tipo de trayectoria cada articulación evoluciona desde su posición inicial a una final sin realizar consideración alguna sobre el estado o evolución de las demás articulaciones. Normalmente, cada actuador trata de llevar su respectiva articulación al punto de destino en el menor tiempo posible, para la cual se tiene dos casos:

- **Movimiento de eje a eje:** solo se mueve un eje a la vez. Comenzará a moverse la primera articulación y una vez que esta haya alcanzado su punto final, lo hará el siguiente y así sucesivamente. La ventaja es que se consume menos potencia instantánea por parte de los actuadores debido a que es secuencial pero la desventaja es que toma mayor tiempo ejecutarlo.
- **Movimiento simultaneo de ejes:** Los actuadores comienzan simultáneamente a mover las articulaciones del robot a una determinada velocidad. Debido a que las distancias a recorrer y las velocidades serán diferentes, cada articulación terminará su movimiento en un instante diferente. El movimiento no terminará hasta que se alcance por completo el punto final, lo que sucede en el momento cuando el ultimo actuador se detenga por completo, así el tiempo total invertido en el movimiento del manipulador coincidirá con el tiempo invertido por el eje que más tiempo emplee en realizar su movimiento particular, por lo cual, generalmente el resto de actuadores fuerzan su desempeño generando elevadas velocidades y aceleraciones, viéndose obligados a esperar a la articulación más lenta.

Debido a las características de estas trayectorias, la trayectoria punto a punto solo está implementadas en sistemas robóticos muy simples o con sistemas de control muy limitados.

**Trayectorias Coordinadas o Isócronas**

Para evitar que algunos actuadores trabajen forzando sus velocidades y aceleraciones, teniendo que esperar a que finalice la articulación más lenta, puede realizarse un cálculo previo. Si se conoce previamente cual es la articulación más lenta, entonces se puede adecuar el movimiento del resto de las articulaciones para que todas inviertan el mismo tiempo en sus movimientos, con lo cual se conseguirá que todas las articulaciones se detengan simultáneamente. De esta manera, todas las articulaciones se coordinarán, comenzando y

terminando su movimiento al mismo tiempo, todos adaptándose a la más lenta. Gracias a esto se logra que el tiempo total invertido en el movimiento sea el menor posible y no se exigirán aceleraciones y velocidades elevadas a los actuadores de manera innecesaria.

**Trayectorias Cartesianas**

La trayectoria que un sistema robótico tiene que realizar es planeada por el usuario que por su familiarización con el entorno expresan las especificaciones de la tarea en el espacio cartesiano, así pues, las trayectorias cartesianas son aquellas cuyas restricciones y características están completamente descritas en este sistema de coordenadas y su generación obedece a procedimientos matemáticos que muestran secuencia de puntos que forman curvas en el espacio y cuya evolución en el tiempo se llama trayectoria [11].

**Trayectorias Analíticas**

Son aquellas que se obtienen a partir de una función que define la curva que debe seguir el efector final del manipulador. Las curvas más frecuentes para este tipo de trayectorias son la recta, el arco de circunferencia y la circunferencia [11]. La evolución de la trayectoria puede ser controlada sin cambiar la forma de la curva, es decir que su geometría no depende necesariamente de su comportamiento en el tiempo.

**Trayectorias Interpoladas**

Es común que al especificar las tareas para un manipulador en el espacio cartesiano solo sea de interés para el usuario la orientación y velocidad a la que pasa por los puntos definidos, denominados *knot points*. Para generar un camino se puede transformar estos puntos y sus parámetros al espacio articular y realizar la interpolación, sin embargo esto no le da una idea suficiente sobre la geometría de la trayectoria ni de las variables de velocidad y aceleración durante toda la evolución del movimiento. Para generar una trayectoria que cumpla con los requisitos especificados en cada *knot point* se pueden utilizar métodos de interpolación como las basadas en polinomios, como los *splines*[10], que describen la evolución de cada una de las coordenadas cartesianas en función del tiempo [11]. La ventaja en utilizar polinomios es que se pueden generar trayectorias suaves y continuas, que es importante para la dinámica del sistema. Pero este tipo de curvas la geometría del camino está fuertemente ligado a los parámetros cinemáticos de los *knot points*.

### 1.2.2.2. Generación de trayectorias

Describiremos el tipo de movimiento el cual todas sus condiciones y parámetros son especificados en el espacio cartesiano, la posición y orientación del efector final del manipulador que sigue un camino cartesiano con ciertas características cinemáticas la evolución del movimiento en el tiempo. Es importante tener en cuenta que en coordenadas cartesianas o cualquier clase de coordenadas curvilíneas ortogonales la evolución de la orientación y la traslación sobre un camino están desligadas, y se soluciona cada una de manera independiente, mientras que en coordenadas articulares una configuración define la posición y orientación del efector final, ya que no son coordenadas ortogonales [13].

[10] Es una curva diferenciable definida en porciones mediante polinomios.

La desventaja de utilizar trayectorias cartesianas es que involucra la evaluación continua del punto de referencia del efector final del manipulador la cual implica la solución cinemática inversa en cada punto de trayectoria interpolada para transformar a coordenadas de articulación.

Existen varios métodos numéricos y analíticos para generar trayectorias cartesianas que por su costo computacional no son muy utilizados para hacer aplicaciones de control en tiempo real. Sin embargo hoy en día dado al gran avance en el campo de procesadores y velocidades de procesamiento de datos, permite que estos métodos puedan ser implementados incluso ya no representarían impedimento alguno.

**Generación de Trayectorias con Funciones Analíticas**

La curva menos larga entre dos puntos es la línea recta, es decir para ir de un punto a otro la trayectoria que describe la línea recta necesitara menos energía además que es la trayectoria conceptualmente más simple y directa, esto la convierte en una trayectoria muy versátil para la generación de trayectorias más complejas [13].

Para generar una recta en el espacio cartesiano se puede utilizar la ecuación paramétrica de la recta, es decir:

$$f\left(x,y,z\right)=\lambda\left(t\right)\left(P_1-P_0\right)+P_0 \tag{1.29}$$

Donde:

$P_1$: Punto final.
$P_0$: Punto inicial.
$\lambda\left(t\right)$: Función que define cuanto y como varían los puntos de la recta en función del tiempo, es decir controla la velocidad del movimiento.

En primer lugar se analizaran las formas básicas de movimiento rectilíneo:

- Velocidad constante.
- Aceleración constante.

La forma más sencilla es moverse con velocidad constante, entonces la derivada de la posición será constante, por ello derivamos la ecuación 1.29, e igualamos a una constante.

$$V_0=\left(P_1-P_0\right)\frac{d\lambda}{dt} \tag{1.30}$$

Luego de resolver la ecuación diferencial resultante, podemos definir completamente la ecuación de la recta a velocidad constante:

$$\lambda\left(t\right)=\frac{V}{\left|P_1-P_0\right|}t=\frac{t}{T}=\frac{t_k-t_i}{t_f-t_i}\rightarrow t\in\left[0,T\right] \tag{1.31}$$

$$f_k\left(x,y,z\right)=\frac{t_k-t_i}{t_f-t_i}\left(P_1-P_0\right)+P_0 \tag{1.32}$$

Donde:

$P_0 \le f_k \le P_1$
$t_k = k\Delta t$
$\Delta t = \frac{T}{n}$

Si se re-escribe la ecuación resultante en forma paramétrica se obtiene la evolución del movimiento de cada una de las coordenadas en función del tiempo.

De la misma forma obtenemos la función para una aceleración constante sobre la recta igualando la derivada de la ecuación 1.30, a una variación lineal de la velocidad, luego aplicamos el mismo método de resolución en las ecuaciones anteriores:

$$at+V_0=\left(P_1-P_0\right)\frac{d\lambda}{dt} \tag{1.33}$$

$$\lambda\left(t\right)=\frac{1}{\left|P_1-P_0\right|}\left(\frac{1}{2}at^2+V_0t\right) \tag{1.34}$$

$$p\left(t\right)=\frac{1}{\left|P_1-P_0\right|}\left(\frac{1}{2}at^2+V_0t\right)\left(P_1-P_0\right)+P_0 \tag{1.35}$$

$$a=\frac{2\left(\left|P_1-P_0\right|-V_0t\right)}{T^2} \tag{1.36}$$

Con estas ecuaciones podemos encontrar la función general de la ecuación paramétrica de la recta, la cual resulta la expresión que genera puntos entre $P_0$ y $P_1$ de manera acelerada durante un periodo de tiempo $T$ :

$$f_k\left(x,y,z\right)=\frac{1}{\left|P_1-P_0\right|}\left[\left(\frac{\left|P_1-P_0\right|-V_0T}{T^2}\right)t_k^2-V_0t_k\right]\left(P_1-P_0\right)+P_0 \tag{1.37}$$

Donde:
$t_k \in [0,T]$

Es posible expresar esta misma función en términos de la velocidad inicial y final del movimiento y hallar el tiempo necesario para que el movimiento se realice entre los puntos $P_0$ y $P_1$, es decir:

$$a=\frac{V_f-V_0}{T} \tag{1.38}$$

$$T = \frac{2|P_1 - P_0|}{V_f + V_0} \tag{1.39}$$

$$f_k(x, y, z) = \frac{1}{|P_1 - P_0|}\left[\left(\frac{V_f^2 - V_0^2}{4|P_1 - P_0|}\right)t^2 - V_0 t\right](P_1 - P_0) + P_0 \tag{1.40}$$

Donde:

$$t_k \in \left[0, \frac{2|P_1 - P_0|}{V_f + V_0}\right]$$

Dado que la aceleración $a$ es solución única del sistema, cuando la aceleración es menor que cero se está trabajando con una parábola abierta hacia abajo y es posible que la posición y el tiempo suceda en la sección de la parábola que ha cambiado de signo en su pendiente respecto a la inicial, esto conllevaría a sobrepasar el valor de la posición esperado para poder cumplir los parámetros de tiempo y espacio finales. Para que esto no suceda se deben acotar el tiempo invertido o la velocidad inicial, para asegurar que no exista cambio de signo en la velocidad:

$$V_0 \leq \frac{2|P_1 - P_0|}{T} \tag{1.41}$$

Cuando los parámetros son la velocidad inicial y la velocidad final lo único que tenemos que hacer es asignar valores mayores o iguales a cero (positivos), ya que la dirección de la velocidad le da el vector unitario formado en dirección de $P_0$ y $P_1$.

Las funciones resultantes servirán para el cálculo de los puntos sobre la trayectoria deseada, la cual será útil para la implementación de nuestro algoritmo de seguimiento de trayectoria. Para evaluar la función de la recta necesitaremos generar valores de tiempo igualmente espaciados de acuerdo al número de puntos que se deseen calcular para construirla.

También es importante conocer la velocidad en cada punto generado, para esto existe dos formas para hacerlo, la primera seria derivando las funciones de posición generadas resultando las funciones de velocidad para después evaluarla de la misma forma que se hizo para obtener las posiciones y la segunda forma seria implementar un algoritmo mediante métodos numéricos para que a partir de los puntos de posición obtenidos hallar la derivada y así obtener la velocidad para cada tiempo.

Para el presente proyecto se optó por la segunda forma (utilizando métodos numéricos), debido a que si bien puede generar pequeños errores de cálculo dependiendo de la cantidad de puntos considerados, su enorme ventaja es que calcula la derivada de un conjunto de puntos sin importar que función de posición se esté utilizando (a diferencia de la primera forma) y también es útil por ejemplo si queremos saber la aceleración ya que solo utilizamos la derivada generada por métodos numéricos una segunda vez.

### 1.2.3. Perfil de velocidad trapezoidal

A la hora de trabajar con un sistema robótico es recomendable utilizar curvas de velocidades compuestas de los movimientos básicos tales como velocidad o aceleración constante, esto se debe a que le resulta imposible a un sistema robótico estar en un estado reposo y pasar a un estado de velocidad constante, o la de cambiar su velocidad de forma instantánea para cambiar de trayectoria, ya que esto conduce a realizar aceleraciones infinitas para poder lograrlo.

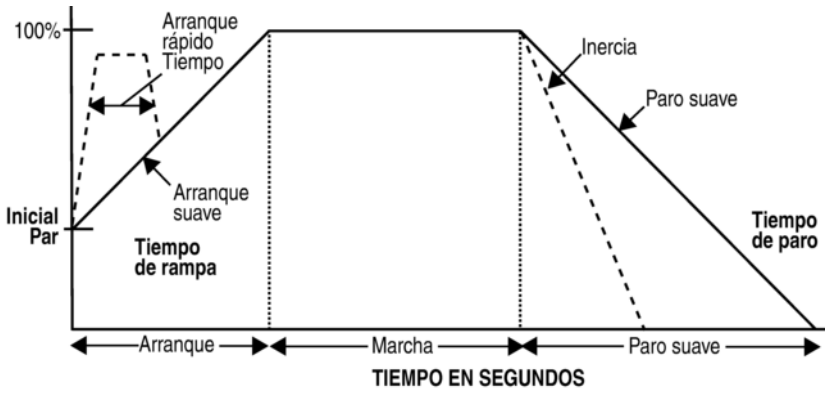



Figura 1.8: Ejemplo de Perfil de Velocidad Trapezoidal.
Fuente: https://www.emaze.com/@AZFRIRLT/VARIADORES

Las curvas de velocidad, las cuales pueden componerse fácilmente de movimientos a velocidad constante y aceleración constante son de forma trapezoidal, como en la figura 1.8, estas curvas no son más que funciones a trozos o tramos que tienen algunos detalles en su construcción, la cual se explicará en esta sección.

#### 1.2.3.1. Perfil de velocidad trapezoidal acelerada y velocidad constante

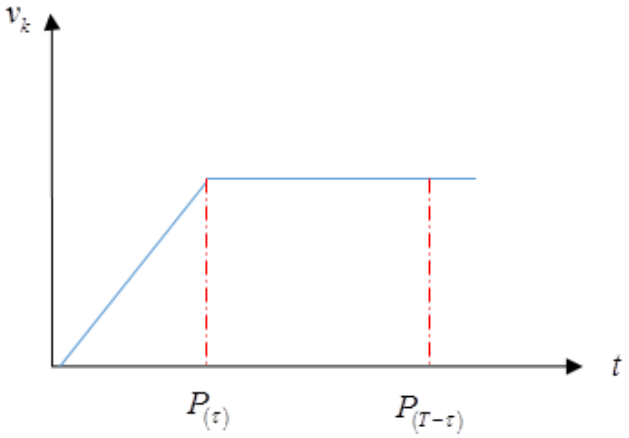



Figura 1.9: Curva de velocidad trapezoidal acelerada + velocidad constante.
Fuente: Elaboración propia.

Cuando se trata de líneas rectas se puede representar fácilmente a través de funciones las cuales describen las coordenadas en el tiempo. Como se muestra en la figura 1.9, cuenta con 2 segmentos, el primer segmento es de aceleración constante y el segundo segmento de velocidad constante. Entonces se tiene la siguiente ecuación:

$$\left|P_1 - P_0\right| = \frac{1}{2}at^2 + v_k\left(T - \tau\right) \tag{1.42}$$

Donde:

$a = \frac{v_k}{\tau}$

$T$ : Tiempo total.

$\tau$ : Tiempo donde termina la aceleración.

$P_0$ : Punto inicial.

$P_1$ : Punto final.

Podemos decir que el tiempo que dura el segmento de aceleración se puede tomar como un porcentaje del tiempo total invertido en la trayectoria, por esto, la velocidad se puede representar como:

$$v_k = \frac{\left|P_1 - P_0\right|}{T\left(1 - \frac{pt}{2}\right)} \tag{1.43}$$

Donde:

$pt$ : Porcentaje del tiempo invertido en la trayectoria que dura la parte acelerada del movimiento.

El trabajar porcentualmente el tiempo de los tramos permite variar el tiempo total de la trayectoria sin tener la necesidad de volver a calcular el valor del segmento correspondiente a $\tau$ . Tener en cuenta que $pt$ debe ser mayor que 0 y menor que 1 (0-100 %) para que la trayectoria siempre tenga los dos tramos.

El siguiente paso es hallar el punto en el espacio en el que se cambia de velocidad subiendo (acelerado) a velocidad constante (aceleración=0), esto se puede realizar suponiendo que todo el movimiento es acelerado, es decir:

$$p(t) = \frac{1}{\left|P_1 - P_0\right|}\left(\frac{1}{2}at^2 + v_0 t\right)\left(P_1 - P_0\right) + P_0 \tag{1.44}$$

Donde:

$t = \tau$

$a = \frac{v_k}{\tau}$

$$p(t) = \frac{1}{\left|P_1 - P_0\right|}\left(\frac{1}{2}v_k\tau\right)\left(P_1 - P_0\right) + P_0 \tag{1.45}$$

El último paso es generar la nueva función para el primer y segundo segmento basado en las variables de cada segmento, en la tabla 1.1 podemos observar los valores de las variables en cada tramo de aceleración y velocidad constante.

Tabla 1.1: Variables para segmento.

| Tipo de movimiento | Posición inicial | Posición final | Valor variable |
|---|---|---|---|
| Acelerado | $P_0$ | $P(\tau)$ | $v_0 = 0$<br>$v_f = v_k$ |
| Velocidad constante | $P(\tau)$ | $P_1$ | $v_c = v_k$ |

Fuente: Elaboración propia.

Con esto reemplazamos las variables del tramo acelerado y las variables del tramo de velocidad constante, lo que resulta:

$$f(t) = \begin{cases} \dfrac{1}{\left|P_{(\tau)} - P_1\right|}\left(\dfrac{1}{2}\dfrac{v_k}{\tau}t^2\right)\left(P_{(\tau)} - P_0\right) + P_0 \rightarrow 0 \le t \le \tau \\ \dfrac{v_k}{\left|P_1 - P_{(\tau)}\right|}(t-\tau)\left(P_0 - P_{(\tau)}\right) + P_{(\tau)} \rightarrow \tau < t \le T \end{cases} \tag{1.46}$$

### 1.2.3.2. Perfil de velocidad trapezoidal velocidad constante y desacelerada.

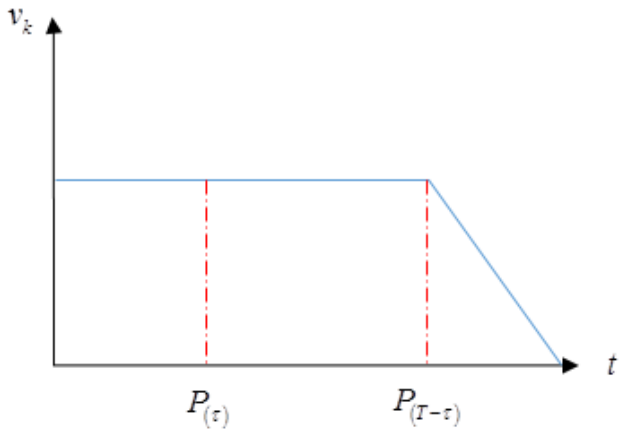



Figura 1.10: Curva de velocidad trapezoidal velocidad constante + desaceleración.
Fuente: Elaboración propia.

Para generar la función a trozos de una recta con la curva de la figura 1.10, se aplica un procedimiento similar al anterior partiendo de la siguiente ecuación:

$$\left|P_1 - P_0\right| = v_k\left(T - \tau\right) - \frac{1}{2}a\tau^2 + v_k\tau \tag{1.47}$$

Donde:

$$a = \frac{v_k}{\tau}$$

$$v_k = \frac{\left|P_1 - P_0\right|}{T\left(1 - \dfrac{pt}{2}\right)}$$

Con todas las variables conocidas se puede generar la función a trozos utilizando primero un movimiento a velocidad constante y luego uno desacelerado. Considerar el intervalo de tiempo, es necesario redefinir la variable $T$ :

$$p(\tau)=\frac{(T-\tau)}{T}(P_1-P_0)+P_0 \tag{1.48}$$

$$f(t)=\begin{cases}\dfrac{v_k}{\left|P_{(\tau)}-P_0\right|}t\left(P_{(\tau)}-P_0\right)+P_0 \rightarrow 0\le t\le T-\tau \\ \dfrac{1}{\left|P_1-P_{(\tau)}\right|}\left(-\dfrac{v_k}{2\tau}\left(t-T+\tau\right)^2+v_k\left(t-T+\tau\right)\right)\left(P_1-P_{(\tau)}\right)+P_{(\tau)} \rightarrow T-\tau<t\le T\end{cases} \tag{1.49}$$

### 1.2.3.3. Perfil de velocidad trapezoidal acelerada, velocidad constante y desacelerada

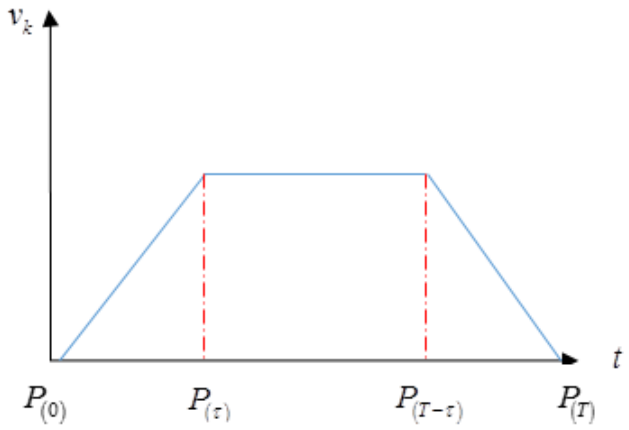



Figura 1.11: Curva de velocidad trapezoidal acelerada + velocidad constante+ desaceleración.
Fuente: Elaboración propia.

Ahora para representar una recta con curva trapezoidal como la figura 1.11, debemos encontrar los dos puntos donde hay cambio de movimiento sobre la recta. El primero ocurre de una aceleración constante ( $a>0$ ) a una velocidad constante ( $a=0$ ) y el otro punto ocurre cuando empieza a desacelerar ( $a<0$ ). Es importante mencionar que los dos puntos ocurren con simetría en el tiempo total (trapecio isósceles) que dura el movimiento, acelera durante $\tau$ segundos y desacelera $\tau$ segundos antes de terminar la trayectoria, para llegar al estado de reposo y continuar con el siguiente punto establecido por el usuario.

Las ecuaciones son las mismas utilizadas en los anteriores:

$$\left|P_1-P_0\right|=\frac{1}{2}a\tau^2+v_k\left(T-2\tau\right)-\frac{1}{2}a\tau^2+v_k\tau \tag{1.50}$$

Donde:

$$a=\frac{v_k}{\tau}$$

$$v_k = \frac{|P_1 - P_0|}{T - \tau}$$

De la expresión obtenida para calcular la velocidad se puede deducir que ese tiempo no puede ser igual a $T$ , porque daría división por cero, recordemos que la trayectoria está dividida en tres segmentos, de esta manera $\tau$ no puede superar el valor de $\frac{T}{2}$ pues solo quedarían dos segmentos, el final y el inicial.

Entonces lo siguiente es encontrar los puntos donde la recta cambia su movimiento, donde el tramo acelerado en $\tau$ :

$$p(\tau) = \frac{1}{|P_1 - P_0|}\left(\frac{v_k}{2}\tau\right)(P_1 - P_0) + P_0 \tag{1.51}$$

$$p(T - \tau) = \frac{T - 2\tau}{T - \tau}\left(P_1 - P_{(\tau)}\right) + P_{(\tau)} \tag{1.52}$$

Finalmente teniendo los puntos, las velocidades iniciales y finales de cada tramo se puede generar la función a trozos utilizando las ecuaciones que definen el movimiento de cada segmento como se hizo anteriormente con los otros casos:

$$f(t) = \begin{cases} \frac{1}{|P_{(\tau)} - P_1|}\left(\frac{1}{2}\frac{v_k}{\tau}t^2\right)(P_{(\tau)} - P_0) + P_0 \rightarrow 0 \le t \le \tau \\ \frac{vk}{|P_{(T-\tau)} - P_{(\tau)}|}(t - \tau)\left(P_{(T-\tau)} - P_{(\tau)}\right) + P_{(\tau)} \rightarrow \tau < t \le T - \tau \\ \frac{1}{|P_1 - P_{(T-\tau)}|}\left(-\frac{v_k}{2\tau}(t - T + \tau)^2 + v_k(t - T + \tau)\right)\left(P_1 - P_{(T-\tau)}\right) + P_{(T-\tau)} \rightarrow T - \tau < t \le T \end{cases} \tag{1.53}$$

La implementación de las interpolaciones tiene una serie de consideraciones que surgen por la discretización de las funciones en conjuntos de puntos, el problema más crítico que surge es el cálculo de la cantidad de los puntos que corresponde hallar en cada una de las secciones que define la función a trozos.

# Capítulo 2

# Software robótico

## 2.1. *Frameworks* para robots

Este capítulo se enfoca en el núcleo de esta tesis, el cual es *ROS*. Se detallará de sus antecedentes y orígenes, como también de su arquitectura, funcionamiento, entorno de trabajo y otros conceptos nuevos quizás para muchos para poder comprender este *framework* robótico y su posterior utilización, en el trabajo de Ceriani [14], lo describe como *middleware*[11], pero según *ROS*, se describe como un *framework*, esto se explicará a continuación.

### 2.1.1. *Framework*

Los que trabajamos con software robótico nos hemos encontrado con el término *framework* (su traducción literal sería "marco de trabajo"), pero esa traducción no es en sí correcta, ya que involucra otros elementos, a pesar de que cualquiera con experiencia programando entenderá el concepto de esta de manera casi intuitivamente y muy posiblemente esté utilizando su propio *framework* sin ser consciente de ello.

Todo programador sabe lo importante que es la normalización de datos en cualquier aplicación. Los usuarios o clientes pueden manejar su propia información en papel por ejemplo, tenerla duplicada, con incoherencias u omisiones, etc. en fin, pero él se entiende en su propio desorden. En cambio para una aplicación informática se necesita que esa

[11] Software que asiste a una aplicación para interactuar o comunicarse con otras aplicaciones, o paquetes de programas, redes, hardware y/o sistemas operativos.

información este estructurada de un modo conocido o estándar de modo que se pueda manejarla, almacenarla, recuperarla, etc. Es por ello que el concepto de *framework* hace su aparición para poder estructurar y normalizar la información.

#### 2.1.1.1. Definición

Un *framework* es un esquema para el desarrollo y/o la implementación de una aplicación, está definición es muy genérica ya que existe otros *frameworks* que pueden llegar al detalle de definir nombres de ficheros, su estructura, las convenciones de programación. etc.

Los *frameworks* no necesariamente están ligados a un lenguaje de programación concreto, aunque la mayoría de veces sucede, pero cuando más detallado es el *framework*, más necesidad tendrá de ceñirse a un lenguaje en específico. También es posible que el *framework* defina una estructura para una aplicación completa o bien sólo se centre en un aspecto de ella [15].

#### 2.1.1.2. Ventajas

Como se mencionó anteriormente sobre la estandarización, se tienen otras como:

- El programador no necesita plantearse una estructura global, sino que el *framework* le proporciona un "marco de trabajo" el cual se tiene que "llenar".
- Facilita la colaboración. Cualquiera que haya tenido problemas con el código fuente de otro programador o incluso con su propio código, sabrá lo difícil que es entenderlo y modificarlo, por ello resulta un ahorro de tiempo y esfuerzo para aquellos desarrollos colaborativos la estandarización.
- Es mucho más sencillo encontrar herramientas (librerías, etc.) adapta-das para un *framework* determinado para facilitar el desarrollo.

#### 2.1.1.3. Necesidad de utilizar un *framework*

Cualquier desarrollador de software puede crear toda una aplicación sin seguir ningún *framework* conocido, puede que la aplicación sea tan pequeña que no lo considere necesario, no encuentre uno de acuerdo a sus necesidades o tal vez no quiera gastar tiempo en ello.

No obstante, a medida que la aplicación crece, un desarrollador competente procurará seguir unas determinadas pautas que le faciliten su trabajo de desarrollo y mantenimiento: separación de presentación y lógica, una sintaxis coherente, etc. La pregunta obvia sería, ¿qué estoy haciendo?, la respuesta sería que está construyendo su propio *framework*.

Resultaría más útil utilizar uno ya construido y aprovechar el trabajo de otros desarrolladores, la curva de aprendizaje de utilizar un *framework* se compensa probablemente en cuanto el trabajo de desarrollo crezca mínimamente.

#### 2.1.1.4. Criterios para elegir un *framework*

En Internet podemos encontrar varios *framework* para diferentes plataformas y lenguajes, tal vez el problema como existen varios, nos resulta complicado hacer la mejor elección, es por ello que se deben tener en consideración lo siguiente:

- El tipo de aplicación a desarrollar.
- El lenguaje de programación y otras tecnologías concretas, base de datos, sistema operativo y demás.

El empleo de un *framework* en el desarrollo de una aplicación implica un cierto coste inicial de aprendizaje, aunque a largo plazo es seguro que facilite tanto el desarrollo como el mantenimiento.

### 2.1.2. Softwares robóticos anteriores a *ROS*

Existen varios proyectos entorno a intentar lanzar un *framework* para sistemas robóticos, comparten casi los mismos conceptos y objetivos básicos, pero sólo están como propuestas u otros no han seguido desarrollándose, en esta parte se mencionará aquellos que han sido utilizados por la comunidad robótica y son mantenidos hasta la actualidad.

#### 2.1.2.1. ORCA

*Open Robot Control Architecture*, es un *framework* de código abierto para el desarrollo de sistemas robóticos construido a base de componentes, semejante al software *LabVIEW*[12] como se observa en la figura 2.1 mediante el uso de bloques llamados componentes, proporciona los medios para definir y desarrollar los bloques de construcción que pueden ser colocados juntos para formar sistemas robóticos complejos, desde dispositivos individuales hasta redes de sensores distribuidos.

Los objetivos de ORCA son:

- Reutilización del software a través de la definición de un conjunto de interfaces de uso común.
- Simplificar la reutilización del software, brindando librerías con una API[13] conveniente de alto nivel.
- La reutilización de software se da a través del mantenimiento de un depósito de componentes.

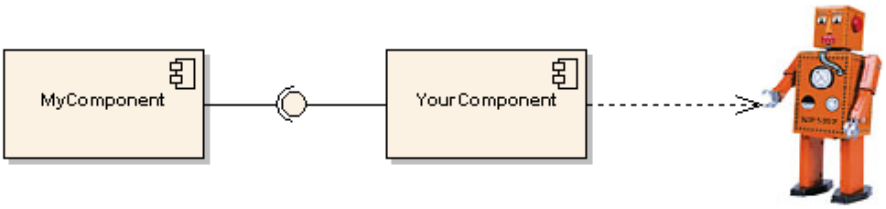



Figura 2.1: ORCA está basado en componentes [16].

ORCA se define a sí misma como un sistema basado en componentes sin restricciones, lo que significa que quiere proporcionar una infraestructura para construir software robótico, para no obligar al usuario a re-escribir todo su código desde cero. De esta manera, ORCA adopta un enfoque de ingeniería de software basado en componentes sin aplicar ninguna

[12] Acrónimo de *Laboratory Virtual Instrument Engineering Workbench*, plataforma y entorno de desarrollo para diseñar sistemas con un lenguaje de programación visual gráfico.
[13] Acrónimo de *Application Programming Interface*, es un conjunto de funciones y procedimientos que cumplen una o varias funciones con el propósito de ser utilizadas por otro software.

restricción arquitectónica adicional y utiliza una librería de código abierto comercial para la comunicación y la definición de la interfaz [16].

También proporciona algunas librerías para aplicaciones comunes y herramientas para simplificar el desarrollo de componentes pero los hace estrictamente opcionales para mantener pleno acceso al subyacente motor de comunicación y servicios. ORCA tiene 2 servicios principales las cuales son:

- **Servicio *IceGrid*:** Ofrece un servicio de nombres, mapeo de nombres de interfaces lógicas a direcciones físicas, siendo la única forma para que los componentes se encuentren el uno al otro.
- **Servicio *IceStorm*:** Se utiliza para desacoplar los publicadores de mensajes de los suscriptores. Básicamente existe un servicio *IceStorm* por *host*.

#### 2.1.2.2. OROCOS

*Open Robot Control Software*, es otro *framework* robótico, el cual se describe [17] como:

- Software libre, modular y flexible. Esto permite que un desarrollador pueda construir su sistema robótico desde cero, mientras otros desarrolladores pueden contribuir a los módulos en los que están interesados sin necesidad de revisar el código de todo el sistema.
- Bien estructurado, viéndolo desde la documentación técnica y la ingeniería de software.
- Independiente de los fabricantes de robots comerciales, para convertirlo en un estándar abierto.
- Disponible para cualquier sistema operativo y dispositivo robótico.
- Con componentes que realizan cinemática, dinámica, planificación, control, etc. de esta forma se pueden agregarlos o eliminarlos de forma dinámica de una red.

Básicamente OROCOS se compone de tres bibliotecas de C++ [17] como se observa en la figura 2.2 y una herramienta para el manejo en tiempo real de las aplicaciones:

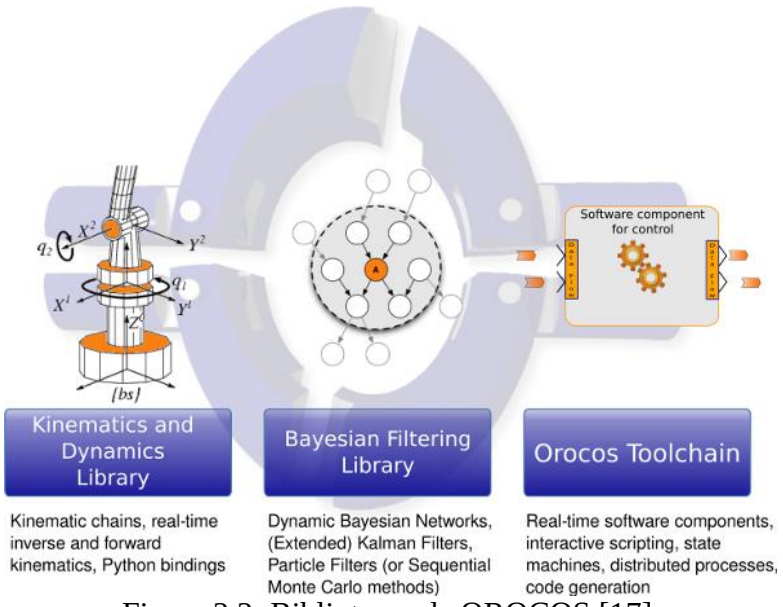



Figura 2.2: Bibliotecas de OROCOS [17].

- ***Real-Time Toolkit (RTT):*** No es una aplicación en sí misma ya que proporciona la infraestructura y las funcionalidades para crear aplicaciones robóticas en C++, lo importante es de que está en tiempo real, aplicaciones interactivas en línea y aplicaciones basadas en componentes.
- **OROCOS *Components Library (OCL):*** Proporciona algunos componentes de trabajo listos para su uso para interactuar con los dispositivos de hardware, manejar rutas de acceso, controlar diversos tipos de robots, también para grabar y visualizar los flujos de datos.
- ***Kinematics and Dynamics Library (KDL):*** Permite calcular cadenas cinemáticas en tiempo real. Desarrolla un marco independiente de aplicaciones para el modelado y el cálculo de las cadenas cinemáticas, como los robots, modelos humanos biomecánicos, etc.
- ***Bayesian Filtering Library (BFL):*** Proporciona un marco de trabajo de aplicaciones independientes para la inferencia en redes bayesianas dinámicas, es decir, algoritmos recursivos de la información y su estimación basados en la regla de Bayes.

#### 2.1.2.3. YARP

*Yet Another Robot Plataform*, es un conjunto de bibliotecas, protocolos y herramientas para mantener los módulos y dispositivos adecuadamente desacoplados. YARP es un intento de hacer que el software robótico sea más estable y de larga duración, sin comprometer la capacidad de cambiar constantemente los sensores, actuadores, procesadores y redes [18].


Figura 2.3: *iCub* desarrollado en YARP.
Fuente: http://www.talentour.org/open-yarp-workshop-july-15th-2015/

YARP está desarrollado por y para investigadores en robótica, específicamente en robótica humanoide, que se encuentra con una complicada pila de hardware para controlar con una igualmente complicada pila de software, la aplicación más conocida es iCub humanoide creado en YARP tal y como se muestra en la figura 2.3.

Los componentes de YARP son [18]:

- ***libYARP__OS:*** Hace de interfaz con el sistema operativo y permite un sencillo flujo de datos a lo largo de diferentes hilos en diferentes máquinas. YARP está diseñado para ser neutral en cuanto al sistema operativo y está casi íntegramente escrito en

C++. Siendo este el componente principal el cual deberá estar disponible antes de utilizar los demás componentes.

- ***libYARP__sig:*** Realiza tareas comunes de proceso de señal (vídeo y audio) de manera abierta fácilmente interconectada entre otras librerías usadas, como *OpenCV*[14] por ejemplo.
- ***libYARP__dev:*** Hace de interfaz con dispositivos comunes usados en robótica como cámaras digitales, drivers de motores y demás.

Para el funcionamiento en tiempo real, la sobrecarga de la red tiene que ser minimizada, por lo que YARP está diseñado para funcionar en una red aislada o detrás de un *firewall*[15]. Si se exponen máquinas que funcionan con YARP a internet, es de esperar que el robot algún día pueda ser controlado remotamente sin que se pueda evitar.

### 2.1.3. *ROS* una nueva alternativa

*Robotic Operating System* o simplemente *ROS* prácticamente será el estándar de los sistemas robóticos complejos en un futuro, en sus 8 años de creación, se han desarrollado innumerables aplicaciones, su éxito no sólo se debe a que es código abierto sino principalmente a nosotros (los desarrolladores) los cuales vienen dándole soporte a este magnífico *framework* y a través de los foros donde todos tratamos de ayudarnos. La comunidad ROS se concentra más en Europa y Estados Unidos como se muestra en la figura 2.4.


Figura 2.4: Comunidad de *ROS* a nivel mundial [5].

El dominio perfecto de un sistema como *ROS* no se logra en un par de meses o tutoriales, se necesita mucho tiempo y esfuerzo para ser un experto en él, debido a las actualizaciones, variaciones y mejoras en cada versión. Por tanto, el lector que quiera empezar está aventura en *ROS* en el mundo de la robótica es una ardua tarea, pero de gratificantes resultados.

---

[14] Acrónimo de *Open Source Computer Vision* ,es una librería de visión por computadora y procesamiento de imágenes de código abierto

[15] Programa informático que controla el acceso de una computadora a la red y de elementos de la red a la computadora, por motivos de seguridad.

## 2.2. *Robotic Operating System*

Las comunidades de desarrolladores se han convertido en parte fundamental para la investigación y desarrollo sobre diferentes temas. Prueba de esto, es que no es necesario ser un experto para poder entrar en campos que hasta hace poco tiempo atrás parecían inalcanzables o de difícil comprensión sobre áreas de la tecnología. En estos días es posible aprender e investigar sobre cualquier campo que uno desee, desde tutoriales, foros, blogs, wikis, etc., se puede profundizar en diferentes campos todo esto gracias a la aparición del Internet.

Dentro de este marco, en específico sobre la robótica, se cuenta con un *framework* de código libre como lo es *ROS* el cual es una herramienta que nos facilita el adentrarnos en el mundo de la robótica. *ROS*, no sólo es un "sistema operativo", es un ecosistema completo, que cuenta con código y herramientas que nos ayudan a dar vida a nuestros proyectos, pero fundamental es una activa comunidad en desarrollo siempre.

La filosofía de *ROS* se basa en "no reinventar la rueda", es decir, a la hora de desarrollar un proyecto y no partir desde cero, sino la de partir desde un proyecto previo la cual ha sido evaluado y mejorado, esto permite ahorrar tiempo y el desarrollador poder enfocarse en la aplicación la cual quiere abordar, siendo esta la principal ventaja el uso de *ROS* [5].

En el presente trabajo dado que *ROS* es desarrollado en el idioma inglés, se utilizará los nombres en este idioma de los diferentes elementos que constituyen *ROS*, para evitar confusión y poder consultar otras referencias en caso de ser necesario.

### 2.2.1. Definición de *ROS*

*Robot Operating System* o *ROS* como se le conoce coloquialmente, es un pseudo-sistema operativo (*framework*, para ser más específico) de código libre para el desarrollo de software para robots. *ROS* proporciona librerías y herramientas para ayudar a los investigadores y aficionados a la robótica a desarrollar aplicaciones para robots. También proporciona abstracción de hardware, controladores de dispositivos, visualizadores, interfaz de intercambio de mensajes, gestora de paquetes y demás. Entonces un sistema construido usando *ROS* consiste en un número de procesos, potencialmente en diferentes máquinas, conectados en tiempo real en una red punto a punto [19].

*ROS* se ha desarrollado principalmente para poder reutilizar controladores y código de otras aplicaciones ya desarrollados y ya probadas, esto se logra encapsulando el código en librerías independientes que no tienen dependencias en *ROS*.

La gran ventaja para el uso de *ROS* frente a otros sistemas, es que parte de un sistema colaborativo (comunidad de *ROS*) [20]. Debido a la gran cantidad de robots y software de inteligencia artificial que se han desarrollado y se están desarrollando en diferentes universidades, centros de investigación, empresas, etc., con esto se manifiesta la importancia de colaboración entre estas diferentes entidades. Para dar soporte a estos desarrollos, *ROS* proporciona un sistema de paquetes. Un paquete de *ROS* es un directorio el cual contiene un archivo `.xml` que describe el paquete y sus dependencias respectivas.

### 2.2.2. Ventajas de *ROS*

El objetivo de *ROS* no es ser el primer sistema operativo para robots, sino su principal objetivo es la de apoyar el código reutilizable en el desarrollo de robots. *ROS* es una estructura distribuida de procesos *nodes*, que proporciona la ventaja de que estos *nodes* puedan ser diseñados de forma individual y utilizarse de forma flexible en tiempo real. *ROS* también es compatible con el sistema de repositorios, permitiendo la colaboración a nivel mundial entre las instituciones desarrolladoras de robots, el cual permite utilizar código mejorado y probado por otros.

Las ventajas que podemos mencionar de *ROS* frente a otros sistemas operativos robóticos, son [21]:

**a. Sencillez:** *ROS* está diseñado para ser tan sencillo como se pueda, de modo que sea posible que el código escrito para *ROS* se pueda utilizar con otros marcos de software, como la integración de *ROS* con PLAYER y OROCOS.
**b. Modelo de bibliotecas:** Es el modelo de desarrollo preferido, el cual consiste en escribir una serie de programas con interfaces funcionales y claras que permitan una modificación rápida y eficaz.
**c. Independencia de lenguajes de programación:** *ROS* permite implementarse en diferentes lenguajes de programación modernos, como en Python, C++ y LISP, estando en etapa de prueba los lenguajes en Java y Lua.
**d. Modo test:** *ROS* además cuenta con un comando llamado `rostest` que hace que sea fácil acceder a un modo prueba de sistema.
**e. Escala:** El diseño de *ROS* se basa en una arquitectura modular, que es adecuada para sistemas grandes y ejecutar procesos de gran escala.

### 2.2.3. Configuración de *ROS*

Desde el año que fue lanzado *ROS*, alrededor del 2009, existen varias versiones tales como:

- *ROS Electric.*
- *ROS Fuerte.*
- *ROS Groovy.*
- *ROS Indigo.*
- *ROS Jade.*
- *ROS Kinetic.*

Claro que cada versión es adecuada para cada versión de Ubuntu, quizás esta sea una de las desventajas de *ROS*, ya que cada versión tiene algunas variaciones frente a sus predecesoras, por lo que se está tratando de uniformizar todas las herramientas utilizadas.

En las primeras versiones de *ROS* se contaba con el simulador en un espacio 3D, Gazebo, con las mejoras de las versiones de Gazebo y el potencial que tenía este simulador, se separó de *ROS* y se viene desarrollando a la par con *ROS*, existiendo varias versiones, claro que se puede integrar a otros sistemas.

*ROS* prácticamente se ha desarrollado totalmente en Ubuntu, existiendo una vasta documentación, tutoriales, foros, etc. para este sistema operativo. Aunque también existen

versiones en desarrollo para otros sistemas operativos (Windows por ejemplo), la cual aún se intenta darle soporte así como para otras distribuciones de Linux.

En el presente trabajo se utilizó el sistema operativo Linux con su distribución Ubuntu 14.04 *Trusty*, con una versión de *ROS Indigo*, ya que existe tutoriales las cuales usan las mismas herramientas y comandos que las últimas versiones de *ROS Jade* y *ROS Kinetic*, además trabaja con normalidad el simulador Gazebo sin problemas en su versión 2.0, la cual se instala por defecto al instalar *ROS* el paquete completo.

Dado a la experiencia adquirida con *ROS* hasta ahora, es recomendable tener una máquina real con Ubuntu (no máquina virtual), ya que para robots mucho más complejos el requerimiento de recursos (memoria RAM, tarjeta gráfica, etc.) puede llegar a ser alto, más que todo para el simulador Gazebo el cual consume muchos recursos.

### 2.2.4. Arquitectura de *ROS*

La arquitectura de *ROS* ha sido diseñada y dividida en tres secciones [22]:

- **El nivel de sistema de archivos:** Un grupo de conceptos es usado para explicar cómo *ROS* está formado internamente, la estructura de la carpeta, y los mínimos archivos que se necesita para poder trabajar.
- **El nivel gráfico de cálculo:** Donde la comunicación entre proceso y sistema suceden, se verán conceptos y sistemas que *ROS* tiene para configurar el sistema, para manejar todo el proceso, para comunicar con más de una computadora, etc.
- **El nivel comunidad**: Se verá las herramientas y conceptos para compartir conocimiento, algoritmos, código desde cualquier desarrollador, es la parte más importante porque *ROS* está creciendo rápidamente con el apoyo de la comunidad.

#### 2.2.4.1. Nivel de sistema de archivos

Cuando uno empieza a usar o desarrollar un proyecto en *ROS* nos veremos en la necesidad de ver este concepto ya que al principio podría ser extraño o de difícil compresión, pero a medida que uno vaya usando *ROS* este concepto se volverá familiar para nosotros. Parecido a un sistema operativo, un programa en *ROS* es dividido en carpetas y estas carpetas contienen algunos archivos que describen sus funcionalidades.

***Packages***

Los *packages*, forman el nivel atómico de ROS. Un *package* tiene la estructura mínima y el contenido para crear un programa dentro de *ROS*. Puede tener el tiempo de ejecución de procesos (*nodes*), archivos de configuración y demás.

Usualmente cuando hablamos de *packages*, nos referimos a una estructura típica de archivos y carpetas [23]. Esta estructura muestra lo siguiente:

- `bin/`: Esta carpeta es donde nuestro programa compilado y enlazado son almacenados después de construirlos (*building*/*making*).
- `include/packages_name/`: Este directorio incluye los encabezados de las librerías que necesitaremos.
- `msg/`: Si se desarrolla mensajes no estándares, se los coloca aquí.

- `scripts/`: estos son *scripts*[16] ejecutables que pueden ser *bash*[17], *Python* o cualquier otro *script*.
- `src/`: Es el lugar donde los archivos fuentes de nuestro programa están (*source* file). Podemos crear una carpeta para *nodes* u organizarlo como deseemos.
- `CMakeList.txt`: Este es el archivo `CMake` de construcción.
- `manifest.xml`: Este es el archivo *package* `manifest`.

Para crear, modificar o trabajar con packages, *ROS* provee algunas herramientas:

- `rospack`: Este comando es usado para obtener información o encontrar un *package* en el sistema.
- `roscreate-pkg`: Este comando es usado para crear un *package* nuevo.
- `rosmake`: Este comando es usado para compilar un *package*.
- `rosdep`: Este comando instala las dependencias del sistema de un *package*.

Para el manejo de *package* y carpetas, al igual que los comandos en Linux, *ROS* provee comandos parecidos a Linux, tales como:

- `roscd`: Este comando es usado para cambiar de directorio.
- `roscp`: Este comando es usado para copiar un *package* desde otro *package*.
- `rosls`: Este comando muestra los archivos del *package*.

El archivo `manifest.xml` debe ser un *package* y es usado para especificar la información sobre el *package*. Si encontramos este archivo dentro de una carpeta, probablemente esta carpeta es un *package*.

***Stacks***

Los *stacks* son varios packages con algunas funcionalidades. En *ROS* existen muchos de estos *stacks* con diferentes usos, por ejemplo, el *stack* de navegación.

Los *packages* en *ROS* son organizados en *stacks*. Mientras que el objetivo de los *packages* es la de crear la mínima colección de código para su fácil re-uso, el objetivo de los *stacks* es simplificar el proceso de compartir código [22].

Un *stacks* necesita una estructura básica de archivos y carpetas. Se los puede crear manualmente o de forma automática, es decir mediante comandos proporcionados por *ROS*, tal como:

```
catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

***Messages***

Un *message* es la información que un proceso envía a otro proceso. *ROS* tiene muchos tipos de formatos de mensajes. Las descripciones de los mensajes son almacenadas en `mi_paquete/msg/MiTipoMessage.msg`.

[16] Es un documento que contiene instrucciones, escritas en códigos de programación.
[17] Bourne again Shell, es un programa informático, cuya función consiste en interpretar órdenes y un lenguaje de programación de consola. Está basado en la Shell de Unix.

*ROS* usa un lenguaje de descripción de mensaje simplificado para describir los valores de los datos que los *nodes* de *ROS* publican. Con esta descripción, *ROS* puede generar el correcto código fuente para estos tipos de mensajes en varios lenguajes de programación. *ROS* tiene muchos *messages* predefinidos, pero si desarrollamos un nuevo *message*, será en la carpeta `msg/` de nuestro *package*. Dentro de esa carpeta, algunos archivos con extensión `.msg` definen los *messages* [22].

Un *message* debe tener dos partes principales: campos y constantes. Los campos de un *message* definen el tipo de dato para ser transmitido en el *message*, por ejemplo: *`int32`*, *`float32`* y *`string`*, o nuevos tipos que hallamos creado antes, tal como *`type1`* y *`type2`*. Las constantes en un *message* definen el nombre de los campos. Un ejemplo de un archivo `.msg` es como:

- `int32 id`
- `float32 vel`
- `string name`

En *ROS* podemos encontrar muchos formatos estándar para usarlos en *messages* como se muestra en la siguiente tabla 2.1:

Tabla 2.1: Tipos de *message* estándar que se pueden utilizar.

| Tipo primitivo | Formato | C++ | Python |
|---|---|---|---|
| *bool* | *Unsigned 8 bit int* | *uint8_t* | *bool* |
| *int8* | *Signed 8 bit int* | *int8_t* | *int* |
| *uint8* | *Unsigned 8 bit int* | *uint8_t* | *int* |
| *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* |
| *uint64* | *Unsigned 64 bit int* | *uint64_t* | *long* |
| *float32* | *32 bit IEEE float* | *float* | *float* |
| *float64* | *64 bit IEEE float* | *double* | *float* |
| *string* | *ASCII string (4 bit)* | *std::string* | *string* |
| *time* | *Secs/nsecs signed 32 bit int* | *ros::Time* | *rospy.Time* |
| *duration* | *Secs/nsecs signed 32 bit int* | *ros::Duration* | *rospy.Duration* |

Fuente: Martinez, Aaron, and Enrique Fernández. *Learning ROS for robotics programming*. Packt Publishing Ltd, 2013.

***Services***

Es un mecanismo de comunicación entre *nodes*, que funciona a modo de petición/respuesta, es decir, el primer *node* envía una petición a otro y este responde. En este modelo solo dos *nodes* interactúan.

*ROS* usa un lenguaje de descripción de servicio simplificado para describir los tipos de servicios de *ROS*. Esto construye directamente sobre el formato `.msg` de *ROS* para habilitar la comunicación petición/respuesta entre nodes. La descripción *service* son almacenadas en los archivos `.srv` en el subdirectorio `srv/` de un *package* [22].

Para llamar un *service*, es necesario utilizar el nombre del *package* juntos con el nombre del *service*, por ejemplo, para el archivo `simple_package1/srv/sampe1.srv` nos referiremos a este archivo como `simple_package1/simple1`.

Existen algunas herramientas que llevan a cabo algunas funciones con *services*. La herramienta `rossrv` imprime la descripción del *service, packages* que contienen los archivos `.srv` y pueden encontrar los archivos fuentes que usa un tipo de *service*.

Si deseamos crear un *service, ROS* puede ayudarnos con el generador *services*. Solo necesitaremos adicionar la línea `gensrv()` para el archivo `CMakeLists.txt`.

Parecido al método usando publicadores y suscriptores los *services* sólo envían cuando hay requerimiento, mientras que el otro método se envían los datos de forma constante.

#### 2.2.4.2. Nivel gráfico de *ROS*

*ROS* proporciona herramientas para la comprensión de un proceso (programa o *script*) desarrollado, para ello proporciona un entorno gráfico para su fácil compresión.

**Nodes**

Los *nodes* son procesos donde los cálculos son realizados. Si deseamos tener un proceso que pueda interactuar con otros *nodes*, necesitaremos crear un *node* con este proceso para conectarlo a la red de *ROS*. Usualmente, un sistema tendrá muchos *nodes* para el control de diferentes funciones. Veremos que esto es mejor que tener que tener muchos *nodes* que proveen una única funcionalidad, en lugar de un gran *node* que hace todo el sistema. Los *nodes* son escritos con una librería de *ROS* cliente, por ejemplo: `roscpp` o `rospy`.

Los *nodes* en sí son ejecutables que pueden comunicarse con otros procesos usando *topics*, *services* o *parameter server*. Usar *nodes* en *ROS* nos provee con tolerancia a fallos y separando el código y funcionalidades haciendo el sistema simple. *ROS* tiene herramientas para el manejo de *nodes* y nos da información sobre ellos. Los comandos que se pueden utilizar son [22]:

- `rosnode info <node>`: Esto muestra la información sobre el *node*.
- `rosnode kill <node>`: Cancela un *node* ejecutándose o envía una señal.
- `rosnode list`: Muestra los *nodes* activos.
- `rosnode machine hostname`: Muestra los *nodes* ejecutándose sobre una máquina en particular o muestra la lista de máquinas.
- `rosnode ping <node>`: Esto prueba la conectividad con el *node*.
- `rosnode cleanup`: Elimina la información de registro desde un *node* inalcanzable.

***Topics***

Los *topics* son buses usados por los *nodes* para transmitir datos. Los *topics* pueden ser transmitidos sin una directa conexión entre *nodes*. Un *topic* puede tener varios suscriptores. Cada mensaje debe tener un nombre para ser direccionado por la red de *ROS*. Cuando un *node* envía datos, decimos que el *node* está publicando un *topic*. Los *nodes* pueden recibir *topics* de otros *nodes* simplemente suscribiéndose al *topic*.

Un *node* puede suscribirse a un *topic* y no es necesario que el *node* que está publicando este *topic* deba existir. Esto nos permite desacoplar la producción del consumo. Es importante que el nombre del *topic* deba ser único para evitar problemas y confusión entre *topics* con el mismo nombre.

Los *topics* en *ROS* pueden ser transmitidos usando TCP/IP[18] y UDP[19]. El transporte basado en TCP/IP es conocido como TCPROS y usa la conexión TCP/IP, esta es la transporte por defecto usado por *ROS*. Mientras que el transporte basado en UDP es conocido como UPDROS y es de baja latencia, con bajas perdidas, lo que le hace adecuado para tareas como tele-operación [20].

*ROS* también provee herramientas para trabajar con los *topics*, tal como `rostopic`, el cual se puede usar para los siguientes comandos:

- `rostopic bw/topic`: Muestra el ancho de banda usado por el *topic*.
- `rostopic echo/topic`: Muestra el *message* en la pantalla, es decir los datos que actualmente se encuentran en el *topic*.
- `rostopic find message_type`: Encuentra los *topics* por su tipo.
- `rostopic hz/topic`: Muestra la velocidad de publicación del *topic*.
- `rostopic info/topic`: Muestra la información sobre el *topic* activo, los *topics* que están publicando, los que se están suscribiendo y los *services*.
- `rostopic list`: Muestra los *topics* activos.
- `rostopic pub/topic type args`: Este comando publica datos en el *topic*. Esto nos permite crear o publicar datos en cualquier *topic* que nosotros queramos directamente desde la línea de comandos.
- `rostopic type/topic`: Muestra el tipo de *topic*, el tipo de *message* que está publicando.

***Services***

Cuando publicamos *topics*, estamos enviando datos, pero cuando necesitamos una petición o una respuesta de un *node*, no podemos hacer eso con *topics*. Los *services* nos dan la posibilidad para interactuar con los *nodes*. También, los *services* deben tener un único nombre. Cuando un *node* tiene un *service*, todos los *nodes* pueden comunicarse con él, gracias a la librería de *ROS client*.

Los *services* son desarrollados por el usuario y los *services* estándar que no existen para los *nodes*. Los archivos con el código fuente de los *messages* son almacenados en la carpeta `srv`.

Como los *topics*, *services* tienen un tipo de *service* asociado, que es el nombre de la extensión del *package* del archivo `.srv`. Por ejemplo, el archivo `chapter2_tutorials/srv/chapter2_srv1.srv` tiene el tipo *service* `chapter2_tutorials/chapter2_srv1`.

[18] *Transmission Control Protocol/Internet Protocol* sistema de protocolos que hacen posibles Telnet y E-mail.
[19] *User Datagram Protocol*, protocolo sin conexión, proporciona muy pocos servicios de recuperación de errores, ofreciendo una manera directa de enviar y recibir datagramas a través de una red IP.

*ROS* proporciona dos herramientas para poder ejecutar en la línea de comandos para poder trabajar con los *services*, `rossrv` y `rosservice`. Con `rossrv`, podemos ver información sobre la estructura de los datos de los *services*. Y con `rosservice`, los siguientes comandos están disponibles [22]:

- `rosservice call/<service> args`: Esto llama el *service* con los argumentos provistos.
- `rosservice find msg-type`: Encuentra el *service* de acuerdo al tipo.
- `rosservice info /<service>`: Muestra información sobre el *service*.
- `rosservice list`: Muestra una lista de los *services* activos y disponibles para invocado con el comando `call/<service> args`.
- `rosservice type /<service>`: Muestra el tipo de *service*.

***Messages***

Los *nodes* se comunican con otros a través de *messages*, un *message* contiene datos para enviar información a otros *nodes*. *ROS* tiene muchos tipos de *messages* y podemos también desarrollar nuestro propio tipo de *message* usando los *messages* estándar.

Los tipos de *messages* usan la siguiente convención estándar para nombrarlos: el nombre del *package*, seguido de / y el nombre del archivo `.msg`. Por ejemplo, `std_msgs/msg/String.msg` tiene el tipo de *message*, `std_msgs/String`.

Al igual que los anteriores, *ROS* también proporciona herramientas para poder obtener información sobre los *messages*, tales comandos son [22]:

- `rosmsg show`: Muestra los campos de un *message*.
- `rosmsg list`: Muestra una lista de todos los *messages*.
- `rosmsg package`: Muestra una lista de todos los *messages* en un *package*.
- `rosmsg packages`: Muestra una lista de todos los *packages* que tienen el *message*.
- `rosmsg users`: Busca archivos de código que utilizan el tipo de *message*.

***Bags***

Los *bags* son formatos para guardar y reproducir los datos de ROS message, son importantes mecanismos para el almacenamiento de datos, tal como los datos de un sensor, que pueden ser difíciles de recoger pero es necesario para el desarrollo y pruebas de los algoritmos. Usaremos bastantes los *bags* mientras trabajemos con robots mucho más complejos.

Un *bag* es un archivo creado por *ROS* con la extensión `.bag`. Los archivos *bag* pueden ser reproducidos en *ROS* como una sesión real[20], enviando los *topics* al mismo tiempo con los mismos datos. Normalmente esto se usa para depurar nuestros algoritmos. Los siguientes comandos pueden ser utilizados en la línea de comandos [22]:

- `rosbag`: Es usado para grabar, reproducir y otras operaciones de desempeño.
- `rxbag`: Usado para visualizar los datos en un entorno gráfico.

[20] Sesión real se refiere cuando se realiza la ejecución del programa en *ROS*, procesando los datos, pero estos datos ya están almacenados en los *bags*.

***Master***

El *master* provee los registros de nombres y busca para el resto de los *nodes*, sino lo tenemos en el sistema, no se podrá comunicar con los *nodes*, *services*, *messages* y otros. Pero es posible tenerlo en una computadora donde los *nodes* están trabajando en otras computadoras, es decir, no es necesario que el *master* y los *nodes* estén en la misma computadora.

Se encarga de rastrear los publicadores y suscriptores a los *topics*, es decir habilitar los *nodes* individuales para que los *nodes* se puedan encontrar con otros *nodes* y así poder comunicarse [22]. El *master* también provee el *parameter server*. Para poder ejecutar el *master* se usa el comando `roscore`, el cual carga el *ROS master* con otros componentes esenciales.

***Parameter server***

El *parameter server* nos da la posibilidad de tener datos almacenados usando las llaves de una ubicación central. Con estos parámetros es posible configurar *nodes* mientras está corriendo o cambiar el trabajo de los *nodes*.

Un *parameter server* es un diccionario multivariable compartido que es accesible vía una red. Los *nodes* usan este *server* para almacenar y recuperar parámetros en los tiempos de ejecución de los *nodes*. *ROS* también proporciona la herramienta `rosparam` para trabajar con el *Parameter Server*, los comandos que se pueden utilizar son [22]:

- `rosparam list`: Muestra una lista de todos los *parameters* en el *server*.
- `rosparam get parameter`: Se obtiene el valor de un *parameter*.
- `rosparam set parameter value`: Colocamos un valor dado de un *parameter*.
- `rosparam delete parameter`: Borramos un *parameter*.
- `rosparam dump file`: Guarda el p*arameter server* a un archivo.
- `rosparam load file`: Carga un archivo (con *parameters*) sobre el *parameter server*.

### 2.2.4.3. Nivel comunidad de *ROS*

El concepto de nivel de comunidad de *ROS* son recursos de *ROS* que permite separar comunidades para el intercambio de software y conocimiento, estos recursos incluyen:

- **Distribuciones:** Las distribuciones de *ROS* son colecciones de versiones apiladas que podemos instalar de acuerdo a nuestra versión de sistema operativo (Linux, Mac, etc.) haciendo fácil la instalación de *ROS*.
- **Repositorios:** ROS se basa sobre una red federada de repositorios de códigos, donde diferentes instituciones pueden desarrollar y pueden publicar los componentes de sus propios robots.
- **La wiki de ROS:** Es el principal foro para documentar de forma ordenada la información sobre *ROS*, cualquiera puede acceder a una cuenta y contribuir (mejoras de código, tutoriales, pruebas, recomendaciones, etc.) para crear nuestra propia documentación.
- **Envío de listas:** *ROS* usa el envío de listas como canal primario de comunicación sobre nuevas actualizaciones para *ROS*, como un foro para preguntar sobre el software de *ROS*.

**Navegando a través de nuestro sistema de archivos de *ROS***

Una vez revisado conceptos previos hasta aquí, revisaremos todo lo visto con diferentes códigos proporcionados por la comunidad de *ROS*.

Lo primero que se debe hacer es descarga desde los repositorios uno de los más utilizados es *GitHub*[21], los códigos que necesitemos, para luego crear un espacio de trabajo (*workspace*) y luego construirlo a través de líneas de comandos y poder utilizar el código. En los siguientes temas veremos cómo hacer lo mencionado anteriormente, esto nos servirá para relacionarnos con el entorno de *ROS*, y desde luego debemos tener algunos conocimientos básicos sobre Ubuntu.

**Creando nuestro propio *workspace***

Como cualquier otro programa, en nuestro sistema necesitaremos un lugar donde trabajar, en este caso se requiere un área de trabajo para crear nuestros *stacks* y *packages*, y su posterior modificación en caso de ser necesario.

Antes de hacer cualquier labor en *ROS*, lo que tenemos que hacer es crear nuestro propio *workspace*. En este *workspace*, tendremos todo nuestro código que usaremos para esta parte.

Entonces abrimos una terminal en Ubuntu y ejecutamos el siguiente comando:

Terminal en Ubuntu 2-1

```
$ mkdir -p ~/ws/src
$ cd ~/ws/src
$ catkin_init_workspace
Creating symlink "/home/curso/ ws /src/CMakeLists.txt" pointing to
"/opt/ros/indigo/share/catkin/cmake/toplevel.cmake"
```

Lo que crea un *workspace* vacío, es decir no existe ningún *package* tan solo un archivo `CMakeLists.txt`, con este archivo es suficiente para construir el *workspace* y para hacerlo utilizamos la herramienta `catkin`[22] creando las carpetas `src` y `devel` como se muestra en la figura 2.5.

Terminal en Ubuntu 2-2

```
christian@yeymi:~/ ws /-> catkin_make
Base path: /home/curso/ ws
Source space: /home/curso/ ws /src
Build space: /home/curso/ ws /build
Devel space: /home/curso/ ws /devel
Install space: /home/curso/ ws /install
.
-- Build files have been written to: /home/curso/ ws /build
####
#### Running command: "make -j2 -l2" in "/home/curso/ ws /build"
####
```

[21] Es una plataforma de desarrollo colaborativo de software para alojar proyectos utilizando el sistema de control de versiones *Git*.

[22] Es el sistema de construcción oficial de *ROS* y sucesor del sistema original de *ROS*, `rosbuilt`.

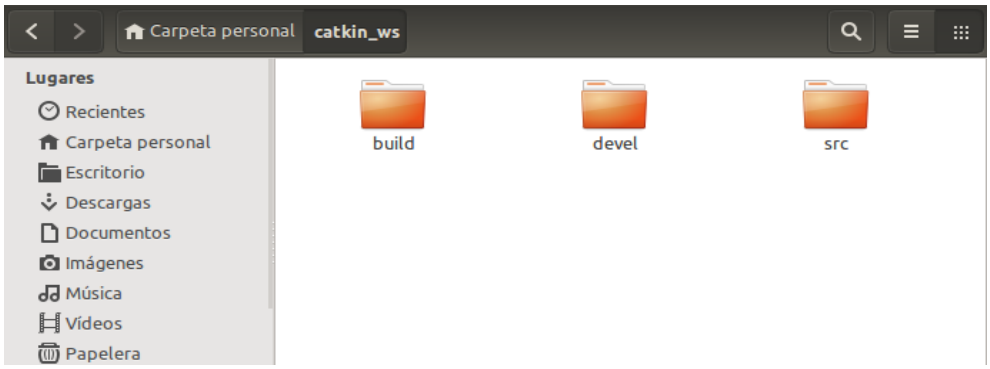



Figura 2.5. Carpetas creadas.
Fuente: Elaboración propia.

Hay que tener en cuenta que un *workspace* no está listo para ser usado si no tienen las carpetas: `source` , `build` y `devel`, esto es importante para *ROS*, ya que en nuestra carpeta `devel` se encuentra nuestro archivo `setup.bash` como se observa en la figura 2.6, el cual *ROS* necesita para saber en que workspace trabajaremos, por ello, podemos tener varios *workspace* creados y construidos pero *ROS* no los reconocerá como tal sino se le indica a *ROS* la ruta donde se encuentra el archivo `setup.bash`, esto lo hacemos cada vez que abrimos una terminal[23] nueva, es decir:

| Terminal en Ubuntu 2-3 |
| --- |
| `christian@yeymi:~/ws/-> source devel/setup.bash` |

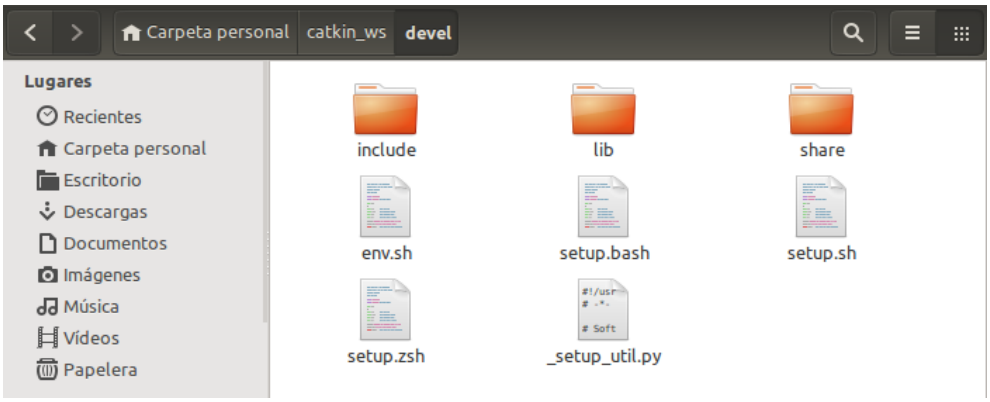



Figura 2.6. Archivo `setup.bash` necesario para nuestro *workspace*.
Fuente: Elaboración propia.

Para saber que estamos trabajando en nuestro *workspace* deseado, usamos el comando:

| Terminal en Ubuntu 2-4 |
| --- |
| `christian@yeymi:~/-> echo $ROS_PACKAGE_PATH`<br>`/home/curso/ws/src:/opt/ros/indigo/share:/opt/ros/indigo/stacks` |

[23] Es un intérprete de órdenes que permite al usuario interactuar con el ordenador sin utilizar una interfaz gráfica, se la conoce también como línea de comandos.

Para los usuarios que están más familiarizados con el uso de Linux y demás, podría resultar útil agregar el `setup.bash` del *workspace* que deseamos al archivo con extensión `.bashrc`[24] de Ubuntu directamente, este procedimiento se realizará de forma permanente y ya no será necesario decirle cada vez que abramos un terminal nuevo para especificar la ruta de nuestro archivo `setup.bash`.

### Creando un *package*

Como se mencionó anteriormente, podemos crear un *package* manualmente. Para evitar este tedioso trabajo, usaremos comandos proporcionados por *ROS* como se muestra en la siguiente terminal.

Terminal en Ubuntu 2-5

```
christian@yeymi:~/ws/src/-> roscreate-pkg noob_tutorial std_msgs rospy
roscpp
Created package directory /home/curso/ws/src/noob_tutorial
Created include directory
/home/curso/ws/src/noob_tutorial/include/noob_tutorial
Created cpp source directory /home/curso/ws/src/noob_tutorial/src
Created package file /home/curso/ws/src/noob_tutorial/Makefile
Created package file /home/curso/ws/src/noob_tutorial/manifest.xml
Created package file /home/curso/ws/src/noob_tutorial/CMakeLists.txt
Created package file /home/curso/ws/src/noob_tutorial/mainpage.dox

Please edit noob_tutorial/manifest.xml and mainpage.dox to finish
creating your package
```

Para identificar un *package* debe contener los archivos que se encuentran dentro de este como se muestra en la figura 2.7. El formato del comando incluye el nombre del *package* y las dependencias que tendrá el *package*, en nuestro caso las dependencias son: `std_msgs`, `rospy` y `roscpp`. Estas dependencias indican lo siguiente:

- `std_msgs`: Este contiene el tipo común de *message* representando los tipos de datos primitivos y otro constructor básico de *message*, tal como *multiarrays*.
- `rospy`: Esto es una librería cliente en *Python* para *ROS*.
- `roscpp`: Al igual que `rospy` solo que esto es en C++, permite a los programadores rápidamente conectarse con los *topics*, *services* y *parameters* de *ROS*.

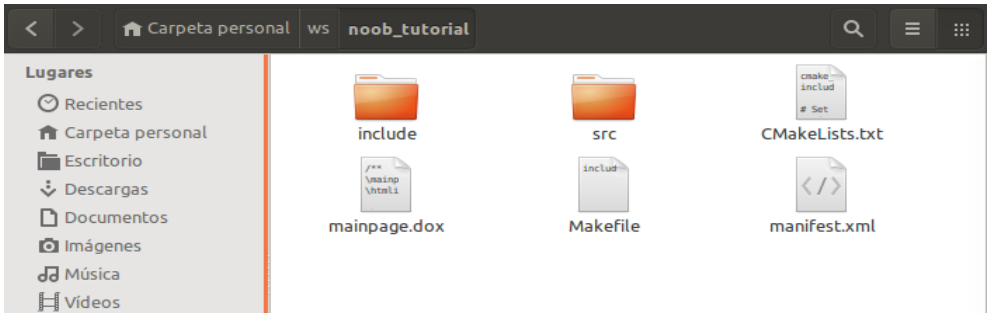



Figura 2.7: Archivos generados en la creación de nuestro *package*.
Fuente: Elaboración propia.

[24] Es un script que se ejecuta cada vez que una nueva sesión de terminal server se inicia en el modo interactivo, esto se da cada vez que se abre una nueva ventana de terminal.

**Construyendo un *package***

Una vez creado nuestro *package* el siguiente paso será construirlo, para esto, en versiones anteriores de *ROS* se utilizaba el comando `rosmake`, pero versiones recientes *ROS* utiliza la herramienta *catkin* con el comando `catkin_make`, lo que facilita la construcción de *packages* en *ROS*. Entonces simplemente ejecutamos el comando dentro de nuestro *workspace*, es decir:

Terminal en Ubuntu 2-6

```
christian@yeymi:~/ws/src/-> cd ..
christian@yeymi:~/ws/-> catkin_make
Base path: /home/curso/ws
Source space: /home/curso/ws/src
Build space: /home/curso/ws/build
Devel space: /home/curso/ws/devel
Install space: /home/curso/ws/install
####
#### Running command: "make cmake_check_build_system" in
"/home/curso/ws/build"
####
#### Running command: "make -j2 -l2" in "/home/curso/ws/build"
```

Si revisamos el archivo `manifest.xml`, veremos que las dependencias de nuestro *package* están declaradas de forma correcta.

Archivo `manifest.xml`

```
<package>
  <description brief="noob_tutorial">

     noob_tutorial

  </description>
  <author>curso</author>
  <license>BSD</license>
  <review status="unreviewed" notes=""/>
  <url>http://ros.org/wiki/noob_tutorial</url>
  <depend package="std_msgs"/>
  <depend package="rospy"/>
  <depend package="roscpp"/>
</package>
```

**Jugando con los *nodes***

Para practicar y aprender sobre los *nodes*, usaremos un típico *package* como ejemplo en *ROS* llamado *turtlesim*.

Antes de empezar cualquier cosa, debemos de inicializar el `roscore`.

Terminal en Ubuntu 2-7

```
christian@yeymi:~/-> roscore
... logging to /home/curso/.ros/log/e65420e8-8a53-11e6-b384-
000c29a26cfa/roslaunch-ubuntu-21957.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.
```

```
started roslaunch server http://ubuntu:47333/
ros_comm version 1.11.20

SUMMARY
========

PARAMETERS
 * /rosdistro: indigo
 * /rosversion: 1.11.20

NODES

auto-starting new master
process[master]: started with pid [21969]
ROS_MASTER_URI=http://ubuntu:11311/

setting /run_id to e65420e8-8a53-11e6-b384-000c29a26cfa
process[rosout-1]: started with pid [21982]
started core service [/rosout]
```

Podemos utilizar los comandos presentados anteriormente en *topics*, *nodes*, etc. Pero ahora solo iniciaremos un nuevo *node*, con el comando `rosrun`:

Terminal en Ubuntu 2-8

```
christian@yeymi:~/-> rosrun turtlesim turtlesim_node
[ INFO] [1475600540.442868811]: Starting turtlesim with node name
/turtlesim
[ INFO] [1475600540.491379739]: Spawning turtle [turtle1] at
x=[5,544445], y=[5,544445], theta=[0,000000]
```

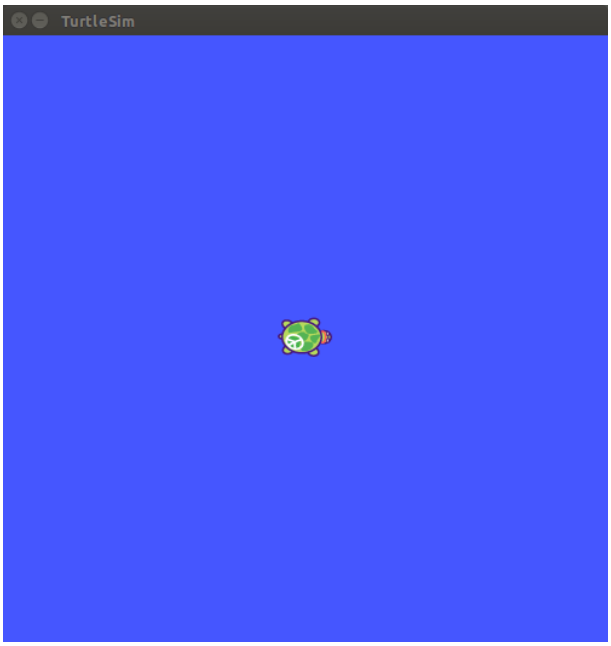



Figura 2.8: Tortuga generada por el *node* `/turtlesim`.
Fuente: Elaboración propia.

Con ello iniciamos una tortuga tal como se muestra en la figura 2.8. Si ejecutamos el comando para ver los *nodes* activos, veremos el *node* `/turtlesim`

Terminal en Ubuntu 2-9

```
christian@yeymi:~/-> rosnode info /turtlesim
--------------------------------------------------------------------
Node [/turtlesim]
Publications:
 * /turtle1/color_sensor [turtlesim/Color]
 * /rosout [rosgraph_msgs/Log]
 * /turtle1/pose [turtlesim/Pose]

Subscriptions:
 * /turtle1/cmd_vel [unknown type]

Services:
 * /turtle1/teleport_absolute
 * /turtlesim/get_loggers
 * /turtlesim/set_logger_level
 * /reset
 * /spawn
 * /clear
 * /turtle1/set_pen
 * /turtle1/teleport_relative
 * /kill


contacting node http://ubuntu:36711/ ...
Pid: 22086
Connections:
 * topic: /rosout
    * to: /rosout
    * direction: outbound
    * transport: TCPROS
```

**Interactuando con *topics***

Para interactuar con los *topics*, usaremos las herramientas proporcionadas por *ROS*.

Con el parámetro `pub`, podemos publicar *topics* que pueden suscribirse a cualquier *node*, solo necesitamos publicar el *topic* con el nombre correcto, eso haremos más adelante. Ahora usaremos un *node* que trabaje para nosotros, en otra *terminal* ejecutamos el comando siguiente para poder mover como se observa en la figura 2.9 la tortuga generada:

Terminal en Ubuntu 2-10

```
christian@yeymi:~/-> rosrun turtlesim turtle_teleop_key
Reading from keyboard
---------------------------
Use arrow keys to move the turtle.
```

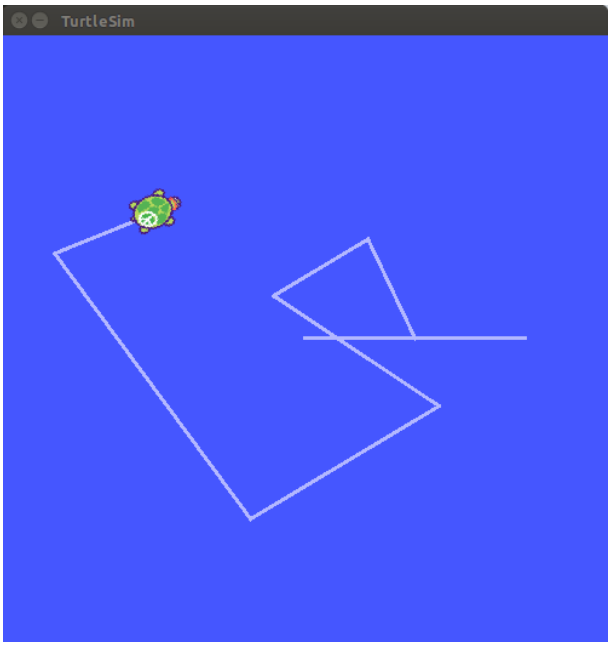



Figura 2.9: Tortuga guiado mediante el teclado.
Fuente: Elaboración propia.

Si queremos ver la información de los *nodes* /teleop_turtle y /turtlesim, podemos podemos verla en los códigos de los *nodes* que existe un *topics* llamado `*/turtle1/cmd_vel [geometry_msgs/Twist]` en la sección publicaciones del primer *node*, en la sección suscripciones del segundo *node*, existe `*/turtle1/cmd_vel [geometry_msgs/Twist]` [24]:

Terminal en Ubuntu 2-11

```
christian@yeymi:~/-> rosnode info /teleop_turtle
--------------------------------------------------------------------------------
Node [/teleop_turtle]
Publications:
 * /turtle1/cmd_vel [geometry_msgs/Twist]
 * /rosout [rosgraph_msgs/Log]

Subscriptions: None

Services:
 * /teleop_turtle/get_loggers
 * /teleop_turtle/set_logger_level


contacting node http://ubuntu:60490/ ...
Pid: 22225
Connections:
 * topic: /rosout
    * to: /rosout
    * direction: outbound
    * transport: TCPROS
 * topic: /turtle1/cmd_vel
    * to: /turtlesim
    * direction: outbound
    * transport: TCPROS
```

Ahora tenemos el *node* `/turtlesim`:

Terminal en Ubuntu 2-12

```
christian@yeymi:~/-> rosnode info /turtlesim
--------------------------------------------------------------------------------
Node [/turtlesim]
Publications:
 * /turtle1/color_sensor [turtlesim/Color]
 * /rosout [rosgraph_msgs/Log]
 * /turtle1/pose [turtlesim/Pose]
Subscriptions:
 * /turtle1/cmd_vel [geometry_msgs/Twist]
Services:
 * /turtle1/teleport_absolute
 * /turtlesim/get_loggers
 * /turtlesim/set_logger_level
 * /reset
 * /clear
 * /turtle1/set_pen
 * /turtle1/teleport_relative
 * /kill
contacting node http://ubuntu:36711/ ...
Pid: 22086
Connections:
 * topic: /rosout
    * to: /rosout
    * direction: outbound
    * transport: TCPROS
 * topic: /turtle1/cmd_vel
    * to: /teleop_turtle (http://ubuntu:60490/)
    * direction: inbound
    * transport: TCPROS
```

Como se ve, cuando se ejecutó la primera vez `rosnode info /turtlesim` no mostraba ninguna suscripción, pero cuando se ejecutó el *node* `/teleop_turtle` que publico el *topic* `* /turtle1/cmd_vel [geometry_msgs/Twist]` el *node* `/turtlesim` tenía el mismo *topic* para suscribirse. Con el *parameter* `echo`, podemos ver información del *node* que deseemos, para ello, usamos:

Terminal en Ubuntu 2-13

```
christian@yeymi:~/-> rostopic echo turtle1/cmd_vel
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
linear:
  x: 0.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 2.0
```

También podemos ver el tipo de *message* enviado por el *topic* usando el comando:

Terminal en Ubuntu 2-14

```
christian@yeymi:~/-> rostopic type turtle1/cmd_vel
geometry_msgs/Twist
```

Si deseamos ver los campos del *message*, podemos hacerlo con el siguiente comando:

Terminal en Ubuntu 2-15

```
christian@yeymi:~/-> rosmsg show geometry_msgs/Twist

geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Estas herramientas son usadas porque con esta información, podemos publicar *topics* usando el comando: `rostopic pub [topic] [msg_type] [args]`

Terminal en Ubuntu 2-16

```
christian@yeymi:~/-> rostopic pub -1 /turtle1/cmd_vel
geometry_msgs/Twist -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'

publishing and latching message for 3.0 seconds
```

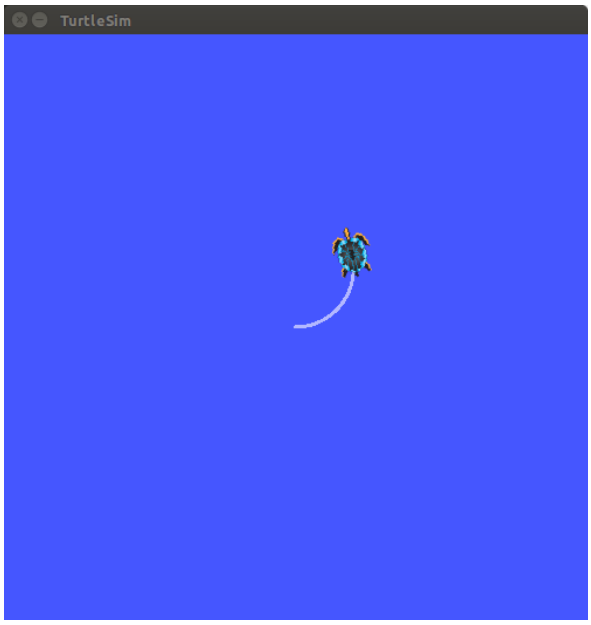



Figura 2.10: Tortuga, ejemplo de uso del comando: `rostopic pub`.
Fuente: Elaboración propia.

Con lo que se genera una trayectoria de un sector circular como se muestra en la figura 2.10.

**Usando los *services***

Ahora veremos los *services* disponibles para el *node* `/turtlesim`:

Terminal en Ubuntu 2-17

```
christian@yeymi:~/-> rosservice list
/clear
/kill
/reset
/rosout/get_loggers
/rosout/set_logger_level
/spawn
/teleop_turtle/get_loggers
/teleop_turtle/set_logger_level
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/get_loggers
/turtlesim/set_logger_level
```

Si queremos ver el tipo de cualquier *service*, simplemente usamos el siguiente comando:

Terminal en Ubuntu 2-18

```
christian@yeymi:~/-> rosservice type /clear
std_srvs/Empty
```

Ahora con el uso de los *service* crearemos otra tortuga en diferente lugar con diferente orientación pero en el mismo entorno de la primera tortuga, pero primero veamos el *service* `spawn` para conocer el formato, es decir:

Terminal en Ubuntu 2-19

```
christian@yeymi:~/-> rosservice type /spawn | rossrv show
float32 x
float32 y
float32 theta
string name
---
string name
```

Con estos campos, conocemos como invocar el *service* `spawn`, la posición, orientación y el nombre que deseamos [24]:

Terminal en Ubuntu 2-20

```
christian@yeymi:~/-> rosservice call /spawn 2 2 0.2 ""
name: turtle2
```

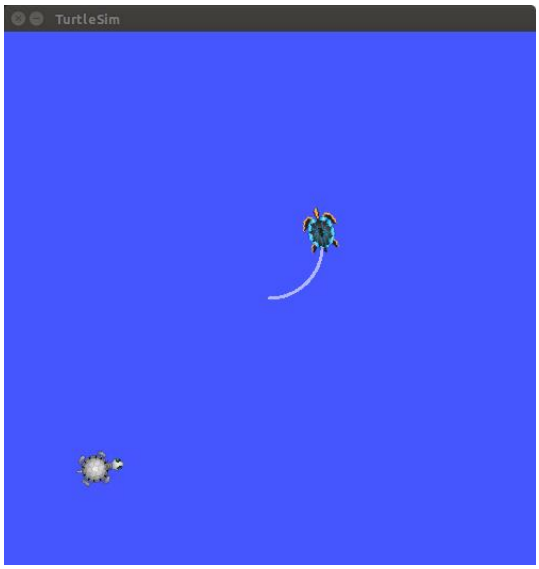



Figura 2.11: Tortuga, ejemplo para el uso de *services*.
Fuente: Elaboración propia.

Utilizando el service `spawn` se situa otra tortuga como se observa en la figura 2.11.

**Usando el *parameter server***

El *parameter server* es usado para almacenar datos para que sea accesible a todos los nodes. *ROS* tiene la herramienta llamada `rosparam`.

Terminal en Ubuntu 2-21

```
christian@yeymi:~/-> rosparam list
/background_b
/background_g
/background_r
/rosdistro
/roslaunch/uris/host_ubuntu__47333
/rosversion
/run_id
```

Los *parameters background* son del *node* `/turtlesim`. Estos *parameters* cambian el color de la ventana que inicialmente es azul. Si deseamos leer el valor, utilizamos `get parameter` y para poder colocar un nuevo valor usamos `set parameter` [24] y el resultado del cambio de color se observa en la figura 2.12:

Terminal en Ubuntu 2-22

```
christian@yeymi:~/-> rosparam get /background_b
255
christian@yeymi:~/-> rosparam set /background_b 100
christian@yeymi:~/-> rosservice call /clear
```

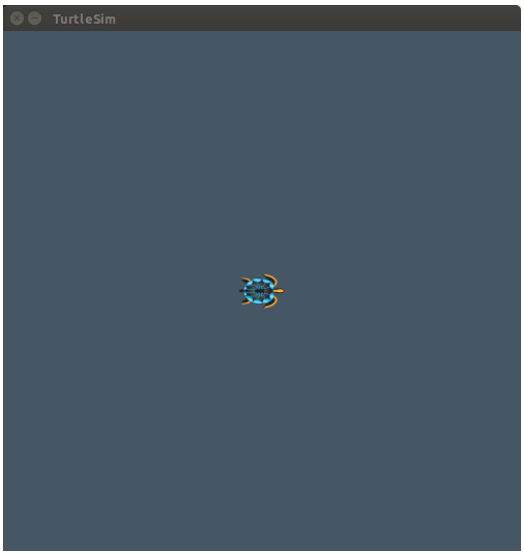



Figura 2.12: Tortuga, ejemplo para el uso de *services* (cambio de color de fondo).
Fuente: Elaboración propia.

### Creando un *node*

En esta sección crearemos *nodes* y manejaremos los datos.

Terminal en Ubuntu 2-23

```
christian@yeymi:~/ws/-> source devel/setup.bash
christian@yeymi:~/ws/src/-> roscd noob_tutorial/
```

En esta parte crearemos dos *nodes*, `listener.py` y `talker.py`, un *node* "escuchará" lo que el otro *node* "habla" [24].

Programa `listener.py`

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(data):
  rospy.loginfo(rospy.get_caller_id() + "Yo escucho %s", data.data)

def listener():
  # In ROS, nodes are uniquely named. If two nodes with the same
  # node are launched, the previous one is kicked off. The
  # anonymous=True flag means that rospy will choose a unique
  # name for our 'listener' node so that multiple listeners can
  # run simultaneously.
  rospy.init_node('listener', anonymous=True)

  rospy.Subscriber("chatter", String, callback)

  # spin() simply keeps python from exiting until this node is stopped
  rospy.spin()

if __name__ == '__main__':
  listener()
```

Programa `talker.py`

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def talker():
  pub = rospy.Publisher('chatter', String, queue_size=10)
  rospy.init_node('talker', anonymous=True)
  rate = rospy.Rate(10) # 10hz
  while not rospy.is_shutdown():
    hello_str = "Hola mundo %s" % rospy.get_time()
    rospy.loginfo(hello_str)
    pub.publish(hello_str)
    rate.sleep()

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

Una vez creados los *nodes*, procedemos a convertirlos en ejecutables, con el siguiente comando:

Terminal en Ubuntu 2-24

```
christian@yeymi:~/ws/src/noob_tutorial/src/-> chmod +x listener.py
christian@yeymi:~/ws/src/noob_tutorial/src/-> chmod +x talker.py
```

**Construyendo un node**

Usamos `CMake` como nuestro sistema constructor, es decir, accedemos a nuestro *workspace* y lo construimos:

Terminal en Ubuntu 2-25

```
christian@yeymi:~/-> cd ws
christian@yeymi:~/ws/-> catkin_make
Base path: /home/curso/ws
Source space: /home/curso/ws/src
Build space: /home/curso/ws/build
Devel space: /home/curso/ws/devel
Install space: /home/curso/ws/install
####
#### Running command: "make cmake_check_build_system" in
"/home/curso/ws/build"
####
####
#### Running command: "make -j2 -l2" in "/home/curso/ws/build"
####
```

Para ejecutar los *nodes* creados, abrimos dos terminales para cada *node*, y ejecutamos el comando `rosrun`:

Terminal en Ubuntu 2-26

```
christian@yeymi:~/-> christian@yeymi:~/ws/-> rosrun noob_tutorial
talker.py
[INFO] [WallTime: 1475611077.132792] Hola mundo 1475611077.13
```

```
[INFO] [WallTime: 1475611077.232246] Hola mundo 1475611077.23
[INFO] [WallTime: 1475611077.331879] Hola mundo 1475611077.33
[INFO] [WallTime: 1475611077.432652] Hola mundo 1475611077.43
```

Terminal en Ubuntu 2-27

```
christian@yeymi:~/ws/-> rosrun noob_tutorial listener.py
[INFO] [WallTime: 1475611076.534857] /listener_23219_1475611066621Yo
escucho Hola mundo 1475611076.53
[INFO] [WallTime: 1475611076.636445] /listener_23219_1475611066621Yo
escucho Hola mundo 1475611076.63
[INFO] [WallTime: 1475611076.735227] /listener_23219_1475611066621Yo
escucho Hola mundo 1475611076.73
[INFO] [WallTime: 1475611076.836380] /listener_23219_1475611066621Yo
escucho Hola mundo 1475611076.83 [INFO] [WallTime: 1475611077.331879]
Hola mundo 1475611077.33
[INFO] [WallTime: 1475611077.432652] Hola mundo 1475611077.43
```

**Creando los archivos msg y svr**

En esta parte, crearemos el *node service* (`add_two_ints_server`), el cual recibe dos números enteros y regresa la suma [24].

Programa `add_two_ints_server.py`

```
#!/usr/bin/env python
from noob_tutorial.srv import *
import rospy

def handle_add_two_ints(req):
  print "Devolviendo la suma [%s + %s = %s]"%(req.a, req.b, (req.a +
req.b))
  return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
  rospy.init_node('add_two_ints_server')
  s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
  print "Listo para sumar 2 numeros enteros."
  rospy.spin()

if __name__ == "__main__":
  add_two_ints_server()
```

En esta otra parte, crearemos el *node client* (`add_two_ints_client`).

Programa `add_two_ints_client.py`

```
#!/usr/bin/env python
import sys
import rospy
from noob_tutorial.srv import *

def add_two_ints_client(x, y):
  rospy.wait_for_service('add_two_ints')
  try:
    add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
    resp1 = add_two_ints(x, y)
    return resp1.sum
  except rospy.ServiceException, e:
    print "Fallo el llamado del servicio: %s"%e
```

```
def usage():
  return "%s [x y]"%sys.argv[0]

if __name__ == "__main__":
  if len(sys.argv) == 3:
    x = int(sys.argv[1])
    y = int(sys.argv[2])
  else:
    print usage()
    sys.exit(1)
  print "Requeriendo %s+%s"%(x, y)
  print "%s + %s = %s"%(x, y, add_two_ints_client(x, y))
```

Igual tenemos que convertir estos *nodes* en ejecutables, es decir:

Terminal en Ubuntu 2-28

```
christian@yeymi:~/ws/src/noob_tutorial/-> chmod +x
src/add_two_ints_client.py
christian@yeymi:~/ws/src/noob_tutorial/-> chmod +x
src/add_two_ints_server.py
```

Ejecutamos los *nodes* creados:

Terminal en Ubuntu 2-29

```
christian@yeymi:~/-> cd ws/
christian@yeymi:~/ws/-> source devel/setup.bash
christian@yeymi:~/ws/-> rosrun noob_tutorial add_two_ints_server.py
Listo para sumar 2 numeros enteros.
Devolviendo la suma [43 + 12 = 55]
```

Por último, vemos que la suma se realiza de forma exitosa.

Terminal en Ubuntu 2-30

```
christian@yeymi:~/-> cd ws/
christian@yeymi:~/ws/-> source devel/setup.bash
christian@yeymi:~/ws/-> rosrun noob_tutorial add_two_ints_client.py 43
12
Requeriendo 43+12
43 + 12 = 55
```

Con estos ejemplos realizados, ahora estamos más familiarizados con *ROS* y tenemos los conocimientos básicos para adentrarnos en este magnífico *framework* y así poder crear nuestro código para el control del cuadricóptero, pero antes debemos revisar algunos conceptos sobre control y el algoritmo de perfil de velocidad trapezoidal para esta aplicación, esto se realizará en el siguiente capítulo.

# Capítulo 3

# Estrategias de control

## 3.1. Justificación de la estrategia de control a implementarse

Existen varios controladores desarrollados en el campo de los *UAVs*, algunos de los cuales usan el control clásico [1] que asume aproximaciones lineales del sistema sobre un punto de operación. Por otra parte, los métodos de control no lineal como los desarrollados por Vianna [3] son necesarios para obtener mejores desempeños y movimientos más agresivos durante su funcionamiento, pero dicha implementación de los controladores asume un costo computacional mayor a los controladores tradicionales. Los controladores no lineales desarrollados en otros trabajos [25], muchos de estos solo están implementados en simuladores. Los inconvenientes para implementarlos físicamente se deben a lo siguiente:

- Algoritmos muy complejos y costo computacional muy elevado.
- Sensores de precisión para la posición. En este caso se tienen en la actualidad dos casos, para vuelos *indoor* [26] (para entornos cerrados en la cual utiliza cámaras para determinar la posición del cuadricóptero por ejemplo) y para vuelos *outdoor* (para entornos a campo abierto, utiliza Sistema de Posicionamiento Global, tema en el que está enfocado este trabajo).

Cuando se trabaja con un sistema robótico (cuadricóptero para este caso) es recomendable utilizar curvas de velocidad compuestas de los movimientos básicos con velocidad constante o aceleración constante, debido a que le resulta imposible a un cuadricóptero salir de un estado de reposo a otro de velocidad constante o la de cambiar de un segmento de la trayectoria a una velocidad cualquiera a otra diferente bruscamente en un sólo periodo de muestreo debido a que esto conllevaría a realizar aceleraciones infinitas para lograrlo [11].

Las curvas de velocidad que se pueden componer fácilmente de movimientos a velocidad constante y aceleración constante pueden ser, por poner un ejemplo, en forma de trapecios. Estas curvas son en realidad funciones a trozos que tienen ciertas condiciones en su construcción, por ejemplo si se trata de una línea recta, para poder hallar la función que describe la evolución de cada una de las coordenadas en el tiempo sólo hay que referirse a las diferentes etapas de la gráfica de velocidad que puede estar compuesto de diferentes partes.

En este capítulo se explicará el control que se utilizará para nuestro cuadricóptero, un clásico controlador proporcional, integral y derivativo (PID) y la utilización de un algoritmo que se denomina perfil de velocidad trapezoidal. Dado que el propósito de este trabajo es el seguimiento de una trayectoria definida se decidió utilizar el algoritmo mencionado por las ventajas mencionadas anteriormente.

## 3.2. Estructuras de control

Como se mencionó en el primer capítulo al ser el cuadricóptero un sistema sub-actuado[25], es ahí donde se encuentra la complejidad de poder controlar dicho sistema. Muchos de los controladores desarrollados en diferentes trabajos [7], desarrollan e implementan los controladores para los ángulos del cuadricóptero pero pocos para el control de posición ya sea en ambientes cerrados [3], [26] o abiertos [27], [28], [29] dado a lo complejidad de la misma.

Para este trabajo, como se está trabajando con el sistema embebido *ArduCopter*[26], se enviarán los comandos de control de tal manera que el cuadricóptero ejecute la trayectoria deseada. En el siguiente capítulo se explicará con mejor detalle sobre las acciones de control.

### 3.2.1. Control proporcional, integral y derivativo

Es el controlador más usado (y lo continua siendo) debido a lo sencillo que resulta implementarlo con operaciones matemáticas sencillas, por lo que es utilizada para lazos de control simples, por ello que la mejor comprensión de sus características puede resultar útil para estos sistemas como ejemplo la técnica de *anti-windup* [30].

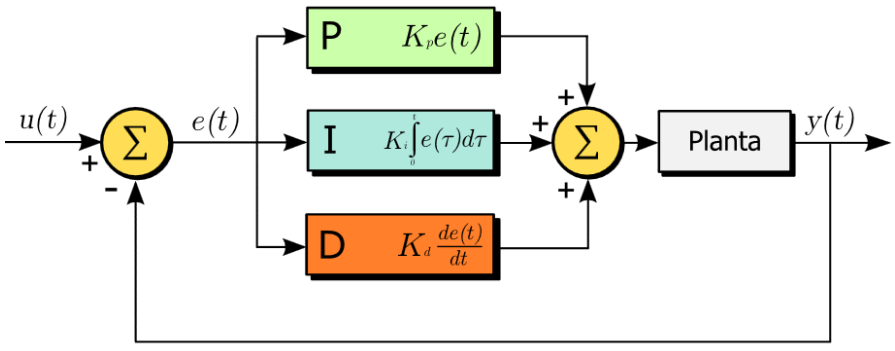



Figura 3.1. Diagramas de bloques del controlador PID implementando en una planta. Fuente: Wikipedia.

[25] Sistema cuyas variables manipulables no permiten regular todos los grados de libertad del mismo.
[26] Es una placa que se basa en *ArduPilot* creado por la comunidad *DIY Drones* con procesador *Atmel* 2560.

La estructura de control PID clásica, está compuesto por tres bloques principales: una parte proporcional, una parte derivativa y una parte integral como se observa en la figura 3.1. La ecuación 3.1 en el dominio del tiempo muestra la contribución de los bloques del controlador PID y sus constantes:

$$u(t) = K_p\left[e(t) + \frac{1}{T_i}\int_0^t e(\tau)\,d\tau + T_d\,\frac{de(t)}{dt}\right] \tag{3.1}$$

$$e(t) = u(t) - y(t) \tag{3.2}$$

Donde $u(t)$ es la entrada de control, $e(t)$ es el error que constituye la diferencia entre $u(t)$ y $y(t)$, $K_p$, $T_i$ y $T_d$ son los parámetros para los elementos proporcional, integral y derivativo del controlador PID.

- **Bloque P:** La acción del bloque proporcional es la de atenuar o amplificar la señal de control en contrapuesta con la señal de error, esto provoca una penalización en la salida del sistema cuando esta se aleje del valor de referencia. El parámetro de este bloque es $K_p$.
- **Bloque I:** El bloque integrador genera una variación en la señal de control, la misma para su cálculo hace uso de la integral de la señal de error (sumatoria). Visto de otro modo opera en base a la memoria de la señal de error. La principal ventaja de este bloque es la de minimizar o incluso eliminar el error en estado estacionario del sistema, pero esto conlleva a tener una respuesta lenta del sistema. El parámetro de este bloque es $K_i$ o expresada de otra forma $T_i$ (tiempo de integración), ambos parámetros cumplen la igualdad $K_i = \frac{K_p}{T_i}$.
- **Bloque D:** El bloque derivativo, obtiene la derivada de la señal de error permitiendo anticiparse a la tendencia de crecimiento de la señal de error, para luego generar una acción de control con anticipación que regule el sobrepaso máximo y los tiempos de respuesta del sistema. Parámetro $K_d$ o también de la forma $T_d$ (tiempo de diferenciación) cumpliendo la igualdad $K_d = K_p \cdot T_d$.

La ecuación 3.1 puede ser representado de la siguiente forma:

$$u(t) = u_p(t) + u_i(t) + u_d(t) \tag{3.3}$$

Con un tiempo de muestreo $T_s$, podemos obtener una expresión del controlador expresado en la ecuación 3.2 en forma discreta, para que sea posible implementar en un sistema embebido.

Para la acción proporcional realizamos la discretización para un sistema digital, es decir:

$$u_p(t) = K_p e(t) \rightarrow u_p(kT_s) = K_p e(kT_s) \rightarrow u_p(k) = K_p e(k) \tag{3.4}$$

La acción integral debemos tener la acción integral anterior, para ello usamos la memoria del sistema embebido, para las sumas acumulativas:

$$u_i(t) = K_p \frac{1}{T_i}\int_0^t e(\tau)\,d\tau \rightarrow u_i(kT_s) = K_p \frac{T_s}{T_i}\sum^{kT_s} e(iT_s)$$
$$u_i(k) = u_i(k-1) + K_p \frac{T_s}{T_i} e(k) \tag{3.5}$$

Y para la acción derivativa, debemos almacenar el error anterior, con todo esto ya podemos implementar nuestro algoritmo.

$$u_d(t) = K_p T_d \frac{de(t)}{dt}$$
$$u_d(k) = K_p \frac{T_d}{T_s}\left[e(k) - e(k-1)\right] \tag{3.6}$$

### 3.2.1.1. Técnica *Antiwindup*

En control de procesos, es sabido que cuando se implementa un controlador PID sino se considera la saturación de los actuadores los resultados pueden resultar distintos de lo diseñado. El efecto *windup* es conocido debido a que el controlador si tiene una acción integral y se satura los actuadores se rompe el lazo de realimentación, como resultado de esto el sistema puede tardar mucho tiempo en recuperarse para llegar a la referencia de nuevo después de que al sistema le llegué una perturbación [30]. Para evitar este efecto *windup* debemos tener en cuenta 2 aspectos:

- La contribución del término integral del controlador.
- La saturación de los actuadores físicos.

Usaremos el método de limitación del término integral, limitando el término integral haremos que su acción de control no salga de unos límites establecidos, con esto la señal de salida del controlador evitará crecer innecesariamente.

El código generado para este controlador, se muestra a continuación, su implementación con operaciones matemáticas básicas hace que este controlador sea fácilmente realizable, en el siguiente código se muestra la implementación de la misma y el cual se utilizará para nuestro sistema de control.

Algoritmo controlador PID con *anti-windup*

```
%-------Control Proporcional, Integral y Derivativo-----
%----------------con Anti Reset - Windup----------------
%--------------CHRISTIAN YEYMI MAMANI MAMANI------------
Up=Kp*(Wt-Yt)                              %Acción proporcional
Ui=Ui_last+Kp*(Ts/Ti)*(Wt-Yt)              %Acción integral
Ud=Kp*(Td/Ts)*(Wt-Yt)-(Wt_last-Yt_last)    %Acción derivativa
Ut=Up+Ui+Ud                                %Acción de control
if Ut<Umin, Ut=Umin                        %Restriccion minimo
if Ut>Umax, Ut=Umax                        %Restriccion maximo
Ui=Ut-Up-Ud                                %Anti Reset - Windup
Wt_last=Wt, Yt_last=Yt, Ui_last=Ui         %Almacenamos en memoria
```

Para el cuadricóptero nuestros actuadores son los motores y a la vez están controlados por unos controladores electrónicos de velocidad entonces necesitamos conocer los valores máximo y mínimos a los cuales trabaja, en el siguiente capítulo se verá con mayor detalle este proceso.

### 3.2.2. Control de los subsistemas del cuadricóptero

Se ha considerado importante realizar la simulación del modelo matemático, para ello me baso en el capítulo 1 y la utilización del lenguaje de programación *Python* con el fin de conocer el comportamiento del cuadricóptero e identificar las variables del mismo. Es importante señalar también que para la simulación de nuestro sistema de control de trayectoria, se utilizó el *stack* `hector_quadrotor` para ver resultados en un simulador 3D señalando que este *stack* ya viene con el modelo del cuadricóptero implementado, tan sólo se tiene que enviar los comandos de control.

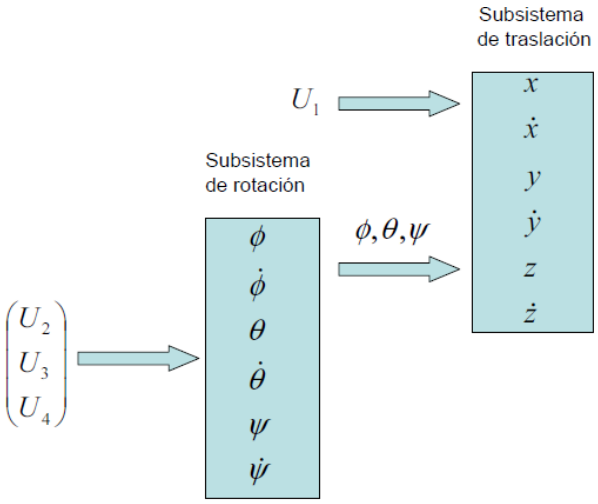



Figura 3.2. Subsistemas rotación y traslación de un cuadricóptero [3].

De acuerdo al trabajo de Vianna [3], el sistema de cuadricóptero puede ser dividido en dos subsistemas: rotación y traslación, como se observa en la figura 3.2.

#### 3.2.2.1. Control subsistema de rotación

Para estabilizar el cuadricóptero, un PID es utilizado, las ventajas del controlador PID son la simple estructura y fácil implementación del controlador, ya mencionados en el apartado anterior. Definimos nuevamente al error, como: $e(t) = x_d(t) - x(t)$, donde el estado deseado es $x_d(t)$ y el estado actual es $x(t)$.

En un cuadricóptero existen seis estados, tres de posición y tres de ángulos, pero solo cuatro entradas de control, la velocidad angular de los cuatro motores $\omega_i$. Las interacciones entre los estados y la fuerza total $U_1$ y los torques $U_i$ creados por los motores, son visibles desde la dinámica del cuadricóptero de la ecuación 1.26. El empuje total $U_1$ afecta la aceleración en la dirección del eje $z$ y mantiene en el aire al cuadricóptero. El torque $U_2$ tiene un efecto sobre la aceleración del ángulo $\phi$, el torque $U_3$ afecta la aceleración del ángulo $\theta$ y el torque

$U_4$ contribuye en la aceleración del ángulo $\psi$. De acuerdo al trabajo de Luukkonen [31] el controlador PD se presenta como:

$$\begin{aligned} U_1 &= \left(g + K_{z,D}\left(\dot{z}_d - \dot{z}\right) + K_{z,P}\left(z_d - z\right)\right)\frac{m}{cos\phi cos\theta} \\ U_2 &= \left(K_{\phi,D}\left(\dot{\phi}_d - \dot{\phi}\right) + K_{\phi,P}\left(\phi_d - \phi\right)\right)I_{xx} \\ U_3 &= \left(K_{\theta,D}\left(\dot{\theta}_d - \dot{\theta}\right) + K_{\theta,P}\left(\theta_d - \theta\right)\right)I_{yy} \\ U_4 &= \left(K_{\psi,D}\left(\dot{\psi}_d - \dot{\psi}\right) + K_{\psi,P}\left(\psi_d - \psi\right)\right)I_{zz} \end{aligned} \tag{3.7}$$

Donde son considerados la gravedad $g$, masa $m$ y los momentos de inercia $I$ del cuadricóptero. Los valores de las velocidades angulares de los rotores $\omega_i$ pueden ser calculados a partir de las ecuaciones 1.24 y 1.25, lo que resulta:

$$\begin{aligned} \omega_1^2 &= \frac{U_1}{4k} - \frac{U_3}{2kl} - \frac{U_4}{4b} \\ \omega_2^2 &= \frac{U_1}{4k} - \frac{U_2}{2kl} + \frac{U_4}{4b} \\ \omega_3^2 &= \frac{U_1}{4k} + \frac{U_3}{2kl} - \frac{U_4}{4b} \\ \omega_4^2 &= \frac{U_1}{4k} + \frac{U_2}{2kl} + \frac{U_4}{4b} \end{aligned} \tag{3.8}$$

El desempeño del controlador PD es probado mediante la simulación de la estabilización del cuadricóptero. Los parámetros del controlador PD son presentados en la tabla 3.1, la condición inicial para la posición es $\xi = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T$ en metros y para los ángulos $\eta = \begin{bmatrix} 10 & 10 & 10 \end{bmatrix}^T$ en grados sexagesimales. La posición deseada para la altitud es $z_d = 3$. El propósito de la estabilización es que este suspendido de forma estable, con los ángulos deseados $\eta_d = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$.

Tabla 3.1: Valores de las constantes para el controlador PD.

| Parámetro | Valor |
|---|---|
| $K_{z,D}$ | 2.5 |
| $K_{\phi,D}$ | 1.75 |
| $K_{\theta,D}$ | 1.75 |
| $K_{\psi,D}$ | 1.75 |
| $K_{z,P}$ | 1.5 |
| $K_{\phi,P}$ | 6 |
| $K_{\theta,P}$ | 6 |
| $K_{\psi,P}$ | 6 |

Fuente: T. Luukkonen. *Modelling and control of quadcopter Independent research project in applied mathematics, Espoo (2011).*

Los parámetros del controlador PD se presenta en la tabla 3.1 y valores de las constantes del sistema para la simulación se muestran en la tabla 3.2.

Tabla 3.2: Valores de los parámetros físicos del cuadricóptero para simulación.

| Parámetro | Valor | Unidad |
|---|---|---|
| $m$ | 0.468 | kg |
| $g$ | 9.81 | m/s$^2$ |
| $l$ | 0.225 | m |
| $k$ | $2.980 \cdot 10^{-6}$ | [ ] |
| $b$ | $1.140 \cdot 10^{-7}$ | [ ] |
| $I_{xx}$ | $4.856 \cdot 10^{-3}$ | kg m$^2$ |
| $I_{yy}$ | $4.856 \cdot 10^{-3}$ | kg m$^2$ |
| $I_{zz}$ | $8.801 \cdot 10^{-3}$ | kg m$^2$ |

Fuente: T. Luukkonen. *Modelling and control of quadcopter Independent research project in applied mathematics, Espoo (2011).*

Las entradas de control $\omega_i$, la posición $\xi$ y los ángulos $\eta$ de la simulación son presentadas en las figuras 3.3, 3.4, 3.5. El código para está simulación se encuentra en el Apendice A.

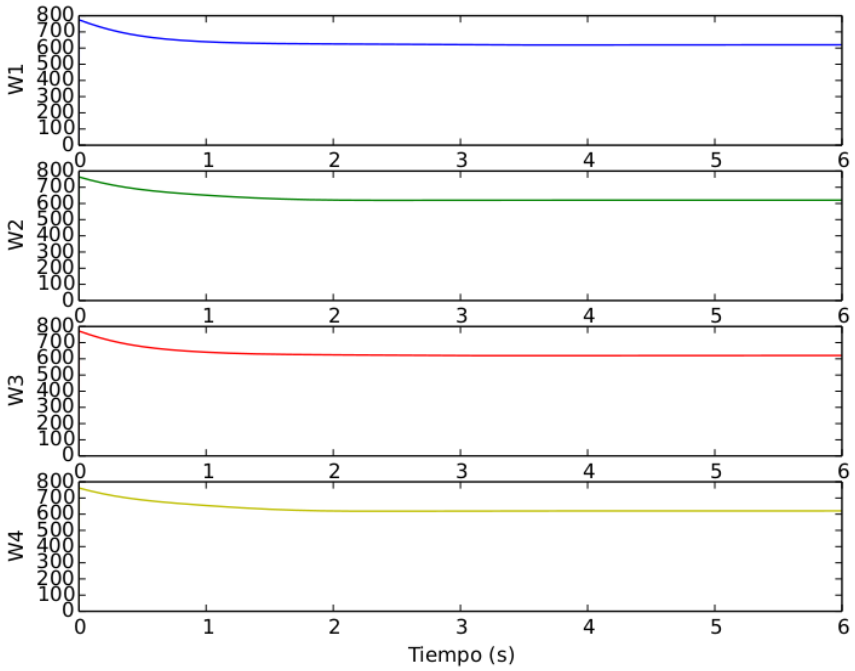



Figura 3.3. Velocidad angular de cada rotor $\omega_i$ .
Fuente: Elaboración propia.

En la figura 3.3, muestra la velocidad angular en cada motor, se puede observar que al inicio es mayor la velocidad en comparación cuando llega al estacionario, no se puede ver en la figura 3.3, la parte transitoria donde alcanza la velocidad de 800 rad/seg casi de manera instantánea.

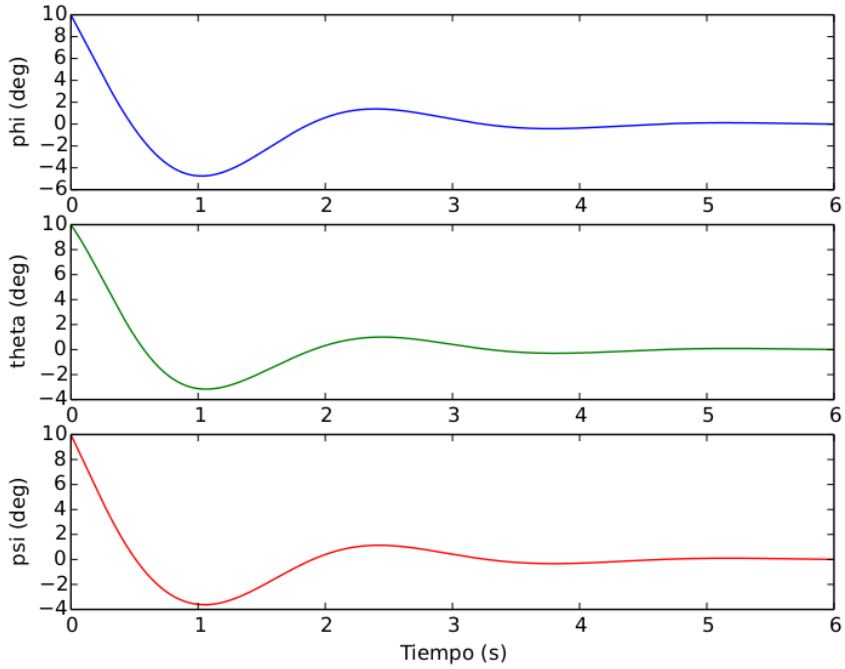



Figura 3.4. Evolución de los ángulos del cuadricóptero.
Fuente: Elaboración propia.

Como se ve en la figura 3.4, como las condiciones iniciales son diferentes de cero y los valores deseados son cero, es que mediante el control implementado llegan los ángulos a los valores deseados.

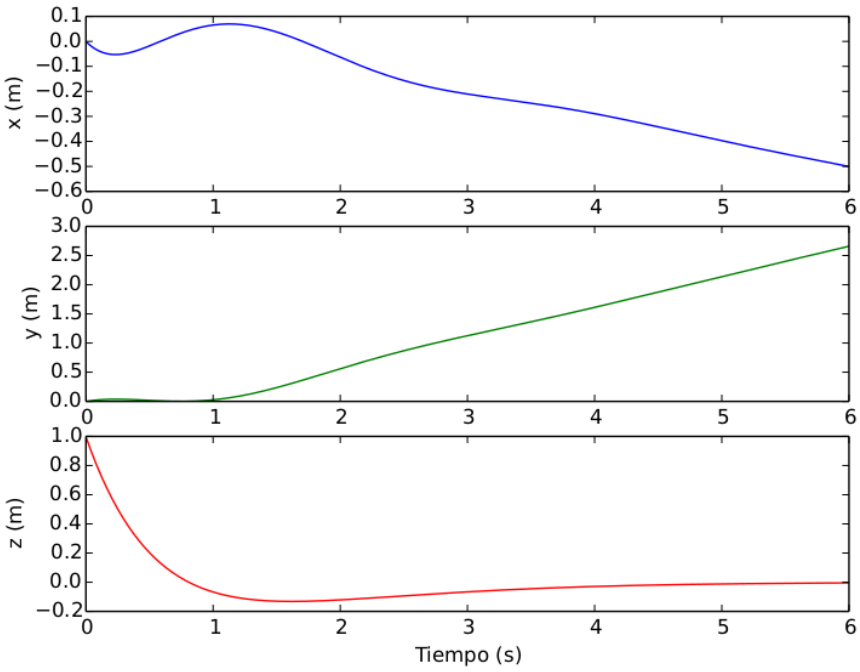



Figura 3.5. Evolución de la posición del cuadricóptero.
Fuente: Elaboración propia.

La figura 3.5 muestra la posición del cuadricóptero en los 3 ejes en función del tiempo, como el valor deseado en el eje $z$ es cero, se ve que lo logra alcanzar.

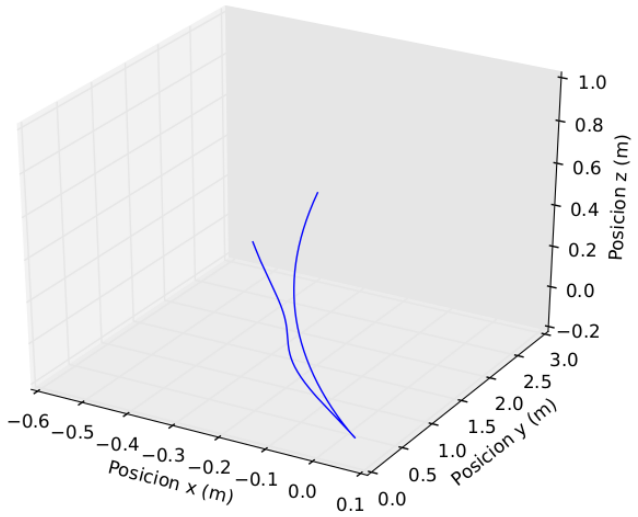



Figura 3.6. Movimiento en 3 dimensiones del cuadricóptero.
Fuente: Elaboración propia.

Y por último la figura 3.6, se visualiza mejor el movimiento en el espacio del cuadricóptero y como llega al valor deseado en el eje $z$ .

### 3.2.2.2. Control subsistema de traslación

Es importante señalar que para el control de este subsistema, no se realizará el control como el subsistema anterior, ya que para esto necesitaríamos un desarrollo más extenso y el uso de controladores más avanzados, desarrollados como en el trabajo de Raffo [25], el cual no es el objetivo de este trabajo. Para esto se va aprovechar las ventajas del `hector_quadrotor` y *ArduCopter*, debido a que una de sus características es que se pueden controlar la posición del cuadricóptero con un señales de control, realizando diferentes movimientos como: derecha-izquierda, adelante-atrás y arriba-abajo. Con esto en cuenta primero tenemos que identificar el sistema para la obtención de un modelo y posteriormente tenemos que desarrollar el algoritmo para que siga los puntos de acuerdo a los movimientos descritos.

**Identificación del subsistema de traslación**

Como se desarrolló en el capítulo 1, la dinámica del cuadricóptero es compleja si tenemos en cuenta todas las fuerzas que intervienen, por ello antes de realizar el desarrollo se hizo mención de ciertas condiciones para la simplificación del mismo y la obtención del modelo y simulación del mismo.

Como el objetivo de este trabajo no es identificar el modelo basándonos en sus ecuaciones, lo que se realizará es obtener un modelo ya estabilizado por los lazos internos de control que son propios del sistema, tanto para simulación generadas por el paquete `hector_quadrotor` implementación por el *ArduCopter*.

Para esto suponemos el sistema como una caja negra, para obtener un modelo relacionando las variables de entrada y de salida, como se observa en la figura 3.7.

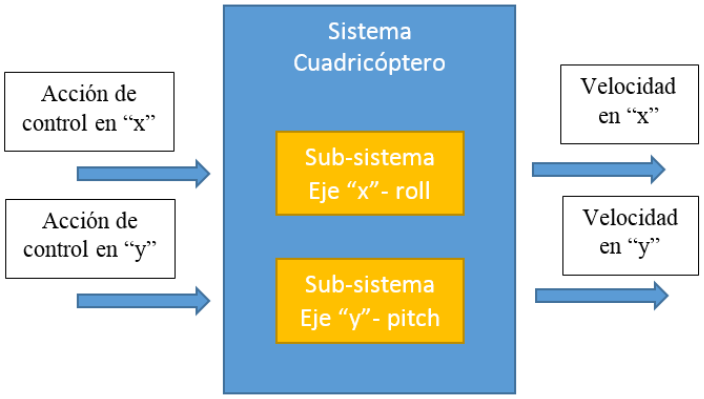



Figura 3.7. Modelo caja negra del cuadricóptero.
Fuente: Elaboración propia.

El seguimiento de trayectoria se realizará en el plano (dos dimensiones) relacionando el eje de las abscisas con la acción de control en el eje $x$ y el eje de las ordenadas con la acción de control en el eje $y$. Con esto tenemos las variables de entrada y salida identificadas.

Para la identificación, existen varios métodos disponibles, pero durante el transcurso de la maestría se utilizó con buenos resultados *System Identification Toolbox* de MATLAB, tanto para el control del nivel de un tanque de agua y control de un motor AC, en las instalaciones del laboratorio SAC. Con esta experiencia adquirida se decidió utilizar esta herramienta [32].

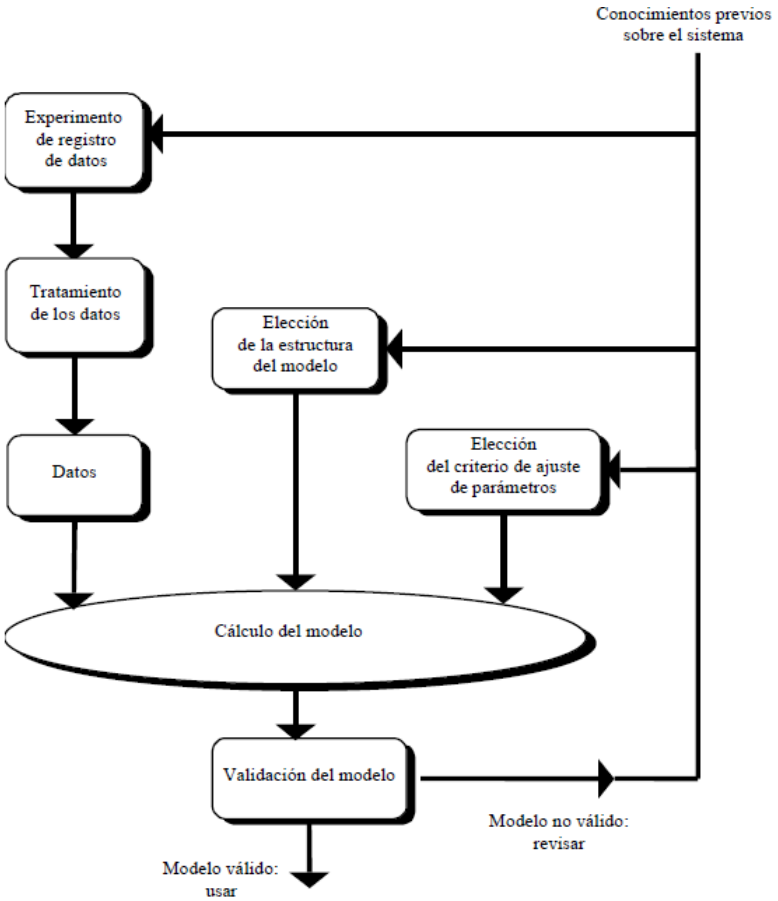



Figura 3.8. Proceso para la identificación del sistema. [33]

En la figura 3.8 podemos observar el proceso de identificación, todo parte del conocimiento del sistema para saber cual es mi variable de control y variables de salida, con esto en cuenta se ingresa una señal de control, escoger una adecuada señal de control es importante para captar mejor la dinámica de nuestro sistema [33], para esto se utilizó una señal pseudo binaria aleatoria o mejor conocida por sus siglas en inglés como *PRBS*, si se tiene un offset por ejemplo lo que se tiene que realizar un tratamiento de datos y quitar el offset, la elección de la estrutucra del modelo lo elegimos de acuerdo a las opciones del *toobox* para este caso una función de transferencia, durante el calculo del modelo el toolbox utiliza el algoritmo *transfer function estimation* [34], y para verificar que nuestro modelo es aceptable se compara con los otros datos obtenidos que se compara con los datos generados por el modelo estimado con un valor 75<FIT[27]<100 nos indica que el modelo obtenido es adecuado [35].

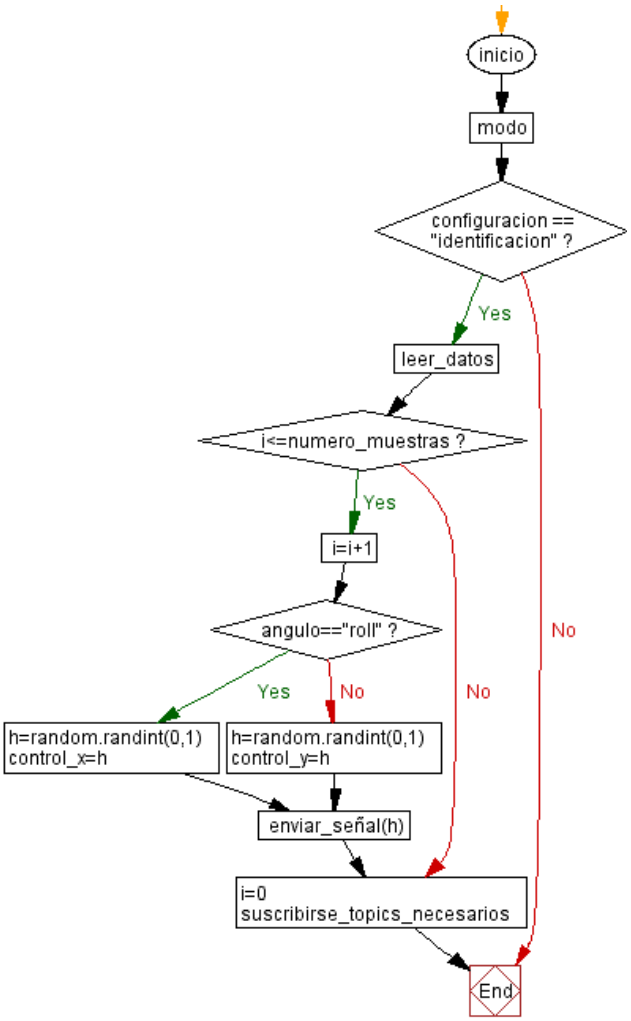



Figura 3.9. Diagrama de flujo para la creación de la señal PRBS y lectura de datos de medición.
Fuente: Elaboración propia.

Con todo lo mencionado en cuenta se crea el algoritmo para inyectar la señal de control al sistema, en la figura 3.9 se muestra el diagrama de flujo del algoritmo, en el apéndice F se muestra el código generado en *Python* utilizando *ROS* y el paquete `hector_quadrotor`.

[27] Error cuadrático medio entre los datos medidos y los datos de salida del modelo. Un 100% corresponde a un fit perfecto (no existe error) y 0% el modelo obtenido no explica las variaciones de la salida del sistema.

**Utilizando el perfil de velocidad trapezoidal para generar los waypoints**

Para realizar el seguimiento de trayectorias haciendo que el cuadricóptero vaya de un punto inicial a un punto final, generaremos varios segmentos de recta en el espacio utilizando el algoritmo de perfil de velocidad trapezoidal, como se explicó en el primer capítulo mediante la utilización de la ecuación paramétrica de la recta, se puede representar el perfil de velocidad en funciones a tramos, como se puede observar en la figura 3.10 se tiene la función y en la ecuación 3.9 la ecuación paramétrica de la función.

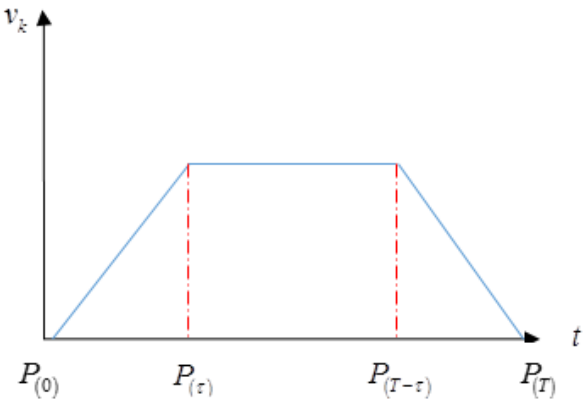



Figura 3.10. Perfil de velocidad trapezoidal.
Fuente: Elaboración propia.

$$f(t)=\begin{cases} \dfrac{1}{\left|P_{(\tau)}-P_{(0)}\right|}\left(\dfrac{1}{2}a\cdot t^2\right)\left(P_{(\tau)}-P_{(0)}\right)+P_{(0)} \rightarrow 0\le t\le \tau \\ \dfrac{v_k\cdot t}{\left|P_{(T-\tau)}-P_{(\tau)}\right|}\left(P_{(T-\tau)}-P_{(\tau)}\right)+P_{(\tau)} \rightarrow \tau < t\le T-\tau \\ \dfrac{1}{\left|P_{(T)}-P_{(T-\tau)}\right|}\left(\dfrac{1}{2}a\cdot t^2+v_k\cdot t\right)\left(P_{(T)}-P_{(T-\tau)}\right)+P_{(T-\tau)} \rightarrow T-\tau < t\le T \end{cases} \quad (3.9)$$

Donde $v_k$ es la velocidad máxima:

$$v_k=\frac{\left|P_{(T)}-P_{(0)}\right|}{T\left(1-pt\right)} \quad (3.10)$$

Donde $pt$ es el porcentaje de aceleración (valores entre 0 y 1), este porcentaje indica cuanto tiempo del total estará destinado para los tramos de aceleración y desaceleración. La aceleración está dado por:

$$\pm a=\frac{v_k}{\tau} \quad (3.11)$$

El segmento con $0 \le t \le \tau$ corresponde a la función en donde existe aceleración, el otro segmento de $\tau < t \le T - \tau$ aquí la función de velocidad es constante y el último segmento corresponde $T - \tau < t \le T$ a la función donde existe desaceleración.

Para poder solucionar el sistema se deben conocer los puntos intermedios de la trayectoria $P_{(\tau)}$ y $P_{(T-\tau)}$, para ello se puede utilizar las siguientes ecuaciones:

$$P_{(\tau)} = \frac{\left(P_{(T)} - P_{(0)}\right)}{\left|P_{(T)} - P_{(0)}\right|}\left(\frac{1}{2} a \cdot t^2\right) + P_{(0)} \tag{3.12}$$

$$P_{(T-\tau)} = \frac{v_k \cdot (T - 2\tau)}{\left|P_{(T)} - P_{(0)}\right|}\left(P_{(T)} - P_{(0)}\right) + P_{(\tau)} \tag{3.13}$$

Donde:

$\tau = T \cdot pt$

Con las ecuaciones 3.12 y 3.13, solo necesitamos conocer el punto inicial y final para empezar a despejar los diferentes valores. También se requieren el valor de velocidad máxima o el tiempo que debería demorar en ejecutar la trayectoria. Pero hay que tener en cuenta que el cuadricóptero puede tener diferentes distancias unas de las otras, con lo que se tendría que ingresar un tiempo distinto para cada trayectoria, en la práctica esto resulta inservible, por lo que se determinará como valor fijo la velocidad máxima del cuadricóptero y el porcentaje de aceleración para que a partir de estos dos valores se pueda hallar el valor de tiempo total. Se utilizará también un porcentaje de aceleración de 30 %. Con estos valores definidos garantizamos que el sistema no estará forzado y realizará cada tramo en un tiempo proporcional a la distancia a recorrer.

De esta forma, el algoritmo comenzará calculando el módulo del punto final e inicial $\left|P_{(T)} - P_{(0)}\right|$. Luego se calcula el tiempo que requiere para realizar el recorrido planeado en términos de la distancia entre ambos puntos y con este tiempo total $T$, se calcula los tiempos de aceleración y desaceleración. Lo siguiente es calcular la aceleración requerida a partir de los valores hallados. Ahora se calculan los puntos intermedios de la trayectoria $P_{(\tau)}$ y $P_{(T-\tau)}$, correspondientes a las intersecciones entre los tramos usando las ecuaciones 3.12 y 3.13. Con los puntos se hallan los módulos correspondientes para conocer las distancias que se deben recorrer en cada tramo usando la función por tramos, luego se calculan las funciones de la recta para cada tramo del recorrido. Con estos cálculos realizados ya se pueden conocer cada punto del recorrido las cuales realizará el cuadricóptero de forma precisa y brindar señales de control al sistema para realizar la trayectoria planeada.

Por ejemplo deseamos crear una trayectoria en línea recta entre los puntos $P_{(0)} = \begin{bmatrix} 1 & 2 & 2 \end{bmatrix}$ y $P_{(T)} = \begin{bmatrix} 6 & 3 & 9 \end{bmatrix}$, esta trayectoria se debe realizar en 5 segundos con características de velocidad trapezoidal, donde n=30 (n=número de puntos) y con un porcentaje de aceleración del 30%. Para esto se tiene entonces:

$$\tau = T \cdot pt = 1.5$$

$$v_k = \frac{\left|P_{(T)} - P_{(0)}\right|}{T\left(1 - pt\right)} = 2.4743$$

$$a = \frac{v_k}{\tau} = 1.6495$$

Con estas variables halladas, el procedimiento se simplifica en hallar una recta con aceleración constante entre los puntos $P_{(0)}$ y $P_{(\tau)}$. Una recta con velocidad constante entre $P_{(\tau)}$ y $P_{(T-\tau)}$ y por último la recta con desaceleración constante entre $P_{(T-\tau)}$ y $P_{(T)}$ para poder generar estas rectas se necesitan unos puntos intermedios los cuales son asignados según el porcentaje de tiempo deseado para los segmentos acelerados, si n = 30, entonces:

$$k = pt \cdot n = 9$$
$$k_C = n - 2k = 12$$

Donde:
$k$ : Número de puntos en los segmentos acelerado y desacelerado.
$k_C$ : Número de puntos en el segmento de velocidad constante.

Luego se hallan los puntos intermedios como:

Punto $P_{(\tau)}$:

$$x_{punto(\tau)} = \frac{1}{2\left|P_{(T)} - P_{(0)}\right|} a \cdot t^2 \left(P_{(T)} - P_{(0)}\right)_x + P_{(0)_x} = 2.0714$$

$$y_{punto(\tau)} = \frac{1}{2\left|P_{(T)} - P_{(0)}\right|} a \cdot t^2 \left(P_{(T)} - P_{(0)}\right)_y + P_{(0)_y} = 2.2142$$

$$z_{punto(\tau)} = \frac{1}{2\left|P_{(T)} - P_{(0)}\right|} a \cdot t^2 \left(P_{(T)} - P_{(0)}\right)_z + P_{(0)_z} = 3.5$$

$$P_{(\tau)} = \begin{bmatrix} 2.0714 & 2.2142 & 3.5 \end{bmatrix}$$

Para el punto $P_{(T-\tau)}$:

$$x_{punto(T-\tau)} = \frac{v_k\left(T - 2\tau\right)}{\left|P_{(T)} - P_{(0)}\right|}\left(P_{(T)} - P_{(0)}\right)_x + x_{punto(\tau)} = 4.9285$$

$$y_{punto(T-\tau)} = \frac{v_k\left(T - 2\tau\right)}{\left|P_{(T)} - P_{(0)}\right|}\left(P_{(T)} - P_{(0)}\right)_y + y_{punto(\tau)} = 2.7857$$

$$z_{punto(T-\tau)} = \frac{v_k\left(T - 2\tau\right)}{\left|P_{(T)} - P_{(0)}\right|}\left(P_{(T)} - P_{(0)}\right)_z + z_{punto(\tau)} = 7.5$$

$$P_{(T-\tau)} = \begin{bmatrix} 4.9285 & 2.7857 & 7.5 \end{bmatrix}$$

**Recta acelerada**

Se tiene el punto $P_{(0)}$ y el punto $P_{(\tau)}$ la velocidad inicial es 0 y la velocidad final es $v_k$, el número de puntos a utilizar serán de $k$, por tanto:

$$P_{(\tau)} - P_{(0)} = \begin{bmatrix} 1.0714 & 0.2142 & 1.5 \end{bmatrix}$$

$$T_{seg1} = \frac{2\left|P_{(\tau)} - P_{(0)}\right|}{v_k + v_0} = 1.5$$

$$a = \frac{v_k - v_0}{T_{seg1}} = 1.6495$$

Donde $T_{seg1}$ es el tiempo que se demora este segmento de recta y $a$ es la aceleración.

$$x_{seg1} = \frac{1}{2\left|P_{(\tau)} - P_{(0)}\right|}\left(a \cdot t_{k_{seg1}}^2 + v_0 t_{k_{seg1}}\right)\left(P_{(\tau)} - P_{(0)}\right)_x + P_{(0)_x}$$

$$y_{seg1} = \frac{1}{2\left|P_{(\tau)} - P_{(0)}\right|}\left(a \cdot t_{k_{seg1}}^2 + v_0 t_{k_{seg1}}\right)\left(P_{(\tau)} - P_{(0)}\right)_y + P_{(0)_y}$$

$$z_{seg1} = \frac{1}{2\left|P_{(\tau)} - P_{(0)}\right|}\left(a \cdot t_{k_{seg1}}^2 + v_0 t_{k_{seg1}}\right)\left(P_{(\tau)} - P_{(0)}\right)_z + P_{(0)_z}$$

Donde $t_{k_{seg1}}$ es un vector de tiempos espaciados $T_{seg1} \cdot \frac{1}{k}$, desde 0 hasta $T_{seg1}$.

**Recta velocidad constante**

$T_{seg2} = \frac{\left|P_{(T-\tau)} - P_{(\tau)}\right|}{v_k}$, donde $t_{k_{seg2}}$ es el valor de tiempo con valores espaciados $\frac{1}{n-2k}$ desde 0 hasta 1.

$$x_{seg2} = t_{k_{seg2}}\left(P_{(T-\tau)} - P_{(\tau)}\right)_x + P_{(0)_x}$$

$$y_{seg2} = t_{k_{seg2}}\left(P_{(T-\tau)} - P_{(\tau)}\right)_y + P_{(0)_y}$$

$$z_{seg2} = t_{k_{seg2}}\left(P_{(T-\tau)} - P_{(\tau)}\right)_z + P_{(0)_z}$$

**Recta desacelerada**

Para el tercer segmento se resuelve igual que el primer segmento, la diferencia es que el punto inicial es $P_{(T-\tau)}$ y punto final es $P_{(T)}$, con velocidad inicial $v_k$ y velocidad final 0.

$$x_{seg3} = \frac{1}{2\left|P_{(T)} - P_{(T-\tau)}\right|}\left(-a \cdot t_{k_{seg3}}^2 + v_k t_{k_{seg3}}\right)\left(P_{(T)} - P_{(T-\tau)}\right)_x + P_{(T-\tau)_x}$$

$$y_{seg3} = \frac{1}{2\left|P_{(T)} - P_{(T-\tau)}\right|}\left(-a \cdot t_{k_{seg3}}^2 + v_k t_{k_{seg3}}\right)\left(P_{(T)} - P_{(T-\tau)}\right)_y + P_{(T-\tau)_y}$$

$$z_{seg3} = \frac{1}{2\left|P_{(T)} - P_{(T-\tau)}\right|}\left(-a \cdot t_{k_{seg3}}^2 + v_k t_{k_{seg3}}\right)\left(P_{(T)} - P_{(T-\tau)}\right)_z + P_{(T-\tau)_z}$$

**Posición, velocidad y aceleración**

Todo el algoritmo fue implementado en *Python*, de la cual se obtuvo los 30 puntos necesarios como se muestra en la tabla 3.3 para realizar la trayectoria, además es necesario conocer a qué velocidad debería estar el cuadricóptero en cada punto hallado, entonces para ello, en el mismo algoritmo creamos una rutina para conocer las velocidades en cada punto.

Tabla 3.3. Con n=30, los puntos generados son 30, muestra las posiciones que deben alcanzar.

| | Tiempo | x | y | z |
|---|---|---|---|---|
| **Punto 1** | 0 | 1 | 2 | 2 |
| **Punto 2** | 0.16666667 | 1.01322751 | 2.0026455 | 2.01851852 |
| **Punto 3** | 0.33333333 | 1.05291005 | 2.01058201 | 2.07407407 |
| **Punto 4** | 0.5 | 1.11904762 | 2.02380952 | 2.16666667 |
| **Punto 5** | 0.66666667 | 1.21164021 | 2.04232804 | 2.2962963 |
| **Punto 6** | 0.83333333 | 1.33068783 | 2.06613757 | 2.46296296 |
| **Punto 7** | 1 | 1.47619048 | 2.0952381 | 2.66666667 |
| **Punto 8** | 1.16666667 | 1.64814815 | 2.12962963 | 2.90740741 |
| **Punto 9** | 1.33333333 | 1.84656085 | 2.16931217 | 3.18518519 |
| **Punto 10** | 1.5 | 2.07142857 | 2.21428571 | 3.5 |
| **Punto 11** | 1.68181818 | 2.33116883 | 2.26623377 | 3.86363636 |
| **Punto 12** | 1.86363636 | 2.59090909 | 2.31818182 | 4.22727273 |
| **Punto 13** | 2.04545455 | 2.85064935 | 2.37012987 | 4.59090909 |
| **Punto 14** | 2.22727273 | 3.11038961 | 2.42207792 | 4.95454545 |
| **Punto 15** | 2.40909091 | 3.37012987 | 2.47402597 | 5.31818182 |
| **Punto 16** | 2.59090909 | 3.62987013 | 2.52597403 | 5.68181818 |
| **Punto 17** | 2.77272727 | 3.88961039 | 2.57792208 | 6.04545455 |
| **Punto 18** | 2.95454545 | 4.14935065 | 2.62987013 | 6.40909091 |
| **Punto 19** | 3.13636364 | 4.40909091 | 2.68181818 | 6.77272727 |
| **Punto 20** | 3.31818182 | 4.66883117 | 2.73376623 | 7.13636364 |
| **Punto 21** | 3.5 | 4.92857143 | 2.78571429 | 7.5 |
| **Punto 22** | 3.66666667 | 5.15343915 | 2.83068783 | 7.81481481 |
| **Punto 23** | 3.83333333 | 5.35185185 | 2.87037037 | 8.09259259 |
| **Punto 24** | 4 | 5.52380952 | 2.9047619 | 8.33333333 |
| **Punto 25** | 4.16666667 | 5.66931217 | 2.93386243 | 8.53703704 |
| **Punto 26** | 4.33333333 | 5.78835979 | 2.95767196 | 8.7037037 |
| **Punto 27** | 4.5 | 5.88095238 | 2.97619048 | 8.83333333 |
| **Punto 28** | 4.66666667 | 5.94708995 | 2.98941799 | 8.92592593 |
| **Punto 29** | 4.83333333 | 5.98677249 | 2.9973545 | 8.98148148 |
| **Punto 30** | 5 | 6 | 3 | 9 |

Fuente: Elaboración propia.

Las velocidades se obtienen, mediante la utilización de métodos numéricos, es decir conocidos los vectores de posición y tiempo se realiza la derivada de los puntos con respecto al tiempo que existe entre cada punto oteniendose la tabla 3.4, al crear esta función podemos

utilizarla de nuevo para hallar las aceleraciones de la misma forma, para ello tomamos el vector de velocidad de cada eje y lo ingresamos en la función derivada creada con lo que se obtiene el vector de aceleraciones.

Tabla 3.4. Con n=30, los puntos generados son 30, muestra las velocidades que deben alcanzar.

| | Vx | Vy | Vz | Vt |
|---|---|---|---|---|
| **Punto 1** | 0 | 0 | 0 | 0 |
| **Punto 2** | 0.07936508 | 0.01587302 | 0.11111111 | 0.13746435 |
| **Punto 3** | 0.23809524 | 0.04761905 | 0.33333333 | 0.41239305 |
| **Punto 4** | 0.3968254 | 0.07936508 | 0.55555556 | 0.68732175 |
| **Punto 5** | 0.55555556 | 0.11111111 | 0.77777778 | 0.96225045 |
| **Punto 6** | 0.71428571 | 0.14285714 | 1 | 1.23717915 |
| **Punto 7** | 0.87301587 | 0.17460317 | 1.22222222 | 1.51210785 |
| **Punto 8** | 1.03174603 | 0.20634921 | 1.44444444 | 1.78703655 |
| **Punto 9** | 1.19047619 | 0.23809524 | 1.66666667 | 2.06196525 |
| **Punto 10** | 1.34920635 | 0.26984127 | 1.88888889 | 2.33689395 |
| **Punto 11** | 1.42857143 | 0.28571429 | 2 | 2.4743583 |
| **Punto 12** | 1.42857143 | 0.28571429 | 2 | 2.4743583 |
| **Punto 13** | 1.42857143 | 0.28571429 | 2 | 2.4743583 |
| **Punto 14** | 1.42857143 | 0.28571429 | 2 | 2.4743583 |
| **Punto 15** | 1.42857143 | 0.28571429 | 2 | 2.4743583 |
| **Punto 16** | 1.42857143 | 0.28571429 | 2 | 2.4743583 |
| **Punto 17** | 1.42857143 | 0.28571429 | 2 | 2.4743583 |
| **Punto 18** | 1.42857143 | 0.28571429 | 2 | 2.4743583 |
| **Punto 19** | 1.42857143 | 0.28571429 | 2 | 2.4743583 |
| **Punto 20** | 1.42857143 | 0.28571429 | 2 | 2.4743583 |
| **Punto 21** | 1.42857143 | 0.28571429 | 2 | 2.4743583 |
| **Punto 22** | 1.34920635 | 0.26984127 | 1.88888889 | 2.33689395 |
| **Punto 23** | 1.19047619 | 0.23809524 | 1.66666667 | 2.06196525 |
| **Punto 24** | 1.03174603 | 0.20634921 | 1.44444444 | 1.78703655 |
| **Punto 25** | 0.87301587 | 0.17460317 | 1.22222222 | 1.51210785 |
| **Punto 26** | 0.71428571 | 0.14285714 | 1 | 1.23717915 |
| **Punto 27** | 0.55555556 | 0.11111111 | 0.77777778 | 0.96225045 |
| **Punto 28** | 0.3968254 | 0.07936508 | 0.55555556 | 0.68732175 |
| **Punto 29** | 0.23809524 | 0.04761905 | 0.33333333 | 0.41239305 |
| **Punto 30** | 0.07936508 | 0.01587302 | 0.11111111 | 0.13746435 |

Fuente: Elaboración propia.

Como se puede observar en la tabla 3.4, vemos como la velocidad se mantiene constante desde el punto 11 hasta el punto 21 y en los extremos tenemos aceleraciones constantes, pero para observar bien como están distribuidos estos puntos en el espacio tenemos la figura 3.11.

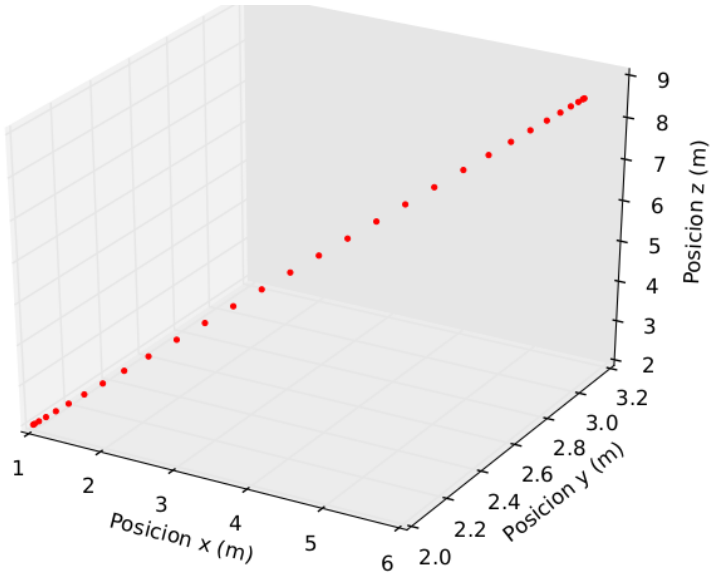



Figura 3.11. Muestra las posiciones que deben alcanzar.
Fuente: Elaboración propia.

Como se puede observar en la figura 3.11, los extremos muestran los puntos más juntos que el tramo del medio, esto se debe a que se tiene más puntos en el medio a velocidad constante (como la velocidad es constante la distancia recorrida es igual para tiempos iguales) cosa que no sucede en los extremos donde las velocidades cambian.

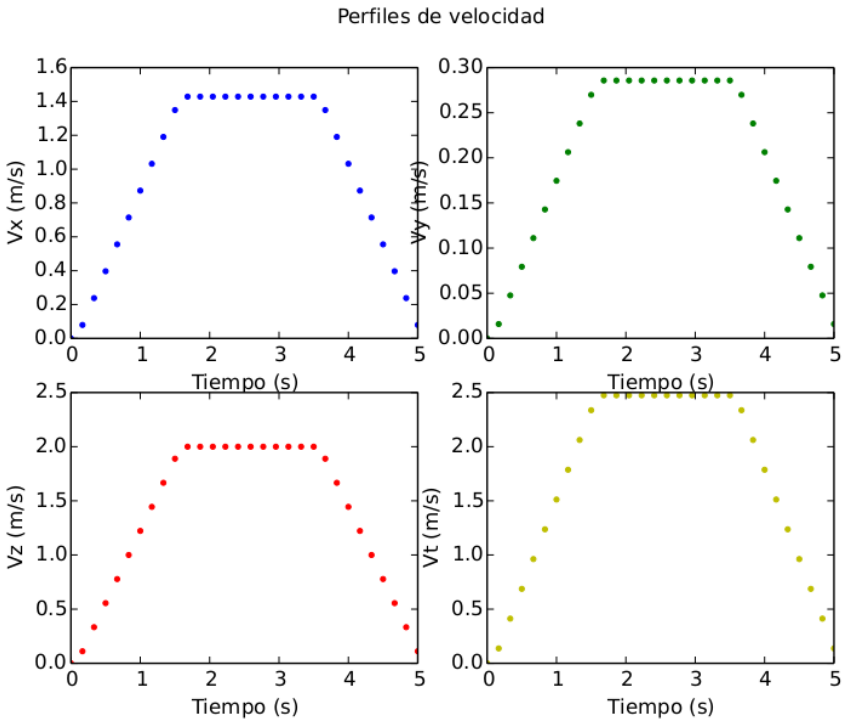



Figura 3.12. Los puntos generados son 30, muestra las velocidades que deben alcanzar.
Fuente: Elaboración propia.

La figura 3.12, muestra claramente el perfil de velocidad trapezoidal generado, también se tiene la velocidad total (módulo de los tres ejes). De igual forma se realiza el mismo procedimiento si solo fuese el movimiento en dos dimensiones, es decir, en los ejes $x$ y $y$. En este trabajo se utilizará esta última debido a que la altura se controlará de forma manual e independiente.

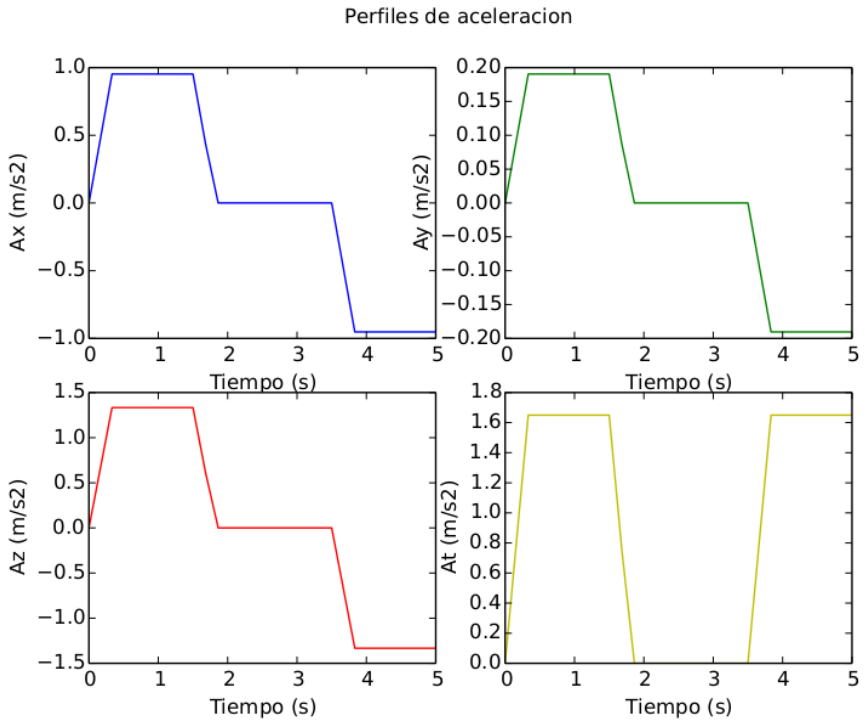



Figura 3.13. Los puntos generados son 30, muestra las aceleraciones en cada eje y resultante.
Fuente: Elaboración propia.

De la figura 3.13, las aceleraciones como podemos deducir previamente será 0 en donde la velocidad es constante (segmento del medio) y tendrán variaciones (diferente de cero) en donde las velocidades varían (segmentos de los extremos).

### 3.2.3. Métodos utilizados para el perfil de velocidad trapezoidal

Una vez realizada el cálculo de las variables necesarias para el algoritmo de perfil de velocidad trapezoidal, lo siguiente es realizar la medición de la posición y el control para ejecutar la trayectoria en base a las variables calculadas en el algoritmo.

Para está etapa existen dos métodos para ejecutar el algoritmo de perfil de velocidad trapezoidal, los cuales son:

- Método de movimiento en una dirección.
- Método de movimiento en dos direcciones.

En lo siguiente se describirá cada uno de estos métodos y cuáles son sus ventajas y desventajas, en el Apéndice B se tiene el diagrama de flujo para ambos métodos a implementar utilizados para simulación y las pruebas de campo.

### 3.2.3.1. Método de movimiento en dos direcciones

Se dice en una dirección, porque la dirección del movimiento de traslación se ejecuta en un solo eje, en este caso hacia adelante como se observa en la figura 3.14.

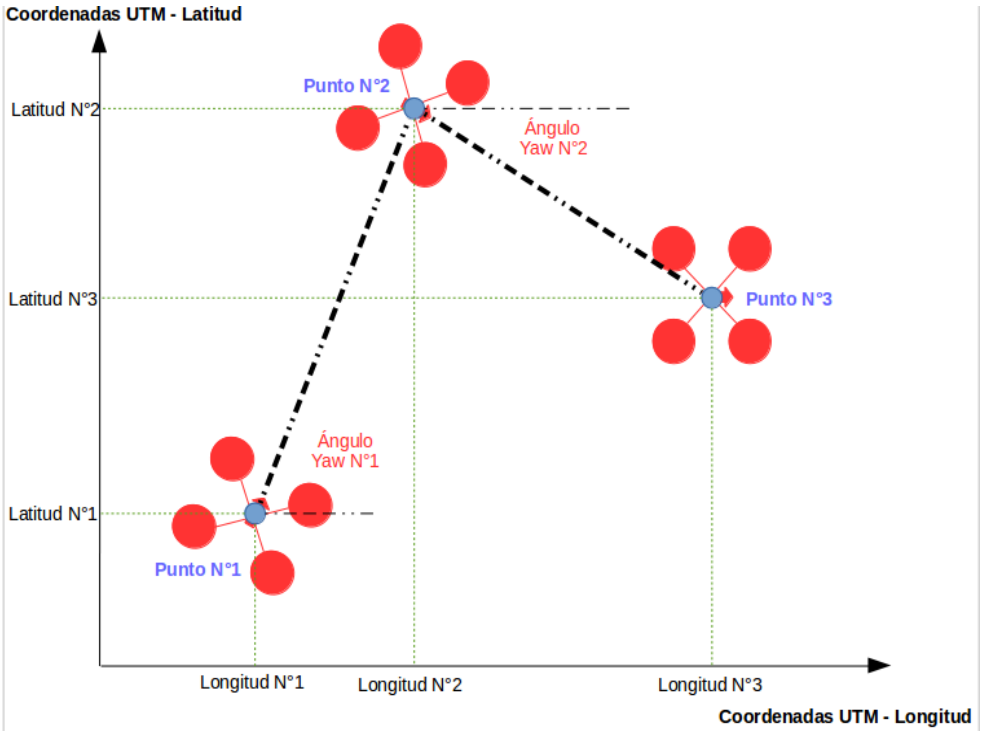



Figura 3.14. Método de movimiento en una dirección.
Fuente: Elaboración propia.

La secuencia que debe seguir este método es el siguiente:

1) Encontrar los ángulos de giro en el eje $z$, estos se hallan como el ángulo entre dos puntos de la trayectoria y un eje de referencia, para este caso el eje $x$ será nuestro eje de referencia.
2) El cuadricóptero siempre debe estar orientado hacia el siguiente punto (mirar hacia adelante), para esto gira un ángulo *yaw* anteriormente calculado.
3) El único movimiento que debe realizar el cuadricóptero es hacia adelante, es decir en un sólo eje, hasta llegar al siguiente punto, de acuerdo a los valores hallados en el algoritmo de perfil de velocidad trapezoidal.
4) El control PID está en el ángulo *yaw*, para que no se desvíe del ángulo *yaw* calculado en la primera parte.
5) Terminado de recorrer el tramo, se detiene el cuadricóptero y toma el siguiente ángulo *yaw* calculado, se reinician los valores y se repite el mismo procedimiento hasta completar todos los puntos de la trayectoria.

Un punto importante es que se tiene el control PID sólo en el ángulo *yaw*, pero no en los ejes de movimiento $x$ y $y$.

### 3.2.3.2. Método de movimiento en dos direcciones

Es semejante al método anterior, la diferencia es que el ángulo *yaw* siempre debe estar en 0° es decir el norte terrestre debe coincidir con el norte del cuadricóptero y la dirección de movimiento se da en dos ejes como se muestra en la figura 3.15.

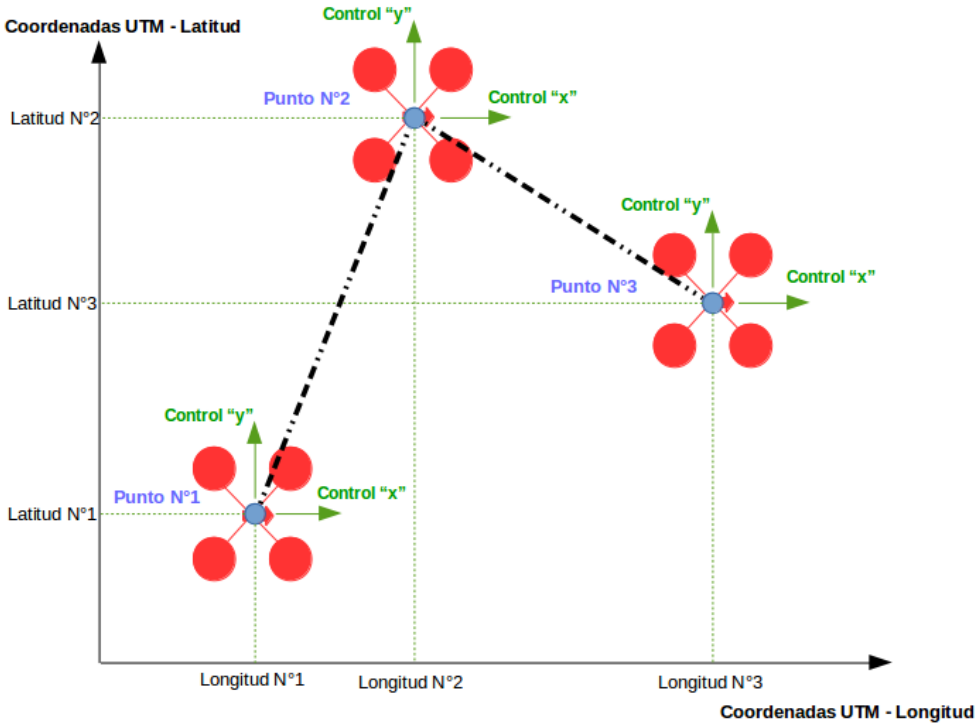



Figura 3.15. Método de movimiento en dos direcciones.
Fuente: Elaboración propia.

Para su ejecución se deben seguir los siguientes pasos:

1) Un control PID en el ángulo *yaw* hace del cuadricóptero quede orientado 0° permanentemente.
2) Teniendo los valores del perfil de velocidad trapezoidal para ambos ejes $x$ y $y$ con los valores calculados, se procede a recorrer los puntos en los dos ejes $x$ y $y$.
3) Se implementa un control PID para ambos ejes, de tal manera se con-sigue un mejor desempeño, la acción de control además de estar en el ángulo *yaw* también estará en los ejes $x$ y $y$.
4) Terminado de recorrer el primer tramo, se reinicia todas las variables y se procede con el siguiente tramo hasta completar la trayectoria.

Con esto ya estamos en la capacidad de poder implementar nuestros algoritmos en el entorno de *ROS*, pero primero debemos crear nuestro *workspace* y demás configuraciones, esto se verá con mayor detalle en el siguiente capítulo.

# Capítulo 4

# Pruebas y resultados

## 4.1. Arquitectura del sistema

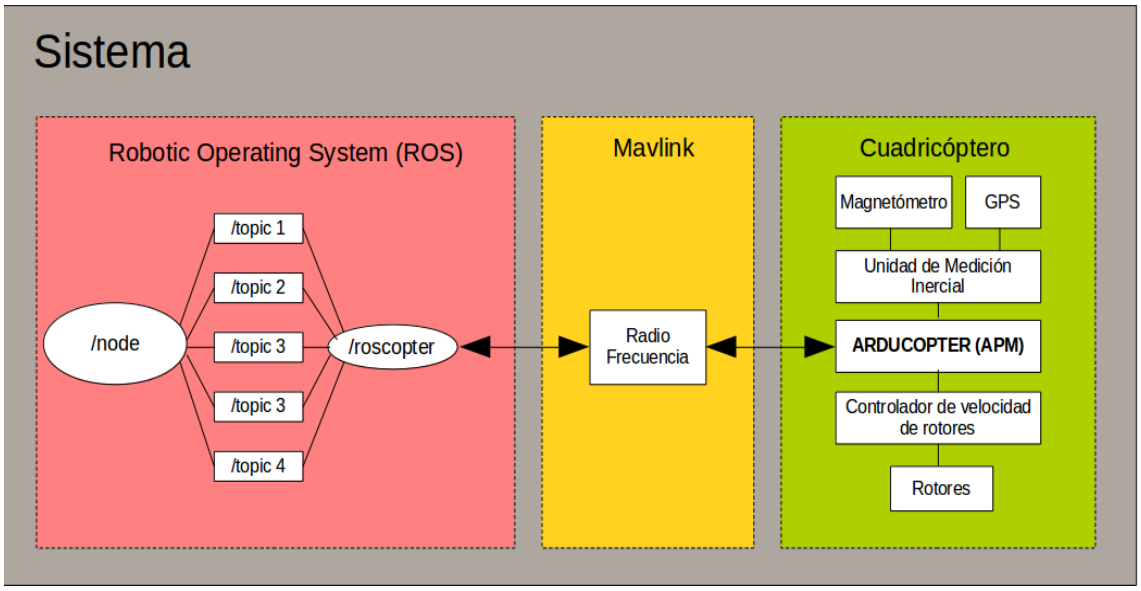



Figura 4.1. Arquitectura del Sistema.
Fuente: Elaboración propia.

Se puede observar en la figura 4.1 la arquitectura general del sistema diseñado e implementado, en diagrama de bloques.

El primer bloque se muestra la organización en *ROS*, donde los rectángulos son los *topics* y los elipsoides son los *nodes*. Estos *nodes* procesan los datos de los *topics*, se tiene un *node* para el control y otro *node* llamado `roscopter`, este último *node* se encarga de leer las señales de datos del *ArduCopter* y enviar las señales de control calculados en el *node* de control.

El bloque central es el medio donde se transmite y recibe la información, para este caso se utiliza módulos de radiofrecuencia (RF) utilizando el protocolo usado para los VANTs denominado *MAVlink*[28] [36].

El tercer bloque tenemos el *ArduCopter*, el cual consta del sistema embebido principal denominado *Ardupilot*, magnetómetro[29], *GPS*, *IMU*[30] [37] la cual recibe la información del magnetómetro y por último los controladores *ESC*[31] que a través de 4 señales *PWM*[32] del *ArduCopter* controla la velocidad de los 4 motores, los cuales en conjunto conforman nuestro cuadricóptero [38].

Con la arquitectura del sistema descrita, ahora nos enfocaremos en crear el código para el funcionamiento de nuestro sistema de control. Se desarrollaron tanto para un entorno de simulación y otro entorno real, siendo las diferencias entre ambos sólo en la suscripción y publicación en los *topics* como se explicará posteriormente.

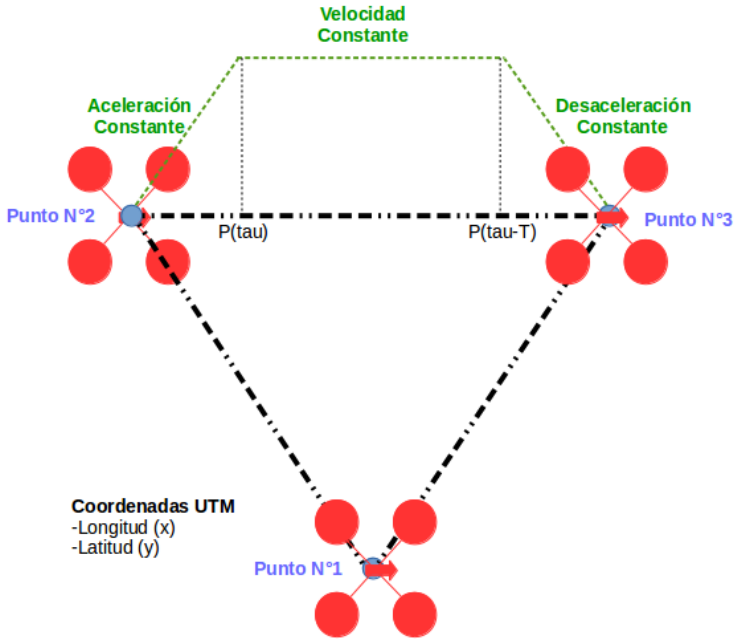



Figura 4.2. Descripción gráfica sobre la ejecución del algoritmo de control.
Fuente: Elaboración propia.

[28] *Micro Air Vehicle Link*, es un protocolo de comunicación para pequeños vehículos no tripulados.
[29] Instrumento para medir la fuerza y dirección de un campo magnético.
[30] *Inertial Measurement Unit*, es un dispositivo electrónico que mide la velocidad, aceleración y fuerzas gravitacionales de un aparato.
[31] *Electronic Speed Controller*, dispositivo electrónico que sirve para controlar la velocidad del motor *brushless*.
[32] *Pulse Width Modulation*, es un tipo de señal de voltaje utilizado para enviar información o para modificar la cantidad de energía que se envía a una carga.

Con la base conceptual desarrollada en los capítulos anteriores nuestro algoritmo de control seguirá la trayectoria deseada, tal y como se muestra en la figura 4.2

## 4.2. Programación y desarrollo

Hoy en día gracias al avance tanto en software como en hardware de las computadoras, es posible mediante diversas herramientas desarrolladas (*toolboxs*) modelar, simular y evaluar diferentes sistemas robóticos en cualquier entorno o ambiente que deseamos, esto resulta de gran utilidad porque nos permite evaluar nuestro sistema robótico sin necesidad de implementarlo físicamente, mejorarlo y sobre todo en un entorno virtual no tenemos tantas limitaciones como en un entorno real (sistema robótico implementado). Por eso que en la actualidad para todo sistema robótico es un hecho que primero tuvo que pasar por muchas pruebas en simulaciones para su posterior construcción e implementación.

Para el desarrollo del sistema de control, primero todos los desarrollos y pruebas se hicieron en entornos de simulación, una vez realizadas estás pruebas de manera satisfactoria se pasó a su implementación para un entorno real, en esta parte describiremos cada una de ellas pero primero se describirá las herramientas utilizadas.

### 4.2.1. Software utilizado

Debido a que las herramientas disponibles para poder desarrollar nuestro sistema de control para el cuadricóptero eran demasiadas (desde las más simples herramientas hasta las más complejas), se tuvo que elegir las herramientas de acuerdo a 2 criterios principales:

- Software libre, para no tener una restricción en licencia o versión de prueba.
- Sencillo y que pueda manejar diversos lenguajes de programación y tenga buen desempeño a la hora de realizar las simulaciones.

Como ya se mencionó anteriormente la utilización de *ROS*, su mejor desempeño y desarrollo se encuentra sobre Linux en su distribución Ubuntu, claro que existen versiones en desarrollo o prueba para Windows pero *ROS* se desarrolló inicialmente desde Linux. *ROS* además de las ventajas mencionadas en los capítulos anteriores, también se puede integrar con otros simuladores de manera sencilla, tal es el caso de Gazebo. *ROS* de igual forma cuenta con un visualizador de sensores como lo es *ROS Visualization* o *RViz*, ambas herramientas permitirá a nuestro sistema robótico interactuar en un entorno creado de forma virtual.

#### 4.2.1.1. Software Gazebo y *RViz*

**Gazebo**

La simulación de todo robot es una herramienta esencial en cada software robótico. Un buen simulador diseñado hace posible una evaluación rápida de los algoritmos, diseño de robots y desempeño usando ambientes realistas. Gazebo ofrece esta capacidad de simular de forma precisa y eficiente grupos de robots en complejos ambientes tanto en interiores como en exteriores.

Gazebo fue desarrollado al final del 2002 en la Universidad de California del Sur. El creador original fue el Dr. Andrew Howard y su estudiante Nate Koenig. El concepto de un

simulador de alta fidelidad nació de la necesidad de simular robots en ambientes abiertos (*outdoor*) bajo varias condiciones [39].

A través de los años, Nate Koenig continuo desarrollando Gazebo mientras completaba su PhD. en 2009, John Hsu, un ingeniero investigador de Willow Garage, integro *ROS* y el PR2 como se observa en la figura 4.2 dentro de Gazebo, lo que se convirtió en una de las primeras herramientas usadas en la comunidad de *ROS*. Unos pocos años después en 2011, Willow Garage empezó el apoyo financiero para el desarrollo de Gazebo [39].


Figura 4.3. Entorno del simulador GAZEBO, robot PR2.
Fuente: http://answers.ros.org

En un principio Gazebo fue pensado para trabajar solo en *ROS*, pero dado a su gran capacidad, fácil integración y bajo la filosofía de software libre, es que se viene desarrollando de manera independiente. En versiones iniciales de *ROS*, el paquete de instalación de Gazebo ya venía incluido, pero en las últimas versiones de *ROS* ya no es así, por eso es que se tiene que descargar desde la página web de Gazebo.

A nuestro alcance tenemos a Gazebo como un robusto motor físico, gráficos de alta calidad y cómodas interfaces gráficas. Lo mejor de todo esto, es que Gazebo es software libre con una comunidad siempre aportando y mejorando este simulador.

***ROS Visualization* (*RViz*)**

*RViz* significa visualización de *ROS*. Se trata de un entorno de visualización 3D de uso general para robots, sensores y algoritmos. Como la mayoría de las herramientas de *ROS*, estas pueden ser usados para cualquier robot y rápidamente configurado para una aplicación en particular. *RViz* puede representar una variedad de tipos de datos que fluyen a través de un sistema *ROS* típico, principalmente en la naturaleza tridimensional de los datos. En *ROS*, todas las formas de datos están unidas a un marco de referencia en la que se tomaron los datos [40].

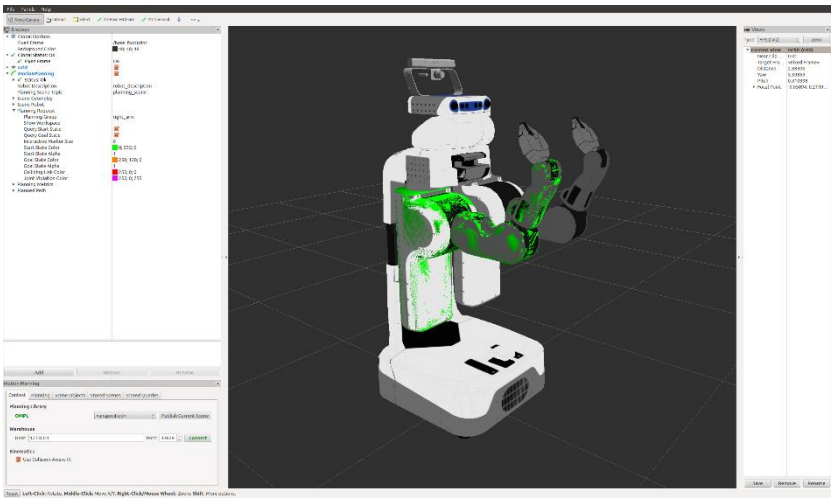


Figura 4.4. Entorno grafico *ROS Visualization* (*RViz*), robot PR2.
Fuente: http://answers.ros.org

Al igual que muchas complejas interfaces gráficas de usuario (*GUIs*), *RViz* tiene varios paneles y complementos que se pueden configurar según sea necesario para una tarea determinada como se muestra en la figura 4.4. Configurar *RViz* puede tomar algo de tiempo y esfuerzo, por lo que el estado de visualización puede ser guardado en la configuración de archivos para su posterior reutilización. Adicionalmente, al cerrar *RViz*, por defecto el programa guardará esta configuración en un archivo local especial, de modo que la próxima vez que *RViz* sea ejecutado instanciará y configurará los mismos paneles y *plugins*[33].

#### 4.2.1.2. Lenguaje de programación *Python*

Este lenguaje de programación fue creado por Guido Van Rossum a principios de 1990 en Holanda. Básicamente *Python* es un intérprete de instrucciones que permite usar el lenguaje en forma interactiva.

Los lenguajes interpretados, a diferencia de los lenguajes compilados, permiten experimentar interactivamente en una ventana y también mediante programas que pueden desarrollarse y probarse a medida que son implementados, esta interacción facilita el aprendizaje del lenguaje de programación ya que permite analizar el código mientras se ejecuta y ubicar posibles errores o la de optimizar el algoritmo. Los programas compilados en cambio, deben estar completos para que sean probados y no admiten experimentar separadamente con las instrucciones. La principal ventaja de los programas compilados es que el tiempo de ejecución es mucho menor comparado con los desarrollados en lenguajes interpretados [23]. En la figura 4.5 se puede observar la declaración de funciones en *Python* asi como lo fácil que puede ser compresnsión frente a otros lenguajes de programación.

[33] Es una aplicación que se relaciona con otra para agregarle una nueva función y generalmente muy específica.

```
# -----Algoritmo para el calculo de los angulos para el
def calculo_angulos(vector_Lon,vector_Lat):
    angulos=[]
    n=len(vector_lon)
    for j in range(n-1):
        x=vector_lon[j+1]-vector_lon[j]
        y=vector_lat[j+1]-vector_lat[j]
        ang=math.atan2(y,x)
        ang_degree=math.degrees(ang)
        angulos.append(ang_degree)
    return angulos

#--------------------Algoritmo de Control--------------
def control(k_xyz,k_yaw,error,Last_error,Last_ui):
    u=[];   ui=[]
    for i in range(4):
        if i==3:
            uP= k_yaw[0]*error[i]   # Para el control d
            uI= k_yaw[1]*error[i]+last_ui[i]
            uD= k_yaw[2]*(error[i]-last_error[i])
            ut= uP+uI+uD
        else:
            uP= k_xyz[0]*error[i]   # Para el control e
            uI= k_xyz[1]*error[i]+last_ui[i]
            uD= k_xyz[2]*(error[i]-last_error[i])
            ut= uP+uI+uD
        u.append(ut)
        ui.append(uI)
    return u,ui

#--------------------Algoritmo conversion de coordenada
def geodesicas_to_utm(Lon,Lat): # Conversion de coorden
    lon_rad=lon*math.pi/180
    lat_rad=lat*math.pi/180
    huso=int(lon/6+31)          # Calculo del huso, so
```

Figura 4.5. Programa en código *Python*.
Fuente: Elaboración propia

*Python* es un lenguaje de propósito general, es decir, su diseño no está destinado a obligar a los programadores a tener solo un estilo, esta ventaja del lenguaje propicia la creatividad del programador y permite la versatilidad entre varias metodologías de programación. Con todos los recursos del lenguaje y el soporte de las librerías disponibles, el programador puede explayarse para crear nuevas soluciones, partiendo de un conocimiento inicial básico, puede avanzar en el aprendizaje de *Python* a su propio ritmo. Lo mejor es que es un producto público de distribución libre y disponible en internet.

#### 4.2.1.3. *Heterogeneous Cooperating Team of Robots – HECTOR*

El *Team HECTOR*, empezó en 2009 como un esfuerzo interdisciplinario de investigación por la *Technishe Universitat Darmstad* (Alemania) de los departamentos de ciencias de la computación e ingeniería mecánica dentro del programa PhD GRK 1362. Los miembros de este grupo son estudiantes de post-doctorados del departamento de ciencias de la computación y de maestrías en sistemas autónomos, ciencias de la computación, ingeniería computacional y tecnologías de sistemas de información [41].


Figura 4.6. *Team HECTOR* ganador de la *ROBOCUP* 2016 [41].

El objetivo de este grupo es la investigación y desarrollo en cooperación con cualquier otro grupo de investigación en temas sobre la supervisión humana a distancia para lograr una misión en común.

Después de varios años de exitosa investigación y desarrollo en robótica terrestre y aérea de búsqueda y rescate, el grupo *HECTOR* empezó recientemente a investigar y desarrollar habilidades avanzadas en locomoción y manipulación para robots humanoides frente a desastres tal y como lo demostraron al ganar sus integrantes la *ROBOCUP* 2016 como se muestra en la figura 4.6.

*HECTOR* ha desarrollado varias colecciones de pilas (*stacks*) en *ROS*, estos *stacks* proporcionan varias herramientas para simular o interactuar con robots. De esta colección se utilizará el *stack* `hector_quadrotor`, que será capaz de simular un cuadricóptero con parámetros físicos ya determinados, utilizando Gazebo como se muestra en la figura 4.7 o *RViz*, para evaluar el desempaño del controlador a desarrollar.

Figura 4.7. Paquete `hector_quadrotor`, en un ambiente abierto utilizando Gazebo.
Fuente: Elaboración propia.

### 4.2.2. Desarrollo en entorno virtual

Para tener una idea como se realizará la simulación del sistema de control, se presenta la arquitectura de simulación en la figura 4.8.

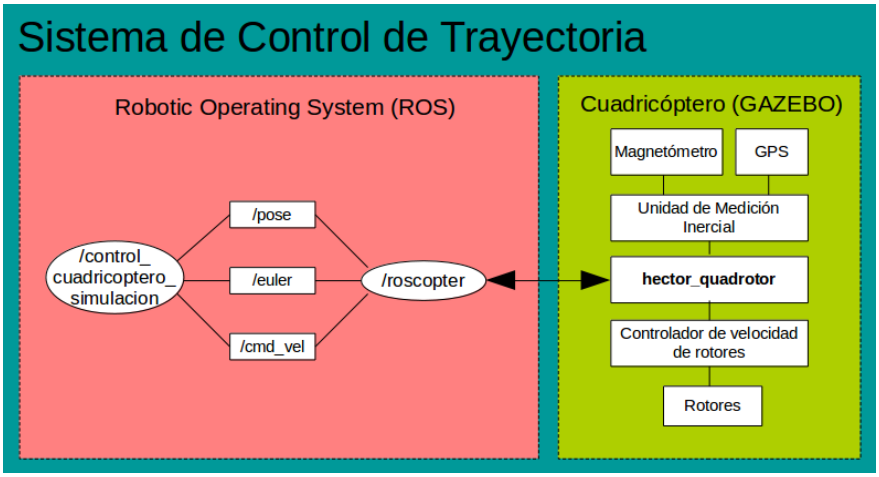



Figura 4.8. Diagrama de bloques del sistema de control para simulación.
Fuente: Elaboración propia.

Al igual que la arquitectura del sistema presentada al inicio, en esta sección no se considera *MAVlink* debido a que eso se realiza para conexiones físicas, pero para simulaciones no es necesario. Ahora procedemos a describir como crear un entorno de trabajo, tanto para la simulación como la implementación se creará un mismo *stack* con diferentes *packages* donde albergarán los *package* para simulación, implementación y `roscopter`.

#### 4.2.2.1. Creación de un entorno de trabajo en *ROS*

Para el desarrollo del sistema de control, una vez familiarizado con el entorno de *ROS* y Ubuntu procedemos a crear nuestro propio espacio de trabajo (*workspace*) donde podremos generar, guardar y compilar nuestros códigos para el sistema de control.

**Creación de packages**

Crearemos un *package* dentro de nuestro *workspace* `ros_workspace`, esto nos permitirá organizar de forma adecuada nuestros archivos generados.

**Creación de nodes**

El *node* generado fue `control_simulacion` y necesitará leer y escribir sobre los *topics* creados por el *package* `hector_quadrotor`.

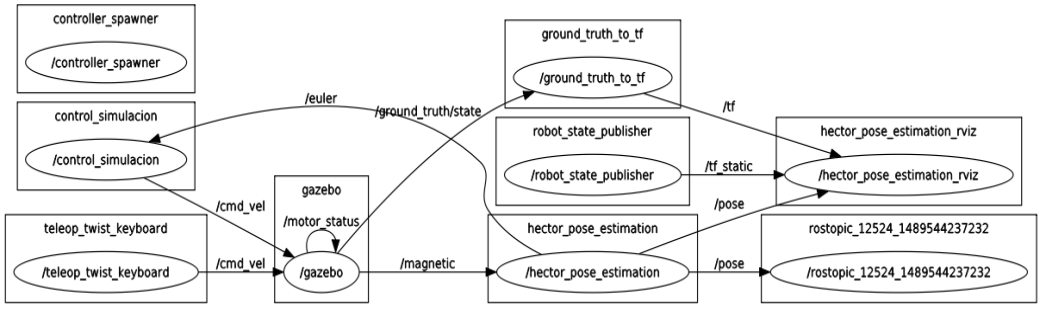



Figura 4.9. Visualización de los *topics* y *nodes* ejecutados de `hector_quadrotor`, mediante la utilización del comando `rqt_graph`.
Fuente: Elaboración propia.

En la gráfica 4.9 se muestra todos los *nodes* y *topics* utilizados para simular el cuadricóptero con el sistema de control implementado, cabe mencionar que los *nodes* son los elipsoides y los *topics* son los rectángulos.

#### 4.2.2.2. Suscripción y publicación en topics necesarios

Utilizaremos otros *topics* ya creados por el *package* `hector_quadrotor`, tales como `pose`, `euler` y `cmd_vel`, explicaremos a continuación que datos contienen estos *topics*.

- `pose`: Contiene los datos de posición del cuadricóptero. Para la implementación en nuestro cuadricóptero cambiaremos este *topic* por los datos enviados por el GPS.

- `euler`: Contiene los datos de ángulos en radianes del cuadricóptero. Tener en cuenta que el *topic* `pose` también contiene los ángulos pero en forma de cuaterniones[34].
- `cmd_vel`: Contiene los comandos de control que se envía al cuadricóptero, con esto podemos moverlo según deseemos. Para la implementación real, tener en cuenta que `move.linear.x` se utilizará para mover el cuadricóptero en el eje $x$, `move.linear.y` se utilizará para mover el cuadricóptero en el eje $y$ y `move.angular.z` para girar el cuadricóptero en el eje $z$.

Con los *topics* definidos, ahora pasamos a escribir nuestro código en el *node* antes creado, para indicarle donde leer los datos así como donde enviar los datos procesados.

El *node* `control_simulacion`, necesitará suscribirse en los *topics* `pose` y `euler`, tanto para leer datos de posición y ángulos del cuadricóptero, luego necesitamos publicar en el *topic* `cmd_vel` para controlarlo de acuerdo al control diseñado.

Programa `control_simulacion.py`

```
...
      rospy.Subscriber('pose', PoseStamped,
leer_m_trayectoria_coordenadas)
      rospy.Subscriber('euler', Vector3Stamped, leer_angulos)
...
#------------Suscripcion al topico cmd_vel para control---------
inp_control = rospy.Publisher('cmd_vel', Twist, queue_size=10)
move=Twist()
#============================================================
def parar():
      inp_control.publish(Twist())
      print ("Nodo terminado")
#----------------------Bucle Principal-----------------------
def mainloop():
      rospy.init_node ('control_simulacion')
      rospy.on_shutdown(parar)
      while not rospy.is_shutdown():
            rospy.sleep (0.001)
if __name__ == '__main__':
      try:
            mainloop ()
      except rospy.ROSInterruptException : pass
```

Una vez que tengamos todo listo para realizar la simulación lo primero que debemos hacer es inicializar Gazebo y *RViz*. En el entorno Gazebo, podremos observar las maniobras ejecutadas por el cuadricóptero al momento que se tiene el vector de datos y la ejecución del algoritmo de control.

Para la visualizar el cuadricóptero en Gazebo, es necesario ejecutar el archivo `.launch` del paquete `hector_quadrotor`, desde una terminal realizamos esta operación, no olvidar que se debe indicar sobre el *workspace* en el cual se está trabajando, en este caso tenemos un *workspace* para el *stack* descargado del cuadricóptero con el nombre

[34] Son una extensión de los números reales, similar a la de los números complejos. Mientras que los números complejos son una extensión de los reales por la adición de la unidad imaginaria $i$, tal que $i^2 = -1$, los cuaterniones son una extensión generada de manera análoga añadiendo las unidades imaginarias: $i$, $j$ y $k$ a los números reales y tal que $i^2 = j^2 = k^2 = ijk = -1$.

`hector_quadrotor_tutorial`. Ejecutamos el archivo `quadrotor_empty_world.launch`, el cual muestra el cuadricóptero en Gazebo.

Terminal en Ubuntu 4-1

```
christian@yeymi:~/hector_quadrotor_tutorial$ roslaunch
hector_quadrotor_gazebo quadrotor_empty_world.launch
...
SUMMARY
========
PARAMETERS
 * /base_link_frame: /base_link
 * /controller/imu_topic:
 * /controller/motor/type: hector_quadrotor_...
 * /controller/pose/type: hector_quadrotor_...
...
Loaded the following quadrotor propulsion model parameters from
namespace /quadrotor_propulsion:
k_m     = -7.01163e-05
k_t     = 0.0153369
CT2s    = 0
CT1s    = -0.00025224
CT0s    = 1.53819e-05
Psi     = 0.00724218
J_M     = 2.57305e-05
R_A     = 0.201084
l_m     = 0.275
alpha_m = 0.104864
beta_m  = 0.549262
Loaded the following quadrotor drag model parameters from namespace
/quadrotor_aerodynamics:
C_wxy = 0.12
C_wz = 0.1
C_mxy = 0.0741562
C_mz = 0.0506433
```

Se puede visualizar los parámetros físicos que utiliza este cuadricóptero, para nuestra simulación dejaremos estos valores como están, pero estos pueden ser cambiados si se tienen los parámetros correctos. Ejecutado el archivo `.launch` se visualiza el cuadricóptero tal y como se muestra en la figura 4.10.

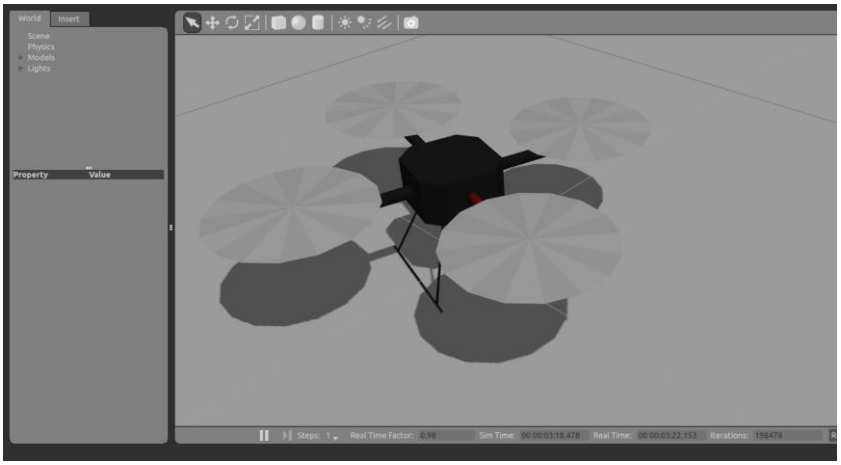



Figura 4.10. Cuadricóptero en el entorno Gazebo.
Fuente: Software Gazebo.

Seguidamente es importante iniciar *RViz* debido a que varios *topics* serán utilizados en nuestro *node* generados por este archivo `.launch`, para su ejecución tenemos que ejecutar: `hector_pose_estimation.launch`.

Terminal en Ubuntu 4-2

```
christian@yeymi:~/hector_quadrotor_tutorial$ roslaunch
hector_pose_estimation hector_pose_estimation.launch
... logging to /home/christian/.ros/log/b805345a-0903-11e7-b7db-
303a64981e85/roslaunch-yeymi-8850.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://yeymi:53351/
SUMMARY
========
PARAMETERS
 * /rosdistro: indigo
 * /rosversion: 1.11.20
NODES
  /
    hector_pose_estimation (hector_pose_estimation/pose_estimation)
    hector_pose_estimation_rviz (rviz/rviz)

ROS_MASTER_URI=http://localhost:11311

core service [/rosout] found
process[hector_pose_estimation-1]: started with pid [8871]
process[hector_pose_estimation_rviz-2]: started with pid [8873]
```

Cuando se ejecuta el archivo `.launch` se visualiza *RViz* con los datos del cuadricóptero, como se muestra en la figura 4.11.

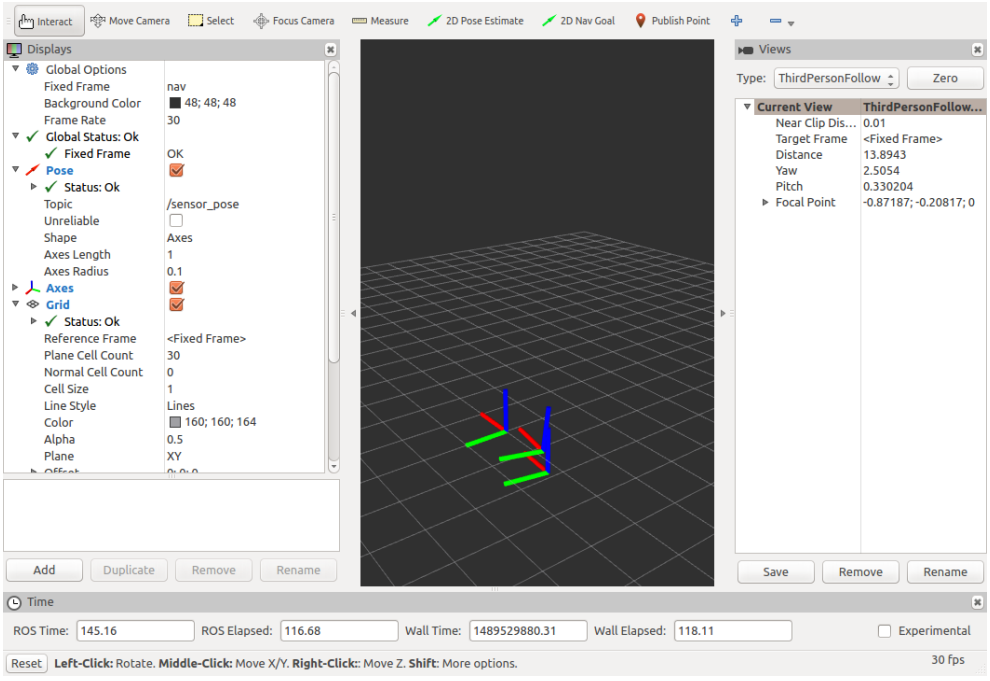



Figura 4.11. Cuadricóptero en el entorno *RViz*.
Fuente: Software *RViz*.

### 4.2.2.3. Identificación del sistema y sintonización del controlador PID

Para leer y escribir las variables de nuestro cuadricóptero en la simulación, tomamos los *topics* relacionados con estos que son creados por el paquete `hector_quadrotor`, estos son:

- Velocidad: suscribirse al *topic* `velocity`.
- Acción de control: publicar en el *topic* `cmd_vel`.

Lo que haremos es crear un node para realizar la identificación del sistema, el código generado se encuentra en el Apéndice C. Con esto se obtuvo alrededor de 180 datos con un periodo de muestreo de 1 segundo, como se muestra en la figura 4.12.

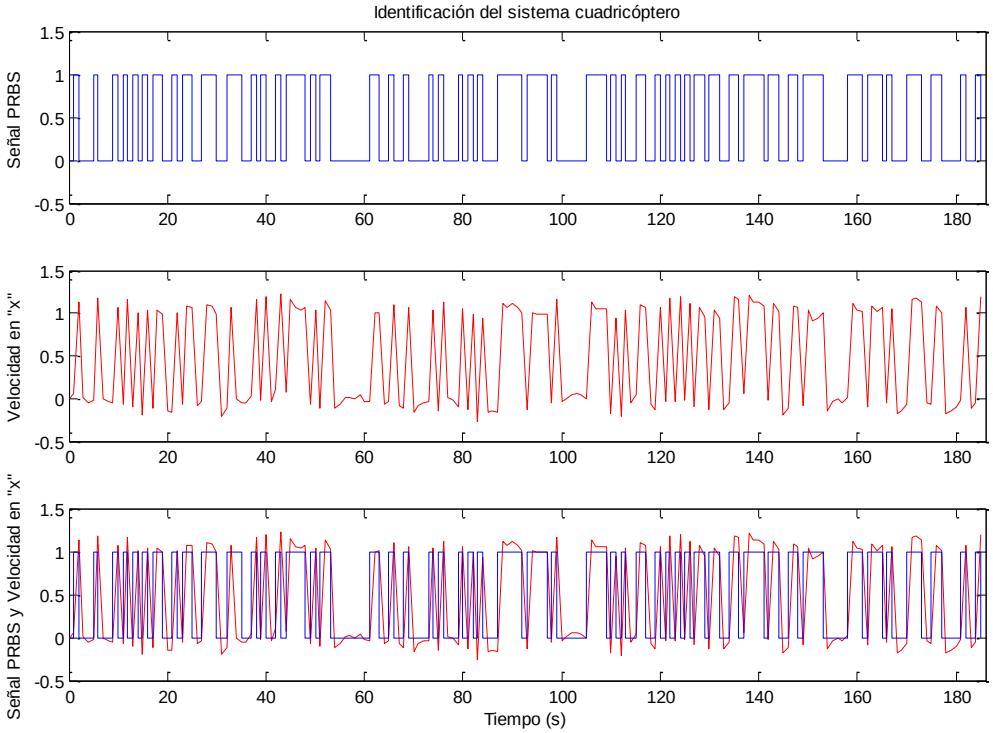



Figura 4.12. Señal PRBS generada y medición de la variable de salida.
Fuente: Elaboración propia.

Con estos datos obtenidos lo siguiente es coger la mitad de ambos datos, esto se hace con el objetivo de que una mitad sirva para la identificación y la otra mitad para la validación de nuestro modelo.

**Utilizando *System Identification Toolbox* de MATLAB**

Con los datos obtenidos anteriormente, ahora abrimos la intefaz gráfica del *toolbox* e ingresamos los datos para la identificación y validación. Como se observa en la figura 4.13.

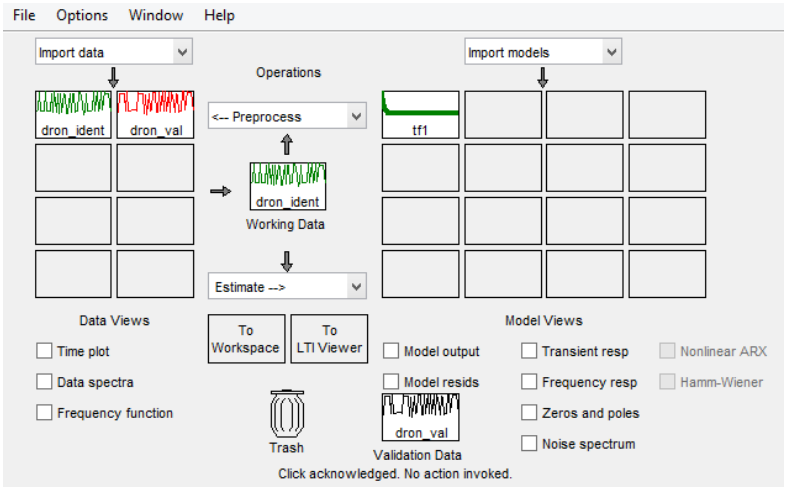



Figura 4.13. *System Identification Toolbox* de MATLAB.
Fuente: Elaboración propia.

Los datos que se han ingresado al toolbox y obtenidos son los siguientes:

- `dron_ident`: Son los datos para la identificación del modelo.
- `dron_val`: Son los datos para la validación del modelo obtenido.
- `tf1`: Modelo obtenido en función de transferencia.

Con la identificación se obtuvo la siguiente función de transferencia:

$$G(s) = \frac{38.14s + 34.58}{s^2 + 26.77s + 37.13}$$

Un criterio adecuado para saber que nuestro modelo obtenido es adecuado y con ello validar nuestro modelo es verificar el valor del FIT obtenido, este valor realiza la comparación entre los datos de validación y la predicción de la salida utilizando el modelo obtenido, tal como se muestra en la figura 4.14.

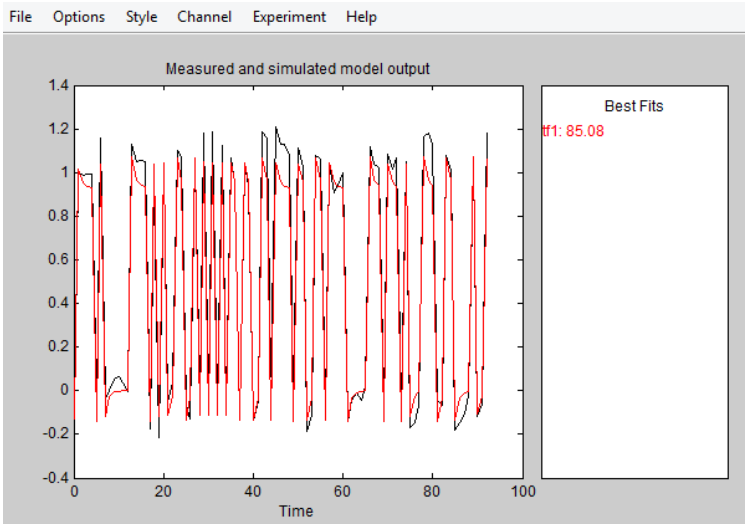



Figura 4.14. Validación del modelo obtenido de acuerdo al FIT.
Fuente: Elaboración propia.

De acuerdo a lo observado en la figura 4.14, los datos para la validación están en color negro y los predichos por el modelo están en rojo, de acuerdo con [35] y con un FIT= 85.08% el modelo obtenido es debido a que el valor de FIT esta entre los valores 75<FIT<100.

Con la función de transferencia obtenida procedemos a implementarlo en *Simulink* de MATLAB junto con un controlador PID, esto con el fin de realizar la auto-sintonización de los parámetros del controlador, el esquema se muestra en la figura 4.15.

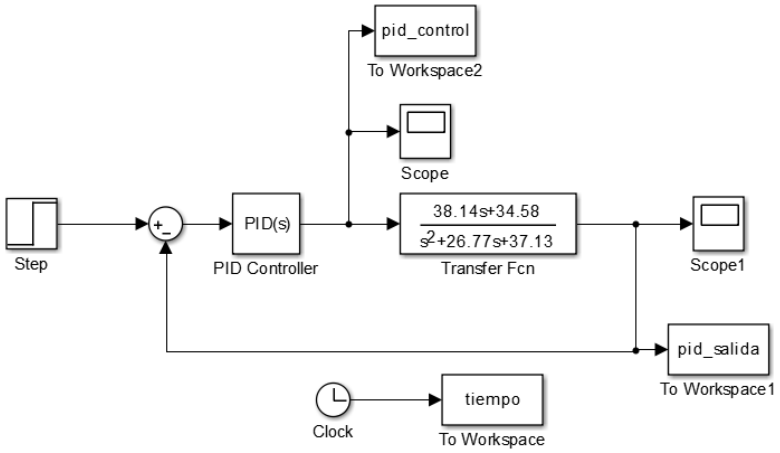



Figura 4.15. Implementación del controlador PID en *Simulink*.
Fuente: Elaboración propia.

Con el esquema realizado, ahora utilizamos el *toolbox* de auto-tuning PID, para obtener un sobre-impulso del 10% y un tiempo de establecimiento de 20 mseg, los parámetros PID obtenidos fueron:

- Kp=0.53
- Ki=29.26
- Kd=0.0057

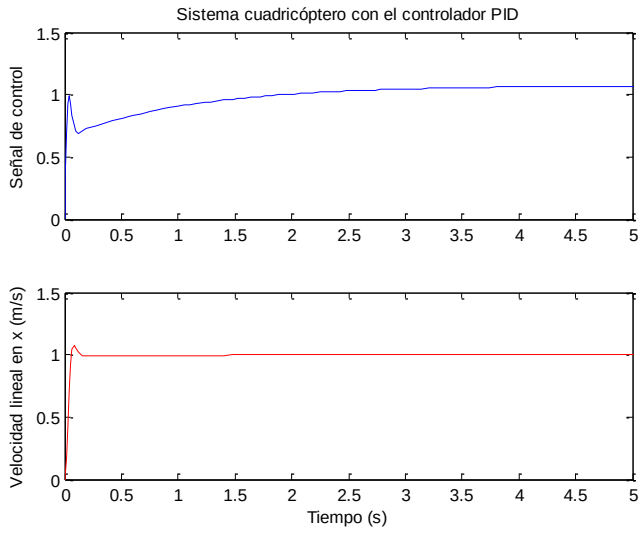



Figura 4.16. Señal de control y la variable de salida.
Fuente: Elaboración propia.

Con estos parámetros se obtuvieron los siguientes resultados para nuestro sistema, como se observa en la figura 4.16. Es importante tener en cuenta la señal de control, esto con el objetivo de no tener valores muy altos para la simulación y a la hora de implementación física también se tuvo en cuenta esto, para no saturar los actuadores y realice movimientos bruscos que dañen el cuadricóptero.

Al ser el simulador un entorno perfecto sin perturbaciones y considerando las suposiciones realizadas en el capitulo 1, los mismos parámetros del controlador son considerados para el otro eje $y$, en lo que se refiere al ángulo *yaw* se realizó el mismo procedimiento obteniéndose parámetros cercanos a los otros controladores PID con diferencia en la constante proporcional, donde el valor hallado fue diez veces el valor de $K_p$ obtenido para el control de los ejes.

Esta metodología es utilizada para la simulación, en lo que respecta a la implementación los parámetros PID se partió de los valores obtenidos en el trabajo de Monzón [42], debido a que no disponía mucho tiempo y no tener una forma adecuada de medir la velocidad lineal del cuadricóptero. De acuerdo a la información proporcionada del ArduCopter [4] nos indica que la velocidad máxima que alcanza es 5 m/s, esto resulta útil a la hora de indicarle la velocidad máxima al algoritmo de perfil de velocidad trapezoidal, en base a estos valores se fue ajustando los parámetros del controlador PID para mejorar el desempeño del controlador.

#### 4.2.2.4. Pruebas realizadas

Se debe ejecutar los archivos `.launch` del paquete `hector_quadrotor` para Gazebo y *RViz*, para luego ejecutar nuestro *node* creado `control_simulacion`, una vista de los comandos usados en la *terminal* se ve en la figura 4.17, donde se observa varias terminales para la ejecución de la simulación.

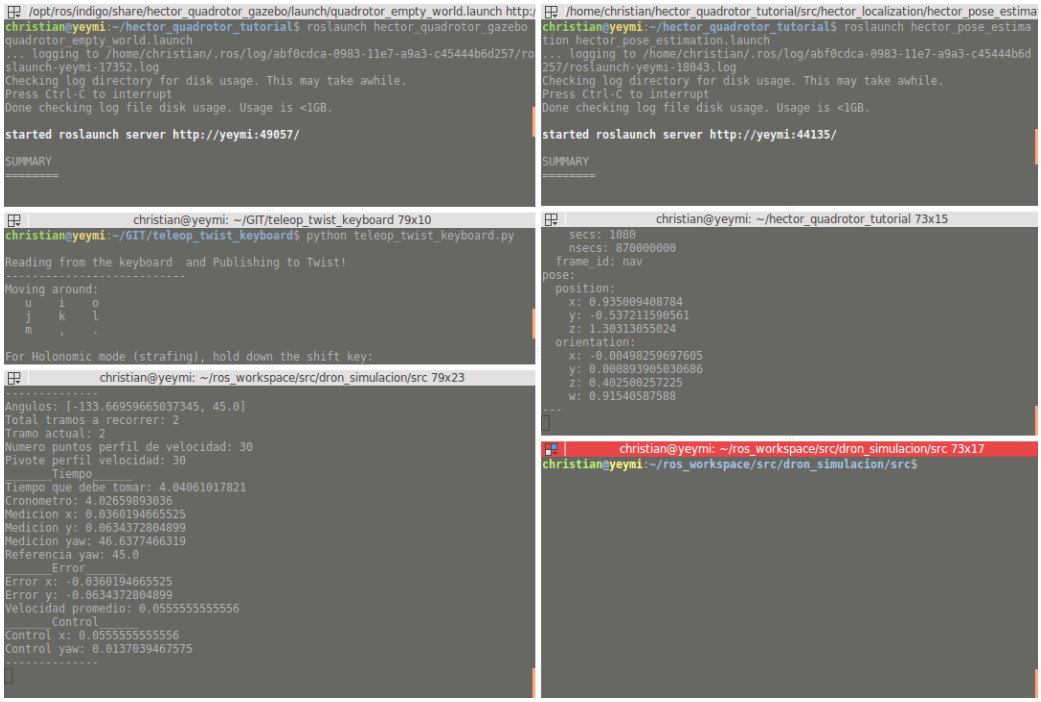



Figura 4.17. Terminales abiertas para la ejecución del sistema de control.
Fuente: Elaboración propia.

A continuación mostramos los datos que se ven la *terminal* primero cuando se utiliza el método en una dirección:

Terminal en Ubuntu 4-3

```
christian@yeymi:~/ros_workspace/src/dron_simulacion/src$ python
control_cuadricoptero_simulacion.py -f trayectoria -m angulo -w -2 -2 0
0
...
Posicionando angulo
Angulo Yaw:  -100.470868618
Vector yaw: [-138.89756547528242, 45.0]
Referencia yaw: -138.897565475
Error yaw: -39.0795052203
Control: -3.53043319965
Numero de angulo: 1
--------------------------
Angulos: [-133.66959665037345, 45.0]
Total tramos a recorrer: 2
Tramo actual: 2
Numero puntos perfil de velocidad: 30
Pivote perfil velocidad: 30
_______Tiempo______
Tiempo que debe tomar: 4.04061017821
Cronometro: 4.02659893036
Medicion x: 0.0360194665525
Medicion y: 0.0634372804899
Medicion yaw: 46.6377466319
Referencia yaw: 45.0
_______Error______
Error x: -0.0360194665525
Error y: -0.0634372804899
Velocidad promedio: 0.0555555555556
_______Control______
Control x: 0.0555555555556
Control yaw: 0.0137039467575
```

Como se describió en el capítulo anterior, primero obtenemos los ángulos y después el único movimiento que realizamos es hacia adelante, como se ve en la terminal se posiciona de acuerdo al ángulo n°1, después va hacia adelante hasta el siguiente punto, para luego girar de acuerdo al ángulo n°2 e ir hacia adelante nuevamente y llega al último punto, cabe mencionar que los valores de los puntos para la trayectoria son [−2, −2, 0, 0] y para el cálculo del ángulo se toma otro punto que es el punto inicial, para nuestro caso es la posición de donde parte el cuadricóptero.

En la figura 4.18 se visualiza el error tanto de la longitud (recordar que el eje $x$ está asociado al eje del meridiano de Greenwich) y latitud (este es el eje $y$ relacionado a la línea ecuatorial), se ve claramente el cambio en pequeños segmentos como escalones, esto se debe a que al ejecutar el algoritmo de perfil de velocidad trapezoidal el tramo conformado entre dos puntos está dividido en pequeños tramos, para nuestra simulación; *punto 1* (-2,-2) y *punto 2* (0,0), lo sub-divide en 30 tramos, creando otros valores de referencia pero dentro de este rango, por esta razón se debe la forma de la gráfica.

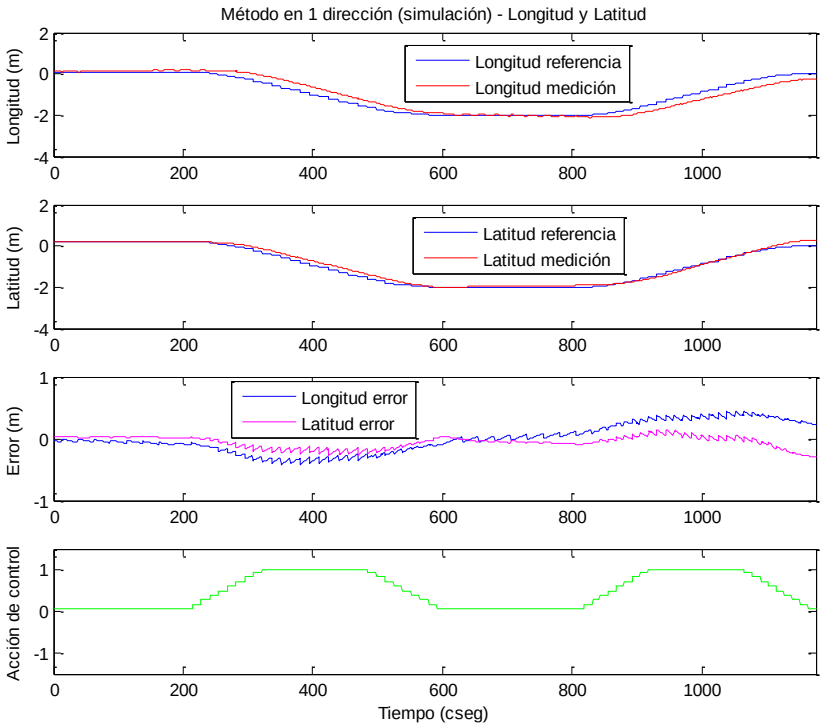



Figura 4.18. Longitud, Latitud y señal de control para método en una dirección.
Fuente: Elaboración propia.

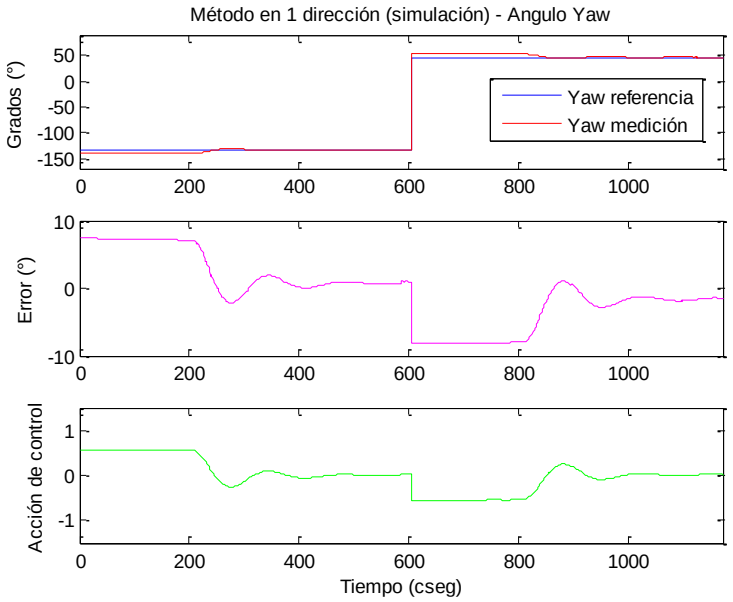



Figura 4.19. Ángulo *yaw* y señal de control para método en una dirección.
Fuente: Elaboración propia.

Para la gráfica 4.19 se observa que existen unos cambios instantáneos, esto se debe a que el ángulo *yaw* va cambiando conforme se llegué a un punto de referencia de acuerdo al vector de ángulos calculados en el programa. Según se calculó para esta simulación (ver terminal en Ubuntu 4-3) los ángulos yaw calculados son: -133° y 45°, los cuales son los valores de referencia. Ahora veremos los datos en terminal para el siguiente método en dos direcciones:

Terminal en Ubuntu 4-4

```
christian@yeymi:~/ros_workspace/src/dron_simulacion/src$ python
control_cuadricoptero_simulacion.py -f trayectoria -m vectores -w -2 -2
0 0
...
Total tramos a recorrer: 2
Tramo actual: 2
Numero puntos perfil de velocidad: 30
Pivote perfil velocidad: 30
_______Tiempo______
Tiempo que debe tomar: 4.04061017821
Cronometro: 4.03640198708
_______Medicion______
Medicion x: -0.0108997579593
Medicion y: 0.271729207096
_______Comparar______
Pivote Vx: 0.0392837100659
Pivote Vy: 0.0392837100659
_______Error______
Medicion yaw: 0.526108307079
Error x: 0.0108997579593
Error y: -0.271729207096
_______Control______
Control x: 0.0450549958536
Control y: -0.0846986199264
```

También como se describió en el capítulo anterior, la acción de control es en dos direcciones pero siempre tratando de mantener el ángulo *yaw* = 0°, con esto nos aseguramos que el movimiento sea correcto, los puntos para la trayectoria son los mismos que el método anterior y también se considera la posición inicial del cuadricóptero.

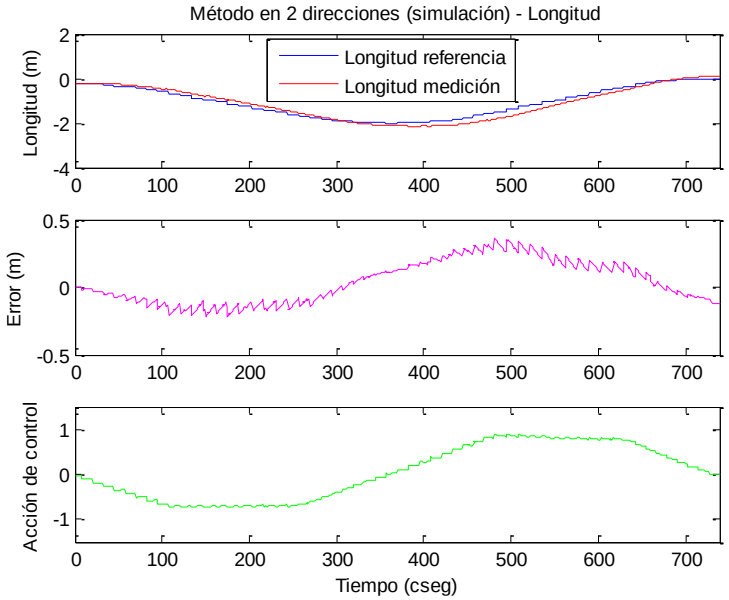



Figura 4.20. Longitud y señal de control para método en dos direcciones.
Fuente: Elaboración propia.

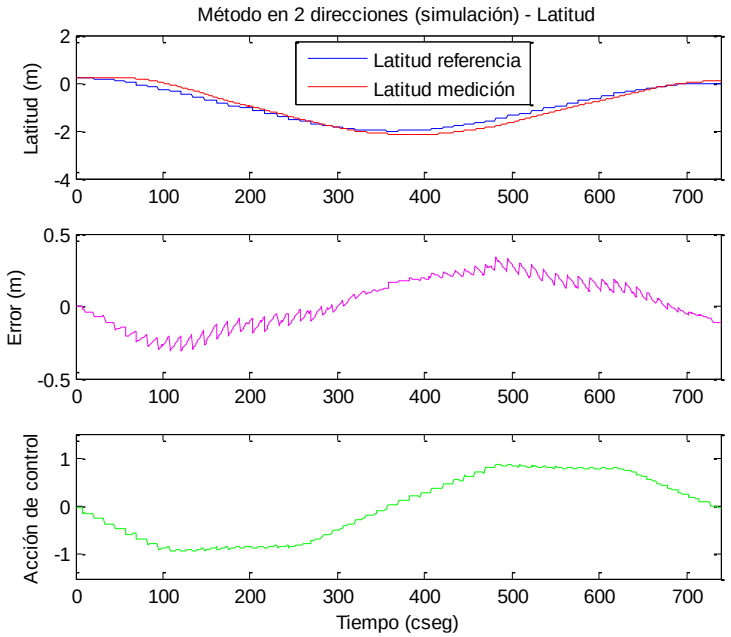



Figura 4.21. Latitud y señal de control para método en dos direcciones.
Fuente: Elaboración propia.

Al igual que el método en una dirección se tiene el mismo comportamiento de división por tramos de los valores de referencia tal como se observa en la figura 4.20 y 4.21.

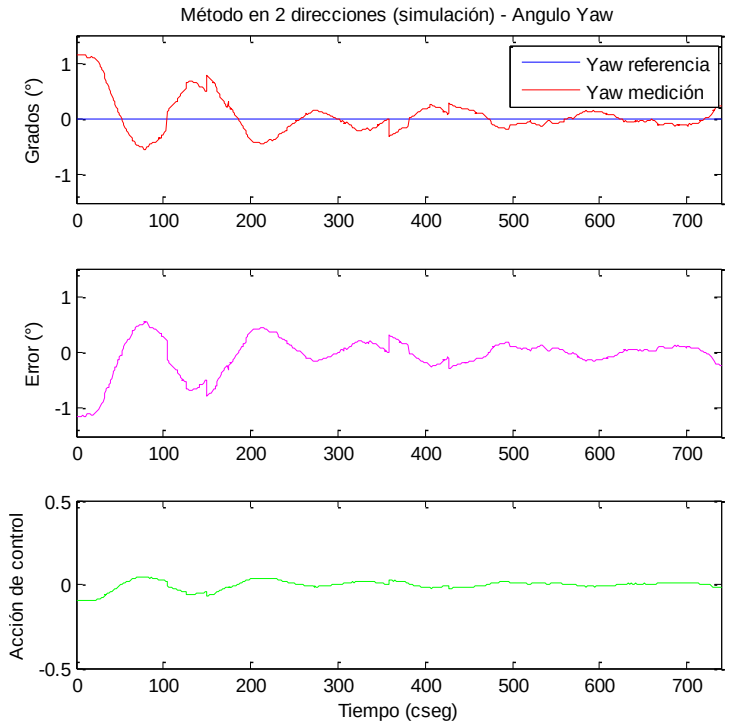



Figura 4.22. Ángulo *yaw* y señal de control para método en dos direcciones.
Fuente: Elaboración propia.

A diferencia del método anterior, el ángulo *yaw* siempre debe ser cero, por lo cual el programa ejecuta este control tratando de minimizar el error como se ve en la figura 4.22.

También se realizó pruebas adicionales utilizando la conversión de coordenadas, es decir se utilizó el *gps* virtual creado por el paquete `hector_quadrotor`, para ello sólo se realizó algunas modificaciones en el programa como cambiar algunos *topics* por ejemplo utilizar el *topic* `geopose` en lugar `pose` y el algoritmo de conversión de coordenadas, tal como se muestra en el Apéndice D, los comandos utilizados en la terminal de Ubuntu son los mismos utilizados anteriormente, para esta prueba se dibujó una trayectoria en el entorno de Gazebo, con las siguientes coordenadas dadas en la tabla 4.1:

Tabla 4.1: Coordenadas empleadas para seguir una trayectoria.

| Puntos | Longitud (°) (Coordenadas Geodésicas) | Latitud (°) (Coordenadas Geodésicas) | Longitud (m) (Coordenadas UTM) | Latitud (m) (Coordenadas UTM) |
|---|---|---|---|---|
| **Origen** | 8.68687260615 | 49.8602402809 | 477496 | 5523249 |
| **Punto 1** | 8.68713948904 | 49.8603561009 | 477512 | 5523262 |
| **Punto 2** | 8.68723011293 | 49.8602086682 | 477518 | 5523246 |
| **Punto 3** | 8.68695353872 | 49.8601685486 | 477498 | 5523241 |
| **Punto 4** | 8.68688768524 | 49.8603054688 | 477494 | 5523257 |

Fuente: Elaboración propia.

Primero se realizó las pruebas para el método en una dirección, el código para está simulación se encuentra en el Apendice B, con lo que se obtuvieron los siguientes resultados.

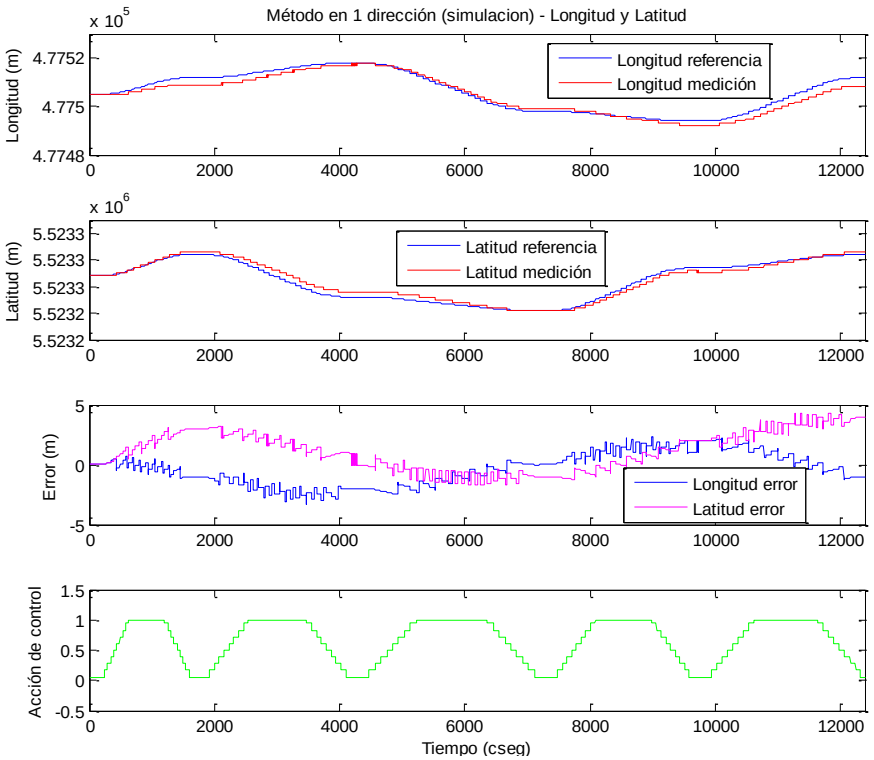



Figura 4.23. Longitud, latitud, error y señal de control para método en una dirección.
Fuente: Elaboración propia.

En la figura 4.23 y 4.24 se puede observar las coordenadas longitud y latitud en UTM, el error y la acción de control.

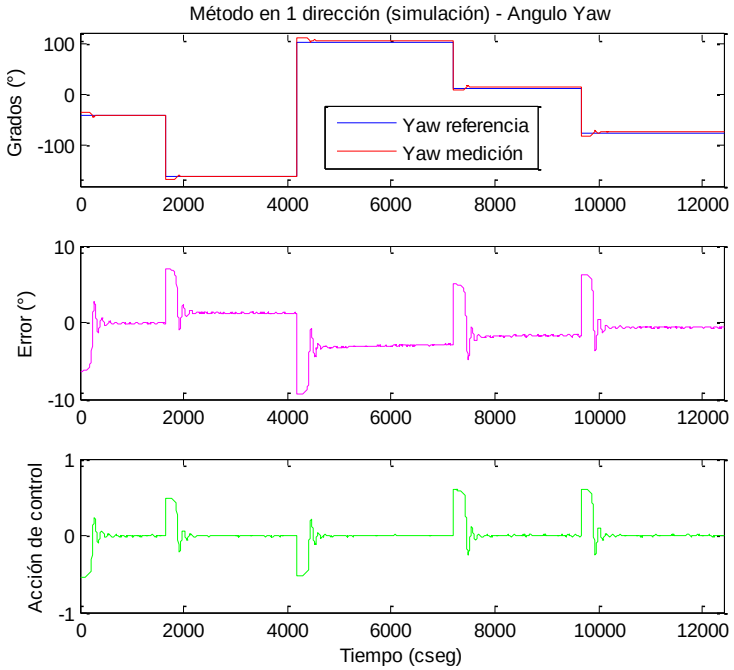



Figura 4.24. Ángulo *yaw* y señal de control para método en una dirección.
Fuente: Elaboración propia.

Se puede observar en la figura 4.24, que los ángulos de referencia son negativos, esto se debe al simulador Gazebo, en sus ejes de referencia se encuentran diferentes, es decir el eje $x$ están los datos de latitud y en el eje $y$ los datos de longitud esto resulta que el norte geográfico del simulador no coincide con nuestro sistema de referencia, por ello se hizo este cambio, la acción de control hace que se siga la referencia de forma adecuada como se muestra. Para el método en dos direcciones, se obtuvieron los siguientes resultados:

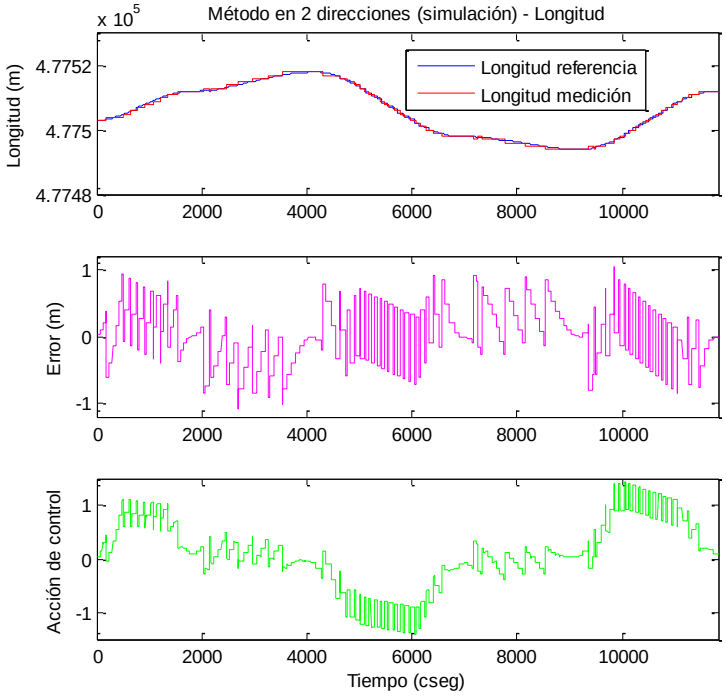



Figura 4.25. Longitud, error y señal de control para método en dos direcciones.
Fuente: Elaboración propia.

El seguimiento a la longitud de referencia es mucho mejor, como se puede observar en la figura 4.25, el error es menor en comparación con el método anterior.

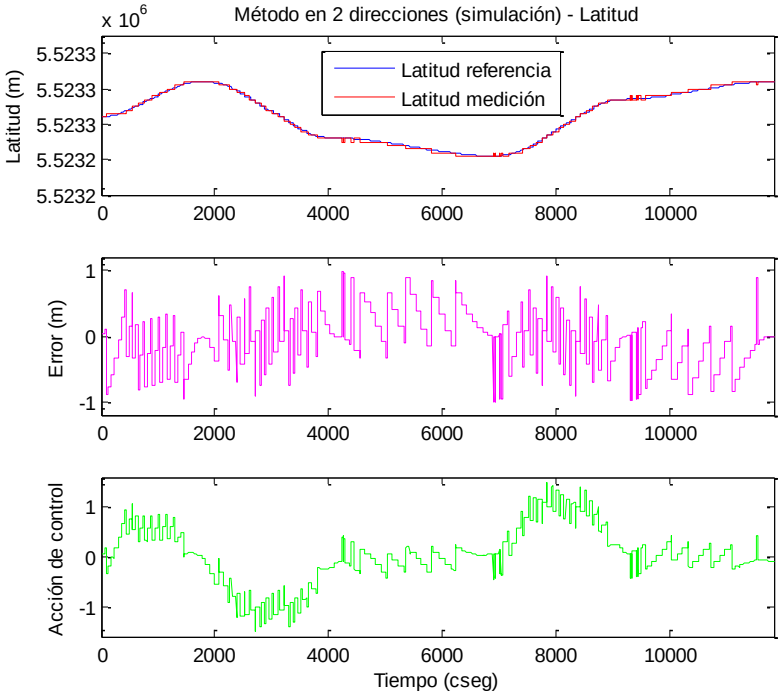



Figura 4.26. Latitud, error y señal de control para método en dos direcciones.
Fuente: Elaboración propia.

Lo mismo sucede para el eje *x* relacionado con la latitud, el error es menor en comparación con el método anterior como se observa en la figura 4.26.

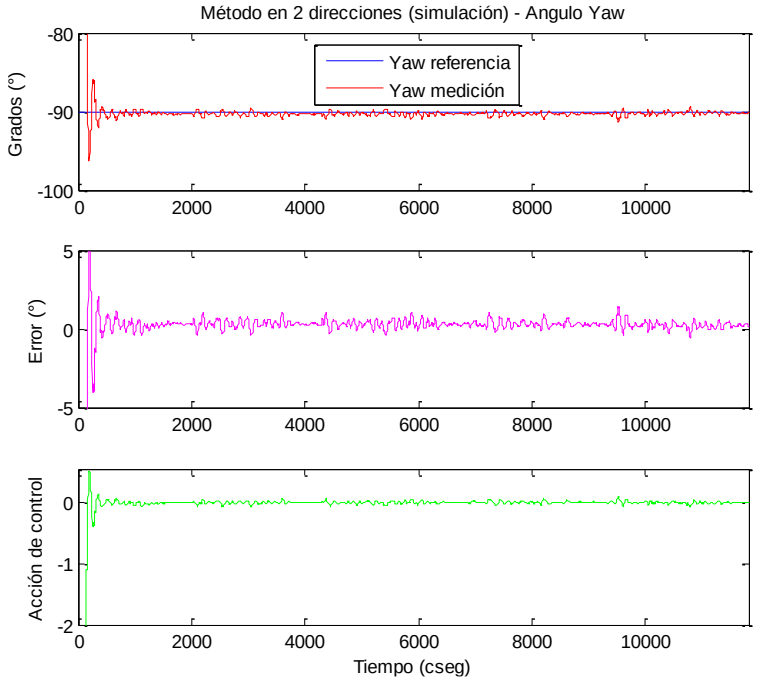



Figura 4.27. Ángulo *yaw*, error y señal de control para método en dos direcciones.
Fuente: Elaboración propia.

Y finalmente para el control del ángulo *yaw*, como se muestra en la figura 4.22, se tuvo en cuenta el cambio de los ejes de referencia en el simulador de Gazebo, para que coincidiera con nuestro control, el ángulo *yaw* de referencia fue de -90° a diferencia de las simulaciones que se realizaron con puntos en $x$ y $y$, con un ángulo *yaw* de referencia=0°.

Es importante tener en cuenta que el norte geográfico no coincide con el norte de nuestro cuadricóptero en otras palabras el sistema de referencia (es decir 0° en el eje de los datos de longitud) no coincide con el eje de referencia de nuestro cuadricóptero como se observó en la figura 4.24, lo que podría traer errores en el cálculo de los ángulos *yaw* o equivocarse con las acciones de control.

Con todas estas simulaciones realizadas, ya se tiene todo preparado para la implementación final y así realizar las pruebas de campo, pero antes se debe tener en cuenta algunas consideraciones en la utilización del *ArduCopter* y *ROS* como se menciona en el siguiente apartado.

### 4.2.3. Desarrollo en entorno real

Para esta parte de implementación, el código no varía mucho, más sólo en los *topics*, esto se hace con el fin de suscribirse y publicar en los *topics* creados por el *package* `roscopter`, por eso primero explicaremos que es `roscopter`, datos y que funciones ofrece para su utilización.

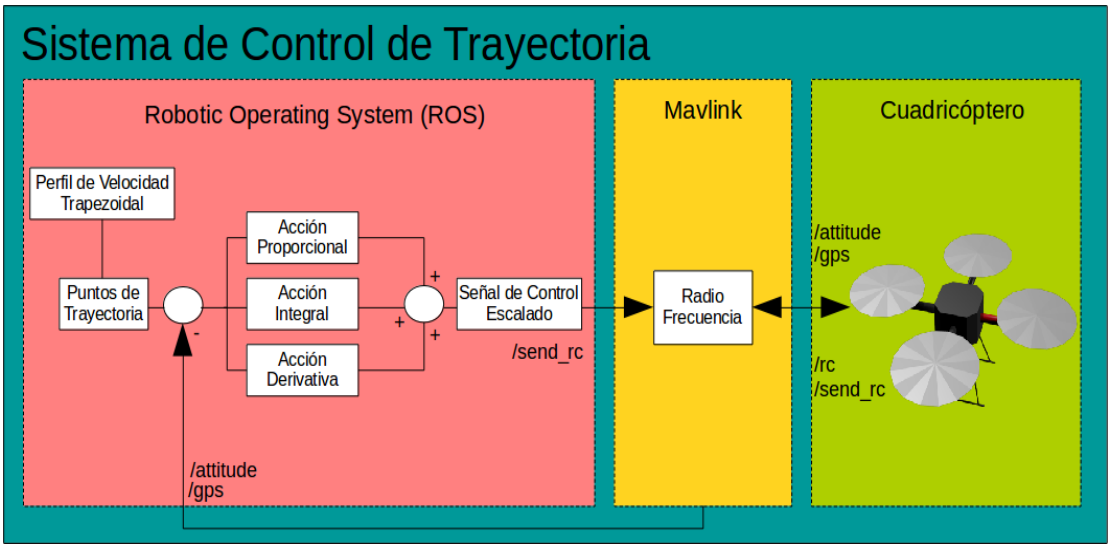



Figura 4.28. Diagrama de bloques del sistema de control implementado.
Fuente: Elaboración propia.

En la figura 4.28 observamos los bloques que conforman nuestro sistema de control para la implementación del mismo. A diferencia de los diagramas de bloques anteriores se le ha agregado un escalado de la señal de control.

#### 4.2.3.1. Módulo de interfaz roscopter

Es un paquete de *ROS* desarrollado y actualizado por la comunidad de *ROS*, el cual se encarga de comunicar el *ArduCopter* con *ROS* usando como medio *MAVlink* 1.0, usando

`roscopter` se puede leer los datos como posición (`gps`), acelerómetro, etc. los cuales están publicados en diferentes *topics* además se puede sobre-escribir sobre el *topic* donde se publican los comandos de radio control (`rc`) y de esta forma controlar nuestro cuadricóptero [43]. El código de `roscopter` utilizado se encuentra en el Apéndice E.

Explicaremos como se realiza su compilación:

a. Instalamos las dependencias necesarias de acuerdo a la versión de *ROS* que tenemos instalado en nuestra computadora, las versiones probadas de *ROS* con `roscopter` fueron *ROS HYDRO* y *ROS INDIGO*.

| Terminal en Ubuntu 4-5 |
|---|
| `christian@yeymi:sudo aptitude install ros-indigo-sensor-msgs python-serial python-tz` |

b. Descargamos desde los repositorios el paquete `roscopter` e inicializamos el sub-módulo `mavlink`.

| Terminal en Ubuntu 4-6 |
|---|
| `christian@yeymi:cd ~/ros_workspace/src`<br>`git clone https://github.com/ssk2/roscopter.git`<br>`cd roscopter`<br>`git submodule update –init` |

c. Construimos el paquete con los comandos `catkin`.

| Terminal en Ubuntu 4-7 |
|---|
| `christian@yeymi:cd ~/ros_workspace/src`<br>`catkin_make --pkg roscopter` |

d. Generamos los enlaces entre *Python* y *MAVlink*.

| Terminal en Ubuntu 4-8 |
|---|
| `christian@yeymi:cd ~/ros_workspace/src/roscopter/mavlink`<br>`./pymavlink/generator/mavgen.py --output pymavlink/dialects/v10/ardupilotmega.py message_definitions/v1.0/ardupilotmega.xml` |

Con esto ya tenemos listo `roscopter` para poder utilizarlo, para poder comprobar esto, conectamos el *ArduCopter* vía USB, utilizando un cable y abrimos una terminal:

| Terminal en Ubuntu 4-9 |
|---|
| `christian@yeymi:cd ~/ros_workspace`<br>`rosrun roscopter roscopter_node.py  --device=/dev/ttyACM0 --baudrate=115200` |

Pero si necesitamos que la comunicación sea usando radio frecuencia, se debe utilizar otra frecuencia para la transmisión de datos, es decir:

| Terminal en Ubuntu 4-10 |
|---|
| `christian@yeymi:cd ~/ros_workspace`<br>`rosrun roscopter roscopter_node.py  --device=/dev/ttyUSB0 --baudrate=57600` |

Tener en cuenta que debemos darle permisos a nuestro usuario en Ubuntu para poder leer y escribir datos sobre el *ArduCopter*. Esto se realiza de forma sencilla abriendo otra terminal:

| Terminal en Ubuntu 4-11 |
| --- |
| `christian@yeymi:sudo usermod -a -G dialout username` |

Con esto ya deberíamos recibir las señales del *ArduCopter* y los *heartbeat*[35], luego mensajes de inicialización del *ArduCopter*, para finalmente mostrar *"Ready to FLY".*

Para poder controlar la *ArduCopter* desde *ROS*, debemos habilitar el modo control del paquete `roscopter`, con esto `roscopter` creará otro *topic* `send_rc` en este *topic* debemos re-escribir las señales de control calculados de nuestros algoritmos de control, esto es importante porque el *topic* `rc` solo hace la lectura de los datos, pero para poder enviar las señales de control se hace en el *topic* `send_rc`.

| Terminal en Ubuntu 4-12 |
| --- |
| `christian@yeymi:cd ~/ros_workspace`<br>`rosrun roscopter roscopter_node.py  --device=/dev/ttyUSB0 --baudrate=57600 --enable-control=true` |

#### 4.2.3.2. ArduCopter *y roscopter*

Una vez instalado el paquete `roscopter` podemos ver los diferentes *topics* ya sea para leer o escribir datos así como también los *services* creados, en esta parte se hará una descripción de todo esto para conocer nuestro cuadricóptero y cómo implementar el controlador [42].

| Terminal en Ubuntu 4-13 |
| --- |
| `christian@yeymi:rostopic list`<br>`/attitude`<br>`/gps`<br>`/raw_imu`<br>`/rc`<br>`/rosout`<br>`/rosout_agg`<br>`/send_rc`<br>`/state`<br>`/vfr_hud` |

***Topics* de `roscopter`**

Una vez conectado ROS  y el *ArduCopter* utilizando `roscopter`, podemos ver los *topics* publicados como se muestra en la figura 4.29.

[35] Señales que envía el *ArduCopter* al mando de radio frecuencia, para confirmar la comunicación entre ambos es la adecuada.

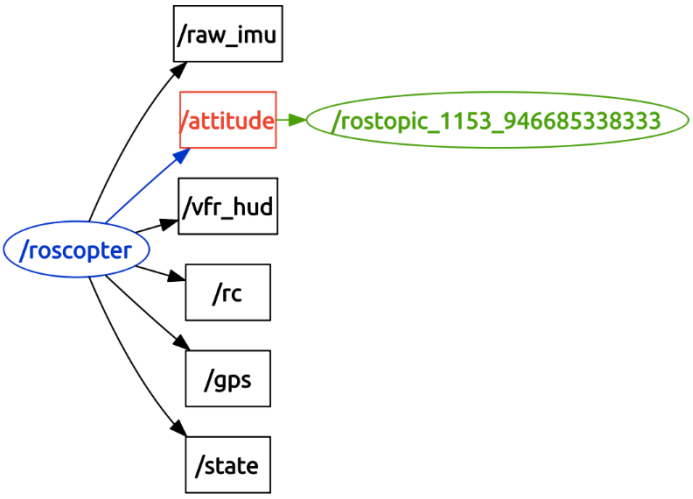



Figura 4.29. *Topics* de `roscopter`.
Fuente: Erle Robotic.

Explicaremos cada uno de estos *topics* y el formato de los mismos:

- `/attitude`: Se tiene los ángulos y velocidades angulares, por lo que tenemos 6 datos: `roll`, `pitch`, `yaw`, `rollspeed`, `pitchspeed` y `yawspeed`. El formato es *float32*.
- `/gps`: Se tiene datos y estados del *GPS*, para nuestro caso sólo utilizaremos: `longitude` y `altitude`, ambos tienen el formato *float64*.
- `/raw_imu`: Se tiene los datos del magnetómetro con los siguientes datos: `time_usec`, `xacc`, `yacc`, `zacc`, `xgyro`, `ygyro`, `zgyro`, `xmag`, `ymag` y `zmag`. El formato es *int32*.
- `/rc`: Se tiene las señales de control de radio control, con un vector de 8 datos: `channel`. El formato es *int32*.
- `/state`: Se tiene 3 datos importantes que son los estados del cuadricóptero: `armed`, `guided` y `mode`. El formato es *bool* y *string*.
- `/vfr_hud`: Se tiene 6 datos: `airspeed`, `groundspeed`, `heading`, `throttle`, `alt` y `climb`. El dato que nos interesa es `alt` ya que es la altura que el barómetro mide con formato *float32*.

### *Services* de `roscopter`

En otra terminal podemos ver los *services* disponibles:

Terminal en Ubuntu 4-14

```
christian@yeymi:rosservice list
/arm
/disarm
/rosout/get_loggers
/rosout/set_logger_level
```

- `/arm`: Con este *service* armamos los motores del cuadricóptero y quede listo para volar, se utiliza el comando `rosservice call arm`.
- `/disarm`: Este otro *service* se utiliza para desarmar los motores del cuadricóptero con esto se tiene los motores quietos, parecido al anterior se utiliza el comando `rosservice call disarm`.

- `/rosout/get_loggers` y `/rosout/set_logger_level`: Son propios de *ROS*, es decir están por defecto al iniciar `roscore`.

Otro factor a tener en cuenta es la configuración del mando de radio control, el que utilizaremos es de la empresa *Turnigy*, como se muestra en la figura 4.30.


Figura 4.30. Mando radio control *Turnigy*.
Fuente: Elaboración propia.

Se debe conocer las señales de control que el *ArduCopter* puede recibir, los máximos y mínimos valores que utiliza así como la dirección de movimiento de cada palanca.

La figura 4.30 muestra el mando a utilizar, la palanca de la derecha controla los movimientos de adelante-atrás y derecha-izquierda, la palanca de la izquierda controla los movimientos de ascenso y descenso y giro en su propio eje (ángulo *yaw*), este mando envía señales de control *PWM* en 8 canales con valores entre 1000 (mínimo) y 2000 (máximo) milisegundos, a continuación describimos cada canal [4]:

- `channel[0]`: Este canal envía señales para control de movimientos de izquierda-derecha, el mando está configurado de forma que el movimiento hacia la izquierda son valores de 1500 a 1000, es decir cuanto más cercano este a 1000 el cuadricóptero irá a su máxima velocidad y para la derecha son valores de 1500 a 2000 con valores cercanos a 2000 irá el cuadricóptero a la máxima velocidad, es importante tener en cuenta todo esto para evitar accidentes.
- `channel[1]`: En este canal se envía señales para control de movimientos de adelante-atrás, el mando está configurado de forma que el movimiento hacia la adelante son valores de 1500 a 1000, es decir cuanto más cercano este a 1000 el cuadricóptero irá a su máxima velocidad y para atrás son valores de 1500 a 2000 con valores cercanos a 2000 irá el cuadricóptero a su máxima velocidad.

- `channel[2]`: En este canal se envía señales para control del ángulo *yaw*, giro horario y anti-horario, el mando está configurado de forma que el giro horario son valores de 1500 a 1000, es decir cuanto más cercano este a 1000 el cuadricóptero irá a su máxima velocidad angular y para un giro anti-horario son valores de 1500 a 2000 con valores cercanos a 2000 girará el cuadricóptero a su máxima velocidad angular.
- `channel[3]`: Mediante este canal se controla la elevación del cuadricóptero, la configuración del mando con valores de 1000 a 2000, con un valor cercano a 2000 el cuadricóptero ascenderá de la forma más rápida y valores cercanos a 1000 descenderá y dejará de girar las hélices.
- `channel[4]`: Este canal indica los modos de vuelo previamente configurados en el *ArduCopter*. En el *ArduCopter* se pueden configurar hasta 6 modos de los 12 que tiene disponible, cada modo se representa como un rango de valores de señal PWM, es decir: modo 1=0-1230 , modo 2=1231-1360, modo 3=1361-1490, modo 4=1491-1620, modo 5=1621-1749 y modo 6=1750-2000. Para nuestro *ArduCopter* configuramos los modos: *STABILIZE* (estabilizado) y *ALT_HOLD* (mantener altura) en los modos 1 y 6 del *ArduCopter*.
- `channel[5]`: Para configuraciones adicionales como control deposición de una cámara abordo del cuadricóptero.
- `channel[6]`: Para configuraciones auxiliares.
- `channel[7]`: Para configuraciones auxiliares.

Figura 4.31. Cuadricóptero implementado en el banco de pruebas del laboratorio SAC.
Fuente: Elaboración propia.

Con todo esto en cuenta, ahora podemos realizar algunas pruebas en un módulo para pruebas, con el objetivo que la acción de control se ejecute de forma satisfactoria, pero no se puede controlar la posición debido a que el cuadricóptero está fijado a la estructura como se observa en la figura 4.31, por ello solo podemos controlar el ángulo *yaw*, este control aplica para ambos métodos utilizados.

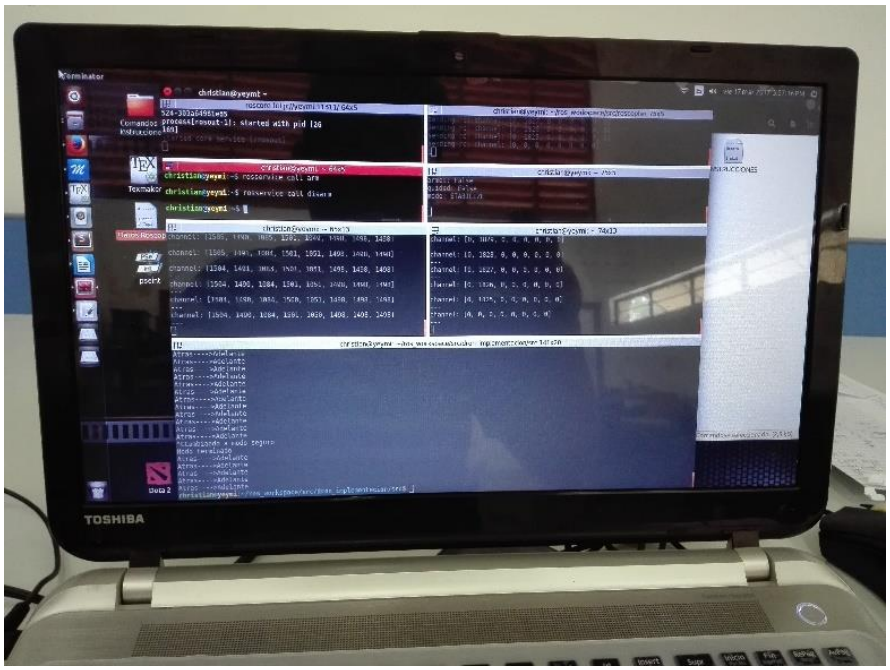



Figura 4.32. Terminales abiertos en Ubuntu para la ejecución de *ROS* y comunicación con *ArduCopter*.
Fuente: Elaboración propia.

En la figura 4.32 observamos que se necesitan abrir varias terminales para hacer pruebas de nuestro sistema de control.

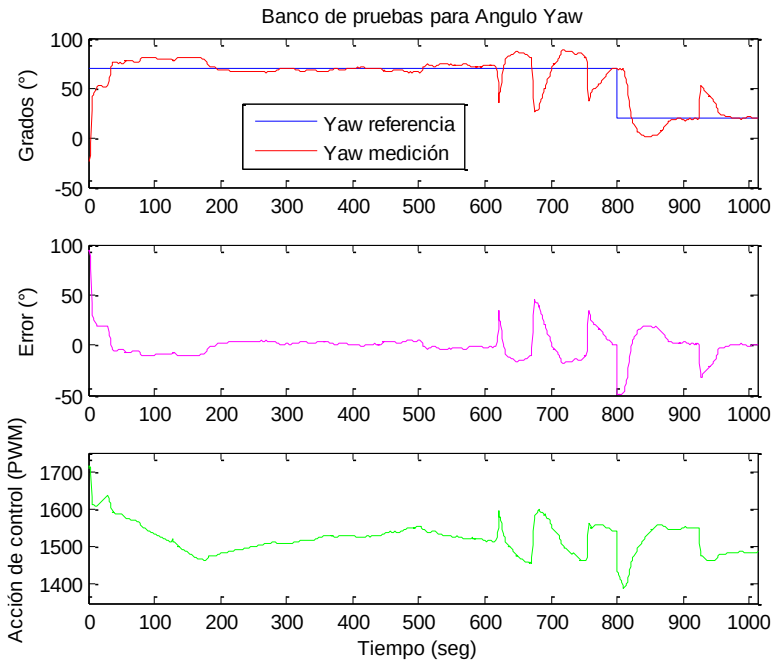



Figura 4.33. Pruebas realizadas para el control del ángulo *yaw*.
Fuente: Elaboración propia.

De acuerdo a los resultados obtenidos en la figura 4.33 los valores de referencia dados son 70° y 20° respectivamente, cuando el valor de referencia era 70° se aplicó varios disturbios entre los 600 y 800 segundos, es por ello que la acción de control tiene unos picos, pero tiene un buen comportamiento el control para el ángulo *yaw* con esto estamos preparados para realizar las pruebas de control de posición.

Para las pruebas en un entorno abierto, se eligió un lugar donde no existan obstáculos y de suelo plano, las pruebas se realizaron durante la mañana entre las 9:00 – 11:00 horas, por ello se realizó las pruebas en el campo de futbol de la universidad como se observa en la figura 4.34 y las coordenadas alrededor del campo están en la tabla 4.2.

Tabla 4.2: Coordenadas del campo de futbol de la universidad.

| Puntos | Longitud (°) (Coordenadas Geodésicas) | Latitud (°) (Coordenadas Geodésicas) | Longitud (m) (Coordenadas UTM) | Latitud (m) (Coordenadas UTM) |
|---|---|---|---|---|
| **Punto 1** | -80.6379538 | -5.1738210 | 540125 | 9428104 |
| **Punto 2** | -80.6386389 | -5.1736995 | 540049 | 9428117 |
| **Punto 3** | -80.6385687 | -5.1730322 | 540057 | 9428191 |
| **Punto 4** | -80.6379866 | -5.1730605 | 540121 | 9428188 |

Fuente: Elaboración propia.

Figura 4.34. Campo deportivo de la universidad para la realización de las pruebas.
Fuente: Elaboración propia.

En primer lugar se hizo pruebas con el método en una dirección, en la que se asignó un punto de referencia al que tenía que llegar, es decir sólo dos puntos incluido la posición inicial, el código de implementación se encuentra en el Apéndice F, con esto se obtuvieron los siguientes resultados en las figuras 4.35 y 4.36.

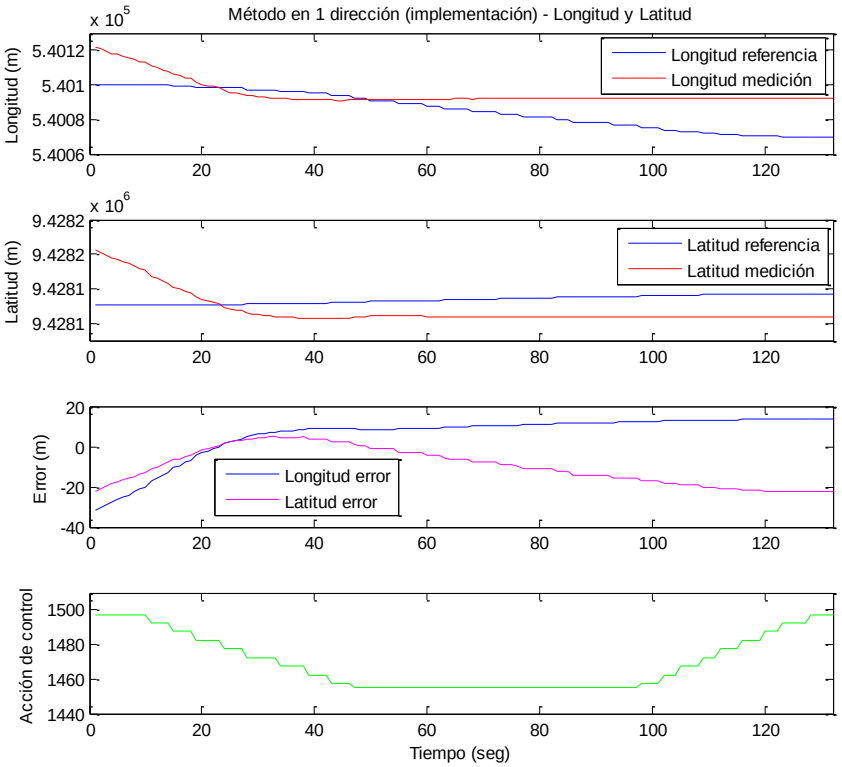



Figura 4.35. Pruebas realizadas en campo, datos de longitud y latitud.
Fuente: Elaboración propia.

Como se observa en la figura 4.35, el error tanto en la longitud y latitud va aumentando en comparación a los resultados de simulación, cabe mencionar que esto es de esperarse porque no se tiene ningún controlador implementado para corregir estos errores.

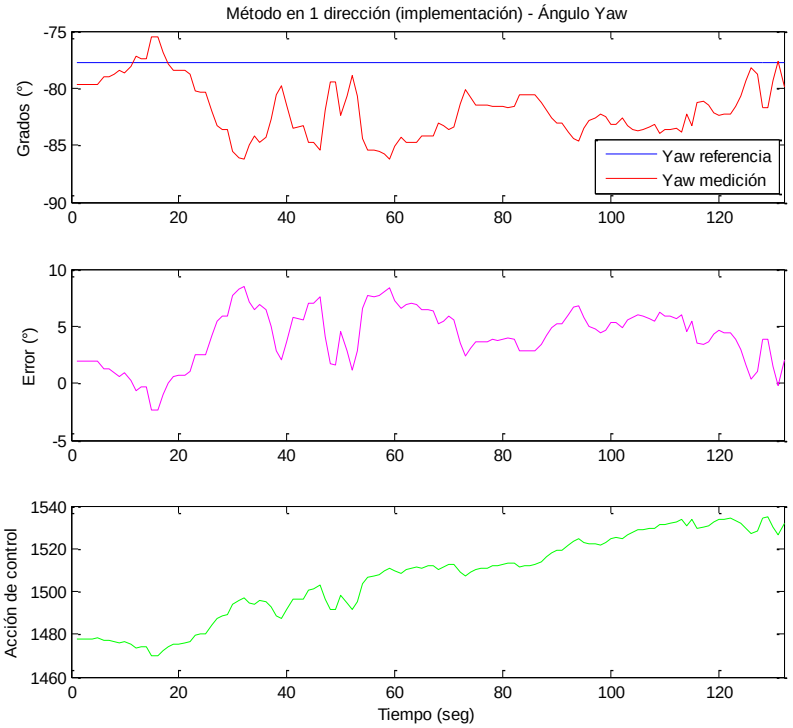



Figura 4.36. Pruebas realizadas en campo, datos del ángulo *yaw*.
Fuente: Elaboración propia.

La figura 4.36 muestra el error del ángulo *yaw* que también es considerable. Es importante destacar que la acción de control se realiza en este ángulo, por ello este método depende de que el error del ángulo *yaw* sea minimizado para no obtener errores considerables en la longitud y la latitud como se observó en la figura 4.35.

Realizando las pruebas utilizando el método de dos direcciones se pudo obtener mejores resultados en lo que se refiere a precisión, debido al control ejecutado en los dos ejes además del existente en el ángulo yaw, esto se puede observar en las figuras 4.37, 4.38 y 4.39.

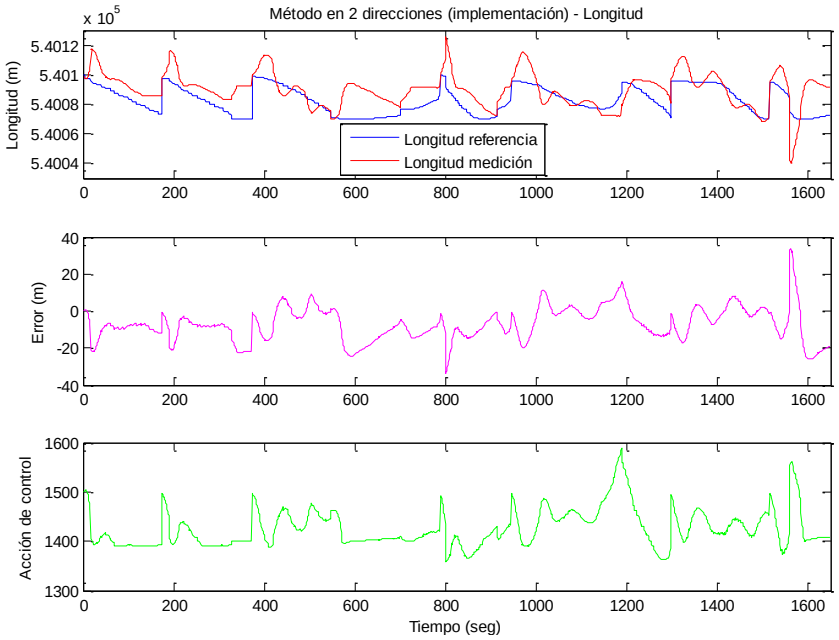



Figura 4.37. Pruebas realizadas en campo, datos de la longitud.
Fuente: Elaboración propia.

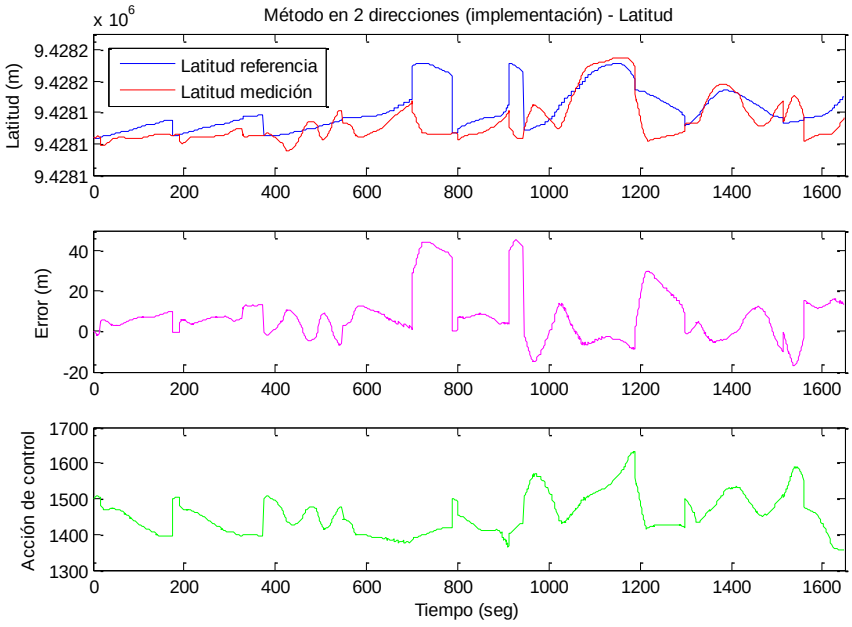



Figura 4.38. Pruebas realizadas en campo, datos de la latitud.
Fuente: Elaboración propia.

Se tiene en ambas gráficas de la figura 4.37, 4.38 y 4.39 la referencia, medición del dato, error y la acción de control. Los resultados obtenidos en las figuras 4.37 y 4.38 son de los puntos mencionados en la tabla 4.2, se debe considerar en lo que respecta al error debido a la precisión del *GPS* utilizado y las perturbaciones como el viento, ocurridos durante las pruebas.

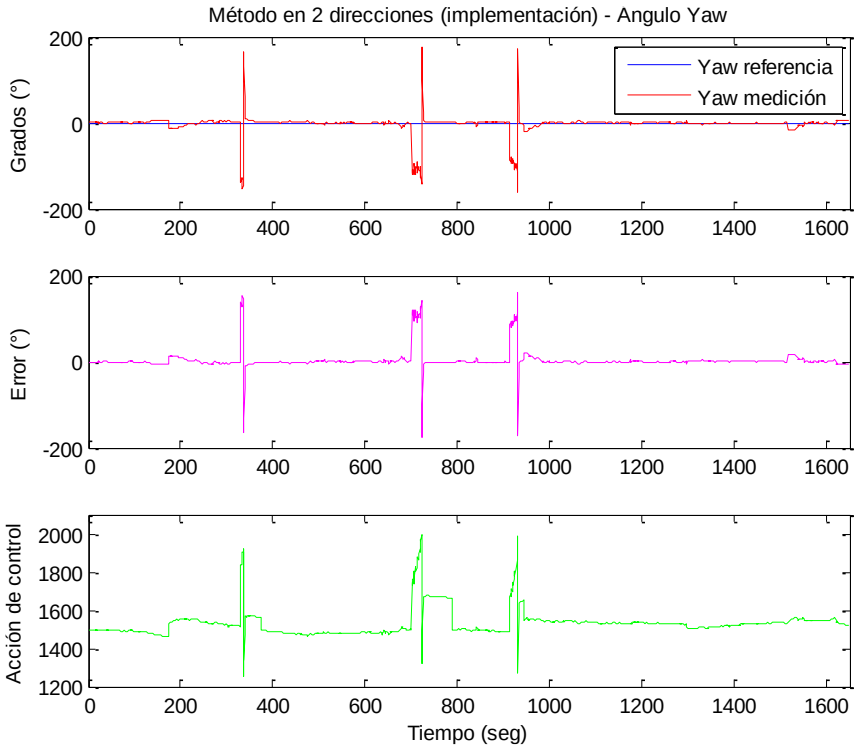



Figura 4.39. Pruebas realizadas en campo, datos del ángulo *yaw*.
Fuente: Elaboración propia.

Y por último los resultados en el control del ángulo *yaw*, son buenas, tener en cuenta que la acción de control no debe saturar al actuador, como se mencionó anteriormente los valores mínimo y máximo en señal *PWM* son entre 1000 y 2000 mseg respectivamente como se muestra en la figura 4.39.

### 4.3. Síntesis de resultados

La identificación, obtención y validación del modelo para la simulación fue el adecuado de acuerdo al valor FIT=85% obtenido para luego realizar la sintonización de los parámetros del controlador PID el cual se ajustó partiendo de los valores obtenidos con el auto-tuning de MATLAB.

Para el método en una dirección al tener sólo un controlador PID (para el ángulo *yaw*) y no existir un control para los ejes de referencia se tuvo un mayor error en la posición del cuadricóptero durante las pruebas de simulación y de campo. En las pruebas del segundo método en dos direcciones, además del control del ángulo yaw se cuenta con un controlador PID para cada eje $x$ y $y$ esto supone un mejor desempeño del sistema de control y los resultados obtenidos así lo demostraron. No obstante para las pruebas en simulación se tuvieron buenos resultados utilizando ambos métodos (en una dirección y en dos

direcciones), considerando el error del *GPS* así como el generado en el algoritmo de transformación de coordenadas otro detalle a incluir son las cortas distancias en las qu se realizaron las pruebas como se observa en la tabla 4.2 las coordenadas geodésicas muestran variación a partir de las milésimas de los datos obtenidos.

Para la etapa de implementación se tuvo que realizar una ligera modificación, sobre todo en lo que se refiere a las señales de salida (control de los motores) ya que para evitar el efecto *windup*, se tuvo que acotar la parte integral del controlador PID. También se tuvo que utilizar el mando radio frecuencia para VANTs, ya que permite controlar de forma manual el cuadricóptero en caso de cualquier desperfecto del sistema de control y también para seleccionar los modos de vuelo. Antes de las pruebas de campo se tuvo que hacer algunas pruebas en un banco de pruebas para evitar errores de código, modos de vuelo del *ArduCopter*, así como conocer los valores máximos y mínimos de las señales de control.

Durante las pruebas en campo se tuvo que trabajar con valores mucho menores a los valores máximos (evitando los valores *PWM* cercanos a 2000 o 1000 mseg) para evitar que el cuadricóptero realice movimientos muy violentos y sufra algún daño, como sucedió durante las primeras pruebas.

# Conclusiones

Se construyó un cuadricóptero prototipo para realizar las diferentes pruebas en un banco de pruebas así como en un espacio libre (campo deportivo de la universidad). Se logró controlar el cuadricóptero usando *ROS* y el *ArduCopter* mediante el código generado y los paquetes `hector_quadrotor` y `roscopter` disponibles en los repositorios.

En la etapa de identificación y validación del modelo obtenido, este se utilizó para la obtención de los parámetros del controlador para las pruebas en simulación, lo que respecta al cuadricóptero real, su identificación resultó más complicado por el tema de sensores como medición de la velocidad lineal y el error presente en los datos de posición de las coordenadas, pero en base a otros trabajos y la información proporcionado por *ArduCopter*, se pudó encontrar los parámetros del controlador PID aceptables.

En lo que se refiere a la parte de software se tuvieron algunos problemas durante el desarrollo del proyecto como aprender y familiarizarse con *ROS* y Ubuntu en poco tiempo debido a variaciones entre versiones *ROS* y la incompatibilidad entre versiones de *ROS* y versiones de Ubuntu por eso su dificultad de aprenderlo. En la parte de los algoritmos creados, para el método en una dirección sólo existe un control PID en el ángulo *yaw* y no en la dirección hacia donde avanza (hacia adelante), por eso en los datos obtenidos se observó mayor error de posición en longitud y latitud, para futuros trabajos se podría implementar un controlador adicional para ello, por esta razon para un mejor control se recomienda utilizar el otro método de dos direcciones ya que este si cuenta con un controlador PID en cada eje además del controlador para el ángulo *yaw*.

Con este trabajo de investigación e implementación se logró ampliar mis conocimientos sobre *ROS*, *Python* y Ubuntu siendo útiles para futuros trabajos debido a que son softwares de código abierto. Resulta importante mencionar que se ha utilizado un sistema robótico (cuadricóptero) a partir de repositorios y código libre existentes lo que compensó el tiempo

en aprender *ROS* y enfocarme sólo en el algoritmo de contro, es por ello que *ROS* se está convirtiendo en un estándar en lo que se refiere a sistemas robóticos.

También existieron múltiples limitaciones físicas que se encontraron durante el desarrollo del proyecto, por ejemplo el corto alcance de los módulos de radio frecuencia en este caso se pueden recomendar la utilización de módulos *Xbee* que tienen mayor alcance, la falta de precisión del *GPS* y esto resulta importante para un mejor desempeño del sistema de control para trabajos futuros se podría utilizar otro sensor o implementar un algoritmo para disminuir el error como utilizar un filtro de *Kalman* por ejemplo, la utilización de una estación base (laptop) durante las pruebas, esto sumado al limitado alcance de los módulos de radio frecuencia con la estación base ya que este realiza los cálculos y es donde ejecuta el nodo principal (`roscore`) y el nodo de control para evitar esto se podría colocar una microcomputadora (*Raspberrry Pi*) junto al *ArduCopter* (en el mismo cuadricóptero) para que los datos procesados ya no sean enviados por radio frecuencia sino mediante un cable.

Lo más importante es que este trabajo servirá como base importante para futuros trabajos relacionados no sólo para cuadricópteros sino para sistemas robóticos en general por la utilización de la paltaforma *ROS* y el perfil de velocidad trapezoidal, una de las aplicaciones prácticas por ejemplo sería en agricultura que actualmente la universidad viene desarrollando para la obtención de imágenes, con lo que se podría utilizar el método en una dirección, con una cámara en el cuadricóptero aprovechando el movimiento solo hacia adelante y usar el otro método de dos direcciones si se requiere mayor precisión, por eso se desarrollaron ambos métodos. Otro trabajo a futuro sería la de mejorar este sistema de control para evitar obstáculos ya sea mediante visión artificial o utilización de sensores. También podría utilizarse otro tipo de controladores no lineales por ejemplo que presentan mejor desempeño frente al clásico controlador PID dado a las enormes ventajas de trabajar con *ROS* y herramientas con las cuenta para el manejo de matrices como EIGEN.

# Referencias

[1] S. Bouabdallah, A. Noth, and R. Siegwart. "PID vs LQ control techniques applied to an indoor micro quadrotor." Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on. Vol. 3. IEEE, 2004.

[2] L. R. G. Carrillo, A. E. D. López, R. Lozano, and C. Pégard. "Modeling the quad-rotor mini-rotorcraft." Quad Rotorcraft Control. Springer London, 2013. 23-34.

[3] G. Vianna. "Modelado y control de un helicóptero Quadrotor". PhD thesis, Tesis (Master), universidad de Sevilla, programa de posgraduación en ingenierías, pp12, 2007.

[4] ArduCopter, "Copter Home". [En línea]. Available: http://ardupilot.org/copter/. [Último acceso: 25 marzo 2017].

[5] ROS, "Robotic Operating System". [En línea]. Available: http://wiki.ros.org/. [Último acceso: 2 febrero 2017].

[6] G. Hoffmann, S. Waslander, and C. Tomlin. "Quadrotor helicopter trajectory tracking control." AIAA guidance, navigation and control conference and exhibit. 2008.

[7] D. Schermuk. "Diseño e implementación de un controlador para la orientación de un quadrotor". Facultad de Ingeniería de la Universidad de Buenos Aires, 2012.

[8] S. Bouabdallah. Design and control of quadrotors with application to autonomous flying. Diss. Ecole Polytechnique Federale de Lausanne, 2007.

[9] Cotticia, Alberto, and Luciano Surace. "Bolletino Di Geodesia E Science Affini."

[10] G. Ortiz. "Aprenda a convertir coordenadas geográficas en UTM y UTM en geográficas". [En línea]. Available: http: //www.gabrielortiz.com/index.asp?Info=058a. [Último acceso: 30 enero 2017].

[11] J. Feng and Q. Peng. "Speed control in path motion." Wuhan-DL tentative. International Society for Optics and Photonics, 1996.

[12] H. Goldstein. Mecánica clásica. Reverté, 1987.

[13] A. Barrientos, L. F. Peñín, C. Balaguer, and R. Aracil. Fundamentos de robótica, volume 256. McGraw-Hill, 1997.

[14] S. Ceriani and M. Miglianvacca. Middleware in robotics. Internal Report for "Advanced Methods of Information Technology for Autonomous Robotics". [En línea]. Available: http://home. deib. polimi. it/gini/AdvancedRobotics/docs/CerianiMigliavacca. pdf [Último acceso: 12 diciembre 2016].

[15] P. Iñigo-Blasco, F. Diaz-del Rio, M. C. Romero-Ternero, D. Cagigas- Muñiz, and S. Vicente-Diaz. "Robotics software frameworks for multi-agent robotic systems development." Robotics and Autonomous Systems 60.6 (2012): 803-821.

[16] ORCA, "Open Robot Control Architecture". [En línea]. Available: http://orca robotics.sourceforge.net/. [Último acceso: 20 diciembre 2016].

[17] OROCOS, "Open Robot Control Software". [En línea]. Available: http://www.orocos.org/. [Último acceso: 20 diciembre 2016].

[18] YARP, "Yet Another Robot Platform". [En línea]. Available: http://www.yarp.it. [Último acceso: 20 diciembre 2016].

[19] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. "ROS: an open-source Robot Operating System." ICRA workshop on open source software. Vol. 3. No. 3.2. 2009.

[20] ROS wiki, "Robotic Operating System wiki". [En línea]. Available: http://wiki.ros.org/. [Último acceso: 20 febrero 2017].

[21] M. Quigley, B. Gerkey, and W. D. Smart. Programming Robots with ROS: a practical introduction to the Robot Operating System. O'Reilly Media, Inc.", 2015.

[22] A. Martinez and E. Fernández. Learning ROS for robotics programming. Packt Publishing Ltd, 2013.

[23] L. Joseph. Learning Robotics Using Python. Packt Publishing Ltd, 2015.

[24] ROS Tutorials, "ROS/Tutorials". [En línea]. Available: http://wiki.ros.org/ROS/Tutorials. [Último acceso: 12 enero 2017].

[25] G. V. Raffo. "Robust control strategies for a quadrotor helicopter: An underactuated mechanical system". PhD thesis, Universidad de Sevilla, 2011.

[26] S. Bouabdallah, P. Murrieri, and R. Siegwart. "Design and control of an indoor micro quadrotor." Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on. Vol. 5. IEEE, 2004.

[27] T. Puls, M. Kemper, R. Küke, and A. Hein. "GPS-based position control and waypoint navigation system for quadrocopters." Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on. IEEE, 2009.

[28] T. Puls, H. Winkelmann, S. Eilers, M. Brucke, and A. Hein. "Interaction of Altitude Control and Waypoint Navigation of a 4 Rotor Helicopter." Advances in Robotics Research (2009): 287-298.

[29] P. F. Rizo Gonzalez and D. Ruiz Restrepo. "Planeación de trayectorias para un robot aéreo ar. drone 2.0 usando gps". B.S. thesis, 2014.

[30] W. Ipanaqué. Control Automático de Procesos. Innovando los procesos productivos. (1era ed.), ISBN (2012): 978-9972.

[31] T. Luukkonen. "Modelling and control of quadcopter". Independent research project in applied mathematics, Espoo, 2011.

[32] L Ljung. The System Identification Toolbox: The Manual. The MathWorks Inc. 1986.

[33] L. Ljung. System Identification. Theory for the user. Prentice Hall. 1987.

[34] Transfer Function Estimation, "tfest". [En línea]. Available: https://www.mathworks.com/help/ident/ref/tfest.html. [Último acceso: 12 marzo 2017].

[35] J. Schoukens y R. Pintelon. Identification of Linear Systems: a Practical Guideline to Accurate Modeling. Pergamon Press, 1991.

[36] L. Meier, "Mavlink: Micro air vehicle communication protocol.". [En línea]. Available: http://qgroundcontrol. org/mavlink/start. [Último acceso: 10 febrero 2017].

[37] J. Wendel, O. Meister, C. Schlaile, and G. F. Trommer. "An integrated gps/mems-imu navigation system for an autonomous helicopter". Aerospace Science and Technology, 10(6):527–533, 2006.

[38] J. Engel, J. Sturm, and D. Cremers. "Accurate figure flying with a quadrocopter using onboard visual and inertial sensing." Imu 320 (2012): 240.

[39] N. Koenig, J. Hsu, M. Dolha and A. Howard, "Gazebo". [En línea]. Available: http: //gazebosim.org/. [Último acceso: 30 noviembre 2016].

[40] D. Hershberger, D. Gossow, and J. Faust, “RViz.” [En línea]. Available: /www.ros.org/wiki/rviz/. [Último acceso: 10 diciembre 2016].

[41] Technische Universität Darmstadt, "Team Heterogeneous Cooperating Team Of Robots", [En línea]. Available: http://www.teamhector.de/. [Último acceso: 15 noviembre 2016].

[42] C. Berzan, "ROS interface for Arducopter using Mavlink 1.0 interface." [En línea]. Available: https://github.com/cberzan/roscopter [Último acceso: 10 febrero 2017].

[43] Monzon, Ivan. "Developing a ROS Enabled Full Autonomous Quadrotor." 2013.

# Apéndice A

## Código para sobre el modelamiento matemático del cuadricóptero

**programa_principal.py**

```
#============MODELO MATEMATICO DE UN CUADRICOPTERO==================
#---------------CHRISTIAN YEYMI MAMANI MAMANI-----------------------
#----------------------PIURA - UDEP 2016----------------------------
#====================PROGRAMA PRINCIPAL============================
from controladores import controlador
from graficos import graficos_quadrotor
from model_quadrotor2 import model_mat_quad
from modo import modo_vuelo
import matplotlib.pyplot as plt
import math
import numpy as np
#________Parametros del Sistema_____________
Ixx = 4.856e-3
Iyy = 4.856e-3
Izz = 8.801e-3
m = 0.468
g = 9.81
l = 0.225
k= 2.98e-6
b= 1.14e-7
Jtp= 6.0e-5
#_________VARIABLES PARA ALMACENAR DATOS__________
tiempo=[]
x = [];y = [] ;z = []
phi= [] ;the = [] ;psi = []
w1=[];w2=[];w3=[];w4=[]
```

```
u1=[];u2=[];u3=[];u4=[]
u=[0,0,0,0]
ud=[0,0,0,0]
w=[0,0,0,0]
phi_degree=[]
the_degree=[]
psi_degree=[]
#-------Periodo de muestreo
dt=0.001
i=0
iterac=6000
#__________MODO DE VUELO__________
autonomo=0
angulo_psi_referencia=0
#-------Asistido---> Referencias de angulos y altura
angulo_phi_referencia=0
angulo_theta_referencia=0
altura_referencia=0
#-------Autonomo
retorno=1
#-------Seleccion de trayectoria (Autonomo) o Angulos (Asistido)
if autonomo==1:
    x_sp=waypoints[:,0]
    y_sp=waypoints[:,1]
    z_sp=waypoints[:,2]
    psi_sp=angulo_psi_referencia
else:
    phi_sp=angulo_phi_referencia
    the_sp=angulo_theta_referencia
    psi_sp=angulo_psi_referencia
    z_sp=altura_referencia
#________PARAMETROS DE LOS CONTROLADORES__________
#-------Modo de vuelo asistido
PID_phi=[6,0,1.75,0,0,0,0]
PID_the=[6,0,1.75,0,0,0,0]
PID_psi=[6,0,1.75,0,0,0,0]
PID_z=[1.5,0,2.5,0,0,0,0]
#-------Modo de vuelo autonomo
PD_x=[1.85,0.75,1,0,0,0,0]
PD_y=[8.55,0.75,1,0,0,0,0]
PD_z=[1.85,0.75,1,0,0,0,0]
PD_phi=[3,0.75,0,0,0,0,0]
PD_the=[3,0.75,0,0,0,0,0]
PD_psi=[3,0.75,0,0,0,0,0]

#________Condiciones iniciales_____________
x0 = 0;y0 = 0;z0 = 1                #Posicion inicial
phi0 = 10;the0 = 10;psi0 = 10       #Angulo inicial

xdot0 = 0;ydot0 = 0;zdot0 = 0       #Velocidad inicial
phi0=math.radians(phi0)
the0=math.radians(the0)
psi0=math.radians(psi0)
p0 = 0;q0 = 0;r0 = 0
phidot0 = 0;thedot0 = 0;psidot0 = 0     #Velocidad angular inicial
estados=[x0,y0,z0,phi0,the0,psi0,p0,q0,r0,xdot0,ydot0,zdot0,phidot0,thedo
t0,psidot0]
#============================================================
quadrotor=model_mat_quad(dt,Ixx,Iyy,Izz,m,g,l,Jtp,k,b)
tipo_control=controlador(dt)
control=modo_vuelo(Ixx,Iyy,Izz,m,g,autonomo)
```

```
#----------------------BUCLE PRINCIPAL-------------------------
while (i<iterac):
    if autonomo==1:
        # Control de posicion

[ud[0],PD_x]=tipo_control.control_posicion(x_sp[i],estados[0],PD_x)

[ud[1],PD_y]=tipo_control.control_posicion(y_sp[i],estados[1],PD_y)

[ud[2],PD_z]=tipo_control.control_posicion(z_sp[i],estados[2],PD_z)

        # Hallando los angulos de referencia
        [phi_sp,the_sp]=control.angulos_referencia(estados, ud)
        # Control de angulos
        [u[1],PD_phi]=tipo_control.control_pid(phi_sp,estados[3],PD_phi)
        [u[2],PD_the]=tipo_control.control_pid(the_sp,estados[4],PD_the)
        [u[3],PD_psi]=tipo_control.control_pid(psi_sp,estados[5],PD_psi)
    else:
        [u[0],PID_z]=tipo_control.control_pid(z_sp,estados[2],PID_z)

[u[1],PID_phi]=tipo_control.control_pid(phi_sp,estados[3],PID_phi)

[u[2],PID_the]=tipo_control.control_pid(the_sp,estados[4],PID_the)

[u[3],PID_psi]=tipo_control.control_pid(psi_sp,estados[5],PID_psi)

    u=control.control_quadrotor(estados, ud,u)
    w=quadrotor.velocidad_ang(u)
    estados=quadrotor.dinamica_quad(estados,u,w)

    tiempo.append(i*dt)
    x.append(estados[0])
    y.append(estados[1])
    z.append(estados[2])
    phi.append(estados[3])
    the.append(estados[4])
    psi.append(estados[5])
    w1.append(w[0])
    w2.append(w[1])
    w3.append(w[2])
    w4.append(w[3])
    u1.append(u[0])
    u2.append(u[1])
    u3.append(u[2])
    u4.append(u[3])

    phi_degree.append(math.degrees(estados[3]))
    the_degree.append(math.degrees(estados[4]))
    psi_degree.append(math.degrees(estados[5]))
    i=i+1

#================================================================
graficar=graficos_quadrotor(tiempo)
graficar.grafico_posicion(1,x, y, z)
graficar.grafico_angulo(2, phi_degree, the_degree, psi_degree)
graficar.grafico_pos3d(3, x, y, z)
graficar.grafico_w(4, w1, w2, w3, w4)
graficar.grafico_control(5,u1, u2, u3, u4)
plt.show()
```

**controladores.py**

```
import math
class controlador:

    def __init__(self,Ts):
        self.Ts=Ts
    def control_pid(self,referencia,valor_real,PID_para):
        Kp=PID_para[0]
        Ki=PID_para[1]
        Kd=PID_para[2]
        last_u=PID_para[6]
        last_error=PID_para[5]
        error=referencia-valor_real
        up=Kp*error
        ui=last_u+Ki*(self.Ts*error)
        ud=Kd*(error-last_error)/self.Ts
        ut=up+ud#+ui
        PID_para[5]=error
        PID_para[6]=ut
        return ut,PID_para

    def control_posicion(self,referencia,valor_real,PD_para):
        Kp=PD_para[0]
        Kd=PD_para[1]
        Kdd=PD_para[2]
        last_error=PD_para[5]
        last_errordot=PD_para[6]
        error=referencia-valor_real
        errordot=(error-last_error)/self.Ts
        errorddot=(errordot-last_errordot)/self.Ts
        up=Kp*error
        ud=Kd*errordot
        udd=Kdd*errorddot
        ut=up+ud+udd
        PD_para[5]=error
        PD_para[6]=errordot
        return ut,PD_para
```

**graficos.py**

```
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import math
class graficos_quadrotor:

    def __init__(self,array_tiempo):
        self.tiempo=array_tiempo

    def grafico_posicion(self,n,posx,posy,posz):
        fig = plt.figure(n)
        plt.suptitle("Movimiento en los ejes x-y-z")
        plt.subplot(3,1,1)
        plt.plot(self.tiempo,posx,'b')
        plt.xlabel('Tiempo (s)')
        plt.ylabel('x (m)')
        plt.subplot(3,1,2)
        plt.plot(self.tiempo,posy,'g')
        plt.xlabel('Tiempo (s)')
```

```
        plt.ylabel('y (m)')
        plt.subplot(3,1,3)
        plt.plot(self.tiempo,posz,'r')
        plt.xlabel('Tiempo (s)')
        plt.ylabel('z (m)')

    def grafico_w(self,n,w1,w2,w3,w4):
        fig = plt.figure(n)
        plt.suptitle("Velocidad angular de cada rotor")
        plt.subplot(4,1,1)
        plt.plot(self.tiempo,w1,'b')
        plt.xlabel('Tiempo (s)')
        plt.ylabel('W1')
        plt.subplot(4,1,2)
        plt.plot(self.tiempo,w2,'g')
        plt.xlabel('Tiempo (s)')
        plt.ylabel('W2')
        plt.subplot(4,1,3)
        plt.plot(self.tiempo,w3,'r')
        plt.xlabel('Tiempo (s)')
        plt.ylabel('W3')
        plt.subplot(4,1,4)
        plt.plot(self.tiempo,w4,'y')
        plt.xlabel('Tiempo (s)')
        plt.ylabel('W4')

    def grafico_angulo(self,n,ang_phi,ang_the,ang_psi):
        fig = plt.figure(n)
        plt.suptitle("Angulos Tait-Bryan")
        plt.subplot(3,1,1)
        plt.plot(self.tiempo,ang_phi,'b')
        plt.xlabel('Tiempo (s)')
        plt.ylabel('phi (deg)')
        plt.subplot(3,1,2)
        plt.plot(self.tiempo,ang_the,'g')
        plt.xlabel('Tiempo (s)')
        plt.ylabel('theta (deg)')
        plt.subplot(3,1,3)
        plt.plot(self.tiempo,ang_psi,'r')
        plt.xlabel('Tiempo (s)')
        plt.ylabel('psi (deg)')

    def grafico_control(self,n,u1,u2,u3,u4):
        fig = plt.figure(n)
        plt.suptitle("Entradas de control del Sistema")
        plt.subplot(4,1,1)
        plt.plot(self.tiempo,u1,'b')
        plt.xlabel('Tiempo (s)')
        plt.ylabel('U1')
        plt.subplot(4,1,2)
        plt.plot(self.tiempo,u2,'g')
        plt.xlabel('Tiempo (s)')
        plt.ylabel('U2')
        plt.subplot(4,1,3)
        plt.plot(self.tiempo,u3,'r')
        plt.xlabel('Tiempo (s)')
        plt.ylabel('U3')
        plt.subplot(4,1,4)
        plt.plot(self.tiempo,u4,'y')
        plt.xlabel('Tiempo (s)')
        plt.ylabel('U4')
```

```
    def grafico_pos3d(self,n,posx,posy,posz):
        fig = plt.figure(n)
        plt.suptitle("Movimiento en el espacio tridimensional")
        ax= fig.gca(projection='3d')
        ax.plot(posx,posy,posz,label='Posicion en xyz')
        ax.set_xlabel('Posicion x (m)')
        ax.set_ylabel('Posicion y (m)')
        ax.set_zlabel('Posicion z (m)')
```

**model_quadrotor2.py**

```
import numpy as np
import math
class model_mat_quad:

    def __init__(self,dt,Ixx,Iyy,Izz,m,g,l,Jtp,k,b):
        self.dt=dt
        self.Ixx=Ixx
        self.Iyy=Iyy
        self.Izz=Izz
        self.m=m
        self.g=g
        self.l=l
        self.Jtp=Jtp
        self.k=k
        self.b=b

    def velocidad_ang(self,u):
        def acotar(v,minimo,maximo):
            if v<minimo:
                valor=0
                return valor
            elif v>maximo:
                valor=maximo
                return valor
            else:
                valor=v
                return valor
        w1 = math.sqrt(acotar(((u[0]/(4*self.k))-
(u[2]/(2*self.k*self.l))-(u[3]/(4*self.b))),0,1e6))
        w2 = math.sqrt(acotar(((u[0]/(4*self.k))-
(u[1]/(2*self.k*self.l))+(u[3]/(4*self.b))),0,1e6))
        w3 =
math.sqrt(acotar(((u[0]/(4*self.k))+(u[2]/(2*self.k*self.l))-
(u[3]/(4*self.b))),0,1e6))
        w4 =
math.sqrt(acotar(((u[0]/(4*self.k))+(u[1]/(2*self.k*self.l))+(u[3]/(4*sel
f.b))),0,1e6))
        Wt = w1-w2+w3-w4
        w=[w1,w2,w3,w4,Wt]
        return w

    def dinamica_quad(self,state,u,w):
        x=state[0]
        y=state[1]
        z=state[2]
        phi=state[3]
        the=state[4]
        psi=state[5]
        p=state[6]
        q=state[7]
```

```
        r=state[8]
        xdot=state[9]
        ydot=state[10]
        zdot=state[11]
        phidot=state[12]
        thedot=state[13]
        psidot=state[14]

        #Dinamica Traslacional
xddot=(math.cos(psi)*math.sin(the)*math.cos(phi)+math.sin(psi)*math.sin(p
hi))*(u[0]/self.m)
        yddot=(math.sin(psi)*math.sin(the)*math.cos(phi)-
math.cos(psi)*math.sin(phi))*(u[0]/self.m)
        zddot=(math.cos(the)*math.cos(phi))*(u[0]/self.m)-self.g
        xdot= xdot + xddot*self.dt
        x= x + xdot*self.dt
        ydot= ydot + yddot*self.dt
        y= y + ydot*self.dt
        zdot= zdot + zddot*self.dt
        z= z + zdot*self.dt

        #Dinamica Rotacional
        pdot=((self.Iyy-self.Izz)*q*r-(self.Jtp*q*w[4])+(u[1]))/self.Ixx
        qdot=((self.Izz-self.Ixx)*p*r+(self.Jtp*p*w[4])+(u[2]))/self.Iyy
        rdot=((self.Ixx-self.Iyy)*p*q+u[3])/self.Izz
        p= p + pdot*self.dt
        q= q + qdot*self.dt
        r= r + rdot*self.dt


phidot=p+q*math.sin(phi)*math.tan(the)+r*math.cos(phi)*math.tan(the)
        thedot=q*math.cos(phi)-r*math.sin(phi)

psidot=q*math.sin(phi)/math.cos(the)+r*math.cos(phi)/math.cos(the)
        phi= phi + phidot*self.dt
        the= the + thedot*self.dt
        psi= psi + psidot*self.dt


estados=[x,y,z,phi,the,psi,p,q,r,xdot,ydot,zdot,phidot,thedot,psidot]
        return estados
```

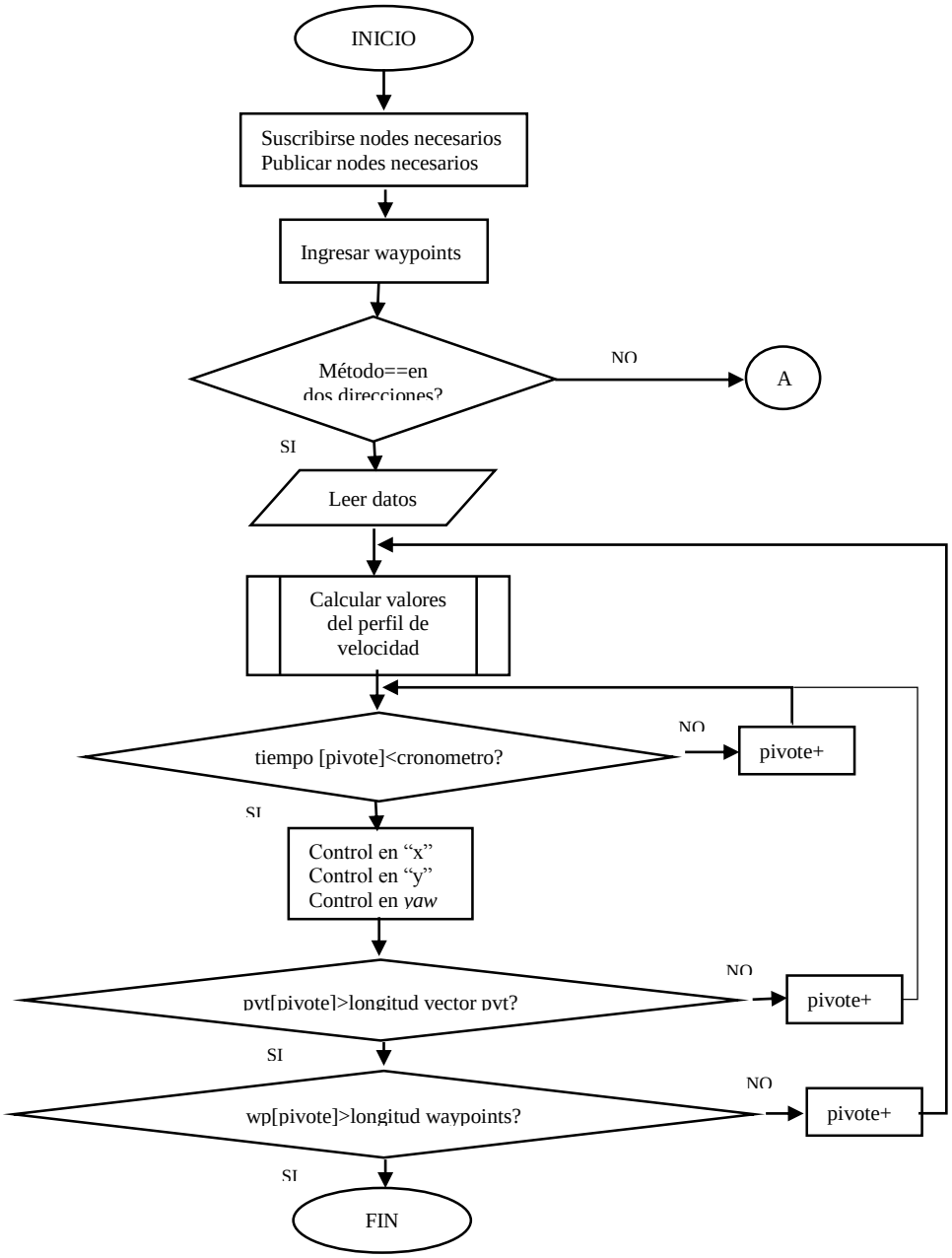
**modo.py**

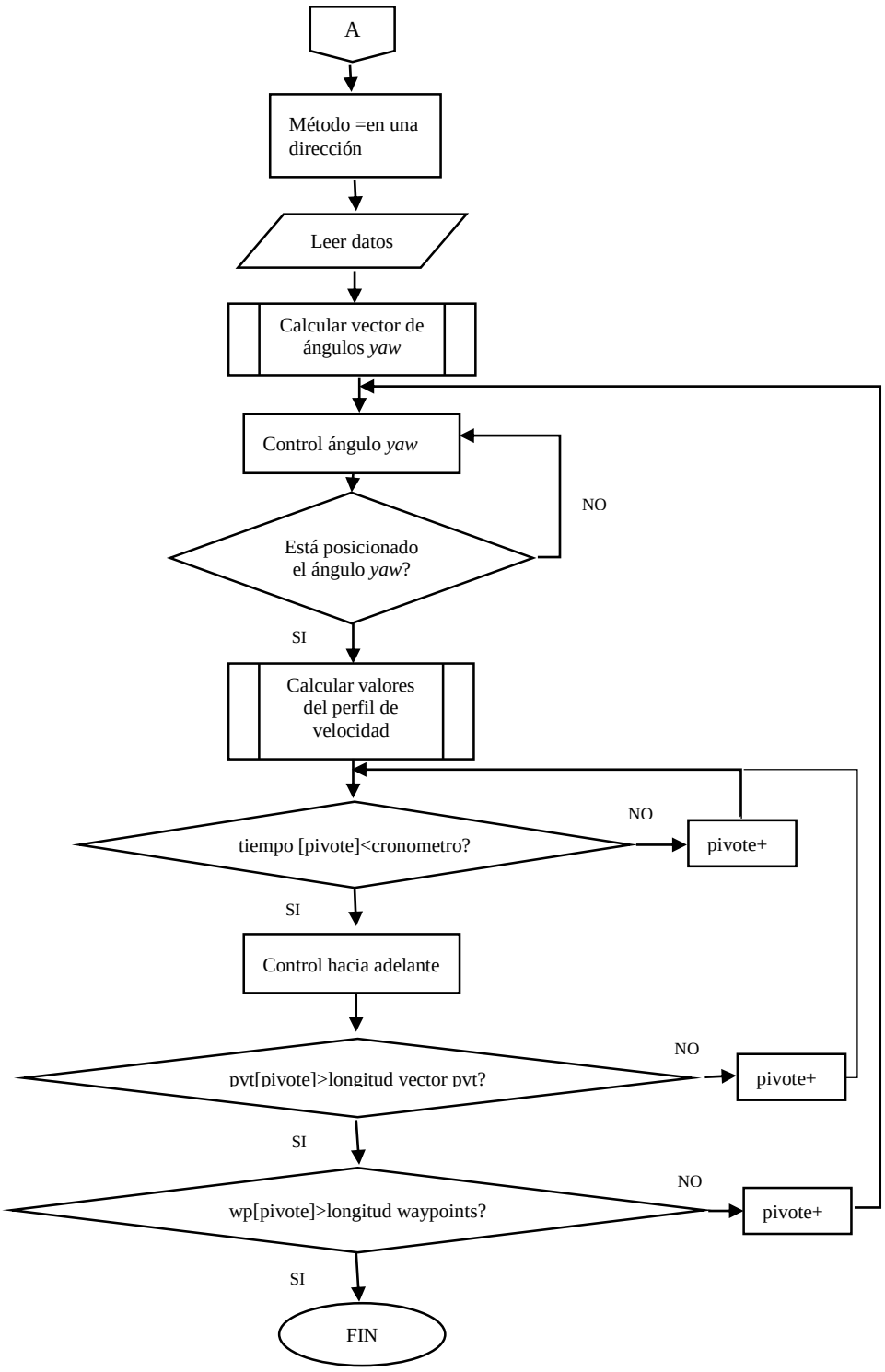
```
import math
class modo_vuelo:

    def __init__(self,Ixx,Iyy,Izz,m,g,autonomo):
        self.Ixx=Ixx
        self.Iyy=Iyy
        self.Izz=Izz
        self.m=m
        self.g=g
        self.autonomo=autonomo

    def control_quadrotor(self,estados,ud,u):
        if self.autonomo==0:

u[0]=(u[0]+self.g)*(self.m/(math.cos(estados[3])*math.cos(estados[4])))
        else:
```

```
a=math.sin(estados[4])*math.cos(estados[5])*math.cos(estados[3])+math.sin
(estados[5])*math.sin(estados[3])

b=math.sin(estados[4])*math.sin(estados[5])*math.cos(estados[3])-
math.cos(estados[5])*math.sin(estados[3])
            c=math.cos(estados[4])*math.cos(estados[3])
            u[0]=self.m*(ud[0]*a+ud[1]*b+(ud[2]+self.g)*c)
        u[1]=u[1]*self.Ixx
        u[2]=u[2]*self.Iyy
        u[3]=u[3]*self.Izz
        return u

    def angulos_referencia(self,state,ud):
        phi=state[3]
        the=state[4]
        psi=state[5]
        dx=ud[0]
        dy=ud[1]
        dz=ud[2]
        phi_sp=math.asin((dx*math.sin(psi)-
dy*math.cos(psi))/(dx**2+dy**2+(dz+self.g)**2))
        the_sp=math.atan((dx*math.cos(psi)+dy*math.sin(psi))/(dz+self.g))

        def acotar_angulo(angulo):
            ref=math.radians(20)
            if angulo<-ref:
                angulo_ref=-ref
                return angulo_ref
            elif angulo>ref:
                angulo_ref=ref
                return angulo_ref
            else:
                angulo_ref=angulo
                return angulo_ref
        angulos=[phi_sp,the_sp]
        return angulos
```

# Apéndice B

## Diagrama de flujo del algoritmo de control utilizado


INICIO
Suscribirse nodes necesarios
Publicar nodes necesarios
Ingresar waypoints
Método==en dos direcciones?
NO
A
SI
Leer datos
Calcular valores del perfil de velocidad
tiempo [pivote]<cronometro?
NO
pivote+
SI
Control en “x”
Control en “y”
Control en yaw
pvt[pivote]>longitud vector pvt?
NO
pivote+
SI
wp[pivote]>longitud waypoints?
NO
pivote+
SI
FIN


A
Método =en una dirección
Leer datos
Calcular vector de ángulos yaw
Control ángulo yaw
Está posicionado el ángulo yaw?
NO
SI
Calcular valores del perfil de velocidad
tiempo [pivote]<cronometro?
NO
pivote+
SI
Control hacia adelante
pvt[pivote]>longitud vector pvt?
NO
pivote+
SI
wp[pivote]>longitud waypoints?
NO
pivote+
SI
FIN

# Apéndice C

## Código para la identificación del modelo usando el paquete `hector_quadrotor`

```
#=======SISTEMA DE CONTROL PARA VUELO AUTONOMO QUADROTOR - OUTDOOR=======
#---------------------CHRISTIAN YEYMI MAMANI MAMANI----------------------
#---------------------------PIURA - UDEP 2017----------------------------
#-----------------------IDENTIFICACION DEL SISTEMA-----------------------
#!/ usr/bin/env python
from __future__ import print_function
import rospy
from std_msgs.msg import String
from geometry_msgs.msg import Twist,PoseStamped,Vector3Stamped
import sys,struct,time,os
import math, random, argparse
#--------------------Opciones de menu-----------------------
parser = argparse.ArgumentParser(description='Algoritmo para la
identificacion del sistema usando señal PRBS')
parser.add_argument('-f',     dest ="configuracion",        help ="Pruebas
de angulo: prueba",     default ="prueba")
parser.add_argument('-v',     dest ="angulo",                    help
="Angulo: roll, pitch",       default ="roll")
parser.add_argument('-m',     dest ="muestras", type =int,  help ="Numero
de muestras")
parser.add_argument('-w',     dest ="maximo",         type =int,  help
="Angulo maximo")
opts=parser.parse_args()
#============================================================
#----------------------Lectura de datos----------------------
def leer_posicion(data):
      global pos_x,pos_y
      pos_x = data.pose.position.x
      pos_y= data.pose.position.y
```

```
def leer_velocidad(data):
      global vel_x,vel_y
      vel_x=data.vector.x
      vel_y=data.vector.y
#======================Programa Principal==================
if opts.configuracion == "prueba":
      def leer_m_angulos(data):
            global ang_roll,ang_pitch,ang_psi,i
            ang_roll=math.degrees(data.vector.x)
            ang_pitch=math.degrees(data.vector.y)
            if i<=opts.muestras:
                  i=i+1
                  if opts.angulo=="roll":
                        h=random.randint(0,1)
                        move.linear.x=h
                  else:
                        h=random.randint(0,1)
                        move.linear.y=h
                  inp_control.publish(move)
                  print ("Angulo Roll: ",ang_roll)
                  print ("Angulo Pitch: ",ang_pitch)
                  print ("Velocidad x:",vel_x)
                  print ("Velocidad y:",vel_y)
                  print ("Posicion x:",pos_x)
                  print ("Posicion y:",pos_y)
                  print ("Velocidad",h)
                  print ("Muestra",i)
                  print ("--------------------------")
                  f= open('identificacion','a')
                  tiempo = time.time()
                  medicion_datos = {'posicion_x':pos_x,
'posicion_y':pos_y, 'vel_x':vel_x, 'vel_y':vel_y,}
                  medicion_control = {'accion_control':h,
'angulo_roll':ang_roll, 'angulo_pitch':ang_pitch,'tiempo':tiempo}
                  f.write (" %(posicion_x)0.3f %(posicion_y)0.3f
%(vel_x)0.3f %(vel_y)0.3f" % medicion_datos)
                  f.write (" %(accion_control)0.3f %(angulo_roll)0.3f
%(angulo_pitch)0.3f %(tiempo)d \n" % medicion_control)
                  rospy.sleep(1)
      #-------------------Constantes---------------------
      i=0
      #--------Suscripcion a los topicos necesarios------
      rospy.Subscriber('pose', PoseStamped, leer_posicion)
      rospy.Subscriber('euler', Vector3Stamped, leer_m_angulos)
      rospy.Subscriber('velocity', Vector3Stamped, leer_velocidad)
#------------Suscripcion al topico cmd_vel para control------------
inp_control = rospy.Publisher('cmd_vel', Twist, queue_size=10)
move=Twist()
def parar():
      inp_control.publish(Twist())
      print ("Nodo terminado")
#----------------------Bucle Principal-----------------------
def mainloop():
      rospy.init_node ('sistema_identificacion')
      rospy.on_shutdown(parar)
      while not rospy.is_shutdown():
            rospy.sleep (0.001)
if __name__ == '__main__':
      try:
            mainloop ()
      except rospy.ROSInterruptException : pass
```

# Apéndice D

## Código para simulación del sistema de control utilizando coordenadas geodésicas y *UTM*.

```
#=======SISTEMA DE CONTROL PARA VUELO AUTONOMO QUADROTOR - OUTDOOR=======
# ----------------------CHRISTIAN YEYMI MAMANI MAMANI-------------------
# ----------------------------PIURA - UDEP 2017--------------------------
# -------------------------------SIMULACION------------------------------
#======================CONTROL Y PERFIL DE VELOCIDAD=====================
#!/ usr/bin/env python
from __future__ import print_function
import rospy
from std_msgs.msg import String
from geometry_msgs.msg import Twist,PoseStamped,Vector3Stamped
from geographic_msgs.msg import GeoPose
import sys,struct,time,os
import math
import argparse
#--------------------------Opciones de menu-----------------------------
parser = argparse.ArgumentParser(description='Sistema de Control de
trayectoria para cuadricoptero')
parser.add_argument('-f',     dest ="configuracion",        help ="Modos
de vuelo: trayectoria,rotacion",          default ="trayectoria")
parser.add_argument('-c',     dest ="coordenadas",          help ="Tipos
de coordenadas: geodesica,utm",           default ="utm")
parser.add_argument('-m',     dest ="metodo",                     help
="Metodo para Perfil Velocidad Trapezoidal: vectores,angulo",
      default ="angulo")
parser.add_argument('-w',     dest ="arreglo_waypoints",    type =float,
            help ="Waypoints de la trayectoria: long1 lat1 long2 lat2
...", nargs='*')
```

```
parser.add_argument("-r",     dest ="yaw_ref",              type =int,
            help ="Angulo yaw de referencia (deg): yaw1 yaw2 yaw3 ...",
            nargs='*')
opts=parser.parse_args()
#=====================================================================
#...................RUTINA PARA EL CALCULO DE LOS WAYPOINTS DEL PERFIL DE
VELOCIDAD TRAPEZOIDAL.....................
def puntos(punto1,punto2):
      global vk,k,kc
      v_ini=0     # Velocidad inicial
      pt=0.3      # Porcentaje de aceleracion
      vk=1        # Velocidad maxima
      n=30        # Numero de puntos para hallar
      tiempo=[]
      x=[];y=[]

      # Para el calculo de la distancia entre puntos
      def modulo(pto_a,pto_b):
          mod=math.sqrt(((pto_b[0]-pto_a[0])**2)+((pto_b[1]-
pto_a[1])**2))
          return mod
      # Para el calculo de la resultante de las componentes x-y
(velocidad y aceleracion) de un vector de datos
      def modulo2(a,b):
          vtot=[]
          n=len(a)
          for item in range(n):
              pen=math.sqrt(((a[item])**2)+((b[item])**2))
              vtot.append(pen)
          return vtot
      # Para el calculo del punto intermedio P(tao)
      def punto_tao(ace,tao,d,n,m):
          pto_tao=(0.5*ace*(tao)**2)*(m-n)/d+n
          return pto_tao
      # Para el calculo del punto intermedio P(T-tao)
      def punto_T_tao(v,T,tao,d,pto_tao,n,m):
          pto_T_tao=(v*(T-2*tao))*(1/d)*(m-n)+pto_tao
          return pto_T_tao
      # Para el calculo de los segmentos de aceleracion y desaceleracion
constante
      def segmento1(ace,tk,v_ini,d,n,m):
          pto_seg1=(0.5*ace*(tk)**2+v_ini*tk)*((m-n)/d)+n
          return pto_seg1
      # Para el calculo del segmento de velocidad constante
      def segmento2(tk,vk,dsgmnt2,n,m):
          pto_seg2=(vk/dsgmnt2)*tk*(m-n)+n
          return pto_seg2
      # Para el calculo de las derivadas con metodos numericos
      def pendiente(a,t):
          vec=[0]
          n=len(t)
          for ite in range(n-1):
              pen=(a[ite+1]-a[ite])/(t[ite+1]-t[ite])
              vec.append(pen)
          return vec

      d=modulo(punto1, punto2)
      T=d/(vk*(1-pt))
      tao=T*pt
      ace=vk/tao
```

```
      k=int(round(pt*n))  # Numero de puntos en la recta acelerada y
desacelerada
      kc=n-2*k            # Numero de puntos en la recta de velocidad
constante
      # Hallamos P(tao)
      pto_x1=punto_tao(ace,tao,d,punto1[0],punto2[0])
      pto_y1=punto_tao(ace,tao,d,punto1[1],punto2[1])
      # Hallamos P(T-tao)
      pto_x2=punto_T_tao(vk,T,tao,d,pto_x1,punto1[0],punto2[0])
      pto_y2=punto_T_tao(vk,T,tao,d,pto_y1,punto1[1],punto2[1])
      # Puntos intermedios
      pto_tao=[pto_x1,pto_y1]       # P(tao)
      pto_T_tao=[pto_x2,pto_y2]     # P(T-tao)
      # Segmento 1 (Aceleracion constante)
      dsgmnt1=modulo(punto1, pto_tao)
      for i in range(k):
          tseg1=(tao/k)*i
          seg1x=segmento1(ace, tseg1,v_ini,dsgmnt1, punto1[0],
pto_tao[0])
          seg1y=segmento1(ace, tseg1,v_ini,dsgmnt1, punto1[1],
pto_tao[1])
          x.append(seg1x)
          y.append(seg1y)
          tiempo.append(tseg1)
      # Segmento 2 (Velocidad constante)
      dsgmnt2=modulo(pto_tao,pto_T_tao)
      for j in range(kc):
          tseg2=((T-2*tao)/(kc-1))*j
          seg2x=segmento2(tseg2,vk,dsgmnt2,pto_tao[0],pto_T_tao[0])
          seg2y=segmento2(tseg2,vk,dsgmnt2,pto_tao[1],pto_T_tao[1])
          x.append(seg2x)
          y.append(seg2y)
          tiempo.append(tseg2+tao)
      # Segmento 3 (Desaceleracion constante)
      dsgmnt3=modulo(pto_T_tao,punto2)
      for h in range(k):
          tseg3=(tao/k)*(h+1)
          seg3x=segmento1(-ace, tseg3,vk,dsgmnt3, pto_T_tao[0],
punto2[0])
          seg3y=segmento1(-ace, tseg3,vk,dsgmnt3, pto_T_tao[1],
punto2[1])
          x.append(seg3x)
          y.append(seg3y)
          tiempo.append(tseg3+T-tao)
      # Calculo de las velocidades
      vx=pendiente(x, tiempo)
      vy=pendiente(y, tiempo)
      vt=modulo2(vx, vy)
      # Calculo de las aceleraciones
      acex=pendiente(vx, tiempo)
      acey=pendiente(vy, tiempo)
      return x,y,vx,vy,vt,acex,acey,tiempo
#=========================================================================
#-----Algoritmo para el calculo de los angulos para el metodo angulo-----
def calculo_angulos(vector_lon,vector_lat):
      angulos=[]
      n=len(vector_lon)
      for j in range(n-1):
          x=vector_lon[j+1]-vector_lon[j]
          y=vector_lat[j+1]-vector_lat[j]
          ang=math.atan2(y,x)
```

```
            ang_degree=math.degrees(ang)
            angulos.append(ang_degree)
      return angulos
#-----------------------Algoritmo de Control--------------------------
def control(k_xyz,k_yaw,error,last_error,last_ui):
      u=[]; ui=[]
      for i in range(4):
            if i==3:
                  uP= k_yaw[0]*error[i]   # Para el control del angulo
psi
                  uI= k_yaw[1]*error[i]+last_ui[i]
                  uD= k_yaw[2]*(error[i]-last_error[i])
                  ut= uP+uI+uD
            else:
                  uP= k_xyz[0]*error[i]   # Para el control en x-y
                  uI= k_xyz[1]*error[i]+last_ui[i]
                  uD= k_xyz[2]*(error[i]-last_error[i])
                  ut= uP+uI+uD
            u.append(ut)
            ui.append(uI)
      return u,ui
#-------------------Algoritmo conversion de coordenadas-----------------
def geodesicas_to_utm(lon,lat):     # Conversion de coordenadas
geodesicas-> UTM
      lon_rad=lon*math.pi/180
      lat_rad=lat*math.pi/180
      huso=int(lon/6+31)                  # Calculo del huso, solo la
parte entera
      lambda_0=huso*6-183
      delta_lambda=lon_rad-(lambda_0*math.pi/180)
      A=math.cos(lat_rad)*math.sin(delta_lambda)      # Calculo de
parametros
      epsilon=0.5*(math.log((1+A)/(1-A)))
      n=math.atan(math.tan(lat_rad)/math.cos(delta_lambda))-lat_rad
      v=6397376.633466756/math.sqrt(1+0.00676817019657*math.cos(lat_rad)*
*2)
      epsi=0.00676817019657*(epsilon**2)*(math.cos(lat_rad)**2)/2
      A1=math.sin(2*lat_rad)
      A2=A1*(math.cos(lat_rad))**2
      J2=lat_rad+(A1/2)
      J4=(3*J2+A2)/4
      J6=(5*J4+A2*(math.cos(lat_rad))**2)/3
      Bphi=6397376.633466756*(lat_rad-0.005076128*J2+(4.29451198217e-
5)*J4-(1.69551596705e-7)*J6)
      x_utm=epsilon*v*(1+(epsi/3))+500000       # Calculo final de
coordenadas
      if lat<0:
            y_utm=n*v*(1+epsi)+Bphi+1e7
      else:
            y_utm=n*v*(1+epsi)+Bphi
      x_utm=int(x_utm)
      y_utm=int(y_utm)
      return x_utm,y_utm
#----------------------Lectura de datos----------------------
def leer_angulos(data):
      global ang_psi
      ang_psi=data.vector.z
#==========================Programa Principal==========================
#.....................Modo seguidor de trayectoria.....................
if opts.configuracion == "trayectoria":
```

```
      def leer_m_trayectoria_coordenadas(data):# Leer las coordenadas en
tiempo real
            global
wn,pvtn,x_sp,y_sp,vx_sp,vy_sp,vt_sp,ax_sp,ay_sp,t,crono,aux,punto_inicial
            global
k_xyz,k_yaw,wpx,wpy,ui,error,last_error,tolerancia,vector_angulos,aux2,au
x3,aux4
            latitud_geo = data.position.latitude
            longitud_geo = data.position.longitude
            pos=geodesicas_to_utm(longitud_geo,latitud_geo)
            pos_x=pos[0]
            pos_y=pos[1]
            if punto_inicial==0:
                  wpx.insert(0,pos_x)
                  wpy.insert(0,pos_y)
                  if opts.metodo=="angulo":
                        vector_angulos=calculo_angulos(wpy,wpx)
                  punto_inicial=1
            last_error[0]=error[0]
            last_error[1]=error[1]
            last_error[2]=error[2]
            last_error[3]=error[3]
            # Hallamos los valores de los waypoints al inicio y al final
de cada tramo
            if ((wn==0) or (len(x_sp)==pvtn and wn<(len(wpx)-1))):

      [x_sp,y_sp,vx_sp,vy_sp,vt_sp,ax_sp,ay_sp,t]=puntos([wpx[wn],wpy[wn]
],[wpx[wn+1],wpy[wn+1]])
            error[0]=x_sp[pvtn]-pos_x
            error[1]=y_sp[pvtn]-pos_y
            error[2]=0
            last_ui=ui
            tt = rospy.Time.from_sec(time.time())     # Creamos un
cronometro
            if aux==0:        # Reiniciamos el cronometro
                  crono=tt.to_sec()
            ttt=tt.to_sec()-crono
            # ---------------Metodo Perfil de Velocidad Trapezoidal
mediante giro del angulo yaw-------------
            if opts.metodo=="angulo":
                  error[3]=math.radians(-vector_angulos[wn])-ang_psi
                  if aux2==0:                         # Ejecutamos el
control para el angulo yaw

      [ut,ui]=control(k_xyz,k_yaw,error,last_error,last_ui)
                        move.angular.z=ut[3]
                        inp_control.publish(move)
                        print ("Posicionando angulo")
                        print ("Angulo Yaw: ",math.degrees(ang_psi))
                        print ("Vector yaw:",vector_angulos)
                        print ("Referencia yaw:",vector_angulos[wn])
                        print ("Error yaw:",math.degrees(error[3]))
                        print ("Control:",ut[3])
                        print ("Numero de angulo:",wn+1)
                        print ("--------------------------")
                        if ((abs(math.degrees(error[3]))<0.3) and
(wn<(len(vector_angulos)))):
                              inp_control.publish(Twist())
                              print ("Listo ---> Control")
                              aux2=1;           aux3=0
                              rospy.sleep(2)
```

```
                if (aux3==0):                           # Ejecutamos el
control para el tiempo calculado t[]
                        if (t[pvtn]>ttt):
                                aux=1                   # Indicamos que ya
no estamos en t[0]

      [ut,ui]=control(k_xyz,k_yaw,error,last_error,last_ui)
                                move.linear.x=vt_sp[pvtn]
                                move.angular.z=ut[3]    # Angulo yaw
                                inp_control.publish(move)
                                f= open('angulo','a')
                                tiempo = time.time()
                                tiempo_pivote = {'tiempo':tiempo,
'tramos':len(wpx)-1, 'tramo_actual':wn+1, 'cronometro':ttt,
'tiempo_necesario':t[pvtn]}
                                coorde =
{'longitud_geo':data.position.longitude,
'latitud_geo':data.position.latitude}
                                medicion_datos = {'longitud':pos_x,
'latitud':pos_y, 'angulo_yaw':math.degrees(ang_psi)}
                                referencia_datos =
{'longitud_ref':x_sp[pvtn], 'latitud_ref':y_sp[pvtn],
'angulo_yaw_ref':vector_angulos[wn]}
                                senal_error = {'error_longitud':error[0],
'error_latitud':error[1], 'error_angulo_yaw':math.degrees(error[3])}
                                u_control = {'u_roll':move.linear.x,
'u_yaw':move.angular.z}
                                f.write (" %(tiempo)d %(tramos)d
%(tramo_actual)d %(cronometro)0.6f %(tiempo_necesario)0.6f" %
tiempo_pivote)
                                f.write (" %(longitud_geo)0.6f
%(latitud_geo)0.6f" % coorde)
                                f.write (" %(longitud)0.3f %(latitud)0.3f
%(angulo_yaw)0.3f" % medicion_datos)
                                f.write (" %(longitud_ref)0.3f
%(latitud_ref)0.3f %(angulo_yaw_ref)0.3f" % referencia_datos)
                                f.write (" %(error_longitud)0.3f
%(error_latitud)0.3f %(error_angulo_yaw)0.3f" % senal_error)
                                f.write (" %(u_roll)0.4f %(u_yaw)0.4f \n" %
u_control)

                                print ("Angulos:",vector_angulos)
                                print ("Total tramos a recorrer:",len(wpx)-
1)
                                print ("Tramo actual:",wn+1)
                                print ("Numero puntos perfil de
velocidad:",len(x_sp))
                                print ("Pivote perfil velocidad:",pvtn+1)
                                print ("_______Tiempo______")
                                print ("Tiempo que debe tomar:",t[pvtn])
                                print ("Cronometro:", ttt)
                                print ("Medicion x:",pos_x)
                                print ("Medicion y:",pos_y)
                                print ("Referencia x:",x_sp[pvtn])
                                print ("Referencia y:",y_sp[pvtn])
                                print ("Medicion
yaw:",math.degrees(ang_psi))
                                print ("Referencia yaw:",vector_angulos[i])
                                print ("_______Error______")
                                print ("Error x:",error[0])
                                print ("Error y:",error[1])
```

```
                              print ("Velocidad promedio:",vt_sp[pvtn])
                              print ("_______Control______")
                              print ("Control x:",move.linear.x)
                              print ("Control yaw:",move.angular.z)
                              print ("--------------")
                        elif pvtn<len(t)-1:
                              pvtn=pvtn+1
                        elif (wn<len(wpx)-2):
                              wn=wn+1;    aux2=0;           aux3=1;
                              pvtn=0;           aux=0
                              x_sp=[];    vx_sp=[];   ax_sp=[]
                              y_sp=[];    vy_sp=[];   ay_sp=[]
                              t=[];       vt_sp=[];
                              inp_control.publish(Twist())
                              rospy.sleep(2)
                        else:
                              aux2=1;           aux3=1
                              inp_control.publish(Twist())
            # -------Metodo Perfil de Velocidad Trapezoidal mediante
vectores de velocidad en x-y-------
            elif opts.metodo=="vectores":
                  error[3]=math.radians(-90)-ang_psi
                  if aux4==0:
                        if (t[pvtn]>ttt):             # Ejecutamos el
control para el tiempo calculado t[]
                              aux=1                         # Indicamos
que ya no estamos en t[0]

      [ut,ui]=control(k_xyz,k_yaw,error,last_error,last_ui)
                              move.angular.z=ut[3]
                              move.linear.x=vx_sp[pvtn]+ut[0]
                              move.linear.y=vy_sp[pvtn]+ut[1]
                              inp_control.publish(move)
                              f= open('vectores','a')
                              tiempo = time.time()
                              tiempo_pivote = {'tiempo':tiempo,
'tramos':len(wpx)-1, 'tramo_actual':wn+1, 'cronometro':ttt,
'tiempo_necesario':t[pvtn]}
                              coorde =
{'longitud_geo':data.position.longitude,
'latitud_geo':data.position.latitude}
                              medicion_datos = {'longitud':pos_x,
'latitud':pos_y, 'angulo_yaw':math.degrees(ang_psi)}
                              referencia_datos =
{'longitud_ref':x_sp[pvtn], 'latitud_ref':y_sp[pvtn]}
                              senal_error = {'error_longitud':error[0],
'error_latitud':error[1], 'error_angulo_yaw':math.degrees(error[3])}
                              u_control = {'u_roll':move.linear.x,
'u_pitch':move.linear.y, 'u_yaw':move.angular.z}
                              f.write (" %(tiempo)d %(tramos)d
%(tramo_actual)d %(cronometro)0.6f %(tiempo_necesario)0.6f" %
tiempo_pivote)
                              f.write (" %(longitud_geo)0.6f
%(latitud_geo)0.6f" % coorde)
                              f.write (" %(longitud)0.3f %(latitud)0.3f
%(angulo_yaw)0.3f" % medicion_datos)
                              f.write (" %(longitud_ref)0.3f
%(latitud_ref)0.3f" % referencia_datos)
                              f.write (" %(error_longitud)0.3f
%(error_latitud)0.3f %(error_angulo_yaw)0.3f" % senal_error)
```

```
                                f.write (" %(u_roll)0.4f %(u_pitch)0.4f
%(u_yaw)0.4f \n" % u_control)

                                print ("Total tramos a recorrer:",len(wpx)-
1)
                                print ("Tramo actual:",wn+1)
                                print ("Numero puntos perfil de
velocidad:",len(x_sp))
                                print ("Pivote perfil velocidad:",pvtn+1)
                                print ("_______Tiempo______")
                                print ("Tiempo que debe tomar:",t[pvtn])
                                print ("Cronometro:", ttt)
                                print ("_______Medicion______")
                                print ("Medicion x:",pos_x)
                                print ("Medicion y:",pos_y)
                                print ("Referencia x:",x_sp[pvtn])
                                print ("Referencia y:",y_sp[pvtn])
                                print ("_______Comparar______")
                                print ("Pivote Vx:",vx_sp[pvtn])
                                print ("Pivote Vy:",vy_sp[pvtn])
                                print ("_______Error______")
                                print ("Medicion
yaw:",math.degrees(ang_psi))
                                print ("Error x:",error[0])
                                print ("Error y:",error[1])
                                print ("_______Control______")
                                print ("Control x:",move.linear.x)
                                print ("Control y:",move.linear.y)
                                print ("--------------")
                          elif pvtn<len(t)-1:
                                pvtn=pvtn+1
                          elif (wn<len(wpx)-2):
                                wn=wn+1;    pvtn=0;           aux=0;
      t=[]
                                x_sp=[];    vx_sp=[];   ax_sp=[]
                                y_sp=[];    vy_sp=[];   ay_sp=[]
                          else:
                                aux4=1
                                inp_control.publish(Twist())
      #-----------------Variables para modo trayectoria-----------------
      kp_xyz = 0.5;     ki_xyz = 0.0002;  kd_xyz = 0.00005
      kp_yaw = 5;       ki_yaw = 0.0002;  kd_yaw = 0.00005
      k_xyz=[kp_xyz,ki_xyz,kd_xyz];       k_yaw=[kp_yaw,ki_yaw,kd_yaw]
      wpx=[];           wpy=[];           x_sp=[];    y_sp=[];    vt_sp=[]
      vx_sp=[];   vy_sp=[];   ax_sp=[];   ay_sp=[];   t=[];       v=[]
      wn=0;       pvtn=0;           crono=0;    aux=0;            aux2=0;
            aux3=1
      ui=[0,0,0,0];                 error=[0,0,0,0];  last_error=[0,0,0,0]
      lon_wp=[];  lat_wp=[];  punto_inicial=0;  vector_angulos=[]
      tolerancia=0.05;        aux4=0
      #----------Arreglo y conversion de waypoints de referencia---------
      l=len(opts.arreglo_waypoints)
      for i in range(l/2):
          lon=opts.arreglo_waypoints[2*i]
          lat=opts.arreglo_waypoints[2*i+1]
          lon_wp.append(lon)
          lat_wp.append(lat)
      if opts.coordenadas=="geodesica":
            for n in range(len(lat_wp)-1):
                  wp=geodesicas_to_utm(lon_wp[n],lat_wp[n])
                  x_utm=wp[0]
```

```
                  y_utm=wp[1]
                  wpx.append(x_utm)
                  wpy.append(y_utm)
      else:
            wpx=lon_wp
            wpy=lat_wp
      #--------------Suscripcion a los topicos necesarios----------------
      rospy.Subscriber('geopose', GeoPose,
leer_m_trayectoria_coordenadas)
      rospy.Subscriber('euler', Vector3Stamped, leer_angulos)
#.........................Modo control de angulo.....................
elif opts.configuracion == "rotacion":
      def leer_m_rotacion_angulos(data):        # Leer los angulos
            global error,ui,kp,ki,kd,i
            ang_psi=data.vector.z
            if i<(len(opts.yaw_ref)):
                  last_error=error
                  last_ui=ui
                  error=math.radians(opts.yaw_ref[i])-ang_psi
                  up= kp*error
                  ui= ki*error+last_ui
                  ud= kd*(error-last_error)
                  u=up+ui+ud
                  move.angular.z=u
                  inp_control.publish(move)
                  print ("Vector yaw:",opts.yaw_ref)
                  print ("Angulo yaw:",math.degrees(ang_psi))
                  print ("Referencia yaw:",opts.yaw_ref[i])
                  print ("Error yaw:",math.degrees(error))
                  print ("Control:",u)
                  print ("Numero de angulo:",i+1)
                  print ("--------------")
                  if (abs(math.degrees(error))<0.05):
                        i=i+1
                        inp_control.publish(Twist())
                        print ("Siguiente angulo")
                        rospy.sleep(2)
            else:
                  print ("Angulos terminados")
      kp=5;       ki=0.0002;        kd=0.00005
      error = 0;  ui = 0;                 i=0
      rospy.Subscriber('euler', Vector3Stamped, leer_m_rotacion_angulos)

#------------Suscripcion al topico cmd_vel para control---------------
inp_control = rospy.Publisher('cmd_vel', Twist, queue_size=10)
move=Twist()

#================================================================
def parar():
      inp_control.publish(Twist())
      print ("Nodo terminado")
#----------------------Bucle Principal-------------------------
def mainloop():
      rospy.init_node ('control_simulacion')
      rospy.on_shutdown(parar)
      while not rospy.is_shutdown():
            rospy.sleep (0.001)
if __name__ == '__main__':
      try:
            mainloop ()
      except rospy.ROSInterruptException : pass
```

# Apéndice E

## Código para interfaz entre *ROS* y la *ArduCopter.*

roscopter.py

```
#!/usr/bin/env python
import roslib ; roslib.load_manifest ('roscopter')
import rospy
from std_msgs.msg import String , Header
from std_srvs.srv import *
from sensor_msgs.msg import NavSatFix , NavSatStatus , Imu
import roscopter.msg
import sys,struct,time,os

mavlink_dir = os.path.realpath(os.path.join(
    os.path.dirname(os.path.realpath(__file__)),
    '..', 'mavlink'))
sys.path.insert(0, mavlink_dir)

pymavlink_dir = os.path.join(mavlink_dir, 'pymavlink')
sys.path.insert(0, pymavlink_dir)

from optparse import OptionParser
parser = OptionParser("roscopter.py [options]")

parser.add_option("--baudrate", dest="baudrate", type='int',
                  help="master port baud rate", default=57600)
parser.add_option("--device", dest="device", default="/dev/ttyUSB0",
help="serial device")
parser.add_option("--rate", dest="rate", default=10, type='int',
help="requested stream rate")
parser.add_option("--source-system", dest='SOURCE_SYSTEM', type='int',
                  default=255, help='MAVLink source system for this GCS')
```

```
parser.add_option("--enable-control",dest="enable_control",
default=False, help="Enable listning to control messages")

(opts, args) = parser.parse_args()

import mavutil

# create a mavlink serial instance
master = mavutil.mavlink_connection(opts.device, baud=opts.baudrate)

if opts.device is None:
    print("You must specify a serial device")
    sys.exit(1)

def wait_heartbeat(m):
    '''wait for a heartbeat so we know the target system IDs'''
    print("Waiting for APM heartbeat")
    m.wait_heartbeat()
    print("Heartbeat from APM (system %u component %u)" %
(m.target_system, m.target_system))


def send_rc(data):
    master.mav.rc_channels_override_send(
        master.target_system,
        master.target_component,
        data.channel[0],
        data.channel[1],
        data.channel[2],
        data.channel[3],
        data.channel[4],
        data.channel[5],
        data.channel[6],
        data.channel[7])
    print "sending rc: %s" % data

def set_arm(req):
    master.arducopter_arm()
    return True

def set_disarm(req):
    master.arducopter_disarm()
    return True

pub_gps = rospy.Publisher('gps', NavSatFix)
pub_rc = rospy.Publisher('rc', roscopter.msg.RC)
pub_state = rospy.Publisher('state', roscopter.msg.State)
pub_vfr_hud = rospy.Publisher('vfr_hud', roscopter.msg.VFR_HUD)
pub_attitude = rospy.Publisher('attitude', roscopter.msg.Attitude)
pub_raw_imu =  rospy.Publisher('raw_imu', roscopter.msg.Mavlink_RAW_IMU)
if opts.enable_control:
    rospy.Subscriber("send_rc", roscopter.msg.RC , send_rc)

#define service callbacks
arm_service = rospy.Service('arm', Empty, set_arm)
disarm_service = rospy.Service('disarm', Empty, set_disarm)

#state
gps_msg = NavSatFix()

def mainloop():
```

```
    rospy.init_node('roscopter')
    while not rospy.is_shutdown():
        rospy.sleep(0.001)
        msg = master.recv_match(blocking=False)
        if not msg:
            continue
        #print msg.get_type()
        if msg.get_type() == "BAD_DATA":
            if mavutil.all_printable(msg.data):
                sys.stdout.write(msg.data)
                sys.stdout.flush()
        else:
            msg_type = msg.get_type()
            if msg_type == "RC_CHANNELS_RAW" :
                pub_rc.publish([msg.chan1_raw, msg.chan2_raw,
msg.chan3_raw, msg.chan4_raw, msg.chan5_raw, msg.chan6_raw,
msg.chan7_raw, msg.chan8_raw])
            if msg_type == "HEARTBEAT":
                pub_state.publish(msg.base_mode &
mavutil.mavlink.MAV_MODE_FLAG_SAFETY_ARMED,
                                  msg.base_mode &
mavutil.mavlink.MAV_MODE_FLAG_GUIDED_ENABLED,
                                  mavutil.mode_string_v10(msg))
            if msg_type == "VFR_HUD":
                pub_vfr_hud.publish(msg.airspeed, msg.groundspeed,
msg.heading, msg.throttle, msg.alt, msg.climb)

            if msg_type == "GPS_RAW_INT":
                fix = NavSatStatus.STATUS_NO_FIX
                if msg.fix_type >=3:
                    fix=NavSatStatus.STATUS_FIX
                pub_gps.publish(NavSatFix(latitude = msg.lat/1e07,
                                          longitude = msg.lon/1e07,
                                          altitude = msg.alt/1e03,
                                          status =
NavSatStatus(status=fix, service = NavSatStatus.SERVICE_GPS)
                                          ))
            #pub.publish(String("MSG: %s"%msg))
            if msg_type == "ATTITUDE" :
                pub_attitude.publish(msg.roll, msg.pitch, msg.yaw,
msg.rollspeed, msg.pitchspeed, msg.yawspeed)


            if msg_type == "LOCAL_POSITION_NED" :
                print "Local Pos: (%f %f %f) , (%f %f %f)" %(msg.x,
msg.y, msg.z, msg.vx, msg.vy, msg.vz)

            if msg_type == "RAW_IMU" :
                pub_raw_imu.publish (Header(), msg.time_usec,
                                     msg.xacc, msg.yacc, msg.zacc,
                                     msg.xgyro, msg.ygyro, msg.zgyro,
                                     msg.xmag, msg.ymag, msg.zmag)

wait_heartbeat(master)

print("Sleeping for 10 seconds to allow system , to be ready")
rospy.sleep (10)
print("Sending all stream request for rate %u" % opts.rate)
master.mav.request_data_stream_send(
    master.target_system,
    master.target_component,
```

```
    mavutil.mavlink.MAV_DATA_STREAM_ALL,
    opts.rate,
    1)

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

# Apéndice F

## Código para la implementación del sistema de control

```
#=======SISTEMA DE CONTROL PARA VUELO AUTONOMO QUADROTOR - OUTDOOR=======
#---------------------CHRISTIAN YEYMI MAMANI MAMANI----------------------
#--------------------------PIURA - UDEP 2017-----------------------------
#---------------------------IMPLEMENTACION-------------------------------
#====================CONTROL Y PERFIL DE VELOCIDAD=======================
#!/ usr/bin/env python
from __future__ import print_function
import roslib ; roslib.load_manifest ('roscopter')
import rospy
from std_msgs.msg import String
from sensor_msgs.msg import NavSatFix,NavSatStatus,Imu
import roscopter.msg
import sys,struct,time,os
import math
import argparse
#---------------------------Opciones de menu-----------------------------
parser = argparse.ArgumentParser(description='Sistema de Control de
trayectoria para cuadricoptero')
parser.add_argument('-f',     dest ="configuracion",        help ="Modos
de vuelo: trayectoria,rotacion,prueba",        default ="trayectoria")
parser.add_argument('-c',     dest ="coordenadas",          help ="Tipos
de coordenadas: geodesica,utm",          default ="utm")
parser.add_argument('-m',     dest ="metodo",                    help
="Metodo para Perfil Velocidad Trapezoidal: vectores,angulo",
      default ="angulo")
parser.add_argument('-w',     dest ="arreglo_waypoints",    type =float,
           help ="Waypoints de la trayectoria: long1 lat1 long2 lat2
...", nargs='*')
```

```
parser.add_argument("-r",     dest ="yaw_ref",              type =int,
            help ="Angulo yaw de referencia (deg)",
                  nargs='*')
opts=parser.parse_args()

# Modos de vuelo del APM
# modo_1=0-1230         # PWM ---> ideal:615    ---> Para nuestro caso del
APM configurado (modo 1= STABILIZE)
# modo_2=1231-1360      # PWM ---> ideal:1295
# modo_3=1361-1490      # PWM ---> ideal:1426
# modo_4=1491-1620      # PWM ---> ideal:1556
# modo_5=1621-1749      # PWM ---> ideal:1686
# modo_6=1750 - +       #      ---> ideal:1815   ---> Para nuestro caso del
APM configurado (modo 6= ALT_HOLD)
#=============================================================================
#...................RUTINA PARA EL CALCULO DE LOS WAYPOINTS DEL PERFIL DE
VELOCIDAD TRAPEZOIDAL.....................
def puntos(punto1,punto2):
      global vk,k,kc
      v_ini=0           # Velocidad inicial
      pt=0.3       # Porcentaje de aceleracion
      vk=1         # Velocidad maxima
      n=30         # Numero de puntos para hallar
      tiempo=[]
      x=[];y=[]
      # Para el calculo de la distancia entre puntos
      def modulo(pto_a,pto_b):
          mod=math.sqrt(((pto_b[0]-pto_a[0])**2)+((pto_b[1]-
pto_a[1])**2))
          return mod
      # Para el calculo de la resultante de las componentes x-y
(velocidad y aceleracion) de un vector de datos
      def modulo2(a,b):
          vtot=[]
          n=len(a)
          for item in range(n):
              pen=math.sqrt(((a[item])**2)+((b[item])**2))
              vtot.append(pen)
          return vtot
      # Para el calculo del punto intermedio P(tao)
      def punto_tao(ace,tao,d,n,m):
          pto_tao=(0.5*ace*(tao)**2)*(m-n)/d+n
          return pto_tao
      # Para el calculo del punto intermedio P(T-tao)
      def punto_T_tao(v,T,tao,d,pto_tao,n,m):
          pto_T_tao=(v*(T-2*tao))*(1/d)*(m-n)+pto_tao
          return pto_T_tao
      # Para el calculo de los segmentos de aceleracion y desaceleracion
constante
      def segmento1(ace,tk,v_ini,d,n,m):
          pto_seg1=(0.5*ace*(tk)**2+v_ini*tk)*((m-n)/d)+n
          return pto_seg1
      # Para el calculo del segmento de velocidad constante
      def segmento2(tk,vk,dsgmnt2,n,m):
          pto_seg2=(vk/dsgmnt2)*tk*(m-n)+n
          return pto_seg2
      # Para el calculo de las derivadas con metodos numericos
      def pendiente(a,t):
          vec=[0]
          n=len(t)
          for ite in range(n-1):
```

```
                pen=(a[ite+1]-a[ite])/(t[ite+1]-t[ite])
                vec.append(pen)
            return vec

        d=modulo(punto1, punto2)
        T=d/(vk*(1-pt))
        tao=T*pt
        ace=vk/tao
        k=int(round(pt*n))  # Numero de puntos en la recta acelerada y
desacelerada
        kc=n-2*k            # Numero de puntos en la recta de velocidad
constante
        # Hallamos P(tao)
        pto_x1=punto_tao(ace,tao,d,punto1[0],punto2[0])
        pto_y1=punto_tao(ace,tao,d,punto1[1],punto2[1])
        # Hallamos P(T-tao)
        pto_x2=punto_T_tao(vk,T,tao,d,pto_x1,punto1[0],punto2[0])
        pto_y2=punto_T_tao(vk,T,tao,d,pto_y1,punto1[1],punto2[1])
        # Puntos intermedios
        pto_tao=[pto_x1,pto_y1]       # P(tao)
        pto_T_tao=[pto_x2,pto_y2]     # P(T-tao)
        # Segmento 1 (Aceleracion constante)
        dsgmnt1=modulo(punto1, pto_tao)
        for i in range(k):
            tseg1=(tao/k)*i
            seg1x=segmento1(ace, tseg1,v_ini,dsgmnt1, punto1[0],
pto_tao[0])
            seg1y=segmento1(ace, tseg1,v_ini,dsgmnt1, punto1[1],
pto_tao[1])
            x.append(seg1x)
            y.append(seg1y)
            tiempo.append(tseg1)
        # Segmento 2 (Velocidad constante)
        dsgmnt2=modulo(pto_tao,pto_T_tao)
        for j in range(kc):
            tseg2=((T-2*tao)/(kc-1))*j
            seg2x=segmento2(tseg2,vk,dsgmnt2,pto_tao[0],pto_T_tao[0])
            seg2y=segmento2(tseg2,vk,dsgmnt2,pto_tao[1],pto_T_tao[1])
            x.append(seg2x)
            y.append(seg2y)
            tiempo.append(tseg2+tao)
        # Segmento 3 (Desaceleracion constante)
        dsgmnt3=modulo(pto_T_tao,punto2)
        for h in range(k):
            tseg3=(tao/k)*(h+1)
            seg3x=segmento1(-ace, tseg3,vk,dsgmnt3, pto_T_tao[0],
punto2[0])
            seg3y=segmento1(-ace, tseg3,vk,dsgmnt3, pto_T_tao[1],
punto2[1])
            x.append(seg3x)
            y.append(seg3y)
            tiempo.append(tseg3+T-tao)
        # Calculo de las velocidades
        vx=pendiente(x, tiempo)
        vy=pendiente(y, tiempo)
        vt=modulo2(vx, vy)
        # Calculo de las aceleraciones
        acex=pendiente(vx, tiempo)
        acey=pendiente(vy, tiempo)

        return x,y,vx,vy,vt,acex,acey,tiempo
```

```
#=============================================================================
#-----Algoritmo para el calculo de los angulos para el metodo angulo-----
def calculo_angulos(vector_lon,vector_lat):
      angulos=[]
      n=len(vector_lon)
      for j in range(n-1):
          x=vector_lon[j+1]-vector_lon[j]
          y=vector_lat[j+1]-vector_lat[j]
          ang=math.atan2(y,x)
          ang_degree=math.degrees(ang)
          angulos.append(ang_degree)
      return angulos
#--------------------Algoritmo de Control---------------------
def control(k_xyz,k_yaw,error,last_error,last_ui):
      u=[]; ui=[]
      for i in range(4):
            if i==3:
                  uP= k_yaw[0]*error[i]   # Para el control del angulo
psi
                  uI= k_yaw[1]*error[i]+last_ui[i]
                  uD= k_yaw[2]*(error[i]-last_error[i])
                  ut= uP+uI+uD
            else:
                  uP= k_xyz[0]*error[i]   # Para el control en x-y
                  uI= k_xyz[1]*error[i]+last_ui[i]
                  uD= k_xyz[2]*(error[i]-last_error[i])
                  ut= uP+uI+uD
            if ut<-500:       # Minimo
                  ut=-500
            elif ut>500:      # Maximo
                  ut=500
            uI=ut-uP-uD       # Anti Reset-Windup
            u.append(ut)
            ui.append(uI)
      return u,ui

#--------------------Algoritmo conversion de coordenadas-----------------
def geodesicas_to_utm(lon,lat):     # Conversion de coordenadas
geodesicas-> UTM
      lon_rad=lon*math.pi/180
      lat_rad=lat*math.pi/180
      huso=int(lon/6+31)                  # Calculo del huso, solo la
parte entera
      lambda_0=huso*6-183
      delta_lambda=lon_rad-(lambda_0*math.pi/180)
      A=math.cos(lat_rad)*math.sin(delta_lambda)           # Calculo de
parametros
      epsilon=0.5*(math.log((1+A)/(1-A)))
      n=math.atan(math.tan(lat_rad)/math.cos(delta_lambda))-lat_rad
      v=6397376.633466756/math.sqrt(1+0.00676817019657*math.cos(lat_rad)*
*2)
      epsi=0.00676817019657*(epsilon**2)*(math.cos(lat_rad)**2)/2
      A1=math.sin(2*lat_rad)
      A2=A1*(math.cos(lat_rad))**2
      J2=lat_rad+(A1/2)
      J4=(3*J2+A2)/4
      J6=(5*J4+A2*(math.cos(lat_rad))**2)/3
      Bphi=6397376.633466756*(lat_rad-0.005076128*J2+(4.29451198217e-
5)*J4-(1.69551596705e-7)*J6)
      x_utm=epsilon*v*(1+(epsi/3))+500000       # Calculo final de
coordenadas
```

```
      if lat<0:
            y_utm=n*v*(1+epsi)+Bphi+1e7
      else:
            y_utm=n*v*(1+epsi)+Bphi
      return x_utm,y_utm

#----------------------Lectura de datos----------------------
def leer_rc(data):                          # Leer comandos para control
del cuadricoptero
      global throttle,modo
      throttle=data.channel[2]
      modo=data.channel[4]

def leer_angulos(data):             # Leer los angulos
      global ang_phi,ang_the,ang_psi
      ang_phi=data.roll*180/math.pi
      ang_the=data.pitch*180/math.pi
      ang_psi=data.yaw*180/math.pi

def leer_altura(data):              # Leer altura del cuadricoptero
      global altura
      altura=data.alt

#-----------------Algoritmos para seguridad cuadricoptero----------------
def security_pwm(channel):
      channel_sec = channel
      if ( modo > 1751):
            print("Cambiando a modo seguro")
            channel=[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0]
            pub_rc.publish(channel)
      elif (modo < 1231):
            pub_rc.publish(channel_sec)
      # else:
      #     for i in range (990,1010):
      #           channel = [0 ,0 ,i ,0 ,0 ,0 ,0 ,0]
      #           pub_rc.publish(channel)

#============================Programa Principal============================
#.....................Modo seguidor de trayectoria.....................
if opts.configuracion == "trayectoria":
      def leer_m_trayectoria_coordenadas(data):      # Leer las
coordenadas en tiempo real
            global
wn,pvtn,x_sp,y_sp,vx_sp,vy_sp,vt_sp,ax_sp,ay_sp,t,crono,aux,punto_inicial
            global
k_xyz,k_yaw,wpx,wpy,ui,error,last_error,tolerancia,vector_angulos,aux2,au
x3,aux4
            pos_x_geo = data.longitude
            pos_y_geo = data.latitude

            if opts.coordenadas=="geodesica":
                  pos=geodesicas_to_utm(pos_x_geo,pos_y_geo)
            elif opts.coordenadas=="utm":
                  pos=[pos_x_geo,pos_y_geo]
            pos_x=pos[0]
            pos_y=pos[1]
            if punto_inicial==0:
                  wpx.insert(0,pos_x)
                  wpy.insert(0,pos_y)
                  if opts.metodo=="angulo":
                        vector_angulos=calculo_angulos(wpx,wpy)
```

```
                punto_inicial=1
            last_error[0]=error[0]
            last_error[1]=error[1]
            last_error[2]=error[2]
            last_error[3]=error[3]
            # Hallamos los valores de los waypoints al inicio y al final
de cada tramo
            if ((wn==0) or (len(x_sp)==pvtn and wn<(len(wpx)-1))):

      [x_sp,y_sp,vx_sp,vy_sp,vt_sp,ax_sp,ay_sp,t]=puntos([wpx[wn],wpy[wn]
],[wpx[wn+1],wpy[wn+1]])
            error[0]=x_sp[pvtn]-pos_x
            error[1]=y_sp[pvtn]-pos_y
            error[2]=0
            last_ui=ui

            tt = rospy.Time.from_sec(time.time())     # Creamos un
cronometro
            if aux==0:                    # Reiniciamos el cronometro
                  crono=tt.to_sec()
            ttt=tt.to_sec()-crono
            #--------------Metodo Perfil de Velocidad Trapezoidal
mediante giro del angulo yaw-------------
            if opts.metodo=="angulo":
                  error[3]=vector_angulos[wn]-ang_psi
                  if aux2==0:

      [ut,ui]=control(k_xyz,k_yaw,error,last_error,last_ui)
                        u_rc_yaw=ut[3]+1500           # Angulo yaw
                        channel=[0,0,0,u_rc_yaw,0,0,0,0]
                        security_pwm(channel)
                        print ("Posicionando angulo")
                        print ("Angulo Yaw: ",ang_psi)
                        print ("Vector yaw:",vector_angulos)
                        print ("Referencia yaw:",vector_angulos[wn])
                        print ("Error yaw:",error[3])
                        print ("Control:",ut[3])
                        print ("Numero de angulo:",wn)
                        print ("--------------------------")
                        if ((abs(error[3])<0.9) and
(wn<(len(vector_angulos)))):
                              print ("Listo ---> Control")
                              aux2=1;           aux3=0
                              rospy.sleep(2)
                  if (aux3==0):
                        if (t[pvtn]>ttt):
                              aux=1                   # Indicamos que ya
no estamos en t[0]

      [ut,ui]=control(k_xyz,k_yaw,error,last_error,last_ui)
                              u_rc_roll=1500-300*vt_sp[pvtn]      # Hacia
adelante
                              u_rc_yaw=ut[3]+1500                       #
Angulo yaw
                              channel=[u_rc_roll,0,0,u_rc_yaw,0,0,0,0]
                              security_pwm(channel)

                              print ("Angulos:",vector_angulos)
                              print ("Total tramos a recorrer:",len(wpx)-
1)
                              print ("Tramo actual:",wn+1)
```

```
                              print ("Numero puntos perfil de
velocidad:",len(x_sp))
                              print ("Pivote perfil velocidad:",pvtn+1)
                              print ("_______Tiempo______")
                              print ("Tiempo que debe tomar:",t[pvtn])
                              print ("Cronometro:", ttt)
                              print ("_______Medicion______")
                              print ("Medicion x:",pos_x)
                              print ("Medicion y:",pos_y)
                              print ("Medicion yaw:",ang_psi)
                              print ("Referencia yaw:",vector_angulos[i])
                              print ("_______Error______")
                              print ("Error x:",error[0])
                              print ("Error y:",error[1])
                              print ("Error yaw:",error[3])
                              print ("_______Control______")
                              print ("Velocidad promedio:",vt_sp[pvtn])
                              print ("Control promedio:",u_promedio)
                              print ("Control x:",move.linear.x)
                              print ("Control yaw:",move.angular.z)
                              print ("--------------")
                        elif pvtn<len(t)-1:
                              pvtn=pvtn+1
                        elif (wn<len(wpx)-2):
                              wn=wn+1;     aux2=0;           aux3=1;
                              pvtn=0;            aux=0
                              x_sp=[];     vx_sp=[];    ax_sp=[]
                              y_sp=[];     vy_sp=[];    ay_sp=[]
                              t=[];        vt_sp=[];
                              rospy.sleep(2)
                        else:
                              aux2=1;            aux3=1
                              print ("Recorrido terminado")
                              channel=[0,0,0,0,0,0,0,0]
                              security_pwm(channel)
            #-------------Metodo Perfil de Velocidad Trapezoidal mediante
vectores de velocidad en x-y-------------
            elif opts.metodo=="vectores":
                  error[3]=0-ang_psi
                  if aux4==0:
                        if (t[pvtn]>ttt):
                              aux=1       # Indicamos que ya no estamos
en t[0]

      [ut,ui]=control(k_xyz,k_yaw,error,last_error,last_ui)
                              u_rc_roll=1500-ut[0]-500*vx_sp[pvtn]      #
Eje x
                              u_rc_pitch=1500+ut[1]+500*vy_sp[pvtn]     #
Eje y
                              u_rc_yaw=1500+ut[3]
            # Angulo yaw

      channel=[u_rc_roll,u_rc_pitch,0,u_rc_yaw,0,0,0,0]
                              security_pwm(channel)

                              print ("Total tramos a recorrer:",len(wpx)-
1)
                              print ("Tramo actual:",wn+1)
                              print ("Numero puntos perfil de
velocidad:",len(x_sp))
                              print ("Pivote perfil velocidad:",pvtn+1)
```

```
                        print ("_______Tiempo______")
                        print ("Tiempo que debe tomar:",t[pvtn])
                        print ("Cronometro:", ttt)
                        print ("_______Medicion______")
                        print ("Medicion x:",pos_x)
                        print ("Medicion y:",pos_y)
                        print ("Medicion yaw:",ang_psi)
                        print ("Pivote Vx:",vx_sp[pvtn])
                        print ("Pivote Vy:",vy_sp[pvtn])
                        print ("_______Error______")
                        print ("Error x:",error[0])
                        print ("Error y:",error[1])
                        print ("Error yaw:",error[3])
                        print ("_______Control______")
                        print ("Control x:",move.linear.x)
                        print ("Control y:",move.linear.y)
                        print ("--------------")
                    elif pvtn<len(t)-1:
                        pvtn=pvtn+1
                    elif (wn<len(wpx)-2):
                        wn=wn+1;    pvtn=0;           aux=0;
t=[]
                        x_sp=[];    vx_sp=[];   ax_sp=[]
                        y_sp=[];    vy_sp=[];   ay_sp=[]
                    else:
                        aux4=1
                        print ("Recorrido terminado")
                        channel=[0,0,0,0,0,0,0,0]
                        security_pwm(channel)
#-----------------Variables para modo trayectoria------------------
kp_xyz = 4;       ki_xyz = 0.1;          kd_xyz = 0.07
kp_yaw = 2;       ki_yaw = 0.1;          kd_yaw = 0.07
k_xyz=[kp_xyz,ki_xyz,kd_xyz];       k_yaw=[kp_yaw,ki_yaw,kd_yaw]
wpx=[];           wpy=[];           x_sp=[];    y_sp=[];    vt_sp=[]
vx_sp=[];   vy_sp=[];   ax_sp=[];   ay_sp=[];   t=[];       v=[]
wn=0;       pvtn=0;           crono=0;    aux=0;            aux2=0;
      aux3=1
ui=[0,0,0,0];                 error=[0,0,0,0];  last_error=[0,0,0,0]
lon_wp=[];  lat_wp=[];  punto_inicial=0;  vector_angulos=[]
tolerancia=0.05

#---------Arreglo y conversion de waypoints de referencia----------
l=len(opts.arreglo_waypoints)
for i in range(l/2):
    x_lon=opts.arreglo_waypoints[2*i]
    y_lat=opts.arreglo_waypoints[2*i+1]
    lon_wp.append(x_lon)
    lat_wp.append(y_lat)

if opts.coordenadas=="geodesica":
      for n in range(len(lat_wp)-1):
            wp=geodesicas_to_utm(lon_wp[n],lat_wp[n])
            x_utm=wp[0]
            y_utm=wp[1]
            wpx.append(x_utm)
            wpy.append(y_utm)
else:
      wpx=lon_wp
      wpy=lat_wp
#--------------Suscripcion a los topicos necesarios-------------
rospy.Subscriber ('gps',NavSatFix,leer_m_trayectoria_coordenadas)
```

```
      rospy.Subscriber ('attitude', roscopter.msg.Attitude ,leer_angulos)
      rospy.Subscriber ('vfr_hud', roscopter.msg.VFR_HUD,leer_altura)

#.......................Modo control de angulo.......................
elif opts.configuracion == "rotacion":
      def leer_m_rotacion_angulos(data):        # Leer los angulos
            global error,ui,kp,ki,kd,i
            ang_psi=data.yaw*180/math.pi
            if i<(len(opts.yaw_ref)):
                  last_error=error
                  last_ui=ui
                  #error=math.radians(opts.yaw_ref[i])-ang_psi
                  error=opts.yaw_ref[i]-ang_psi
                  up= kp*error
                  ui= ki*error+last_ui
                  ud= kd*(error-last_error)
                  u=up+ui+ud
                  if u<-500: #Minimo
                        u=-500
                  elif u>500: #Maximo
                        u=500
                  ui=u-up-ud  #Anti Reset-Windup
                  u_rc_yaw=1500+u
                  channel=[0,0,0,u_rc_yaw,0,0,0,0]
                  security_pwm(channel)

                  f= open('angulo_yaw','a')
                  datos = {'yaw':ang_psi, 'yaw_ref':opts.yaw_ref[i],
'error':error,'control':u_rc_yaw}
                  f.write (" %(yaw)0.4f %(yaw_ref)0.4f %(error)0.4f
%(control)0.4f \n" % datos)

                  print ("Vector yaw:",opts.yaw_ref)
                  print ("Angulo yaw:",ang_psi)
                  print ("Referencia yaw:",opts.yaw_ref[i])
                  print ("Error yaw:",error)
                  print ("Control:",u)
                  print ("Numero de angulo:",i+1)
                  print ("--------------")
                  if (abs(math.degrees(error))<0.9):
                        i=i+1
                        print ("Siguiente angulo")
                        rospy.sleep(3)
            else:
                  print ("Angulos terminados")
                  channel=[0,0,0,0,0,0,0,0]
                  security_pwm(channel)

      kp=2;       ki=0.1      ; kd=0.07
      error = 0;  ui = 0      ; i=0
      rospy.Subscriber ('attitude', roscopter.msg.Attitude
,leer_m_rotacion_angulos)

#------------Suscripcion al topico cmd_vel para control---------------
pub_rc = rospy.Publisher ('send_rc', roscopter.msg.RC, queue_size=10 )
rospy.Subscriber ('rc', roscopter.msg.RC,leer_rc )

#================================================================
def parar():
      print("Cambiando a modo seguro")
      channel=[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0]
```

```
      pub_rc.publish(channel)
      print ("Nodo terminado")

#----------------------Bucle Principal-----------------------
def mainloop():
      rospy.init_node ('control_cuadricoptero')
      rospy.on_shutdown(parar)
      while not rospy.is_shutdown():
            rospy.sleep (0.001)
if __name__ == '__main__':
      try:
            mainloop ()
      except rospy.ROSInterruptException : pass
```