Proyecto Fin de Carrera
Ingeniería Electrónica, Robótica y Mecatrónica

ESTRUCTURA DE CONTROL EN CASCADA PARA VEHÍCULO AÉREO MULTIRROTOR

Autor: Javier Murillo Burgos

Tutor: Manuel Vargas Villanueva

Tutor: Manuel Gil Ortega Linares

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

Sevilla, 2020

Proyecto Fin de Carrera
Ingeniería Electrónica, Robótica y Mecatrónica

ESTRUCTURA DE CONTROL EN CASCADA PARA VEHÍCULO AÉREO MULTIRROTOR

Autor:
Javier Murillo Burgos

Tutores:
Manuel Vargas Villanueva
Profesor Titular
Manuel Gil Ortega Linares
Catedrático de Universidad

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

Sevilla, 2020

Proyecto Fin de Carrera: ESTRUCTURA DE CONTROL EN CASCADA PARA VEHÍCULO AÉREO MULTIRROTOR

Autor: Javier Murillo Burgos
Tutor: Manuel Vargas Villanueva
Tutor: Manuel Gil Ortega Linares

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

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Resumen

Los vehículos aéreos no tripulados cobran cada vez más importancia en la sociedad actual. No solamente se han convertido en una gran herramienta en ciertos sectores, también se han convertido en uno de los hobbies más populares hoy en día.

Hay una gran variedad de drones, con diferentes configuraciones y composiciones, pero en este trabajo se pretende entender concretamente los conocidos como quadrotors. Se estudiará su comportamiento y se diseñará una estrategia de control poniendo en práctica los conocimientos adquiridos durante la carrera.

Se realizará una primera toma de contacto con un autopiloto, con el objetivo de que sirva de orientación para futuros trabajos. Se analizará su estructura con el fin de entender su funcionamiento y aprender a implementar nuevas estrategias de vuelo.

Abstract

The unmanned aerial vehicles are becoming increasingly important in our society today. They have revolutionized both the world of work, eliminating risks and facilitating tasks, as well as the world of entertainment, as they have become one of the most popular hobbies today.

There are a wide variety of drones, with different configurations and compositions, but this work aims to specifically understand those known as quadrotors. Their behavior will be studied and a control strategy will be designed, putting into practice the knowledge acquired during the race.

A first contact with an autopilot will be made, with the aim of serving as an orientation for future work. Its structure will be analyzed in order to understand its operation and learn to implement new flight strategies.

Índice Abreviado

Índice

Notación

$\{B\}$	Sistema de coordenadas del cuerpo
$\{G\}$	Sistema de coordenadas global
m_B	Masa del quadrotor
I_{XX}	Inercia del quadrotor en X^B $(Kg \cdot m^2)$
I_{YY}	Inercia del quadrotor en Y^B $(Kg \cdot m^2)$
I_{ZZ}	Inercia del quadrotor en Z^B $(Kg \cdot m^2)$
g	Aceleración de la gravedad $(\frac{m}{s^2})$
l	Distancia de los rotores al centro de masas (m)
ω_{max}	Velocidad máxima del rotor $(\frac{rad}{s})$
k_{th}	Coeficiente de empuje
k_d	Coeficiente de resistencia aerodinámica de los rotores
f_i^T	Fuerza de empuje del motor
ω_i	Velocidad del motor i $(\frac{rad}{s})$
F_B^T	Fuerza total de empuje en ejes cuerpo
τ_ϕ	Momento angular en X^B
τ_θ	Momento angular en Y^B
τ_ψ	Momento angular en Z^B
R_Z	Matriz de rotación en Z
R_Y	Matriz de rotación en Y
R_X	Matriz de rotación en X
R_G^B	Matriz de transformación de ejes globales a ejes cuerpo
R_B^G	Matriz de transformación ejes cuerpo a ejes globales
p	Velocidad angular del quadrotor en X^B $(\frac{m}{s})$
q	Velocidad angular del quadrotor en Y^B $(\frac{m}{s})$
r	Velocidad angular del quadrotor en Z^B $(\frac{m}{s})$
ϕ	Roll (rad)
θ	Pitch (rad)
ψ	Yaw (rad)
ω	Vector de velocidades angulares en ejes cuerpo
$\dot{\eta}$	Vector de variaciones de ángulos
$W(\dot{\eta})$	Matriz de transformación de variaciones de ángulos a velocidades angulares
P^G	Vector posición en ejes globales

$\dot{P}^G$	Vector velocidades lineales en ejes globales
$\ddot{P}^G$	Vector aceleraciones lineales en ejes globales
F_g^G	Fuerza de la gravedad
Kd_x	Coeficiente de fricción viscosa en X
Kd_y	Coeficiente de fricción viscosa en Y
Kd_z	Coeficiente de fricción viscosa en Z
F_d^G	Fuerza de resistencia aerodinámica en ejes globales
I^b	Tensor de inercias
T_g^B	Vector de pares debidos al efecto giroscópico
$u(t)$	Señal de control en tiempo continuo
K_p	Ganancia proporcional
$e(t)$	Error en tiempo continuo
T_i	Tiempo integral
T_d	Tiempo derivativo
K_i	Ganancia integral
K_p	Ganancia proporcional
∂	Derivada parcial
$\mathscr{L}$	Transformada de Laplace
$G(s)$	Planta en tiempo continuo
$C(s)$	Controlador en tiempo continuo
$F(s)$	Filtro en tiempo continuo
T_m	Tiempo de muestreo
$G_p(s)$	Modelo para el control de p
$G_q(s)$	Modelo para el control de q
$G_r(s)$	Modelo para el control de r
$C_p(s)$	Controlador diseñado para el control de p
$C_q(s)$	Controlador diseñado para el control de q
$C_r(s)$	Controlador diseñado para el control de r
$G_\phi(s)$	Modelo para el control del roll
$G_\theta(s)$	Modelo para el control del pitch
$G_\psi(s)$	Modelo para el control de yaw
$PI_\phi(s)$	Controlador PI diseñado para el roll
$PI_\theta(s)$	Controlador PI diseñado para el pitch
$PI_\psi(s)$	Controlador PI diseñado para el yaw
$U1$	Fuerza total de empuje equivalente a $F_T^B T$
Z^G	Altura en ejes globales
$\dot{Z}^G$	Velocidad de ascenso en ejes globales ($\frac{m}{s}$)
$\ddot{Z}^G$	Aceleración de ascenso en ejes globales ($\frac{m}{s^2}$)
$G_{Z^G}(s)$	Modelo para el control de altura
$PID_{Z^G}(s)$	Controlador diseñado para el control de altura
$F_{Z^G}(s)$	Filtro para la referencia en altura
$\ddot{X}^G$	Aceleración lineal en X^G
$\ddot{Y}^G$	Aceleración lineal en Y^G
$G_{X^G}(s)$	Modelo para el control de la posición X en ejes globales
$G_{Y^G}(s)$	Modelo para el control de la posición Y en ejes globales
$PID_{posicion}(s)$	Controlador diseñado para el control de posición
$F_{posicion}(s)$	Filtro para la referencia en posición

1 Introducción

Actualmente la popularidad de los vehículos aéreos autónomos no tripulados (*UAVs*), o más comúnmente conocidos como, drones, crece exponencialmente.

Hace años este tipo de vehículo comenzó desarrollarse con fines bélicos. Hoy en día este fin cobra cada vez mas importancia, ya que los avances tecnológicos han permitido crear drones capaces de fijar y bombardear objetivos con gran precisión y exactitud. Además se les está dotando de la capacidad de ser imperceptibles por los radares, esto y la ausencia de un piloto físico, los hacen perfectos para el espionaje. El mal uso que pueda hacerse de esta tecnología, hace replantearse si en el futuro, los drones traerán más inconvenientes que ventajas a la sociedad.

Figura 1.1 MQ 9 Reaper Drone.

Dejando a un lado el entorno militar, los avances tecnológicos han permitido acercar los drones a la sociedad. El desarrollo de componentes electrónicos cada vez más pequeños, pero a la vez más potentes, como baterías, sensores, cámaras... y la reducción de costes en la fabricación han permitido crear UAVs útiles para todo tipo de tareas.

Debido a que se trata de vehículos no tripulados, permiten acceder a lugares peligrosos y de difícil accesibilidad sin riesgo, así como adentrarse en lugares donde se han producido desastres naturales o humanos, permitiendo hacer un reconocimiento del terreno y facilitando las labores de actuación.

Estos vehículos también son muy utilizados dentro del área climática, pues permiten acercarse a huracanes y tormentas, pudiendo estudiarlos en profundidad, además alcanzan alturas considerables, permitiendo realizar estudios sobre la calidad del aire y el estado de nuestra atmósfera. Son muy empleados en la agricultura, aplicando pesticidas, regando o controlando el desarrollo y crecimiento de las cosechas, así como analizando el estado del terreno. A parte, y gracias a las cámaras de hoy en día, son muy utilizados en la industria del cine, ya que se consiguen unos planos increíbles a bajo coste.

Dentro de todos los usos actuales que tienen los UAVs, hay uno en especial que ha cobrado gran popularidad, y es el destinar este tipo de vehículos al transporte y el reparto de mercancías. Este campo de investigación ha sido abordado por múltiples autores, además ha sido tratado en anteriores trabajos de la escuela, concretamente para el transporte de cargas colgantes.([2] y [16]).

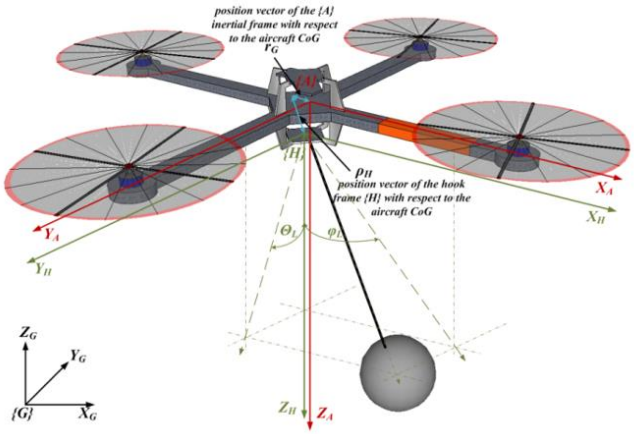


Figura 1.2 Quadrotor con carga colgante.

Por último cabe destacar que gracias a la tecnología FPV (*First Person View*), los drones se han vuelto un ”juguete” muy popular entre la población. Se ha hecho tan popular este tipo de pilotaje, que se han creado carreras profesionales, convirtiéndose en uno de los deportes del futuro.

Está claro que los UAVs son el futuro, pero este trabajo se centra en la idea de que los drones acabarán siendo completamente autónomos, eliminando totalmente la acción de un piloto. Primero partiremos de entender la física de este tipo de vehículos, concretamente un quadrotor. Se desarrollarán las ecuaciones que describen su comportamiento, con el fin de obtener un modelo matemático sobre el que diseñar un software de control.

Partiendo del modelo matemático deducido, se pretende crear un piloto automático desde cero, que se encargue de controlar el vehículo dando simplemente como referencia una posición en ejes globales. El control de un quadrotor es complejo, de forma que habrá que descomponer el control en varios niveles anidados, siendo el más bajo nivel el más simple, y el más alto nivel el control completo y más complejo. Se realizará la implementación del control diseñado en MATLAB® con el fin de estudiar el comportamiento del vehículo y se creará una sencilla simulación en 3D para que pueda verse en tiempo real el vuelo del quadrotor.

Por último se tomará un código real de un autopiloto que es muy conocido hoy en día y es compatible con una gran variedad de vehículos, con el fin de estudiar su código y entender como funciona. Utilizaremos

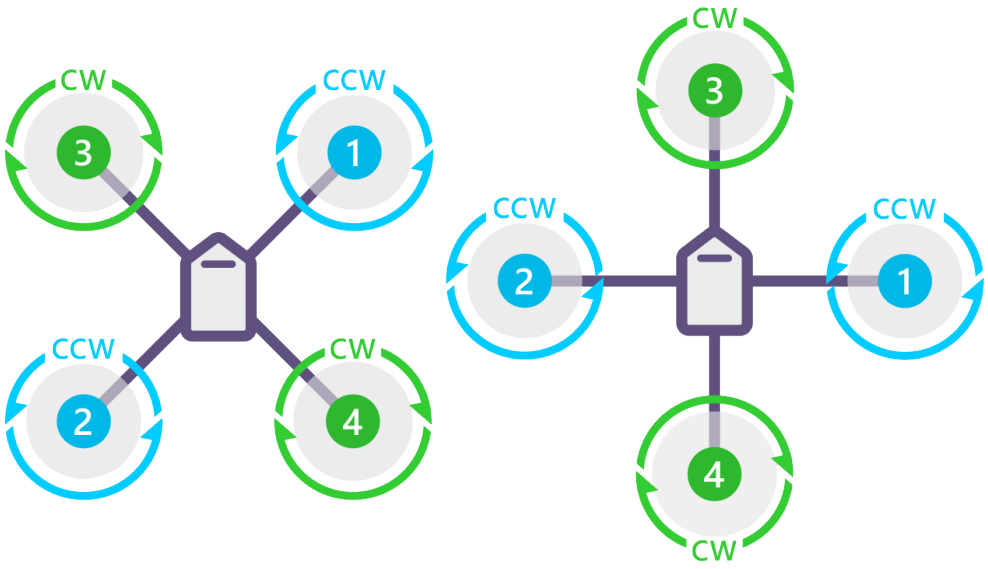
una simulación más compleja que la que montamos en MATLAB® , concretamente una simulación en GAZEBO® e intentaremos incorporar el control que hemos diseñado. Proprocionando así una toma de contacto con un autopiloto real y que consta de una gran comunidad, con el fin de que futuros trabajos puedan incorporar sus propios modos de vuelo.

2 Modelo Matemático del Quadrotor

En el presente capítulo se obtienen las ecuaciones cinemáticas y dinámicas que definen el comportamiento de un quadrotor. En primer lugar, se detalla la disposición de los ejes en el sistema $\{\mathbf{B}\}$ y se describen una serie de hipótises y suposiciones que facilitan los cálculos .Atendiendo a estas hipótesis, se plantean las ecuaciones cinemáticas y dinámicas, y por último, se realiza la linealización de las mismas, dando como resultado un modelo lineal con el que desarrollar una estrategia de control en los siguientes capítulos.

2.1 Variables e Hipótesis

Para los *UAVs* de tipo quadrotor, existen dos configuraciones principales a la hora de repartir la potencia y el control entres sus motores. La configuración en X, que es la que será utilizada en este trabajo y la configuración en $+$.



(a) Configuración X. **(b)** Configuración $+$.

Figura 2.1 Configuraciones.

Los ejes del sistema $\{\mathbf{B}\}$ ligados al quadrotor , se han dispuesto de tal forma, que el eje $\mathbf{X^B}$ es positivo hacia adelante, el eje $\mathbf{Y^B}$ es positivo hacia la izquierda del quadrotor, y por último, el eje $\mathbf{Z^B}$ es positivo hacia arriba.

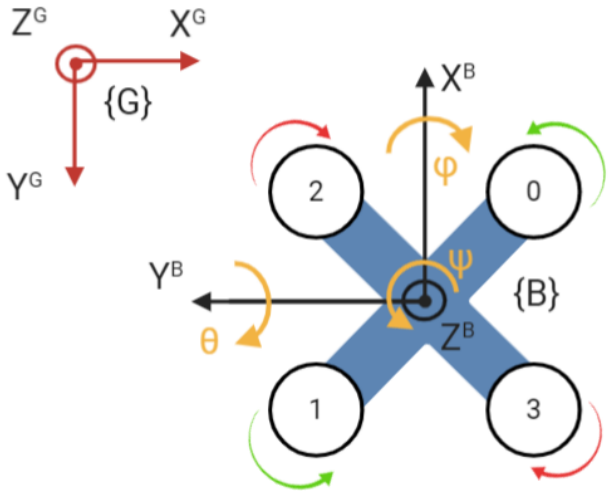


Figura 2.2 Configuración X.

Esta configuración de ejes y distribución de los motores, hacen que se reduzca la carga individual que

tiene que soportar cada motor, debido a que intervienen los cuatro motores a la hora de realizar los giros en los ejes locales. Por contra, se hace un poco más complejo el cálculo de los pares en $\mathbf{X^B}$ e $\mathbf{Y^B}$. Si se hubiesen alineado los ejes con los brazos del quadrotor, el cálculo de los momentos sería más sencillo, pero la carga individual de cada motor sería mayor, ya que intervendrían solamente dos de ellos en el giro del quadrotor sobre los ejes locales.

Un quadrotor se define mediante una serie de ecuaciones no lineales que dificultan su simulación y su control, pero es posible simplificarlas mediante una serie de hipótesis y suposiciones. Esta simplificación aleja nuestro modelo de la realidad, lo que conlleva a una pérdida de precisión, pero nos permite tener un modelo lo suficientemente preciso como para diseñar estrategias de control que podrán ser aplicadas a un sistema real.

- La estructura del quadrotor es rígida y simétrica con respecto al centro de masas.
- El empuje y la resistencia de cada motor es proporcional al cuadrado de la velocidad del motor.
- Las fuerzas aerodinámicas y de sustentación se suponen aplicadas al centro de masas del vehículo.
- Las hélices se consideran rígidas.
- Las velocidades de los vehículos circundantes son despreciables.

A continuación se recogen los valores numéricos de los parámetros dinámicos que intervienen en las ecuaciones que describen nuestro quadrotor. Estos datos han sido proporcionados por Erle Robotics (Apéndice A) para la simulación de su quadrotor Erle-Copter en el entorno Gazebo.

Tabla 2.1 Variables características del quadrotor.

Variable	Unidades	Valor	Descripción
m_B	K_g	1.2	Masa del quadrotor
I_{XX}	$K_g \cdot m^2$	0.0347563	Inercia en X
I_{YY}	$K_g \cdot m^2$	0.0458929	Inercia en Y
I_{ZZ}	$K_g \cdot m^2$	0.0977	Inercia en Z
g	m/s^2	9.81	Gravedad
l	m	0.141	Distancia de los rotores al centro de masas
ω_{max}	rad/s	838	Velocidad máxima del rotor
k_{th}	$K_g \cdot m/s^2$	8.5486e-6	Coeficiente de empuje
k_d	-	8.06428e-5	Coeficiente de resistencia aerodinámica de los rotores
J_r	$K_g \cdot m^2$	2.409e-4	Inercia del rotor en Z
T_m	s	0.01	Tiempo de muestreo

2.2 Fuerzas y Pares que actúan sobre el Quadrotor

Un quadrotor se clasifica como un sistema subactuado, debido a que puede moverse en seis grados de libertad (3 translacionales y 3 rotacionales), pero solamente tiene 4 variables de entrada con las que

controlar su comportamiento. Estas entradas de control son las velocidades de los motores, de forma que podemos aproximar las fuerzas de sustentación ejercidas por la rotación de los motores como:

$$f_i^T = k_{th}\omega_i^2 \tag{2.1}$$

Donde k_{th} representa el coeficiente de empuje y ω_i la velocidad de rotación del motor.

La altura se controla mediante la fuerza de empuje (F_B^T), que es el resultado de la suma de la fuerza ejercida por cada uno de los motores:

$$F_B^T = k_{th}(\omega_0^2 + \omega_1^2 + \omega_2^2 + \omega_3^2) \tag{2.2}$$

El par responsable de la variación de roll (τ_ϕ) es el resultado del par generado por el empuje de los motores 1 y 2 , menos el par generado por los motores 0 y 3:

$$\tau_\phi = \frac{\sqrt{2}}{2} \cdot l \cdot k_{th}(-\omega_0^2 + \omega_1^2 + \omega_2^2 - \omega_3^2) \tag{2.3}$$

El par encargado de la variación de pitch (τ_θ) es el resultado del par generado por el empuje de los motores 1 y 3, menos el par generado por los motores 0 y 2:

$$\tau_\theta = \frac{\sqrt{2}}{2} \cdot l \cdot k_{th}(-\omega_0^2 + \omega_1^2 - \omega_2^2 + \omega_3^2) \tag{2.4}$$

Los motores no pueden girar todos en el mismo sentido, esto supondría que el quadrotor girase sobre sí mismo debido al par de reacción.Los motores se dividen en pares, de forma que los motores de un brazo giran en un sentido, y los motores del otro brazo, giran en el sentido contrario, así se elimina el momento de rotación y se evita que el quadrotor gire sobres si mismo.
Como nuestra Z^B es positiva hacia arriba la ecuación para el cálculo de τ_ψ es:

$$\tau_\psi = k_d(-\omega_0^2 - \omega_1^2 + \omega_2^2 + \omega_3^2) \tag{2.5}$$

Estas expresiones pueden reescribirse en forma matricial, de forma que obtenemos una matriz de transformación, que nos relaciona pares y fuerzas ($F_B^T, \tau_\phi, \tau_\theta, \tau_\psi$) con las velocidades de los motores:

$$\begin{bmatrix} F_B^T \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} k_{th} & k_{th} & k_{th} & k_{th} \\ -\frac{\sqrt{2}}{2} \cdot l \cdot k_{th} & \frac{\sqrt{2}}{2} \cdot l \cdot k_{th} & \frac{\sqrt{2}}{2} \cdot l \cdot k_{th} & -\frac{\sqrt{2}}{2} \cdot l \cdot k_{th} \\ -\frac{\sqrt{2}}{2} \cdot l \cdot k_{th} & \frac{\sqrt{2}}{2} \cdot l \cdot k_{th} & -\frac{\sqrt{2}}{2} \cdot l \cdot k_{th} & \frac{\sqrt{2}}{2} \cdot l \cdot k_{th} \\ -k_d & -k_d & k_d & k_d \end{bmatrix} \begin{bmatrix} \omega_0^2 \\ \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \end{bmatrix} \tag{2.6}$$

Estas ecuaciones dependen de la configuración de ejes tomada en el sistema $\{\mathbf{B}\}$, ya que si los ejes cambian, el efecto que producen los motores en cada uno de los ejes varía. Un parámetro importante a considerar es la velocidad máxima (ω_{max}) proporcionada por Erle Robotics en [13], la cuál nos permitirá determinar el punto de saturación de las actuaciones durante el control.

2.3 Cinemática

En esta sección se desarrollan las ecuaciones cinemáticas del quadrotor, de las cuáles obtenemos las matrices necesarias para pasar de ejes globales $\{\mathbf{G}\}$ a ejes cuerpo $\{\mathbf{B}\}$ y viceversa.

Está demostrado que cualquier rotación de un sólido, puede expresarse como la composición de tres rotaciones elementales alrededor de ejes diferentes. Estas rotaciones pueden considerarse en torno a unos

ejes fijos o a unos ejes intrínsecos, en este trabajo se ha tomado la segunda opción.
Se especificará una secuencia de rotaciones que nos llevarán desde el sistema inercial $\{\mathbf{G}\}$ al sistema local $\{\mathbf{B}\}$ utilizando como ángulos de rotación, los ángulos que definen la actitud del quadrotor (ϕ,θ,ψ).

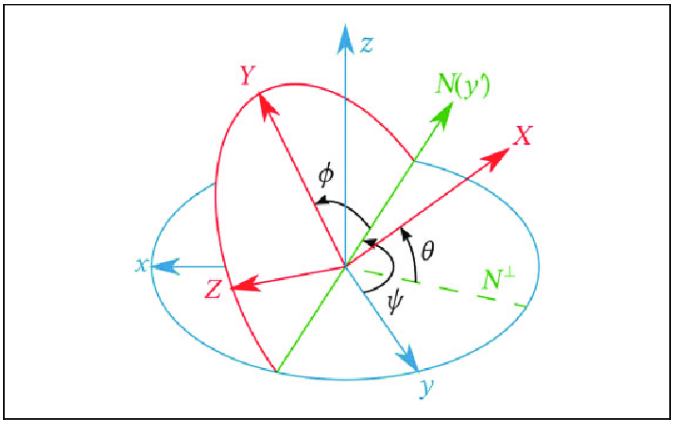


Figura 2.3 Formulación Tait-Bryan.

$$\{G\} \xrightarrow[G u_Z]{\psi_B} \{B_1\} \xrightarrow[B_1 u_Y]{\theta_B} \{B_2\} \xrightarrow[B_2 u_X]{\phi_B} \{B\} \tag{2.7}$$

Primero se rota un ángulo $\psi_{\mathbf{B}}$ alrededor del eje fijo ${}^{\mathbf{G}}\mathbf{u}_{\mathbf{Z}}$,obteniendo un nuevo sistema $\{\mathbf{B_1}\}$. La matriz correspondiente a esta rotación se puede ver en (2.8)

$$\mathbf{R}_{\mathbf{Z}}(\psi) = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.8}$$

La segunda rotación consiste en un ángulo $\theta_{\mathbf{B}}$ alrededor del eje ${}^{\mathbf{B_1}}\mathbf{u}_{\mathbf{Y}}$, la matriz de rotación correspondiente a este giro se puede ver en (2.9)

$$\mathbf{R}_{\mathbf{Y}}(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \tag{2.9}$$

Por último se gira un ángulo $\phi_{\mathbf{B}}$ alrededor del eje ${}^{\mathbf{B_2}}\mathbf{u}_{\mathbf{X}}$, la matriz de rotación correspondiente a este giro se puede ver en (2.10)

$$\mathbf{R}_{\mathbf{X}}(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} \tag{2.10}$$

La matriz de rotación tras realizar los tres giros y que sirve para expresar cualquier vector expresado en ejes cuerpo en ejes globales, viene dada por la expresión:

$$\mathbf{R}_{\mathbf{B}}^{\mathbf{G}} = \mathbf{R}_{\mathbf{X}}(\phi)\cdot\mathbf{R}_{\mathbf{Y}}(\theta)\cdot\mathbf{R}_{\mathbf{Z}}(\psi) = \begin{bmatrix} \cos\psi\cos\theta & \cos\psi\sin\theta\sin\phi - \cos\phi\sin\psi & \sin\phi\sin\psi + \cos\phi\sin\theta\cos\psi \\ \cos\theta\sin\psi & \cos\phi\cos\psi + \sin\phi\sin\theta\sin\psi & \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi \\ -\sin\theta & \cos\theta\sin\phi & \cos\phi\cos\theta \end{bmatrix} \tag{2.11}$$

Al ser matrices ortogonales, la matriz de transformación para pasar vectores de ejes globales a ejes cuerpo, equivale a la transpuesta de la matriz anterior, y viene dada por la expresión (2.12):

$$\mathbf{R}_{\mathbf{G}}^{\mathbf{B}} = \mathbf{R}_{\mathbf{X}}^{\mathbf{T}}(\phi)\cdot\mathbf{R}_{\mathbf{Y}}^{\mathbf{T}}(\theta)\cdot\mathbf{R}_{\mathbf{Z}}^{\mathbf{T}}(\psi) = \begin{bmatrix} \cos\psi\cos\theta & \cos\theta\sin\psi & -\sin\theta \\ \cos\psi\sin\theta\sin\phi - \cos\phi\sin\psi & \cos\phi\cos\psi + \sin\phi\sin\theta\sin\psi & \cos\theta\sin\phi \\ \sin\phi\sin\psi + \cos\phi\sin\theta\cos\psi & \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi & \cos\phi\cos\theta \end{bmatrix} \tag{2.12}$$

Ahora disponemos de dos matrices de transformación, las cuales nos permitirán relacionar vectores en ejes inerciales y vectores en ejes cuerpo. Esto será muy útil a la hora de plantear las ecuaciones dinámicas, ya que no todas las componentes las podremos calcular directamente en el mismo sistema.

La transformación equivalente para velocidades angulares es más compleja, ya que las velocidades angulares vienen expresadas en ejes cuerpo y las variaciones de ángulo vienen expresadas en ejes intermedios. Siguiendo con el procedimiento seguido hasta ahora, la velocidad angular en ejes cuerpo se obtiene como la suma de tres rotaciones en tres ejes de referencia distintos.El vector de variaciones de ángulo se representa como $\dot{\eta}$, y el vector de velocidades angulares como ω.

$$\omega^{\mathbf{B}} = \begin{bmatrix} p \\ q \\ r \end{bmatrix}, \dot{\eta} = \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}, \tag{2.13}$$

Atendiendo a la sección anterior, la rotación de ϕ se produce sobre el eje $\mathbf{X}^{\mathbf{B_2}} \equiv \mathbf{X}^{\mathbf{B}}$.

$$\dot{\phi}^{B} = \mathbf{R}_{\mathbf{X}}(\phi)\cdot\begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix} \tag{2.14}$$

La variación del θ se produce sobre un eje intermedio $\mathbf{Y}^{\mathbf{B_1}}$,en este caso la derivada del pitch en ejes cuerpo se obtiene a partir de la variación θ en $\mathbf{Y}^{\mathbf{B_1}}$ y de una rotación de ϕ sobre $\mathbf{X}^{\mathbf{B_2}}$.

$$\dot{\theta}^{B} = \mathbf{R}_{\mathbf{X}}(\phi)\cdot\mathbf{R}_{\mathbf{Y}}(\theta)\cdot\begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} \tag{2.15}$$

Por último la $\dot{\psi}$ se obtiene a partir de una rotación de ψ en Z^G, una rotación de θ en Y^{B_1} y una rotación de ϕ en X^{B_2}.

$$\dot{\psi}^{B} = \mathbf{R}_{\mathbf{X}}(\phi)\cdot\mathbf{R}_{\mathbf{Y}}(\theta)\cdot\mathbf{R}_{\mathbf{Z}}(\psi)\cdot\begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix} \tag{2.16}$$

La matriz de transformación resultante para obtener las velocidades angulares a partir de las variaciones en ángulos es la siguiente:

$$\omega = \mathbf{W}(\dot{\eta})\cdot\dot{\eta} \tag{2.17}$$

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} 1 & 0 & -\sin\theta \\ 0 & \cos\phi & \sin\phi\cos\theta \\ 0 & -\sin\phi & \cos\phi\cos\theta \end{bmatrix} \cdot \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{2.18}$$

Finalmente para obtener la velocidad angular en ejes inerciales móviles se realizará el proceso inverso. La matriz de transformación no es ortogonal, por tanto, su inversa no coincidirá con su transpuesta.

$$\dot{\eta} = \mathbf{W}^{-1}(\dot{\eta}) \cdot \omega \tag{2.19}$$

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \frac{\sin\phi}{\cos\theta} & \frac{\cos\phi}{\cos\theta} \end{bmatrix} \cdot \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{2.20}$$

Si se supone que los ángulos de Euler son pequeños (≈ 0), la matriz $\mathbf{W}$ se convierte en la matriz identidad ($\mathbf{I}$) y por lo tanto las velocidades angulares son aproximadamente iguales a las derivadas de ángulo.

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} \approx \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{2.21}$$

2.4 Dinámica

En esta sección se detalla la obtención de las ecuaciones dinámicas que describen tanto de la traslación del quadrotor, como de su rotación. Este problema ha sido tratado por una gran variedad de autores, pero en este trabajo se han tomado como referencia los trabajos realizados por Will Selby y Teppo Luukkonen [20] y [15]. Los cuales proponen un método sencillo y de fácil comprensión, que puede ser utilizado como modelo matemático para otros quadrotors con una simple sustitución de los parámetros físicos.

2.4.1 Dinámica Traslacional

Utilizando la **2ª Ley de Newton** calculamos las ecuaciones dinámicas correspondientes al movimiento de traslación del quadrotor. Definimos todas las fuerzas que actúan sobre él y planteamos el equilibrio de fuerzas. Con dichas ecuaciones pretendemos calcular la posición del quadrotor en el sistema de coordenadas inercial $\{\mathbf{G}\}$.

- $\mathbf{P^G}$, $\dot{\mathbf{P}}^{\mathbf{G}}$, $\ddot{\mathbf{P}}^{\mathbf{G}} \rightarrow$ Vector posición y vector velocidad y aceleración en ejes Inerciales $\{\mathbf{G}\}$.

$$\mathbf{P^G} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \qquad \dot{\mathbf{P}}^{\mathbf{G}} = \begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{bmatrix} \qquad \ddot{\mathbf{P}}^{\mathbf{G}} = \begin{bmatrix} \ddot{X} \\ \ddot{Y} \\ \ddot{Z} \end{bmatrix} \tag{2.22}$$

- $\mathbf{F_g^G} \rightarrow$ Fuerza producida por la acción de la gravedad en ejes Inerciales $\{\mathbf{G}\}$.

$$\mathbf{F_g^G} = \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} \tag{2.23}$$

- $\mathbf{F_T^B} \rightarrow$ Fuerza total de empuje, producida por la acción de los motores. Viene expresada en el sistema cuerpo $\{\mathbf{B}\}$.

$$\begin{bmatrix} 0 \\ 0 \\ F_T^B \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ k_{th} \cdot \left(\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2\right) \end{bmatrix} \tag{2.24}$$

- $\mathbf{F_T^G} \rightarrow$ Fuerza total de empuje en el sistema inercial $\{\mathbf{G}\}$. Contamos con la fuerza de empuje en ejes locales, por lo tanto, para calcularla, multiplicamos por la matriz de transformación obtenida en la ecuación 2.11.

$$\mathbf{F_T^G} = \mathbf{R_B^G} \cdot \begin{bmatrix} 0 \\ 0 \\ F_T^B \end{bmatrix} = \begin{bmatrix} (\cos\phi \sin\theta \cos\psi + \sin\phi \sin\psi) \cdot F_T^B \\ (\cos\phi \sin\theta \sin\psi - \sin\phi \cos\psi) \cdot F_T^B \\ (\cos\phi \cos\theta) \cdot F_T^B \end{bmatrix} \tag{2.25}$$

- $\mathbf{F_d^G} \rightarrow$ Fuerza de resistencia aerodinámica en $\{\mathbf{G}\}$.Esta fuerza, cuyo coeficiente será el mismo para las direcciones horizontales, siendo diferente para la dirección vertical, se opone al movimiento del quadrotor.

$$\mathbf{F_d^G} = \begin{bmatrix} Kd_x & 0 & 0 \\ 0 & Kd_y & 0 \\ 0 & 0 & Kd_z \end{bmatrix} \cdot \begin{bmatrix} \dot{X^G} \\ \dot{Y^G} \\ \dot{Z^G} \end{bmatrix} \tag{2.26}$$

Una vez tenemos todas las fuerzas que actúan sobre nuestro quadrotor, planteamos el equilibrio y resolvemos, obteniendo las aceleraciones lineales del quadrotor en el sistema $\{\mathbf{G}\}$.

$$m \cdot \ddot{\mathbf{P}^{\mathbf{G}}} = \mathbf{F_g^G} - \mathbf{F_T^G} - \mathbf{F_d^G} \tag{2.27}$$

$$\begin{bmatrix} \ddot{X^G} \\ \ddot{Y^G} \\ \ddot{Z^G} \end{bmatrix} = \begin{bmatrix} \frac{1}{m} \cdot \left[(\cos\phi \sin\theta \cos\psi + \sin\phi \sin\psi) \cdot F_T^B - Kd_x \cdot \dot{X^G}\right] \\ \frac{1}{m} \cdot \left[(\cos\phi \sin\theta \sin\psi - \sin\phi \cos\psi) \cdot F_T^B - Kd_y \cdot \dot{Y^G}\right] \\ \frac{1}{m} \cdot \left[(\cos\phi \cos\theta) \cdot F_T^B - Kd_z \cdot \dot{Z^G}\right] - g \end{bmatrix} \tag{2.28}$$

2.4.2 Dinámica Rotacional

Las ecuaciones dinámicas correspondientes al movimiento de rotación del quadrotor se obtienen aplicando el balance de los momentos que actúan sobre el quadrotor. Dichas ecuaciones están expresadas en el sistema de coordenadas cuerpo $\{\mathbf{B}\}$.

Se ha tomado como referencia, el procedimiento descrito por el Departamento de Física Aplicada de la Universidad de Sevilla [21]. Partimos de la ecuación de **Euler** para un sólido rígido,

$$\frac{d\vec{\mathbf{L}}}{dt} = \vec{\mathbf{M}} \quad , \quad \vec{\mathbf{L}} = \mathbf{I} \cdot \vec{\omega} \tag{2.29}$$

Se podría pensar que el tensor de inercias es constante, debido a que es un sólido rígido, pero no es así. Este problema se resuelve al emplear un sistema de referencia ligado al sólido, quedando de la siguiente forma:

$$\frac{d\vec{\mathbf{L}}}{dt} + \vec{\omega} \times \vec{\mathbf{L}} = \vec{\mathbf{M}} \tag{2.30}$$

Al aplicar dicha ecuación sobre el sistema de coordenadas cuerpo $\{\mathbf{B}\}$ que está ligado a nuestro vehículo:

$$\frac{d(\mathbf{I} \cdot \omega^B)}{dt} + \omega^{\mathbf{B}} \times (\mathbf{I} \cdot \omega^{\mathbf{B}}) = \mathbf{T}^{\mathbf{B}}_{\mathbf{m}} - \mathbf{T}^{\mathbf{B}}_{\mathbf{g}} \tag{2.31}$$

Donde:

- $\mathbf{I}^{\mathbf{B}} \rightarrow$ Tensor de inercias en el sistema cuerpo $\{B\}$. Resultante de la rotación de la hélice junto con la rotación del cuerpo.

$$\mathbf{I}^{\mathbf{B}} = \begin{bmatrix} I_{XX} & 0 & 0 \\ 0 & I_{YY} & 0 \\ 0 & 0 & I_{ZZ} \end{bmatrix} \tag{2.32}$$

- $\omega^{\mathbf{B}}, \dot{\omega}^{\mathbf{B}} \rightarrow$ Vector de velocidades angulares y vector de aceleraciones angulares en el sistema $\{B\}$.

$$\omega^{\mathbf{B}} = \begin{bmatrix} p \\ q \\ r \end{bmatrix} \quad , \quad \dot{\omega^{\mathbf{B}}} = \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} \tag{2.33}$$

- $\mathbf{T}^{\mathbf{B}}_{\mathbf{m}} \rightarrow$Vector de pares generados por la acción de los motores en los ejes del sistema $\{B\}$.

$$\mathbf{T}^{\mathbf{B}}_{\mathbf{m}} = \begin{bmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} \tag{2.34}$$

$$\tau_\phi = \frac{\sqrt{2}}{2} \cdot l \cdot k_{th}(-\omega_0^2 + \omega_1^2 + \omega_2^2 - \omega_3^2) \tag{2.35}$$

$$\tau_\theta = \frac{\sqrt{2}}{2} \cdot l \cdot k_{th}(-\omega_0^2 + \omega_1^2 - \omega_2^2 + \omega_3^2) \tag{2.36}$$

$$\tau_\psi = k_d(-\omega_0^2 - \omega_1^2 + \omega_2^2 + \omega_3^2) \tag{2.37}$$

- $\mathbf{T}^{\mathbf{B}}_{\mathbf{g}} \rightarrow$Vector de pares generados por el efecto giroscópico en el sistema $\{\mathbf{B}\}$. Estos momentos se generan debido al giro de las hélices con velocidad ω relativa, respecto a otro marco de referencia con velocidad angular ($\dot{\theta}$ y $\dot{\phi}$).[23]

$$\mathbf{T_g^B} = \omega \times \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \cdot \sum_{i=1}^{4} J_r \cdot \omega_i \tag{2.38}$$

$$\mathbf{T_g^B} = \begin{bmatrix} \dot{\theta} \cdot J_r \cdot (-\omega_0 - \omega_1 + \omega_2 + \omega_3) \\ -\dot{\phi} \cdot J_r \cdot (-\omega_0 - \omega_1 + \omega_2 + \omega_3) \\ 0 \end{bmatrix} \tag{2.39}$$

En la Sección 2.3 hemos obtenido una matriz que nos permite relacionar las derivadas de ángulo con las velocidades angulares en ejes cuerpo. Como estamos suponiendo que el quadrotor está volando de forma estable, con variaciones pequeñas de ángulos, se supone que:

$$\omega = \dot{\eta} \quad , \quad \begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{2.40}$$

Tras analizar cada una de las componentes de la ecuación 2.31, desarrollamos y sustituimos:

$$\mathbf{I} \cdot \dot{\omega^{\mathbf{B}}} + \omega^{\mathbf{B}} \times (\mathbf{I} \cdot \omega^{\mathbf{B}}) = \mathbf{T_m^B} - \mathbf{T_g^B} \tag{2.41}$$

$$\begin{bmatrix} I_{XX} & 0 & 0 \\ 0 & I_{YY} & 0 \\ 0 & 0 & I_{ZZ} \end{bmatrix} \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} + \begin{bmatrix} q \cdot r \cdot (I_{ZZ} - I_{YY}) \\ p \cdot r \cdot (I_{XX} - I_{ZZ}) \\ p \cdot q \cdot (I_{YY} - I_{XX}) \end{bmatrix} = \begin{bmatrix} T_\phi \\ T_\theta \\ T_\psi \end{bmatrix} - \begin{bmatrix} q \cdot J_r \cdot (-\omega_0 - \omega_1 + \omega_2 + \omega_3) \\ -p \cdot J_r \cdot (-\omega_0 - \omega_1 + \omega_2 + \omega_3) \\ 0 \end{bmatrix} \tag{2.42}$$

Atendiendo a la suposición anterior, y despejando las aceleraciones angulares:

$$\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} \frac{1}{I_{XX}} \cdot [(I_{YY} - I_{ZZ}) \cdot q \cdot r - J_r \cdot q \cdot (\omega_1 - \omega_2 + \omega_3 - \omega_4) + \tau_\phi] \\ \frac{1}{I_{YY}} \cdot [(I_{ZZ} - I_{XX}) \cdot p \cdot r + J_r \cdot p(\omega_1 - \omega_2 + \omega_3 - \omega_4) + \tau_\theta] \\ \frac{1}{I_{ZZ}} \cdot [(I_{XX} - I_{YY}) \cdot p \cdot q + \tau_\psi] \end{bmatrix} \tag{2.43}$$

Estas ecuaciones se han desarrollado en el marco del cuerpo $\{\mathbf{B}\}$, debido a que la IMU, proporciona medidas de aceleraciones y velocidades en este sistema, y estas medidas serán utilizadas como realimentación en la estrategia de control que se desarrollará a lo largo del presente documento.

2.5 Linealización

Las ecuaciones obtenidas a lo largo de la sección anterior, tanto la que describe la dinámica traslacional del quadrotor (2.28) como la que describe la dinámica rotacional (2.43), son ecuaciones no lineales y altamente acopladas. En esta sección se asumen una serie de suposiciones con el fin de simplificar dichas ecuaciones y obtener un modelo matemático que nos permita diseñar los controladores para nuestra estrategia de control.

La linealización de las ecuaciones dinámicas se realiza en torno a un punto de equilibrio. Este punto de operación equivale a un punto en el que el quadrotor se encuentra estable en el aire, con velocidades lineales y ángulos de inclinación muy pequeños. Además se consideran los efectos aerodinámicos despreciables.

Los efectos despreciados pueden ser tratados como perturbaciones en el sistema y se pueden compensar con un diseño apropiado.

$$\dot{X^G} \approx \dot{Y^G} \approx \dot{Z^G} \approx 0 \tag{2.44}$$

$$\phi \approx \theta \approx \psi \approx 0 \tag{2.45}$$

En este punto de operación, la fuerza de empuje total generada por los motores, debe ser igual a la fuerza ejercida por la gravedad en sentido contrario, de forma que el vehículo se mantenga estable en el aire:

$$F_T^B \approx m \cdot g \tag{2.46}$$

Aplicando estas suposiciones a la ecuación 2.28 y a la ecuación 2.43 deducidas en la sección anterior, obtenemos unas ecuaciones simplificadas con las que poder diseñar estrategias básicas de control:

$$\begin{bmatrix} \ddot{X}^G \\ \ddot{Y}^G \\ \ddot{Z}^G \end{bmatrix} = \begin{bmatrix} \frac{1}{m} \cdot \theta \cdot F_T^B \\ -\frac{1}{m} \cdot \phi \cdot F_T^B \\ \frac{1}{m} \cdot F_T^B - g \end{bmatrix} \tag{2.47}$$

$$\begin{bmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} \frac{1}{I_{XX}} \cdot \tau_\phi \\ \frac{1}{I_{YY}} \cdot \tau_\theta \\ \frac{1}{I_{ZZ}} \cdot \tau_\psi \end{bmatrix} \tag{2.48}$$

Como se puede observar, las aceleraciones angulares han quedado desacopladas de la actuación del resto de los ángulos, de forma que dependen únicamente del par resultante en su eje de giro por la acción de los motores. Los movimientos en los ejes X^G e Y^G se puede controlar variando independientemente los valores de θ y ϕ respectivamente. Y por último, la aceleración en Z^G depende únicamente del empuje generado por los motores y de la gravedad.

3 Estrategia de Control

En el presente capítulo se va a detallar la estructura de la estrategia de control considerada para obtener un modo de vuelo autónomo, detallando desde el control a nivel más bajo hasta el control de más alto nivel.

Nuestra estrategia se basa en tres bucles de control anidados, basados en el regulador más común utilizado en procesos industriales, el controlador *Proportional-Integral-Derivative* (**PID**). El control de más bajo nivel se conoce como el *Rate Control*, es el control más rápido y recibe las velocidades angulares y devuelve como salidas los momentos definidos en las ecuaciones 2.3, 2.4 y 2.5. Por encima nos encontramos el *Control de Actitud* y el *Control de Altura*, que como su propio nombre indica se encargan de controlar la actitud y la altura del quadrotor. Del control de Altura obtenemos la fuerza de empuje (l2.2) y del control de actitud obtenemos las velocidades angulares (q,p,r) que serán utilizadas por el *Rate Control*. Por último, en el nivel más alto se encuentra el *Control de Posición*, que a partir de una posición definida en ejes globales $\{\mathbf{G}\}$, obtenemos los ángulos de roll(ϕ) y pitch(θ) utilizados por el *Control de Actitud*. Con respecto al yaw (ψ), al tratarse de un vuelo autónomo que no requiere de la acción de un piloto, lo mantendremos a 0.

3.1 Control PID

Como se comentaba en la introducción del capítulo es uno de los controladores más conocidos y utilizados en control industrial, y del cual se puede encontrar una gran cantidad de información ([8], [17]). Este tipo de control es la base de este proyecto, y en esta sección se pretende explicar su funcionamiento y la razón por la que se ha escogido este tipo de controlador.

Un controlador o regulador PID permite controlar un sistema en lazo cerrado, permitiendo que el sistema alcance un valor deseado, más conocido como *referencia* $(r(t))$. Este tipo de controlador calcula la acción de control a través de la diferencia entre la $r(t)$ y la medida de los sensores, o salida del sistema $y(t)$, esta diferencia es conmunmente conocida como error ($e(t)$). Por lo tanto, el objetivo del controlador, es reducir a cero el error cometido en un tiempo determinado.

Este controlador está compuesto por tres elementos claramente diferenciables, por lo tanto a partir de la estructura del PID, en ausencia de alguna de esos tres componentes, podemos obtener diferentes controles como PI o PD.

3.1.1 Acción Proporcional

Como su propio nombre indica, esta acción es proporcional a la señal de error, multiplica la señal de error por una constante conocida como $\mathbf{K_p}$. Esta acción deja de influir en régimen permanente cuando el error tiende a cero.

$$u(t) = K_p \cdot e(t) \tag{3.1}$$

Aumentar la acción proporcional K_p tiene los siguientes efectos:

- Aumenta la velocidad de respuesta del sistema.
- Disminuye el error del sistema en régimen permanente
- Aumenta la inestabilidad del sistema

3.1.2 Acción Integral

Esta acción evoluciona a lo largo del tiempo, acumulando el error mientras este sea distinto de cero. Con esto se consigue reducir el error de régimen permanente a cero, de forma que el sistema alcance la referencia.Se ajusta mediante T_i, que se denomina tiempo integral.

$$u(t) = \frac{K_p}{T_i} \int e(t)dt \tag{3.2}$$

Utilizar la acción integral añade una cierta inercia al sistema, por lo tanto, lo hace más inestable. Aumentar la acción integral provoca:

- Elimina el error en régimen permanente del sistema.
- Aumenta la inestabilidad del sistema.
- Aumenta levemente la velocidad del sistema.

3.1.3 Acción Derivativa

Esta acción es proporcional a la derivada de la señal de error. Tiene carácter de previsión,pero nunca se utiliza por sí sola, debido a que sólo es eficaz durante períodos transitorios. Se regula mediante la constante T_d (tiempo derivativo).

$$u(t) = K_p \cdot T_d \cdot de(t) \tag{3.3}$$

Aumentar la acción derivativa supone:

- Aumentar la estabilizad del sistema.
- Disminuye un poco la velocidad del sistema.
- El error en régimen permanente no se ve afectado.

3.1.4 Ecuación PID

La combinación de las tres acciones descritas anteriormente, da como resultado una acción que reune las ventajas de cada una de las tres acciones individuales. Como también se ha dicho al inicio del capítulo, se

podría prescindir de alguna de las acciones, de forma que esta estructura sea lo más eficaz posible. En esta sección se muestra la ecuación genérica del PID:

$$u(t) = K_p \cdot (e(t) + K_i \int e(t)dt + K_d \frac{de(t)}{dt}) \tag{3.4}$$

Donde:

$$K_i = \frac{K_p}{T_i} \tag{3.5}$$

$$K_d = K_p \cdot T_d \tag{3.6}$$

La implementación de los controladores se realizará en tiempo discreto, con un determinado tiempo de muestreo T_m de forma que la ecuación queda:

$$u(t) = K_p \cdot (e(k) + K_i \cdot \sum e(k)T_m + K_d \frac{e(k) - e(k-1)}{T_m}) \tag{3.7}$$

3.2 Rate Control

Este es el control de más bajo nivel, a partir de las salidas de este control se calcularán las velocidades de los motores, que son las actuaciones de nuestro sistema. Para poder diseñar los parámetros de nuestro controlador, primero debemos obtener un modelo.

3.2.1 Función de trasferencia

Partimos de las ecuaciones linealizadas en el capítulo 2, concretamente de la ecuación 2.48.

$$f_p = I_{XX}\ddot{\phi} - \tau_\phi \tag{3.8}$$

$$f_q = I_{YY}\ddot{\theta} - \tau_\theta \tag{3.9}$$

$$f_r = I_{ZZ}\ddot{\psi} - \tau_\psi \tag{3.10}$$

Se le aplicará el desarrollo de Taylor a dichas ecuaciones en torno al punto de equilibrio:

$$\left.\frac{\partial f_\phi}{\partial \ddot{\phi}}\right|_{\ddot{\phi}_{eq}} \ddot{\phi}(t) - \left.\frac{\partial f_\phi}{\partial \tau_\phi}\right|_{\tau_{\phi eq}} \tau_\phi(t) = 0 \tag{3.11}$$

$$\left.\frac{\partial f_\theta}{\partial \ddot{\theta}}\right|_{\ddot{\theta}_{eq}} \ddot{\theta}(t) - \left.\frac{\partial f_\theta}{\partial \tau_\theta}\right|_{\tau_{\theta eq}} \tau_\theta(t) = 0 \tag{3.12}$$

$$\left.\frac{\partial f_\psi}{\partial \ddot{\psi}}\right|_{\ddot{\psi}_{eq}} \ddot{\psi}(t) - \left.\frac{\partial f_\psi}{\partial \tau_\psi}\right|_{\tau_{\psi eq}} \tau_\psi(t) = 0 \tag{3.13}$$

Y por último se calcula la transformada de Laplace ($\mathscr{L}$) de las ecuaciones en diferencias obtenidas, dando como resultado las funciones de transferencia para el cálculo de los controladores. Atendiendo a la aproximación 2.40, estos modelos tendrán como salidas las velocidades angulares (p,q,r), y como entradas los momentos ($\tau_\phi, \tau_\theta, \tau_\psi$):

$$G_p(s) = \frac{p(s)}{\tau_\phi(s)} = \frac{1}{I_{XX} \cdot s} \tag{3.14}$$

$$G_q(s) = \frac{q(s)}{\tau_\theta(s)} = \frac{1}{I_{YY} \cdot s} \tag{3.15}$$

$$G_r(s) = \frac{r(s)}{\tau_\psi(s)} = \frac{1}{I_{ZZ} \cdot s} \tag{3.16}$$

3.2.2 Lugar de las Raíces

La dinámica de un sistema de control en lazo cerrado está íntimamente ligada con la situación de los polos de su función de transferencia, es decir, la distribución de las raíces de su ecuación característica en el plano s. Utilizando el LR diseñaremos los parámetros de los reguladores encargados de controlar la salida del sistema.

Definición 3.2.1 (Lugar de las Raíces) El Lugar de las Raíces se define como el lugar geométrico que recorren los polos de un sistema en lazo cerrado a medida que la ganancia proporcional K de su correspondiente función de transferencia en lazo abierto varía de 0 a ∞.

Nos proporciona una medida de la sensibilidad de las raíces del sistema y la influencia que puede tener un parámetro en la dinámica del mismo. En otras palabras, nos da una medida de la robustez del sistema. El LR nos permite determinar la posición de los polos de la función de transferencia en lazo cerrado a partir de la función de transferencia en bucle abierto.

Comenzaremos por dibujar el lugar de las raíces, para esto se utiliza el programa MATLAB® el cuál posee un comando que se encarga de dibujarlo, tomando como argumento la función de transferencia en bucle abierto.

```
G = tf([1],[Ixx 0]);
rltool(G);
```

Este comando es muy útil, ya que a parte de ver el lugar de las raíces, podemos ver la respuesta del sistema frente a una entrada escalón unitario:

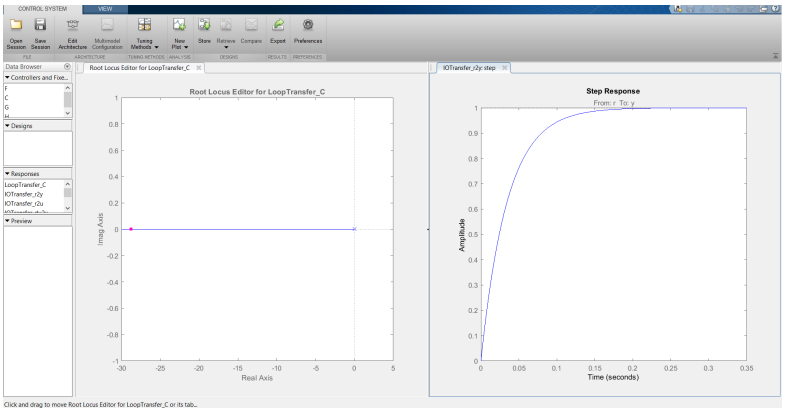


Figura 3.1 rltool.

Disponemos también de un diagrama de bloques que nos indica cada uno de los bloques que componen el bucle cerrado, y los cuales podemos modificar añadiendo polos y ceros en el lugar de las raíces para modificar la salida del sistema acorde a nuestras necesidades.

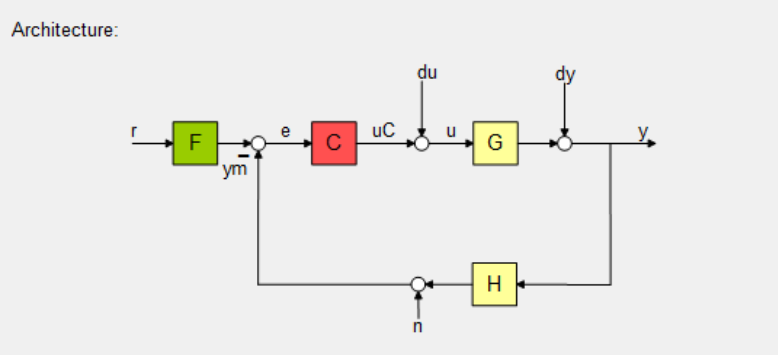


Figura 3.2 Arquitectura.

El objetivo es añadir polos y ceros a la función de transferencia del controlador **C**, hasta obtener un controlador de tipo PI o PID, puede que no sea necesario añadir la parte derivativa en algunas ocasiones, pero la parte integral será siempre necesaria, ya que no queremos error en régimen permanente. Comenzaremos añadiendo un integrador, o lo que es lo mismo, un polo en $s = 0$, pero el integrador por si solo no es suficiente, por lo tanto añadimos un cero en el eje real negativo, por ejemplo en $s = 1$ y nos llevamos las ganancias a dicho eje, para reducir la sobreoscilación.

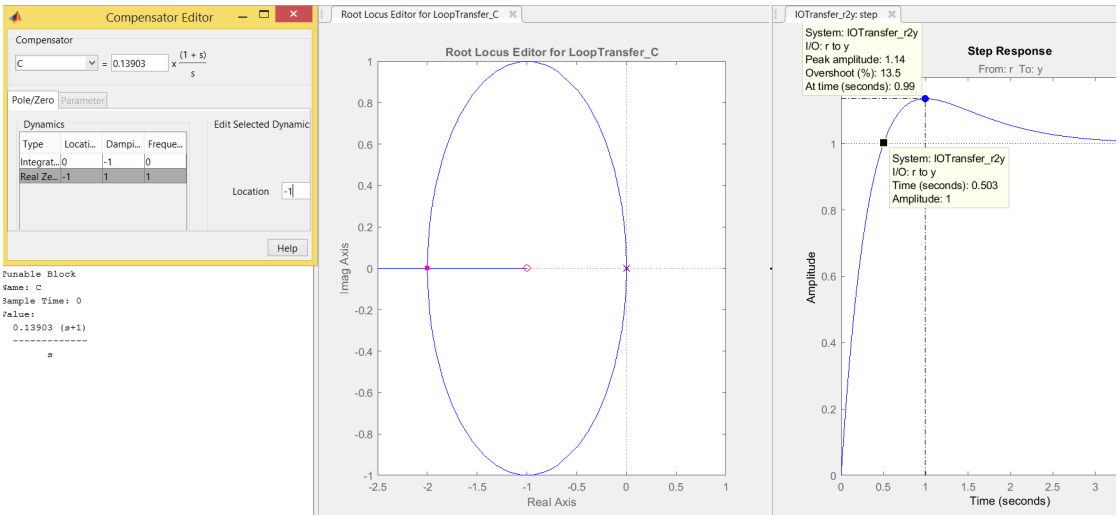


Figura 3.3 Diseño PID para control de p.

Cabe recordar que en este control la respuesta del sistema debe de ser rápida, es el control de más bajo nivel. Vemos que el tiempo de respuesta está en torno a los $0.5(s)$, con una sobreoscilación del $13\,\%$, la velocidad no está mal, pero no queremos tanta sobreoscilación, es deseable eliminarla. Como este es el bucle de control más bajo, en lugar de añadir otro cero y hacer más compleja la dinámica del controlador, podemos añadir un prefiltro a la referencia. Este prefiltro ralentiza la respuesta, por lo tanto, a parte de

añadir el prefiltro, debemos de modificar la posición del cero, desplazándolo hacia la izquierda en el eje real negativo, de forma que compensemos la pérdida de velocidad provocada por el prefiltro.

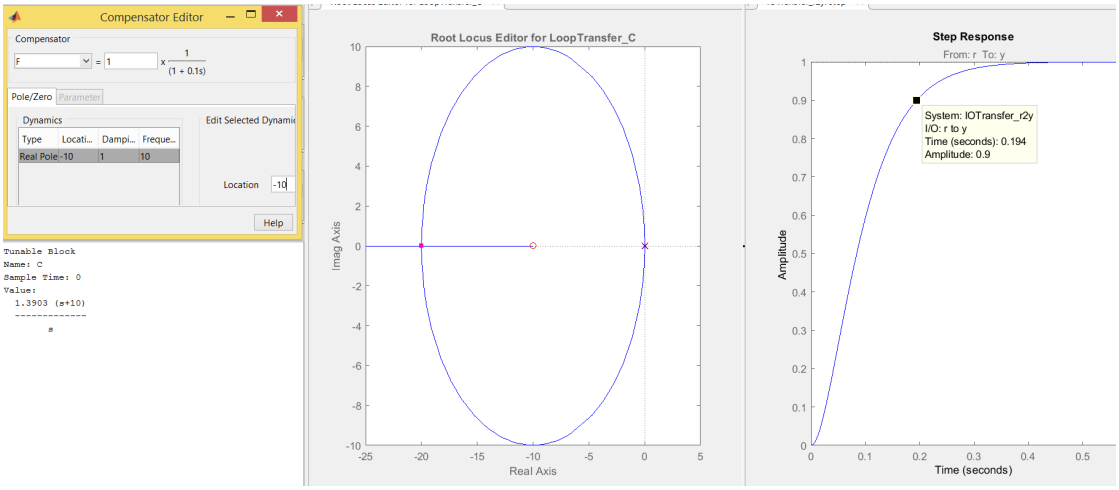


Figura 3.4 Diseño PI + Prefiltro.

Se puede ver como al situar el polo en $s = -10$ el sistema se hace más rápido, y añadiendo el prefiltro, eliminamos la sobreoscilación de la respuesta. Este prefiltro es provisional, ya que en los bucles superiores, se podrá eliminar este prefiltro y controlar la respuesta sin él. Como resultado obtenemos un controlador PI y un prefiltro:

$$C_p(s) = 13.903 \cdot \frac{1+0.1s}{s} \tag{3.17}$$

$$F_p(s) = \frac{1}{1+0.1s} \tag{3.18}$$

Se ha tomado como referencia para explicar el procedimiento el diseño para el controlador de p, para las otras velocidades (q y r), cuyas funciones de transferencia son similares, el procedimiento será el mismo.

Para el control de q también obtenemos un PI y un prefiltro (F) para la referencia:

$$C_q(s) = 18.357 \cdot \frac{1+0.1s}{s} \tag{3.19}$$

$$F_q(s) = \frac{1}{1+0.1s} \tag{3.20}$$

Por último para el control de r:

$$C_r(s) = 39.08 \cdot \frac{1+0.1s}{s} \tag{3.21}$$

$$F_r(s) = \frac{1}{1+0.1s} \tag{3.22}$$

Finalmente, para poder implementar los controladores calculados según la ecuación (3.4)), tenemos que calcular las componentes T_i,T_d,K_p,K_i y K_d. Para ello expresamos la ecuación del PID en formato de función de transferencia y la comparamos con la función de nuestro PI. Tomando como referencia C_p :

$$13.903 \cdot \frac{1+0.1s}{s} = K_p \cdot \frac{T_d T_i s^2 + T_i s + 1}{T_i s} \tag{3.23}$$

Para este cálculo se ha hecho una sencilla función en Matlab, que tiene tres argumentos de entrada, la ganancia del controlador y los valores que multiplican a s de cada uno de los ceros del controlador, y como salida, las componentes que se nombraban anteriormente (T_i,T_d,K_p,K_i y K_d):

Código 3.1 calculo_PID.

```
function [TI,TD,KP,KI,KD] = calculo_PID(K,z1,z2)

TDI = (z1*z2);
TI = (z1+z2);

TD = TDI/TI;

KP = K*TI;
KI = K;
KD = KP*TD;

end
```

Tabla 3.1 Parámetros de los controladores PI del Rate Control.

Descripción	Tipo	T_i	T_d	K_p	K_i	K_d
Control de p	PI	0.1	0	1.3093	13.093	0
Control de q	PI	0.1	0	1.8375	18.357	0
Control de r	PI	0.1	0	3.9080	39.080	0

3.2.3 Experimentos en Matlab

Para verificar que el lugar de las raíces nos mostraba la respuesta del sistema en bucle cerrado, y que efectivamente, con los controladores que hemos diseñado, la respuesta del sistema será la que nos mostraba, se realizará un script en matlab para ver como responde este bucle de control. Se realizará una simulación utilizando las ecuaciones no linealizadas (2.43). Se ha implementado el bucle de control en diferentes funciones (.m), con el fin de que este código sea reutilizable y pueda ser aplicado tanto a una simulación basada solamente en código, como una una simulación en *Simulink* mediante diagrama de bloques, ya que este tipo de implementaciones, nos permiten entender la simulación a nivel visual.

Los controladores diseñados se han implementado en la función *rate_control.m*. Como ya se comentó anteriormente, los controladores se implementan en discreto, utilizando la ecuación 3.7. El tiempo de muestreo (Tm) utilizado, observando los tiempos de respuesta, es de $0.01(s)$ ($100Hz$), el cuál es suficiente, ya que incluso el control de nivel más bajo tardará unas 19 muestras en alcanzar la referencia.

Primero se filtra la referencia, a esa referencia filtrada se le resta la velocidad angular actual. Con el error, y la ecuación del PID(3.7) calculamos la señal de actuación. Y por último realizamos la saturación de la señal de control.

Código 3.2 rate_control.m.

```
function out = rate_control (in)

p_des = in(1);
q_des = in(2);
r_des = in(3);

global erle;

%% Control de variación del roll

erle.p_des_filt = (erle.p_F_TI/(erle.p_F_TI+erle.Tm))*erle.p_des_filt_1 + (
    erle.Tm/(erle.p_F_TI+erle.Tm))*p_des;
p_ek = erle.p_des_filt - erle.p;
% Incremento de la integral del error
erle.p_Int_ek = erle.p_Int_ek + erle.Tm*p_ek;
% Controlador PI
U2 = erle.p_KP*(p_ek + (1/erle.p_TI)*erle.p_Int_ek + erle.p_TD*((p_ek-erle.
    p_ek_1)/erle.Tm));
% Saturación
U2 = min(erle.U2_max,max(erle.U2_min,U2));
erle.p_ek_1 = p_ek;
erle.p_des_filt_1 = erle.p_des_filt;

%% Control de variación del roll
erle.q_des_filt = (erle.q_F_TI/(erle.q_F_TI+erle.Tm))*erle.q_des_filt_1 + (
    erle.Tm/(erle.q_F_TI+erle.Tm))*q_des;
q_ek = erle.q_des_filt - erle.q;
% Incremento de la integral del error
erle.q_Int_ek = erle.q_Int_ek + erle.Tm*q_ek;
% Controlador PI
U3 = erle.q_KP*(q_ek + (1/erle.q_TI)*erle.q_Int_ek + erle.q_TD*((q_ek-erle.
    q_ek_1)/erle.Tm));
% Saturación
U3 = min(erle.U3_max,max(erle.U3_min,U3));
erle.q_ek_1 =q_ek;
erle.q_des_filt_1 = erle.q_des_filt;

%% Control de variación del roll
erle.r_des_filt = (erle.r_F_TI/(erle.r_F_TI+erle.Tm))*erle.r_des_filt_1 + (
    erle.Tm/(erle.r_F_TI+erle.Tm))*r_des;
r_ek = erle.r_des_filt - erle.r;
% Incremento de la integral del error
erle.r_Int_ek = erle.r_Int_ek + erle.Tm*r_ek;
```

```
% Controlador PI
U4 = erle.r_KP*(r_ek + (1/erle.r_TI)*erle.r_Int_ek + erle.r_TD*((r_ek-erle.
    r_ek_1)/erle.Tm));
% Saturación
U4 = min(erle.U4_max,max(erle.U4_min,U4));
erle.r_ek_1 = r_ek;
erle.r_des_filt_1 = erle.r_des_filt;

out = [U2,U3,U4];

end
```

Las salidas del código anterior, deben pasar por un bloque encargado de la saturación y del cálculo de las velocidades de los motores, cuya implementación se realiza en *saturacion_actuaciones.m*. El cuerpo de esta función dependerá de la configuración elegida para el quadrotor, en este caso una configuración en **X**.

Código 3.3 saturacion_actuaciones.m.

```
function out = saturacion_actuaciones (in)

U1 = in(1);
U2 = in(2);
U3 = in(3);
U4 = in(4);

global erle;

w0_2 = U1/(4*erle.Kt) - U2/(2*sqrt(2)*erle.l*erle.Kt) - U3/(2*sqrt(2)*erle.l*
    erle.Kt) - U4/(4*erle.Kd);
w1_2 = U1/(4*erle.Kt) + U2/(2*sqrt(2)*erle.l*erle.Kt) + U3/(2*sqrt(2)*erle.l*
    erle.Kt) - U4/(4*erle.Kd);
w2_2 = U1/(4*erle.Kt) + U2/(2*sqrt(2)*erle.l*erle.Kt) - U3/(2*sqrt(2)*erle.l*
    erle.Kt) + U4/(4*erle.Kd);
w3_2 = U1/(4*erle.Kt) - U2/(2*sqrt(2)*erle.l*erle.Kt) + U3/(2*sqrt(2)*erle.l*
    erle.Kt) + U4/(4*erle.Kd);

if w0_2 > erle.w_max^2
w0_2 = erle.w_max^2;
end
if w0_2 < erle.w_min^2
w0_2 = erle.w_min^2;
end
if w1_2 > erle.w_max^2
w1_2 = erle.w_max^2;
end
if w1_2 < erle.w_min^2
w1_2 = erle.w_min^2;
end
```

```
if w2_2 > erle.w_max^2
w2_2 = erle.w_max^2;
end
if w2_2 < erle.w_min^2
w2_2 = erle.w_min^2;
end

if w3_2 > erle.w_max^2
w3_2 = erle.w_max^2;
end
if w3_2 < erle.w_min^2
w3_2 = erle.w_min^2;
end

%% Calculo las velocidades de los motores en (Deg/s)
erle.w0 = sqrt(w0_2);
erle.w1 = sqrt(w1_2);
erle.w2 = sqrt(w2_2);
erle.w3 = sqrt(w3_2);

%% Señales de control
U1 = erle.Kt*(erle.w0^2+erle.w1^2+erle.w2^2+erle.w3^2);
U2 = (sqrt(2)*erle.l*erle.Kt/2)*(+erle.w1^2+erle.w2^2-erle.w0^2-erle.w3^2);
U3 = (sqrt(2)*erle.l*erle.Kt/2)*(-erle.w0^2-erle.w2^2+erle.w1^2+erle.w3^2);
U4 = erle.Kd*(-erle.w0^2-erle.w1^2+erle.w2^2+erle.w3^2);

out = [U1,U2,U3,U4];
end
```

Y por último se implementa la dinámica del quadrotor. Se han utilizado las ecuaciones no lienales con el fin de comprobar la fiabilidad de los controladores en un sistema lo más real posible, así podremos también verificar la fiabilidad de los modelos utilizados para el diseño. Para este experimento solamente necesitamos la dinámica rotacional (2.43).

Código 3.4 ecuaciones_dinamicas.m.

```
function out = ecuaciones_dinamicas (in)

U1 = in(1);
U2 = in(2);
U3 = in(3);
U4 = in(4);

global erle;

erle.p_d = ((erle.Iyy - erle.Izz)*erle.q*erle.r - erle.Jr*erle.q*(-erle.w0-
    erle.w1+erle.w2+erle.w3) + U2)/erle.Ixx;
erle.q_d = ((erle.Izz - erle.Ixx)*erle.p*erle.r + erle.Jr*erle.p*(-erle.w0-
    erle.w1+erle.w2+erle.w3) + U3)/erle.Iyy;
```

```
erle.r_d = ((erle.Ixx - erle.Iyy)*erle.p*erle.q + U4)/erle.Izz;
%     Cálculo de p,q y r
erle.p = erle.p_d * erle.Tm + erle.p;
erle.q = erle.q_d * erle.Tm + erle.q;
erle.r = erle.r_d * erle.Tm + erle.r;

roll_d = erle.p + sin(erle.roll)*tan(erle.pitch)*erle.q + cos(erle.roll)*tan(
    erle.pitch)*erle.r;
pitch_d = cos(erle.roll)*erle.q - sin(erle.roll)*erle.r;
yaw_d = sin(erle.roll)/cos(erle.pitch) * erle.q + cos(erle.roll)/cos(erle.
    pitch) * erle.r;

% Cálculo de los ángulos
erle.roll = roll_d *erle.Tm + erle.roll;
erle.pitch = pitch_d * erle.Tm + erle.pitch;
erle.yaw = yaw_d * erle.Tm + erle.yaw;

%% Variables para graficar
erle.roll_plot(erle.indice) = erle.roll*erle.Rad_Deg;
erle.p_plot(erle.indice) = erle.p;
erle.U2_plot(erle.indice) = U2;

erle.pitch_plot(erle.indice) = erle.pitch*erle.Rad_Deg;
erle.q_plot(erle.indice) = erle.q;
erle.U3_plot(erle.indice) = U3;

erle.yaw_plot(erle.indice) = erle.yaw*erle.Rad_Deg;
erle.r_plot(erle.indice) = erle.r;
erle.U4_plot(erle.indice) = erle.U4;

erle.U1_plot(erle.indice) = U1;

erle.indice = erle.indice + 1;

p = erle.p;
q = erle.q;
r = erle.r;

out = [p,q,r];

end
```

Tenemos que ponerlo todo junto para poder realizar la simulación, como ya se ha comentado, la forma en la que se han implementado las funciones permiten que la simulación se haga solamente mediante código, o utilizando el código con un diagrama de bloques de simulink. Se mostrarán las dos posibilidades, ya que son bastante parecidas, pero la simulación con simulink, con el diagrama de bloques, nos facilita la comprensión del funcionamiento.

Código 3.5 Simulacion.m en ausencia de simulink.

```
clear all;
close all;
clc;

global erle;
erle_variables;

erle.contador = 0;
erle.T_simulacion = 15;% segundos
erle.T_escalon_p = 2;%(segundios);
erle.T_escalon_q = 5;%(segundios);
erle.T_escalon_r = 7;%(segundios)
U1 = erle.g*erle.m;

for time = 0:erle.Tm:erle.T_simulacion
%% Referencias
% Señal de entrada p_des
if(time < erle.T_escalon_p)
erle.p_des = 0;
end
if(time >= erle.T_escalon_p)

erle.p_des = 20*(erle.Deg_Rad);
end
% Señal de entrada q_des
if(time < erle.T_escalon_q)
erle.q_des = 0;
end
if(time >= erle.T_escalon_q)
erle.p_des = 0;
erle.q_des = 20*(erle.Deg_Rad);
end
% Señal de entrada r_des
if(time < erle.T_escalon_r)
erle.r_des = 0;
end
if(time >= erle.T_escalon_r)
erle.p_des = 0;
erle.q_des = 0;
erle.r_des = 20*(erle.Deg_Rad);
end
%% Lazo de control

U_des = rate_control([erle.p_des,erle.q_des,erle.r_des]);
U_sat = saturacion_actuaciones([U1,U_des]);
rate = ecuaciones_dinamicas(U_sat);

%% Variables estáticas
```

```
erle.p_des_plot(erle.indice) = erle.p_des;
erle.q_des_plot(erle.indice) = erle.q_des;
erle.r_des_plot(erle.indice) = erle.r_des;
erle.time_plot(erle.indice) = time;
erle.indice = erle.indice + 1;
end
```

Código 3.6 Simulacion.m + simulink.

```
clear all;
close all;
clc;

addpath Simulador

global erle;
erle_variables;

erle.T_simulacion = 15;% segundos
erle.T_escalon_yaw = 7;%(segundios)
erle.T_escalon_p = 2;%(segundios);
erle.T_escalon_q = 5;%(segundios);
erle.T_escalon_r = 7;%(segundios)

erle.p_des = 20*(erle.Deg_Rad);
erle.q_des = 10*(erle.Deg_Rad);
erle.r_des = 0*(erle.Deg_Rad);

SimOut = sim('diagrama_de_bloques');
```

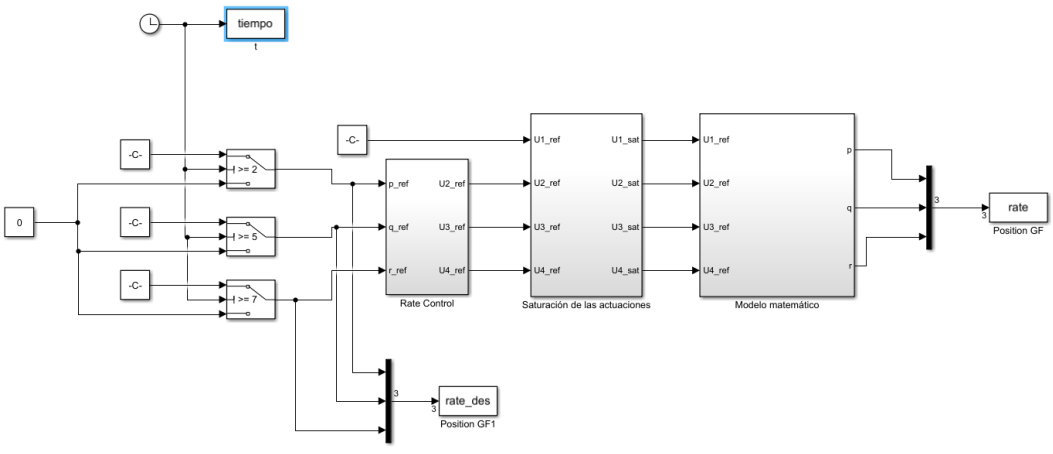


Figura 3.5 Diagrama de bloques Rate Control.

Comenzamos las pruebas por mantener una velocidad constante en cada uno de los ejes del sistema B, y vemos como se comporta el sistema. Primero mantenemos una velocidad p constante en X_B:

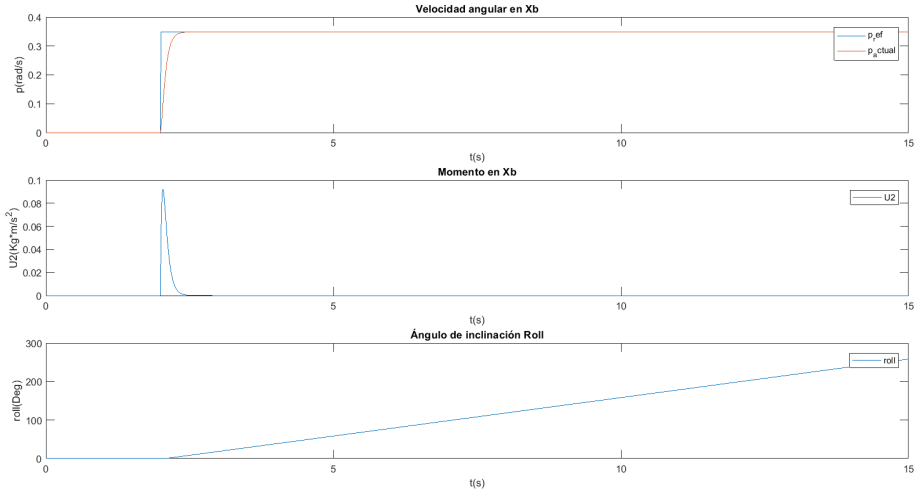


Figura 3.6 $p = 20\ (^\circ/s)$.

Como era de esperar, el ángulo en roll se mantiene en aumento, ya que p se mantiene constante. En caso de que se dejase de aplicar la velocidad, el ángulo debería de alcanzar un valor y mantenerse en él.

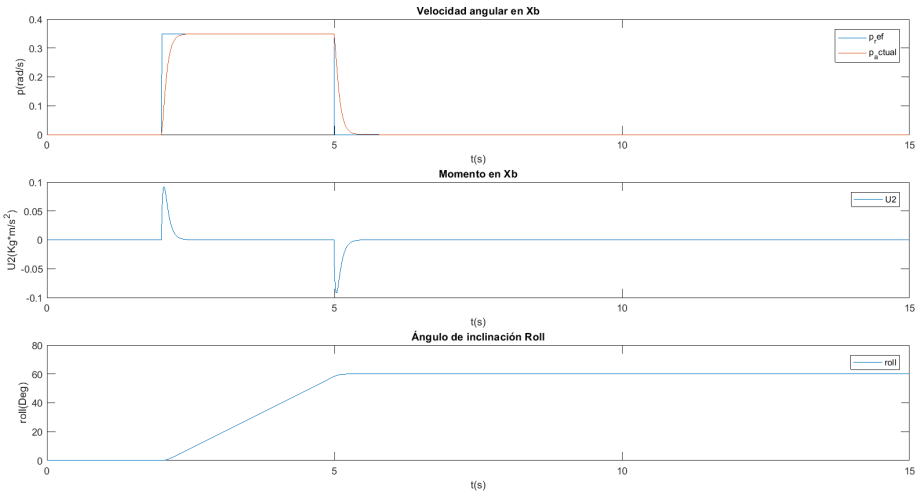
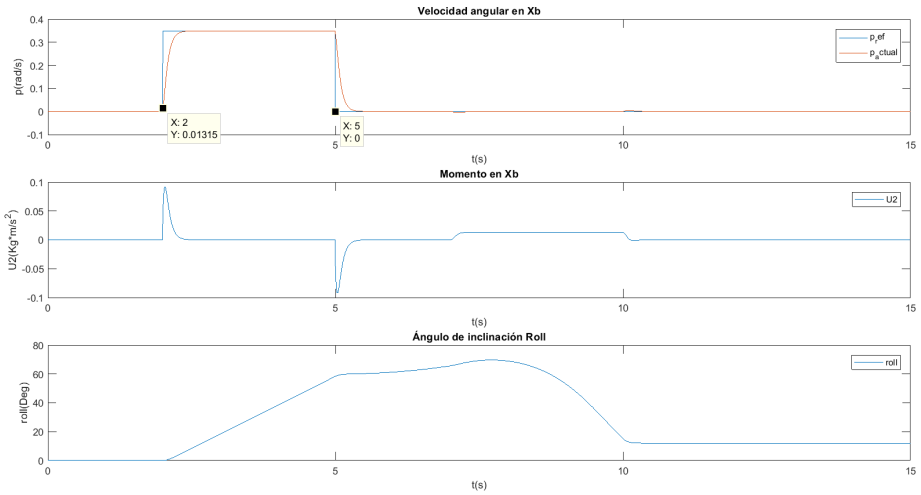
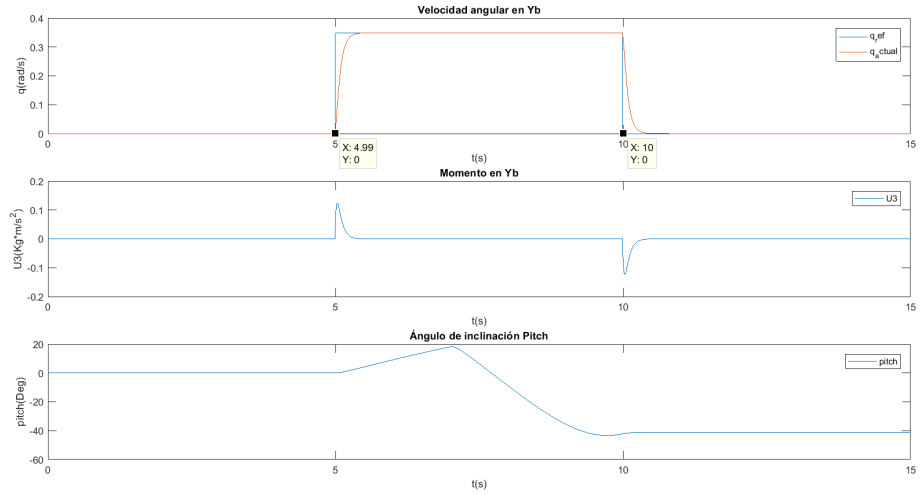


Figura 3.7 $p = 20\ (^\circ/s) -> p = 0\ (^\circ/s)$.

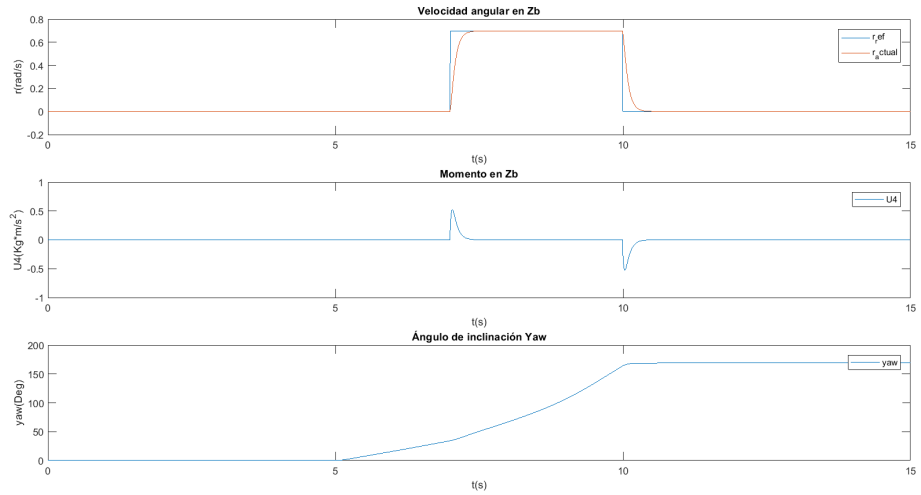
Para q y r la respuesta es similar a las que se muestran. Para finalizar este apartado, se realiza una simulación introduciendo una referencia en cada uno de las velocidades y observamos el comportamiento de la dinámica no lineal:



(a) Control de p.



(b) Control de q.



(c) Control de r.

Figura 3.8 Rate Control.

Podemos ver que la velocidad p en $\mathbf{X^B}$ no tiene efectos sobre el pitch (θ) ni sobre el yaw(ψ), como se indica en la ecuación 2.20. También vemos como en ausencia de p se siguen produciendo cambios en el valor de ϕ debido a los efectos de q y r.

3.3 Control de Actitud y Altura

Se han implementado a la par ambos controles, debido a que a la hora de realizar un vuelo manual, el quadrotor suele presentar pérdidas de altura, lo cuál dificulta el vuelo, haciendo complicado volar para personas que no están familiarizadas con este tipo de vehículos. Añadiendo este tipo de control, podemos centrarnos solamente en la actitud, sin preocuparnos del throttle.

3.3.1 Control de actitud

El objetivo de añadir este bucle de control es mantener el quadrotor estable durante el vuelo, es decir, que en ausencia de acciones del piloto, el vehículo se mantenga con ángulos de inclinación (ϕ,θ,ψ) en torno a 0. Este control es superior al rate control, por lo tanto, es un poco más lento. Las salidas de este control serán las velocidades (p,q,r) que se toman como referencias para el rate control.

Función de Transferencia

Para el diseño de este control, no vamos a utilizar las ecuaciones dinámicas, vamos a aprovechar el bucle que se encuentra a más bajo nivel. Tomaremos la función en bucle cerrado del *RateControl*.

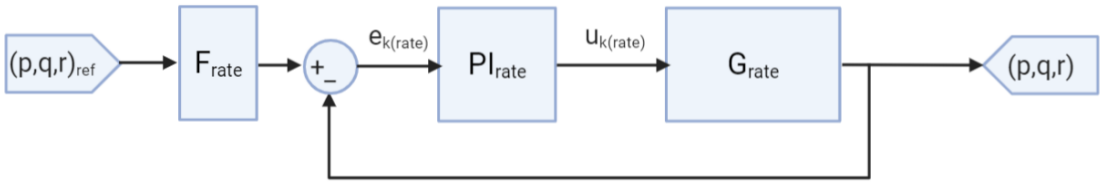


Figura 3.9 Bucle cerrado Rate Control.

$\mathbf{F_{rate}}$ es el *prefiltro* que se aplica a la referencia, con el fin de hacer más suave la entrada del sistema y eliminar sobreoscilaciones.
Ahora podemos eliminar el prefiltro del Rate Control, podremos diseñar un controlador que se encargue de realizar esta acción. Para el control de actitud, queremos controlar los ángulos de inclinación, pero la planta que tenemos actualmente nos da como salida las velocidades, por lo tanto tenemos que integrar la salida para conseguir el sistema que necesitamos:

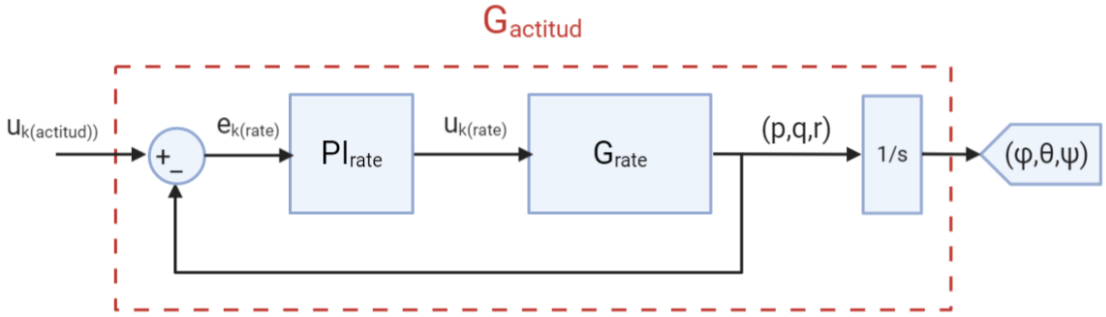


Figura 3.10 Función de transferencia para el Control de Actitud.

De forma que la función de transferencia quedaría de la siguiente forma:

$$G_{actitud}(s) = \left(\frac{PI_{rate} \cdot G_{rate}}{1 + PI_{rate} \cdot G_{rate}} \right) \cdot \frac{1}{s} \tag{3.24}$$

A través de un pequeño *script* de Matlab podemos calcular fácilmente la función de transferencia:

Código 3.7 Modelo.m.

```
%% Cálculo del modelo

Ixx = 0.0347563;% (Kg*m^2)
G_rate = tf([1],[Ixx 0]);

%% Controlador PI
PI_rate = tf([1.3093 13.093],[1 0]);

I = tf([1],[1 0]);

G_actitud = ((PI_rate*G_rate)/(1+PI_rate*G_rate))*I
```

Se ha puesto como ejemplo el código empleado para el cálculo del modelo para el control de ϕ, pero para θ y ψ el procedimiento es el mismo. Finalmente obtenemos los modelos para el diseño de los controladores para cada uno de los ángulos de la actitud.

$$G_{\phi}(s) = \frac{\phi(s)}{p(s)} = \frac{0.04551s + 0.4551}{0.001208s^3 + 0.04551s^2 + 0.4551s} \tag{3.25}$$

$$G_{\theta}(s) = \frac{\theta(s)}{q(s)} = \frac{0.08425s + 0.8425}{0.002106s^3 + 0.08425s^2 + 0.8425s} \tag{3.26}$$

$$G_{\psi}(s) = \frac{\psi(s)}{r(s)} = \frac{0.3818s + 3.818}{0.009545s^3 + 0.3818s^2 + 3.818s} \tag{3.27}$$

Lugar de las Raíces

Utilizamos el comando *rltool* para dibujar el lugar de las raíces de las funciones calculadas en la sección anterior. El procedimiento a seguir es el mismo que para el *RateControl*, añadiremos un Integrador ($s = 0$)

para que el error en régimen permanente sea nulo. Al añadir el integrado el sistema se vuelve inestable, por lo tanto debemos añadir un cero en el eje real negativo. Este control no es tan rápido como el *RateControl* pero debe de ser lo suficientemente rápido como para que el quadrotor pueda mantenerse estable en el aire, por lo que debe de ser bastante rápido también. Se considerará un tiempo de subida válido en torno a 0.6 segundos.

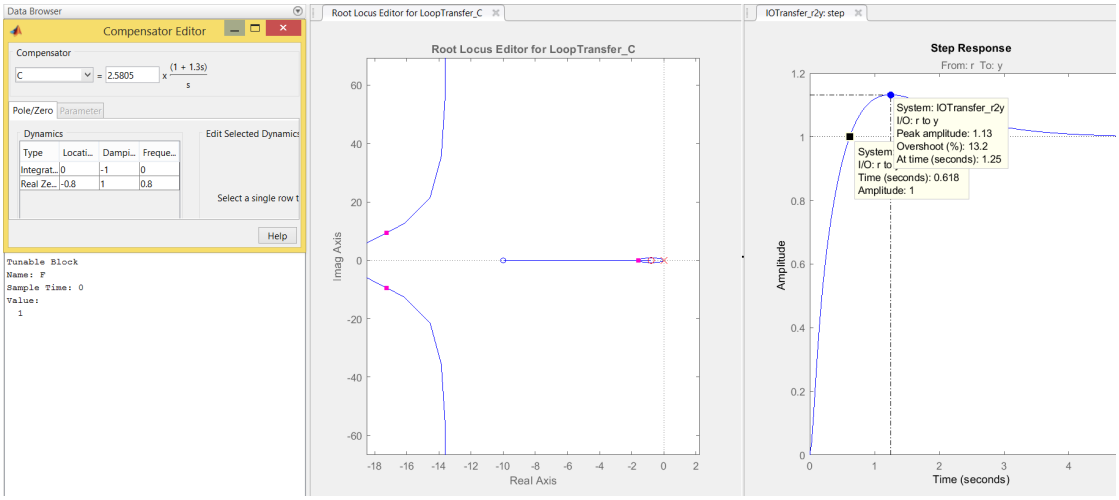


Figura 3.11 PI para el control del roll.

Como podemos ver, con el PI cumple el tiempo de subida que queremos, pero vemos que tiene una sobreoscilación del 13 %, lo cual no es deseable, podemos intentar ajustarlo. Vamos a añadir un cero más para obtener un PID y ver como se comporta el sistema.

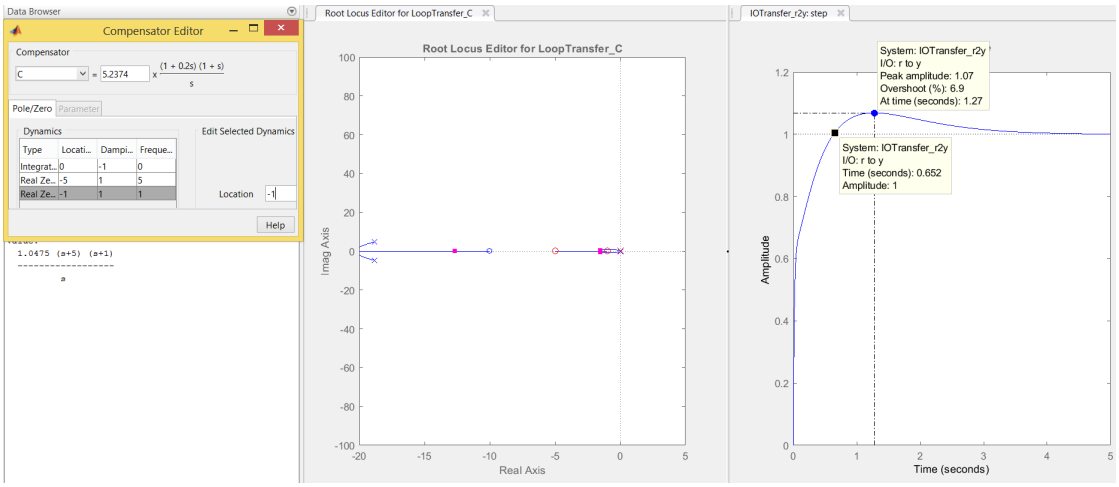


Figura 3.12 PID para el control del roll.

Tras varios intentos por ajustar los parámetros del controlador para intentar eliminar la sobreoscilación y mantener el tiempo de subida, no se ha conseguido un resultado satisfactorio, por lo tanto, se opta por

la estrategia seguida en el *Rate Control*. Vamos a añadir un prefiltro para la referencia y utilizamos una estructurad de tipo *PI*.

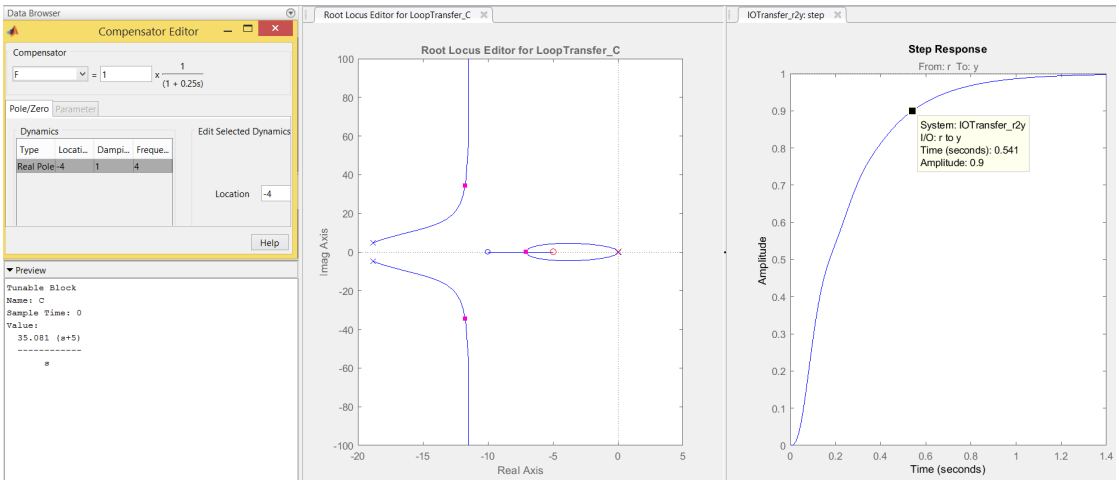


Figura 3.13 PI + F para el control del roll.

Como se puede ver hemos eliminado la sobreoscilación con la acción del prefiltro, además ajustando la posición del cero, conseguimos el tiempo de respuesta deseado. Obtenemos una respuesta suave, evitando así oscilaciones indeseadas en torno al valor objetivo que pudieran ocasionar problemas durante el vuelo. Tanto para el ϕ como para el θ el diseño del control es similar, y los tiempos de respuesta son similares, pero el control del ψ no puede ser tan rápido como el control de estos dos ángulos. Atendiendo a la configuración del quadrotor, el par resistente en Z^B a la hora de realizar variaciones de ψ es mayor que en los ejes horizontales.

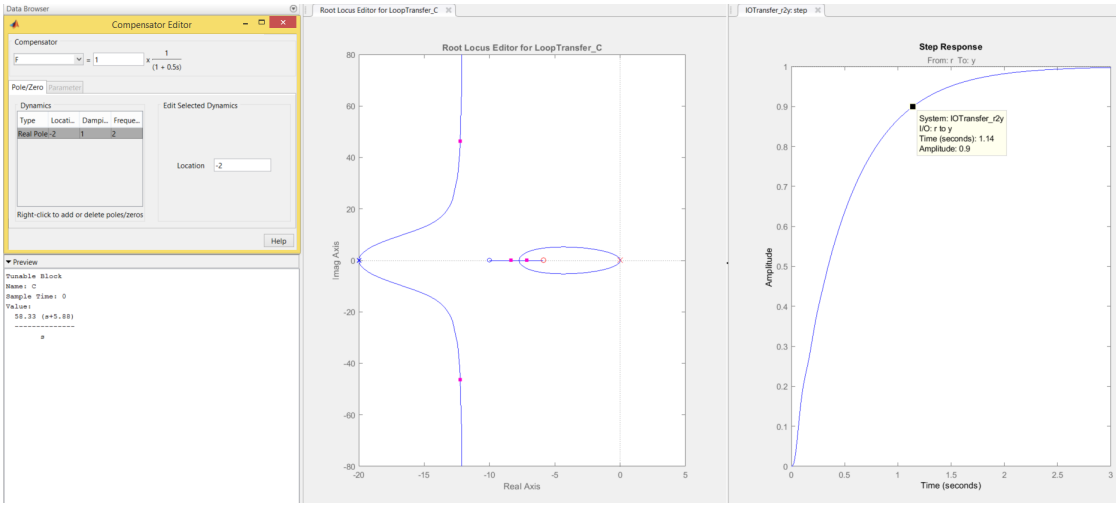


Figura 3.14 PI + F para el control del yaw.

Se ha optado por considerar un tiempo de subida válido en torno a 1.1*segundos*, un tiempo aproximadamente igual al doble del tiempo de respuesta para los otros ángulos. Finalmente tenemos tres controladores PI con tres prefiltros para la referencia:

$$PI_\phi(s) = 175.4 \cdot \frac{0.2s+1}{s} \qquad PI_\theta(s) = 172.18 \cdot \frac{0.2s+1}{s} \qquad PI_\psi(s) = 342.98 \cdot \frac{0.17s+1}{s}$$

$$F_\phi(s) = \frac{1}{0.25s+1} \qquad F_\theta(s) = \frac{1}{0.25s+1} \qquad F_\psi(s) = \frac{1}{0.5s+1}$$

Tabla 3.2 Parámetros de los controladores PI del Control de Actitud.

Descripción	Tipo	T_i	T_d	K_p	K_i	K_d
Control de ϕ	PI	0.2	0	35.0800	175.4000	0
Control de θ	PI	0.2	0	34.4360	172.1800	0
Control de ψ	PI	0.17	0	58.3066	342.9800	0

Experimentos en Matlab

A las funciones vistas en la sección 3.2, se añade una nueva función *.m* donde se van a implementar los controladores diseñados hasta ahora en esta sección. El script sigue la estructura vista hasta ahora, tiene como entradas las referencias en *rad* y como salidas las velocidades angulares.

Código 3.8 attitude_control.m.

```
function out = attitude_control(in)

roll_des = in(1);
pitch_des = in(2);
yaw_des = in(3);

global erle;

%% Control del roll
erle.roll_des_filt = (erle.roll_TI_F/(erle.roll_TI_F+erle.Tm))*erle.
    roll_des_filt_1 + (erle.Tm/(erle.roll_TI_F+erle.Tm))*roll_des;
% Cálculo del error
roll_ek = erle.roll_des_filt - erle.roll;
% Incremento de la integral del error
erle.roll_Int_ek = erle.roll_Int_ek + erle.Tm*roll_ek;
% Controlador PI
p_des = erle.roll_KP*(roll_ek + (1/erle.roll_TI)*erle.roll_Int_ek + erle.
    roll_TD*((roll_ek-erle.roll_ek_1)/erle.Tm));
% Saturación
p_des = min(erle.p_max,max(-erle.p_max,p_des));

erle.roll_des_filt_1 = erle.roll_des_filt;
erle.roll_ek_1 = roll_ek;
```

```
%% Control del pitch
erle.pitch_des_filt = (erle.pitch_TI_F/(erle.pitch_TI_F+erle.Tm))*erle.
    pitch_des_filt_1 + (erle.Tm/(erle.pitch_TI_F+erle.Tm))*pitch_des;
% Cálculo del error
pitch_ek = erle.pitch_des_filt - erle.pitch;
% Incremento de la integral del error
erle.pitch_Int_ek = erle.pitch_Int_ek + erle.Tm*pitch_ek;
% Controlador PI
q_des = erle.pitch_KP*(pitch_ek + (1/erle.pitch_TI)*erle.pitch_Int_ek + erle.
    pitch_TD*((pitch_ek-erle.pitch_ek_1)/erle.Tm));
% Saturación
q_des = min(erle.q_max,max(-erle.q_max,q_des));

erle.pitch_des_filt_1 = erle.pitch_des_filt;
erle.pitch_ek_1 = pitch_ek;

%% Control del yaw
erle.yaw_des_filt = (erle.yaw_TI_F/(erle.yaw_TI_F+erle.Tm))*erle.
    yaw_des_filt_1 + (erle.Tm/(erle.yaw_TI_F+erle.Tm))*yaw_des;
% Cálculo del error
yaw_ek = erle.yaw_des_filt - erle.yaw;
% Incremento de la integral del error
erle.yaw_Int_ek = erle.yaw_Int_ek + erle.Tm*yaw_ek;
% Controlador PI
r_des = erle.yaw_KP*(yaw_ek + (1/erle.yaw_TI)*erle.yaw_Int_ek + erle.yaw_TD
    *((yaw_ek-erle.yaw_ek_1)/erle.Tm));
% Saturación
r_des = min(erle.r_max,max(-erle.r_max,r_des));

erle.yaw_des_filt_1 = erle.yaw_des_filt;
erle.yaw_ek_1 = yaw_ek;

out = [p_des,q_des,r_des];

end
```

Como ya se ha mencionado varias veces, ahora se han eliminado los prefiltros que se diseñaron para el rate control, de forma que hay que modificar el archivo *rate_control.m*.

Código 3.9 rate_control.m sin prefiltros.

```
function out = rate_control (in)

p_des = in(1);
q_des = in(2);
r_des = in(3);

global erle;
```

```
%% Control de variación del roll
p_ek = p_des - erle.p;
% Incremento de la integral del error
erle.p_Int_ek = erle.p_Int_ek + erle.Tm*p_ek;
% Controlador PI
U2 = erle.p_KP*(p_ek + (1/erle.p_TI)*erle.p_Int_ek + erle.p_TD*((p_ek-erle.
    p_ek_1)/erle.Tm));
% Saturación
U2 = min(erle.U2_max,max(erle.U2_min,U2));
erle.p_ek_1 = p_ek;

%% Control de variación del roll
q_ek = q_des - erle.q;
% Incremento de la integral del error
erle.q_Int_ek = erle.q_Int_ek + erle.Tm*q_ek;
% Controlador PI
U3 = erle.q_KP*(q_ek + (1/erle.q_TI)*erle.q_Int_ek + erle.q_TD*((q_ek-erle.
    q_ek_1)/erle.Tm));
% Saturación
U3 = min(erle.U3_max,max(erle.U3_min,U3));
erle.q_ek_1 =q_ek;

%% Control de variación del roll
r_ek = r_des - erle.r;
% Incremento de la integral del error
erle.r_Int_ek = erle.r_Int_ek + erle.Tm*r_ek;
% Controlador PI
U4 = erle.r_KP*(r_ek + (1/erle.r_TI)*erle.r_Int_ek + erle.r_TD*((r_ek-erle.
    r_ek_1)/erle.Tm));
% Saturación
U4 = min(erle.U4_max,max(erle.U4_min,U4));
erle.r_ek_1 = r_ek;

out = [U2,U3,U4];

end
```

El archivo encargado de la simulación es prácticamente igual, solo que en lugar de establecer las velocidades como referencias, se establecen los ángulos (ϕ,θ,ψ). En el diagrama de bloques de simulink también se ha añadido un nuevo bloque con la nueva función creada.

Código 3.10 Simulacion.m + Simulink control de actitud.

```
clear all;
close all;
clc;

global erle;
```

```
erle_variables;

erle.T_simulacion = 15;% segundos
erle.T_escalon_roll = 2;%(segundos);
erle.T_escalon_pitch = 5;%(segundos);
erle.T_escalon_yaw = 7;%(segundos)

erle.roll_des = 15;
erle.pitch_des = 0;
erle.yaw_des = 0;

SimOut = sim('diagrama_de_bloques_actitud');
```

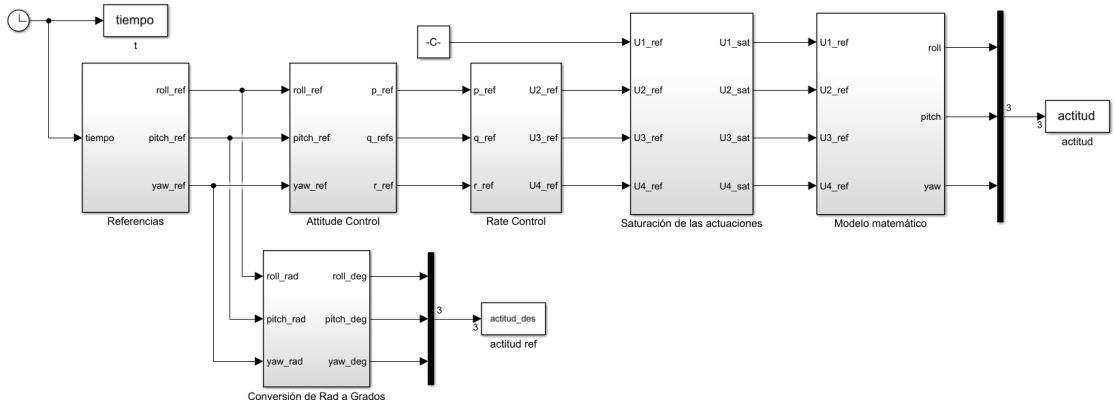


Figura 3.15 Diagrama de bloques del control de actitud.

Se ha cambiado el modo de mandar las referencias a la simulación (*referencias.m*), se ha creado un script por comodidad. Solamente tendremos que definir los tiempos de los escalones y los valores, y podemos hacerlo directamente desde el archivo principal *simulacion.m*.

Código 3.11 referencias.m.

```
function out = referencias(in)

time = in(1);
global erle;

if(time < erle.T_escalon_roll)
roll_des = 0;
end
if(time >= erle.T_escalon_roll)

roll_des = erle.roll_des*(erle.Deg_Rad);
end
% Señal de entrada q_des
```

```
if(time < erle.T_escalon_pitch)
pitch_des = 0;
end
if(time >= erle.T_escalon_pitch)
roll_des = 0;
pitch_des = erle.pitch_des*(erle.Deg_Rad);
end
% Señal de entrada r_des
if(time < erle.T_escalon_yaw)
yaw_des = 0;
end
if(time >= erle.T_escalon_yaw)
roll_des = 0;
pitch_des = 0;
yaw_des = erle.yaw_des*(erle.Deg_Rad);
end
out = [roll_des,pitch_des,yaw_des];
end
```

Como primer paso, vamos a verificar que el sistema sigue la referencia de forma correcta, se introducen las referencias de forma individual, es decir, se da un valor de $\phi, \theta o \psi$ y los otros dos se mantienen a cero.

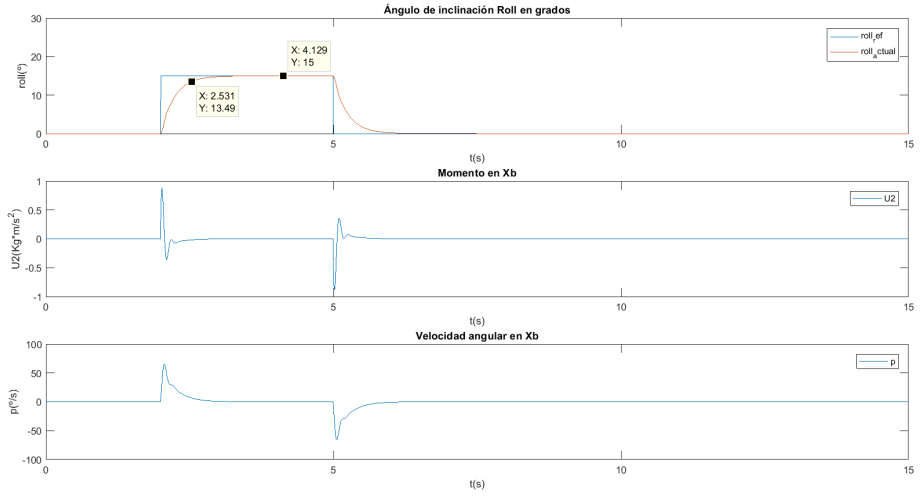


Figura 3.16 $\phi = 15, \theta = \psi = 0.$

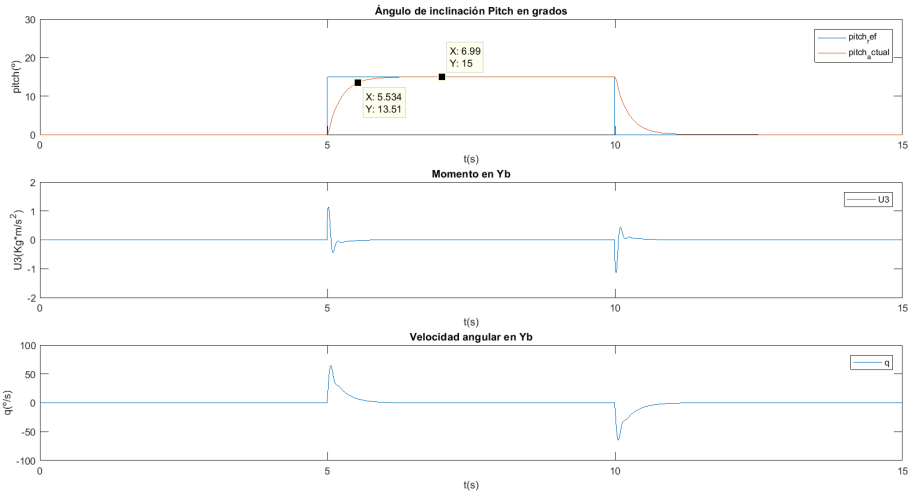


Figura 3.17 $\theta = 15, \phi = \psi = 0$.

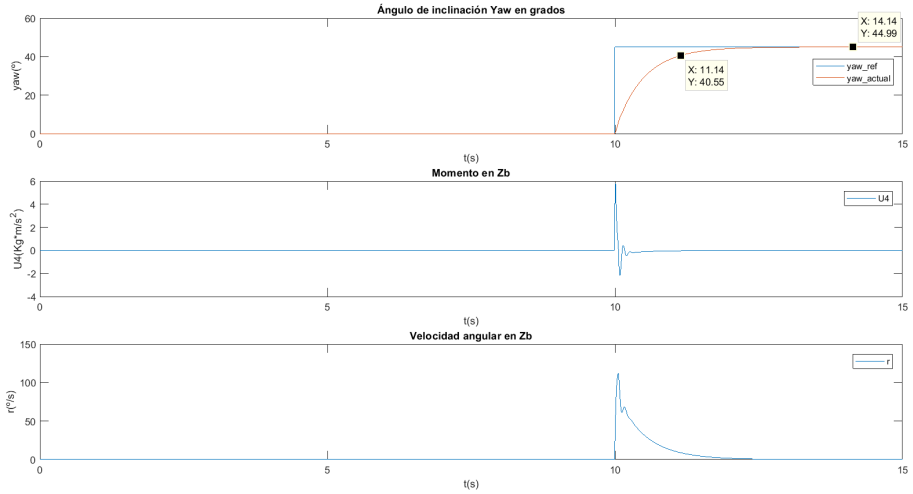
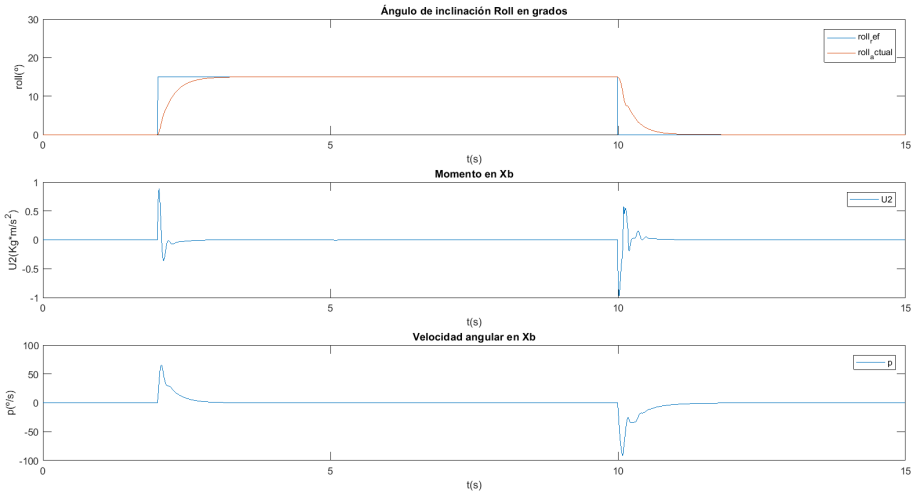


Figura 3.18 $\psi = 45, \phi = \theta = 0$.

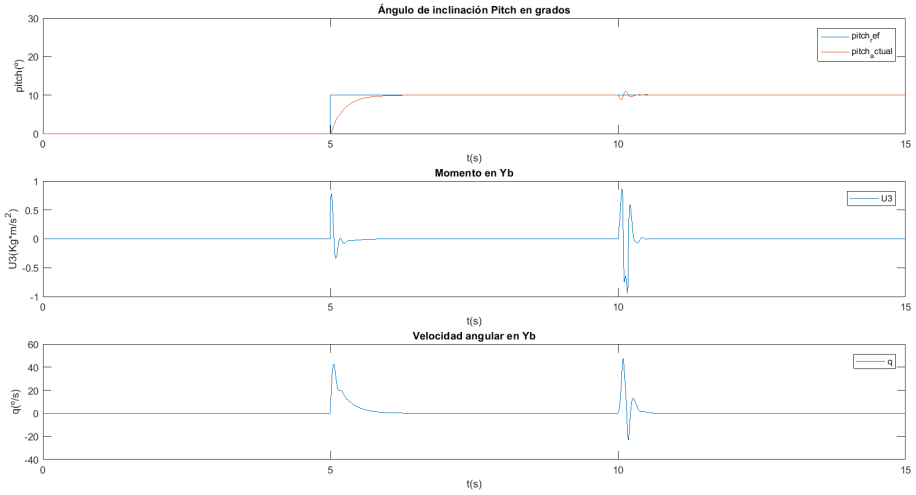
Por último realizamos una prueba donde variamos los tres valores y vemos como se comporta el sistema, y vemos el efecto que producen los ángulos entre si.

Tabla 3.3 Referencias.

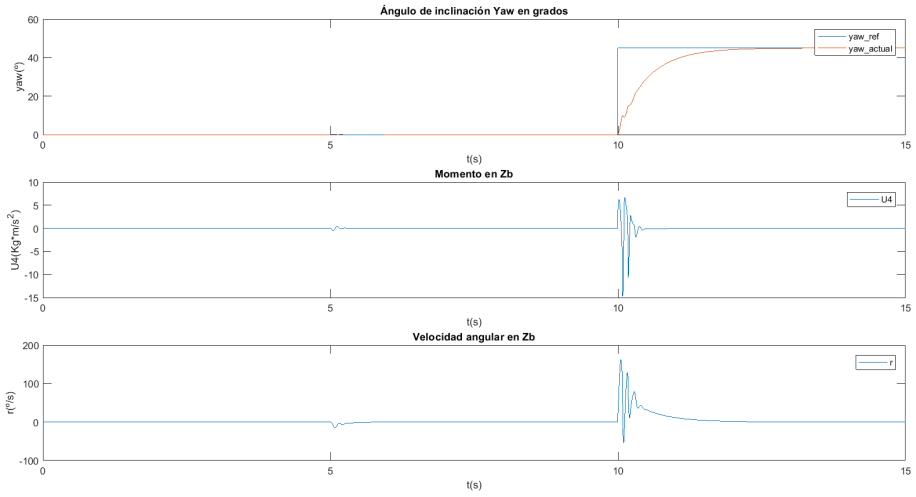
Descripción	Variable	Valor	Unidades	Tiempo
Ángulo ϕ	*erle.roll_des*	15	grados(º)	2(s)
Ángulo θ	*erle.pitch_des*	10	grados(º)	5(s)
Ángulo ψ	*erle.yaw_des*	40	grados(º)	10(s)
Ángulo ϕ	*erle.roll_des*	0	grados(º)	10(s)



(a) Control de roll.



(b) Control de pitch.



(c) Control de yaw.

Figura 3.19 Control de Actitud.

Podemos observar como la variación del θ provoca una pequeña perturbación en U_4 y en r, pero el control reacciona correctamente. Vemos que cuando se produce una variación de ψ las componentes U_3 y U_2 se ven afectadas, debido a la variación que provoca en la velocidad de los motores. Y por último también podemos apreciar pequeñas perturbaciones en ϕ y θ cuando se produce un cambio en ψ, pero nada que pueda suponer un problema a la hora de estabilizar el quadrotor.

3.3.2 Control de Altura

Como ya se ha comentado al inicio de la sección, será el encargado de mantener la altura de nuestro vehículo durante el vuelo, haciendo más cómodo y sencillo el control del quadrotor si quisiéramos realizar un vuelo manual. Cabe destacar que este lazo de control es más lento, ya que hay que ser coherentes con el mundo real, no podemos pretender que el quadrotor recorra 3 metros en 0.1 segundos, ya que sería físicamente imposible. Por lo tanto, el tiempo de subida de este tipo de control deberá ser de varios segundos, pudiendo ajustarse en un futuro.

Función de transferencia

Tomamos la ecuaciones linealizadas en el capítulo 2 para el cálculo de la aceleración en $\ddot{Z}^G$ (2.47).

$$f_{\ddot{Z}^G} = \ddot{Z}^G \cdot m - U_1 + g \cdot m \tag{3.28}$$

Se le aplicará el desarrollo de Taylor a dichas ecuaciones en torno al punto de equilibrio. En el punto de equilibrio, es decir, cuando el vehículo se encuentra estable en el aire la fuerza de empuje es aproximadamente igual a la fuerza que ejerce la gravedad ($U_{1eq} \approx g \cdot m$)

$$\left.\frac{\partial f_{\ddot{Z}^G}}{\partial \ddot{Z}^G}\right|_{\ddot{Z}^G{}_{eq}} \ddot{Z}^G(t) - \left.\frac{\partial f_{\ddot{Z}^G}}{\partial U_1}\right|_{U_1 eq} U_1(t) = 0 \tag{3.29}$$

$$\ddot{Z}^G(t) \cdot m - U_1(t) = 0 \tag{3.30}$$

Calculamos la transformada de Laplace ($\mathscr{L}$) de la ecuación en diferencias obtenida, obteniendo así una ecuación que nos relaciona directamente el la fuerza de empuje con la altura en ejes G, siendo la fuerza (U_1) la variable de entrada y la altura Z^G la salida del sistema:

$$Z^G(s) \cdot s^2 \cdot m - U_1(s) = 0 \tag{3.31}$$

$$G_{Z^G}(s) = \frac{Z^G(s)}{F_T^B(s)} = \frac{1}{s^2 \cdot m} \tag{3.32}$$

Lugar de las Raíces

Seguiremos los pasos para los controladores anteriores. Dibujamos el lugar de las raíces para ver el comportamiento del sistema en bucle cerrado, y le añadimos un integrador para eliminar el error en régimen permanente.

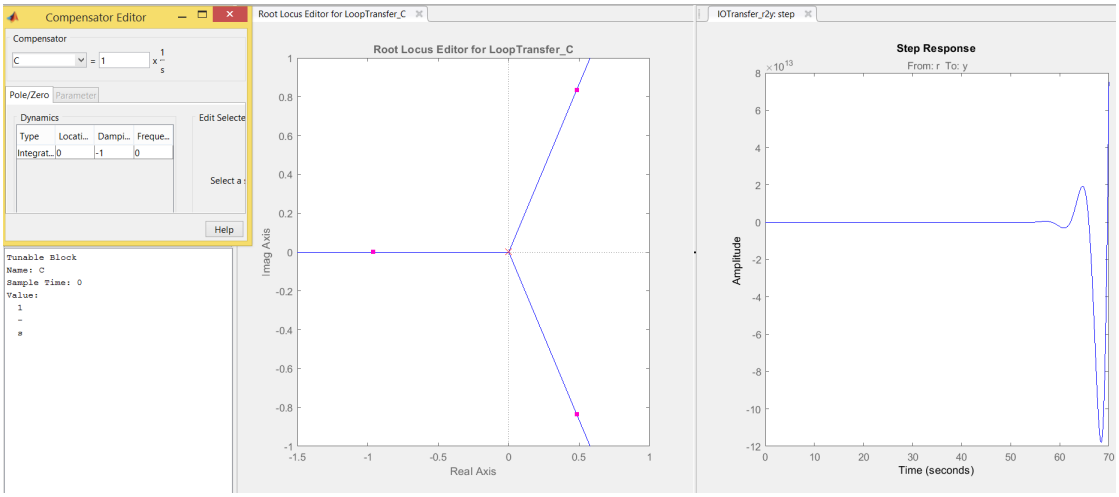


Figura 3.20 I para el control de altura.

Se puede observar que el sistema es inestable, así que vamos a añadirle un cero para ver como se comportaría con un PI.

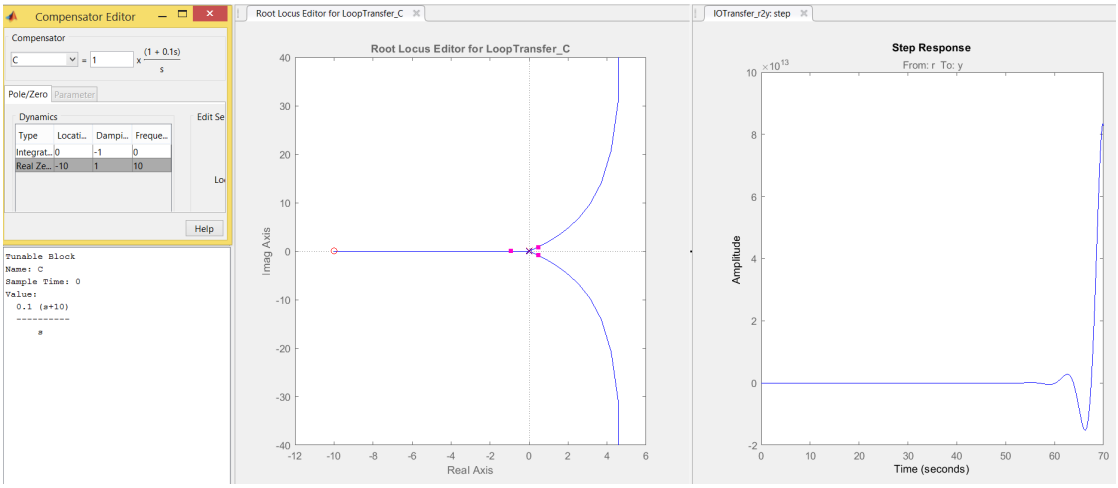


Figura 3.21 PI para el control de altura.

El sistema sigue siendo inestable, debido a que las raíces en bucle cerrado se encuentran en el plano positivo, por lo tanto, en este caso necesitamos un controlador de tipo PID, debemos añadir un nuevo cero. Se añade también un prefiltro para la referencia, al igual que en diseños anteriores para eliminar la sobreoscilación. En este control esto es de gran importancia, ya que una sobreoscilación en altura, podría provocar que el quadrotor chocase con el techo si se estuviese volando en un espacio cerrado.

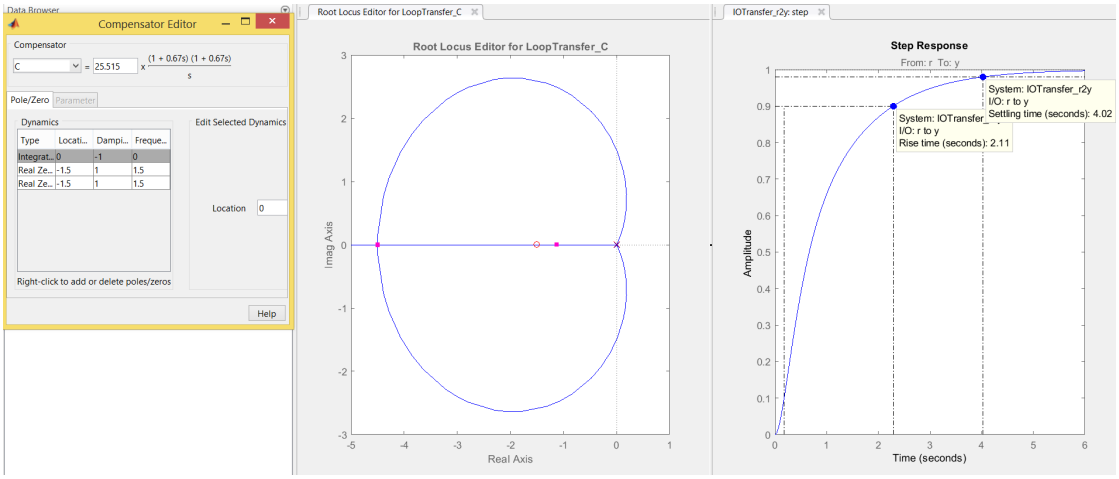


Figura 3.22 PID para el control de altura.

Se ha establecido un tiempo de subida de unos 2.1 *segundos*, dicho tiempo podrá ser ajustado en un futuro una vez se haya conseguido volar correctamente el quadrotor, haciéndolo más rápido si fuera posible sin perder estabilidad.

$$PID_{Z^G}(s) = 25.515 \cdot \frac{(1+0.67s)(1+0.67s)}{s}$$

$$F_{Z^G}(s) = \frac{1}{1+s}$$

Tabla 3.4 Parámetros del controlador PID del Control de Altura.

Descripción	Tipo	T_i	T_d	K_p	K_i	K_d
Control de Z^G	PID	1.34	0.335	34.1901	25.5150	11.4537

Experimentos Matlab

Para este experimento podemos prescindir de la función *rate_control.m* y de la función *attitude_control.m*, ya que vamos a centrarnos solamente en el desplazamiento vertical y en la altura del quadrotor, manteniendo los ángulos de actitud en torno a cero. Más adelante, si se tienen en cuenta los demás bucles de control, en este experimento se han omitido para centrarse únicamente en la validación del control en altura.
El PID diseñado se implementa en la función *altitud_control.m*, que tiene como entrada un valor de referencia en $\mathbf{Z^G}$ que se ha considerado positivo hacia arriba, y como salida tiene la fuerza de empuje generada por los motores.

Código 3.12 altitud_control.m.

```
function out = altitud_control(in)

Z_des = in(1);
```

```
global erle;

%% Control del Z
erle.Z_des_filt = (erle.Z_TI_F/(erle.Z_TI_F+erle.Tm))*erle.Z_des_filt_1 + (
    erle.Tm/(erle.Z_TI_F+erle.Tm))*Z_des;
% Cálculo del error
Z_ek = erle.Z_des_filt - erle.Z;
% Incremento de la integral del error
erle.Z_Int_ek = erle.Z_Int_ek + erle.Tm*Z_ek;
% Controlador PI
U1 = (erle.Z_KP*(Z_ek + (1/erle.Z_TI)*erle.Z_Int_ek + erle.Z_TD*((Z_ek-erle.
    Z_ek_1)/erle.Tm))) + erle.m*erle.g;
% Saturación
U1 = min(erle.U1_max,max(erle.U1_min,U1));

erle.Z_des_filt_1 = erle.Z_des_filt;
erle.Z_ek_1 = Z_ek;

out = [U1];
end
```

Se ha modificado la función *ecuaciones_dinamicas.m*, eliminando la dinámica rotacional y quedándonos solamente con la ecuación que calcula la aceleración lineal en Z^G.

Código 3.13 ecuaciones_dinamicas.m para el control de altura.

```
function out = ecuaciones_dinamicas (in)

U1 = in(1);
U2 = in(2);
U3 = in(3);
U4 = in(4);

global erle;

erle.Z_dd = (+(cos(erle.roll)*cos(erle.pitch))*U1 - 0.3*erle.Z_d)/erle.m -
    erle.g;

% % Velocidad y posición en Z
erle.Z_d = erle.Z_dd * erle.Tm + erle.Z_d;
erle.Z = erle.Z_d * erle.Tm + erle.Z;

erle.U1_plot(erle.indice) = U1;
erle.Z_d_plot(erle.indice) = erle.Z_d;
erle.Z_dd_plot(erle.indice) = erle.Z_dd;

erle.indice = erle.indice + 1;
```

```
out = [erle.Z];

end
```

Para este experimento, una referencia de tipo escalón es idónea, ya que queremos mantener el vehículo a una cierta altura. Se define un tiempo y un valor para el escalón, se ha elegido un valor no muy grande debido a la ausencia de un generador de trayectorias, que podrá ser incorporado más adelante si fuese necesario.

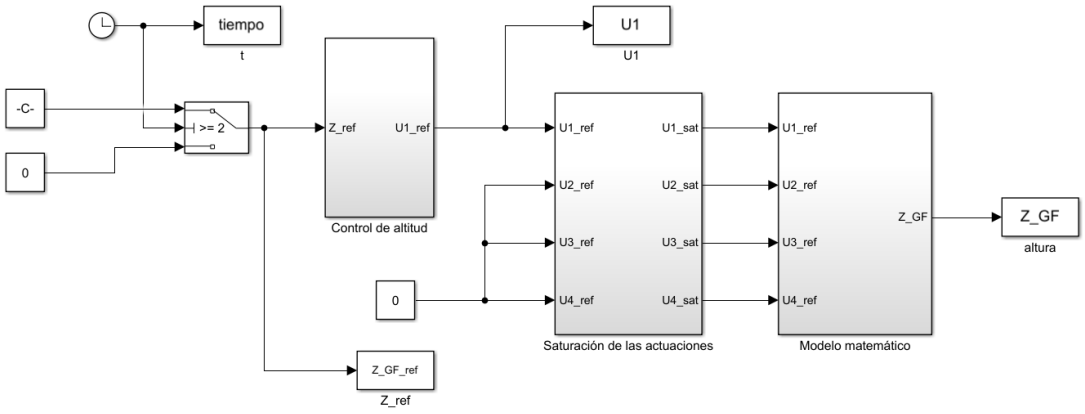


Figura 3.23 Control de altura.

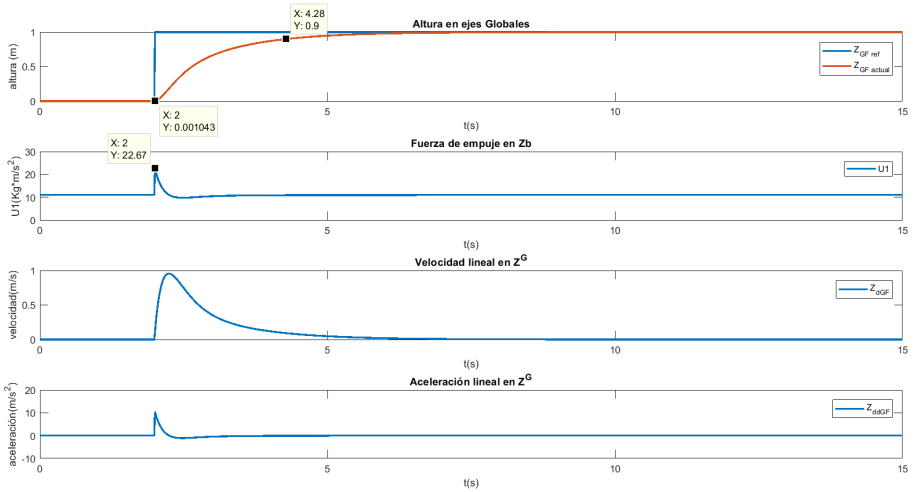


Figura 3.24 Control de altura.

Aquí podemos verificar alguna de las hipótesis que se tomaron a la hora de linealizar las ecuaciones, cuando el quadrotor se encuentra estable en el aire, es decir, alcanza la referencia, la fuerza de empuje es igual al peso del quadrotor. Como es lógico, a medida que se acerca a la referencia la velocidad va

disminuyendo. Una vez se alcanza la referencia la velocidad y la aceleración lineal son nulas, ya que se mantiene a una altura constante en Z^G. Por último no se observan saturaciones en la señal de control, por lo tanto podemos dar el control por válido.

3.4 Control de posición

Finalmente se va a añadir un bucle en el nivel más alto, que se encargará de calcular la actitud de referencia a partir de una posición deseada.

3.4.1 Función de transferencia

A medida que vamos añadiendo bucles de control, el sistema se vuelve más complejo.

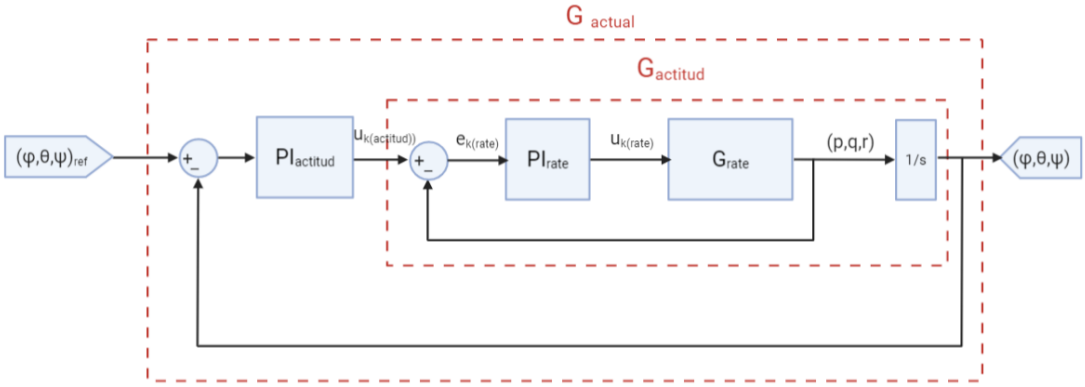


Figura 3.25 Bucles de control de más bajo nivel.

Actualmente disponemos de una planta que nos proporciona como salidas los ángulos de actitud, pero para este control se necesita obtener como salida la posición, ya que la necesitamos para la realimentación, por lo tanto, debemos obtener una forma de relacionar la actitud con la posición en ejes horizontales. Si calculamos el sistema resultante de la Figura 3.25 nos saldrá un sistema de orden muy elevado, por lo tanto vamos a obtener el modelo necesario para el control a partir de las ecuaciones matemáticas.

Partimos de las ecuaciones linealizadas (2.47) para el cálculo de la aceleración lineal en $\ddot{X}^G$ e $\ddot{Y}^G$. En estas ecuaciones tenemos acoplados los ángulos de actitud ϕ y θ con la fuerza de empuje F_T^B. Para simplificar el modelo, vamos a aplicar la hipótesis que veíamos en el diseño del control de altura, en el aire, cuando el vehículo está estable en el aire:

$$F_T^B = m \cdot g \tag{3.33}$$

De forma que ahora las ecuaciones nos quedan de la siguiente forma:

$$\ddot{X}^G = \theta \cdot \frac{1}{m} \cdot mg \tag{3.34}$$

$$\ddot{Y}^G = -\phi \cdot \frac{1}{m} \cdot mg \tag{3.35}$$

Siguiendo el procedimiento descrito en secciones anteriores y aplicando $\mathscr{L}$aplace, obtenemos las funciones de transferencia que nos relacionan la posición en G con la actitud.

$$G_{X^G}(s) = \frac{X^G}{\theta(s)} = \frac{g \cdot \theta}{s^2} \tag{3.36}$$

$$G_{Y^G}(s) = \frac{Y^G}{-\phi(s)} = \frac{g \cdot \phi}{s^2} \tag{3.37}$$

Para poder realizar esta aproximación, y despreciar la dinámica de los bucles de control de bajo nivel, la velocidad de respuesta del control de posición, deberá de ser, como mínimo, del orden de 5 o 6 veces más lenta que la respuesta del control de actitud. Por lo tanto, si la respuesta del control de actitud diseñado para el control del ϕ y θ era de unos 0.55 *segundos*, el control de posición que se diseñará en esta sección, deberá tener una respuesta de unos 6 *segundos* aproximadamente.

3.4.2 Lugar de las Raíces

El lugar de las raíces de este sistema será bastante más complejo que los que se han visto hasta ahora. Será necesaria una estructura de tipo PID para poder obtener una respuesta satisfactoria. Se sigue la misma estructura que en controles anteriores, añadiendo un prefiltro a la referencia para eliminar las sobreoscilaciones y hacer mas suave la respuesta del sistema. Al igual que en el control de altura, es muy importante que no haya sobreoscilaciones en los desplazamientos, ya que podrían provocar que el quadrotor chocase con algún obstáculo.

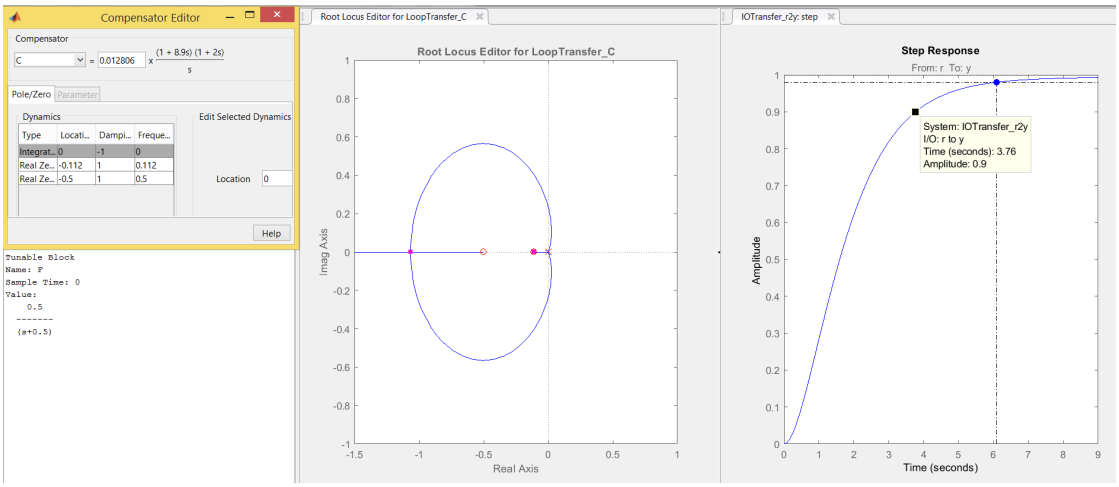


Figura 3.26 PID para el control de altura.

El PID y el F obtenidos, se aplican tanto para el control de X^G como para Y^G. Como era de esperar, este lazo de control es mucho más lento que los de nivel inferior. Hay que ser coherentes físicamente, ya que el quadrotor tiene limitaciones, no podemos dar como referencias escalones de 10 metros, y pretender que el control funcione. Lo más optimo sería añadir un generador de trayectorias para gestionar la posición de referencia.

$$PID_{posicion}(s) = 0.012806 \cdot \frac{(1+8.9s)(1+2s)}{s} \tag{3.38}$$

$$F_{posicion}(s) = \frac{1}{1+2s} \tag{3.39}$$

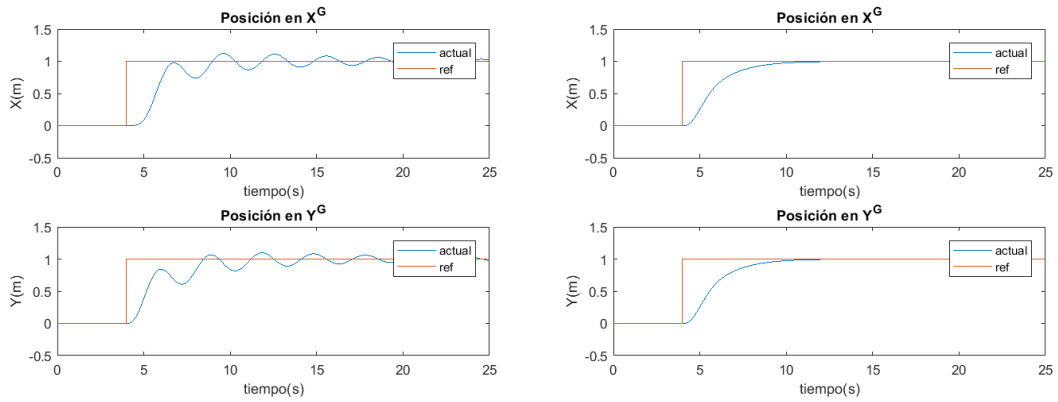
3.4.3 Experimentos

Este control diseñado, tal y como está, solamente funcionará si los ejes del cuerpo se encuentran alineados con los ejes globales, ya que según vemos en las ecuaciones 3.34 y 3.35, se ha despreciado el efecto del ψ, y no siempre un ángulo de ϕ tiene por que suponer un desplazamiento positivo en X^G, ni un ángulo negativo de θ, siempre implica un desplazamiento positivo en Y^G. Se ha añadido una función encargada de realizar una corrección de ψ, es decir, los ángulos ϕ y θ obtenidos del control de posición, serán rotados con respecto a Z^G, un ángulo, el ángulo ψ actual del quadrotor.

Código 3.14 yaw_correction.m para el control de posición.

```
function out = yaw_correccion(in)

roll_des = in(1);
pitch_des = in(2);

global erle;

pitch_des_yaw = cos(erle.yaw)*pitch_des - sin(erle.yaw)*roll_des;
roll_des_yaw = sin(erle.yaw) * pitch_des + cos(erle.yaw) * roll_des;

erle.roll_des_plot(erle.indice) = roll_des*erle.Rad_Deg;
erle.pitch_des_plot(erle.indice) = pitch_des*erle.Rad_Deg;

out = [roll_des_yaw,pitch_des_yaw];
end
```

Vamos a ver la importancia de esta función, colocamos el quadrotor haciendo coincidir los ejes cuerpo con los ejes globales, y a continuación realizamos un giro de $\psi = 45$. Ahora los ejes de los dos sistemas no coinciden, y le damos una posición de referencia (X=1, Y=1).



(a) Sin yaw_correction.m. **(b)** Con yaw_correction.m.

Figura 3.27 Corrección del ψ.

Se puede ver como en ausencia de la corrección del ψ, el quadrotor se queda oscilando en torno al punto de referencia, ya que se está desplazando con respecto a unos ejes intermedios que no se corresponden con los del sistema $\{\mathbf{G}\}$. Cuando no se desprecia el ψ del quadrotor para el desplazamiento, vemos como alcanza correctamente la posición.

Como este es el control a más alto nivel y precisa del control de niveles inferiores diseñados hasta ahora, se estudiará con más detalle su implementación y su funcionamiento en el siguiente capítulo, donde se incluirá una simulación gráfica, que nos permitirá apreciar mejor la acción de este control.

4 Simulación del Modo de Vuelo Autónomo en Matlab

En este capítulo se unifica todo lo visto hasta ahora en el presente documento, con el fin de obtener una simulación clara y precisa del modo de vuelo desarrollado. A parte de la unificación de los diferentes controles desarrollados, se implementará una simulación gráfica, con el fin de facilitar la comprensión del comportamiento del vehículo durante el vuelo. Se realizarán varios vuelos con el fin de verificar el funcionamiento y la fiabilidad de lo que se ha desarrollado a nivel teórico, con el fin de que se pueda realizar una implementación en un entorno más real.

4.1 Simulador Gráfico en Matlab

Hasta ahora se ha analizado el comportamiento del quadrotor en 2D mediante gráficas, lo cuál es necesario a nivel matemático para verificar el funcionamiento y que lo que se está desarrollando tiene lógica y coherencia. Pero una vez hemos conseguido una simulación bastante completa, con el fin de dejar atrás tanta teoría y suposición, lo ideal sería ver el quadrotor en funcionamiento, o en este caso, una representación en 3D en tiempo real del vuelo del quadrotor.

Creamos una pequeña función que se encargue de inicializar los gráficos 3D de la simulación. Definimos la perspectiva, el ángulo de inclinación con el que veremos la simulación y definimos la longitud de los ejes que serán visibles.

Código 4.1 init_plot.m.m.

```
function init_plot

axes('units','normalized','position',[.2 .1 .6 .8]);
axis equal

axis([-5 5 -5 5 -5 5]);
view(30,30)
grid on
hold on

xlabel('x')
```

```
%-----Cámara-----%

camproj perspective
camva(5)
hlight = camlight('headlight');

lighting gouraud
set(gcf,'Renderer','OpenGL')

line([-1 1],[0 0],[0 0])
line([0 0],[-.5 .5],[0 0],'color','r')

end
```

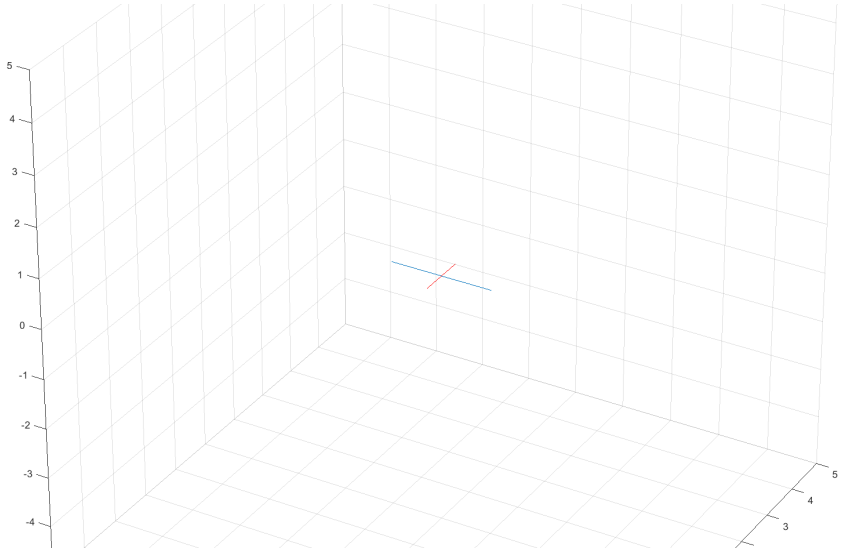


Figura 4.1 Entorno.

Una vez definido el entorno, es necesario implementar la configuración del quadrotor, así como la posición de sus motores y la posición de este en el espacio. El centro de masas del quadrotor se sitúa en el centro de coordenadas (0,0,0) y los ejes X e Y de Matlab se hacen coincidir con los ejes globales (G), de forma que la X es positiva hacia el *Norte* y la Y es positiva hacia el *Oeste*. Al inicio de la simulación los ejes ligados al quadrotor (B) se hacen coincidir con los ejes globales (G).

Creamos para ello una función llamada *define_erle_model.m*. Empezamos por definir la distancia de los motores al centro de masas, el ancho de los brazos del vehículo y el radio de las aspas de los motores.

Código 4.2 define_erle_model.m.

```
erle.plot_arm = 0.141; %Distancia del centro de masas a cada rotor
erle.plot_arm_t = 0.02; % Grosor del brazo
```

```
erle.plot_prop = .08; %Radio de las aspas
```

A continuación definimos las coordenadas de cada uno de los vértices del brazo, tendrá forma de prisma, por lo tanto cuenta con 8 vértices. Los agrupamos en una matriz, cuya fila se corresponde con los vértices que hay que unir para crear cada una de las caras del prisma y creamos la estructura.

Código 4.3 define_erle_model.m.

```
z_positiva = erle.plot_arm_t;
z_negativa = -erle.plot_arm_t;

%Matriz con los vértices del brazo
vertex_matrix_2 = [-0.083,-0.115,z_negativa;
0.115,0.083,z_negativa;
0.083,0.115,z_negativa;
-0.115,-0.083,z_negativa;
-0.083,-0.115,z_positiva;
0.115,0.083,z_positiva;
0.083,0.115,z_positiva;
-0.115,-0.083,z_positiva];

%Matriz que indica las uniones de los vértices (8 vertices)
face_matrix = [1 2 6 5;
2 3 7 6;
3 4 8 7;
4 1 5 8;
1 2 3 4;
5 6 7 8];

X_arm = patch('faces',face_matrix,'vertices',vertex_matrix_2,'facecolor','b',
    'edgecolor',[0 0 0],'facecolor','b');
```

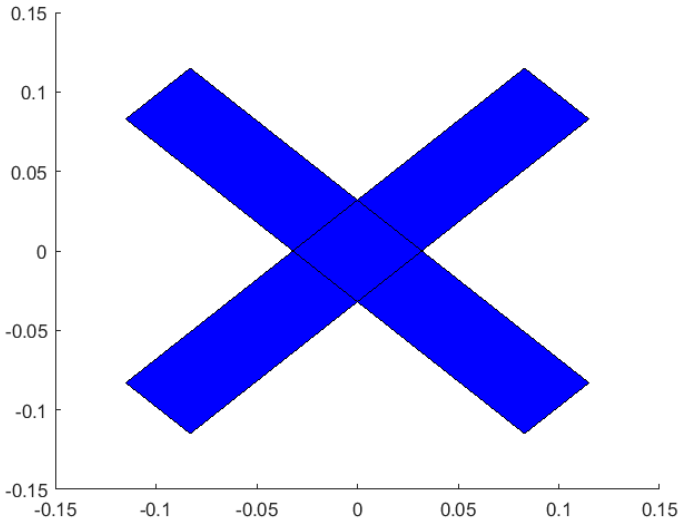


Figura 4.2 Brazos del vehículo.

A continuación colocamos los cuatro motores en los dos brazos que hemos creado Para representarlos creamos círculos cuyo radio es el radio de la hélice que hemos definido anteriormente. Para verificar la orientación pintamos los motores que se encuentran al frente de color verde.

Código 4.4 define_erle_model.m.

```
%Motores
t = 0:pi/10:2*pi;
X = erle.plot_prop*cos(t);
Y = erle.plot_prop*sin(t);
Z = zeros(size(t)) + erle.plot_arm_t;
C = zeros(size(t));
erle.C = C;
hold on;
fplot(0,[-0.15 0.15],'r');
hold on;
y = -0.15:0.01:0.15;
x = zeros(size(y));
plot(x,y,'r');

Motor0 = patch(X+0.1,Y-0.1,Z,C,'facealpha',.1,'facecolor','g');
Motor1 = patch(X-0.1,Y+0.1,Z,C,'facealpha',.1,'facecolor','k');
Motor2 = patch(X+0.1,Y+0.1,Z,C,'facealpha',.1,'facecolor','g');
Motor3 = patch(X-0.1,Y-0.1,Z,C,'facealpha',.1,'facecolor','k');
```

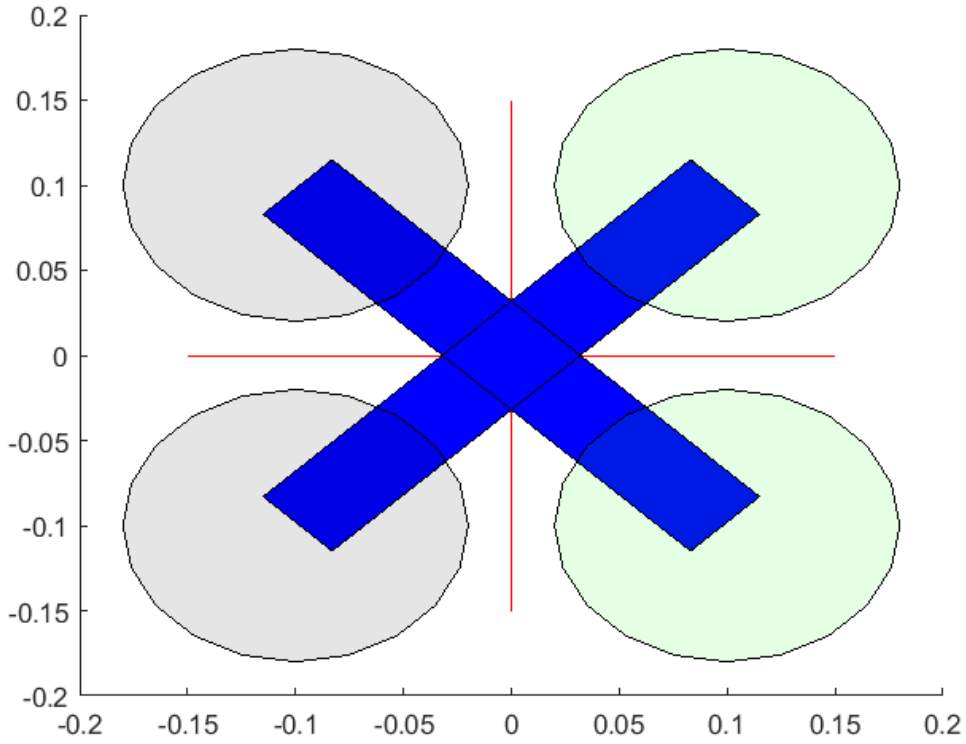


Figura 4.3 Vehículo completo.

Por último tenemos que obtener las coordenadas de cada elemento para poder ir actualizando su posición, de forma que describa el movimiento del quadrotor durante la simulación. Para poder compartir los valores entre las distintas funciones de la simulación se han almacenado en una estructura global.

Código 4.5 define_erle_model.m.

```
erle.X_armX = get(X_arm,'xdata');
erle.X_armY = get(X_arm,'ydata');
erle.X_armZ = get(X_arm,'zdata');

erle.Y_armX = get(Y_arm,'xdata');
erle.Y_armY = get(Y_arm,'ydata');
erle.Y_armZ = get(Y_arm,'zdata');

erle.Motor0X = get(Motor0,'xdata');
erle.Motor0Y = get(Motor0,'ydata');
erle.Motor0Z = get(Motor0,'zdata');

erle.Motor1X = get(Motor1,'xdata');
erle.Motor1Y = get(Motor1,'ydata');
```

```
erle.Motor1Z = get(Motor1,'zdata');

erle.Motor2X = get(Motor2,'xdata');
erle.Motor2Y = get(Motor2,'ydata');
erle.Motor2Z = get(Motor2,'zdata');

erle.Motor3X = get(Motor3,'xdata');
erle.Motor3Y = get(Motor3,'ydata');
erle.Motor3Z = get(Motor3,'zdata');
```

Al inicio de la simulación es necesario establecer la posición del vehículo en el entorno creado por *init_plot.m*, de esto se encarga la función *plot_erle_model.m*. Esta función cargará el modelo creado por *define_elre_model.m* y se encargará de colocar el vehículo en los gráficos creados.

Código 4.6 plot_erle_model.m.

```
load erle_plotting_model

erle.X_arm = patch('xdata',erle.X_armX,'ydata',erle.X_armY,'zdata',erle.
    X_armZ,'facealpha',.9,'facecolor','b');
erle.Y_arm = patch('xdata',erle.Y_armX,'ydata',erle.Y_armY,'zdata',erle.
    Y_armZ,'facealpha',.9,'facecolor','b');
erle.Motor0 = patch('xdata',erle.Motor0X,'ydata',erle.Motor0Y,'zdata',erle.
    Motor0Z,'facealpha',.3,'facecolor','g');
erle.Motor1 = patch('xdata',erle.Motor1X,'ydata',erle.Motor1Y,'zdata',erle.
    Motor1Z,'facealpha',.3,'facecolor','k');
erle.Motor2 = patch('xdata',erle.Motor2X,'ydata',erle.Motor2Y,'zdata',erle.
    Motor2Z,'facealpha',.3,'facecolor','g');
erle.Motor3 = patch('xdata',erle.Motor3X,'ydata',erle.Motor3Y,'zdata',erle.
    Motor3Z,'facealpha',.3,'facecolor','k');
```

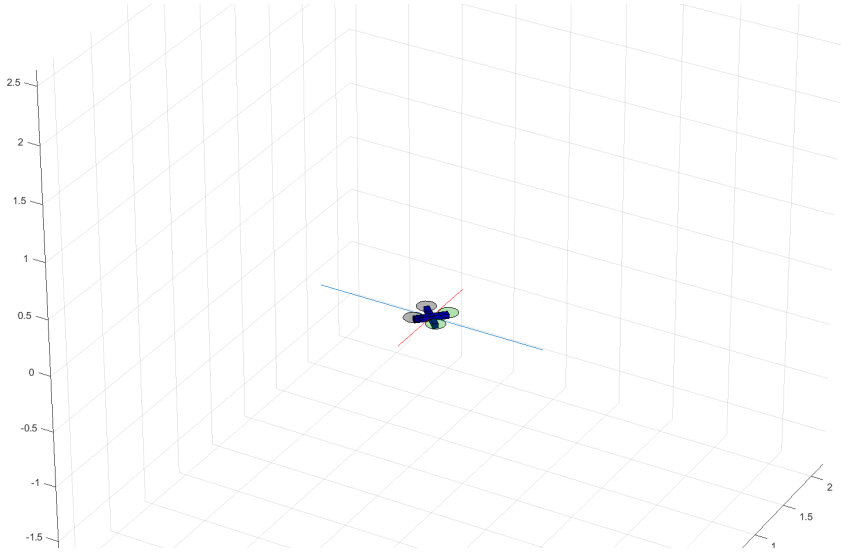


Figura 4.4 Representación del vehículo.

Finalmente queda la función más importante, la que se encarga de actualizar la posición del quadrotor a medida que avanza la simulación. Las coordenadas que tenemos de las componentes del vehículo, tanto para el chasis como para los motores, se encuentran expresadas en el sistema $\{\mathbf{B}\}$, por lo tanto tenemos que transformarlas a coordenadas $\{\mathbf{G}\}$, ya que los desplazamientos se realizan con respecto a este sistema. Primero calculamos las coordenadas de los componentes del quadrotor en ejes globales, y a continuación se le suman las posiciones actuales (X^G,Y^G,Z^G) calculadas en cada iteración, actualizando así la posición del quadrotor en tiempo real.

Código 4.7 plot_erle.m.

```
global erle

roll = erle.roll;
pitch = erle.pitch;
yaw = erle.yaw;
t = .02;
%% Dibujar brazos del quadrotor
[erle.Xtemp,erle.Ytemp,erle.Ztemp] = rotateBFtoGF(erle.X_armX,erle.X_armY,
    erle.X_armZ,roll,pitch,yaw);
set(erle.X_arm,'xdata',erle.Xtemp+erle.X,'ydata',+(erle.Ytemp+erle.Y),'zdata'
    ,-(-erle.Ztemp-erle.Z));

[erle.Xtemp,erle.Ytemp,erle.Ztemp] = rotateBFtoGF(erle.Y_armX,erle.Y_armY,
    erle.Y_armZ,roll,pitch,yaw);
set(erle.Y_arm,'xdata',erle.Xtemp+erle.X,'ydata',+(erle.Ytemp+erle.Y),'zdata'
    ,-(-erle.Ztemp-erle.Z));

%% Dibujar los motores
```

```
[erle.Xtemp,erle.Ytemp,erle.Ztemp] = rotateBFtoGF(erle.Motor0X,erle.Motor0Y,
    erle.Motor0Z,roll,pitch,yaw);
set(erle.Motor0,'xdata',erle.Xtemp+erle.X,'ydata',+(erle.Ytemp+erle.Y),'zdata
    ',-(-erle.Ztemp-erle.Z));

[erle.Xtemp,erle.Ytemp,erle.Ztemp] = rotateBFtoGF(erle.Motor1X,erle.Motor1Y,
    erle.Motor1Z,roll,pitch,yaw);
set(erle.Motor1,'xdata',erle.Xtemp+erle.X,'ydata',+(erle.Ytemp+erle.Y),'zdata
    ',-(-erle.Ztemp-erle.Z));

[erle.Xtemp,erle.Ytemp,erle.Ztemp] = rotateBFtoGF(erle.Motor2X,erle.Motor2Y,
    erle.Motor2Z,roll,pitch,yaw);
set(erle.Motor2,'xdata',erle.Xtemp+erle.X,'ydata',+(erle.Ytemp+erle.Y),'zdata
    ',-(-erle.Ztemp-erle.Z));

[erle.Xtemp,erle.Ytemp,erle.Ztemp] = rotateBFtoGF(erle.Motor3X,erle.Motor3Y,
    erle.Motor3Z,roll,pitch,yaw);
set(erle.Motor3,'xdata',erle.Xtemp+erle.X,'ydata',+(erle.Ytemp+erle.Y),'zdata
    ',-(-erle.Ztemp-erle.Z));
```

4.2 Simulación Completa

Con lo definido en la sección anterior, ya tenemos todo lo necesario para hacer una simulación completa sobre el modo de vuelo guiado que hemos diseñado a lo largo de este documento. La simulación estará compuesta por las funciones que se han ido presentando durante el diseño del control, con pequeñas modificaciones en alguna de ellas.

Como se indicaba en la sección final del capítulo anterior, en esta simulación es donde se va a probar el funcionamiento del control de posición, para ello definimos una función llamada *position_control.m* que tendrá como entradas las posiciones de referencia (X^G,Y^G) y nos proporciona como salida la actitud del quadrotor.

Código 4.8 position_control.m.

```
function out = position_control(in)

X_GF = in(1);
Y_GF = in(2);

global erle;

%% Control de X
erle.X_des_filt = (2.5/(2.5+erle.Tm))*erle.X_des_filt_1 + (erle.Tm/(2.5+erle.
    Tm))*X_GF;
% Cálculo del error
X_ek = erle.X_des_filt - erle.X_GF;
% Incremento de la integral del error
erle.X_Int_ek = erle.X_Int_ek + erle.Tm*X_ek;
```

```
% Controlador PID
erle.pitch_des = ((0.1271*(X_ek + erle.X_Int_ek/7.5000 + 1.6667*((X_ek - erle.
    X_ek_1)/erle.Tm))));
% Saturación
erle.pitch_des = +erle.pitch_des;
erle.pitch_des = min(erle.pitch_max,max(-erle.pitch_max,erle.pitch_des));

erle.X_des_filt_1 = erle.X_des_filt;
erle.X_ek_1 = X_ek;

%% Control de Y
erle.Y_des_filt = (2.5/(2.5+erle.Tm))*erle.Y_des_filt_1 + (erle.Tm/(2.5+erle.
    Tm))*Y_GF;
% Cálculo del error
Y_ek = erle.Y_des_filt - erle.Y_GF;
% Incremento de la integral del error
erle.Y_Int_ek = erle.Y_Int_ek + erle.Tm*Y_ek;
% Controlador PI
erle.roll_des = ((0.1271*(Y_ek + erle.Y_Int_ek/7.5000 + 1.6667*((Y_ek - erle.
    Y_ek_1)/erle.Tm))));
% Saturación
erle.roll_des = -erle.roll_des;
erle.roll_des = min(erle.roll_des,max(-erle.roll_des,erle.roll_des));

erle.Y_des_filt_1 = erle.Y_des_filt;
erle.Y_ek_1 = Y_ek;

out = [erle.roll_des,erle.pitch_des];

end
```

Cabe recordar, que tenemos que invertir el signo del ángulo ϕ calculado, según nos indicaban Ecuación 2.47 calculadas en el Capítulo 2, sección Sección 2.5.

Se recogen tanto el control de altura como el control de actitud en la misma función, se combinan las funciones *atttiude_control.m* (Código 3.8) y *altitud_control.m* (Código 3.12).

En *ecuaciones_dinamicas.m* se implementa tanto la dinámica translacional no linealizada (Ecuación 2.28) como la dinámica rotacional (Ecuación 2.43) calculadas en el Capítulo 2.

4.2.1 Simulación sin Simulink

Se han usado variables globales para las realimentaciones y para la comunicación entre las distintas funciones de la simulación, con el fin de evitar que una función pudiese tener más de 6 parámetros de entrada.

Para establecer las posiciones de referencia podemos utilizar funciones de tipo escalón, ya que la intención es mover el quadrotor a una posición y mantenerlo ahí. Primero establecemos los tiempos de las señales de referencia y la amplitud de los escalones, tenemos cuatro referencias para nuestro sistema, las tres posiciones en ejes G y el ángulo ψ.

Primero inicializamos la simulación gráfica, cargamos los parámetros de la simulación y definimos los tiempos de las señales y de la simulación.

Código 4.9 simulacion.m.

```
clear all;
close all;
clc;

addpath Simulador

global erle;

init_plot;
plot_erle_model;
erle_variables;

erle.contador = 0;

erle.T_simulacion = 15;% segundos
erle.T_escalon_altura = 2;%(segundos);
erle.T_escalon_posicion = 4;%(segundos);
erle.T_escalon_yaw = 7;%(segundos)
```

Creamos un bucle *for* que se ejecuta hasta que se complete el tiempo de simulación que hemos definido. En cada iteración se comprueban las condiciones de tiempo para activar las señales de referencia.

Código 4.10 simulacion.m.

```
for time = 0:erle.Tm:erle.T_simulacion

%% Referencias
if(time >= erle.T_escalon_altura)

erle.Z_des = 1;
end
if(time >= erle.T_escalon_posicion)
erle.X_des = 2;
erle.Y_des = 1;

end
if(time < erle.T_escalon_yaw)
erle.yaw_des = 0;
end
if(time >= erle.T_escalon_yaw)
erle.yaw_des = 0*erle.Deg_Rad;
end
```

Dentro del bucle *for* añadimos las funciones encargadas del control, se ejecutan de forma secuencial, de forma que primero se ejecuta el control a más alto nivel y por último el control de más bajo nivel. Se empieza por llamar a *position_control.m* (Código 4.8) que a partir de las posiciones de referencia $erle.X_{des}$ y $erle.Y_{des}$ obtiene los ángulos ϕ y θ de referencia. A continuación se hace una llamada a *attitude_control.m* que a partir de la salida de *position_control.m* más las referencias en $\mathbf{Z}^{\mathbf{G}}$ y ψ obtiene las aceleraciones angulares (p,q,r). Para finalizar el lazo de control, se llama a *rate_control.m* (Figura 6.11), que a partir de las salidas de *attitude_control.m* obtiene los momentos ($\tau_\phi, \tau_\theta, \tau_\psi$) en ejes $\{\mathbf{B}\}$. Por último se calculan las velocidades de los motores y se saturan las actuaciones en la función *saturacion_actuaciones.m* (Código 3.3). Finalmente se llama a la función que recoge la dinámica del sistema *ecuaciones_dinamicas.m*.

Código 4.11 simulacion.m.

```
    %% Lazo de control
Actitud = position_control([erle.X_des,erle.Y_des]);

Actitud_corr = yaw_correccion(Actitud);

rate_des = attitude_control([erle.Z_des,Actitud_corr,erle.yaw_des]);

U_des = rate_control([rate_des(2),rate_des(3),rate_des(4)]);

U_sat = saturacion_actuaciones([rate_des(1),U_des]);

out = ecuaciones_dinamicas(U_sat);
```

Al final del bucle debemos añadir la función encargada de actualizar la simulación gráfica de nuestro quadrotor (Código 4.7).

Código 4.12 Simulacion.m.

```
  if(erle.contador == 3)
plot_erle;
drawnow
erle.contador = 0;
end

erle.contador = erle.contador + 1;
```

4.2.2 Simulación con Simulink

Como ya se había comentado las funciones se han implementado de forma que las simulaciones se puedan hacer tanto usando simulink, como si no. La principal diferencia entre esta simulación y la anterior, es que disponemos de un diagrama de bloques que nos da una idea más clara sobre la estructura del control y que ahora es SIMULINK® el que se encarga de controlar los ciclos de la simulación. Por lo tanto ahora nuestra función *Simulacon.m* solamente se usará para definir tiempos y valores de referencia, así como para lanzar la ejecución en simulink.

Código 4.13 Simulacion.m + simulink.

```
clear all;
close all;
clc;

addpath Simulador

global erle;

erle.T_simulacion = 15;% segundos
erle.T_escalon_yaw = 7;%(segundios)


init_plot;
plot_erle_model;
erle_variables;

erle.X_t = 4; %Tiempo de escalón X
erle.Y_t = 4; %Tiempo de escalón Y
erle.X_des = 2; %Amplitud del escalón X
erle.Y_des = 1; %Amplitud del escalón Y

SimOut = sim('diagrama_de_bloques');
```

El diagrama final sigue la estructura del diagrama creado para el control de actitud (Figura 3.15), pero con algunos cambios. Se ha añadido el control de posición y se ha añadido la simulación gráfica.

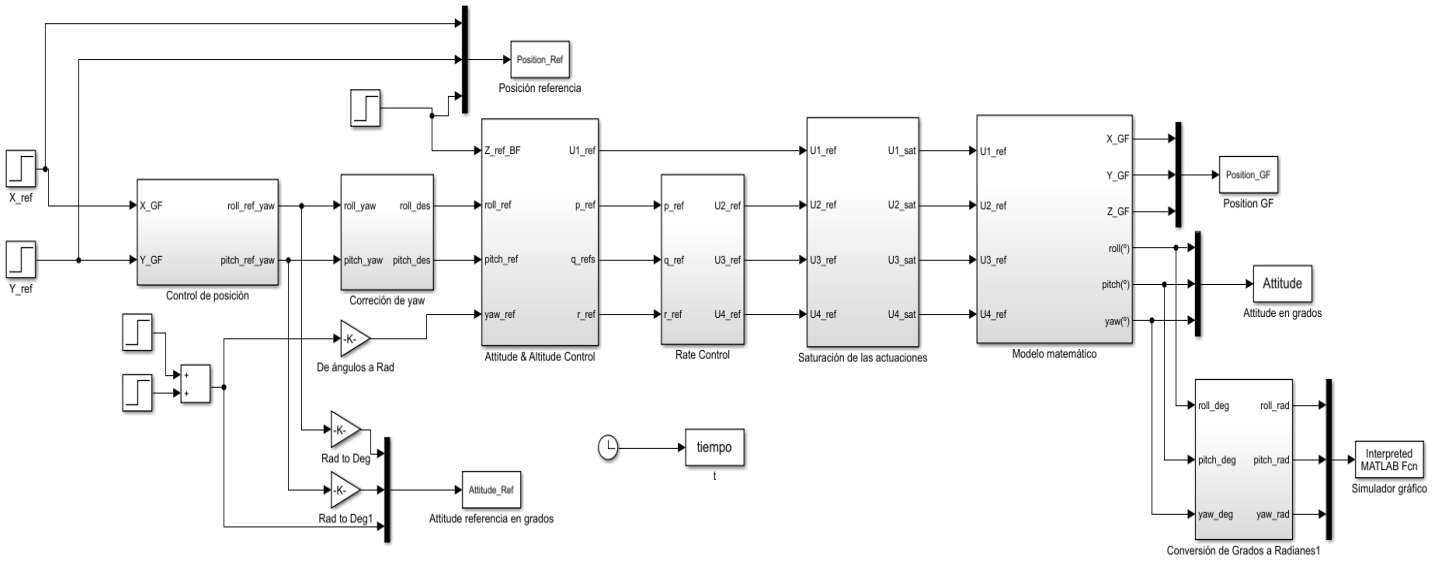


Figura 4.5 Diagrama de bloques final.

4.3 Análisis de Resultados

Una vez implementada la simulación por completo, hay que verificar que todo funciona correctamente y la respuesta del sistema así como su vuelo, se corresponde con lo que hemos diseñado a lo largo de este documento. Lo más interesante del modo de vuelo que hemos creado es asegurarnos de que el quadrotor alcanza la posición que definimos como referencia y se mantiene en ella, sin oscilar y eliminando la deriva provocada por la inercia del quadrotor en movimiento.

Tabla 4.1 Referencias.

Descripción	Variable	Valor	Unidades	Tiempo
Posición en X^G	$erle.X\_des$	2	metros(m)	4
Posición en Y^G	$erle.Y\_des$	1	metros(m)	4
Posición en Z^G	$erle.Z\_des$	1	metros(m)	1
Ángulo de yaw(ψ)	$erle.yaw\_des$	0	grados	1

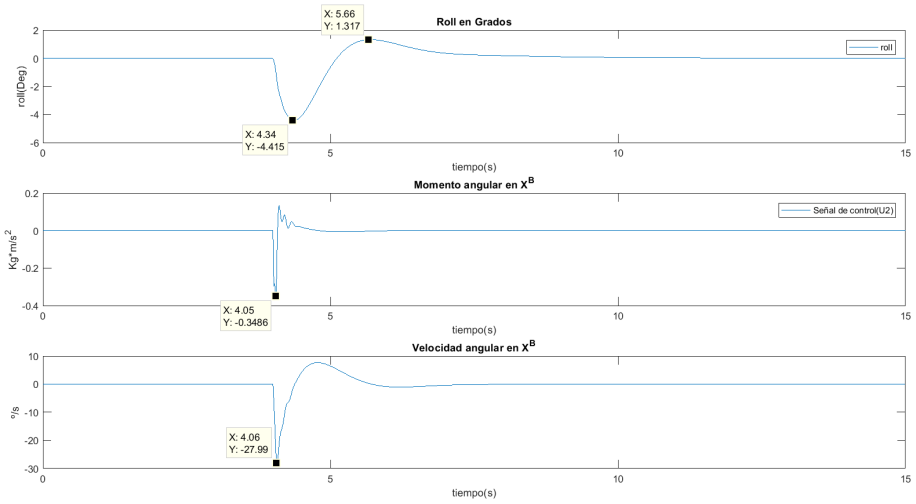


Figura 4.6 Roll (X=2,Y=1,Z=1,ψ=0).

Como vemos en la Figura 4.6 cuando se produce un cambio en la Y de referencia vemos como varía el ϕ. Se ha dado como referencia un valor positivo de Y^G y vemos que según la configuración del quadrotor que veíamos en la Figura 2.2 y la orientación inicial que le hemos dado, el ángulo debe de ser negativo para provocar un movimiento positivo en $\mathbf{Y^G}$.

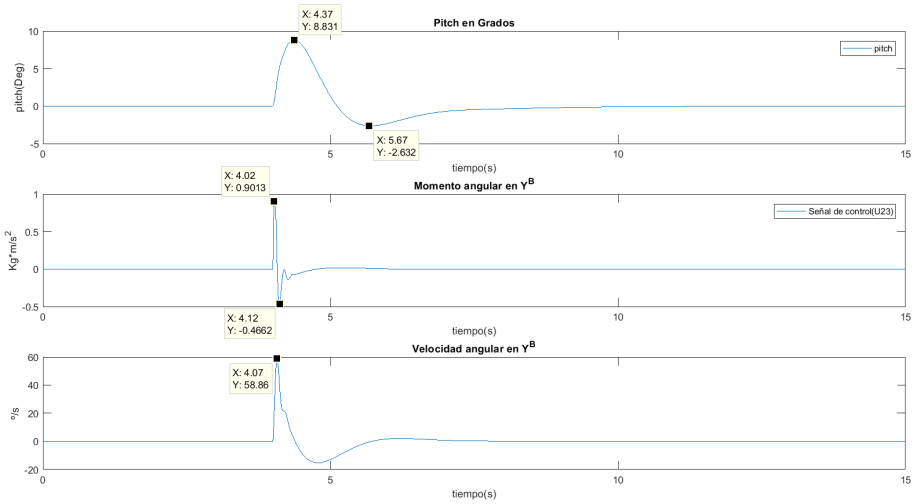


Figura 4.7 Pitch (X=2,Y=1,Z=1,ψ=0).

Por el contrario, para producir un movimiento positivo en X se debe producir un ángulo de θ positivo. Hemos dado como referencia una X^G positiva, y podemos ver como se produce una variación positiva del θ. Hay que aclarar que esto es debido a que los ejes del quadrotor se encuentran alineados con los ejes globales, si se hubiese producido una variación del ψ, esto no tendría por que ser así, ya que los desplazamientos se están realizando en torno a $\{\mathbf{G}\}$ y no a los ejes cuerpo del vehículo.

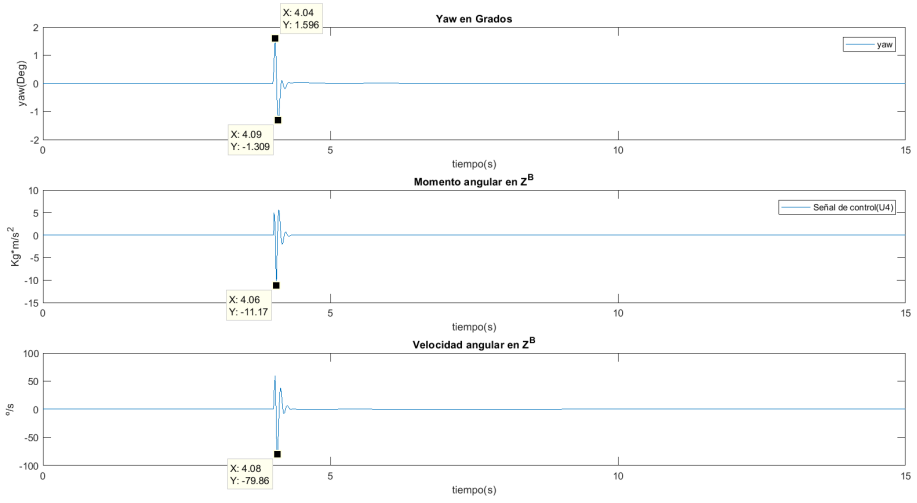


Figura 4.8 Yaw (X=2,Y=1,Z=1,ψ=0).

Vemos como el ψ se mantiene constante sobre 0 grados, aunque se observan pequeñas perturbaciones durante los cambios de posición, pero de valores muy pequeños que no afectan al comportamiento del vehículo.

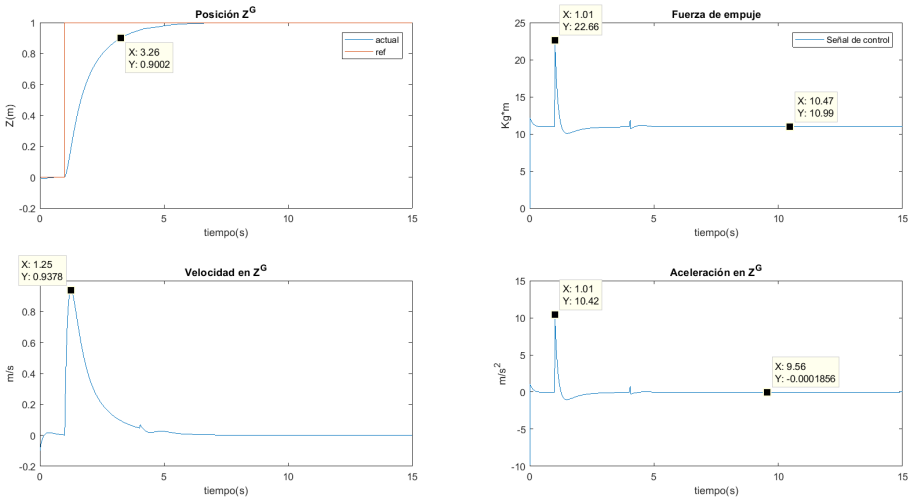
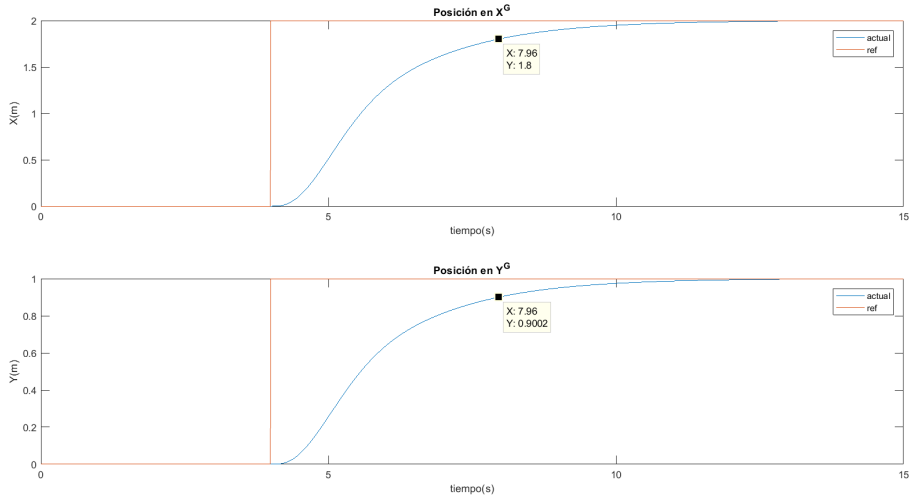
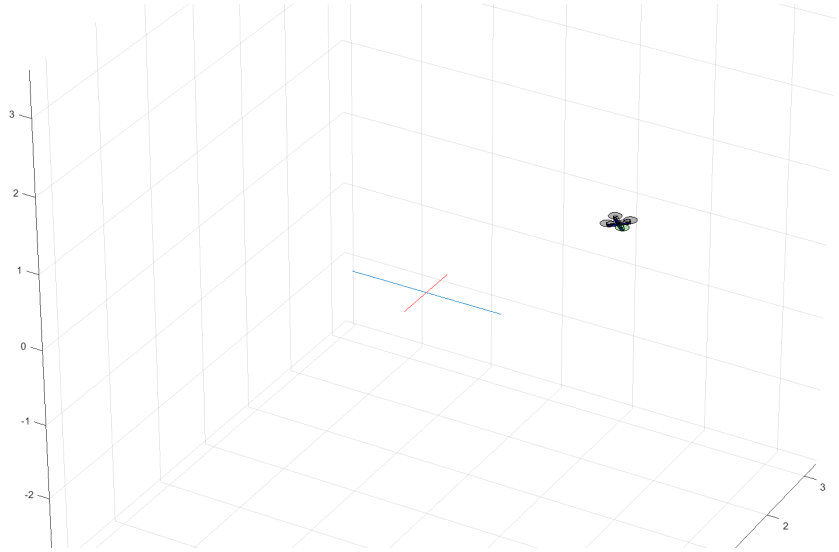


Figura 4.9 Altura en $\mathbf{Z^G}$ (X=2,Y=1,Z=1,ψ=0).

Vemos como alcanza perfectamente la altura deseada, y aunque posteriormente se produzcan variaciones en $\mathbf{X^G}$ e $\mathbf{Y^G}$, el vehículo se mueve a las nuevas posiciones establecidas sin perder altura. También se pueden ver la velocidad y la aceleración en $\mathbf{Z^G}$, que se mantienen a 0 una vez se ha alcanzado la referencia. Y por último vemos como la fuerza de empuje F_T^B se mantiene igual al peso del vehículo una vez se alcanza la referencia.



(a) Representación 2D.



(b) Representación 3D.

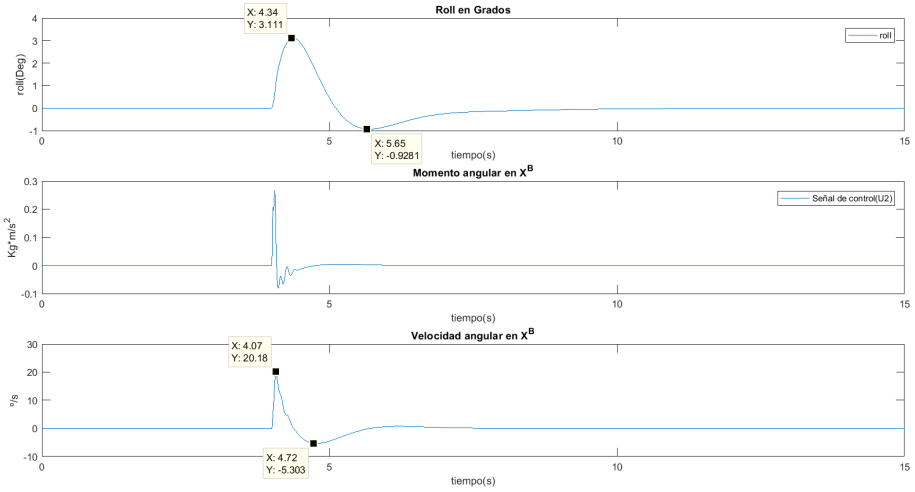
Figura 4.10 Posición (X=2,Y=1,Z=1,ψ=0).

Por último vemos la posición del quadrotor durante la simulación y como alcanza las referencias en los tiempos estimados.

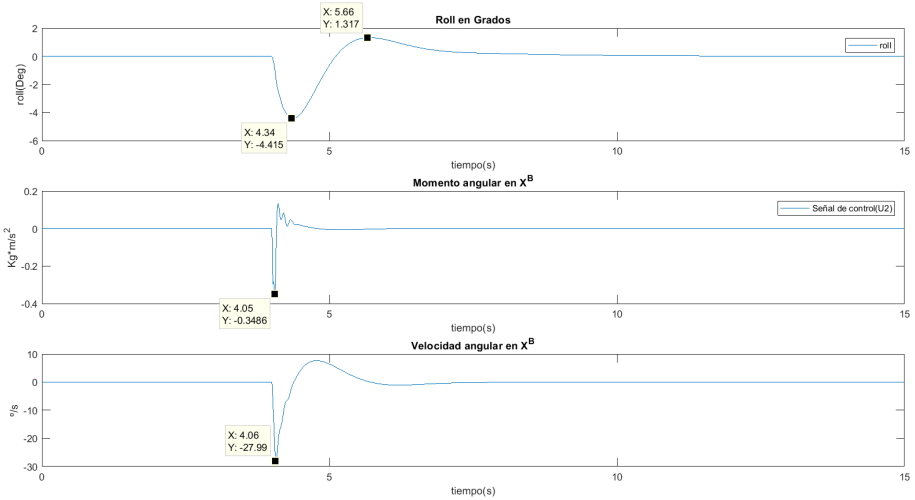
Vamos a realizar otro experimento antes de terminar esta sección, de forma que ahora vamos a cambiar la orientación inicial del quadrotor, vamos a comenzar con un ángulo ψ de 45, y vamos a ver como el quadrotor vuelve a desplazarse hasta ese punto sin importar la orientación que tenga, ya que la posición se está tratando en coordenadas $\{\mathbf{G}\}$, pero ahora los valores de ϕ y θ no serán los mismos que en la simulación anterior.

Tabla 4.2 Referencias.

Descripción	Variable	Valor	Unidades	Tiempo
Posición en X^G	$erle.X\_des$	2	metros(m)	4
Posición en Y^G	$erle.Y\_des$	1	metros(m)	4
Posición en Z^G	$erle.Z\_des$	1	metros(m)	1
Ángulo de yaw(ψ)	$erle.yaw\_des$	45	grados	1

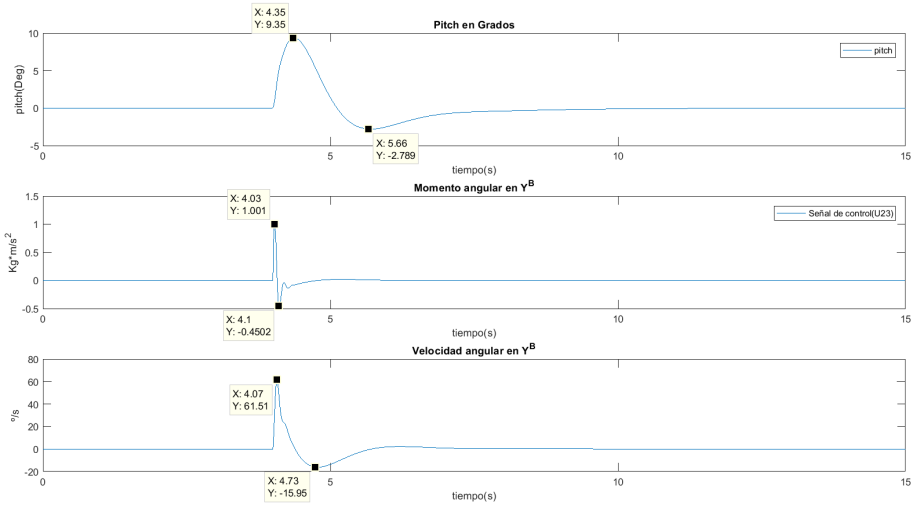


(a) Roll($\psi = 45$).

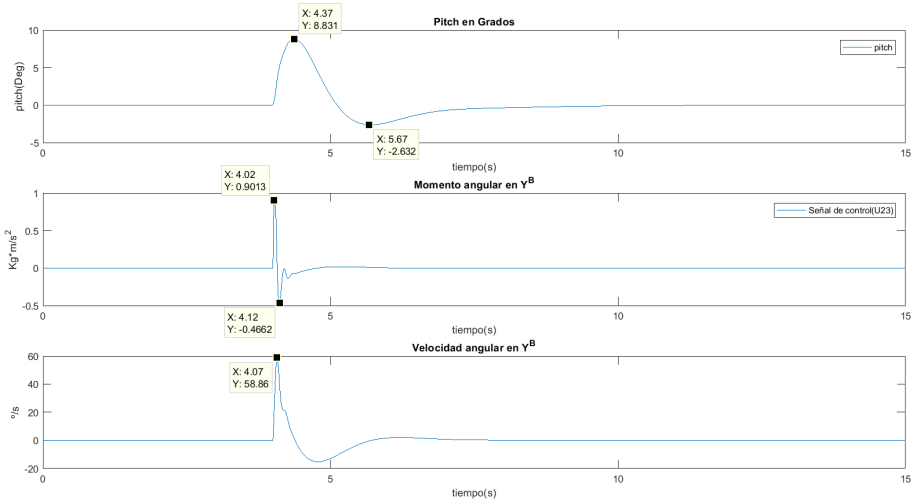


(b) Roll($\psi = 0$).

Figura 4.11 Roll.



(a) Pitch($\psi = 45$).



(b) Pitch($\psi = 0$).

Figura 4.12 Pitch.

Vemos como es necesario un ángulo ϕ positivo para conseguir llegar a la posición deseada cuando $\psi = 45$, mientras que para alcanzar la posición con $\psi = 0$ se comenzaba el movimiento con un ángulo ϕ negativo, el ángulo θ inicial sigue positivo inicialmente.

Se puede ver como a medida que el quadrotor se acerca a la referencia los ángulos de inclinación se aproximan a cero y además, el quadrotor realiza una pequeña inclinación en sentido contrario para compensar la inercia del movimiento y alcanzar la referencia sin producirse sobreoscilaciones.

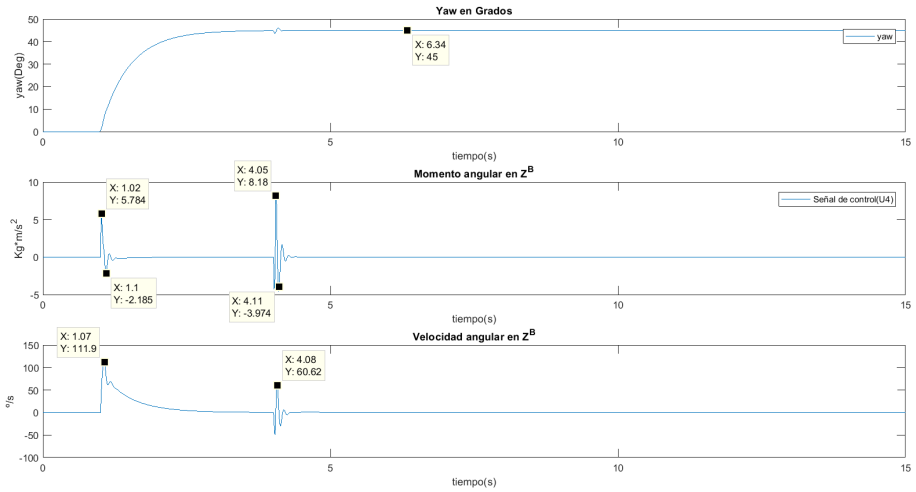
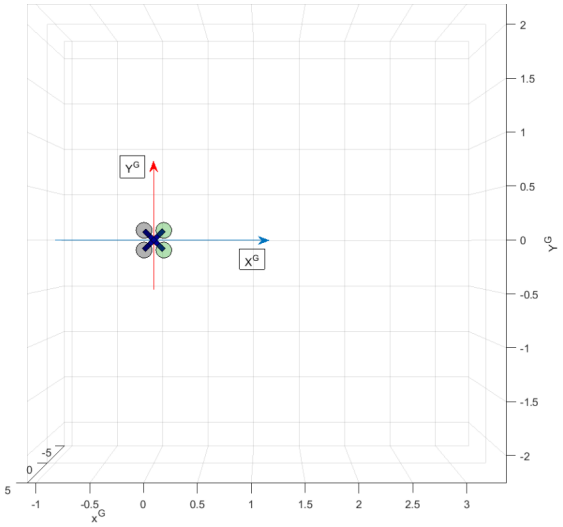
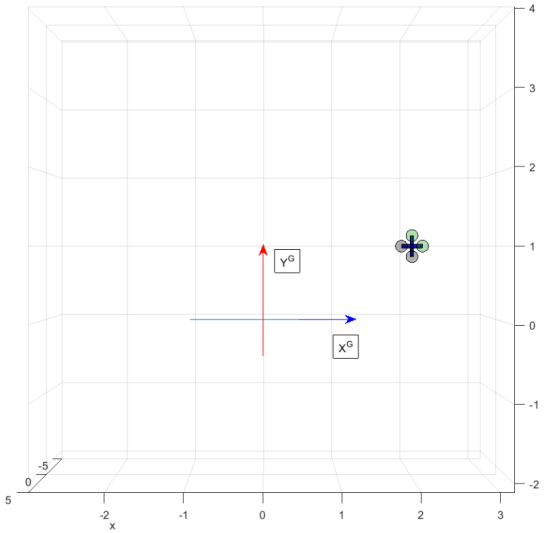


Figura 4.13 Yaw (X=2,Y=1,Z=1,ψ=-45º).

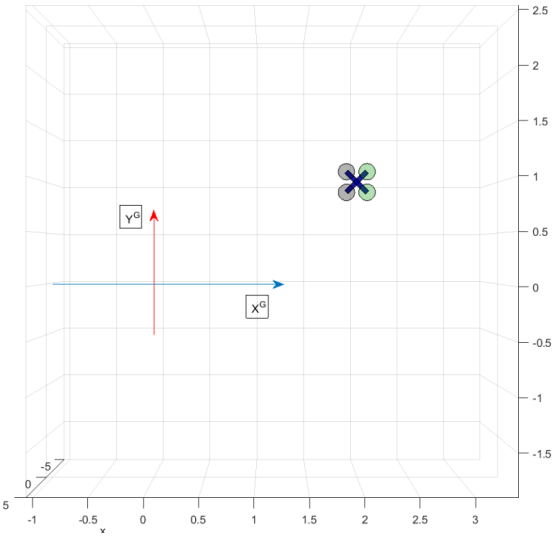
Aquí vemos como el ψ se mantiene a 45. Se omiten las gráficas de altura y posición, ya que son idénticas a las del experimento anterior, pero vamos a ver el efecto que ha tenido el ψ y la orientación con la que ha terminado el quadrotor en ambos experimentos.



(a) Posición inicial.



(b) Posición final $\psi = 45$.



(c) Posición final $\psi = 0$.

Figura 4.14 Efecto del ψ.

Ahora se va a cambiar el signo de los desplazamientos para verificar el funcionamiento del control en otras direcciones.Además se le van a dar dos posiciones de referencia, primero se moverá a una y luego mientras está estable en el primer punto, deberá de moverse al segundo.

Tabla 4.3 Referencias.

Descripción	Variable	Valor	Unidades	Tiempo
Posición en X^G	$erle.X\_des$	-2	metros(m)	4
Posición en Y^G	$erle.Y\_des$	-2	metros(m)	4
Posición en Y^G	$erle.Y\_des$	-1	metros(m)	15
Posición en Z^G	$erle.Z\_des$	1.5	metros(m)	1
Ángulo de yaw(ψ)	$erle.yaw\_des$	0	grados	1

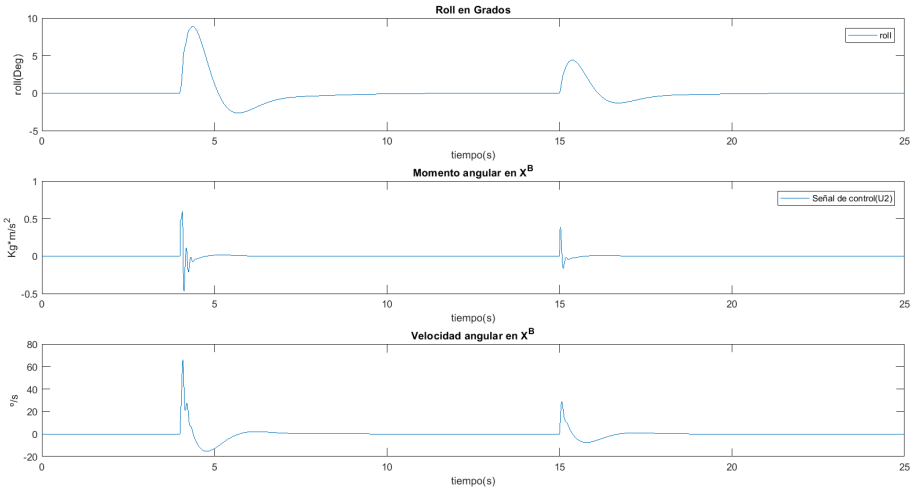


Figura 4.15 Roll (X=-2,Y=-3,Z=1.5,ψ=0).

Se puede ver el efecto que provocan los cambios en la posición de referencia en $\mathbf{Y^G}$ en el ángulo ϕ, debido a que no hemos dado ningún valor de ψ y los ejes del quadrotor se encuentran alineados con los ejes globales.

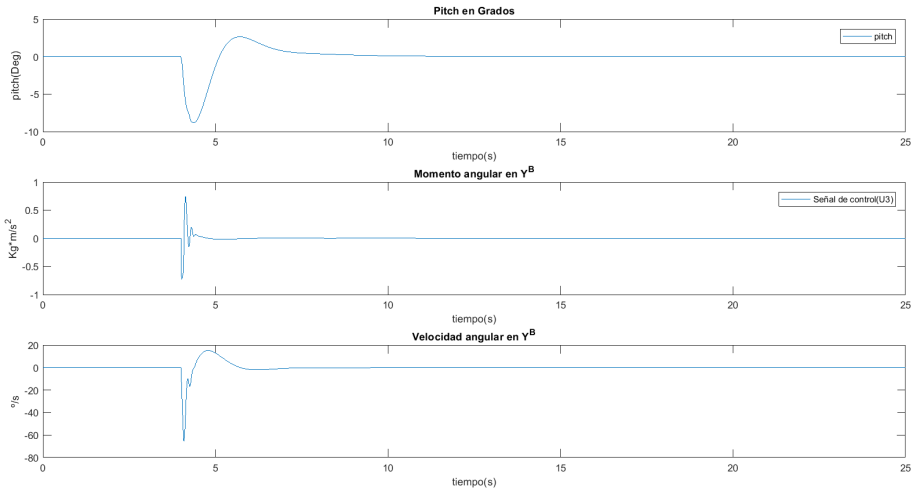


Figura 4.16 Pitch (X=-2,Y=-3,Z=1.5,ψ=0).

Con respecto a θ, vemos que al estar los ejes alineados, una variación en $\mathbf{Y}^{\mathbf{G}}$ no le afecta.

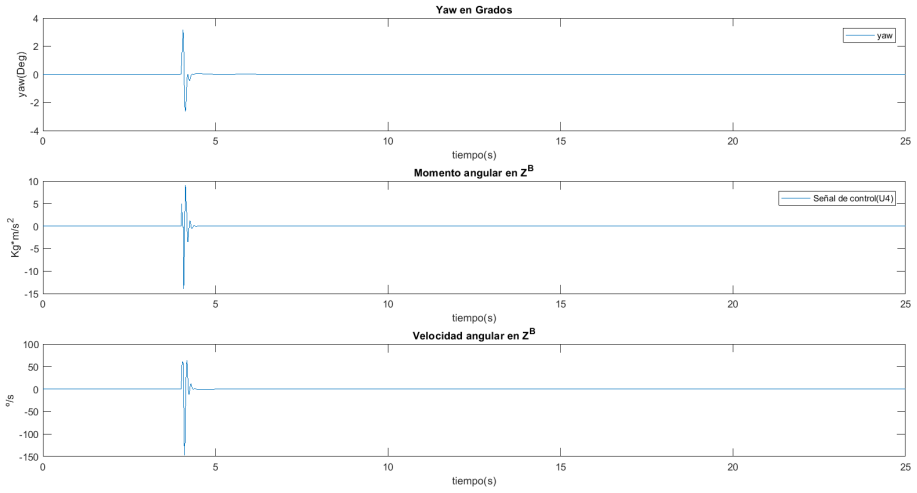


Figura 4.17 Yaw (X=-2,Y=-3,Z=1.5,ψ=0).

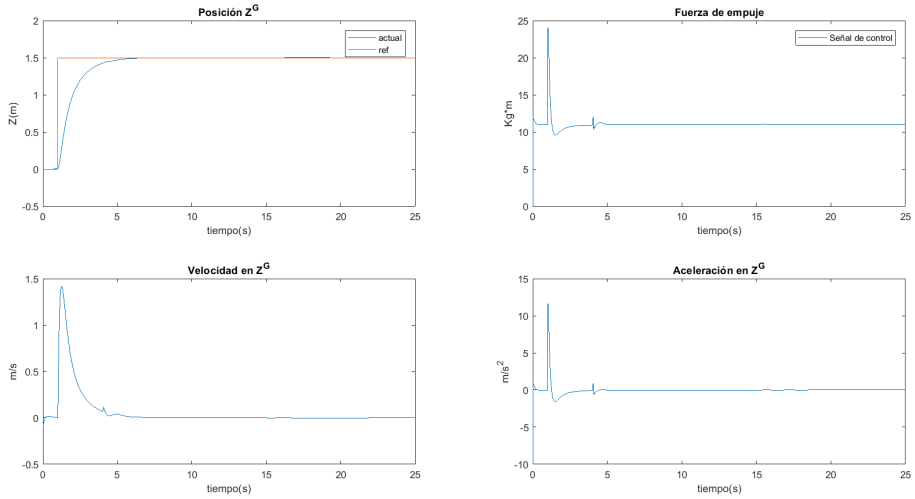


Figura 4.18 Altura en $\mathbf{Z^G}$ (X=-2,Y=-3,Z=1.5,ψ=0).

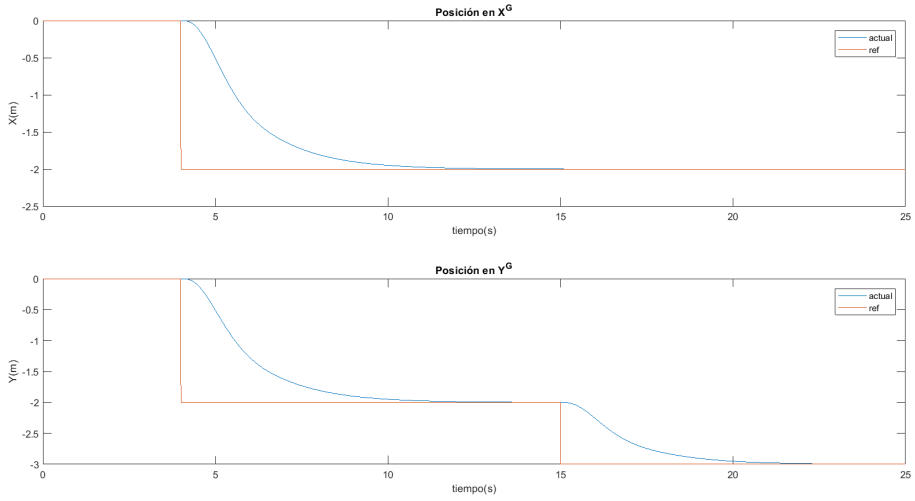


Figura 4.19 Posición (X=-2,Y=-3,Z=1.5,ψ=0).

Se puede ver como el quadrotor se estabiliza en torno al primer punto de referencia (-2,-2,1.5) y a los 15 *segundos* se desplaza al punto (-2,-3,1.5) manteniendo la altura.

Por último se va a realizar un experimento parecido al anterior, pero esta vez vamos variar la referencia en altura durante el vuelo.

Tabla 4.4 Referencias.

Descripción	Variable	Valor	Unidades	Tiempo
Posición en X^G	$erle.X\_des$	-2	metros(m)	4
Posición en Y^G	$erle.Y\_des$	-2	metros(m)	4
Posición en Z^G	$erle.Z\_des$	1	metros(m)	1
Posición en Z^G	$erle.Z\_des$	2	metros(m)	6
Ángulo de yaw(ψ)	$erle.yaw\_des$	0	grados	1

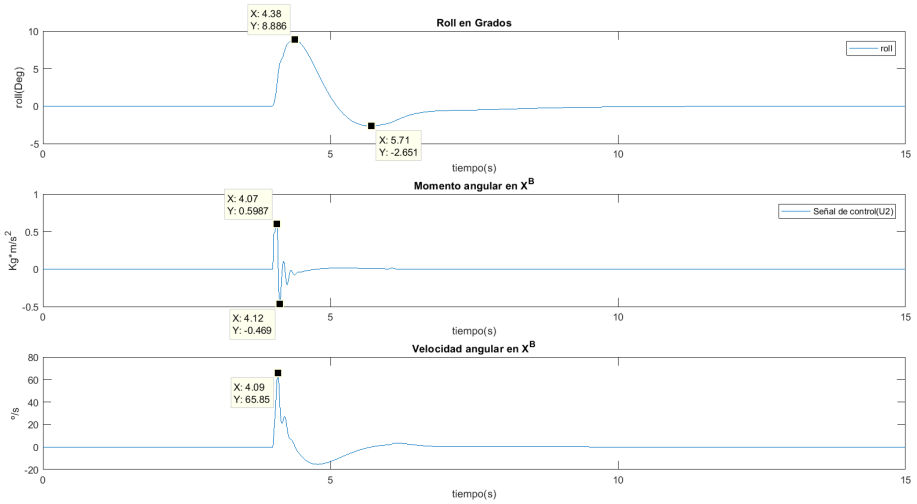


Figura 4.20 Roll (X=-2,Y=-2,Z=2,ψ=0).

Ahora vemos como al desplazarnos a una posición negativa, y según la situación del quadrotor,cuyos ejes cuerpo están alineados con los ejes globales, vemos como un ϕ positivo provoca un desplazamiento negativo en $\mathbf{Y^G}$.

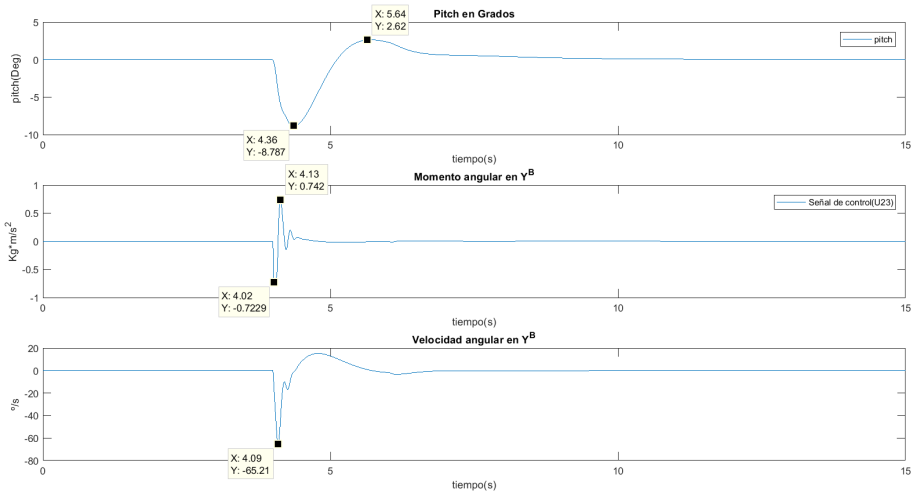


Figura 4.21 Pitch (X=-2,Y=-2,Z=2,ψ=0).

Por el contrario, un θ negativo provoca un desplazamiento negativo en $\mathbf{X^G}$.

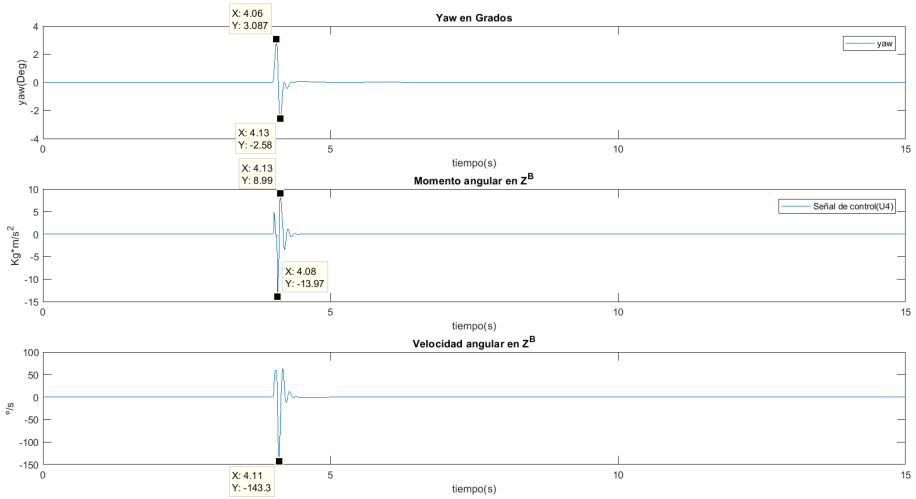


Figura 4.22 Yaw (X=-2,Y=-2,Z=2,ψ=0).

El ψ se mantiene constante en torno a 0, ya que no le hemos introducido ninguna referencia.

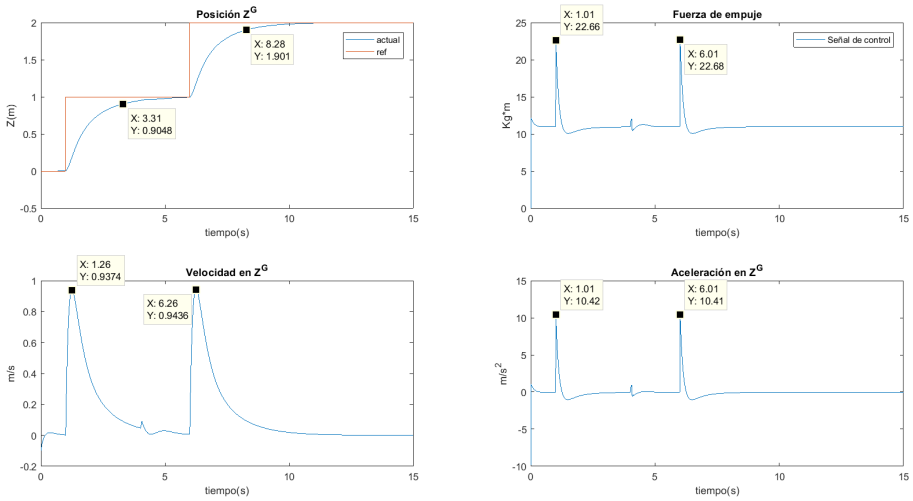
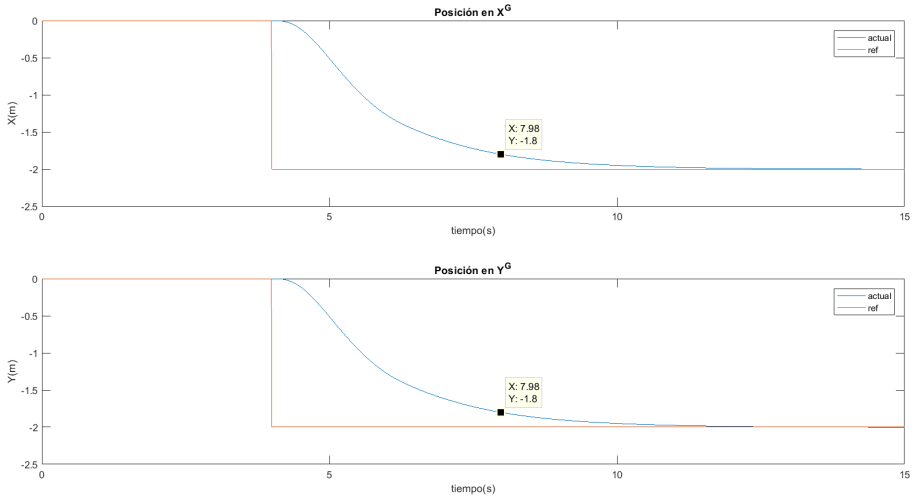
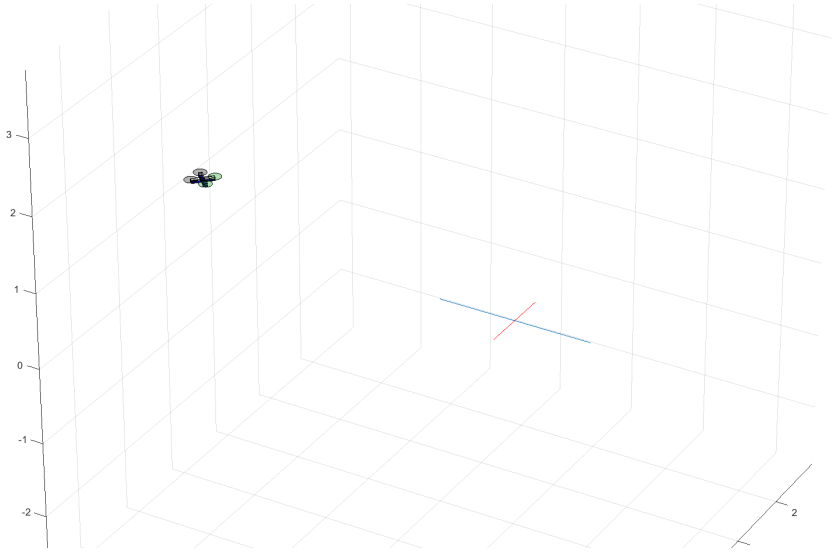


Figura 4.23 Altura en $\mathbf{Z^G}$ (X=-2,Y=-2,Z=2,ψ=0).

Podemos ver el doble cambio de altura, debido a la variación en la referencia, primero alcanza la altura de un metro, y a continuación sube a los 2 metros mientras se desplaza hacia la posición deseada.



(a) Representación 2D.



(b) Representación 3D.

Figura 4.24 Posición (X=-2,Y=-2,Z=2,ψ=0).

Finalmente vemos como alcanza las posiciones indicadas en los ejes $\mathbf{X^G}$ e $\mathbf{Y^G}$ en los tiempos esperados.

Tras varios experimentos podemos ver que el modo de vuelo diseñado cumple con lo que se ha ido realizando durante el documento. Hemos conseguido diseñar un modo de vuelo autónomo basándonos en un modelo teórico, el siguiente paso será aplicar esta estrategia de control diseñada hasta ahora en una simulación más realista, con el fin de obtener un modo de vuelo autónomo con una estructura sencilla y fácil de interpretar, que pueda ser implementado en vehículos reales.

Debido a los buenos resultados que se han obtenido en las pruebas anteriores, se va a realizar una última

prueba añadiendo la dinámica de los motores, con el fin de hacer un poco más real esta simulación y ver el efecto que puede provocar esta dinámica en el control diseñado. El cálculo de esta dinámica se realiza en el Capítulo 6, donde se realizan experimentos con los motores y con un entorno más realista.

Esta dinámica se añade después del cálculo de las velocidades de los motores, y antes de realizar la saturación, por lo que ahora se debe modificar la función *saturacion_actuaciones.m*. Se ha realizado la implementación en SIMULINK® , debido a la facilidad que ofrece para implementar funciones de transferencia, y se ha dividido la función *saturacion_actuaciones.m* en dos funciones, *saturaciones_1.m* y *saturaciones_2.m*.

Código 4.14 saturaciones_1.m.

```
function U = saturaciones_1(in)

global erle;

U1 = in(1);
U2 = in(2);
U3 = in(3);
U4 = in(4);


w0_2 = U1/(4*erle.Kt) - U2/(2*sqrt(2)*erle.l*erle.Kt) - U3/(2*sqrt(2)*erle.l*
    erle.Kt) - U4/(4*erle.Kd);
w1_2 = U1/(4*erle.Kt) + U2/(2*sqrt(2)*erle.l*erle.Kt) + U3/(2*sqrt(2)*erle.l*
    erle.Kt) - U4/(4*erle.Kd);
w2_2 = U1/(4*erle.Kt) + U2/(2*sqrt(2)*erle.l*erle.Kt) - U3/(2*sqrt(2)*erle.l*
    erle.Kt) + U4/(4*erle.Kd);
w3_2 = U1/(4*erle.Kt) - U2/(2*sqrt(2)*erle.l*erle.Kt) + U3/(2*sqrt(2)*erle.l*
    erle.Kt) + U4/(4*erle.Kd);

%% Dinámica de los motores

erle.w0 = sqrt(w0_2);
erle.w1 = sqrt(w1_2);
erle.w2 = sqrt(w2_2);
erle.w3 = sqrt(w3_2);


U = [erle.w0,erle.w1,erle.w2,erle.w3];

end
```

Código 4.15 saturaciones_2.m.

```
function U = saturaciones_2(in)

global erle;
```

```
w0 = in(1);
w1 = in(2);
w2 = in(3);
w3 = in(4);


%% Saturación de las velocidades
if w0 > erle.w_max
w0 = erle.w_max;
end
if w0 < erle.w_min
w0 = erle.w_min;
end
if w1 > erle.w_max
w1 = erle.w_max;
end
if w1 < erle.w_min
w1 = erle.w_min;
end

if w2 > erle.w_max
w2 = erle.w_max;
end
if w2 < erle.w_min
w2 = erle.w_min;
end

if w3 > erle.w_max
w3 = erle.w_max;
end
if w3 < erle.w_min
w3 = erle.w_min;
end

erle.w0 = w0;
erle.w1= w1;
erle.w2 = w2;
erle.w3 = w3;

%% Señales de control
U1_out = erle.Kt*(erle.w0^2+erle.w1^2+erle.w2^2+erle.w3^2);
U2_out = (sqrt(2)*erle.l*erle.Kt/2)*(+erle.w1^2+erle.w2^2-erle.w0^2-erle.w3
    ^2);
U3_out = (sqrt(2)*erle.l*erle.Kt/2)*(-erle.w0^2-erle.w2^2+erle.w1^2+erle.w3
    ^2);
U4_out = erle.Kd*(-erle.w0^2-erle.w1^2+erle.w2^2+erle.w3^2);

erle.U1 = U1_out;
```

```
erle.U2 = U2_out;
erle.U3 = U3_out;
erle.U4 = U4_out;

U = [U1_out,U2_out,U3_out,U4_out];

end
```

La función *saturaciones_1.m* calcula las velocidades de los motores, estas velocidades son las referencias que se introducen en la dinámica de los motores. La función *saturaciones_2.m* se encarga de saturar las velocidades finales y calcular tanto la fuerza de empuje como los momentos angulares en el sistema $\{\mathbf{B}\}$.

La función de transferencia que define la dinámica de los motores tiene la siguiente forma:

$$G(s) = \frac{\omega(s)}{PWM(s)} = \frac{K_{DC}}{\tau \cdot s + 1} \tag{4.1}$$

Esta función tiene como referencia valores PWM, por lo tanto se deben convertir las velocidades calculadas por *saturaciones_1.m* de rad/s a señales PWM,tras varios experimentos en el Capítulo 6, se dedujo que simplemente debemos sumarle 1000. Además, esta dinámica se calculo en torno a un punto de operación donde la entrada tenía un valor de 1200 y daba como resultado una salida de $199.4(rad/s)$.

Ahora el bloque encargado de realizar la saturación de las actuaciones queda de la siguiente forma:

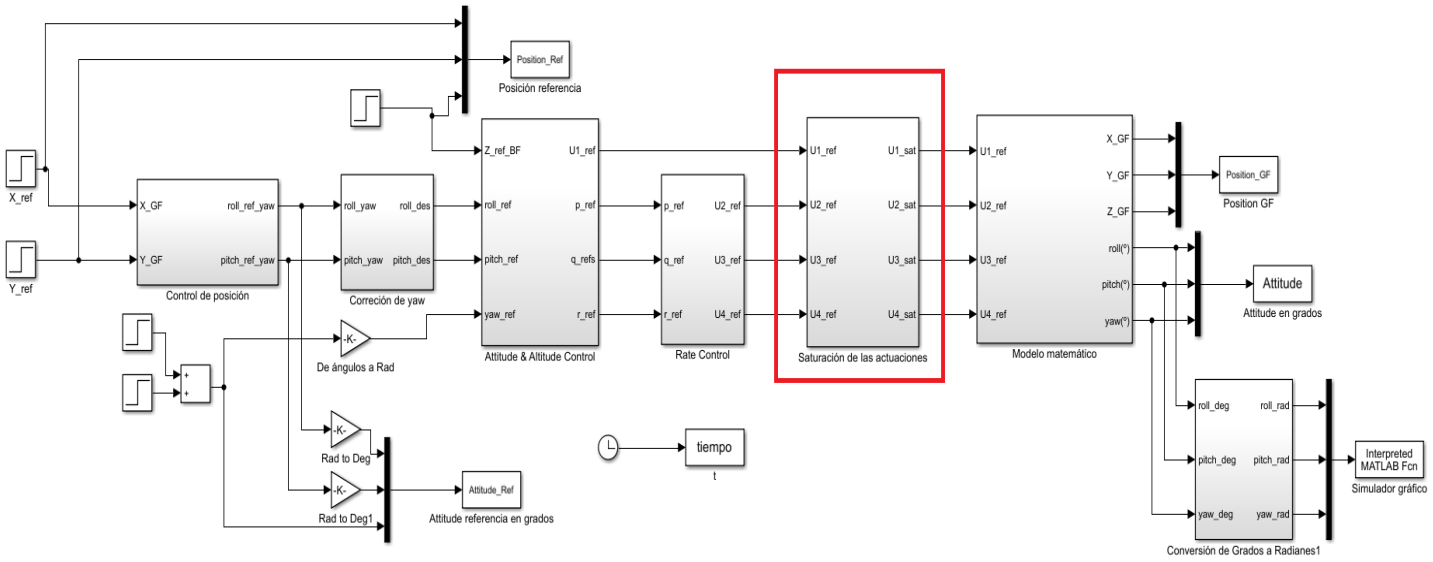


Figura 4.25 Situación del bloque de saturación.

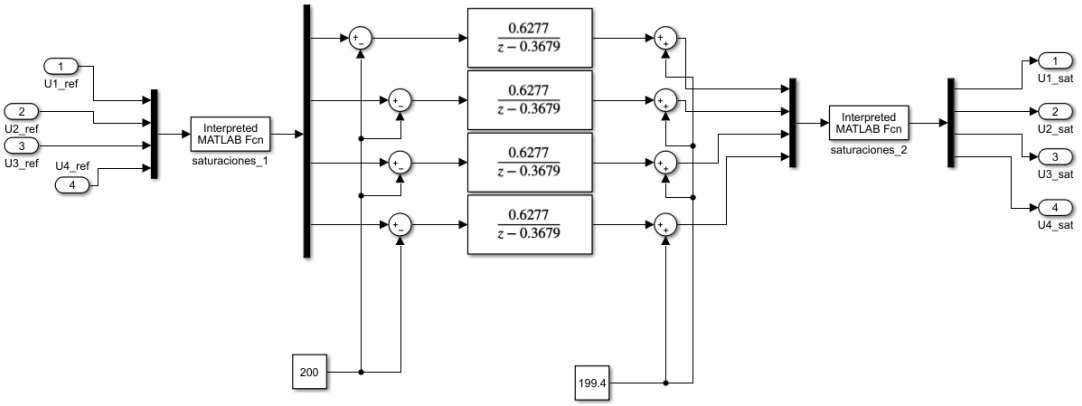
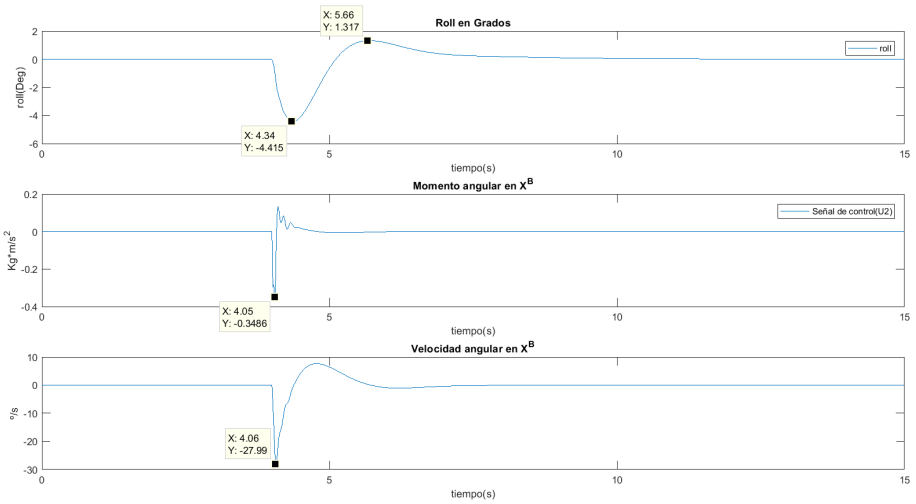
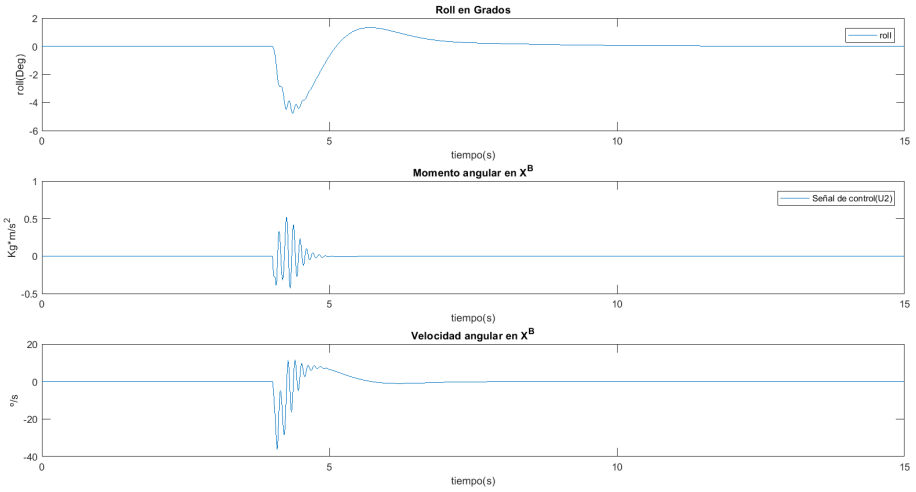


Figura 4.26 Bloque de saturación con la dinámica de los motores.

Se realizaron varios experimentos y se pudo ver que el control se comporta correctamente a pesar de la presencia de esta dinámica. Se añade a continuación uno de los experimentos realizados en esta sección, concretamente el de la Tabla 4.1, con la dinámica de los motores, y se compara con el que se había realizado.

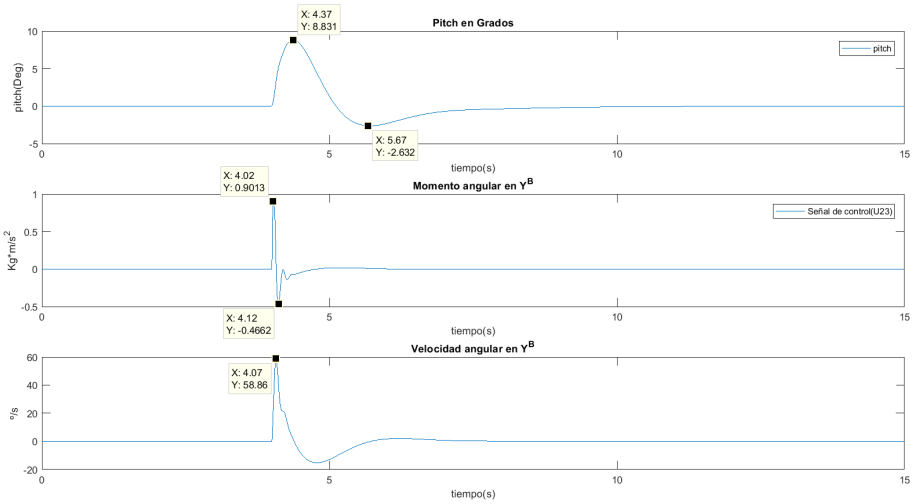


(a) Sin dinámica motores.

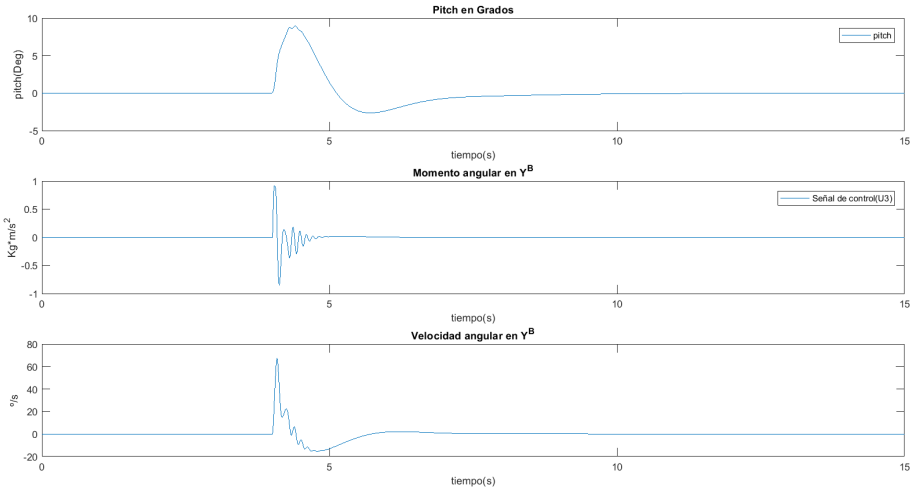


(b) Con dinámica motores.

Figura 4.27 Roll (X=2,Y=1,Z=1,ψ=0).

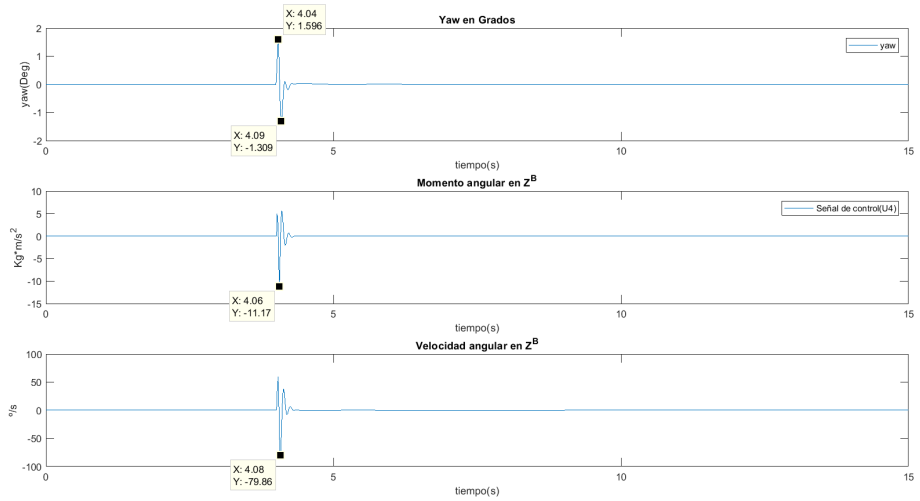


(a) Sin dinámica motores.

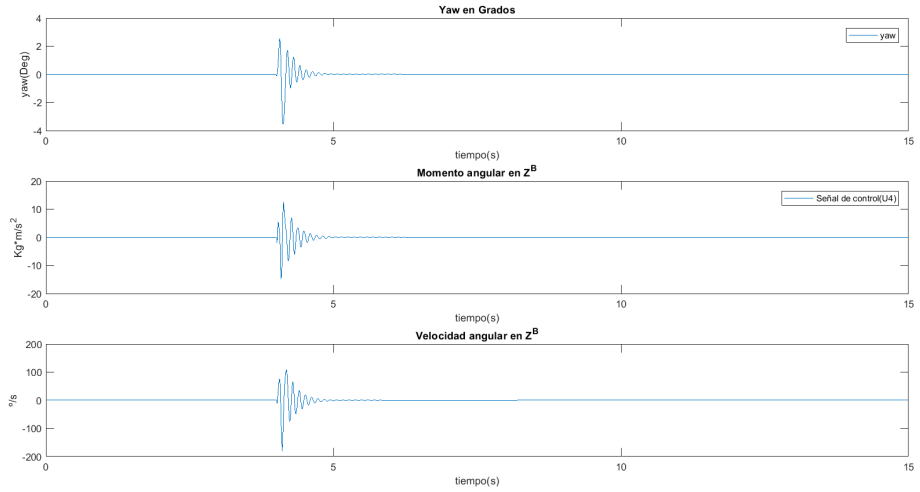


(b) Con dinámica motores.

Figura 4.28 Pitch (X=2,Y=1,Z=1,ψ=0).

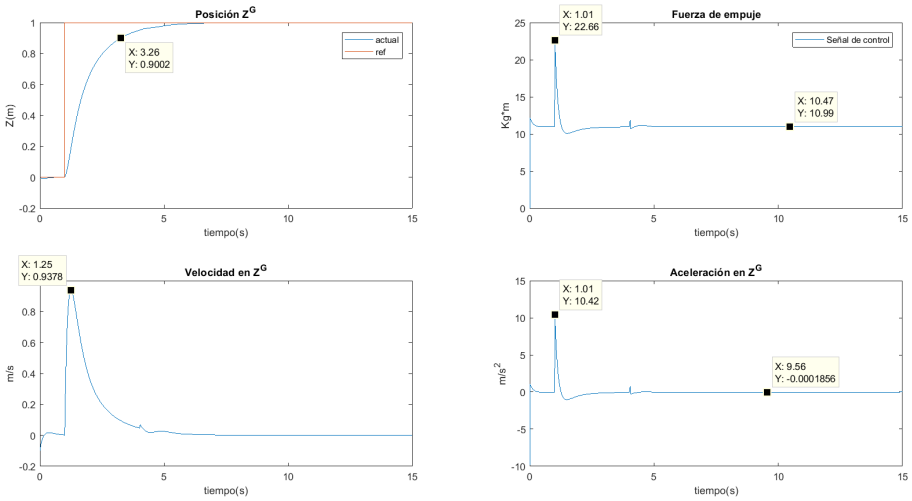


(a) Sin dinámica motores.

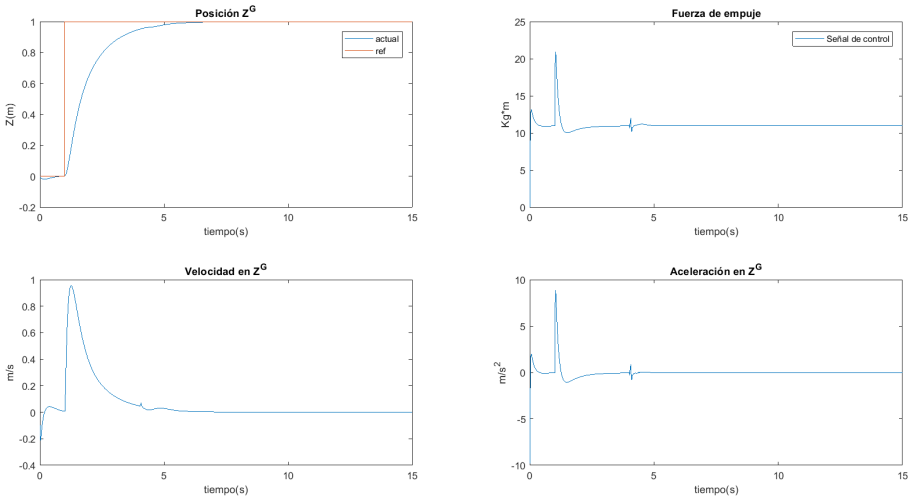


(b) Con dinámica motores.

Figura 4.29 Yaw (X=2,Y=1,Z=1,ψ=0).

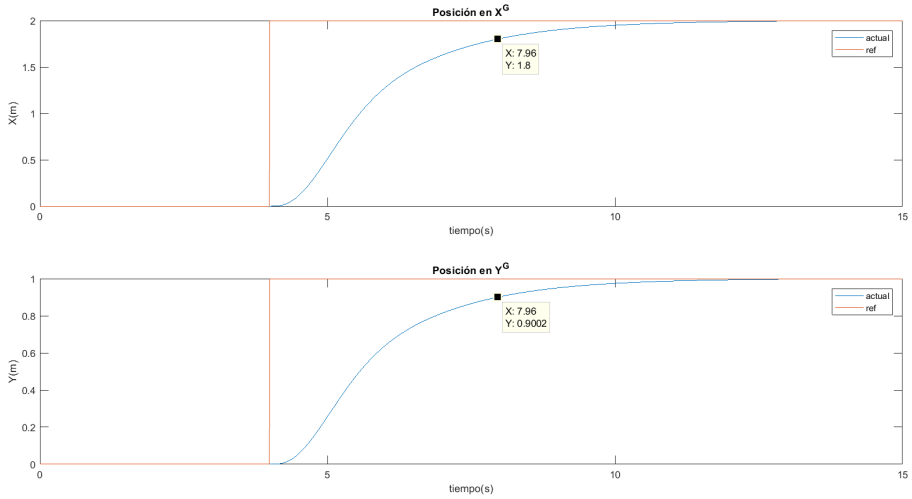


(a) Sin dinámica motores.

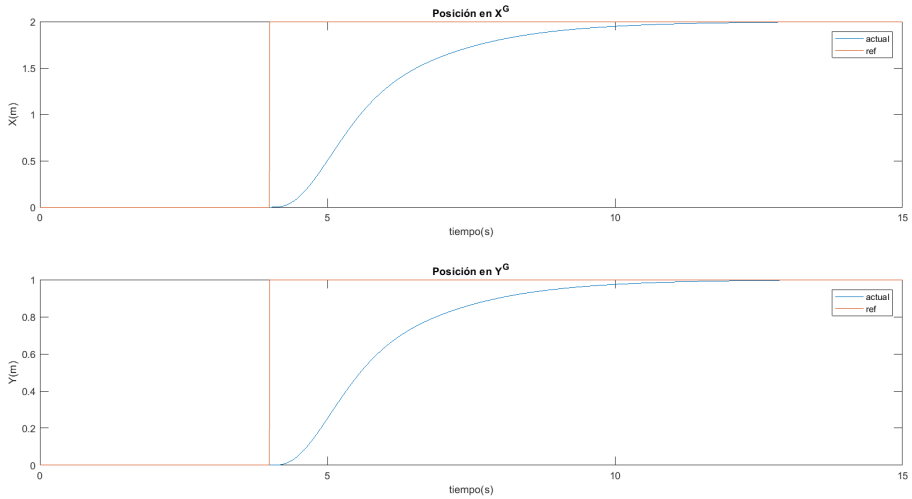


(b) Con dinámica motores.

Figura 4.30 Altura en $\mathbf{Z^G}$ (X=2,Y=1,Z=1,ψ=0).



(a) Sin dinámica motores.



(b) Con dinámica motores.

Figura 4.31 Posición (X=2,Y=1,Z=1,ψ=0).

Podemos ver como el funcionamiento del control se mantiene correctamente. Se puede observar un ligero aumento en las oscilaciones de los controles de bajo nivel, pero nada que resulte un problema a la hora del vuelo. El control de altura y de posición, a penas se ve afectado, la incorporación de esta dinámica a penas deteriora el control diseñado.

5 Ardupilot, ROS y Gazebo

Actualmente disponemos de una simulación teórica basada en las ecuaciones dinámicas del vehículo y que nos permite hacernos una idea del comportamiento del quadrotor durante su vuelo. Se ha diseñado en torno a un modelo teórico, el siguiente paso será incorporar lo obtenido hasta ahora a un entorno más realista. Para esto vamos a utilizar una simulación en GAZEBO® , donde disponemos de un modelo más complejo del quadrotor y que se acerca más a la realidad. En esta simulación vamos a disponer de un código real de un autopiloto, donde se incorporan lectura de sensores, telemetría, comunicaciones,..., y que puede ser implementado en cualquier quadrotor y al que vamos a añadir el modo de vuelo que hemos diseñado. Además disponemos un sistema de comunicaciones con el que controlar la simulación durante el vuelo y acceder a medidas que describen el comportamiento del vehículo. La combinación de todos estos elementos hace que contemos con un entorno que representa fielmente a la realidad.

5.1 Componentes de la Simulación

5.1.1 Erle-Copter

Este es el nombre que recibe el quadrotor que hemos utilizado para este proyecto, aunque el hardware del mismo no llega a ser utilizado en el presente documento, pero disponemos de un modelo simulado que lo representa fielmente.

Figura 5.1 Erle-Copter.

Se trata del primer quadrotor inteligente basado en Linux, lo que sirve como estímulo para los adeptos

al software libre. Utiliza marcos robóticos como ROS® (Robot Operating System) y está controlado por el cerebro artificial Erle-Brain 2. El software de Erle-Copter permite varios modos de vuelo, y destaca especialmente la posibilidad de instalar aplicaciones en el quadrotor usando cualquier dispositivo con un navegador.
Por todo esto se considera una herramienta perfecta con la que desarrollar estudios, investigaciones y experimentos, como es el caso del presente documento.

5.1.2 Erle-Brain

Es un cerebro artificial desarrollado por la empresa Erle Robotics que actúa como un pequeño ordenador Linux, el cual tiene ROS® preinstalado. Proporciona todo el software y los sensores necesarios para convertir el quadrotor en un vehículo autónomo.
Proporciona una Unidad de Control de Vuelo (FCU o *Flight Control Unit*), encargada de aportar controles básicos de vuelo, también denominado autopiloto. Entre los sensores integrados de los que dispone, cabe destacar que contamos con una IMU, un barómetro, GPS y una brújula digital, sensores que nos permiten estimar la posición y la orientación del quadrotor. Además de todo lo mencionado anteriormente, permite conexiones por I2C, UART, Ethernet y USB.

Figura 5.2 Erle Brain.

5.1.3 ArduPilot

Podemos entender un autopiloto como software que permite al vehículo desplazarse por su entorno de forma autónoma sin la necesidad de la intervención humana.
Para este trabajo se ha optado por el software conocido como **Ardupilot**. Este es un autopiloto de código abierto, lo que hace que esté en continuo desarrollo, además, resulta de gran ayuda en la construcción de vehículos autónomos de todo tipo y es compatible con Erle-Brain. **ArduPilot** posee distintos firmwares a elegir dependiendo del vehículo que usemos, en nuestro caso vamos a utilizar un quadrotor, por lo tanto se usará el firmware que se conoce como **ArduCopter**[[22]].

Figura 5.3 ArduPilot.

Posee una amplia variedad de modos de vuelo preprogramados, modos manuales que permiten el control del vuelo mediante acciones del piloto y modos de vuelos autónomos, que permiten el vuelo sin la intervención humana. Además, se trata de un software flexible y personalizable, de manera que , aunque la gran mayoría de parámetros y configuraciones están predefinidas, pueden ser modificadas y ampliadas a gusto del usuario.

5.1.4 Software In The Loop(SITL)

Este simulador permite verificar el funcionamiento del quadrotor sin necesidad de utilizar un sistema físico. Se trata de una compilación del código ArduPilot en C++ que permite realizar pruebas sin necesidad de hardware. Mediante **SITL** se aprovecha la portabilidad de **ArduPilot** que permite que sea implementable en una gran variedad de plataformas, siendo un ordenador una de ellas.[7].

Cuando se ejecuta **SITL**, los datos de los sensores provienen de un modelo de dinámica de vuelo en un simulador de vuelo. ArduPilot tiene una amplia gama de simuladores integrados y puede interactuar con simuladores externos, como el simulador GAZEBO® .

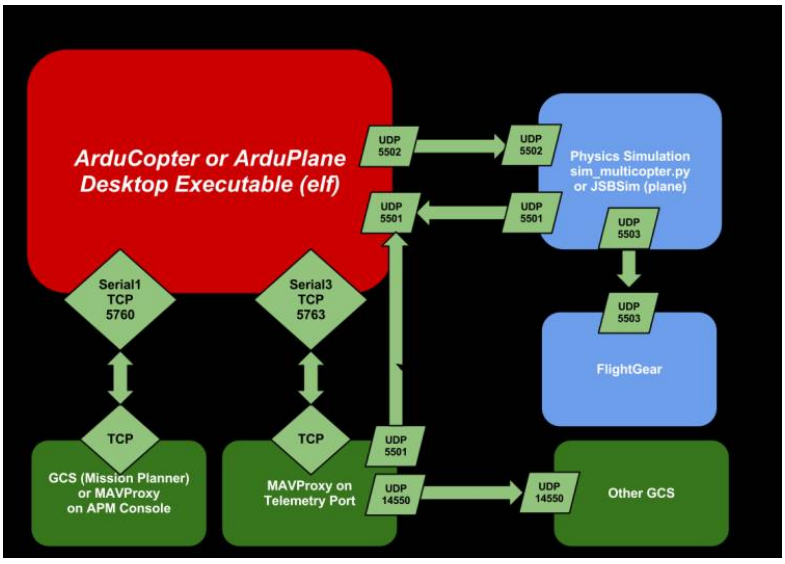


Figura 5.4 SITL.

5.1.5 Gazebo

GAZEBO® es un simulador dinámico en 3D que nos permite analizar el comportamiento un sistema robótico en un entorno virtual. Tiene la capacidad de simular tanto la dinámica como la cinemática mediante un potente motor de física, lo cuál hace de él un simulador bastante fiel a la realidad.

Figura 5.5 Gazebo.

Entre sus muchas características destacan sus múltiples motores de simulación de comportamiento físico, motor de renderizado avanzado, soporte para plugins, un enorme repositorio con la mayoría de robots comerciales y una extensa gama de sensores y cámaras. Además tiene una relación nativa con ROS el cada vez más popular sistema operativo para robots basado en Linux, y el cuál se describirá a continuación.

5.1.6 Robot Operating System (ROS)

ROS® consiste en un framework flexible para el desarrollo de aplicaciones orientadas a la implementación en sistemas robotizados. Se trata de una colección de herramientas y librerías con el objetivo de simplificar la creación de un sistema robusto y complejo en una amplia gama de plataformas orientadas a la robótica. Proporciona los servicios estándar de un sistema operativo, tales como abstracción de hardware, control de dispositivos a bajo nivel, intercambio de mensajes y gestión de paquetes.

Figura 5.6 ROS.

Es un software de código abierto, de forma que brinda la oportunidad de reutilizar aplicaciones existentes desarrolladas por otros usuarios y aportar tu propia contribución a la comunidad, debido a esto, cuenta con una amplia gama de paquetes que están en continuo desarrollo.

5.1.7 Funcionamiento

El grafo de tiempo de ejecución de ROS es una red de procesos "$peer - to - peer$" que se acoplan libremente utilizando la infraestructura de comunicación de ROS® . Se implementan varios tipos de comunicación, incluyendo la comunicación síncrona de estilo RPC sobre servicios, la transmisión asíncrona sobre topics y el almacenamiento de datos en un servidor de parámetros.

A continuación se detallan los conceptos más importantes ([3]) y que son de gran interés para este trabajo:

- Packages: son la unidad principal para organizar el software en ROS® . Los paquetes pueden contener proceso, bibliotecas, conjuntos de datos, archivos de configuración, etcétera. son el elemento de construcción más atómico, lo que significa que lo más granular que puede construir y lanzar es un paquete.

- Nodos: son los procesos que se encargan de la parte computacional del proyecto. Un sistema de control de ROS® se compone por varios nodos trabajando conjuntamente, así se consigue una gran modularidad , ya que por ejemplo, en el caso de un quadrotor, un nodo se ocuparía de controlar la velocidad de los motores, otro se encargaría de actualizar los modos de vuelo,nodos encargados de la medida de los sensores, nodos encargados del control del quadrotor dependiendo del modo seleccionado...

- Master: Este nodo proporciona servicio de nombres y registro al resto de nodos de la red de ROS® . El papel del Master es permitir que los nodos individuales se ubiquen entres sí, una vez que estos se han localizado, pueden comenzar la comunicación. Sin este nodo, los demás nodos no podrían encontrarse.

- Mensajes: los nodos se comunican entre sí a través de mensajes, que son simplemente estructuras de datos. Pueden incluir estructuras anidadas, al igual que en lenguaje C.

- Topics: los mensajes se enrutan a través de un sistema de publicación/suscripción. Un topic es un nombre que se utiliza para identificar el contenido del mensaje. Los nodos envían mensajes para publicar un tema determinado. Por otro lado, si un nodo está interesado en un tipo de datos, se puede puede suscribir al topic correspondiente para obtener la información, además puede suscribirse a más de uno.
  Puede haber múltiples editores y suscriptores de un mismo topic, y además un nodo puede tanto publicar como suscribirse a múltiples topics. En general, los editores y suscriptores no son conscientes de la existencia de los demás.

- Servicios: Se basan en el método de petición/respuesta, de forma que se definen dos estructuras de mensajes, una para solicitudes y otra para respuestas. De esta forma un nodo ofrece un servicio bajo un determinado nombre y el cliente utiliza el servicio enviando un mensaje de solicitud y esperando la respuesta.

  El nodo **Master** actúa como un servicio de nombres, almacena información de registro de **topics** y **servicios**.Cuando los nodos se comunican con el **Master**, pueden acceder a información sobre los nodos registrados y establecer comunicaciones. Los nodos se conectan directamente entre sí, el **Master** solo proporciona información de búsqueda, como un servidor DNS. Los nodos publican **topics** sin importar si alguien está esperando esa información, una vez publicado, los nodos interesados pueden suscribirse a dicho **topic** para obtener la información que necesitan.

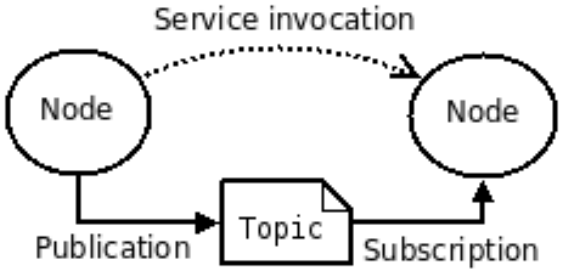


Figura 5.7 Representación simplificada de la comunicación entre nodos de ROS.

Esta arquitectura permite la operación desacoplada, donde los nombres son los parámetros necesarios

para construir sistemas más grandes y complejos.

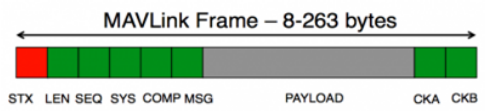
5.1.8 MAVROS

MAVROS es un paquete de ROS® compatible con varios autopilotos y que proporciona los drivers necesarios para utilizar el protocolo de comunicaciones **MAVLink**, lo que nos permite comunicar ROS® con el piloto automático. La gran utilidad de este paquete es que podremos crear nodos que se suscriban y publiquen a los topics proporcionados por **MAVROS**, y este, gracias al protocolo **MAVLink**, transmitirá los mensajes al autopiloto.[4]

5.1.9 MAVLink

MAVLink es el protocolo de comunicaciones en serie más utlizado para enviar datos y comandos entre vehículos y estaciones terrestres. El protocolo define un gran conjunto de mensajes que se pueden enviar a través de casi cualquier conexión serie. No se garantiza que los mensajes se entreguen, lo que significa que las estaciones o computadoras asociadas deben verificar el estado del vehículo para verificar si se ha ejecutado el comando, tiene como prioridad la velocidad de transmisión y la seguridad.

Los mensajes no tienen más de 263 bytes:



Byte Index	Content	Value	Explanation
0	Packet start sign	v1.0: 0xFE (v0.9: 0x55)	Indicates the start of a new packet.
1	Payload length	0 - 255	Indicates length of the following payload.
2	Packet sequence	0 - 255	Each component counts up his send sequence. Allows to detect packet loss
3	System ID	1 - 255	ID of the SENDING system. Allows to differentiate different MAVs on the same network.
4	Component ID	0 - 255	ID of the SENDING component. Allows to differentiate different components of the same system, e.g. the IMU and the autopilot.
5	Message ID	0 - 255	ID of the message - the id defines what the payload "means" and how it should be correctly decoded.
6 to (n+6)	Data	(0 - 255) bytes	Data of the message, depends on the message id.
(n+7) to (n+8)	Checksum (low byte, high byte)	ITU X.25/SAE AS-4 hash, **excluding packet start sign, so bytes 1..(n+6)** Note: The checksum also includes MAVLINK_CRC_EXTRA (Number computed from message fields. Protects the packet from decoding a different version of the same packet but with different variables).	

Figura 5.8 Formato mensaje.

El remitente rellena los campos para que el receptor sepa el origen del mensaje. Existe una identificación única para cada vehículo o estación terrestre. Las estaciones terrestres suelen usar una identificación alta como ”255” y los vehículos usan por defecto ”1”.[4]

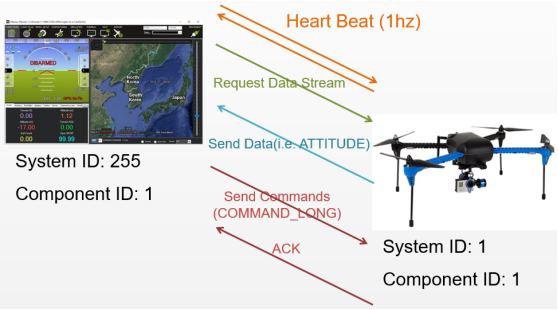


Figura 5.9 Flujo de mensajes de alto nivel.

5.2 Preparación del entorno

En esta sección se recoge paso a paso la instalación del software necesario y la preparación de la máquina para poder lanzar la simulación en GAZEBO® . Considero que es importante añadir esta información debido a que ya no está disponible en la red, además, me he encontrado problemas a la hora de la instalación del entorno, y esta información podrá ser de ayuda para futuros lectores.

5.2.1 Sistema Operativo

El propio fabricante **Erle Robotics** que han desarrollado el entorno que vamos a montar, recomiendan que se instale sobre una máquina con **Ubuntu 14.04**[6]. El entorno ha sido probado por otros miembros de la escuela en este SO y funciona perfectamente.
Una vez instalada la versión que se indica de SO, es muy importante no actualizar a la versión 16 de Ubuntu, ya que pueden surgir nuevos problemas y la simulación puede no funcionar como se espera.

5.2.2 Instalación de paquetes básicos y MAVProxy

En primer lugar se instalan los paquetes base(gawk,make,git,curl,cmake) para la configuración de la simulación.

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

A continuación se instalan las dependencias de MAVProxy:

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

Ahora ya podemos instalar MAVProxy:

```
sudo pip2 install pymavlink catkin_pkg --upgrade
sudo pip install MAVProxy==1.5.2
```

Por último debemos instalar **ArUco**, que se trata de una librería de Realidad Aumentada de código abierto escrita en C++, desarrollada por el grupo A.V.A.[10].
Primero nos descargamos los archivos necesarios desde :
https://sourceforge.net/projects/aruco/files/1.3.0/aruco-1.3.0.tgz/download
Y por último descomprimimos e instalamos:

```
cd ~/Descargas
tar -xvzf aruco-1.3.0.tgz
cd aruco-1.3.0/
mkdir build && cd build
cmake ..
make
sudo make install
```

5.2.3 Código ArduPilot

Descargamos el código del autopiloto desde el repositorio de Erle Robotics en GitHub. Hay que tener en cuenta, que debemos clonar el código desde la rama **gazebo**, ya que es la versión compatible con la simulación. Creamos un nuevo directorio y almacenamos ahí los componentes de nuestra simulación.

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

5.2.4 JSBSim

Es un modelo de dinámica de vuelo de código abierto (FDM) que puede ser compilado y ejecutado en múltiples sistemas operativos[5]. Es esencialmente, el modelo físico que define el vehículo de la simulación. No tiene gráficos, pero puede ser incorporado a simuladores más completo, como GAZEBO® .

Lo descargamos e instalamos, así como algunas dependencias necesarias:

```
cd ~/simulation
git clone git://github.com/tridge/jsbsim.git
# Additional dependencies required
sudo apt-get install libtool automake autoconf libexpat1-dev
cd jsbsim
./autogen.sh --enable-libraries
make -j2
sudo make install
```

5.2.5 ROS Indigo

Antes de instalar la versión de ROS® que necesitamos, debemos preparar la máquina para aceptar el software de packages.ros.org y configurar as claves:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc)
    main" > /etc/apt/sources.list.d/ros-latest.list'
sudo apt-key adv --keyserver hkp://pool.sks-keyservers.net --recv-key 0xB01FA
    116
```

```
sudo apt-get update
```

Ahora ya podemos instalar ROS® , se instala la versión base, ya que con esta es suficiente.

```
sudo apt-get install ros-indigo-ros-base
```

Antes de poder usar ROS® en la máquina, debemos inicializar **rosdep**, herramienta de línea de comandos que permite instalar dependencias y es necesario para poder ejecutar componentes de ROS® .

```
sudo rosdep init
rosdep update

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

Por último instalamos dependencias adicionales que serán necesarias:

```
sudo apt-get  install python-rosinstall \
       ros-indigo-octomap-msgs \
       ros-indigo-joy         \
       ros-indigo-geodesy     \
       ros-indigo-octomap-ros \
       ros-indigo-mavlink     \
       ros-indigo-control-toolbox \
       unzip
```

Crear espacio de trabajo de ROS (ROS workspace)

Necesitamos un directorio donde almacenar todos los componentes de ROS® , donde instalar y compilar packages. Creamos un directorio llamado *ros_catkin_ws*:

```
mkdir -p ~/simulation/ros_catkin_ws/src

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

5.2.6 Instalación de Gazebo

Primero configuramos la máquina para aceptar el software de *packages.orsfundation.org* y configuramos las claves.

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

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

Ahora borramos las versiones instaladas de GAZEBO® y sus dependencias (en este casi se instaló la versión 2 de GAZEBO® por defecto al instalar ROS®) e instalamos GAZEBO® 7.

```
sudo apt-get update
sudo apt-get remove .*gazebo.* '.*sdformat.*' '.*ignition-math.*' && sudo apt
    -get update && sudo apt-get install gazebo7 libgazebo7-dev drcsim7
```

Una vez tenemos todo instalado, es hora de compilar todo junto en el espacio de trabajo que hemos creado. En esta compilación me he encontrado con un par de problemas que se recogen en Apéndice B

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

Y para finalizar descargamos algunos modelos proporcionados por Erle Robotics para GAZEBO® .

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

5.3 Entorno de simulación montado

5.3.1 Características

La versión instalada de GAZEBO® es compatible con la versión de ROS® Indigo, dicha versión es la utilizada por el quadrotor real. Disponemos de MAVROS, lo que nos permite interactuar con la versión virtual de la misma forma que se haría con el quadrotor real. Para este tipo de comunicaciones se utiliza el protocolo MAVLink, permitiéndonos acceder a todos los parámetros del quadrotor simulado.

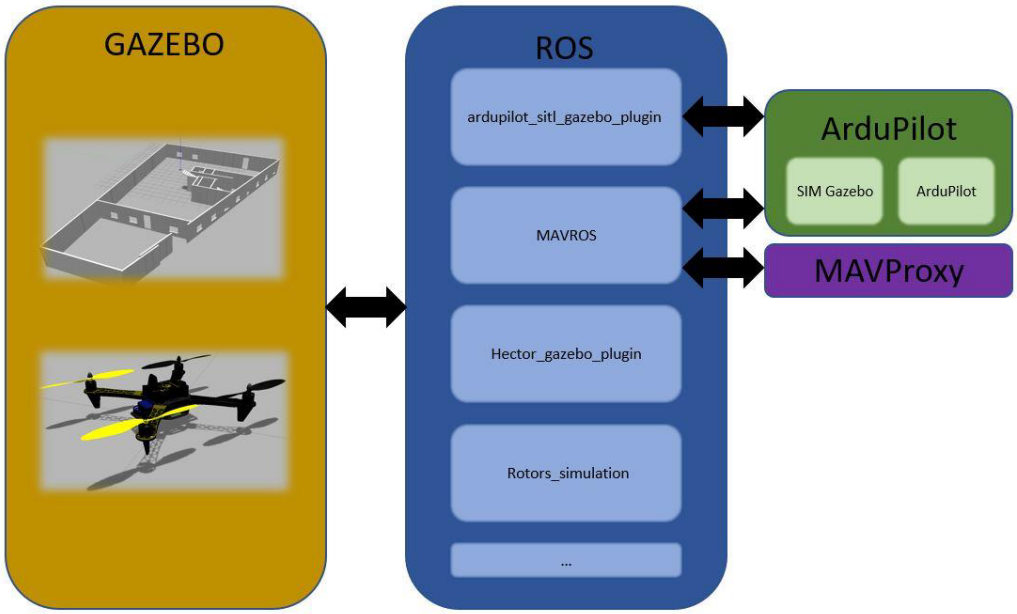


Figura 5.10 Esquema del entorno montado.

Como ya hemos ido describiendo a lo largo de este capítulo, la simulación está compuesta por varios elementos cuya acción conjunta nos proporciona sistema muy fiel a la realidad. El autopiloto (5.1.3) es el que se encarga de todos los cálculos para obtener las actuaciones que controlan la actitud del quadrotor, las señales PWM que se envían a los ESC simulados. Estas señales son utilizas por el simulador de dinámica JSBSim(5.2.4) para calcular el comportamiento del quadrotor y finalmente utilizando la información generada por JSBSim podemos ver una representación gráfica del vuelo del quadrotor gracias al simulador GAZEBO® .

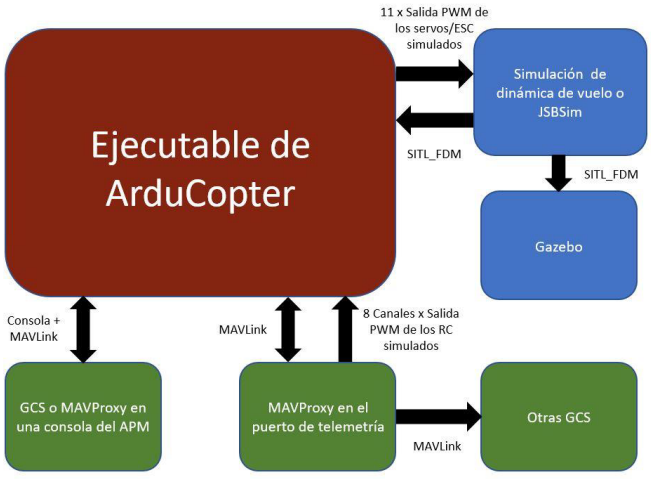


Figura 5.11 Esquema de la simulación.

En el esquema anterior podemos ver como gracias al protocolo MAVLink podemos controlar la simulación en tiempo real desde la consola de MAVProxy que actúa como estación de tierra. Podemos enviar comandos para cambiar el modo de vuelo, cambiar valores de parámetros, conocer el estado del quadrotor, etcétera, además nos permite acceder a 8 canales de entrada, que nos permiten controlar el vuelo del quadrotor mediante las acciones del piloto, utilizando una emisora en un modo manual.

5.3.2 Lanzar la simulación

Primero ejecutamos *MAVProxy* y *ArduCopter* sobre *SITL*. Abrimos una nueva terminal en Ubuntu y lanzamos los siguientes comandos:

```
source ~/simulation/ros_catkin_ws/devel/setup.bash
cd ~/simulation/ardupilot/ArduCopter
../Tools/autotest/sim_vehicle.sh -j 4 -f Gazebo
```

donde *sim_vehicle.sh* es el script que se encarga de ejecutar *SITL*. Este script tiene una serie de parámetros de entrada que permiten definir la simulación, por ejemplo, en este caso se ha lanzado con el frame configurado para GAZEBO® .

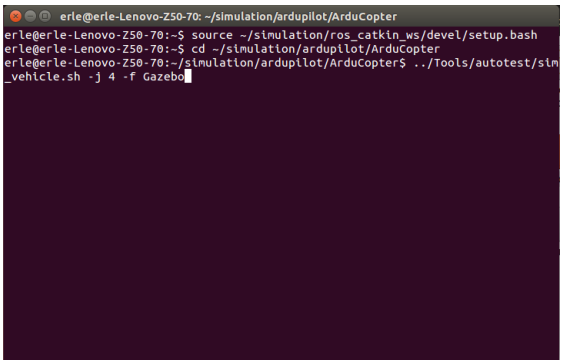


Figura 5.12 Terminal 1 comandos.

Necesitamos un nuevo terminal, donde ejecutar el plugin *ardupilot_sitl_gazebo_plugin*, y es el encargado de proporcionarnos la comunicación entre ROS® y GAZEBO® con ArduCopter.

```
source ~/simulation/ros_catkin_ws/devel/setup.bash
roslaunch ardupilot_sitl_gazebo_plugin erlecopter_spawn.launch
```

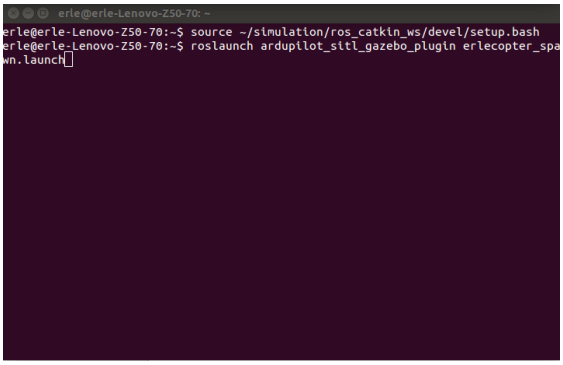


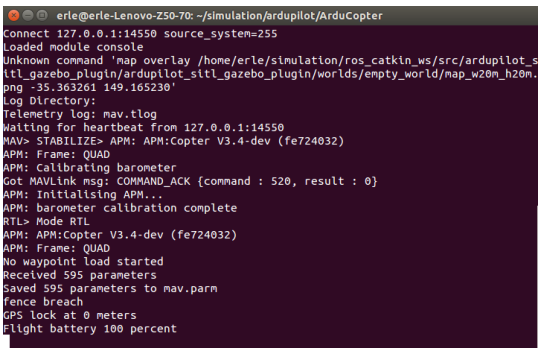
Figura 5.13 Terminal 2 comandos.

Debemos esperar hasta que el mundo cargue completamente, es decir, hasta que en la terminal MAVProxy aparezca el mensaje *Fligth battery* 100 *percent*, y una vez se ha cargado la simulación,en esta misma terminal, podemos importar los parámetros proporcionados por *Erle Robotics* para la simulación.

```
param load /home/erle/simulation/ardupilot/Tools/Frame_params/Erle-Copter.
    param
```

Para este proyecto el directorio principal es *home*/*erle*/, habría que sustituirlo por el que corresponda en la máquina que se ejcute.

Tras lanzar la simulación disponemos de 4 ventanas importantes. Disponemos de dos terminales Ubuntu, la primera que utilizamos es la línea de comandos MAVProxy con la que podemos interactuar con el quadrotor, y en la segunda podemos ir viendo los mensajes del controlador de vuelo.



(a) Línea de comandos MAVPROXY.

(b) Terminal mavros.

Figura 5.14 Terminales Ubuntu.

En la segunda terminal, podemos ver información interesante como por ejemplo los nodos de ROS® de los que disponemos al iniciar la simulación, además de la IP del nodo *Master*, que será la máquina donde se ejecuta la simulación.

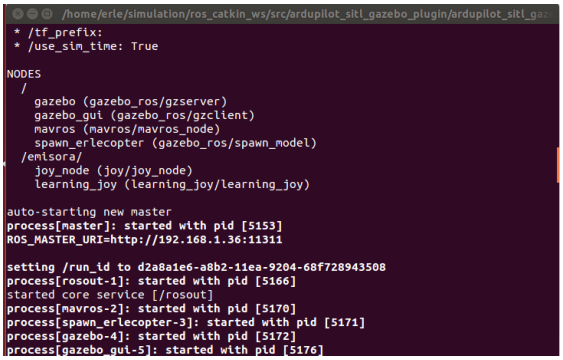


Figura 5.15 Nodos de ROS.

Tenemos la interfaz de GAZEBO® donde podemos ver el quadrotor simulado en tiempo real.

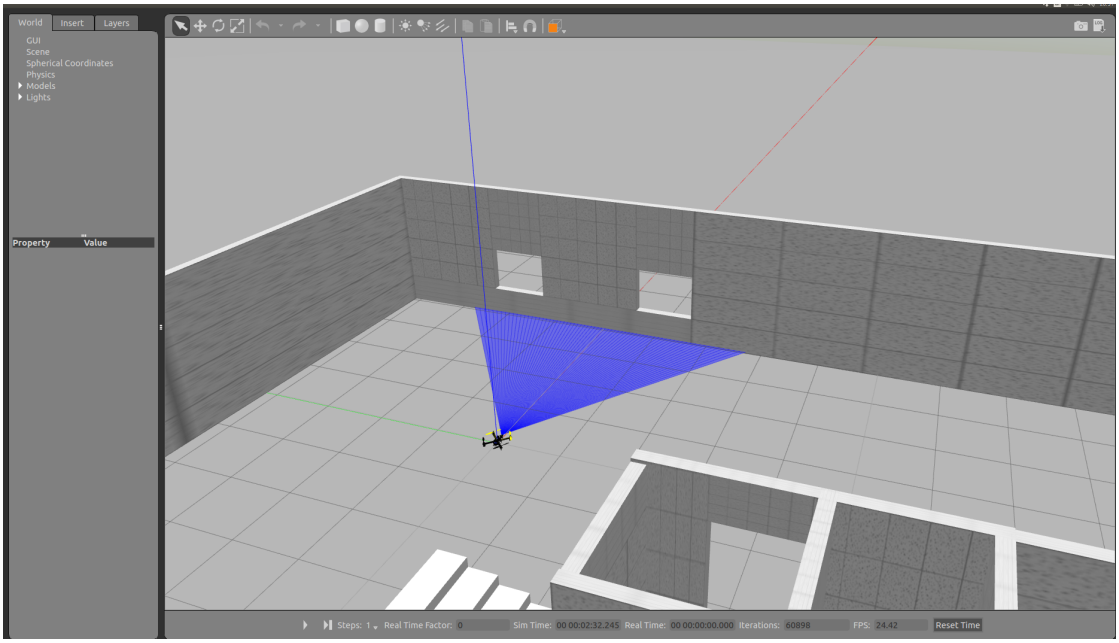


Figura 5.16 Interfaz de Gazebo.

Y por último, tenemos la consola de Ardupilot, que puede ser de utilidad a la hora de depurar código y probar nuevos modos de vuelo.

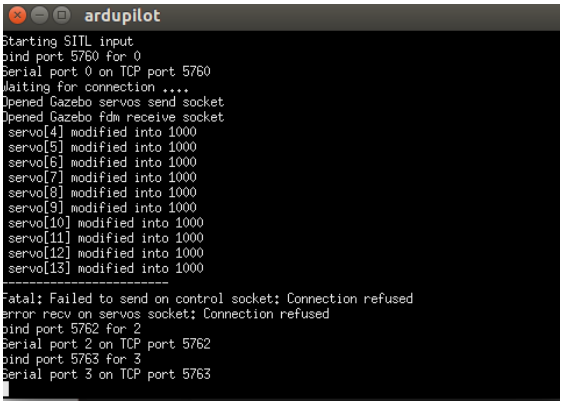


Figura 5.17 Consola ardupilot.

Una vez hemos lanzado la simulación satisfactoriamente, podemos probar el funcionamiento del quadrotor, ya que como se ha comentado anteriormente esta simulación tiene una serie de parámetros predefinidos que permiten trabajar con ella sin necesidad de hacer ajustes adicionales. Para testar que todo funciona correctamente y familiarizarnos con el entorno, disponemos de una serie de tutoriales proporcionados por *Erle Robotics*[9], los cuáles se han seguido durante el desarrollo de este trabajo y se recomienda realizar.

6 Implementación del modo de vuelo diseñado en ArduCopter

En este capítulo se va a implementar el modo de vuelo que hemos diseñado en MATLAB® en el código fuente de *Ardupilot* con el fin de probar su funcionamiento en la simulación que hemos montado en GAZEBO® . Debido a su característica de código abierto, es un código fuente muy extenso, con una gran cantidad de modos de vuelo, librerias, configuraciones, etcétera, y que cuenta con una gran comunidad. Se analizará la estructura del código principal y el funcionamiento del mismo, centrándonos solo en lo necesario para incorporar nuestros propios diseños.
El principal objetivo de este capítulo es que sirva de guía para futuros trabajos, de forma que conozcan lo necesario e imprescindible para implementar nuevos controles y modos de vuelo, evitando perder el tiempo en analizar código irrelevante.

6.1 Visión general del código

El código fuente de ArduPilot es muy extenso (alrededor de 700k líneas), lo que como ya comentaba, puede suponer que el usuario pierda muchas horas analizando líneas de código con el fin de entender su estructura y la forma de implementar sus propias funciones. El código se encuentra escrito en lenguaje **C++**.

La estructura básica de ArduPilot se divide en 5 partes principales:

- Código del vehículo
- Librerías compartidas
- Capa de abstracción de hardware(AP_HAL)
- Directorio de herramientas
- Código de soporte externo (mavlink)

6.1.1 Código del vehículo

Actualmente ArduPilot contempla 5 tipos de vehículos: Plane, Copter,Sub y AntennaTracker, el firmware para cada uno de estos vehículos se divide en 5 directorios de alto nivel. En este trabajo nos centraremos en el tipo *Copter* que es el que se utiliza para quadrotors.

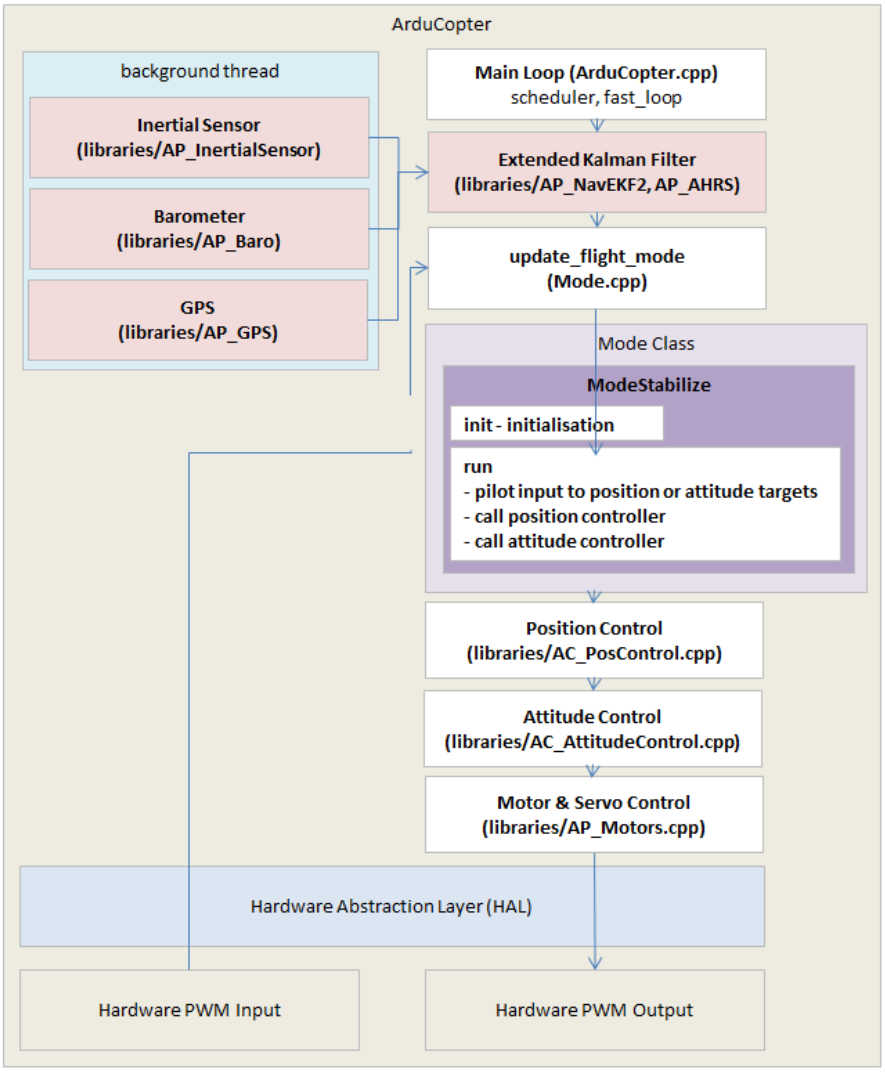


Figura 6.1 Arquitectura de ArduCopter a alto nivel.

En la Figura 6.1 vemos la estructura general del código, con la que podemos hacernos una idea del flujo del mismo, pero vamos a entrar un poco más en detalle y vamos a ver como se realiza el control del quadrotor en ArduCopter dependiendo del modo de vuelo escogido.

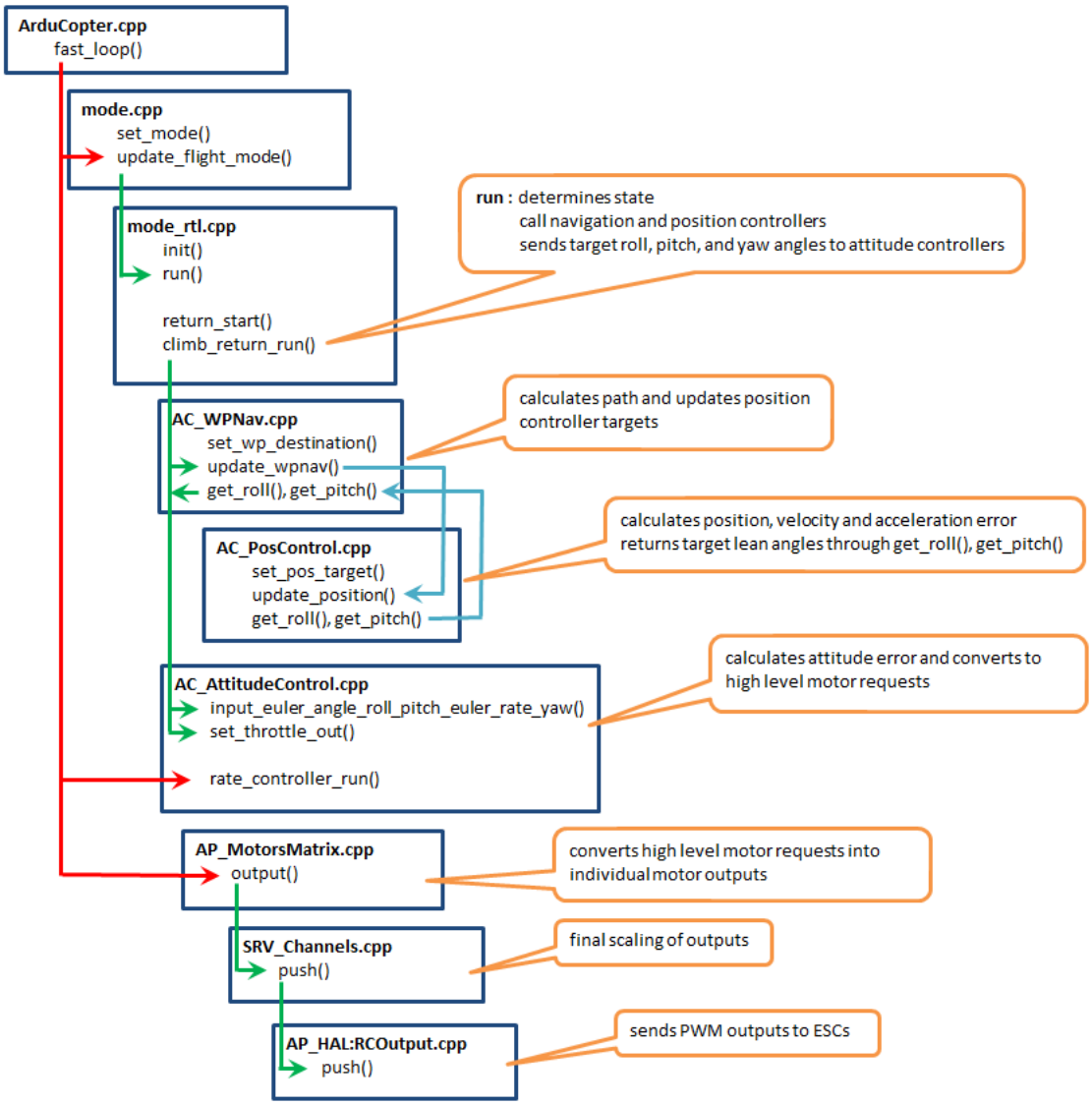


Figura 6.2 Esquema de modos de vuelo autónomos.

Funcionamiento de un Modo de vuelo de alto nivel

El archivo principal es **ArduCopter.cpp**, es el archivo de más alto nivel. Este archivo tiene una estructura de tipo **setup**()/**loop**() heredada de arduino, pero ArduPilot no se trata de un sistema de un solo subproceso. Dentro de este archivo tamién encontramos el **scheduler** del sistema, donde se recogen las distintas funciones del sistema, la prioridad y la frecuencia con la que estas deben ser llamadas.

ArduCopter espera hasta que llega una nueva medida de la **IMU**, cuando la recibe, se produce una llamada a la función **fast_loop**() que se ejecuta cada 400*hz* y es el bucle principal del sistema.

ArduCopter está pendiente de si el piloto realiza un cambio en el modo de vuelo, de ser así, actualiza la estrategia de control cambiado de modo mediante la función **set_mode**(). Dentro de **fast_loop**() se realiza la llamada a la función **update_flight_mode**(), que se encarga de ejecutar la estrategia de control asociada al modo de vuelo actual.

Los modos de vuelo se componen de dos funciones principales, la función **init**(), que se ejecuta solamente una vez, y es donde se inicializa el modo de vuelo. Por otro lado, la función **run**(),se ejecuta mientras el quadrotor se mantenga en ese modo de vuelo. Normalmente en la función **init**() se inicializan las variables y se incian subprocesos en caso de que fuesen necesarios, mientras que en la función **run**() es donde se produce el control del quadrotor, es decir, es donde se realizan las llamadas a los controles de más bajo nivel y se realizan las realimentaciones con las medidas de los sensores.

Por último se convierten las actuaciones obtenidas de los controles de más bajo nivel en señales *PWM* y son enviadas a los *ESCs* que controlan los motores.

Esta es la estructra general que siguen los modos de vuelo implementados en *ArduCopter*, pero puede ser modificada y adaptada a las necesidades del usuario.

6.1.2 HAL

La capa *AP_HAL* es lo que hace que *ArduPilot* sea compatible con una gran variedad de plataformas. Disponemos de un *AP_HAL* dentro del proyecto, concretamente en el directorio *libraries/AP_HAL* que define las funciones necesarias para trabajar con distintas placas. Esta interfaz nos permite trabajar con sensores, motores y ESCs sin tener que preocuparnos por el código de bajo nivel y los constroladores específicos para este hardware.

6.2 Creación de un nuevo Modo de Vuelo

Vamos a añadir un nuevo modo de vuelo de alto nivel a ArduCopter donde vamos a implementar nuestra estrategia de control diseñada en el Capítulo 3.

A continuación se detallan las modificaciones que habría que realizar en el código para insertar nuestro propio modo de vuelo:

- Debemos definir el nuevo modo en el archivo **defines.h**,en este caso el modo *ERLE*:

Código 6.1 defines.h.

```
// Auto Pilot Modes enumeration
enum autopilot_modes {
    STABILIZE =     0,  // manual airframe angle with manual throttle
    ACRO =          1,  // manual body-frame angular rate with manual throttle
    ALT_HOLD =      2,  // manual airframe angle with automatic throttle
    AUTO =          3,  // fully automatic waypoint control using mission commands
    GUIDED =        4,  // fully automatic fly to coordinate or fly at velocity / direction using GCS immediate commands
    LOITER =        5,  // automatic horizontal acceleration with automatic throttle
    RTL =           6,  // automatic return to launching point
    CIRCLE =        7,  // automatic circular flight with automatic throttle
    LAND =          9,  // automatic landing with horizontal position control
    OF_LOITER =     10, // deprecated
```

```
    DRIFT =         11,   // semi-automous position, yaw and throttle control
    SPORT =         13,   // manual earth-frame angular rate control with manual throttle
    FLIP =          14,   // automatically flip the vehicle on the roll axis
    AUTOTUNE =      15,   // automatically tune the vehicle's roll and pitch gains
    POSHOLD =       16,   // automatic position hold with manual override, with automatic throttle
    BRAKE =         17,    // full -brake using inertial /GPS system, no pilot input
    ERLE =        18
};
```

- Crear un fichero llamado **control_** < **nombre_modo** > **.cpp** basado en uno de los modos ya creados, como por ejemplo, tomando como referencia el fichero **control_stabilize.cpp** y nos quedamos solamente con las funciones y las librerias.

Código 6.2 control_erle.cpp.

```
#include "Copter.h"

bool Copter:: erle_init (bool ignore_checks)
{
    return true;
}
void Copter:: erle_run ()
{

}
```

- Hay que añadir la definición de las funciones en el archivo **Copter.h**:

Código 6.3 Copter.h.

```
    bool  erle_init (bool ignore_checks);
    void  erle_run ();
```

- Añadir en el fichero **flight_mode.cpp**, en la función **set_mode**(), un nuevo *case* para la función *init()* de nuestro modo:

Código 6.4 set_mode().

```
    case ERLE:
    success = erle_init (ignore_checks);
    break;
```

Además, añadir en la función **update_flight_mode**() otro *case* para llamar a la función *run()* de nuestro modo:

Código 6.5 update_flight_mode().

```
    case ERLE:
    success = erle_init (ignore_checks);
    break;
```

Y por último añadimos en la función **print_fligth_mode**() las líneas para que se imprima el nombre de nuestro modo en la terminal:

Código 6.6 print_fligth_mode().

```
case ERLE:
  port->print_P(PSTR("ERLE"));
  break;
```

- Para finalizar, hay que agregar el modo a la lista de valores del parámetro *FLTMODE1* en el fichero **Parameters.cpp**:

Código 6.7 set_mode().

```
// @Param: FLTMODE1
// @DisplayName: Flight Mode 1
// @Description: Flight mode when Channel 5 pwm is <= 1230
// @Values: 0: Stabilize ,1: Acro,2: AltHold,3: Auto,4: Guided,5: Loiter ,6: RTL,7: Circle ,9: Land,11: Drift ,13: Sport ,14: Flip ,15:
    AutoTune,16:PosHold,17:Brake,18:Erle
// @User: Standard
GSCALAR(flight_mode1, "FLTMODE1",          FLIGHT_MODE_1),
```

6.3 Dinámica del Motor

La estrategia de control que hemos diseñado en los capítulos anteriores, nos devuelve como señales de actuación las velocidades de los motores ω en rad/s, pero en el mundo real no controlamos directamente los motores, realizamos el control a través de los **ESCs**. Estos dispositivos reciben como entradas señales de tipo **PWM**, por lo tanto tenemos que hacer una conversión de velocidades a PWM antes de enviar las actuaciones. Para ello vamos a realizar un pequeño experimento, aprovechando el modo de vuelo que hemos creado, y de paso calcularemos un modelo matemático del motor de la simulación.

6.3.1 Función de trasnferencia

Con el objetivo de obtener la relación entre los valores PWM que se aplican a los ESCs y la velocidad resultante de los motores, el modelo matemático del motor se aproxima a un sistema de primer orden

$$G(s) = \frac{\omega(s)}{PWM(s)} = \frac{K_{DC}}{\tau \cdot s + 1} \tag{6.1}$$

$$Donde \begin{cases} \text{PWM} & = & \text{Senal PWM} \\ \omega & = & \text{Velocidad del motor } (\frac{rad}{s}) \\ \tau & = & \text{Constante de tiempo (s)} \\ K_{DC} & = & \text{Ganancia} \end{cases}$$

A continuación, mediante unos experimentos, se estimarán los parámetros desconocidos de la función 6.1, la ganancia K_{DC} y la constante de tiempo τ.

6.3.2 Experimento

Gracias al paquete MAVROS, y a la red de nodos ROS creada al lanzar la simulación, podemos acceder a una serie de Topics con información del quadrotor en tiempo real. Además, nos proporciona una serie de servicios que nos permiten interactuar con el quadrotor en tiempo real, como por ejemplo, cambiar el modo de vuelo, lo cuál será útil para esta sección.

Topics

Cada tema está fuertemente tipado por el tipo de mensaje ROS utilizado para publicar en él. Podemos ver todos los temas disponibles mediante el comando:

```
rostopic list
```

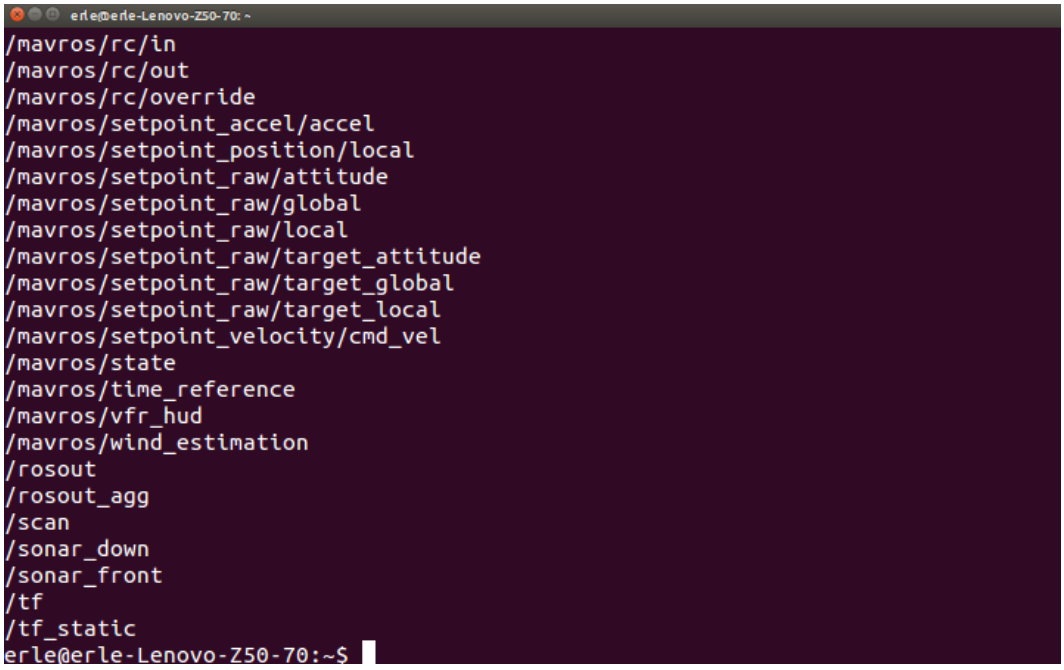


Figura 6.3 Lista de topics.

Disponemos de un Topic donde se publican las velocidades de los motores, podemos obtener la estructura del mensaje utilizado para publicar la información mediante el comando:

```
rostopic info /erlecopter/motorspeed
```

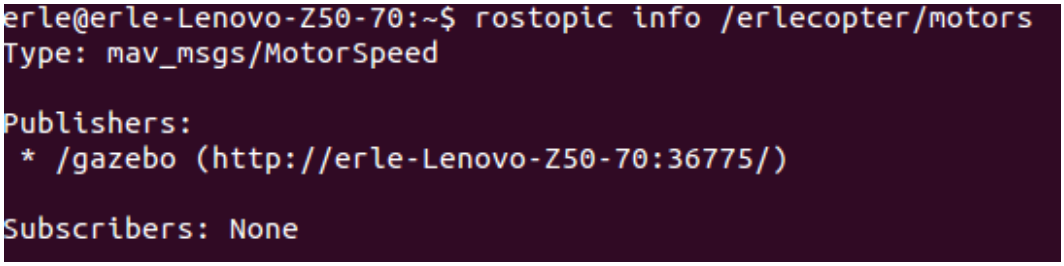


Figura 6.4 Lista de topics.

Podemos ver que utiliza un mensaje de tipo **mav_msgs**/**MotorSpeed**. La definición de este tipo de mensajes, la podemos encontrar en nuestro espacio de trabajo de ROS, concretamente en el archivo

```
simulation/ros_catkin_ws/src/mav_comm/mav_msgs/msg/MotorSpeed.msg
```

Código 6.8 MotorSpeed.msg.

```
Header header

float64[] motor_speed # motor speed [rad/s]
```

Como se puede ver en el documento, la información que publica es la velocidad del motor en $\frac{rad}{s}$.

Código ArduCopter

En el modo de vuelo que se ha creado para este proyecto en el código *ArduCopter*, vamos a implementar una función que se encargue de enviar una serie de valores PWM a los motores, y utilizando el Topic descrito en la sección anterior, podremos obtener la relación entre los valores PWM y las velocidades de los motores, además de estimar un modelo.

La función se ejecuta cada 100Hz, se usará un contador con un incremento de 0.01 para calcular el tiempo, y cada 2 segundos, se le irán mandando diferentes señales PWM a los motores.

Código 6.9 Experimento Motores.

```
 float tiempo = 0; // Contador

 bool Copter::erle_init(bool ignore_checks)
{

  hal.rcout->set_freq(0xF,490);

  // Se habilitan los canales de salida
  for(uint8_t i = 0; i < 8; i++)
  {
    hal.rcout->enable_ch(i);
  }

  tiempo = 0;

    return true;
}

void Copter::erle_run()
{
  if(tiempo <= 2)
  {
    hal.rcout->write(0, 1100);
    printf("PWM0 -> %d (m)\n",1100);
```

```
  }
  else if (tiempo <= 4)
  {
    hal.rcout->write(0, 1200);
    printf("PWM0 -> %d (m)\n",1200);
  }
  else if (tiempo <= 6)
  {
    hal.rcout->write(0, 1300);
    printf("PWM0 -> %d (m)\n",1300);
  }
  else if (tiempo <= 8)
  {
    hal.rcout->write(0, 1400);
    printf("PWM0 -> %d (m)\n",1400);
  }
  else if (tiempo <= 10)
  {
    hal.rcout->write(0, 1500);
    printf("PWM0 -> %d (m)\n",1500);
  }
  else if(tiempo <= 12)
  {
    hal.rcout->write(0, 1600);
    printf("PWM0 -> %d (m)\n",1600);
  }
  else if (tiempo <= 14)
  {
    hal.rcout->write(0, 1700);
    printf("PWM0 -> %d (m)\n",1700);
  }
  else if (tiempo <= 16)
  {
    hal.rcout->write(0, 1800);
    printf("PWM0 -> %d (m)\n",1800);
  }
  else
  {
    hal.rcout->write(0, 1100);
    printf("PWM0 -> %d (m)\n",1100);
  }

  tiempo = tiempo + 0.01;

    return;

}
```

Matlab + ROS

Matlab nos permite trabajar con ROS, pudiendo crear un suscriptor para almacenar la información publicada en los temas, además de crear editores y trabajar con los distintos servicios disponibles en la red de ROS. El proceso para conectar MATLAB® con ROS® y añadir diferentes tipos de mensajes, se recoge en el Apéndice E.2

Código Matlab

Vamos a crear un nodo en Matlab, que se suscriba al tema que publica las velocidades de los motores y almacenarlas para analizar la respuesta.

Primero damos de baja la conexión, para prevenir que no haya problemas con conexiones anteriores. Se establecen las IPs del nodo *Master* y de la máquina en la que vamos a ejecutar MATLAB® , como variables de entorno y se inicia la conexión. Se utiliza el servicio de cambio de vuelo, debemos primero crear un mensaje del tipo que usa este servicio y a continuación asignar el modo de vuelo (el modo 18), de forma que al lanzar el script, la simulación cambie al modo deseado, donde hemos incorporado el código descrito en el apartado **Código ArduCopter**

Código 6.10 Suscriptor Matlab.

```
clear all;
close all;
clc;

rosshutdown;
setenv('ROS_MASTER_URI','http://182.168.1.35:11311');
setenv('ROS_IP','192.168.1.41');
rosinit('http://192.168.1.35:11311');

global velocidades i tiempo
i = 1;
tiempo = zeros(5000,1);
velocidades = zeros(5000,4);
%% Clientes
flight_mode = rossvcclient('/mavros/set_mode');
arming = rossvcclient('/mavros/cmd/arming');


new_mode = rosmessage(flight_mode);
new_mode.BaseMode = 0;
new_mode.CustomMode = '18';
    result = call(flight_mode,new_mode);
    if result.Success
        disp(['NUEVO MODO DE VUELO --> ',new_mode.CustomMode]);
    else
        disp('NO SE HA PODIDO CAMBIAR EL MODO DE VUELO');
    end

    %% Suscribtor
W = rossubscriber('/erlecopter/motors',@lector_topics);
```

Como se puede ver, una vez consigue suscribirse al tema, cada vez que llegue un mensaje nuevo, se ejecutará lo que se recoge en la función *lector_topics*:

Código 6.11 Función lector_topics.

```
function lector_topics(src,msg)

global velocidades i tiempo
    if( i == 1)
        tic
    end
    tiempo(i,1) = toc;
        velocidades(i,1) = msg.MotorSpeed_(1,1);
        velocidades(i,2) = msg.MotorSpeed_(2,1);
        velocidades(i,3) = msg.MotorSpeed_(3,1);
        velocidades(i,4) = msg.MotorSpeed_(4,1);

        i = i +1;

        if( i >= 5000)
            rosshutdown;
        end
end
```

Una vez hemos recogido la cantidad de muestras deseadas, eliminamos el nodo para que no se siga ejecutando.

Resultados

Observamos la relación que existe entre las señales PWM y las velocidades de los motores en *rad/s*:

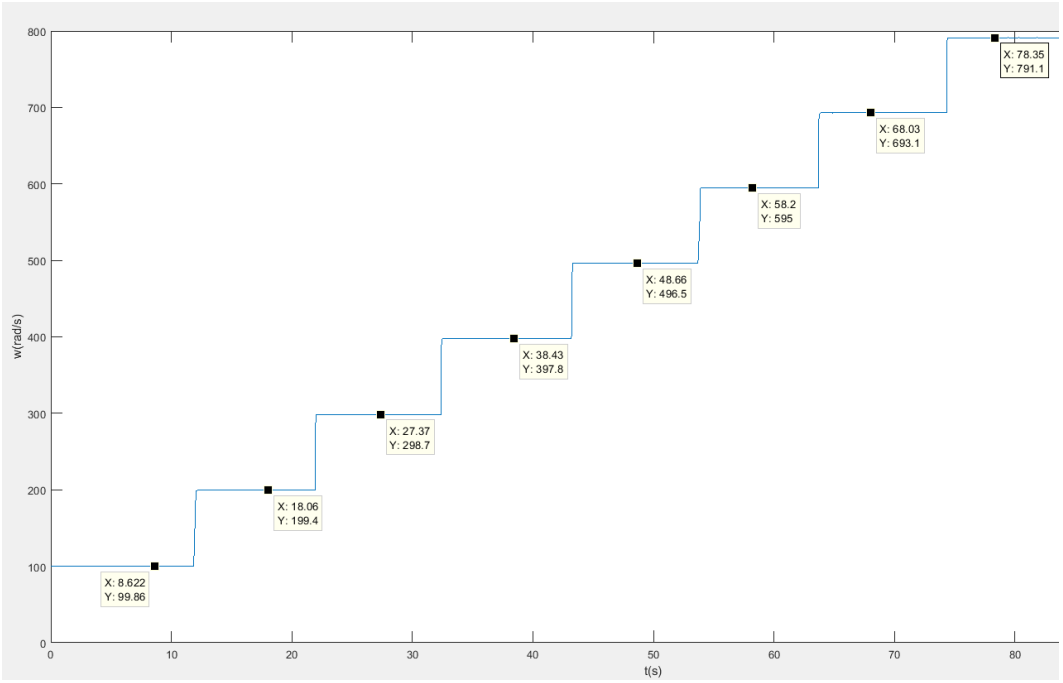


Figura 6.5 Respuesta del Motor.

Observando la gráfica, se puede estimar que la relación entre las señales PWM y las velocidades de los motores, es aproximadamente:

$$\omega \approx PWM - 1000 \tag{6.2}$$

Una vez ajustado este offset, nos centramos en las respuestas incrementales de 1200 y 1300 para intentar estimar un modelo:

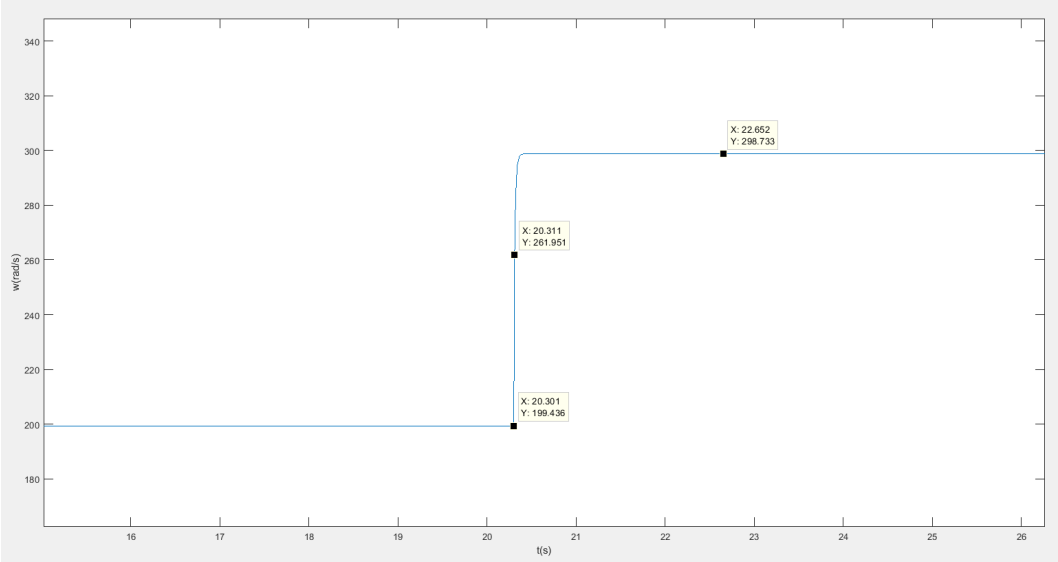


Figura 6.6 Estimación modelo.

Como resultado obtenemos una función de transferencia en tiempo continuo:

$$G(s) = \frac{w(s)}{PWM(s)} = \frac{0.993}{0.01 \cdot s + 1} \tag{6.3}$$

Para verificar la fiabilidad del modelo, realizamos un vuelo manual del quadrotor, obteniendo un archivo *log* con los datos de vuelo. Se convirtió este archivo a *.m* y se hizo un *script* en Matlab para comparar la respuesta del modelo con la del motor de la simulación:

Código 6.12 Experimento verificación.

```
clear all
close all
clc

% Prueba de subida %

m_100;

t_start = 1;
t_end = size(RCOU.data,1);

time = RCOU.data(:,1);
PWM = RCOU.data(:,3);
W = (RCOU.data(:,3)-1000);
RPM = W*(60/pi);

RPMinds = RPM >= 2500;
```

```
time = time(RPMinds);
PWM = PWM(RPMinds);
RPM = RPM(RPMinds);

%% Modelo

tau = 0.0186;
DC = 0.993;

s = tf('s');
G = DC/(tau*s+1)

W_sim = lsim(G,PWM-1200,t);

figure();
plot(time,W,'r');
hold on;
plot(time,W_sim+199.4,'k');
str = sprintf('Simulated vs Actual Motor Dynamics (DC = %.2f TC = %.2f)',DC,
    tau);
title(str);
xlabel('Time(s)');
ylabel('rad/s');
legend('Actual','Simulated');
```

Obtenemos una respuesta bastante parecida al motor de la simulación en *gazebo*:

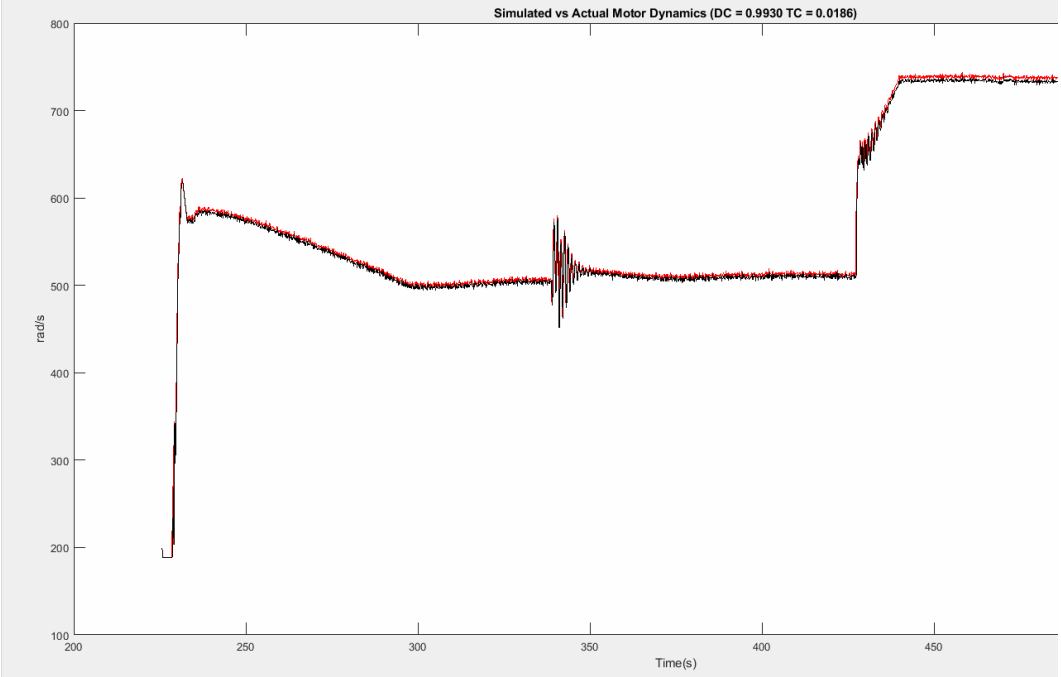


Figura 6.7 Verificación modelo.

6.4 Añadir mensajes al DataFlash

ArduPilot genera un archivo *.BIN* en la ruta */simulation/ArduCopter/logs/*,con los datos almacenados en la memoria *flash* después de cada vuelo, una vez que se han desarmado los motores. Este archivo contiene información muy útil sobre el vuelo del vehículo y el cuál podemos utilizar para realizar gráficas y análisis sobre el comportamiento del quadrotor durante su vuelo. Se va a mostrar como crear mensajes propios para almacenar en la memoria para que finalmente sean añadidos por el autopiloto al archivo generado.

Para ello solo vamos a hacer uso de dos ficheros, el primero *DataFlash.h*, donde se realizan las declaraciones y el segundo el fichero *LogFile.cpp*, donde se implementa la función que se encarga de escribir el mensaje en memoria.

En el archivo *DataFlash.h* se debe añadir el nombre del mensaje a la lista de mensajes ya existentes, que es un *enum* llamado **LogMessages**. En este casos se añaden dos mensajes nuevos, *LOG_ERLE_MSG* y *LOG_ERLE_POS_MSG*:

Código 6.13 DataFlash.h.

```
enum LogMessages {
LOG_FORMAT_MSG = 128,
LOG_PARAMETER_MSG,
LOG_GPS_MSG,
  .
  .
```

```
  .
LOG_ERLE_MSG,
LOG_ERLE_POS_MSG,
};
```

Se debe definir una estructura de tipo *PACKED*, donde se definen las variables que componen el mensaje y el tipo de las mismas.

Código 6.14 Estructura de los mensajes.

```
struct PACKED log_erle{
LOG_PACKET_HEADER;
uint64_t time_us;
float U1;
float U2;
float U3;
float U4;
float w0;
float w1;
float w2;
float w3;
float roll;
float pitch;
float yaw;
float p;
float q;
float r;
};

struct PACKED log_erle_pos{
LOG_PACKET_HEADER;
uint64_t time_us;
float X;
float Y;
float Z;
};
```

Una vez definidos el nombre y el contenido del mensaje, se debe definir el formato de los mensajes. En este caso se han añadido a la lista existente *LOG_EXTRA_STRUCTURES*.

Código 6.15 Formato de los mensajes.

```
/*Format characters in the format string for binary log messages
b   : int8_t
B   : uint8_t
h   : int16_t
H   : uint16_t
i   : int32_t
```

```
I   : uint32_t
f   : float
d   : double
n   : char[4]
N   : char[16]
Z   : char[64]
c   : int16_t * 100
C   : uint16_t * 100
e   : int32_t * 100
E   : uint32_t * 100
L   : int32_t latitude/longitude
M   : uint8_t flight mode
q   : int64_t
Q   : uint64_t
*/

{LOG_ERLE_MSG, sizeof(log_erle), \
"ERLE","Qfffffffffffff","TimeUS,U1,U2,U3,U4,w0,w1,w2,w3,roll,pitch,yaw,p,q,r
    "},\
{LOG_ERLE_POS_MSG, sizeof(log_erle_pos), \
"EPOS","Qfff","TimeUS,X,Y,Z"}
```

Con esto ya tenemos definidos los nuevos mensajes, con el formato y el contenido que van a almacenar. Ahora se debe definir en *DataFlas.h*, e implementar, la función que se encarga de escribir estos mensajes, para ello utilizamos el archivo *LogFile.cpp*. En este caso se les pasa a las funciones una clase como argumento de entrada, y con los miembros de esa clase, se asignan valores a las componentes del mensaje.

Código 6.16 Definición de funciones.

```
void Log_Write_ERLE(ErleFunciones &erle);
void Log_Write_ERLE_POS(ErleFunciones &erle);
```

Código 6.17 Implementación de funciones.

```
void DataFlash_Class::Log_Write_ERLE(ErleFunciones &erle)
{
struct log_erle pkt = {
LOG_PACKET_HEADER_INIT(LOG_ERLE_MSG),
time_us  : hal.scheduler->micros64(),
U1     : erle.U1,
U2     : erle.U2,
U3     : erle.U3,
U4     : erle.U4,
w0     : erle.w0,
w1     : erle.w1,
w2     : erle.w2,
```

```
w3     : erle.w3,
roll   : erle.roll,
pitch  : erle.pitch,
yaw    : erle.yaw,
p    : erle.p,
q    : erle.q,
r    : erle.r
};
WriteBlock(&pkt,sizeof(pkt));
}

void DataFlash_Class::Log_Write_ERLE_POS(ErleFunciones &erle)
{
struct log_erle_pos pkt = {
LOG_PACKET_HEADER_INIT(LOG_ERLE_POS_MSG),
time_us  : hal.scheduler->micros64(),
X    : erle.X,
Y    : erle.Y,
Z    : erle.Z
};
WriteBlock(&pkt,sizeof(pkt));
}
```

Una vez implementado todo, simplemente tenemos que hacer la llamada a las funciones de escritura en nuestro código para que los mensajes sean escritos en el archivo generado por ArduPilot.

6.5 Implementación de la estrategia de Control

En esta sección vamos a añadir al modo de vuelo que hemos creado, la estrategia diseñada a lo largo del Capítulo 3. Vamos a seguir la misma estructura que para la simulación realizada en MATLAB® en Capítulo 4, por lo tanto comenzamos por la creación de una nueva clase para almacenar todos los parámetros necesarios de nuestro código, así como variables de comunicación entre funciones.

Esta clase recibe el nombre de **erlevariables** y es equivalente a la función **erlevariables**.**m** que se utilizó en la simulación de MATLAB® . Se implementa en un archivo llamado **erlevariables**.**h**, y puesto que solamente contiene definiciones e inicializaciones de varialbes no hay mucho más que añadir. Puede consultarse en Apéndice C.

A continuación vamos a definir todas las funciones necesarias de nuestro bucle de control. Para ello se crea una nueva clase llamada **ErleFunciones** que tiene como base, la clase **erlevariables** anteriormente creada. De esta forma, con una misma clase tenemos acceso tanto a funciones como a todas las variables necesarias.

Código 6.18 ErleFunciones.h.

```
#ifndef ERLEFUNCIONES_H_
#define ERLEFUNCIONES_H_

#include " erlevariables .h"
#include <AP_HAL/utility/Socket.h>
#include <AP_InertialNav/AP_InertialNav.h>     // Inertial Navigation library
```

```
#define NB_GAZEBO_SERVOS 16

class ErleFunciones : public erlevariables
{

public:

  ErleFunciones(const AP_AHRS_DCM& ahrs, const AP_InertialSensor& ins,const AP_Baro& barometer);
  void Altitude_PID_Hold();

  void Stabilize_PID_Roll();
  void Stabilize_PID_Pitch();
  void Stabilize_PID_Yaw();

  void Rate_PID_Roll();
  void Rate_PID_Pitch();
  void Rate_PID_Yaw();

  void Saturacion_Actuaciones();

  void Ecuaciones_Dinamicas();

    void Position_Control();
  void Correct_Yaw();

};

#endif /* ERLEFUNCIONES_H_ */
```

Se ha optado por crear una función para cada controlador,exceptuando para el control de posición que se han implementado ambos controladores juntos. Además contamos con una estructura similar a la simulación de MATLAB® , tenemos la función **Saturacion_Actuaciones**(), que como su propio nombre indica, se realiza la saturación de las velocidades de los motores y la función **Ecuaciones_Dinamicas**(), donde se implementan las ecuaciones que modelan el quadrotor. Cabe destacar que las funciones son de tipo **void** ya que la interacción entre funciones se realiza a través de las variables propias de la clase, es decir, las variables que se definen en la clase base **erlevariables**.

El cuerpo de todas estas funciones se implementa en un archivo *.cpp* denominado **ErleFunciones.cpp**. El cuerpo de estas funciones es muy similar a las funciones creadas para la simulación en MATLAB® , considero que no es necesario analizarlo, pudiendo observar el código en el Apéndice D.

Ahora es el momento de completar el modo de vuelo que hemos creado en la Sección 6.2, combinando las nuevas funciones de bajo nivel que hemos implementado, creando así el bucle de control de nuestro quadrotor. El modo de vuelo de alto nivel se implementa en el archivo **control_erle.cpp**.

6.5.1 Implementación con sensores

Lectura de la IMU y AHRS

Disponemos de dos sensores que nos permiten obtener la orientación del quadrotor, contamos con una unidad *IMU* y una unidad *AHRS*. La unidad *IMU* nos proporciona medidas de la velocidad angular alrededor de cada eje (medida por los giroscopios) y medidas de la aceleración lineal del quadrotor (medidas por los acelerómetros). Por otro lado, la unidad *AHRS*(Attitude and Jeading Reference System) es una *IMU* que además incorpora un filtro de *KALMAN*, que a partir de la fusión de sensores, nos proporciona como salidas, directamente los ángulos de inclinación ϕ,θ y ψ.

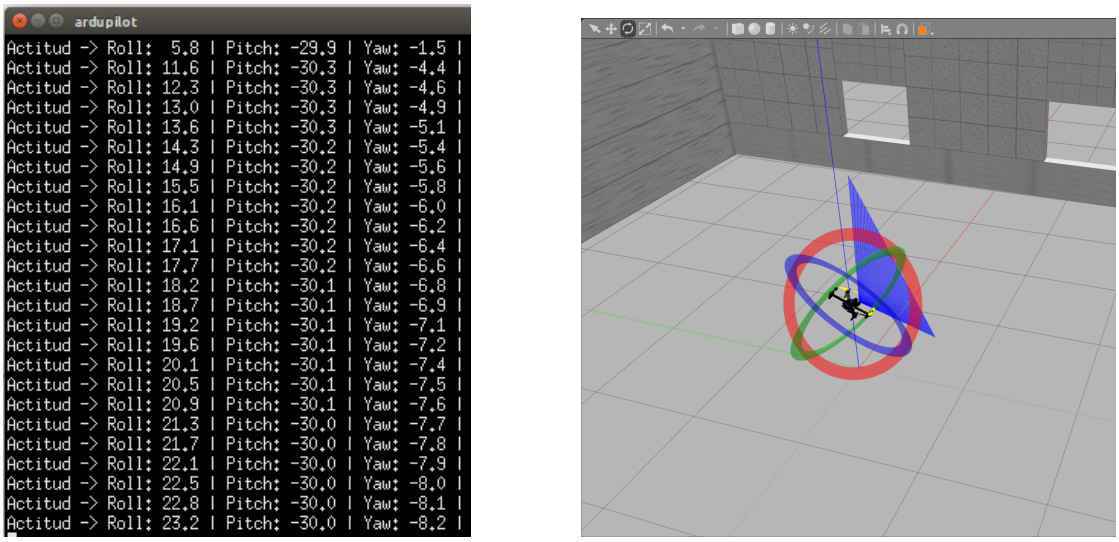
Primero en la función *erle_init*() inicializamos los sensores, en la función *erle_run*() esperamos a que se reciba una nueva medida de la *IMU*, con la función *wait_for_sample*(). Actualizamos la *IMU* y obtenemos las velocidades angulares del quadrotor, seguidamente actualizamos la unidad *AHRS* y

obtenemos la actitud del quadrotor. Por último imprimimos los resultados para ver que los sensores miden los cambios de actitud del quadrotor, para ello utilizamos la consola del autopiloto, pero en lugar de imprimir a la tasa de ejecución, imprimimos con una frecuencia de $1Hz$ para que nos sea más cómodo de visualizar.

Código 6.19 Prueba de lectura de IMU y AHRS.

```
bool Copter:: erle_init (bool ignore_checks)
{
hal.rcout->set_freq(0xF,100);

ins. init (ins.Sample_rate::RATE_100HZ);
ins. init_gyro ();

barometer. init ();

ahrs_erle . init ();

return true;
}


void Copter:: erle_run ()
{

Vector3f gyro;
static uint32_t last_print ;
static uint16_t contador;

uint32_t now = hal.scheduler->micros();

if( last_print == 0)
{
last_print = now;
return;
}

ins.wait_for_sample();

ins.update();

gyro = ins.get_gyro();
erle.p = ToDeg(gyro.x);
erle.q = ToDeg(gyro.y);
erle.r = ToDeg(gyro.z);

ahrs_erle.update();

erle. roll = ToDeg(ahrs_erle. roll );
erle. pitch = ToDeg(ahrs_erle. pitch );
erle.yaw = ToDeg(ahrs_erle.yaw);

contador++;
if(now - last_print >= 1000000)
{
printf("Actitud -> Roll: %4.1f | Pitch: %4.1f | Yaw: %4.1f | Rate: %.1f \n", erle. roll , erle. pitch , erle.yaw,(1.0e6*contador)/(now
      - last_print ));
last_print = now;
contador = 0;
}
```

Si contásemos con el quadrotor real, simplemente tendríamos que moverlo manualmente para que la *IMU* detectase variaciones en su actitud. Para simular este experimento, se han desactivado las físicas de GAZEBO® permitiéndonos mantener el quadrotor en el aire y realizar distintas inclinaciones.



(a) Terminal de ardupilot. **(b)** Gazebo.

Figura 6.8 Terminales Ubuntu.

ArduCopter con sensores

Comenzamos añadiendo las librerías necesarias, en este caso los archivos *.h* que hemos creado anteriormente y la librería general de *ArduCopter*, el archivo **Copter.h** nos será muy útil, ya que en ella se definen e inicializan la gran mayoría de sensores, así como el *HAL*(Subsección 6.1.2). También definimos el canal que se corresponde con cada uno de los motores. El motor delantero derecho (**MOTOR_FR**) se corresponde con el canal 0 de salida, el delantero izquierdo(**MOTOR_FL**) con el canal 2, el trasero derecho(**MOTOR_BR**) con el canal 3 y por último el trasero izquierdo (**MOTOR_BL**) con el 1,

Código 6.20 Include + Define.

```
#include "Copter.h"
#include " erlevariables .h"
#include "ErleFunciones.h"

#define MOTOR_FR 0
#define MOTOR_FL 2
#define MOTOR_BR 3
#define MOTOR_BL 1
```

En la función **erle_init**() realizamos algunas inicializaciones y declaraciones, que solo se ejcutarán la primera vez que se entre en este modo de vuelo. Definimos la frecuencia con la que se enviarán las salidas a los motores, y habilitamos los canales de salida del **HAL**. En esta función se definen las posiciones de referencia y se escribe en la consola de ardupilot a modo informativo. Además se realiza la inicialización de los sensores, y establecemos la tasa de medida de la *IMU* igual a nuestro tiempo de muestreo, es decir, unos $100Hz$.

Código 6.21 erle_init() con sensores.

```
#include "Copter.h"

#define MOTOR_FR 0
#define MOTOR_FL 2
#define MOTOR_BR 3
#define MOTOR_BL 1

float tiempo = 0; // Contador


bool Copter::erle_init(bool ignore_checks)
{
  //cliSerial->print_P(PSTR("ERLE"));

  hal.rcout->set_freq(0xF,100);

  // Se habilitan los canales de salida
  for(uint8_t i = 0; i < 8; i++)
  {
    hal.rcout->enable_ch(i);
  }

  hal.gpio->pinMode(40,HAL_GPIO_OUTPUT);
  hal.gpio->write(40,1);

  ins.init(ins.Sample_rate::RATE_100HZ);
  ins.init_gyro();

  barometer.init();

  ahrs_erle.init();

  erle.X_des = 0;
  erle.Y_des = 0;
  erle.Z_des = 1;

  printf("Altura deseada -> %.1f (m)\n",erle.Z_des);

    tiempo = 0;

    return true;
}
```

Ya solo quedaría crear el bucle de control en la función **erle_run**(), dicha función se ejecuta cada 100 *hz*, es decir, 0.01 segundos, lo que equivale al tiempo de muestreo que se utilizó para el diseño de los controladores.

Creamos un vector de tres componentes, para almacenar las medidas de los giroscopios, las velocidades angulares p,q y r en $\frac{rad}{s}$. Los ángulos de actitud los sacamos directamente de *AHRS* en *rad*.La altura se

obtiene directamente del barómetro, y las posiciones, se obtienen integrando las aceleraciones lineales.

Seguimos la misma estructura que en la simulación de MATLAB® , primero llamamos al control de más alto nivel, el contorl de posición implementado en la función **Position_Control**(). Seguidamente se incorpora el control de actitud y de altura, que se han implementado en funciones separadas, una para cada controlador, por lo tanto tenemos una función que se encarga del control del ϕ, otra del θ, la última función del control de actitud que se encarga del ψ y la función encargada de controlar la altura del quadrotor **Altitude_PID_Hold**(). Y seguimos la misma estructura para el *rate_control* (Sección 3.2).

Una vez tenemos calculadas las señales de control, se realiza la saturación de las actuaciones. A continuación debemos realizar la conversión de velocidades de los morotes $\omega(rad/s)$ a señales **PWM**, esta es la conversión que se ha estimado en la Sección 6.3. Y por último, realizamos la llamada a las funciones encargadas de escribir los datos en el *.log*.

Código 6.22 erle_run() con sensores.

```
void Copter::erle_run()
{

Vector3f gyro;

//ins.wait_for_sample();

ins.update();

gyro = ins.get_gyro();
erle.p = gyro.x;
erle.q = -gyro.y;
erle.r = -gyro.z;

ahrs_erle.update();

erle.X_dd = ahrs_erle.get_accel_ef().x;
erle.Y_dd = -ahrs_erle.get_accel_ef().y;
erle.Z = ahrs_erle.get_baro().get_altitude();

erle.roll = ahrs_erle.roll;
erle.pitch = -ahrs_erle.pitch;
erle.yaw = -ahrs_erle.yaw;

erle.Position_Control();
erle.Correct_Yaw();
erle.Altitude_PID_Hold();

erle.Stabilize_PID_Roll();
erle.Stabilize_PID_Pitch();
erle.Stabilize_PID_Yaw();
```

```
erle.Rate_PID_Roll();
erle.Rate_PID_Pitch();
erle.Rate_PID_Yaw();

erle.Saturacion_Actuaciones();

float pwm0 = erle.w0 + 1000;
float pwm1 = erle.w1 + 1000;
float pwm2 = erle.w2 + 1000;
float pwm3 = erle.w3 + 1000;

hal.rcout->write(MOTOR_FL, pwm2);//2
hal.rcout->write(MOTOR_BL, pwm1);//1
hal.rcout->write(MOTOR_FR, pwm0);//0
hal.rcout->write(MOTOR_BR, pwm3);//3

erle.Ecuaciones_Dinamicas();

if(tiempo >= 10)
{
printf("Posicion actual -> X = %.1f (m) | Y = %.1f (m) | Z = %.1f (m) \n",
    erle.X,erle.Y,erle.Z);
printf("Attitude -> roll %.1f | pitch %1.f | yaw %1.f \n",erle.roll,erle.
    pitch,erle.yaw);
printf("p -> roll %.1f | q %1.f | r %1.f \n",erle.p,erle.q,erle.r);
printf("Attitude -> w0 %.1f | w1 %1.f | w2 %1.f w3 %1.f \n",erle.w0,erle.w1,
    erle.w2,erle.w3);
printf("U1 %.1f | U2 %1.f | U3 %1.f U4 %1.f \n",erle.U1,erle.U2,erle.U3,erle.
    U4);
//printf("CONTADOR = %1.f",erle.contador);
tiempo = 0;
}
tiempo++;

DataFlash.Log_Write_ERLE(erle);
DataFlash.Log_Write_ERLE_POS(erle);

return;

}
```

Análisis de resultados

Tras varios intentos, modificaciones en la simulación y un gran número de experimentos, no se ha conseguido un funcionamiento similar al de la simulación realizada en MATLAB® . El quadrotor inicia el vuelo hasta alcanzar una altura, un poco mayor que la de referencia, pero una vez alcanzada, no mantiene su posición y vuelve a una altura de cero metros.

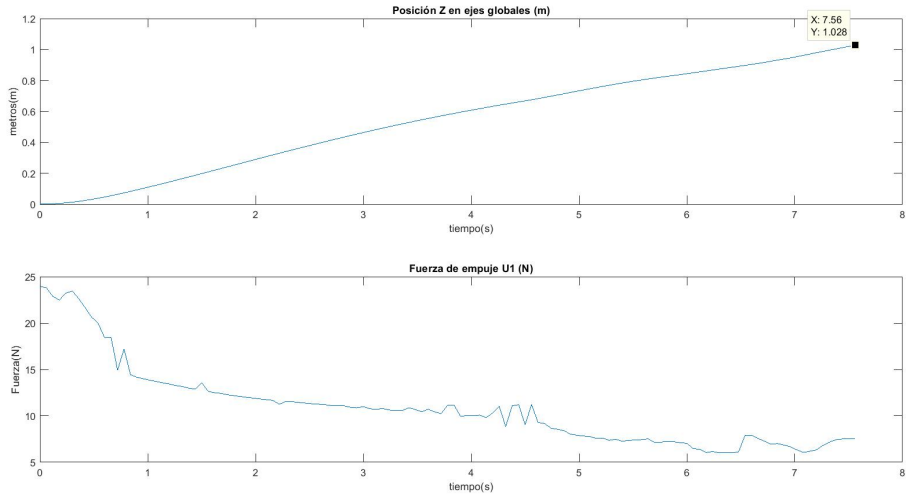


Figura 6.9 Altura y fuerza de empuje.

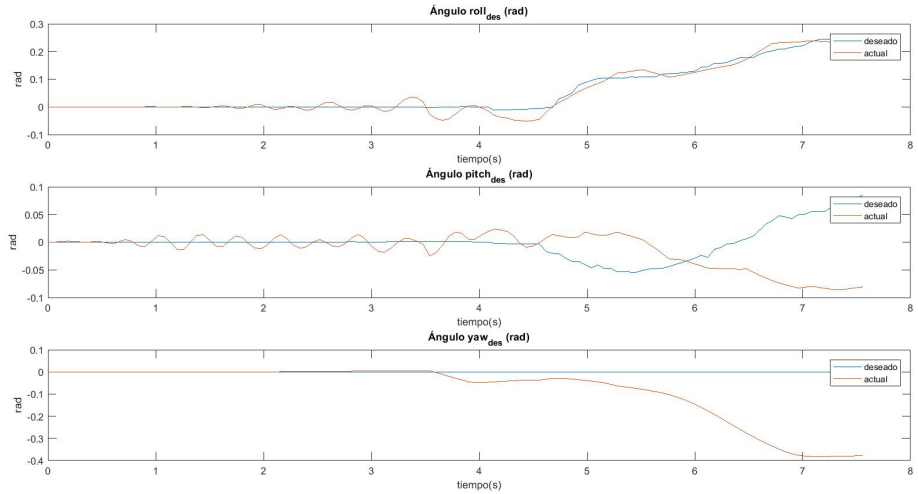


Figura 6.10 Control de actitud.

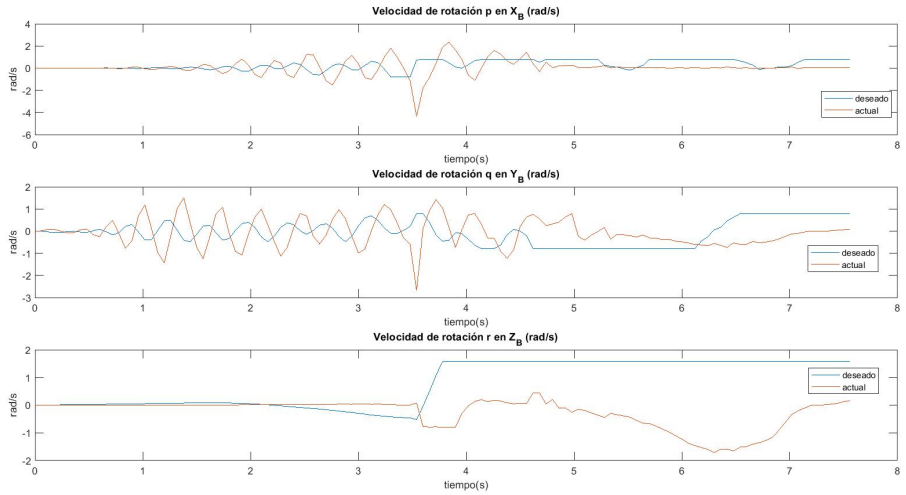


Figura 6.11 Rate Control.

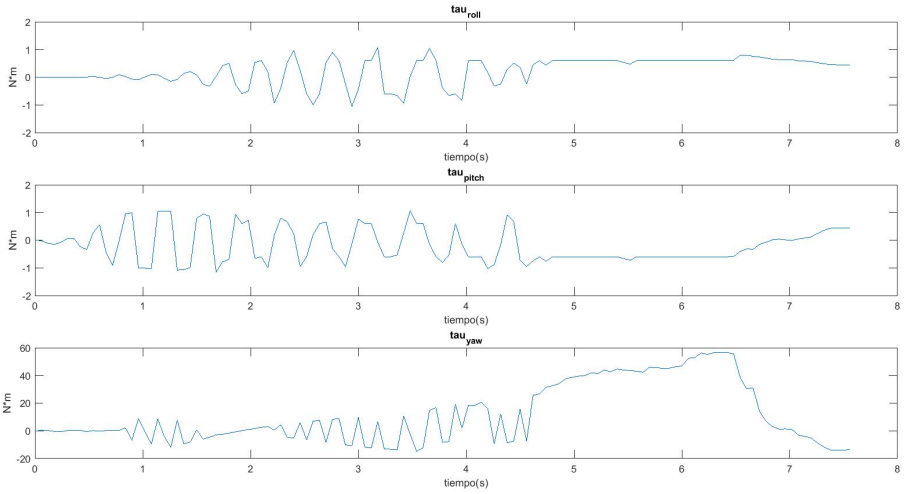
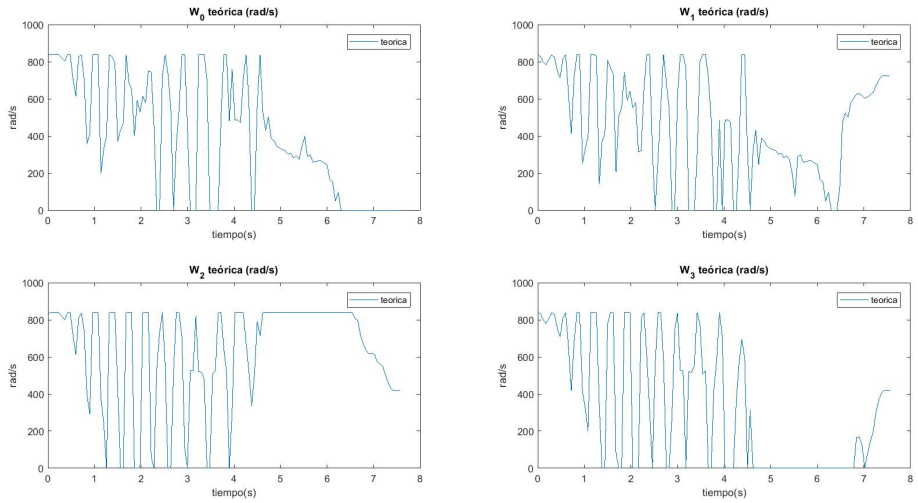
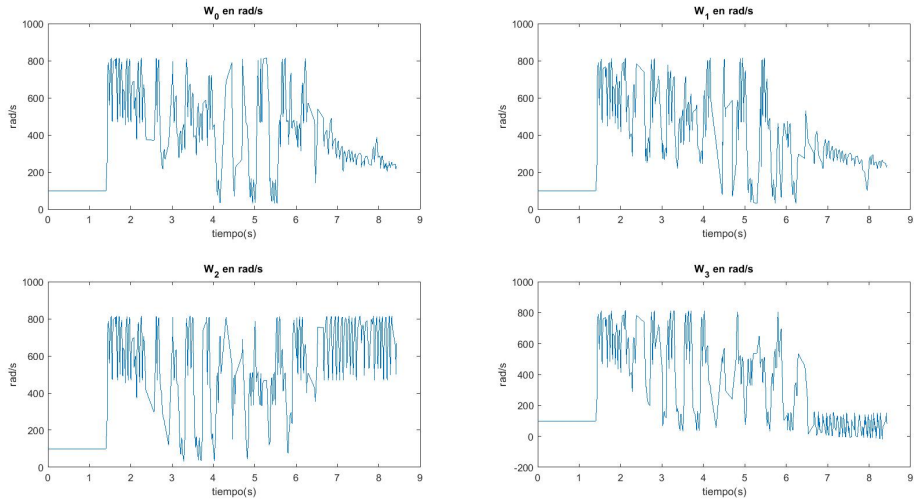


Figura 6.12 Momentos angulares.



(a) Velocidades teóricas.



(b) Velocidades medidas por Gazebo.

Figura 6.13 Velocidades de los motores (rad/s).

Como podemos ver en la gráfica 6.11, que hace referencia al control de más bajo nivel, no se consigue una respuesta estable del sistema, haciendo que el quadrotor comience a balancearse en el aire. Atendiendo a la gráfica 6.10 vemos como el control de actitud encargado del ϕ sigue la referencia, pero obtenemos una respuesta muy mala tanto para el θ como para el ψ. Por último vemos en la gráfica 6.12 como las velocidades de los motores oscilan, lo que supone, que el quadrotor no se mantenga a una altura fija.

Pese a las revisiones el código y diseño del control, no se ha conseguido un vuelo satisfactorio. Tras varias revisiones y cambios, no se ha conseguido dar con una solución válida, debido a todas las partes de las que se compone este control, hacen que la detección del error sea compleja y se demore más de lo

deseado, pudiendo estar en una mala configuración de los sensores, en los tiempos de las ejecuciones y llamadas de las funciones, así como en el propio diseño de los controladores.

7 Conclusiones

Se ha analizado el comportamiento físico de un quadrotor, entendiendo la física de este tipo de vehículos. Se ha conseguido finalmente un modelo matemático intuitivo y de fácil compresión para este tipo de vehículos, que además ha sido utilizado por una gran cantidad de autores, lo cuál hace suponer que es un modelo fiable para proyectos de este tipo. Estas ecuaciones obtenidas se han utilizado para representar el quadrotor de la empresa $Erle-Robotics$, pero con una simple sustitución de los parámetros físicos correspondientes, puede ser utilizado para cualquier vehículo de tipo quadrotor. Esta sencillez hace que el modelo pueda no ser demasiado realista, pudiendo no ser lo suficientemente fiable para trabajos más complejos, ya que se han realizado suposiciones y se han despreciado fuerzas y efectos, como por ejemplo posibles deformaciones en la estructura del vehículo debido a efectos aerodinámicos.

Partiendo del modelo matemático calculado en el Capítulo 2, se ha desarrollado una estrategia de control que representa un modo de vuelo autónomo basado en controladores de tipo *PID*. Se ha montado un entorno de simulación bastante completo en MATLAB® y SIMULINK® , donde se dispone de una simulación gráfica en 3*D* que representa fielmente el vuelo de un quadrotor. Se ha explicado cómo obtener las funciones de transferencia a partir de ecuaciones matemáticas y cómo diseñar los controladores utilizando el lugar de las raíces. Con esto podemos concluir que se ha conseguido uno de los objetivos de este trabajo, que consistía en disponer de un entorno sencillo de simulación y desarrollo de estrategias de control para vehículos de este tipo.

Atendiendo al segundo objetivo de este trabajo, no ha sido posible alcanzarlo satisfactoriamente. Se ha realizado una introducción en el piloto automático *Ardupilot*, que es de los más conocidos hoy en día y que cuenta con una gran cantidad de configuraciones y funcionalidades, además de ser compatible con una gran cantidad de vehículos. Se ha visto cómo añadir una nueva estrategia de vuelo, donde implementar nuevos diseños sin alterar las distintas estrategias que ya vienen implementadas. En este trabajo se ha intentado implementar un control de bajo nivel, con el objetivo de alcanzar una implementación propia lo más sencilla y directa posible.

Muy a mi pesar he tenido que concluir este trabajo sin la estrategia de control diseñada implementada de forma correcta en *ArduCopter*, pero espero que pueda ser de ayuda para futuros lectores, bien sea para alguien que tiene mayores conocimientos en programación o en el control de este tipo de vehículos, o simplemente alguien que quiere iniciarse en el mundo de *ArduCopter*.

Con respecto a futuros trabajos y líneas de investigación, recomendaría que si alguien intenta realizar un trabajo similar, consiga realizar una verificación del modelo estimado, lo cuál es algo complejo para este tipo de vehículos, además que se disponga de un vehículo físico a parte de la simulación, ya que suele

haber mayor información. Lo ideal sería conseguir implementar de forma correcta una estructura de control propia, de forma que se disponga de un modo de vuelo propio dentro del autopiloto, cuya estructura sea genérica y sencilla, de forma que modificando los parámetros de los controladores, pueda ser aplicada a cualquier quadrotor disponible.

Apéndice A
erlecopter.xacro

Código A.1 erlecopter.xacro.

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

<robot name="erlecopter" xmlns:xacro="http://ros.org/wiki/xacro">
<!-- Properties -->
<xacro:property name="namespace" value="erlecopter" />
<xacro:property name="rotor_velocity_slowdown_sim" value="10" />
<xacro:property name="mesh_file" value="erlecopter.dae" />
<xacro:property name="mesh_scale" value="1 1 1"/> <!-- 1 1 1 -->
<xacro:property name="mesh_scale_prop" value="1 1 1"/>
<xacro:property name="mass" value="1.1" /> <!-- [kg] -->
<xacro:property name="body_length" value="0.18" /> <!-- [m] 0.10 -->
<xacro:property name="body_width" value="0.12" /> <!-- [m] 0.10 -->
<xacro:property name="body_height" value="0.10" /> <!-- [m] -->
<xacro:property name="mass_rotor" value="0.005" /> <!-- [kg] -->
<xacro:property name="arm_length_front_x" value="0.141" /> <!-- [m] 0.1425 0.22 -->
<xacro:property name="arm_length_back_x" value="0.141" /> <!-- [m] 0.154 0.22 -->
<xacro:property name="arm_length_front_y" value="0.141" /> <!-- [m] 0.251 0.22 -->
<xacro:property name="arm_length_back_y" value="0.141" /> <!-- [m] 0.234 0.22 -->
<xacro:property name="rotor_offset_top" value="0.030" /> <!-- [m] 0.023-->
<xacro:property name="radius_rotor" value="0.12" /> <!-- [m] -->
<xacro:property name="motor_constant" value="8.54858e-06" /> <!-- [kg.m/s^2] -->
<xacro:property name="moment_constant" value="0.016" /> <!-- [m] -->
<xacro:property name="time_constant_up" value="0.0125" /> <!-- [s] -->
<xacro:property name="time_constant_down" value="0.025" /> <!-- [s] -->
<xacro:property name="max_rot_velocity" value="838" /> <!-- [rad/s] -->
<xacro:property name="sin30" value="0.5" />
<xacro:property name="cos30" value="0.866025403784" />
<xacro:property name="sqrt2" value="1.4142135623730951" />
<xacro:property name="rotor_drag_coefficient" value="8.06428e-05" />
<xacro:property name="rolling_moment_coefficient" value="0.000001" />
<xacro:property name="color" value="DarkGrey" />

<!-- Property Blocks -->
<xacro:property name="body_inertia">
<inertia ixx="0.0347563" ixy="0.0" ixz="0.0" iyy="0.0458929" iyz="0.0" izz="0.0977" /> <!-- [kg.m^2] [kg.m^2] [kg.m^2] [kg.m^2]
    [kg.m^2] [kg.m^2] -->
</xacro:property>

<!-- inertia of a single rotor, assuming it is a cuboid. Height=3mm, width=15mm -->
<xacro:property name="rotor_inertia">
<inertia
ixx="${1/12 * mass_rotor * (0.015 * 0.015 + 0.003 * 0.003) * rotor_velocity_slowdown_sim}"
iyy="${1/12 * mass_rotor * (4 * radius_rotor * radius_rotor + 0.003 * 0.003) * rotor_velocity_slowdown_sim}"
izz="${1/12 * mass_rotor * (4 * radius_rotor * radius_rotor + 0.015 * 0.015) * rotor_velocity_slowdown_sim}"
ixy="0.0" ixz="0.0" iyz="0.0" /> <!-- [kg.m^2] [kg.m^2] [kg.m^2] [kg.m^2] [kg.m^2] [kg.m^2] -->
</xacro:property>
```

Apéndice B
Errores durante la instalación del entorno

En la compilación del espacio de trabajo, cuando se ejecuta el comando:

```
catkin_make --pkg mav_msgs mavros_msgs gazebo_msgs
```

Se obtenía el siguiente error:

```
Project 'rotors\_control' tried to find library 'glog'....
Did you compile project 'glog\_catkin'?...
--Configuring incomplete, errors ocurred!
```

La solución se encontró en *http://forum.erlerobotics.com/t/rotors-control-cannot-find-library-glog/4410/2*.
Se debe instalar el paquete *glog* desde el repositorio *https://github.com/google/glog8*, y luego nos vamos al directorio donde hemos clonado el paquete y ejecutamos:

```
    ./autogen.sh && ./configure && make &&
    sudo make install
```

El segundo problema que se encontró, fue a la hora de ejecutar el comando:

```
catkin_make -j 4
```

Y donde nos aparece el siguiente error:

```
[  0%] Performing patch step for 'glog_src'
/bin/sh: 1: cannot open /home/erle/simulation/ros_catkin_ws/src/fix-unused-
    typedef-warning.patch: No such file
make[2]: *** [glog_catkin/glog_src-prefix/src/glog_src-stamp/glog_src-patch]
    Error 2
make[1]: *** [glog_catkin/CMakeFiles/glog_src.dir/all] Error 2
make[1]: *** Se espera a que terminen otras tareas....
[  0%] Built target mark_plugin
[  1%] Built target mavconn
[  1%] Built target _mav_msgs_generate_messages_check_deps_CommandMotorSpeed
make: *** [all] Error 2
```

```
Invoking "make -j 4" failed
```

Este problema se soluciona ejecutando el siguiente comando:

```
cp fix-unused-typedef-warning.patch ~/simulation/ros_catkin_ws/src
```

Apéndice C
erlevariables.h

Código C.1 erlevariables.h.

```
/*
 * erlevariables .h
 *
 * Created on: 19/03/2020
 *      Author: erle
 */

#ifndef ERLEVARIABLES_H_
#define ERLEVARIABLES_H_

#include <stdio .h>
#include <math.h>
#include <SITL/SIM_Aircraft.h>
using namespace SITL;

class erlevariables {
  public:

    float Deg_Rad = 3.14/180;
    float Rad_Deg = 180/3.14;

    float Tm = 0.01;

    // Parámetros del motor
    float Kt = 8.5486e-6;
    float w_max = 838;// rad/s
    float w_min = 0;
    float Jr = 2.409e-5;

    // Parámetros físicos
    float Kdx = 0.15;
    float Kdy = 0.15;
    float Kdz = 0.3;
    float Kd = 8.06428e-5;
    float l = 0.141; // (m)
    float m = 1.12; // (Kg)
    float Ixx = 0.0347563; // (Kg*m²)
    float Iyy = 0.0458929; // (Kg*m²)
    float Izz = 0.0977;    // (Kg*m^2)
    float g = 9.81; // (m/s²)

    // En coordenadas cuerpo
    // Posiciones
    float X_BF = 0;
    float Y_BF = 0;
```

```
float Z_BF = 0;
// Velocidades lineales
float X_d_BF = 0;
float Y_d_BF = 0;
float Z_d_BF = 0;
// Aceleraciones lineales
float X_dd_BF = 0;
float Y_dd_BF = 0;
float Z_dd_BF = 0;

// Posiciones
float X = 0;
float Y = 0;
float Z = 0;
// Velocidades lineales
float X_d = 0;
float Y_d = 0;
float Z_d = 0;
// Aceleraciones lineales
float X_dd = 0;
float Y_dd = 0;
float Z_dd = 0;

// Ángulos de inclinación
float roll = 0;
float pitch = 0;
float yaw = 0;
// Variaciones de ángulo
float p = 0;
float q = 0;
float r = 0;
float p_ek_1 = 0;
float q_ek_1 = 0;
float r_ek_1 = 0;
// Velocidades de los motores
float w0 = 0;
float w1 = 0;
float w2 = 0;
float w3 = 0;

// Control de posición

float Z_des = 0;
float Z_des_filt = 0;
float Z_des_filt_1 = 0;
float X_des = 0;
float X_des_filt = 0;
float X_des_filt_1 = 0;
float Y_des = 0;
float Y_des_filt = 0;
float Y_des_filt_1 = 0;
float roll_des_yaw = 0;
float pitch_des_yaw = 0;

float X_int = 0;
float X_TI_F = 1;
float X_KP_F = 1;
float X_TI = 10;
float X_TD = 2.5000;
float X_KP = 0.3523;
float X_KI = 0.0352;
float X_KD = 0.8808;
float X_ek_1 = 0;
float X_Int_ek = 0;
float pitch_max = 45*Deg_Rad;

float Y_TI_F = 1;
float Y_KP_F = 1;
float Y_TI = 0.4;
float Y_TD = 0.1;
float Y_KP = 34.4036;
float Y_KI = 86.009;
float Y_KD = 3.4404;
```

```
float Y_Int_ek = 0;
float Y_ek_1 = 0;
float roll_max = 45*Deg_Rad;

// Control de altura
// Control de variación de ángulo

// Control de variación del roll
// C = zpk([-10 -5],[0],2909.8) ;
//F = zpk ([],[-1],1) ;
// [TI, TD,KP,KI,KD] = calculo_PID(2909.8,0.1,0.2)
// Control de altura
float Z_TI = 0.3000;
float Z_TI_F = 1;
float Z_TD = 0.0667;
float Z_KP = 872.9400;
float Z_KI = 2.9098e+03;
float Z_KD = 58.1960;
float U1_max = Kt*4*pow(w_max,2);
float U1_min = 0;
float Z_KI_lim = 0;
float Z_Int_ek = 0;
float Z_ek_1 = 0;

// Control de actitud
//Entradas
float roll_des = 0;
float roll_des_filt = 0;
float roll_des_filt_1 = 0;
float pitch_des = 0;
float pitch_des_filt = 0;
float pitch_des_filt_1 = 0;
float yaw_des = 0;
float yaw_des_filt = 0;
float yaw_des_filt_1 = 0;

// Control del roll
// [TI, TD,KP,KI,KD] = calculo_PID(529.01,0.2,0.1) ;
// F_PID = tf ([1],[0.2 1]) ;
float roll_TI = 0.2;
float roll_TD = 0;
float roll_TI_F = 0.25;
float roll_KP = 35.0800;
float roll_KI = 175.4000;
float roll_KD = 0;
float p_max = 45*Deg_Rad;
float roll_Int_ek = 0;
float roll_ek_1 = 0;

// Control del pitch
// [TI, TD,KP,KI,KD] = calculo_PID(172.18,0.2,0);
// F_PID = tf ([1],[0.25 1]) ;
float pitch_TI = 0.2;
float pitch_TD = 0;
float pitch_TI_F = 0.25;
float pitch_KP = 34.4360;
float pitch_KI = 172.1800;
float pitch_KD = 0;
float q_max = 45*Deg_Rad;
float pitch_Int_ek = 0;
float pitch_ek_1 = 0;

// Control del yaw
// [TI, TD,KP,KI,KD] = calculo_PID(342.98,0.17,0);
// F_PID = tf ([1],[0.5 1]) ;
float yaw_TI = 0.17;
float yaw_TD = 0;
float yaw_TI_F = 0.5;
float yaw_KP = 58.3066;
float yaw_KI = 342.9800;
float yaw_KD = 0;
float r_max = 90*Deg_Rad;
float yaw_Int_ek = 0;
```

```
    float  yaw_ek_1 = 0;

    // Salidas del control de actitud
    float  p_des = 0;
    float  q_des = 0;
    float  r_des = 0;

    // Control de variación de ángulo

    // Control de variación del roll
    // [TI,TD,KP,KI,KD] = calculo_PID(13.093,0.1,0)
    float  p_TI = 0.1;
    float  p_TD = 0;
    float  p_KP = 1.3093;
    float  p_KI = 13.0930;
    float  p_KD = 0;
    float  U2_max = sqrt(2)*l*Kt*pow(w_max,2);
    float  U2_min = -sqrt(2)*l*Kt*pow(w_max,2);
    float  p_Int_ek = 0;
    float  p_d = 0;

    // Control de variación del pitch
    // [TI,TD,KP,KI,KD] = calculo_PID(18.357,0.1,0);
    float  q_TI = 0.1;
    float  q_TD = 0;
    float  q_KP = 1.8357;
    float  q_KI = 18.3570;
    float  q_KD = 0;
    float  U3_max = sqrt(2)*l*Kt*pow(w_max,2);
    float  U3_min = -sqrt(2)*l*Kt*pow(w_max,2);
    float  q_Int_ek = 0;
    float  q_d = 0;

    // Control de variación del yaw
    // [TI,TD,KP,KI,KD] = calculo_PID(39.08,0.1,0);
    float  r_TI = 0.1;
    float  r_TD = 0;
    float  r_KP = 3.9080;
    float  r_KI = 39.0800;
    float  r_KD = 0;
    float  U4_max = 2*Kd*pow(w_max,2);
    float  U4_min = -2*Kd*pow(w_max,2);
    float  r_Int_ek = 0;
    float  r_d = 0;

    // Salidas del rate control
    float  U1 = 0;
    float  U2 = 0;
    float  U3 = 0;
    float  U4 = 0;


};

#endif /* ERLEVARIABLES_H_ */
```

Apéndice D
ErleFunciones.cpp

Código D.1 erlevariables.h.

```
/*
 * ErleFunciones.cpp
 *
 *  Created on: 19/03/2020
 *      Author: erle
 */

#include "ErleFunciones.h"
#include <SITL/SIM_Gazebo.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <time.h>
#include <math.h>

#include <AP_HAL/AP_HAL.h>

extern const AP_HAL::HAL& hal;
using namespace std;

#define PORT_DATA_FROM_GAZEBO_PLUGIN 5760
#define PORT_DATA_TO_GAZEBO_PLUGIN 5760

ErleFunciones::ErleFunciones(const AP_AHRS_DCM& ahrs,const AP_InertialSensor& ins,const AP_Baro& barometer) :
    _ahrs(ahrs),
    _ins(ins),
    _barometer(barometer)

{
  }

void ErleFunciones::Correct_Yaw()
{
  pitch_des_yaw = cos(yaw)*pitch_des - sin(yaw)*roll_des;
  roll_des_yaw = sin(yaw) * pitch_des + cos(yaw) * roll_des;

}

  void ErleFunciones::Position_Control ()
  {
    // Control de X
      X_des_filt = (2.5/(2.5+Tm))*X_des_filt_1 + (Tm/(2.5+Tm))*X_des;
```

```
    // Cálculo del error
        float X_ek = X_des_filt − X;
        // Incremento de la integral del error
         X_Int_ek = X_Int_ek + Tm∗X_ek;
         // Controlador PI
         pitch_des = ((0.1271∗(X_ek + X_Int_ek/7.5000 + 1.6667∗((X_ek − X_ek_1)/Tm))));

    // Saturación
         pitch_des = min(pitch_max,max(−pitch_max,pitch_des));

         X_des_filt_1 = X_des_filt;
         X_ek_1 = X_ek;

     // Control de Y
        Y_des_filt = (2.5/(2.5+ Tm))∗Y_des_filt_1 + (Tm/(2.5+Tm))∗Y_des;

    // Cálculo del error

        float Y_ek = Y_des_filt − Y;
        // Incremento de la integral del error
         Y_Int_ek = Y_Int_ek + Tm∗Y_ek;
         // Controlador PI
         roll_des = ((0.1271∗(Y_ek + Y_Int_ek/7.5000 + 1.6667∗((Y_ek − Y_ek_1)/Tm))));

    // Saturación
         roll_des = −roll_des;
         roll_des = min(roll_des,max(−roll_des, roll_des));

         Y_des_filt_1 = Y_des_filt;
         Y_ek_1 = Y_ek;
}

void ErleFunciones :: Altitude_PID_Hold()
{

     // Control de altitud
    Z_des_filt = (Z_TI_F/(Z_TI_F+Tm))∗Z_des_filt_1 + (Tm/(Z_TI_F+Tm))∗Z_des;
    // Cálculo del error
    double Z_ek = Z_des_filt − Z;
    // Incremento de la integral del error
     Z_Int_ek = Z_Int_ek + Tm∗Z_ek;
     // Controlador PI
     U1 = (+(Z_KP∗(Z_ek + (1/Z_TI)∗Z_Int_ek + Z_TD∗((Z_ek−Z_ek_1)/Tm))));
     // Saturación
     if (U1 < U1_min)
     {
       U1 = U1_min;
     }
     else if (U1 > U1_max)
     {
       U1 = U1_max;
     }

     Z_des_filt_1 = Z_des_filt;
     Z_ek_1 = Z_ek;
}

void ErleFunciones :: Stabilize_PID_Roll ()
{

      // Cálculo del error
      double roll_ek = roll_des_yaw − roll;

      // Incremento de la integral del error
       roll_Int_ek = roll_Int_ek + Tm∗roll_ek;

      // Controlador PI
       p_des = roll_KP∗(roll_ek + (1/roll_TI)∗roll_Int_ek + roll_TD∗((roll_ek−roll_ek_1)/Tm));

       // Saturación
       if ( p_des < −p_max)
       {
         p_des = −p_max;
```

```
        }
        else if (p_des > p_max)
        {
          p_des = p_max;
        }

         roll_des_filt_1  =  roll_des_filt  ;
        roll_ek_1  =  roll_ek ;

}

void ErleFunciones :: Stabilize_PID_Pitch ()
{
    // Cálculo del error
    double pitch_ek  = pitch_des_yaw − pitch ;
    // Incremento de la integral del error
     pitch_Int_ek  =  pitch_Int_ek  + Tm∗pitch_ek;
    // Controlador PI
     q_des = pitch_KP∗(pitch_ek + (1/ pitch_TI )∗pitch_Int_ek  + pitch_TD∗((pitch_ek−pitch_ek_1)/Tm));
     // Saturación
     if ( q_des < −q_max)
     {
       q_des = −q_max;
     }
     else if (q_des > q_max)
     {
       q_des = q_max;
     }

      pitch_des_filt_1  =  pitch_des_filt  ;
     pitch_ek_1  =  pitch_ek ;
}

void ErleFunciones :: Stabilize_PID_Yaw()
{
  // Control del yaw
      yaw_des_filt  = (yaw_TI_F/(yaw_TI_F+Tm))∗yaw_des_filt_1 + (Tm/(yaw_TI_F+Tm))∗yaw_des;
      // Cálculo del error
      double yaw_ek = yaw_des_filt  − yaw;
      // Incremento de la integral del error
       yaw_Int_ek = yaw_Int_ek + Tm∗yaw_ek;
      // Controlador PI
       r_des  = yaw_KP∗(yaw_ek + (1/yaw_TI)∗yaw_Int_ek + yaw_TD∗((yaw_ek−yaw_ek_1)/Tm));
       // Saturación
       if ( r_des  < −r_max)
       {
         r_des  = −r_max;
       }
       else if (r_des  > r_max)
       {
         r_des  = r_max;
       }

       yaw_des_filt_1  = yaw_des_filt ;
       yaw_ek_1 = yaw_ek;
}

void ErleFunciones :: Rate_PID_Roll()
{
  // Control de variación del roll
       double p_ek = p_des − p;
      // Incremento de la integral del error
      p_Int_ek  = p_Int_ek  + Tm∗p_ek;
      // Controlador PI
      U2 = p_KP∗(p_ek + (1/p_TI)∗p_Int_ek + p_TD∗((p_ek−p_ek_1)/Tm));
       // Saturación
       if (U2 < U2_min)
       {
         U2 = U2_min;
       }
       else if (U2 > U2_max)
       {
         U2 = U2_max;
```

```
        }

        p_ek_1 = p_ek;
}
void ErleFunciones :: Rate_PID_Pitch()
{
   // Control de variación del roll
        double q_ek = q_des − q;
       // Incremento de la integral del error
        q_Int_ek = q_Int_ek + Tm∗q_ek;
       // Controlador PI
        U3 = q_KP∗(q_ek + (1/q_TI)∗q_Int_ek + q_TD∗((q_ek−q_ek_1)/Tm));
         // Saturación
        if (U3 < U3_min)
        {
          U3 = U3_min;
        }
        else if (U3 > U3_max)
        {
          U3 = U3_max;
        }

        q_ek_1 =q_ek;
}
void ErleFunciones :: Rate_PID_Yaw()
{
   // Control de variación del roll
        double r_ek = r_des − r;
       // Incremento de la integral del error
        r_Int_ek = r_Int_ek + Tm∗r_ek;
       // Controlador PI
        U4 = r_KP∗(r_ek + (1/r_TI)∗r_Int_ek + r_TD∗((r_ek−r_ek_1)/Tm));
         // Saturación
        if (U4 < U4_min)
        {
          U4 = U4_min;
        }
        else if (U4 > U4_max)
        {
          U4 = U4_max;
        }

        r_ek_1 = r_ek;
}

void ErleFunciones :: Saturacion_Actuaciones ()
{
   double w0_2 = U1/(4∗Kt) − U2/(2∗sqrt(2)∗l∗Kt) − U3/(2∗sqrt(2)∗l∗Kt) − U4/(4∗Kd);
   double w1_2 = U1/(4∗Kt) + U2/(2∗sqrt(2)∗l∗Kt) + U3/(2∗sqrt(2)∗l∗Kt) − U4/(4∗Kd);
   double w2_2 = U1/(4∗Kt) + U2/(2∗sqrt(2)∗l∗Kt) − U3/(2∗sqrt(2)∗l∗Kt) + U4/(4∗Kd);
   double w3_2 = U1/(4∗Kt) − U2/(2∗sqrt(2)∗l∗Kt) + U3/(2∗sqrt(2)∗l∗Kt) + U4/(4∗Kd);

   if (w0_2 > pow(w_max,2))
        {w0_2 = pow(w_max,2);}

   if (w0_2 < pow(w_min,2))
        {w0_2 = pow(w_min,2);}

   if (w1_2 > pow(w_max,2))
        {w1_2 = pow(w_max,2);}

   if (w1_2 < pow(w_min,2))
        {w1_2 = pow(w_min,2);}

   if (w2_2 > pow(w_max,2))
        {w2_2 = pow(w_max,2);}

   if (w2_2 < pow(w_min,2))
        {w2_2 = pow(w_min,2);}

   if (w3_2 > pow(w_max,2))
        {w3_2 = pow(w_max,2);}
```

```
    if  (w3_2 < pow(w_min,2))
        {w3_2 = pow(w_min,2);}

    // Calculo las velocidades de los motores en (Deg/s)
    w0 = sqrt(w0_2);
    w1 = sqrt(w1_2);
    w2 = sqrt(w2_2);
    w3 = sqrt(w3_2);

    // Señales de control
    U1 = Kt*(pow(w0,2)+pow(w1,2)+pow(w2,2)+pow(w3,2));
    U2 = ( sqrt (2)*l*Kt/2)*(+pow(w1,2)+pow(w2,2)−pow(w0,2)−pow(w3,2));
    U3 = ( sqrt (2)*l*Kt/2)*(−pow(w0,2)−pow(w2,2)+pow(w1,2)+pow(w3,2));
    U4 = Kd*(−pow(w0,2)−pow(w1,2)+pow(w2,2)+pow(w3,2));

}

void ErleFunciones :: Ecuaciones_Dinamicas()
{

    X_dd = ((cos( roll )*cos(yaw)*sin(pitch) + sin( roll )*sin(yaw))*U1 − Kdx*X_d)/m;
    Y_dd = ((cos( roll )*sin(yaw)*sin(pitch) − sin( roll )*cos(yaw))*U1 − Kdy*Y_d)/m;
    Z_dd = (+(cos( roll )*cos(pitch))*U1 − Kdz*Z_d)/m−g;

    p_d = ((Iyy − Izz)*q*r − Jr*q*(−w0−w1+w2+w3) + U2)/Ixx;
    q_d = ((Izz − Ixx)*p*r + Jr*p*(−w0−w1+w2+w3) + U3)/Iyy;
    r_d = ((Ixx − Iyy)*p*q + U4)/Izz;
//      Cálculo de p,q y r
    p = p_d * Tm + p;
    q = q_d * Tm + q;
    r = r_d * Tm + r;

//p = _ins.get_gyro().x;
//q = _ins.get_gyro().y;
//r = _ins.get_gyro().z;

    float roll_d = p + sin( roll )*tan( pitch )*q + cos( roll )*tan( pitch )*r;
    float pitch_d = cos( roll )*q − sin( roll )*r;
    float yaw_d = sin( roll )/cos( pitch ) * q + cos( roll )/cos( pitch ) * r;
//
//

// // Velocidad y posición en Z
Z_d = Z_dd * Tm + Z_d;
Z = Z_d*Tm + Z;
// // Velocidad y posición en Y
Y_d = Y_dd * Tm + Y_d;
Y = Y_d * Tm + Y;
//
// // Velocidad y posición en X
X_d = X_dd * Tm + X_d;
X = X_d * Tm + X;

// Cálculo de los ángulos
roll = roll_d *Tm + roll;
pitch = pitch_d * Tm + pitch;
yaw = yaw_d * Tm + yaw;
/*roll = _ahrs.roll;
pitch = _ahrs.pitch;
  yaw =_ahrs.yaw;*/

}
```

Apéndice E
Matlab + ROS

E.1 Comunicacion Matlab + ROS

El proyecto se ha realizado en *Ubuntu*, y se ha optado por utilizar *Matlab* una máquina virtual con sistema operativo *Windows*. Lo primero que debemos hacer es configurar matlab para poder conectarse a la red de ROS creada por la simulación.

Primero debemos configurar la máquina virtual, en este caso se ha utilizado **VirtualBox** :

- En la MV : Dispositivos→Red→Preferencias, seleccionar **Adaptador puente en lugar de NAT**
- Añadir en el archivo host, las direcciones IP de la MV y de la máquina donde se ejecuta la simulación:

```
192.168.1.35  erle-Lenovo-Z50-70
192.168.1.40  javi-PC
```

- En Matlab de la MV tenemos que añadir a las variables de entorno las IP de las máquinas:

```
setenv('ROS_MASTER_URI','http://192.168.1.35:11311')
setenv('ROS_IP','192.168.1.40')
```

En la máquina Ubuntu también hay que añadir las direcciones IP de ambas máquinas, en una terminal:

```
$ export ROS_IP=192.168.1.40
$ export ROS_MASTER_URI=http://192.168.1.35:11311
```

En el archivo .bashrc añadimos la IP de la máquina donde se ejecuta el nodo MASTER:

```
$ cd ~
$ gedit .bashrc
ROS_MASTER_URI=http://192.168.1.35:11311
```

E.2 Añadir mensajes de ROS a Matlab

Puede que los mensajes que queremos utilizar no estén definidos en Matlab, por lo tanto debemos añadirlos para poder trabajar con ROS:

- Añadir en Add-Ons: 'Robotics System Toolbox Interface for ROS Custom Messages'
- Seleccionar el paquete de ROS donde se encuentran los mensajes definidos que queremos añadir. En este caso, por ejemplo vamos a añadir los mensajes de tipo mav_msgs *mav_msgs /home/erle/simulation/ros_catkin_ws/src/mav_comm/mav_msgs* y la copio en la MV.
- En Matlab, en la línea de comandos, se añade el path donde se encuentra la carpeta mav_msgs:

```
folderpath = 'C:/Users/.../[carpeta donde se encuentra mav_msgs]'
```

- La carpeta con la definición de los mensajes, debe contener

$$\begin{cases} \text{- Una carpeta llamada } msg \\ \text{- Una carpeta llamada } srv \\ \text{- Un archivo llamado } packagexml \end{cases}$$

- Ejecutamos el comando

```
rosgenmsg(folderpath)
```

- Por útlimo seguimos los 3 pasos que nos aparecerán en la ventana de comandos de Matlab.

Índice de Figuras

Índice de Tablas

Índice de Códigos

Bibliografía

[1] Conceptos básicos de MAVLink. Disponible en : *https://ardupilot.org/dev/docs/mavlink-basics.html*.

[2] Octavio Alfredo García Campos, *Modelado, simulación y control de un quadrotor para transporte de carga colgante. extensión al caso tridimensional.*, Disponible en : *http://bibing.us.es/proyectos/abreproy/60409/fichero/PFCAlfredo.pdf* .

[3] Wiki de ROS. Disponible en : *http://wiki.ros.org/* .

[4] Información del paquete MAVROS. Disponible en http://wiki.ros.org/mavros.

[5] JSBSim. Disponible en : *http://jsbsim.sourceforge.net/* .

[6] Ubuntu 14.04. Disponible en : *http://releases.ubuntu.com/14.04/* .

[7] Software In The Loop (SITL). Disponible en : *https://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html*.

[8] Controlador PID. Disponible en : *https://www.picuino.com/es/arduprog/control-pid.html*.

[9] Tutoriales Erle Robotics. Disponible en : *https://www.youtube.com/playlist?list=PL39WpgKDjDfU9bmeoYeoUkeGk8VLKxSB*.

[10] A.V.A. Disponible en : *http://www.uco.es/investiga/grupos/ava/node/1*.

[11] Mauricio Vladimir Peña Giraldo, *Modelamiento dinámico y control lqr de un quadrotor.*, Disponible en : *https://pdfs.semanticscholar.org/6906/092a3ad082ea6d40d54208c3c400e2511808.pdf* .

[12] Aleks Emilov Goranov, *Diseño e implementación de un cuadricóptero basado en el microcontrolador stm32f4*, 2016.

[13] Markus Kohm, *A bundle of versatile classes and packages.*, Disponible en : *http://www.ctan.org/pkg/koma-script/* .

[14] Mario Jiménez León, *Montaje y configuración del drone comercial erle-copter*, 2017.

[15] Teppo Luukkonen, *Modelling and control of quadcopter*, Febrero 2011.

[16] Pablo Gómez-Cambronero Martín, *Simulación en gazebo de quadrotor para transporte de carga colgante.*, Disponible en : *https://idus.us.es/bitstream/handle/11441/68448/TFG_Pablo*.

[17] Virginia Mazzone, *Controladores pid*, Disponible en : *https://www-eng.newcastle.edu.au/~jhb519/teaching/caut1/Apuntes/PID.pdf* .

[18] Miguel Ángel Guijarro Martínez, *Modelado y simulación de dron erle-copter con ros/gazebo*, 2018.

[19] Miguel Ángel Sánchez Yoldi, *ModelizaciÓn y simulaciÓn de un vehÍculo aÉreo no tripulado*, 2011.

[20] Will Selby, *System modeling.*, Disponible en : *https://www.wilselby.com/research/arducopter/modeling/*.

[21] Universidad Sevilla, *Modelling and control of quadcopter.*, Disponible en : *http://laplace.us.es/wiki/index.php/Ecuaciones_de_Euler_(CMR)*.

[22] Información sobre ArduCopter. Disponible en : *http://ardupilot.org/copter/index.html*.

[23] Vahram Stepanyan, *Estimation, navigation and control of multi-rotor*.