UNIVERSIDAD AUTÓNOMA DE YUCATÁN

FACULTAD DE MATEMÁTICAS

Desarrollo de un sistema de visión computacional para el vuelo autónomo de un Vehículo Aéreo No Tripulado

Tesis presentada por:
I.E. Manuel Esteban Poot Chin

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

Supervisada por:
Dr. Carlos Brito Loeza
Dr. Ricardo Legarda Sáenz

Mérida, Yucatán, México
Enero 2019

Declaración de Autoría

Yo, I.E. Manuel Esteban Poot Chin, declaro que esta tesis titulada, "Desarrollo de un sistema de visión computacional para el vuelo autónomo de un Vehículo Aéreo No Tripulado" y el trabajo aquí presentado es de mi autoría. Y confirmo que:

- El presente trabajo fue realizado durante el período de maestría que cursé en la Facultad de Matemáticas de la Universidad Autónoma de Yucatán.
- Toda la información pertenecientes a otros autores utilizada en el presente trabajo fue debidamente referenciada y mencionada.
- Las herramientas y material de terceros incluyendo bibliotecas, código fuente, software e imágenes fueron utilizadas exclusivamente con fines educativos y se referenciaron adecuadamente, hasta lo posible.

Resumen

En el presente trabajo, se presenta un algoritmo inspirado en el enfoque de control denominado *Visual Servoing*, el cual consiste de la utilización de información visual y técnicas de control para controlar los movimientos de un robot. La intención fue posibilitar que un Vehículo Aéreo No Tripulado (VANT) sea capaz de aterrizar de forma autónoma sin requerir del GPS en la aeronave, el cual es propenso a fallar en situaciones en donde la señal es nula o muy baja, imposibilitando que esta tarea se complete o se haga de forma precisa. A lo largo del documento, se detallan todos los aspectos utilizados para la implementación del algoritmo, incluyendo las herramientas de visión computacional y de control que le dan soporte. De igual manera, se describe el procedimiento llevado a cabo para validar su funcionamiento mediante simulación. Finalmente, se describen los resultados obtenidos en comparación al GPS convencional.

Es necesario mencionar que parte de este trabajo fue realizado en colaboración con el Ing. Carlos Acosta sirviendo de apoyo en la comprensión del sofware de simulación Gazebo y de las bibliotecas de ROS. De forma independiente, desarrolló un algoritmo de control de posición y seguimiento de trayectorias que complementan este trabajo. Los resultados obtenidos son expuestos de forma resumida en el apartado correspondiente, mientras que el trabajo en detalle puede ser consultado en la tesis titulada: "*Proceso de ensamblado y desarrollo de un entorno de simulación de un dron*"[1].

Agradecimientos

A mis padres Candelaria y José Manuel por haberme dado la oportunidad de estudiar, la educación y valores que me permitieron alcanzar este logro.

A mis tíos Martha y Leonel por haberme aconsejado y apoyado desde siempre, este logro se los debo principalmente a ustedes.

A mis primos Miriam y Diego por brindarme momentos de alegría que me permitieron no perder los ánimos.

A mi abuelito Esteban por estar ahí para mí en todo momento preocupándose y dándome consejo.

A mis hermanos Esperanza y Mauricio por ser parte de mi motivación para lograr mis metas y continuar adelante.

Al Dr. Carlos Brito y Ricardo Legarda por asesorarme en el transcurso de la maestría siendo parte importante primordial para la finalización del trabajo.

Al Dr. Curi, coordinador de la maestría, el cual me proporcionó su apoyo en el proceso de la maestría.

A mis profesores por haberme dado las herramientas que facilitaron el término de mi proyecto, principalmente al Dr. Arturo Espinosa y Dr. Carlos Brito que me asesoraron en todo momento en los tópicos del proyecto.

Al CONACYT por proporcionarme los recursos económicos para cubrir los gastos durante la maestría.

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Motivación

Hoy en día, una gran cantidad de Vehículos Aéreos No Tripulados (VANT) poseen la capacidad de aterrizar de forma automática, auxiliándose de información del GPS y de los sensores inerciales en la aeronave. Sin embargo, a pesar que este enfoque ofrece buenos resultados la mayoría de las casos, existen situaciones en donde ésta no es adecuada. El GPS por su forma de operar, puede verse afectado por variables externas que resultan en una estimación incorrecta de la posición. Como es bien sabido, el GPS funciona a base de satélites que orbitan el planeta, estos transmiten información periódicamente, utilizada por los receptores en Tierra para estimar su posición. Sin embargo, esto propicia que factores como el clima, obstáculos entre los satélites o inclusive la calidad del receptor mismo, produzcan una estimación incorrecta. Produciendo que se pierda la referencia espacial y haga muy complejo e inexacto satisfacer tareas como el aterrizaje o seguimiento de rutas. Es por ello que surge la necesidad de encontrar alternativas que sean capaces de auxiliar al VANT en tales situaciones, siendo una de ellas utilizar información visual como referencia espacial.

1.2. Objetivos

El objetivo general del presente trabajo consiste en implementar un algoritmo para el aterrizaje autónomo de un VANT, utilizando técnicas de visión computacional y control.

1.2.1. Objetivos Específicos

- Plantear el problema de aterrizaje utilizando *Visual Servoing*.
- Definir un marcador visual para indicar el punto de aterrizaje.
- Implementar un algoritmo para la detección del marcador.
- Implementar un algoritmo para la estimación de la pose del VANT.

- Definir una ley de control basada en *Visual Servoing*.
- Implementar un algoritmo que posibilite el aterrizaje del VANT.
- Validar el funcionamiento mediante simulación.

1.3. Organización de la Tesis

- **Capítulo 2: Antecedentes.**
  En este apartado se resumen los aspectos más importante de los trabajos consultados para el desarrollo del presente trabajo.
- **Capítulo 3: Marco Teórico.**
  Consiste de la explicación de todas las herramientas que fueron utilizadas en el presente proyecto desde la perspectiva de implementación.
- **Capítulo 4: Metodología.**
  En esta sección se puede encontrar la metodología llevada a cabo para desarrollar del algoritmo de visión. Es aquí donde se exponen las ideas que fueron utilizadas sin entrar en los detalles de implementación.
- **Capítulo 5: Implementación.**
  En esta parte se detallan como fue implementado en algoritmo en lenguaje C++. Se explica como fueron aprovechas las herramientas de ROS y OpenCV para su construcción así como la integración de las técnicas propuestas en la metodología
- **Capítulo 4: Simulación.**
  Se describen el procedimiento necesario para descargar y configurar el entorno de simulación que fue utilizado en el proyecto. De igual manera, se explican algunos aspectos importantes de como el algoritmo funciona desde la perspectiva de la simulación.
- **Capítulo 5: Resultados.**
  En esta sección se exponen los resultados observados durante el desarrollo del proyecto y las comparativas del algoritmo contra el GPS. Se presentan gráficas en donde se puede observar el rendimiento del algoritmo así como el error que este posee.
- **Capítulo 6: Conclusiones.** En este apartado se exponen las conclusiones a las que se llegó después de haber finalizado el presente proyecto así como el trabajo futuro.

Capítulo 2

Antecedentes

La problemática de automatizar el aterrizaje no es nueva, en la literatura es posible encontrar una gran cantidad de artículos que intentan darle solución, principalmente a través del uso del GPS. Sin embargo, debido a la poca confiabilidad que ofrece en ocasiones, se han propuesto otras alternativas. A continuación, se hará un breve repaso a algunas de ellas.

En el trabajo realizado por Marinela Georgieva Popova [6] se implementaron dos técnicas: una basada en imágenes y otra en pose. Para su funcionamiento, se asume que el objetivo está localizado en el piso y este no cambia en altitud. La información sobre la posición deseada es obtenida de una cámara montada en el vehículo, la cual tiene una orientación fija con respecto al VANT. Esto es aprovechado para medir la pose del VANT y finalmente llevar al vehículo hacia la posición deseada.

Araar et al en [7] propusieron la utilización de técnicas de visión computacional y marcadores cuadrados distintivos en tierra para estimar la pose de un cuadrucóptero. En base a ello y técnicas de fusión de sensores (un filtro de Kalman Extendido y un filtro $H\infty$), lograron que la aeronave aterrizara en una base en movimiento. Parte importante de su trabajo también fue el uso de marcadores a distintas escalas para permitir que estos sean identificados a diferentes altitudes.

De forma similar, Wang Chao en [8], utiliza técnicas de visión computacional y marcadores visuales en una base móvil para lograr el aterrizaje, sin embargo, a diferencia del proyecto anterior, el autor utiliza marcadores circulares en lugar de cuadrados. Mientras que Herisse et al en [9] utilizan una alternativa a los marcadores, un plano con texturas aleatorias y flujo óptico para lograr el control de un cuadrucóptero. Una implementación del algoritmo de Lukas-Kanade en conjunto con un controlador PI es utilizado para las maniobras de despegue y aterrizaje.

Por otro lado, Olivares et al en [10] proponen el uso de técnicas de control difuso y una cámara RGB para controlar el aterrizaje de una aeronave. En este trabajo se explica como la pose es estimada a partir de homografías y como son estas obtenidas utilizando flujo óptico, detección de esquinas y RANSAC para ofrecer robustez al

algoritmo utilizado.

Chitrakaran en [11] utiliza la homografía para estimar la posición de un cuadrucóptero y se concentra principalmente en analizar el problema desde la perspectiva matemática. En el documento se expone el modelo matemático de la aeronave, un controlador no lineal y la demostración de su factibilidad a través de un análisis de estabilidad de Lyapunov.

Otra alternativa es la propuesta de Bartak et al en [12] y Weaver et al en [13] que consiste en la detección de *blobs* y color. Los cuales transforman una imagen al espacio del color HSV y les aplican un proceso de erosión y dilatación lo que permite reforzar los bordes y eliminar las imperfecciones en la imagen, posteriormente esto es aprovechado para obtener un menor rendimiento en el proceso de detección de los *blobs*.

Un trabajo interesante es el de Courbon en [14] en donde se implementa una técnica denominada Memoria Visual la cual consiste en utilizar imágenes sucesivas durante la aeronave se va desplazando para obtener características y utilizarlas para navegación. Eso supuestamente está inspirado en la forma en que los seres humanos obtienen información del entorno.

De igual manera, existen algunos trabajos relacionados al seguimiento de trayectorias. En [15], Oualid Araar y Nabil Aouf utilizan el enfoque conocido como *Visual Servoing* basado en imágenes para posibilitar que un cuadrucóptero inspeccione, de forma autónoma, líneas de transmisión de energía eléctrica. La primicia del proyecto consiste asumir que este tipo de lineas son rectilíneas y paralelas, lo que permite desarrollar un algoritmo capaz de utilizarlas como referencia para la navegación de la aeronave.

Un trabajo a destacar es [6] de Popova, en el cual se explica a gran detalle dos técnicas de *Visual Servoing*: basada en imágenes y basada en pose. El autor comienza describiendo el modelo matemático de un cuadrucóptero analizando los fenómenos físicos que posibilitan su vuelo, posteriormente habla acerca de los enfoques de *Visual Servoing* diseñando para cada caso, un controlador aprovechando las ventajas de cada uno de ellos, después utiliza los mismos para analizar su factibilidad para el control de la aeronave.

Por otro lado, Wang Chao en [8], se motiva en el hecho de que los VANT puedan volar de manera independiente en lugares donde los sistemas de comunicación con la base de telemetría fallen. En el trabajo se utiliza un AR.Drone que implementa *Visual Servoing* basado en imágenes haciendo que el vehículo pueda detectar a través de su cámara un marcador visual con forma de círculo y realizar un aterrizaje.

Existen también otras alternativas no relacionadas con visión computacional. Kapoor en [16] describe como señales inalámbricas WiFi, GSM y CDMA, entre otras como señales de televisión y radio, pueden ser aprovechadas para guiar aeronaves en áreas urbanas donde la señal de los GPS es pobre y/o imprecisa. Menciona como a través

del Ángulo de Arribo (A0A, por sus siglas en inglés), la Fuerza de Señal Recibida (RSS por sus siglas en inglés) y la Diferencia Temporal de arribo (TDOA, por sus siglas en inglés) podrían servir como una referencia para determinar la posición de una aeronave y auxilie en su navegación.

En la publicación [17], Kassas et al explica como señales de antenas LTE y CDMA fueron utilizadas para auxiliar en la navegación de un hexacóptero. En sus conclusiones exponen resultados alentadores y demuestran que estas herramienta son factibles para la navegación de aeronaves en entornos con ausencia del GPS.

Finalmente, Bauer et al en [18] proponen la utilización de señales infrasónicas con fines de navegación. En él, se presentan las principales ideas y los primeros pasos para considerar a este enfoque como una alternativa real describiendo sus ventajas y desventajas contra tecnologías convencionales como el GPS.

Capítulo 3

Marco teórico

3.1. Vehículos Aéreos No Tripulado (VANT)

Los Vehículos Aéreos No Tripulados (VANT) son aeronaves, generalmente de reducidas dimensiones, que no poseen tripulación a bordo y son controlados de forma remota por un piloto en tierra o localmente mediante instrucciones preestablecidas antes de su vuelo.

3.1.1. Desplazamiento y ángulos

El problema de mantener en vuelo un VANT es por sí solo un problema de control complejo, por tal razón, la gran mayoría de las aeronaves poseen a bordo *computadoras de vuelo*, en donde se ejecutan algoritmos encargados de determinar el empuje y sentido de giro de cada motor para asegurar que la aeronave se mantenga en vuelo y se dirija hacia donde el usuario así lo desea. Esto permite controlarlo utilizando una sencilla interfaz, por ejemplo, un mando a distancia.

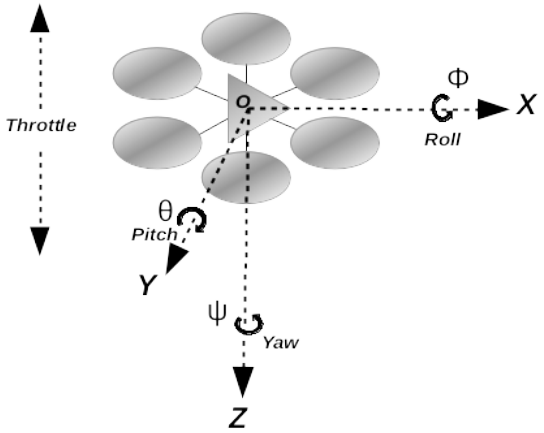


Figura 3.1: Ángulos de rotación y *Throttle* de un VANT.

El movimiento de un VANT se logra modificando su inclinación hacia la dirección a

la cual se desea navegar, resultado de ello, se origina un ángulo que varía dependiendo del empuje de los motores. Por convención, a estos ángulos se les tiene asignado un nombre que permite identificar el desplazamiento, siendo estos los siguientes: *Pitch*(ϕ), rotación en el eje X; *Roll*(θ), rotación en el eje Y y *Yaw*(ψ), rotación en Z_c. Un cuarto elemento, el *Throttle* que consiste en empuje que controla el desplazamiento vertical (elevación) del VANT. Su distribución en un VANT es mostrado en la Figura 3.1 para un hexacóptero, sin embargo, esto es similar en los demás VANT.

3.1.2. Mando a distancia

El control mediante el mando consiste en enviar periódicamente valores (sin unidad) que relacionan la posición actual de la palanca en el mando con la velocidad y sentido esperados en el VANT. Cada ángulo y el *Throttle* de la aeronave se encuentra relacionados con un *canal*, el cual consiste de un número que permite a la computadora de vuelo, diferenciarlos. Los valores que toma cada canal oscilan entre los 1000 y 2000 unidades, siendo el primero la máxima velocidad negativa y el segundo la máxima velocidad positiva. El valor intermedio, 1500, corresponde al punto de estabilidad del canal, en donde no existe velocidad en ningún sentido[1]. En la Figura 3.2 aparece una posible configuración del mando, resaltando la asignación de canales por palanca y los valores que toman dependiendo de su posición.

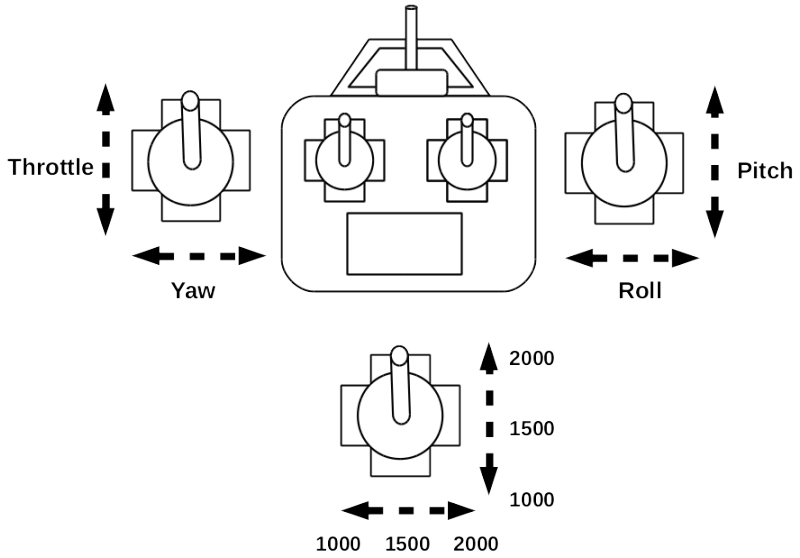


Figura 3.2: Distribución de los canales en el mando.

[1]Depende del modo de vuelo del VANT.

3.2. Transformaciones en 3D

Una transformación es una operación que permite modificar la posición de un punto en un plano cartesiano. Para el caso de puntos en tres dimensiones, las operaciones de transformación más comunes son: el escalado, la traslación y la rotación [19, 20].

Por conveniencia, las operaciones son expresadas utilizando coordenadas homogéneas. Esto permite representar a las transformaciones con una matriz de 4×4 operada con el punto a transformar:

$$P' = MP, \tag{3.1}$$

donde P' es el nuevo punto, M la matriz de transformación y P el punto original, todos ellos expresados en coordenadas homogéneas.

3.2.1. Traslación

La operación de traslación consiste en desplazar un punto, perteneciente a un plano cartesiano, a una nueva posición. Se logra incrementando cada uno de los componente del punto a una distancia t. Esta transformación es expresada como un vector en donde figuran los valores de incremento t para cada componente, como se muestra a continuación:

$$P' = TP = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} + \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \tag{3.2}$$

donde P' es el punto trasladado y (t_1, t_2, t_3) son los incrementos en las componentes (x, y, z) del punto original P, respectivamente.

Convenientemente, la traslación también puede ser representada en forma matricial utilizando coordenadas homogéneas, lo cual permite expresarla como sigue:

$$P' = TP = \begin{bmatrix} 1 & 0 & 0 & t_1 \\ 0 & 1 & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}. \tag{3.3}$$

Si consideramos todos los puntos que conforman una figura geométrica, una operación de traslación ocasiona que ésta se desplace a una nueva posición en el plan cartesiano pero, a diferencia de el escalamiento, esto no ocasiona la modificación en las dimensiones de la figura original. Un ejemplo de traslación es mostrado en la Figura 3.3.

3.2.2. Rotación

La rotación es una transformación que se produce a partir de un punto fijo en el plano cartesiano y no respecto al origen, como sucede en el escalamiento y la traslación.

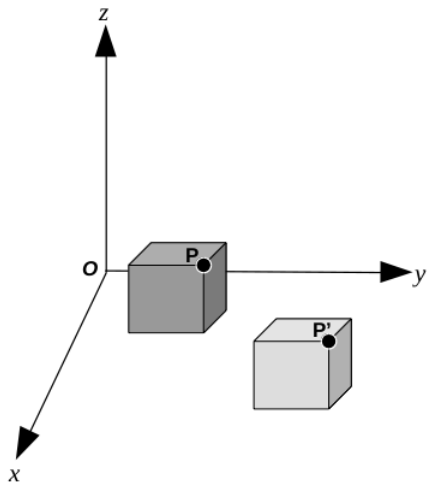


Figura 3.3: Transformación de Traslación.

Es representada por una matriz de 3×3, cuyos componentes permiten rotar al punto en cualquiera de los tres ejes del plano cartesiano (ver Figura 3.4):

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}. \tag{3.4}$$

Una de las características de la matriz de rotación consiste en que cada columna puede ser vista como un vector unitario, correspondiente a un eje del sistema de coordenadas principal, ortogonal a los demás vectores. Esto significa que es posible obtener cualquiera de los ejes a partir del producto cruz de los otros dos.

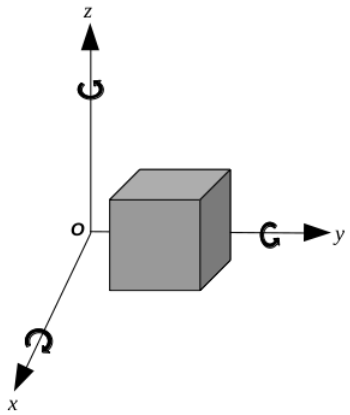


Figura 3.4: Transformación de rotación.

Un punto expresado en tres dimensiones expresado en coordenadas homogéneas pude rotarse aplicando la matriz de transformación homogénea. Para el caso de la

rotación, la transformación se describe como sigue:

$$P' = RP = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}, \tag{3.5}$$

donde P' es el punto rotado, R la matriz de rotación y P el punto original.

3.2.3. Matriz de rotación

Los orientación de un punto puede expresarse a través de los tres ángulos de Euler: *roll*(ϕ), *pitch*(θ) y *yaw*(ψ), que corresponden a la rotación respecto a los ejes x, y y z, respectivamente (ver Figura 3.5).

La rotación en caja eje puede ser vista como una transformación por sí misma, por tal razón, puede expresarse como una matriz para cada ángulo:

$$R_z(\psi) = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 & 0 \\ \sin(\psi) & \cos(\psi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3.6}$$

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3.7}$$

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) & 0 \\ 0 & \sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.8}$$

De igual manera puede compactarse todas las rotaciones en una sola matriz, denominada matriz de rotación generalizada:

$$R(\phi, \theta, \psi) = \begin{bmatrix} c(\psi)c(\theta) & c(\psi)s(\theta)s(\phi) - s(\psi)c(\phi) & c(\psi)s(\theta)c(\phi) + s(\psi)s(\phi) & 0 \\ s(\psi)c(\theta) & s(\psi)s(\theta)s(\phi) + c(\psi)c(\phi) & s(\psi)s(\theta)c(\phi) - c(\psi)s(\phi) & 0 \\ \text{-}s(\theta) & c(\theta)s(\phi) & c(\theta)c(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.9}$$

donde $c(\cdot)$ corresponde a la función coseno y $s(\cdot)$ a la función seno.

Ángulos de Euler

Anteriormente, se explicó como una matriz de rotación puede ser construida a partir de los ángulos de Euler. Sin embargo, existen ocasiones en donde se desea el proceso

inverso, es decir, obtener los ángulos de Euler a partir de la matriz de rotación generalizada. Si observamos la matriz 3.9, podemos reescribirla de la forma siguiente:

$$R(\phi,\theta,\psi) = \begin{bmatrix} R & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3.10}$$

de la cual pueden derivarse las ecuaciones mostradas a continuación:

$$\phi_{roll} = \arctan\left(\frac{r_{32}}{r_{33}}\right),$$

$$\theta_{pitch} = \arctan\left(\frac{-r_{31}}{\sqrt{(r_{32})^2 + (r_{33})^2}}\right), \tag{3.11}$$

$$\psi_{yaw} = \arctan\left(\frac{r_{21}}{r_{11}}\right).$$

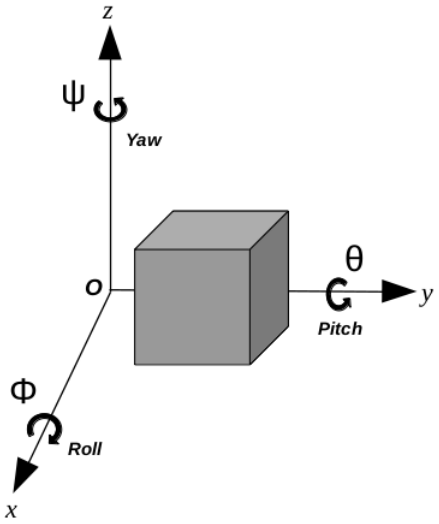


Figura 3.5: Ángulos de Euler.

3.3. Coordenadas Homogéneas

Las coordenadas homogéneas son un sistema de coordenadas utilizado en visión computacional y robótica, mediante la cual es posible representar transformaciones no lineales en forma lineal.

Un punto con coordenadas n dimensionales se representa en coordenadas homogéneas

utilizando $n+1$ elementos, tal y como se muestra a continuación:

$$(x_1, x_2, \ldots, x_n) = w \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_n \\ z \end{bmatrix},$$

donde $w > 0$ es un factor que indica escala y $z \geq 0$ el elemento que homogeneiza las coordenadas.

El uso de este factor de escala w, origina una de las características más importantes de las coordenadas homogéneas: un punto puede ser representado por un gran número de coordenadas variando únicamente su escala. A resumidas cuentas, mientras no se modifiquen sus componentes, todas las coordenadas resultantes del producto de las coordenadas de un punto por un factor de escala w, representan al mismo punto en coordenadas cartesianas. Sin embargo, muchas veces se considera que el punto no posee escala, es decir, que el valor de $w = 1$.

Otra característica importante de las coordenadas homogéneas es que permiten representar puntos en el infinito, algo que no es posible de hacer utilizando coordenadas cartesianas convencionales. Para expresar que un punto se encuentra en la infinidad, basta con asignar el valor de $z = 0$.

Contrario al sistema de coordenadas cartesianas, el origen en el sistema de coordenadas homogéneas no se encuentra en $(0, 0, \ldots, 0)$ si no en $(0, 0, \ldots, 1)$. Por tanto, $(0, 0, \ldots, 0)$ no representa punto alguno.

El procedimiento para convertir un punto de coordenadas homogéneas a su equivalente en coordenadas cartesianas consiste en normalizar sus componentes con respecto a z:

$$\frac{1}{z} \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_n \\ z \end{bmatrix} = (x_1, x_2, \ldots, x_n).$$

3.4. Visión Computacional

3.4.1. Cámara *pinhole* y geometría proyectiva

El modelo de una cámara describe la relación matemática entre un punto en el plano del mundo (tridimensional) y su proyección en el plano de la imagen (bidimensional). Por su simplicidad, el modelo más utilizado es el de la cámara pinhole ideal (ver Figura 3.6) el cual consiste de una caja con un pequeño orificio en la parte frontal (el pinhole) a través del cual cruza un único rayo de luz para cada punto existente en la parte trasera interna de la caja (el plano de la imagen) [2].

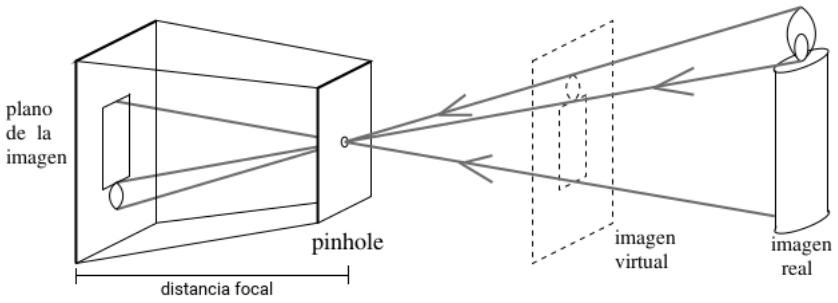


Figura 3.6: Modelo de cámara *pinhole*. Adaptado de [2].

El modelo de la cámara *pinhole* ideal es solo una aproximación de primer orden al modelo de una cámara real ya que obvia muchos factores que influyen en la creación de una imagen, por ejemplo: la distorsión y el desenfoque ocasionado por lentes y la discretización a *pixeles*. Sin embargo, muchos de estos efectos pueden ser compensados matemáticamente y otros despreciados (si la calidad de la cámara así lo permite), lo cual permite que sea utilizado como descripción razonable de como una cámara percibe una escena tridimensional [21]. Como puede observarse en la Figura 3.6, la proyección del mundo en la cámara produce una imagen invertida. No obstante, en ocasiones es conveniente analizar la imagen correctamente orientada. Para lograrlo, el plano de la imagen es desplazado al frente y por fuera de la cámara *pinhole*, dando origen a lo que se le conoce como *imagen virtual* [2].

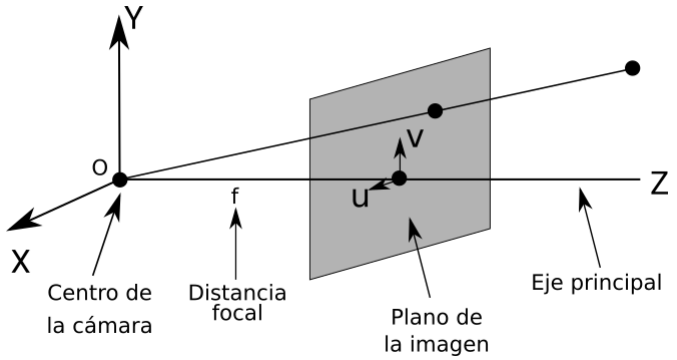


Figura 3.7: Relación geométrica de la cámara *pinhole*

Analizando la *proyección de perspectiva* a partir del plano de la *imagen virtual*, puede establecerse la relación geométrica mostrada en la Figura 3.7, de la cual se desprenden las relaciones siguientes:

$$u = f\frac{X}{Z} \ , \tag{3.12}$$

$$v = f\frac{Y}{Z} \ , \tag{3.13}$$

donde f es corresponde a la distancia focal de la cámara, (X, Y, Z) a las coordenadas en el mundo real y (u, v) a las coordenadas en la imagen.

3.4.2. Matriz de la cámara

El proceso de proyección llevado a cabo en la cámara *pinhole* puede ser expresado como una transformación lineal, utilizando una matriz homogénea $C \in \mathbb{R}^{3\times4}$, a la cual se le denomina matriz de la cámara. Si consideramos un punto $P(X, Y, Z)$ en el mundo real y su proyección $p(u, v)$ en el plano de la imagen, ésta transformación puede expresarse mediante la siguiente ecuación:

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = C \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix},$$

Esta matriz C se compone de los parámetros intrínsecos y extrínsecos de la cámara, los cuales son estimados a partir de un proceso denominado calibración geométrica de la cámara o *camera resectioning*. La matriz descompuesta en sus elementos es mostrada a continuación:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K[R|T] \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}, \tag{3.14}$$

donde K es la matriz de parámetros intrínsecos y R y T los parámetros extrínsecos de la cámara [22].

Finalmente, esto puede expandirse mostrando todos sus elementos (homogeneizados), como se muestra a continuación:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & s & c_x & 0 \\ 0 & f_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{23} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}. \tag{3.15}$$

Parámetros Intrínsecos

La matriz de parámetros intrínsecos consiste de una matriz de 3×3 cuyos elementos relacionan el plano de la imagen con el plano de la cámara. La distribución de sus elementos es la siguiente:

$$K = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \tag{3.16}$$

donde f_x y f_y corresponden a la distancia focal (en pixeles) de la cámara, c_x y c_y a la posición del origen de la imagen (en pixeles) y s un factor de sesgo.

Los componentes f_x y f_y representan la distancia que existe del origen de la cámara hacia el plano de la imagen. En el modelo *pinhole* (ver Figura 3.6), esta distancia

es igual para ambos ejes. En cámaras reales, sin embargo, es posible que exista una distancia focal diferente para cada eje, es por ello que se maneja como dos parámetros independientes.

Por otro lado, el origen de la imagen puede no estar ubicado al centro de la imagen (el origen de la cámara), es por ello que existen los parámetros c_x y c_y mediante los cuales es posible indicar el *offset* existente.

Por último, el valor s es un factor de sesgo que indica la disparidad entre los ejes de un *pixel*, el cual es normalmente asumido como un cuadrado simétrico, a pesar que en la vida real esto puede no ser así.

Parámetros Extrínsecos

Los parámetros extrínsecos describen la ubicación de la cámara en el sistema de coordenadas del mundo real y consiste de una matriz de rotación y un vector de traslación.

El vector de traslación corresponde a la posición del origen del marco de referencia del mundo real, en las coordenadas de la imagen. Mientras que la matriz de rotación indica su posición y dirección.

Normalmente, estos componentes son compactados en una matriz aumentada, como se muestra a continuación:

$$[R|T] = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix}, \tag{3.17}$$

3.5. *Visual Servoing*

Se le denomina *Visual Servo* o *Visual Servoing* a la utilización de técnicas de visión computacional y de teoría de control para manipular los movimientos de un robot. Su funcionamiento consiste en el utilizar información de imágenes capturadas de la escena para estimar el estado actual del robot y en base a ello determinar las acciones que lo lleven a un estado objetivo.

De forma general, el problema de *Visual Servoing* puede ser expresado como una función de error $\mathbf{e}(t)$ que relaciona al vector de características $\mathbf{s}$ del estado actual del robot con el vector de características objetivo $\mathbf{s}^*$ [23], es decir:

$$\mathbf{e}(t) = \mathbf{s}(\mathbf{m}(t), \mathbf{a}) - \mathbf{s}^*, \tag{3.18}$$

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

3.5.1. Enfoques

Dependiendo del tipo de características utilizadas para describir el error entre el estado actual y el objetivo, puede clasificarse a *Visual Servoing* en dos principales enfoques: basado en imagen y basado en posición.

En el enfoque basado en imagen o *IBVS* (*Image Based Visual Servoing*), los estados del robot se definen a partir de características extraídas directamente de la imagen (esquinas, momentos, bordes, etc.) mientras que en el enfoque basado en posición o *PBVS* (*Position Based Visual Servoing*), la información utilizada son parámetros 3D estimados a partir de mediciones realizadas en la imagen [23, 24]. En la Figura 3.8 se muestra una representación del funcionamiento de ambos enfoques resaltando los marcos de referencia y la ubicación de las características de los estados actual (R_c, s) y objetivo (R_{c^*}, s^*).

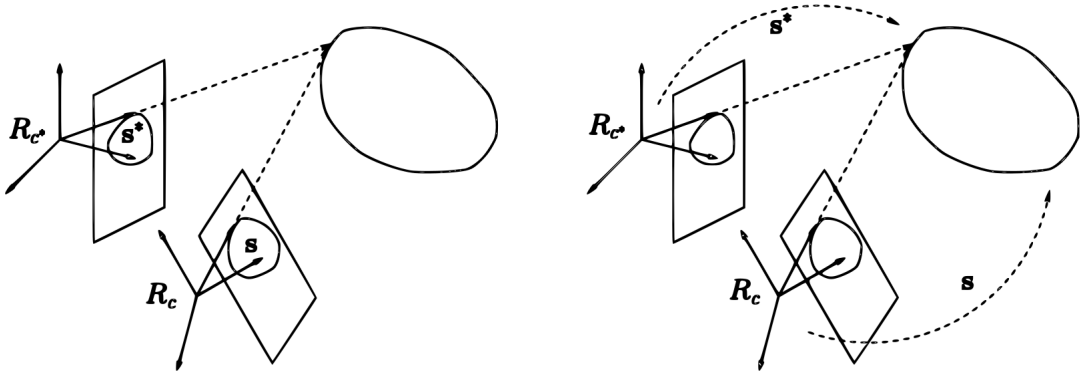


Figura 3.8: Enfoques *IBVS* (Izquierda) y *PBVS* (derecha). Adaptado de [3].

Image Based Visual Servoing (*IBVS*)

Como se menciono anteriormente, en *IBVS* los estados actual $\mathbf{s}$ y objetivo $\mathbf{s}^*$ del robot son descritos a partir de características extraídas directamente de la imagen. Esto significa que la ley de control utiliza únicamente información disponible en un plano bidimensional. Por tal razón, algunas veces se refiere a *IBVS* como *Visual Servoing* 2D.

El enfoque tradicional de *IBVS* utiliza como características, las coordenadas (en pixeles) de un grupo de puntos en la imagen. El error entonces, consiste en la posición actual de estos puntos, respecto a la ubicación objetivo.

El diagrama general de un sistema basado en *IBVS* es mostrado en la Figura 3.9.

Position Based Visual Servoing (*PBVS*)

En el enfoque *PBVS*, la ley de control se establece utilizando la pose en tres dimensiones del objetivo como referencia para determinar los movimientos del robot. La

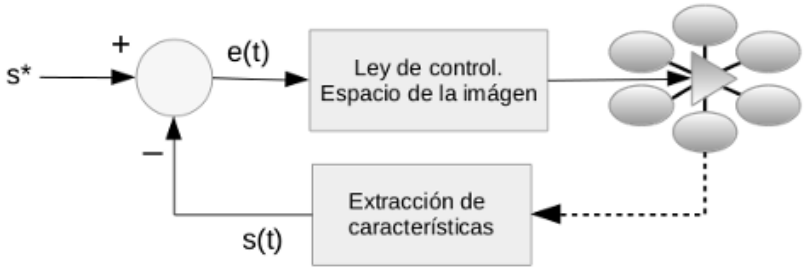


Figura 3.9: Sistema *IBVS*.

pose está formada por la rotación y traslación del objetivo respecto a la cámara y es determinada utilizando información procedente de mediciones en la imagen, realizadas en tiempo real e información del objetivo y la cámara, conocida de antemano. Un ejemplo de información que se requiere proveer al sistema es la matriz de calibración $\mathbf{K}$ de la cámara, la cual contiene información (los parámetros intrínsecos) que permiten determinar la relación *pixeles*-metros en una imagen. En referencia a lo anterior, es de suma importancia asegurar que la información adicional provista al sistema es correcta, ya que de lo contrario, podrían presentarse errores en la estimación de la pose y en consecuencia, en el control.

El funcionamiento de un sistema que opera bajo el enfoque *PBVS* consiste de una o varias cámaras montadas en el robot o en la escena, una etapa de extracción de características, un estimador de pose y un controlador el cual determina las acciones a realizar basándose en el error $\mathbf{e}(t)$ entre la pose actual $\mathbf{s}$ y la pose objetivo $\mathbf{s}^*$. Un diagrama simplificado de éste sistema es mostrado en la Figura 3.9.

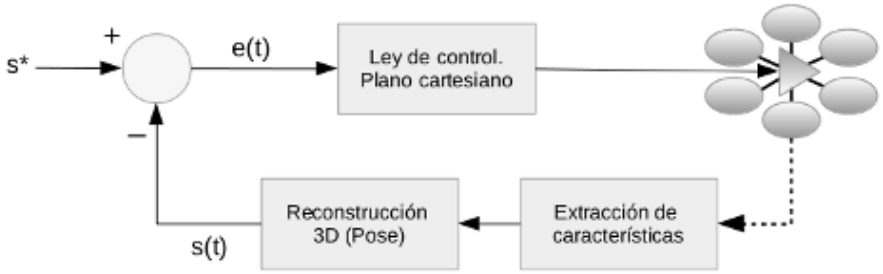


Figura 3.10: Sistema *PBVS*.

A este enfoque de control a veces se le refiere como *Visual Servoing* 3D, debido a que funciona con información tridimensional de la escena.

3.6. Gazebo

Gazebo es un simulador de entornos robóticos de código abierto desarrollado inicialmente por Andrew Howard y Nate Koenig en la Universidad de Carolina del Sur

(*University of Southern California*) [25] y administrado actualmente por la *Open Source Robotics Foundation* (*OSRF*), una organización independiente sin fines de lucro creada para dar soporte al desarrollo, distribución y adopción de de software de código abierto para uso en investigación, educación y desarrollo de productos robóticos [26].

Figura 3.11: Logotipo del simulador Gazebo. Tomado de [25].

Gazebo permite simular de forma precisa y eficiente entornos robóticos, incluyendo la dinámica y la física que rigen su comportamiento. Las simulaciones en Gazebo pueden ser complementadas a través del uso de sensores virtuales, lo que permite a los robots interactuar y adaptarse al entorno. Esto también posibilita observar su comportamiento bajo distintas condiciones. En referencia a lo anterior, Gazebo posee herramientas para la construcción de entornos interiores y exteriores, lo que ofrece un mayor realismo a la simulación. Casas, edificios, obstáculos e inclusive cambios de iluminación y viento, son solo algunos ejemplos de construcciones y condiciones que pueden ser modelados y simulados en Gazebo.

Otra de las características de Gazebo es su soporte a *plugins*. Los *plugins* son piezas de código que pueden ser agregados a una simulación y permiten tener acceso a las funcionalidades de Gazebo a través de clases estándar de *C++* [27]. Esto último es aprovechado por los desarrolladores para enlazar bibliotecas externas a Gazebo, lo que permite extender las capacidades de simulación del programa. Como resultado, pueden crearse entornos robóticos más complejos, que involucren, por ejemplo, técnicas de visión computacional e inteligencia artificial.

3.7. *Robotic Operative System* (ROS)

ROS (del inglés, *Robot Operating System*) es un *framework* flexible y de código abierto administrado por la Open Source Robotics (OSRF) [26]. Los inicios de ROS se remontan a mediados de los 2000 en la Universidad de Stanford (*Stanford University*), con el desarrollo de los programas *STanford AI Robot (STAIR)* y *Personal Robots (PR)*, en donde se crearon prototipos de sistemas de software flexibles y dinámicos, destinados a robótica. Sin embargo, a lo largo de los años, su desarrollo se ha llevado a cabo por distintas instituciones [28]. Hoy en día, ROS es uno de los *frameworks* más populares en el área de la robótica.

Figura 3.12: Logotipo del proyecto ROS. Tomado de [29].

ROS, a pesar de su nombre, no es un sistema operativo convencional sino más bien un sistema meta-operativo el cual provee servicios de abstracción de hardware, control de dispositivos de bajo nivel, intercambio de mensajes entre procesos y gestor de paquetes, así como herramientas y bibliotecas para la obtención, construcción, escritura y ejecución de código en múltiples computadoras [30].

3.7.1. Terminología

Para comprender de mejor manera temas relacionados con ROS, es necesario conocer de antemano, algunos conceptos importantes:

- ***Package***. Los *packages* o *paquetes* son la unidad principal de organización de software en ROS. Un *package* puede contener procesos (*nodes*), bibliotecas, *datasets*, archivos de configuración y cualquier otro tipo de información que desee organizarse.

- ***Metapackage***. Los *metapackages* o *metapaquetes* son un tipo especial de *packages* que almacenan la dependencia de *packages*, pertenecientes a un mismo grupo.

- ***catkin***. *catkin* es el sistema oficial de construcción utilizado en ROS, encargado de automatizar el proceso de compilación y manejar las dependencias entre *packages*.

- ***Node***. Los *nodes* o *nodos* son procesos que realizan una tarea específica. Por ejemplo, controlar la velocidad de un motor o mostrar la lectura de un sensor.

- ***Master***. El *master* o *maestro* es el encargado de registrar los *nodes*, *services* y *topics* en ejecución. Además de permitir que los *nodes* se localicen durante la comunicación.

- ***Parameter Server***. El *Parameter Server* o *servidor de parámetros* es un elemento del *Master* que permite almacenar datos por llave (*key*) en una ubicación central.

- ***Messages***. Los *messages* o mensajes son una estructura de datos formada por campos utilizada para comunicarse entre *nodes*. Un *message* puede contener tipos primitivos (enteros, flotantes, booleanos, etc.), arrays de tipos primitivos e inclusive estructuras *message* anidadas.

- ***Topics***. Un *topic* o *tópico* es el nombre utilizado para identificar el contenido de un mensaje.
- ***Services***. Un *service* o servicio es un sistema de comunicación del tipo cliente/servidor compuesto por dos *messages*: uno para la solicitud y otro para la respuesta.
- ***Bags***. Las *bags* o *bolsas* son contenedores que almacenan los *messages* enviados por algún *node* y opcionalmente, un *timestamp* de cuando fue generado. Su principal uso es con fines de depuración.
- ***Distribution***. Las *distributions* o *distribuciones* ROS son un conjunto de *packages* estables, con un número y nombre de versión asignados, puestos a disposición del público en general para el desarrollo de aplicaciones robóticas.

3.7.2. Modelo de comunicación

El modelo de comunicación en ROS consiste en el intercambio de *mensajes* entre *nodos*. La forma más básica de establecer establecer comunicación entre ellos es utilizando *tópicos*. La comunicación a través de tópicos se logra *suscribiendo* un *nodo* cliente a un *tópico* de interés. El *nodo* servidor entonces, enviará un *mensaje* a los *nodos* que se encuentren *suscritos* a él. En este caso, la comunicación es unidireccional.

Una segunda alternativa de comunicación consiste en el uso de *servicios*. Los *servicios* proveen un mecanismo de solicitud/respuesta (*request/reply*) definido por dos mensajes: uno para la solicitud y otro para la respuesta. Bajo este esquema, un *nodo proveedor* ofrece un *servicio* y queda en espera de un *mensaje* de solicitud. Cuando algún *nodo* cliente solicita el *servicio*, éste queda en espera hasta recibir la respuesta por parte del *nodo* servidor. El *nodo* servidor entonces, envía el *mensaje* respuesta y queda a la escucha de nueva solicitudes.

Ambos esquemas de funcionamiento son ejemplificados en la Figura 3.13.

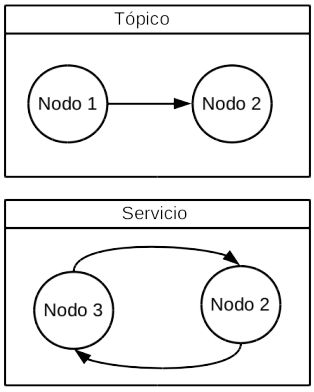


Figura 3.13: Esquemas de comunicación en ROS.

Capítulo 4

Metodología

4.1. Planteamiento del problema utilizando *Visual Servoing*

De forma general, el problema de aterrizaje de un VANT puede ser visto como un problema de cambio de estados, en el cual se requiere llevar a la aeronave de un estado inicial, en vuelo, a un estado objetivo, en tierra. Desde la perspectiva de *Visual Servoing*, existen dos posibles alternativas para describir los estados del VANT: la primera consiste en utilizar características de referencia en la imagen (*IBVS*) y la segunda en usar la pose del VANT (*PBVS*). Analizando el problema de aterrizaje, la intención es dirigir a la aeronave desde su posición actual hacia la posición del marcador, por tanto, la opción de control más adecuada es el enfoque *PBVS*.

Para conocer la posición del VANT, se propuso la utilización de los parámetros extrínsecos de la cámara, estimados a partir del marco de referencia creado por un marcador en tierra (ver Figura 4.1). Para que esto funcione, se asumió que la cámara se encontraba montada fijamente en el centro del VANT y que sus movimientos dependían totalmente del desplazamiento de este último. Bajo estas condiciones, fue posible asegurar que la pose de la cámara era prácticamente igual al de la aeronave. Por tanto, la información de rotación y traslación de la cámara fue utilizada directamente en la etapa de control.

Tomando esto en consideración, la función general de *Visual Servoing* (ecuación 3.18) fue reescrita como sigue:

$$\mathbf{e}(t) = \zeta(\mathbf{R}, \mathbf{T}\ ;\ \alpha, \mathbf{K}, \mathbf{D}) - \zeta^*, \tag{4.1}$$

donde ζ corresponde a la pose actual compuesta por la rotación $\mathbf{R}$ y traslación $\mathbf{T}$ actual de la cámara, α información de las dimensiones del marcador en metros, $\mathbf{K}$ la matriz de calibración , $\mathbf{D}$ los coeficientes de distorsión y ζ^* la pose objetivo.

Los coeficientes de distorsión no influyen directamente en la estimación de la pose, sin embargo, son útiles para incrementar el rendimiento de los algoritmos de detección

y estimación, es por ello que son proporcionados como información adicional junto a las dimensiones del marcador.

La pose objetivo ζ^* indicada en la ecuación 4.1 viene dada por el caso en donde los orígenes de los marcos de referencia de la cámara y el marcador coinciden o lo que es igual, en donde no existe traslación y rotación entre ambos, es decir:

$$\zeta^*(\mathbf{R}^*, \mathbf{t}^*) \implies \mathbf{R}^* = I \ , \ \mathbf{t}^* = (0, 0, 0),$$

donde I corresponde a una matriz identidad de 3×3.

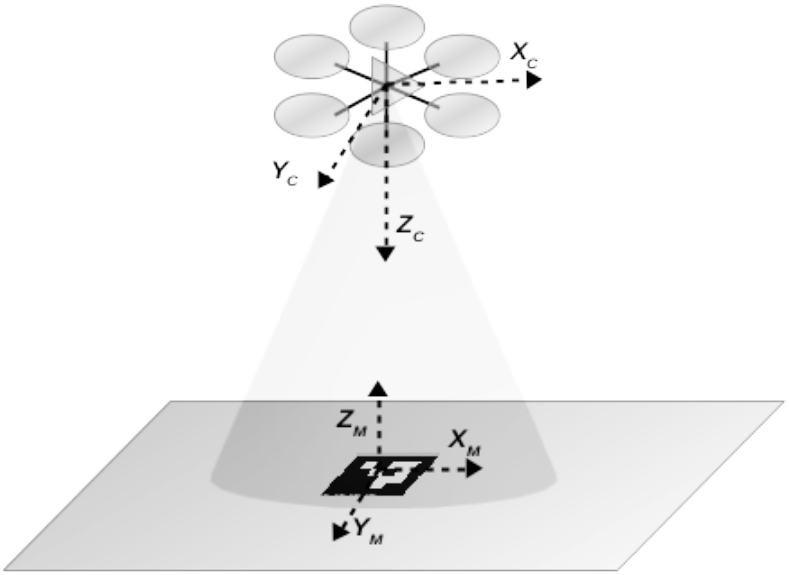


Figura 4.1: Marcos de referencia para el aterrizaje.

4.2. Definición de un marcador visual

Una opción para asegurar un correcto aterrizaje es utilizar un marcador visual que indique la ubicación objetivo. Este marcador debe ser lo suficientemente distintivo como para permitir que sea identificado inequívocamente por el VANT. Además, debe ser de rápida detección para lograr que los algoritmos de control funcionen en tiempo real. La solución utilizada en el presente trabajo consistió del uso de marcadores de ArUco [31], los cuales consisten de cuadrados de color negro y blanco con un identificador numérico codificado. Su elección se debió principalmente a la robustez que han demostrado para su detección, la capacidad de estimar la pose de la cámara a partir de ellos [4] y la disponibilidad de bibliotecas compatibles con OpenCV [32], la herramienta de Visión Computacional elegida para el presente proyecto. El procedimiento de instalación para ambas bibliotecas se describe en el Apéndice A.

ID	Tamaño	x	y
16	5.0	0.0	0.0
32	15.0	-5.0	12.5
64	50.0	3.1	0.0
88	100.0	-5.0	-5.5

Tabla 4.1: Información de los marcadores de ArUco en centímetros.

El marcador de aterrizaje se formó a partir de cuatro marcadores de ArUco con distintas dimensiones, distribuido como se muestra en la Figura 4.2. La distribución y escala elegida fue pensada para permitir que que al menos un marcador sea detectado a pesar del cambio de altitud del VANT e incrementar la robustez durante la etapa de estimación de pose, como se mostrará en la Sección 4.5. El origen del marcador de aterrizaje se estableció en el centro del marcador más pequeño, esto significó que la distancia de los demás marcadores se indicó a partir de dicha posición. En la Tabla 4.1 se resume la información relevante de los marcadores como lo son: su identificador, sus dimensiones en metros y la distancia que hay respecto al origen en el marcador central.

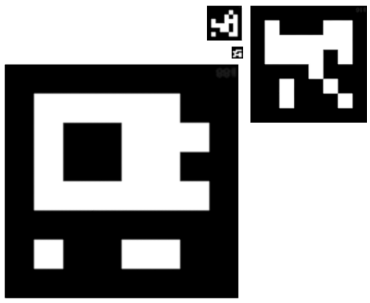


Figura 4.2: Marcador de aterrizaje.

4.3. Creación del marcador visual

La creación del marcador se realizó utilizando la la función de utilería *aruco_print_marker*, provista por la biblioteca ArUco, que implementa el algoritmo propuesto en [4]. La instrucción utilizada fue la siguiente:

```
./aruco_print_marker id marker.png -bs 21
```

donde *id* corresponde al identificador deseado, *marker.png* el nombre de la imagen de salida y *21* las dimensiones de ésta última. La dimensión utilizada fue 21 que corresponde a un marcador de 128×128 *pixeles*.

En principio todos los marcadores fueron creados con las mismas dimensiones, sin embargo, posteriormente estos fueron escalados a sus dimensiones reales y ordenados para integrarlos a la simulación. El procedimiento seguido para lograrlo se detalla en

el Apéndice C.

4.4. Identificación del marcador

La identificación del marcador es parte esencial para la tarea de aterrizaje, ya que permite conocer la ubicación exacta del punto en donde se desea que el VANT descienda. Esta tarea consiste de un proceso en dos partes: la fase de detección y la fase de discriminación. Durante la fase de detección, imágenes de la escena son capturadas y analizadas para encontrar posibles objetos candidatos a marcadores de ArUco. Posteriormente, en la fase de discriminación, los candidatos son revisados para determinar si realmente corresponden a algún marcador válido o simplemente son objetos erróneamente detectados. Esta tarea fue lograda utilizando la función *detect()* de la biblioteca ArUco, que implementa los algoritmos mostrados en [4]. El resultado del proceso de identificación del marcador de aterrizaje es mostrado en la Figura 4.3. En resumen, a una imagen le es aplicado un proceso de umbralización y búsqueda de contornos que permite encontrar los bordes de objetos en la escena. Posteriormente, una aproximación de polinomios es realizada para encontrar figuras que asemejen cuadrados. Después, una etapa de filtrado es aplicada para eliminar los polinomios no coincidentes. Finalmente una corrección de perspectiva es realizada y un proceso es iniciado para hallar el identificador numérico del marcador. Dicho proceso se resume en la Figura 4.4.

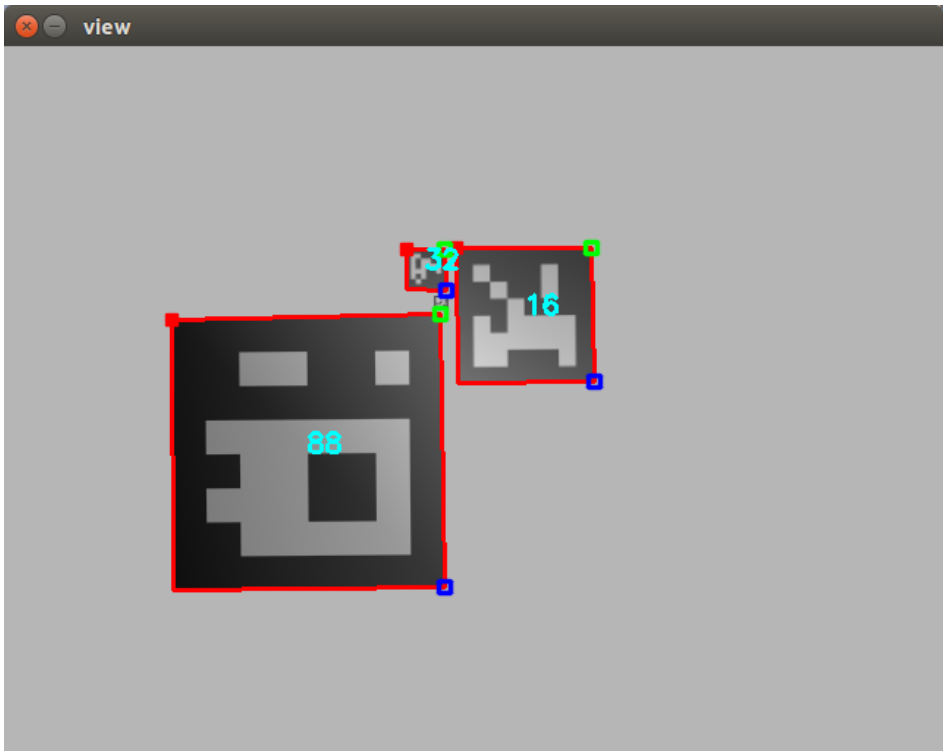


Figura 4.3: Marcadores de ArUco detectados.

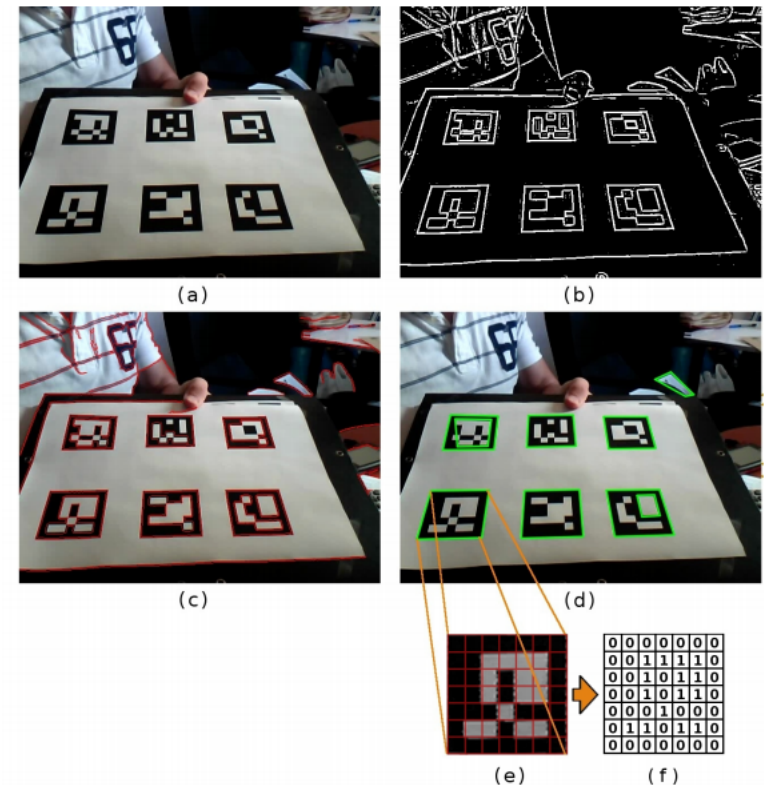


Figura 4.4: Procedimiento de identificación de Marcadores. a) Imagen Original, b) Umbralización, c) Detección de contornos, d) Aproximación de polinomios, e) Corrección de perspectiva, f) Hallado del identificador. Tomado de [4].

4.5. Estimación de la pose

El problema de estimación de la pose consiste en encontrar la rotación y traslación objetivo que permitan el aterrizaje del VANT. Como se mencionó inicialmente, para resolver este problema se utilizaron los parámetros extrínsecos de la cámara, los cuales consisten de la rotación de esta última respecto al plano del mundo real. Para su obtención, se utilizó la función *solvePnP()* [33] disponible en la biblioteca OpenCV, la cual calcula los componentes de la pose utilizando la correspondencias entre puntos en un plano tridimensional y su equivalente en un plano bidimensional, problema que se le conoce como *perspective-n-point* (ver Figura 4.5). En el proyecto, dicha correspondencia consistió de las coordenadas de las esquinas de los marcadores en el mundo (3D, metros) y las de su proyección en la imagen (2D, *pixeles*).

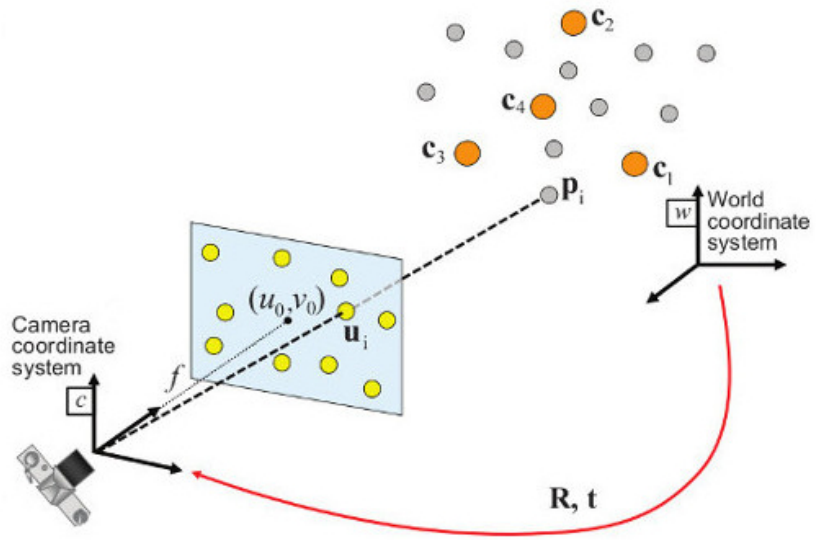


Figura 4.5: Problema Perspectiva-n-Punto. Tomado de [5].

La herramienta utilizada por *solvePnP()* para poder estimar la pose es la homografía entre el plano de la cámara y el plano del mundo real, la cual se obtiene asumiendo que el componente en Z del marcador es igual a cero, lo que permite simplificar la ecuación (3.15) como sigue:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \begin{bmatrix} r_1 & r_2 & t \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}.$$

Lo que resulta en una transformación originalmente de $3D \rightarrow 2D$ a un mapeo de $2D \rightarrow 2D$, misma que suele denominarse homografía H:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = H \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}.$$

Al estar los puntos expresadas en coordenadas homogéneas, esto significa que la transformación puede estar sometida a un cambio de escala, lo cual suele indicarse mediante un factor λ, como se muestra a continuación:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \lambda H \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}.$$

Finalmente, para desplazar los puntos de la imagen al marco de referencia de la cámara, se multiplican ambos lados de la ecuación por la inversa de la matriz $\mathbf{K}$, lo que resulta en lo siguiente:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = H_\lambda \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}. \tag{4.2}$$

La función *solvePnP()* estima la homografía H_λ, la cual consiste de la homografía escalada por un factor λ y posteriormente la normaliza y descompone para obtener los valores de rotación y traslación de la pose. En resumen, un proceso para lograrlo es el siguiente:

De la ecuación (4.2) obtenemos que la homografía se encuentra escalada por el factor α, para eliminarlo es aprovechada la propiedad de ortonormalidad de las matrices de rotación la cual indica que cada columna de la matriz consiste de un vector unitario ortogonal a los otros dos. A sabiendas de esto, podemos estimar λ obteniendo la norma actual de r_1 o r_2, normalizar en base a ello y obtener la tercera columna de la matriz de rotación mediante el producto cruz:

$$\frac{1}{||r_1||} H_\lambda = \frac{1}{||r_1||} \begin{bmatrix} r_1 & r_2 & t \end{bmatrix},$$

$$H = \begin{bmatrix} r_1 & r_2 & (r_1 \times r_2) & t \end{bmatrix}.$$

Por último, como se resalta en [34], el proceso mostrado anteriormente no asegura que la matriz de rotación sea ortogonal, para ello es necesario aplicar un proceso de descomposición SVD a la matriz de rotación.

$$R = \begin{bmatrix} r_1 & r_2 & r_3 \end{bmatrix},$$
$$\text{SVD}(R,\ W,\ U,\ V),$$
$$R = U \cdot V\ .$$

4.6. Ángulos de Euler

La función *solvePnP()* estima la rotación y traslación de la pose y la devuelve como vectores independientes. Aunque la traslación puede ser usada directamente, la rotación requiere ser transformada a su representación de matriz para poder extraer los ángulos

de Euler. Esta transformación se realizó mediante la función *cv::Rodrigues()* [35] de OpenCV, la cual implementa un algoritmo destinado para dicho fin. Habiendo obtenido la matriz de rotación, los ángulos de Euler fueron calculados utilizando las ecuaciones mostradas en 3.11, mismas que se relacionan con el diagrama mostrado en la Figura 3.1. Para el control, solo el componente en Z (ψ_{yaw}) fue utilizado, esto fue posible asumiendo y asegurando que los desplazamientos del VANT en los ejes X y Y fueran tan *suaves*, que los cambios en los ángulos *pitch* y *roll* puedan ser despreciados. Considerando lo antes mencionado, la pose del VANT entonces se simplificó a lo siguiente:

$$\zeta_c = (\psi, t_x, t_y, t_z), \tag{4.3}$$

Siendo ésta la pose utilizada para representar la posición del VANT respecto al marcador, siempre y cuando se satisfagan las condiciones antes mencionadas. Como ejemplo, en la parte inferior derecha de la Figura 4.6, son mostrados los componentes de la pose de la cámara estimados a partir del marcador de ArUco 88.

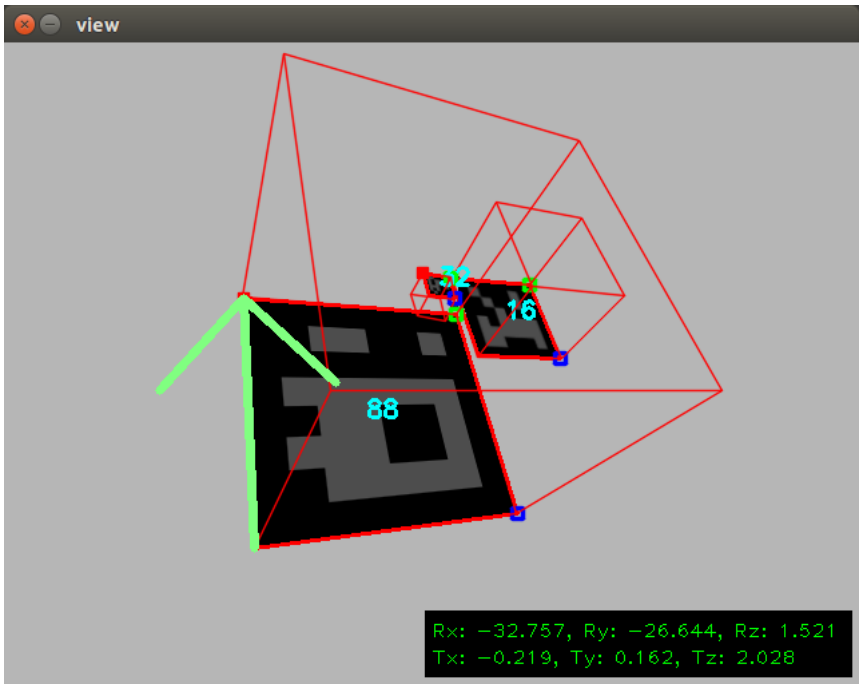


Figura 4.6: Componentes de la pose respecto al marcador 88.

4.7. Definición de la ley de control

La ley de control permite determinar la salida en cada instante de tiempo que lleve a un sistema a satisfacer una tarea concreta. En el problema de aterrizaje, ésta salida consiste en las instrucciones requeridas para que el VANT corrija su posición y lo lleve a descender sobre el marcador de aterrizaje. Estas instrucciones pueden variar con el tiempo, dependiendo de la magnitud de error que exista entre la posición actual de

la aeronave y la objetivo. El valor de este error puede ser obtenido combinando la ecuaciones (4.1) y (4.3), lo cual da como resultado:

$$\mathbf{e}(t) = \zeta(\psi, t_x, t_y, t_z \ ; \ \alpha, \mathbf{K}, \mathbf{D}) - \zeta^*(0, 0, 0, 0). \tag{4.4}$$

Como puede observarse en la ecuación anterior, el error de pose es igual a la posición actual del VANT. Por tanto, obviando los elementos adicionales de la ecuación (4.4), la expresión se redujo a:

$$\mathbf{e}(t) = (\psi, t_x, t_y, t_z).$$

Para corregir la pose y minimizar este error, se utilizó un mecanismo que permitió sobrescribir los valores de los canales *rc* del VANT. Haciendo esto, fue posible controlar la dirección y velocidad de la aeronave, dependiendo de la magnitud asignada al canal.

El controlador utilizado consistió de un PID clásico [36], descrito por la ecuación mostrada a continuación:

$$u(t) = K_p e(t) + K_i \int_0^t e(t')dt' + K_d \frac{de(t)}{dt}, \tag{4.5}$$

donde K_p, K_i, K_d, son constantes que se relacionan con el efecto proporcional, integral y derivativo del controlador PID, respectivamente.

Este valor de salida $u(t)$, sin embargo, no fue utilizado directamente en el proceso de aterrizaje. La entrada de un canal no debe exceder los valores mínimos y máximos *rc*, 1000 y 2000, respectivamente. Por tanto, es necesario asegurar que no se superen dichos valores. Además el control de velocidad se realiza variando el valor *rc* del canal partiendo del valor base (1500), provocando que un valor inferior a él desplace el VANT en sentido negativo y un valor superior lo haga en sentido positivo, con un empuje proporcional a su magnitud. Como consecuencia, fue necesario establecer una regla de normalización que adecuara la salida $u(t)$ a las condiciones antes mencionadas:

$$\Lambda_{rc} = \lambda(u(t)),$$

donde λ corresponde a la regla que relaciona el error actual con las velocidades *rc* requeridas y Λ_{rc} es el vector resultante.

La regla $\lambda : \mathbb{R} \times \mathbb{N} \to \mathbb{R}$ consistió de una función que mapea el valor de salida del PID en valores *rc*, considerando la corrección a partir del valor *rc* base (1500) y un valor α el cual corresponde al *offset* máximo a partir del valor base. Este valor α, establecido por la función $f : \mathbb{N} \to \mathbb{N}$, además de evitar el desbordamiento del valor *rc*, permitió definir una velocidad máxima del VANT durante el ajuste de su posición. Tomando en

consideración lo antes mencionado, la regla de control λ resultó en lo siguiente:

$$f(\alpha) = \begin{cases} 1, & \alpha < 1 \\ 500, & \alpha > 500 \\ \alpha, & (1 \leq \alpha \leq 500). \end{cases}$$

$$rc_{min} = 1500 - f(\alpha)$$

$$rc_{max} = 1500 + f(\alpha)$$

$$\lambda\big(u(t), f(\alpha)\big) = \begin{cases} rc_{max}, & u(t) > rc_{max}, \\ rc_{min}, & u(t) < rc_{min}, \\ u(t), & rc_{min} \leq u(t) \leq rc_{max}). \end{cases} \tag{4.6}$$

Por último, el vector de salida Λ_{rc} se creó a partir de los elementos de velocidad rc obtenidos aplicando λ a la salida del PID:

$$\Lambda_{rc} = \begin{bmatrix} {}_{rc}\dot{\psi} & {}_{rc}\dot{t_x} & {}_{rc}\dot{t_y} & {}_{rc}t_z \end{bmatrix}^T.$$

Nótese que en la implementación, un controlador PID por componente de la pose fue necesario, a excepción del control de velocidad de la altitud, que se estableció un valor constante de velocidad durante el aterrizaje. Es por ello que el valor t_z no es expresado como una velocidad en el vector Λ_{rc}.

4.8. Planteamiento del algoritmo de aterrizaje

El desarrollo de un algoritmo eficiente y robusto fue necesario para asegurar el aterrizaje del VANT, cuidando la mayor precisión posible y un descenso controlado. Para ello, se utilizaron todas las herramientas antes mencionadas: detección de marcadores, estimación de la pose y obtención de ángulos de Euler, así como técnicas para incrementar la robustez del algoritmo. A continuación, se expondrán el esquema del algoritmo planteado.

4.8.1. Identificación de los marcadores de aterrizaje

El proceso de aterrizaje requiere contar con la mayor precisión posible, es por ello que el marcador de aterrizaje mostrado en la Figura 4.2 fue propuesto. Su intención fue permitir que al menos un marcador de ArUco sea detectado a distintas altitudes. El proceso de detección fue delegado a la función *detect*() de la biblioteca ArUco, la cual devuelve todos los posibles marcadores de ArUco identificados en una imagen, inclusive aquellos que no corresponden al marcador de aterrizaje. Por esta razón, parte del algoritmo consistió en evaluar si alguno de ellos está relacionado con el marcador de aterrizaje. Para lograrlo, una lista con los identificadores esperados fue almacenado en el código del programa. Tras un proceso de identificación de marcadores, los identificados fueron cotejados con esta lista, conservando solo aquellos que coincidieran con alguno

de los marcadores deseados.

$$\{m\} : \{\text{detected}\} \cap \{\text{Markers}\},$$

donde m corresponde al conjunto de marcadores del indicador detectados, *detected* a todos los marcadores detectados en la imagen, *Markers* el conjunto de marcadores válidos (88, 64, 32, 16) del indicador de aterrizaje y $\cap$ el operador de intersección.

4.8.2. Estimación de la pose

La estimación de la pose del VANT consiste en determinar la traslación y rotación que existe respecto al indicador de aterrizaje, necesaria para determinar las instrucciones de control que lleven al VANT a su aterrizaje. La forma de hacerlo fue utilizando la función *solvePnP()* de OpenCV, que dada las coordenadas de un objeto en la imagen (en *pixeles*) y su equivalente en la escena (en metros), estima la rotación y la traslación de la cámara. Esta correspondencia entre puntos se logró utilizando las coordenadas de la esquinas de los marcadores y sus equivalentes en el marco de referencia del mundo, generadas a partir de las dimensiones del marcador y su distancia hacia el marcador principal (número 16), siendo estas últimas las conocidas de antemano. Utilizando la información de la Tabla 4.1 y realizando los cálculos pertinentes, las coordenadas expresadas en el marco de referencia del mundo tridimensional son las mostradas en la Tabla 4.2.

ID	x	y	z
88	-1.0	-0.05	0.0
	0.0	-0.05	5.0
	0.0	-1.05	0.0
	-1.0	-1.05	0.0
64	0.06	0.25	0.0
	0.56	0.25	0.0
	0.56	-0.25	0.0
	-0.06	-0.25	0.0
32	-0.125	0.2	0.0
	0.025	0.2	0.0
	0.025	0.5	0.0
	-0.125	-0.05	0.0
16	-0.025	0.025	0.0
	0.025	0.025	0.0
	0.025	-0.025	0.0
	-0.025	-0.025	0.0

Tabla 4.2: Coordenadas de las esquinas de los marcadores (metros).

4.8.3. Altura y margen de aterrizaje

Para incrementar la robustez durante el aterrizaje, la altura (componente tz de la pose) fue considerada para establecer un margen de aterrizaje mínimo, en donde una vez que el VANT se encuentre dentro de dicha área, el proceso de aterrizaje sea iniciado. Dicho margen decrece a medida que la altura del VANT desciende y es calculado utilizando la ecuación $m : \mathbb{R}^5 \to \mathbb{R}$ mostrada a continuación:

$$m(\epsilon, h_c, h_{max}, h_{min}, m_{max}, m_{min}) = \begin{cases} m_{max}, & h_c > h_{max} \\ m_{min}, & h_c < h_{min} \\ m_{max}, & \epsilon \cdot h_c > m_{max} \\ m_{min}, & \epsilon \cdot h_c < m_{min} \\ \epsilon \cdot h_c, & m_{min} \leq \epsilon \cdot h_c \leq m_{max}. \end{cases} \tag{4.7}$$

donde ϵ es el factor de decremento, h la altura actual en metros, h_{max} la altura máxima a considerar, h_{min} la altura mínima, m_{max} el margen de error máximo para el aterrizaje y m_{min} el margen mínimo.

4.8.4. Algoritmo del PID

Para controlar los movimientos del VANT, se implementó un controlador PID (ver ecuación 4.5) para determinar automáticamente la magnitud y dirección de las correcciones que llevaran a la aeronave a la pose objetivo. En su versión digital, el controlador puede ser implementado siguiendo el pseudocódigo mostrado en el Algoritmo 4.1.

```
1  error = 0.0 - setpoint
2  integral = integral + error * dt
3  derivate = (error - prev_error) / dt
4  out = (kp * error) + (kix * int) + (kd * derivate)
5  prev_error = error
```

Algoritmo 4.1: Algoritmo PID

La obtención de las constantes Kp, Ki y Kd adecuadas para el control, fueron obtenidas mediante experimentación,, variando sus valores y observando el comportamiento del VANT hasta alcanzar una estabilidad y velocidad. De igual manera, para asegurar la fiabilidad y tiempo de actualización de las correcciones, se estableció un base de tiempo dt constante, permitiendo su funcionamiento en tiempo real.

En la implementación del algoritmo final, fue utilizado un PID para cada elemento de la pose, a excepción del componente t_z que se mantuvo a un valor constante durante el proceso de aterrizaje. Los valores de salida de los PID, fueron normalizados utilizando la regla mostrada en la ecuación (4.7) para asegurar que sean consistentes con el método de control basado en sobreescritura de canales rc.

4.8.5. Proceso de aterrizaje

El proceso de aterrizaje consistió en descender el VANT a una velocidad fija cuando se detecta que la aeronave se encuentra dentro de un margen establecido por la ecuación (4.7). En caso contrario, la altura es mantenida hasta que la pose es corregida. Este proceso es controlado por una función $l : \mathbb{R}^3 \to \mathbb{R}$ que determina el estado actual de aterrizaje basado en los componentes de traslación de la pose actual y el margen de error requerido. Dicha función es la siguiente:

$$l\big(t_x, t_y, m(\cdot)\big) = \begin{cases} \text{Iniciar}, & (t_x \leq m(\cdot) \text{ y } t_y \leq m(\cdot)) \\ \text{Detener}, & \text{de otra manera.} \end{cases}$$

El estado *Iniciar* de la función $l(\cdot)$ consiste en utilizar los valores de salida del PID para corregir la pose y mantener la velocidad de aterrizaje fija, mientras que un estado *Detener* fija el *throttle* del VANT a su valor de estabilidad (1500) para detener el descenso.

4.9. Pseudocódigo

El pseudocódigo del proceso general de aterrizaje implementado en el proyecto es mostrado en el Algoritmo 4.2. Este fue la base para el desarrollo del algoritmo principal utilizado para la validación de la corrección de la pose. Intencionalmente, el pseudocódigo no involucra ningún aspecto relacionado con las bibliotecas OpenCV y/o ROS, esto para conservar la idea general y no concentrarse en detalles de implementación correspondientes a dichas bibliotecas.

```
image = getImage()
AllMarkers = detectMarkers(image)
Markers = AllMarkers ∧ ValidMarkers
n = Markers.lenght()
validMarkers = {}
margin = 0

if n > 0:
  validMarkers = getValidMarkers(Markers)
  [R, T] = computeExtrinsics(validMarkers)

  angles = computeEulersAngles(R)
  margin = m(ϵ, h_c, h_max, h_min, m_max, m_min)
  out = PID(angles.yaw, x, y)

  if l(T.t_x, T.t_y, margin) == Iniciar:
    rc_out.z = 1400
  else:
    rc_out.z = 1500

  rc_out[yaw,x,y] = convertToRC(out[angles.yaw,T.t_x,T.t_y])
else:
  rc_out = 1500

update_rc(rc_out)
```

Algoritmo 4.2: Pseudocódigo del algoritmo general.

Capítulo 5

Implementación

5.1. Introducción

Para validar el funcionamiento del algoritmo mostrado en el Algoritmo 4.2, se adaptó el modelo de simulación de un *Erle-copter*, un cuadrucóptero desarrollado y comercializado por la compañía *Erle Robotics*, la cual provee herramientas para su simulación en el software Gazebo. Se eligió esta opción, debido a que al igual que su contraparte real, el *Erle-copter* utiliza para su funcionamiento la herramienta ROS, un sistema operativo ampliamente utilizado en robótica, el cual permite desarrollar aplicaciones robustas en robots.

ROS brinda una capa de abstracción de hardware mediante la cual es posible acceder a elementos del robot, tales como sensores, cámaras y actuadores, a través interfaces de software simples. Esta abstracción fue la aprovechada para la implementación del algoritmo de aterrizaje en el VANT.

El programa está formado por dos partes, una en donde implementa la lógica para el aterrizaje del VANT y otra en donde se muestra la información visual al usuario. Esta distribución se realizó para evitar que el rendimiento del algoritmo de aterrizaje se vea afectado por las tareas de visualización. El funcionamiento general de este enfoque es mostrado en la Figura 5.1.

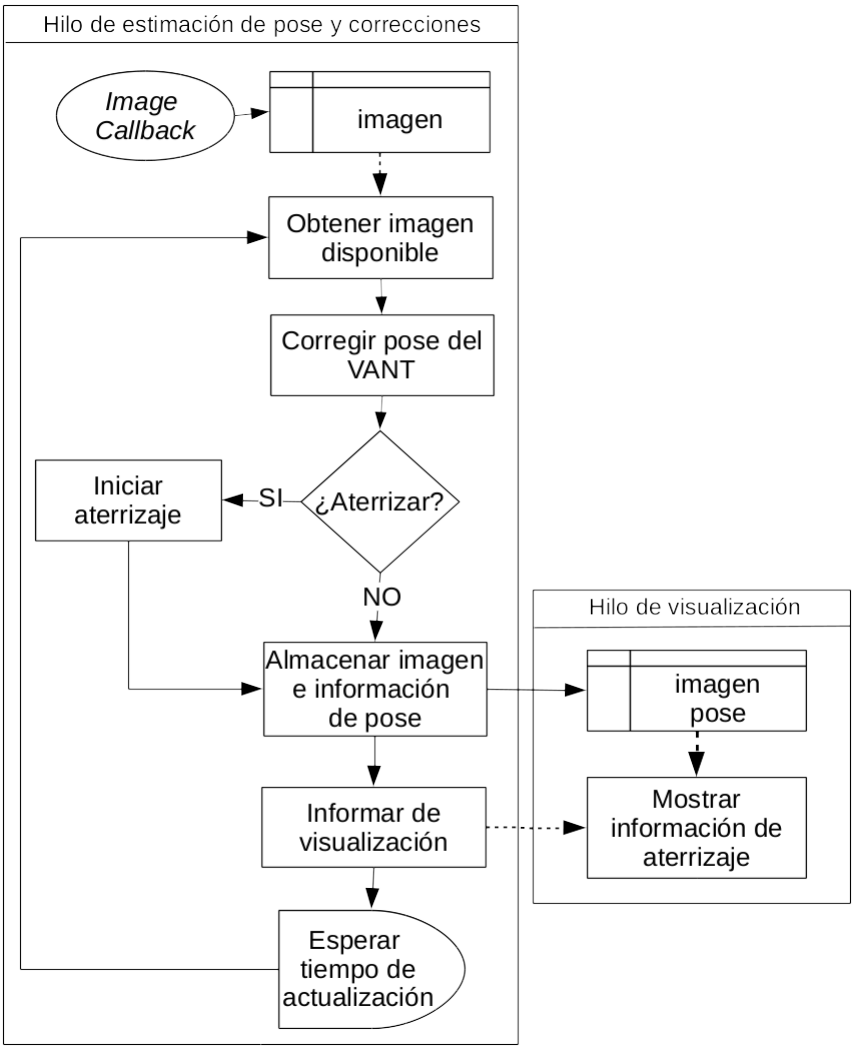


Figura 5.1: Diagrama de flujo del programa.

5.1.1. Código básico de ROS

La comunicación en ROS se da a partir de un esquema *cliente-servidor*, en donde un *nodo* envía información y el otro la recibe. En la terminología ROS, a la entidad que transmite información se le denomina *nodo publicador* y el que la obtiene, *nodo suscriptor*. Esta designación se deriva de la primicia que la información del *nodo publicador* es obtenida a través de un proceso de *suscripción*, en donde el *nodo* cliente indica el tópico especifico que desea escuchar. En principio, una programa muy básico de este tipo puede consistir del mostrado en el Algoritmo 5.1.

```
void callBack(const example::GenericData &data)
{
  // Hacer algo con data
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "image_listener");
    ros::NodeHandle nh;
    ros::Subscriber sub_generic;

  sub_generic = nh.subscribe("/example/generic", 1 ,callBack);
  ros::spin();
}
```

Algoritmo 5.1: Código Básico en ROS.

Explicación del código

El primer elemento, la función *ros::init(argc, argv, "image_listener")*, es la encargada de inicializar los elementos del nodo *image_listener*, y debe ser llamada antes de utilizar cualquier otra función de ROS. El hacer lo antes mencionado no inicia el *nodo*, simplemente lo configura. Esta tarea se delega a un manejador de nodos, *ros::NodeHandle nh*), el cual permite iniciarlo y detenerlo. Posteriormente, un objeto *ros::Suscriber* es creado, el cual permite suscribirse a un tópico ROS. La suscripción se realiza a llamando a la función *suscribe* del manejador de *nodos*, a la cual se le indica el nombre del tópico deseado, el tamaño de la *pila* de recepción y la función que es invocada cuando un mensaje es recibido. Finalmente, llamando *ros::spin()*, el nodo entra en un bucle hasta que una instrucción de *finalización* es recibida. Además, ROS espera el llamado a esta función para procesar los *callback* en el programa.

5.2. Hilo de estimación y corrección de pose

El punto central del proceso de aterrizaje consiste en la detección del marcador de aterrizaje, esto significa que es necesario obtener imágenes de la cámara en *tiempo real*, es decir, a una velocidad adecuada que permita al algoritmo estimar la pose y realizar las tareas de control con la suficiente velocidad para que el VANT sea capaz de estabilizarse. Sin embargo, a pesar que existe un tópico que *publica* una imagen de la cámara a una frecuencia determinada, en la práctica este valor no es contante. Por tal razón, fue necesario implementar una técnica que permitiera realizar dicho proceso en un tiempo constante. La solución utilizada consistió en delegar el algoritmo de estimación y corrección de posición (ver Algoritmo 4.2) a un *hilo* que no dependiera de la recepción de la imagen, y establecer un retardo para su ejecución. Los detalles relacionados con su implementación serán explicados a continuación.

5.2.1. Obtención de imagen de la cámara

Parte esencial para el funcionamiento del algoritmo es poder acceder a las imágenes capturadas por la cámara inferior del VANT. Para dicho fin, el VANT posee un tópico que publica las imágenes a una frecuencia determinada[1]. Este tópico lleva por nombre */erlecopter/bottom/image_raw* y es necesario suscribirse a él utilizando un manejador de nodos para obtener la imagen disponible. El método utilizado en ROS para manejar los mensajes de imagen consiste del objeto *image_transport::ImageTransport* que gestiona la recepción y ejecuta una función *callback* cuando una imagen esta disponible.

Descripción del proceso

El código utilizado para la obtención de la imagen es el mostrado en el Algoritmo 5.2. La explicación de código es la siguiente: al inicio del código, se inicializa el *nodo* llamando a la función *ros::init()*, indicando como argumento el nombre *image_listener* con el cual será identificado. Después, un manejador de *nodos* es creado para permitir que éste se ejecute. Posteriormente, una referencia a este manejador es pasado como argumento a un objeto de tipo *image_transport::Image_Transport*, vinculando el transporte de la imagen al *nodo image_listener*. Finalmente, el *nodo* es suscrito al tópico */erlecopter/bottom/image_raw* mediante la función *suscribe()* del objeto anterior, guardando una referencia al objeto *suscriptor* recién creado en un objeto *image_transport::Suscriber*. Habiendo realizo lo anterior, la función *imageCallback* será ejecutada cada vez que una nueva imagen esté disponible.

```
void imageCallback(const sensor_msgs::ImageConstPtr& msg)
{
  // Hacer algo con msg
}

int main(int argc, char **argv)
{

    ros::init(argc, argv, "image_listener");
    ros::NodeHandle nh;
    image_transport::ImageTransport img_transp(nh);
    image_transport::Subscriber sub_img;

  sub_img = img_transp.subscribe("/erlecopter/bottom/image_raw",
                                 1, imageCallback);
  ros::spin();
}
```

Algoritmo 5.2: Código de recepción de imagenes.

[1]La información del tópico indica que la frecuencia de actualización es de aproximadamente 15Hz, sin embargo, durante las pruebas no fue posible obtener una imagen en tiempo constante.

5.2.2. Ejecución en tiempo constante

Como se mencionó inicialmente, la función *callback* de la imagen no se realiza en un tiempo constante, debido a que depende de la velocidad con el que el tópico *publica* la imagen, la sobrecarga del sistema en el instante que se genera el mensaje y el tiempo que es consumido en el código a realizar el *callback* mismo. Esta incertidumbre en la frecuencia de su ejecución, no es deseable, debido a que se requiere que las instrucciones de control del VANT se transmitan a una velocidad suficiente para brindar estabilidad, el hacerlo a una velocidad muy baja podría significar un comportamiento errático en la aeronave. Es por ello que fue necesario hallar una herramienta que permitiera ejecutar el proceso de estimación de pose y correcciones a una frecuencia constante. La solución utilizada en el proyecto consistió en utilizar un *hilo* de ejecución independiente y utilidades de ROS para asegurar una actualización periódica de las instrucciones de control del VANT. Para la creación del *hilo* se utilizó *boost*, una biblioteca de software escrita en lenguaje *C++*, la cual brinda herramientas que pueden ser aprovechadas para el desarrollo de aplicaciones en ROS. De igual manera, se aprovechó el objeto *ros::Rate* que permite ejecutar *bucles* a una frecuencia determinada.

Descripción del proceso

La creación de un hilo en *boost* se realiza creando un objeto de la clase *boost::thread* e indicando la función *callback* que es ejecutada de forma independiente al proceso principal *main*() (ver Algoritmo 5.3). Para asegurar la ejecución del algoritmo en tiempo constante, un objeto *ros::Rate* es creado indicando una frecuencia de ejecución igual a 50Hz, suficiente para las tareas de estimación de pose y transmisión de correcciones al VANT. La tarea debe ejecutarse continuamente, por tal razón, se encuentra implementado dentro de un *bucle* el cual finaliza cuando la función *ros::ok()* devuelve *false*, indicando que el *nodo* ha cesado. Por último, el tiempo de ejecución se asegura próximo a los 50Hz llamando a la función sleep() del objeto rate, el cual genera una demora hasta satisfacer el tiempo $t = 1.0/50Hz$. En el código *main*(), es necesario llamar a la función *joint()* del *hilo* para asegurar que el *callback* finalice antes de terminar el programa principal.

```
void callback_algo(int *dummy)
{
  ros::Rate rate(50);
  while (ros::ok())
  {
    // Algoritmo
    rate.sleep ();
  }
}

int main(int argc, char **argv)
{
  ...
    int dummy = 0;
  ...

  boost::thread thread_algo(callback_algo, &dummy);
  ros::spin();
  thread_algo.join();
}
```

Algoritmo 5.3: Tiempo Constante.

5.2.3. Conversión de mensaje de imagen a cv::Mat

El mensaje *sensor_msgs::ImageConstPtr* recibido en la función *callback* de la imagen almacena la información de la imagen en un formato compatible con ROS. Sin embargo, la biblioteca OpenCV requiere de objetos *cv:Mat* para su funcionamiento, siendo esto en principio un inconveniente para su uso. Afortunadamente, ROS brinda la herramienta *CvBridge* que permite realizar la transformación de un mensaje a *cv::Mat*. Esta conversión no es realizada directamente en el *callback*, si no en el *hilo* destinado al algoritmo de estimación de pose y correcciones del VANT. Por tanto, fue necesario implementar un mecanismo que permita al *hilo* acceder al mensaje de imagen. La solución consistió en copiar el mensaje a una variable global, haciéndola visible para las todo el programa. Ningún método de sincronización fue requerido para asegurar la integridad del mensaje, debido a que los objetos *sensor_msgs::ImageConstPtr* son seguros en *multihilo*, por defecto.

Un diagrama del flujo que toma una imagen desde su captura en el VANT hasta su uso, es simplificado en la Figura 5.2.

Descripción del proceso

El primer paso de la implementación fue crear una variable global que almacenara una copia del mensaje *sensor_msgs::ImageConstPtr* recibido en la función *callback* de imagen. Posteriormente, fue el utilizado en el *callback* del *hilo* para convertirlo a una imagen compatible con OpenCV, mediante la función *toCvShare()* del la clase *cv::Bridge*. El llamado a esta función devuelve como resultado un objeto *CvImage-*

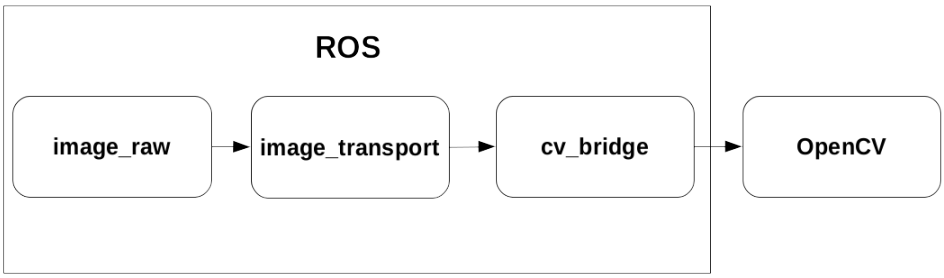


Figura 5.2: Esquema de funcionamiento del proceso de intercambio de imágenes.

ConstPtr, el cual posee el atributo *image* que consiste de la imagen convertida en un objeto *cv::Mat*. Para su funcionamiento, la función *toCvShare()* requiere como parámetros el mensaje de imagen ROS y la codificación de la imagen deseada, en este caso *"bgr8"* que significa: una imagen de tres canales en orden azul, verde, rojo con 8 bits de profundidad. El proceso de conversión puede fallar, lo que genera una *excepción*, por ello es necesario realizarlo dentro de un bloque *try-catch*, como puede observarse en el Algoritmo 5.4.

```
    sensor_msgs::ImageConstPtr gImageMsg;

  void callback_algo()
  {
      cv::Mat image;

    ros::Rate rate(50);
    while (ros::ok())
    {
      try
      {
        image = cv_bridge::toCvShare(gImageMsg, "bgr8")->image;
      }
      catch (cv_bridge::Exception& e)
      {
        // errores
      }

      // Hacer algo con image
      rate.sleep ();
    }
  }

  void imageCallback(const sensor_msgs::ImageConstPtr& msg)
  {
    gImageMsg = msg;
  }
  ...
```

Algoritmo 5.4: Conversión a cv::Mat.

5.2.4. Acceso a parámetros intrínsecos y coeficientes de distorsión

Uno de los requerimientos para que el proceso de detección del marcador de pose y la estimación de los parámetros extrínsecos funcionen adecuadamente, es contar con los parámetros intrínsecos de la cámara y los coeficientes de distorsión. Estos valores normalmente son obtenidos tras un proceso de calibración, sin embargo, para el proyecto esto no fue necesario. El VANT posee un tópico llamado */erlecopter/bottom/camera_info* el cual publica información relacionada con la cámara, entre ella sus parámetros intrínsecos y coeficientes de distorsión. Estos valores son constantes, por lo cual no es necesario acceder a ellos cada vez. Por esta razón, en lugar de suscribirse al tópico desde el código, se optó por utilizar la herramienta *rostopic* que permite visualizar dicha información en una terminal e implementarlos como constantes en el código. La instrucción requerida para mostrar esta información, es la siguiente:

```
rostopic echo /erlecopter/bottom/camera_info
```

Como resultado, una lista de elementos es mostrado en la terminal (ver Algoritmo 5.5), de los cuales solo nos interesan la matriz de parámetros intrínsecos (**K**) y el vector de coeficientes de distorsión (**D**).

```
---
header:
  seq: 189
  stamp:
    secs: 60
    nsecs: 605000000
  frame_id: erlecopter_bottomcam
height: 480
width: 640
distortion_model: plumb_bob
D: [0.0, 0.0, 0.0, 0.0, 0.0]
K: [374.6706070969281, 0.0, 320.5, 0.0, 374.6706070969281, ...
R: [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]
P: [374.6706070969281, 0.0, 320.5, -0.0, 0.0, 374.67060709 ...
binning_x: 0
binning_y: 0
roi:
  x_offset: 0
  y_offset: 0
  height: 0
  width: 0
  do_rectify: False
---
```

Algoritmo 5.5: Salida del comando *rostopic.*

Descripción del proceso

En la implementación en el código, la matriz K y el vector D fueron establecidos como constantes. Posteriormente, en base a ello se construyó un objeto del tipo *aru-*

co::CameraParameters requerido por la función de detección y estimación de parámetros extrínsecos.

```
cv::Mat K = (cv::Mat_<float>(3,3) << 374.6706070969281, 0.0,
                                     320.5, 0.0,
                                     374.6706070969281, 240.5,
                                     0.0, 0.0, 1.0);

cv::Mat distCoeffs = cv::Mat::zeros(1, 5, CV_32FC1);
...
aruco::CameraParameters camParam(K, distCoeffs,
                                          cv::Size(640, 480));
...
```

Algoritmo 5.6: Parámetros intrínsecos en el código.

5.2.5. Detección de marcadores

El proceso de detección de marcadores es realizado después de haber convertido el mensaje del *callback* de imagen a su equivalente en *cv::Mat*. Para llevarlo a cabo, es necesario utilizar la función *detect*() del objeto *aruco::MarkerDetector* que implementa la lógica del proceso de detección e identificación de marcadores. Como resultado, esta función devuelve un vector de objetos *cv::Marker*, siendo cada objeto una abstracción de un marcador de ArUco detectado, conteniendo información relacionada como su identificador y las coordenadas de sus cuatro esquinas en la imagen. Para una correcta identificación, la función *detect*() requiere, además de la imagen y el vector en donde se almacenarán los marcadores encontrados, los parámetros intrínsecos y los coeficientes de distorsión de la cámara.

El código implementado en el programa es el mostrado a continuación:

```
   cv::Mat K = (cv::Mat_<float>(3,3) << 374.6706070969281, 0.0,
                                        320.5, 0.0,
                                        374.6706070969281, 240.5,
                                        0.0, 0.0, 1.0);

   cv::Mat distCoeffs = cv::Mat::zeros(1, 5, CV_32FC1);
   aruco::CameraParameters camParam(K, distCoeffs,
                                          cv::Size(640, 480));
   sensor_msgs::ImageConstPtr gImageMsg;

 void callback_algo()
 {
     cv::Mat image;
     aruco::MarkerDetector mdetector;
     std::vector<aruco::Marker> markers;

   ros::Rate rate(50);
   while (ros::ok())
   {

     // Codigo para obtener la imagen
     mdetector.detect(image, markers, camParam);
     ...
     rate.sleep ();
   }
 }
 ...
```

Algoritmo 5.7: Detección de marcadores.

5.2.6. *Estructura* de información del marcador

El indicador de aterrizaje esta formado por cuatro marcadores de ArUco, cada uno con distintas dimensiones y posicionado (su centro) a una distancia conocida del centro del marcador más pequeño, el número 16. Esta información es necesaria para calcular la traslación y rotación promedio de los marcadores. Por tanto, para utilizarla en el código, se creó una *estructura* similar a la mostrada a continuación:

```
struct InfoMark {
  float size; // Dimensiones del marcador en metros
  float x;              // Distancia del centro del marcador 16
  float y;    // Distancia del centro del marcador 16
};
```

Utilizando la información de la Tabla 4.1, se crearon cuatro elementos de esta estructura, una para cada marcador. Siendo estas inicializadas como sigue:

```
   InfoMark mrk88 = {1.0, -0.5, -0.55};
   InfoMark mrk64 = {0.5, 0.31,  0.0};
   InfoMark mrk32 = {0.15, -0.05, 0.125};
   InfoMark mrk16 = {0.05, 0, 0};
```

Por conveniencia, estas estructuras se almacenaron en un *map* para poder acceder a ellos más fácilmente en el código utilizando como *llave* el identificador de los marcadores:

```
   std::map<int, InfoMark> mrkInfoMap;

mrkInfoMap[88] = mrk88;
mrkInfoMap[64] = mrk64;
mrkInfoMap[32] = mrk32;
mrkInfoMap[16] = mrk16;
```

Utilizando esta herramienta, fue posible acceder a los elementos de un marcador utilizando la siguiente notación:

```
mrkInfoMap[id].size
mrkInfoMap[id].x
mrkInfoMap[id].y
```

5.2.7. Exclusión de marcadores

Idealmente, se esperaría que solo los marcadores objetivo (88, 64, 32 y 16) sean identificados por la función *detect*(), sin embargo, en la práctica esto no es así. Para abordar un posible escenario en donde un marcador distinto a los esperados halla sido detectado, se implementó una fase de filtrado que permitió excluir los marcadores no deseados. Para hacerlo, se aprovechó el map creado con la información de los marcadores.

Descripción del proceso

Como puede observarse en el Algoritmo 5.8, después de la etapa de detección, un vector de marcadores identificados es creado. Para determinar si alguno de ellos corresponde a un marcador objetivo, se aprovechó la función *find()* del objeto *map*. Esta función se encarga de buscar una *llave* en el *map*, retornando como resultado un *iterador std::map<int, InfoMark>::iterator* desde la posición de la *llave* encontrada. En caso contrario, si la *llave* no existe, el *iterador* se posiciona al final del *map*. Este hecho fue utilizado como indicador para conocer si el marcador actual correspondía a uno objetivo. La llave utilizada en este caso, fue el identificador del marcador actual.

```
  sensor_msgs::ImageConstPtr gImageMsg;

void callback_algo()
{
    cv::Mat image;
    aruco::MarkerDetector mdetector;
    std::vector<aruco::Marker> markers;
    aruco::Marker mrk;

    std::map<int, InfoMark> mrkInfoMap;
    std::map<int, InfoMark>::iterator itr;

    InfoMark mrk88 = {1.0, -0.5, -0.55};
    InfoMark mrk64 = {0.5, 0.31,  0.0};
    InfoMark mrk32 = {0.15, -0.05, 0.125};
    InfoMark mrk16 = {0.05, 0, 0};

  mrkInfoMap[88] = mrk88;
  mrkInfoMap[64] = mrk64;
  mrkInfoMap[32] = mrk32;
  mrkInfoMap[16] = mrk16;

  ros::Rate rate(50);
  while (ros::ok())
  {

    // Codigo para obtener la imagen
    ...

    mdetector.detect(image, markers, camParam);

    for (int idx = 0; idx < markers.size(); idx++)
    {
      mrk = markers[idx];

      itr = mrkInfoMap.find(mrk.id);
      if (itr != mrkInfoMap.end())
      {
        // Algoritmo de estimacion de pose
        mrk.draw(frame,cv::Scalar(0,0,255),2);
      }
    }

    rate.sleep ();
  }
}'
...
```

Algoritmo 5.8: Exclusión de marcadores.

5.2.8. Estimación de pose

Para incrementar la confiabilidad de la información utilizada en el control del VANT, se consideró la contribución de todos los marcadores objetivo detectados en la imagen,

unificándola en una única pose. Con esta intensión, se compactaron en un vector todos las coordenadas de las esquinas de los marcadores detectados y en otro, sus coordenadas pero expresadas en el marco de referencia del mundo. Después, esta información fue utilizada junto a los parámetros intrínsecos para estimar los extrínsecos de la cámara, mediante la función *solvePnP()* de OpenCV, la cual devuelve la rotación y la traslación existente. Esto significó utilizar la información de la tabla 4.2, aunque no de forma directa, como se explicará a continuación.

Las coordenadas de las esquinas en la imagen fueron obtenidos directamente de la etapa de detección, las coordenadas del mundo sin embargo, fueron generados a partir de la información contenida en *map* de marcadores. Para ello se crearon dos funciones que se encargaban de almacenar en el vector correspondiente, las coordenadas expresadas en metros o en *pixeles*. Respectivamente, las funciones *addWorldCoordToVector()* y *addImageCoordToVector()*.

Para estimar las coordenadas de las esquinas de los marcadores en el plano de referencia del mundo, se estableció como origen el centro del marcador principal. En base a ello, la función *addWorldCoordToVector()* generaba dinámicamente las coordenadas de las esquinas del marcador (ver Tabla 4.2) y las agregaba al vector *worldCoordinates*. De igual forma, la función *addImageCoordToVector()* tomaba las coordenadas del marcador actual y las almacenaba en el vector *imageCoord*. Esto se realizaba en cada etapa de detección no olvidando borrar el contenido de ambos vectores cada vez.

Descripción del proceso

El primer paso de la implementación consistió en crear los dos vectores para almacenar las coordenadas en la imagen o en el mundo, *imageCoord* o *worldCoord*, respectivamente. Estos vectores son llenados dinámicamente utilizando las funciones *addImageCoordToVector()* y *addWorldCoordToVector()*, usando como parámetro las dimensiones del marcador y la información de su origen almacenada en el *map* (para más detalles de su implementación, consúltese el código fuente en el Apéndice D). Posteriormente, la pose es estimada utilizando ambos vectores y los parámetros intrínsecos en la función *solvePnP()* que devuelve un vector de rotación y no de traslación. Esta representación de la rotación, sin embargo, no es útil para estimar los ángulos de Euler de forma sencilla, por tanto, se utilizo la función *cv::Rodrigues()* para convertirlo a matriz de rotación. Después, los ángulos fueron obtenidos mediante las ecuaciones en 3.11. La traslación por otro lado, fue utilizada directamente, como puede observarse en el Algoritmo 5.9.

```
void callback_algo()
{
    cv::Mat image;
    aruco::MarkerDetector mdetector;
    std::vector<aruco::Marker> markers;
    aruco::Marker mrk;
    float tx = ty = tz = 0;
    int cntValMrks = 0;
    float r11 = r21 = r32 = r32 = r33 = 0;
    std::map<int, InfoMark> mrkInfoMap;
    std::map<int, InfoMark>::iterator itr;

  ros::Rate rate(50);
  while (ros::ok())
  {
    mdetector.detect(image, markers, camParam);
    worldCoord.clear();
    imageCoord.clear();
    for (int idx = 0; idx < markers.size(); idx++)
    {
      mrk = markers[idx];
      itr = mrkInfoMap.find(mrk.id);
      if (itr != mrkInfoMap.end())
      {
        mrkSize = mrkInfoMap[mrk.id].size;
        mrkOriX = mrkInfoMap[mrk.id].x;
        mrkOriY = mrkInfoMap[mrk.id].y;
        addWorldCoordToVector(worldCoord, mrkSize, mrkOriX,
                              mrkOriY);
        addImageCoordToVector(mrk, imageCoord);
        cntValMrks++;
      }
    }

    if (cntValMrks > 0)
    {
      cv::solvePnP(worldCoord, imageCoord, K, distCoeffs,
                                                Raux, Taux);
      Raux.convertTo(Rc, CV_32F); Taux.convertTo(Tc, CV_32F);
      tx = Tc.at<float>(0, 0); ty = Tc.at<float>(1, 0);
      tz = Tc.at<float>(2, 0);
      cv::Rodrigues(Rc, Raux);
      r11 = Raux.at<float>(0, 0);  r21 = Raux.at<float>(1, 0);
      r31 = Raux.at<float>(2, 0);  r32 = Raux.at<float>(2, 1);
      r33 = Raux.at<float>(2, 2);
      ang_roll = radToDeg(atanf(r32/r33));
      ang_pitch = radToDeg(atanf(-r31/sqrtf((r32*r32)+
                                            (r33*r33))));
      ang_yaw = radToDeg(atanf(r21/r11));
    }
    rate.sleep ();
  }
}
```

Algoritmo 5.9: Estimación de pose.

5.2.9. Margen de aterrizaje

Para incrementar la precisión durante el proceso de aterrizaje, se utilizó la altitud promedio para determinar la distancia máxima permitida entre el VANT y el marcador, para iniciar dicho proceso. Una constante α fue considerado para decrementar el margen de error a medida que lo hacía la altura. Una valor máximo de altura de diez metros fue establecido para evitar que el margen de aterrizaje supere el metro de error.

Descripción del proceso

En la implementación, el margen de aterrizaje fue indicado utilizando la variable minXYland. El criterio de valor máximo de altura fue resuelto mediante un condicional, que evaluaba su valor actual y establecía el margen de error a su máximo, un metro en este caso o al producto de la altitud promedio por el constante α. El valor utilizado fue $\alpha = 0.1$.

```
    sensor_msgs::ImageConstPtr gImageMsg;

  void callback_algo()
  {
      ...
      float tz = 0;
    ...
    ros::Rate rate(50);
    while (ros::ok())
    {

      // Codigo para obtener la imagen y estimar la pose acumulada
      ...

      if (cntValMrks > 0) // Si se detecto algun marcador valido
      {
        // Codigo para obtener la pose promedio
        ...

        if (tz >= 10 )
        {
          minXYland = 1.0;
        }
        else
        {
          minXYland = 0.1*tz;
        }
      }

      rate.sleep ();
    }
  }
  ...
```

Algoritmo 5.10: Margen de aterrizaje.

5.2.10. Modo de vuelo

El método utilizado para iniciar el proceso de aterrizaje consiste en cambiar el modo de vuelo a *LOITER*, el cual permite sobreescribir los canales del VANT. La forma de saber el modo en el cual se encuentra la aeronave es suscribiéndose al tópico */mavros/state* que genera una llamada a un *callback* cuando un cambio de estado es realizado.

Descripción del proceso

Para suscribirse al tópico de estado, fue necesario crear un objeto del tipo *ros::Suscriber* e inicializarlo mediante el llamado a la función *suscribe()* del manejador de nodos. Como parámetros, fue indicado el nombre del tópico */mavros/state* y el *callback* que es llamado cada vez que un cambio en el estado del VANT es realizado. En el *callback*, el estado del VANT es determinado revisando la propiedad *mode* del mensaje recibido, el cual consiste de una cadena de texto que indica el nombre del modo actual. Por seguridad, un *guard* es considerado para evitar el modo sea cambiado erróneamente por un mensaje nulo.

```
std::string mode;

void mavrosStateCb(const mavros_msgs::StateConstPtr &msg)
{
  if(msg->mode == std::string("CMODE(0)"))
    return;
  mode = msg->mode;
}

void callback_algo()
{
    ...
    if (cntValMrks > 0)
    {
      // Algoritmos relacionados con el aterrizaje
      if (mode == "LOITER")
      ...
    }
}

int main(int argc, char **argv)
{
    ros::NodeHandle nh;
    ros::Subscriber sub_mavstate;

  sub_mavstate = nh.subscribe("/mavros/state", 1,
                                              mavrosStateCb);
}
```

Algoritmo 5.11: Modo de vuelo.

5.2.11. PID

El controlador es el encargado de realizar las correcciones de pose de forma automática. Como se mencionó en un principio, la herramienta utilizada para dicho fin fue un PID, mismo que se encarga de obtener la magnitud y dirección de las correcciones en base al error y de minimizar este último. En código, el programa es muy simular al mostrado en el Algoritmo 4.1. Esta pieza de código, es ejecutada cuando el VANT cambia a modo *LOITER*, permitiendo de esta manera sobreescribir los canales *rc*. Por simplicidad, solo los componentes del PID del *roll* son mostrado a continuación, sin embargo, la estructura es básicamente la misma para el *pitch* y el *yaw*, a excepción de las constantes *kp*, *ki* y *kd* que son distintas para cada caso. En la Tabla 5.1 aparecen las constantes utilizadas en el proyecto.

```
if (mode == "LOITER")
{
  errorx = 0.0 - tx;
  intx = intx + errorx * dt;
  devx = (errorx - prev_errorx) / dt;
  outx = (kpx * error) + (kix * intx) + (kdx * devx);
  prev_errorx = errorx;

  rc = BASERC-outx;

  if (rc > MAXRC)
  {
    rc = MAXRC;
  }
  else
  {
    rc = MINRC;
  }

  ...
}
```

Algoritmo 5.12: PID en código.

Const.	*roll*	*pitch*	*yaw*
kp	3.0	3.0	1.5
ki	0.2	0.2	0.1
kd	10.0	10.0	1.0

Tabla 5.1: Constantes del PID utilizadas.

5.2.12. Sobreescritura de canales RC

Habiendo obtenidos los valores adecuados para la corrección de la pose, es necesario indicarlos a la etapa de control interna del VANT para su ejecución. La forma de hacerlo es publicando un mensaje al tópico */mavros/rc/override*, indicando el valor *rc* del *roll*,

pitch y *yaw* estimados con el PID y el *throttle* establecido para el aterrizaje. Haciendo esto a una frecuencia estable, el VANT fue capaz de aterrizar adecuadamente.

Descripción del proceso

El primer paso realizado en la implementación fue crear un objeto *ros::Publisher* para transmitir el mensaje de sobreescritura de canales. Esto mediante la suscripción al tópico */mavros/rc/override* utilizando la función *advertise()* del manejador de nodos. El mensaje requerido para la sobreescritura es del tipo *mavros_msgs::OverrideRCIn* el cual consiste de un vector de 8 elementos consistiendo de los canales del control del VANT. Correspondiendo los cuatros primeros al *roll*, *pitch*, *throttle* y *yaw*, respectivamente. Los elementos del mensaje son llenados con el valor estimado por el PID, implementado en el hilo del algoritmo. Una vez realizado, el mensaje es publicado llamando al método *publish()* del objeto *ros::Publisher*.

```
  ros::Publisher pub_rc;
void callback_algo()
{
    mavros_msgs::OverrideRCIn msg_rc;
  while (ros::ok())
  {
    if (cntValMrks > 0) // Si se detecto algun marcador valido
    {
      // Codigo de aterrizaje
      rc_roll = ...;
      rc_pitch = ...;
      rc_yaw = ...;
      rc_throttle = ...;
    }
    else
    {
      rc_roll = rc_pitch = rc_yaw = rc_throttle = BASERC;
    }

    msg_rc.channels[0] = rc_roll;
    msg_rc.channels[1] = rc_pitch;
    msg_rc.channels[2] = rc_throttle;
    msg_rc.channels[3] = rc_yaw;

    pub_rc.publish(msg_rc);
    rate.sleep ();
  }
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "image_listener");
    ros::NodeHandle nh;
  pub_rc = nh.advertise<mavros_msgs::OverrideRCIn>(
                                        "/mavros/rc/override", 1);
}
```

Algoritmo 5.13: Sobreescritura de canales.

5.3. Hilo de visualización

La etapa de control del VANT es un proceso que involucra el uso de una gran cantidad de herramientas, por tanto, esta tarea puede tomar un tiempo en finalizarse. En control, es deseable que el algoritmo funcione a una frecuencia adecuada para permitir que las lecturas y correcciones sean aplicadas en tiempo real, es decir, a una velocidad adecuada que no ocasione inestabilidad en el sistema. Con esta intención, se optó por separar la etapa de visualización a un hilo de ejecución distinto para evitar que interfiera con el algoritmo de control.

Para conocer el estado actual del VANT, se creó la interfaz mostrada en la Figura 5.3, en donde se plasmó información relevante como la pose actual de la aeronave, el es-

tado del aterrizaje, el margen máximo permitido y la imagen observada por la cámara. Esta información, en principio no es conocida por el hilo de visualización, únicamente por el algoritmo principal de control, por tanto fue necesario informarla al primero de alguna manera. La herramienta utilizada para dicho fin consistió de una *estructura* en donde se almacenaban los datos relevantes del proceso de aterrizaje, mismos que se muestran a continuación:

```
struct InfoVant {
  bool markDetected;
  float x;
  float y;
  float z;  // Altitud
  float rx; // Pitch
  float ry; // Roll
  float rz; // Yaw
  bool landing;
  float minXY;
  cv::Mat img;
};
```

donde x y y corresponden a la posición actual del VANT, rx, ry y rz a su rotación, *landing* una bandera que indica si se encuentra en proceso de aterrizaje, $minXY$ el margen de aterrizaje en metros e *img* la última de donde se obtuvieron los datos anteriores.

Es importante señalar que la información mostrada en la visualización puede no corresponder con la utilizada por la etapa de control, como consecuencia que la etapa de visualización es más tardada que la etapa de control. Sin embargo, la frecuencia de actualización es suficiente para los fines de depuración requeridos.

Debido a que la información es intercambiada entre hilos, para evitar conflictos con los datos en memoria se utilizó un mecanismo básico de sincronización: una bandera *flagShow* que informaba al hilo de control que se requería una imagen y a su vez indicaba la disponibilidad de la misma al hilo de visualización. La implementación de esta idea se muestra en el Algoritmo 5.14.

```
void thread_algo(int *dummy)
{
  ros::Rate rate (50);
  while (ros::ok())
  {
    ...
    if (gImageMsg != NULL)
    {
      if (flagShow == false)
      {
        infoVant.markDetected = markDetected;
        infoVant.x = tx;
        infoVant.y = ty;
        infoVant.z = tz;
        infoVant.rx = ang_roll;
        infoVant.ry = ang_pitch;
        infoVant.rz = ang_yaw;
        infoVant.landing = landingStarted;
        infoVant.minXY = minXYland;
        image.copyTo(infoVant.img);
        flagShow = true;
      }
    }
    ...
  }
}


void thread_view(int *dummy)
{
  ros::Rate rate (50);
  while (ros::ok())
  {
    ...
    if (gImageMsg != NULL)
    {
      if (flagShow == false)
      {
        infoVant.markDetected = markDetected;
        infoVant.x = tx;
        infoVant.y = ty;
        infoVant.z = tz;
        infoVant.rx = ang_roll;
        infoVant.ry = ang_pitch;
        infoVant.rz = ang_yaw;
        infoVant.landing = landingStarted;
        infoVant.minXY = minXYland;
        image.copyTo(infoVant.img);
        flagShow = true;
      }
    }
    ...
  }
}
```

Algoritmo 5.14: Sincronización e intercambio de información entre hilos.

La implementación completa se muestra en el Apéndice D.

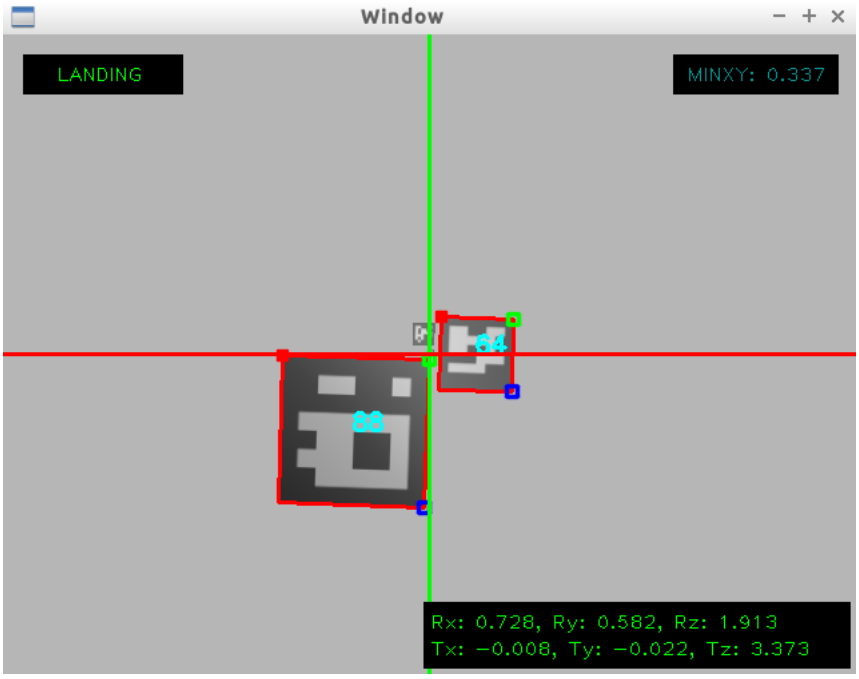


Figura 5.3: Esquema de funcionamiento del proceso de intercambio de imágenes.

Capítulo 6

Simulación

Para validar el funcionamiento y rendimiento del algoritmo propuesto, comparado con el uso de solo información GPS, se realizó la simulación de un *Erle-copter*, el cual consiste de un cuadrucóptero desarrollado y comercializado por la compañía *Erle Robotics*[37], el cual además de contar con aeronaves reales, provee con los componentes de software que permiten su simulación en Gazebo. El procedimiento de configuración requerido para su funcionamiento es mostrado en el Apéndice B.

6.1. Descarga y compilación del modelo

La simulación se basa en un *mundo* llamado *pattern_follower*[38], el cual consiste de un *Erle-copter* que sigue un marcador de ArUco el cual se mueve en rutas circulares o formando una figura rectangular, esto utilizando información de la cámara apoyado con un algoritmo muy básico de visión computacional, mismo que fue sustituido por el propuesto en el presente documento (ver Apéndice D).

Una vez configurado el entorno como se describe en el Apéndice B, es necesario descargar el proyecto y mundo requerido, para ello basta con abrir una terminal de comandos y teclear los comandos siguientes:

```
cd ~/simulation/ros_catkin_ws/src
git clone https://github.com/erlerobot/ros_erle_pattern_follower
cd ..
catkin_make --pkg ros_erle_pattern_follower
```

6.2. Integración del marcador de aterrizaje

El modelo de simulación *pattern_follower*[38], por defecto no cuenta con el marcador de aterrizaje propuesto en el presente documento, por el cual es necesario modificar el *mundo* para integrarlo. Primeramente es necesario crearlo a medida, tal y como se describe en el Apéndice C y posteriormente agregarlo a la carpeta de modelos de Gazebo. Para lograrlo, debe construirse una estructura de directorios dentro de *.gazebo/models*, en donde se incluirán los archivos de configuración (*.config* y *.sdf*), el modelo del mar-

cador (archivo *.dae*) y los recursos requeridos para la simulación (*aruco*xx.*png*), resultando en la estructura siguiente:

```
.gazebo
└── models
    └── landing_mark
        ├── model.config
        └── model.sdf
            ├── materials
            │   └── textures
            │       ├── aruco16.png
            │       ├── aruco32.png
            │       ├── aruco64.png
            │       └── aruco88.png
            └── meshes
                └── marker.dae
```

6.2.1. Archivo de configuración

El archivo de configuración contiene información relacionada con el modelo de simulación como lo es la versión del archivo de descripción *.sdf* utilizado, el nombre del modelo y el autor. El archivo *model.config* del marcador de aterrizaje es el siguiente:

```
<?xml version="1.0" ?><model>
  <name>landing_mark</name>
  <version>1.0</version>
  <sdf version="1.6">model.sdf</sdf>

  <author>
    <name>mpoot</name>
  </author>
</model>
```

6.2.2. Archivo de descripción

SDF [39] es un archivo que describe los objetos de un entorno de simulación generado en Gazebo. Está escrito en *XML* e incluye la información del objeto (dimensiones, pose, etc.), los recursos (texturas) y los *plugins* para su funcionamiento. En el caso del marcador de aterrizaje, el archivo *.sdf* es el mostrado a continuación:

```
<?xml version="1.0" ?><sdf version="1.6">
  <model name="landing_marker">
    <static>true</static>
    <link name="link">
      <visual name="visual">
        <geometry>
          <mesh>
            <uri>model://landing_mark/meshes/marker.dae</uri>
          <scale>1.0 1.0 1.0</scale></mesh>
        </geometry>
      </visual>
    </link>
    <plugin name="mark_driver"
            filename="libmark_plugin.so">
      <shape>dot</shape>
    </plugin>
  </model>
</sdf>
```

Siendo *libmark_plugin.so* el *plugin* que permite al marcador realizar movimientos circulares, rectangulares o mantener una posición fija en la simulación[38].

6.3. Lanzamiento de la simulación

Habiendo completado los pasos de configuración y modificado el *mundo* de simulación, el proceso de simulación puede ser realizado, mismo que se logra en dos etapas: la primera consiste en lanzar el nodo ROS y MAVProxy, los cuales se encargan de proveer y mantener las comunicaciones entre el simulador Gazebo y ROS, de la misma manera que se hiciera con una aeronave real y la segunda etapa la cual es simplemente iniciar el modelo en Gazebo. Ambos procesos se encuentran automatizados mediante *scripts* que se encargan de encontrar e inicializar todos los requerimientos necesarios para la simulación.

La primera parte de la simulación, el lanzamiento del nodo ROS, se realiza ingresando los comandos siguientes en una terminal (Terminal A):

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

Al hacerlo, todos los procesos ROS relacionados a la simulación serán iniciados y mantenidos hasta el cierre de la ventana, motivo por el cual es requerido que dicha ventana permanezca abierta.

Posteriormente, debe iniciarse el *mundo* en Gazebo, lo que se realiza ingresando los comandos siguientes en una segunda terminal (Terminal B):

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

De ser esto exitoso, se abrirá el programa con una vista similar a la mostrada en la Figura 6.1, sin embargo, la aeronave permanecerá en tierra hasta que el comando de despegue sea enviado desde la terminal anterior mediante la serie de comandos siguientes:

```
mode GUIDED
arm throttle
takeoff 10
param set SYSID_MYGCS 1
```

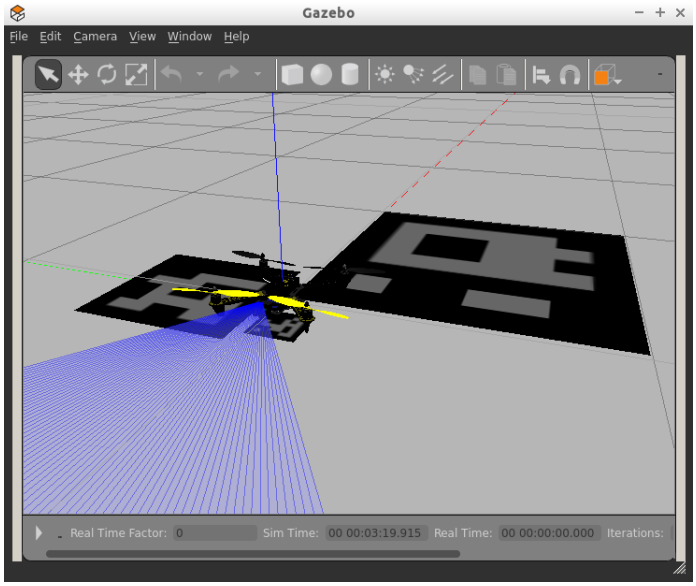


Figura 6.1: Simulación en Gazebo.

Después, en una tercera terminal (Terminal C) debe ser iniciado el algoritmo propuesto en el presente trabajo (ver Apéndice D), lo cual se logra utilizando las instrucciones mostradas a continuación:

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

6.4. Inicio del proceso de aterrizaje

Para iniciar el proceso de aterrizaje, se requiere la aeronave se encuentre en vuelo y que el modo de se cambie a *LOITER*, esto se logra ingresando en la Terminal B (ver Figura 6.2) la instrucción **mode LOITER**.

Habiendo realizado lo anterior, el comportamiento del algoritmo consistirá en intentar detectar el marcador de aterrizaje y en base a ello calculará la corrección necesaria que lleven a la aeronave a su aterrizaje, éste se ejecutará constantemente y brindará una retroalimentación visual con fines de depuración, como se puede observar en la

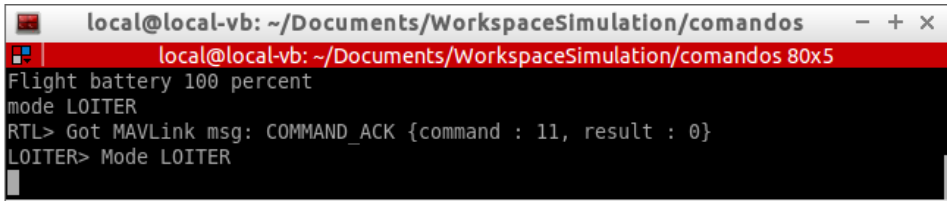


Figura 6.2: Terminal de simulación.

Figura 6.3.

Finalmente, si durante el proceso de aterrizaje se ha mantenido los marcadores en el rango de vista de la cámara, la tarea se completará aproximando la aeronave lo más cerca posible del marcador de aterrizaje.

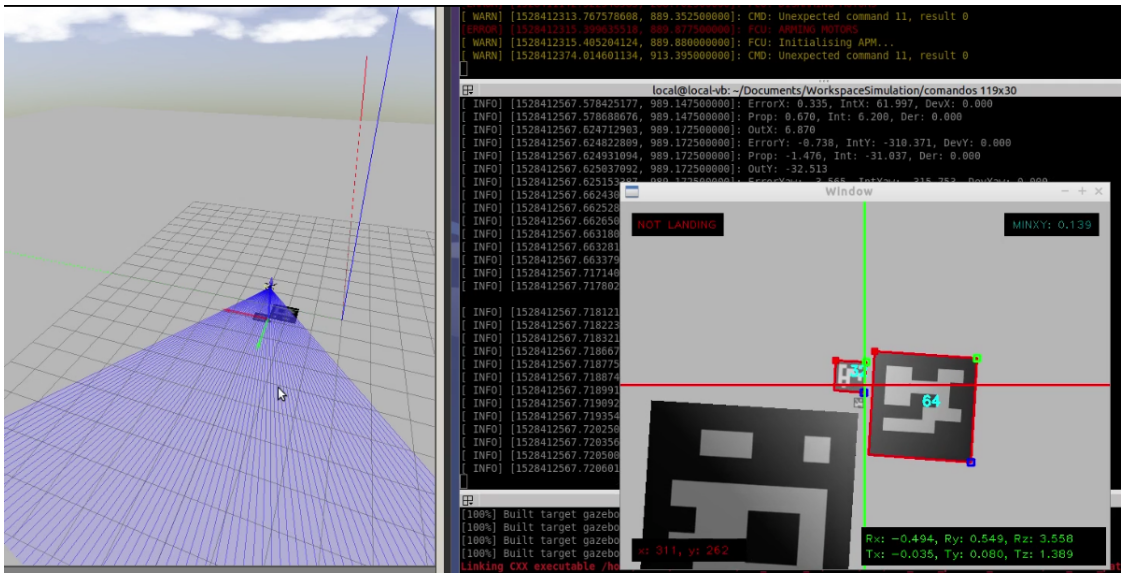


Figura 6.3: Vista de simulación.

6.5. Descripción de aspectos de simulación

La forma de trabajar del algoritmo consiste en aproximar el VANT hacia el punto objetivo, indicado por el marcador de aterrizaje. Esto lleva a considerar a este último como el origen del marco de referencia utilizado para determinar la distancia a la cual se encuentra la aeronave. Esto produce que el comportamiento de las coordenadas, observado desde la perspectiva de la simulación, sea igual al mostrado en la Figura 6.4. En resumen, dependiendo del cuadrante en donde el VANT se ubique será el signo con el que se consideren sus componentes x y y. El componente z (altitud), sin embargo, es tomado siempre como positivo.

Por otro lado, para observar el comportamiento del algoritmo en tiempo real, se superpusieron algunos elementos en la imagen obtenida por el VANT (ver Figura 6.4).

Dichos elementos son los siguientes:

a) Indicador de Aterrizaje. Muestra "*LANDING*" cuando se cumplen las condiciones para iniciar el aterrizaje y "*NOT LANDING*" en caso contrario.

b) Indicador de margen de aterrizaje. Muestra la distancia máxima requerida para considerar que el aterrizaje puede ser iniciado.

c) Posición actual. Posición del VANT relativa al marcador de aterrizaje.

d) Pose del VANT. Componentes de la pose del VANT, siendo Rx, Ry y Rz la rotación del VANT respecto al marcador y Tx, Ty y Tz su traslación.

e) Marcador de aterrizaje detectado.

Gracias a estos elementos fue posible observar el desplazamiento del VANT y el funcionamiento del algoritmo propuesto, lo que permitió realizar los ajustes pertinentes para obtener un rendimiento aceptable.

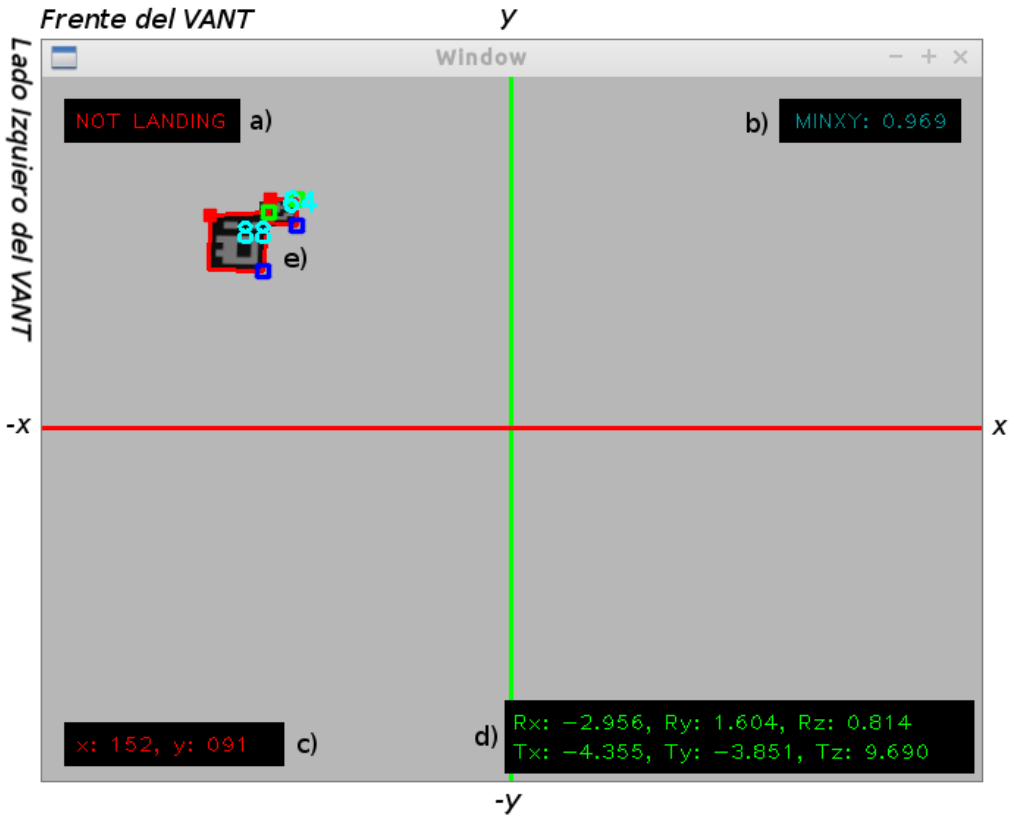


Figura 6.4: Vista de simulación desde el VANT.

Capítulo 7

Resultados

7.1. Aterrizaje

Con la intención de poder comparar el rendimiento de estimación de la posición del algoritmo propuesto contra el obtenido a través del GPS, se realizó un vuelo de prueba en el software Gazebo ubicando el VANT a una altura de diez metros y se permitió que el algoritmo controlara los movimientos de la aeronave hasta su aterrizaje. Al mismo tiempo, se almacenaron los datos generados por el GPS, el algoritmo de visión y el tópico */mavros/local_position/pose/*, el cual provee la posición y altura actual del VANT relativa al punto de partida.

Para poder compararlos, estos datos fueron normalizados tomando el primer valor leído de cada una de las fuentes (algoritmo de visión, GPS y servicio) y utilizándolo como punto inicial (*cero*), a partir del cual se ajustaron todos los demás valores del su conjunto. Sin embargo, antes de poder hacer eso, fue necesario convertir los datos del GPS a una unidad de distancia, ya que estos se encontraban expresados en sus componentes de latitud y longitud. Las fórmulas utilizadas para lograrlo son las presentadas en [40], que obtiene la distancia en kilómetros entre dos puntos, partiendo de sus coordenadas GPS expresadas en grados decimales:

$$x = (lon_{act} - lon_{ref}) \cdot rad(R) \cdot \cos\left(rad(lat_{ref})\right)$$
$$y = (lat_{act} - lat_{ref}) \cdot rad(R)$$

donde x y y son los componentes horizontal y vertical, respectivamente, expresados en kilómetros, lon_{ref} y lat_{ref} las coordenadas utilizadas como punto de origen, lon_{act} y lat_{act} las coordenadas actuales y R la circunferencia de la Tierra en kilómetros (6371km).

A continuación se expondrán los resultados obtenidos al analizar los datos generados durante el proceso de aterrizaje.

7.1.1. Algoritmo Propuesto

Resultado de la experimentación, se determinó que el algoritmo de visión computacional propuesto tuvo un rendimiento aceptable, muy apegado a lo informado por el servicio de *mavros*. Algo que puede apreciase en la gráfica de la Figura 7.1 en donde se resume el comportamiento de la traslación en los ejes x, y y z que sufrió el VANT con el paso del tiempo. Algunas oscilaciones tanto en el eje x y y son apreciadas, esto como consecuencia del desplazamiento del VANT y el funcionamiento del algoritmo en tiempo real que intenta compensar el empuje de los motores. Por otro lado, se nota como una vez que el VANT se ubicó por encima del marcador, el algoritmo inicio de forma automática el proceso de aterrizaje disminuyendo la altura paulatinamente hasta aproximarse a tierra. En ciertas zonas de la gráfica de altura (z) se notan pequeñas mesetas, esto se debe a que en ese instante de tiempo el algoritmo de visión detectó que la distancia al marcador no era la esperada y de continuara con el aterrizaje podría perderse esta referencia visual imposibilitando continuar con el proceso, por lo cual, primeramente compensó la disparidad y luego continuó con el proceso de aterrizaje. Nótese como la altura nunca llega a cero, esto en consecuencia a la dificultad que representa para el algoritmo de control calcular las correcciones ya que a esta escala el error es muy pequeño y el impacto en el PID no es significativo. Sin embargo, la distancia entre el VANT y el marcador es muy pequeña, alrededor de veintisiete centímetros, altura en donde los motores podrían ser apagados de forma segura para completar el aterrizaje, si se considera que el VANT posee soportes para dicho fin.

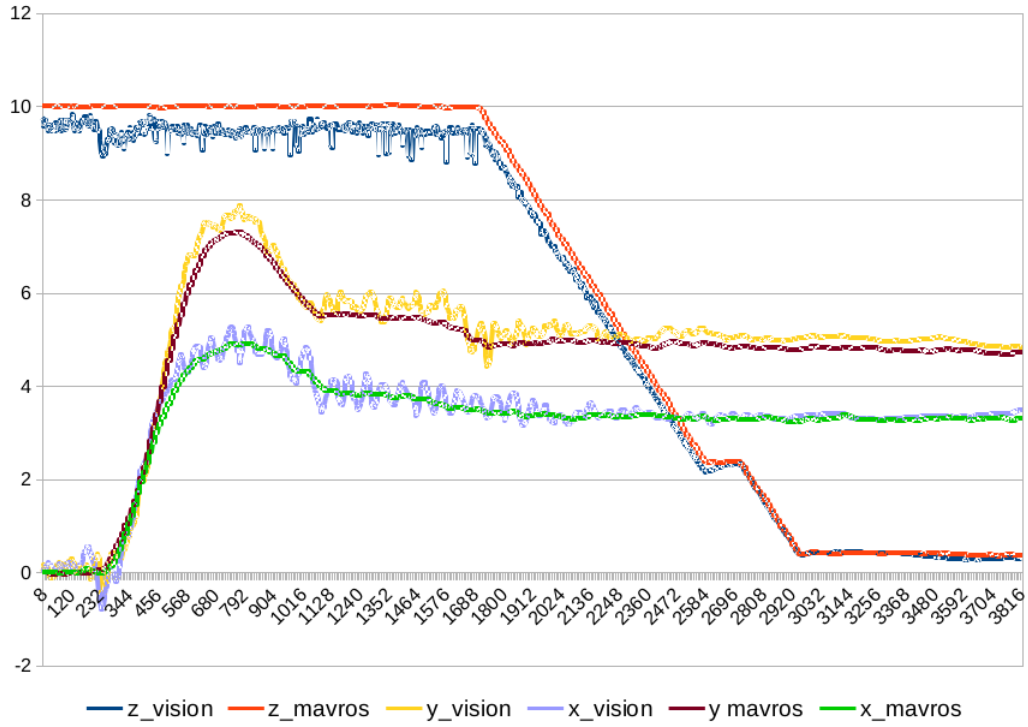


Figura 7.1: Evolución de la traslación. Algoritmo de visión contra servicio *mavros*.

7.1.2. GPS

Por otro lado, también se realizó el análisis utilizando los datos obtenidos del receptor GPS disponible en el VAN. Es necesario recordar que el aterrizaje fue controlado únicamente por la información obtenida por la cámara y la ubicación del GPS no fue utilizada. Sin embargo, la intención de esta comparativa es poder comparar el rendimiento obtenido contra del algoritmo de visión propuesto. Cabe mencionar que el componente en z no fue considerado en la comparativa ya que en el VANT éste no es utilizado para estimar la altura, en su lugar, son utilizados otros sensores como el barómetro, lo cuales ofrecen una mayor precisión. El comportamiento de las lecturas respecto al tiempo puede observarse en las gráficas de la Figura 7.2. En principio, aparenta que el GPS brinda un rendimiento muy bueno, sin embargo, debe considerase que los datos fueron obtenidos de un modelo de simulación ideal en donde se asume que la calidad de la señal de los satélites es perfecta y aspectos comunes en los receptores GPS como la perdida de señal no existen. Algo que sucede de forma similar en el algoritmo de visión en donde cambios de iluminación o deformaciones en la imagen por temperatura no están presentes. A pesar de ello, da una aproximación a lo que se esperaría de un GPS de buena calidad funcionando con condiciones idóneas.

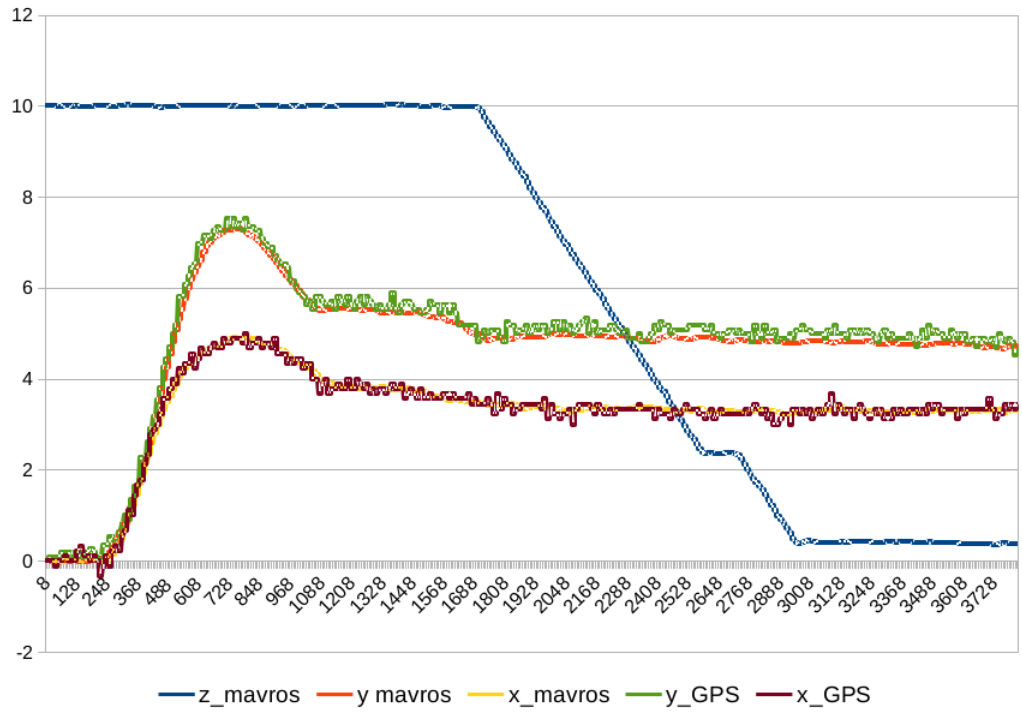


Figura 7.2: Evolución de la traslación. GPS contra servicio *mavros*.

7.1.3. Estadísticas de error

Finalmente, comparando el rendimiento de ambas fuentes de información (GPS y del algoritmo propuesto) se concluyó que estás proveen un comportamiento bastante similar, muy apegado a lo que se publica en el servicio de *mavros*. Como se puede observar en la Tabla 7.1, el GPS ofrece mejores resultados en general, pero el algoritmo de visión tiene un mejor comportamiento a bajas altitudes, algo que es deseable ya que se espera es aterrizar lo más cercano posible al centro del marcador.

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

Tabla 7.1: Estadísticas de error.

Si analizamos las gráficas de error absoluto, podemos observar como éste iba disminuyendo a medida que la aeronave se aproximaba al centro del marcador. Es necesario mencionar que el error observado es el resultado de las correcciones realizadas para ajustar la posición, lo que ocasionaba que el VANT se desplazara horizontal y verticalmente cambiando su distancia respecto al marcador. Por tanto, como en un principio

el error era significativo, la etapa del control del algoritmo propuesto generaba correcciones bruscas, dado a que su comportamiento es lineal pero la dinámica de vuelo de la aeronave no lo es.

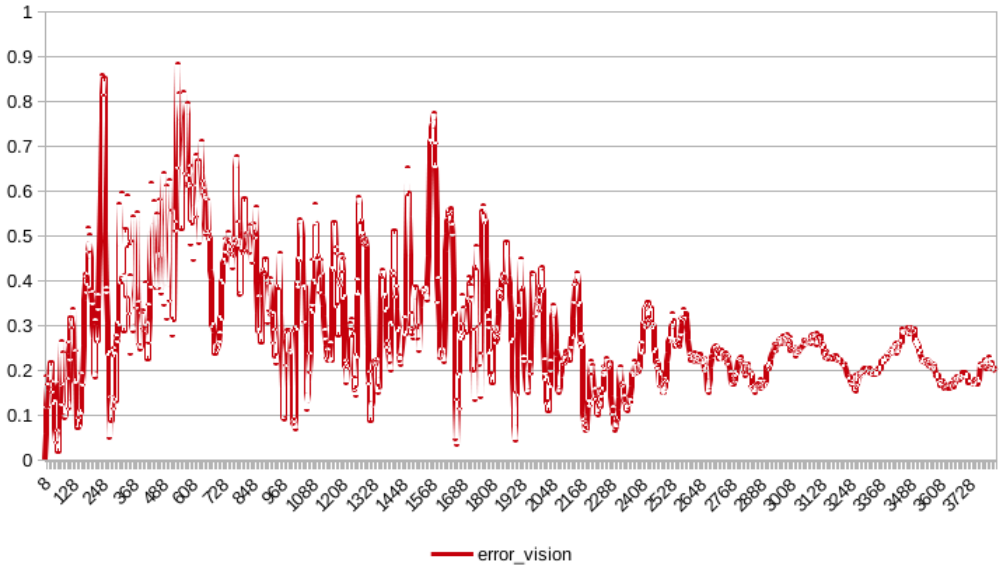


Figura 7.3: Error absoluto de traslación. Algoritmo de visión contra servicio *mavros*.

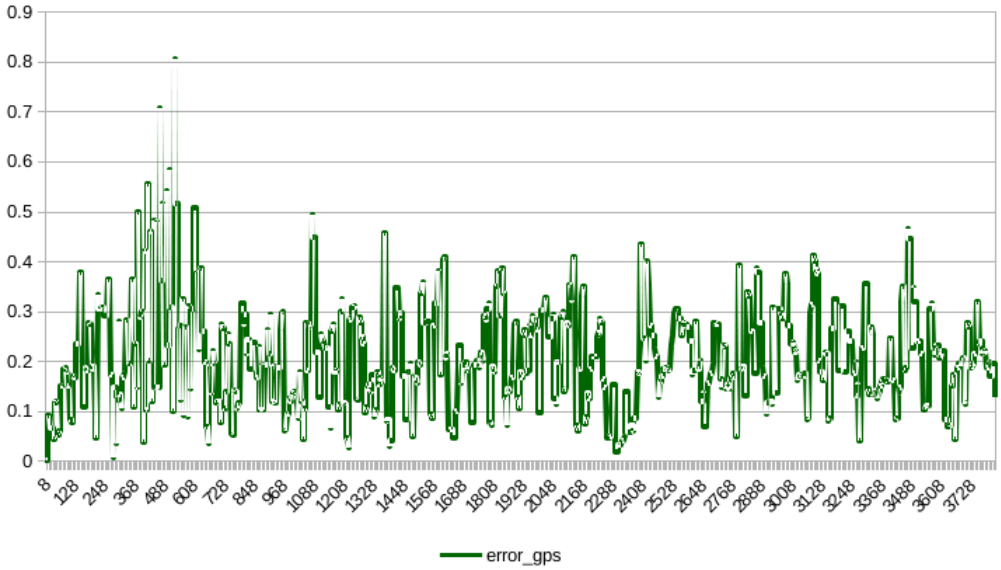


Figura 7.4: Error absoluto de traslación. GPS contra servicio *mavros*.

7.2. Control de Posición

De igual forma, en conjunto con el Ing. Carlos Acosta [1] se implementó un algoritmo de control fuera de borda (*Offboard*), utilizado para permitir que la aeronave se dirija a un punto en específico y una vez alcanzado, ésta mantenga dicha posición. Dicho

algoritmo utiliza el punto de despegue como coordenada inicial $P_0(x, y, z) = (0, 0, 0)$ y en base a ello calcula la dirección y altura que debe considerar el VANT para llegar a la posición objetivo. Para validar su funcionamiento se realizaron dos vuelos de prueba en el entorno de simulación, el primero con una posición objetivo $P_1(0, -9, 5)$ y el segundo, $P_2(4, 3, 6)$ y se graficaron los valores publicados por el servicio de *mavros*. En las Figuras 7.5 y 7.6 puede observarse como la distancia fue disminuyendo a medida que el VANT se desplazaba al objetivo para posteriormente mantenerse en ese punto sin cambio alguno. Nótese como ambas gráficas difieren debido a que cada una se posiciona en un cuadrante distinto (véase la Figura 6.4) lo que produce que el componente de posición evolucione en distintas direcciones.

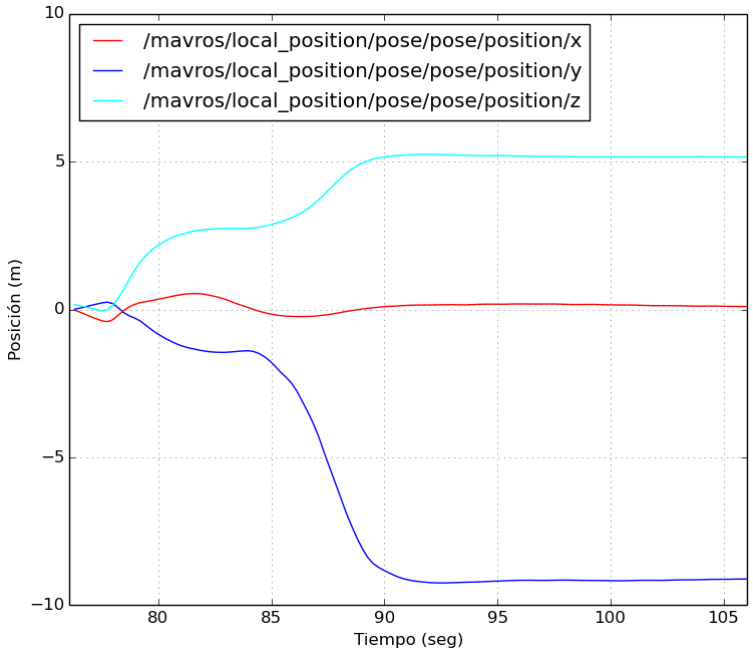


Figura 7.5: Gráfica para la posición $P_1(0, -9, 5)$.

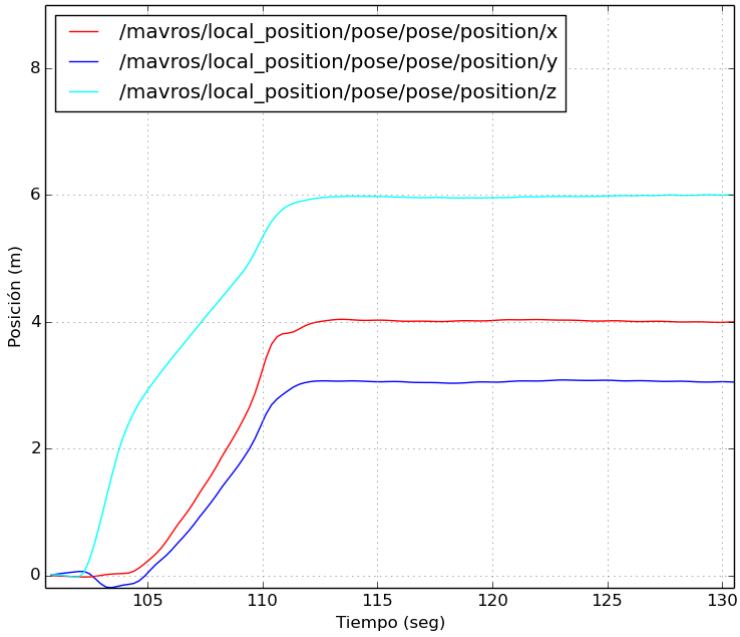


Figura 7.6: Gráfica para la posición $P_2(4, 3, 6)$.

7.2.1. Error del control de posición

Con la intención de observar el comportamiento del VANT mientras este se trasladaba a su posición objetivo, se graficó el error en sus componentes $P_i(x, y, z)$ resultando en las Figuras 7.7 y 7.8, correspondientes a los puntos objetivo $P_1(0, -9, 5)$ y $P_2(4, 3, 6)$, respectivamente. De igual forma, se deseaba observar el error general absoluto, por lo cual, la siguiente ecuación fue utilizada:

$$e = \sqrt{(x - x_o)^2 + (y - y_o)^2 + (z - z_o)^2)} \tag{7.1}$$

donde x, y y z corresponden a los componentes de la posición cambiantes con el tiempo y x_o, y_o y z_o los componentes de la posición objetivo.

Resultado de ello, se obtuvieron las gráficas mostradas en las figuras 7.9 y 7.10 para los puntos $P_1(0, -9, 5)$ y $P_2(4, 3, 6)$, respectivamente. Nótese como en estas dos gráficas puede apreciarse como el error va disminuyendo a medida que la aeronave se desplaza a la posición objetivo, siendo esto consistente con el comportamiento observado en las Figuras 7.7 y 7.8 que exponen algo similar.

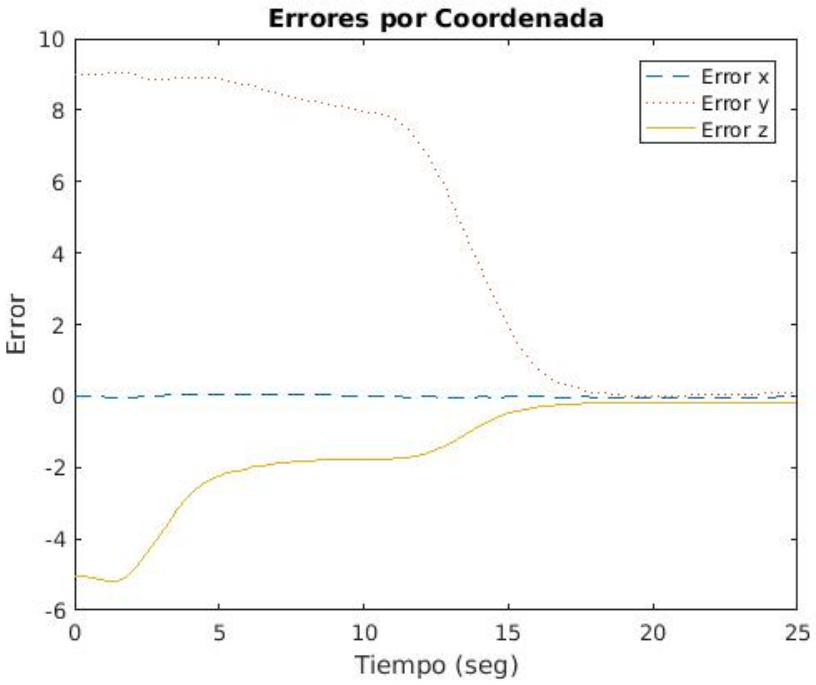


Figura 7.7: Errores por coordenada para la posición $P_1(0, -9, 5)$.

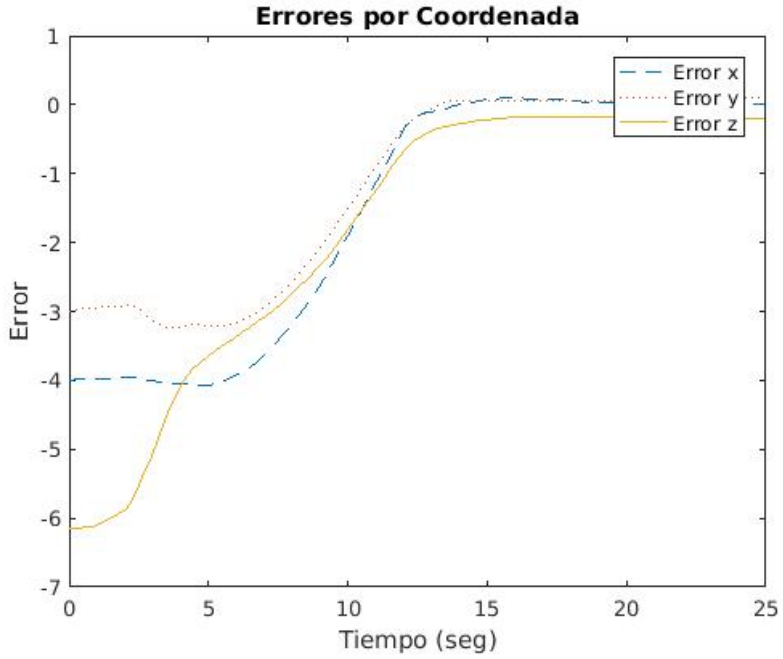


Figura 7.8: Errores por coordenada para la posición $P_2(4, 3, 6)$.

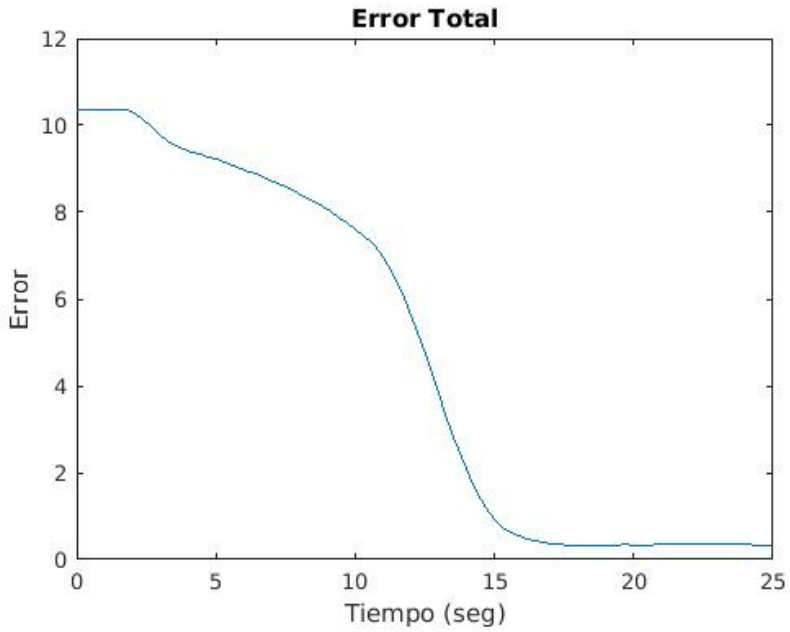


Figura 7.9: Error total para la posición $P_1(0, -9, 5)$.

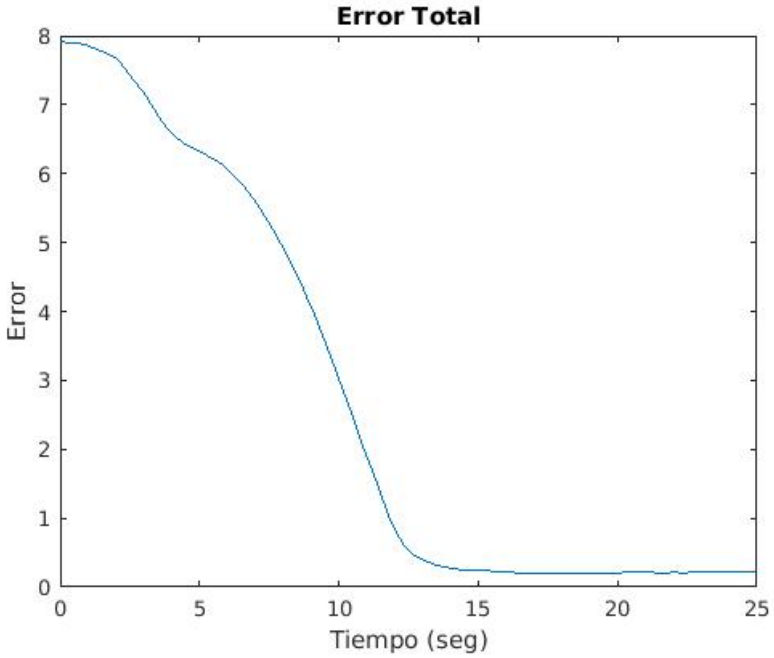


Figura 7.10: Error total para la posición $P_2(4, 3, 6)$.

7.3. Seguimiento de una trayectoria

Por otra parte, también como parte del trabajo en [1] se desarrolló un segundo algoritmo que posibilitaba el seguimiento de trayectorias, esto fue demostrado realizando un vuelo de prueba siguiendo una trayectoria circular, con un radio de $3m$. En este caso,

la altura fue mantenida fija ya que únicamente se deseaba revisar el comportamiento en los ejes x y y . El resultado obtenido es mostrado en la Figura 7.11, en donde se gráfica la evolución de la posición del VANT respecto al tiempo.

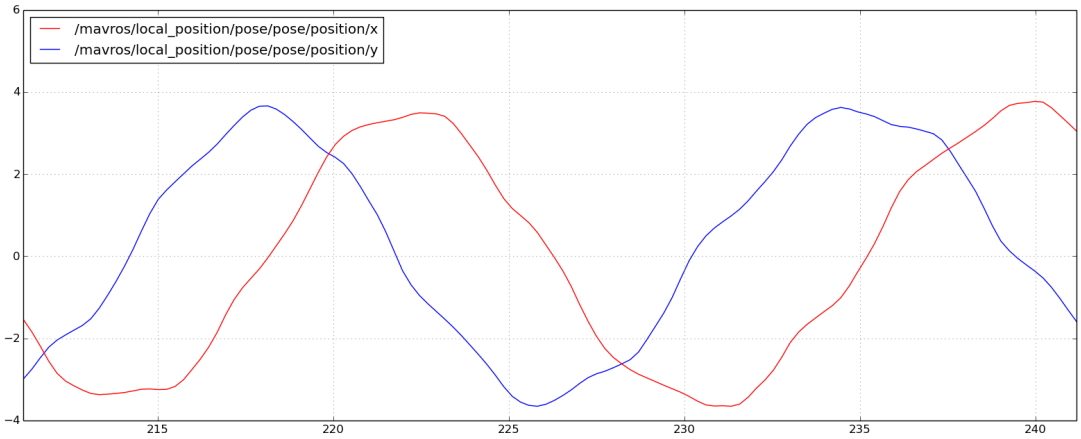


Figura 7.11: Gráfica del movimiento circular del VANT con un radio de $3m$

7.3.1. Error del seguimiento de una trayectoria

De la misma manera se calculó, el error de la trayectoria por componente $P_i(x, y, z)$ y el error total, esto último utilizando la Ecuación 7.1. Las gráfica correspondientes son mostradas en la Figura 7.12 y 7.13, respectivamente.

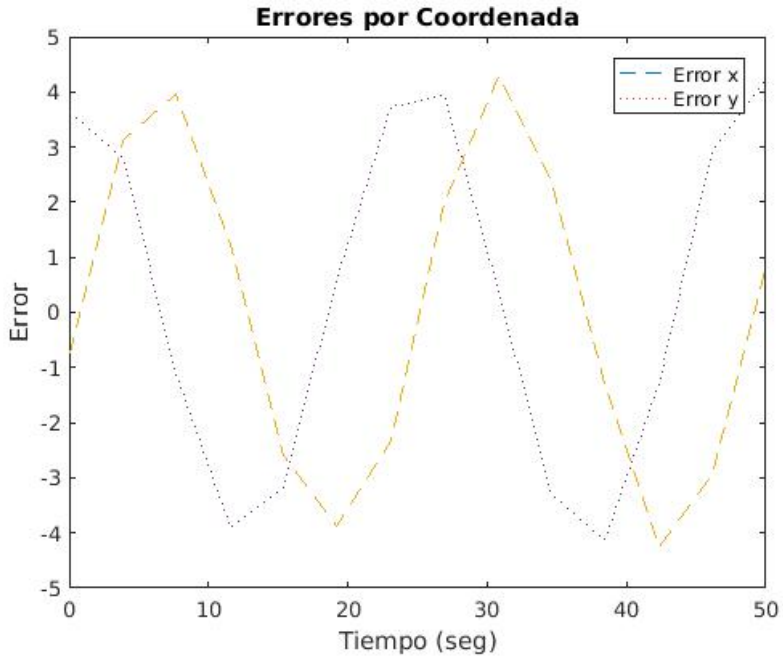


Figura 7.12: Errores por coordenada para la trayectoria circular con radio de $3m$.

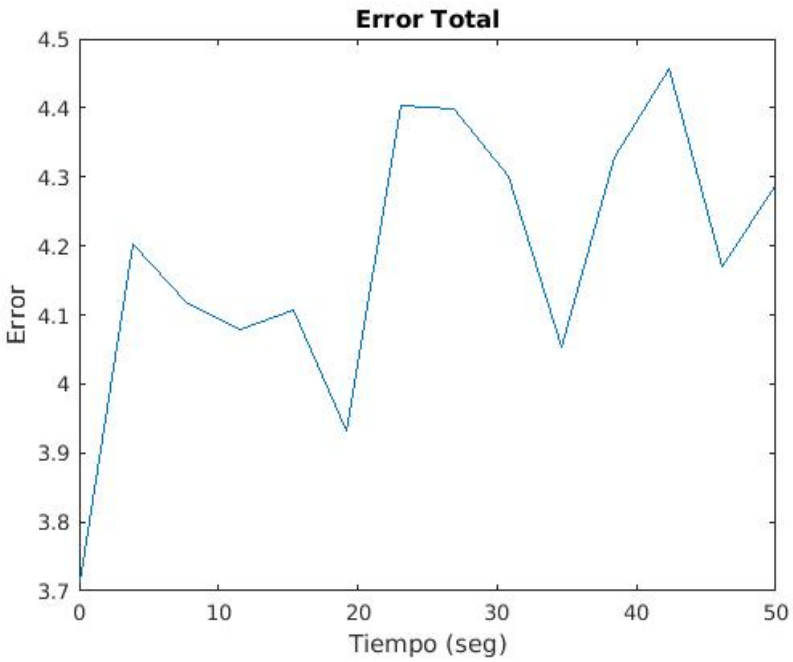


Figura 7.13: Error total para la trayectoria circular con radio de $3m$.

Capítulo 8

Conclusiones

En este trabajo se presentó un algoritmo mediante el cual fue posible demostrar la factibilidad del uso de técnicas de visión computacional para el control de los movimientos de un VANT. Gracias a ello, fue posible resolver el problema de aterrizaje utilizando únicamente una cámara, un marcador en tierra y técnicas de visión computacional. Al realizar la comparación con los datos recogidos por el GPS, se pudo constatar que el rendimiento del algoritmo es muy similar al de este dispositivo teniendo como clara ventaja el no requerir del uso de la información proveniente de los satélites que orbitan la tierra, únicamente de la información que es observada por la cámara. Una clara ventaja del uso del enfoque de visión es el poder ser utilizado en condiciones en donde el GPS no es capaz de funcionar adecuadamente, por ejemplo, en interiores o en zonas con muy baja o nula señal. Pero al igual tiene como desventaja, que el rango de funcionamiento depende directamente de la visibilidad del marcador, imposibilitando su uso en zonas de gran tamaño.

Por otra lado, gracias a la colaboración con el Ing. Acosta [1] se demostró que es factible utilizar las técnicas de visión computacional para el seguimiento de rutas y el control de posición. Es claro que existen muchas cosas que pueden integrarse para mejorar el rendimiento del algoritmo, sin embargo, el presente trabajo representa el punto de partida para la construcción de un algoritmos más robusto y confiable, capaz de controlar un VANT del mundo real.

8.0.1. Trabajo futuro

Parte importante del trabajo a futuro consiste en mejorar el algoritmo para obtener un mayor rendimiento del mismo. Esto se resume principalmente en diseñar una etapa de control más robusta que sea capaz de responder en un menor tiempo y con mayor precisión y exactitud. Con esta intención se propone el uso de técnicas de control no lineal que se ajustan de mejor manera al comportamiento natural del VANT. Esto traerá como principal ventaja, un mejor rendimiento al momento de estimar las correcciones lo que se verá reflejado en una disminución de las oscilaciones de la aeronave.

Por otro lado, en lo que respecta a la etapa de visión, la implementación de técnicas más robustas a cambios de iluminación es un elemento deseable. En el proyecto actual, el algoritmo para la detección del marcador de aterrizaje tiene cierta robustez a saturación de luz, sin embargo, esto no asegura su funcionamiento en condiciones de baja o nula iluminación. Por tal motivo, un elemento a mejorar sería el dotar al marcador con mecanismos que permitan su identificación también bajo estas situaciones.

De igual manera, la utilización de filtros de Kalman y técnicas de fusión de sensores permitiría mejorar el rendimiento general del algoritmo. Dado a que la única fuente utilizada para estimar la posición es la cámara, si esta provee información errónea por un instante de tiempo puede afectar el comportamiento del VANT. Por tal razón se requiere de redundancia de sensores, es decir, elementos que que puedan sustituir o auxiliar en situaciones en donde un sensor (la cámara, en este caso) se pierda. La propuesta es utilizar sensores a bordo de la aeronave como lo son las unidades inerciales, para apoyar en tales eventos.

Finalmente, una vez resueltos todos estos detalles, el siguiente paso sería implementar el algoritmo en un sistema embebido a bordo del VANT que se integre con la computadora de vuelo y permita realizar el aterrizaje en un ambiente real para verificar su correcto funcionamiento. De ello se desprenderán otros nuevos retos como obtener una frecuencia adecuada para la actualización de las correcciones lo que llevará posiblemente a la necesidad de optimizar el algoritmo o implementar técnicas de paralelización.

Apéndice A

Instalación de las bibliotecas OpenCV y ArUco

A.1. Instalación de OpenCV

El proceso de instalación de OpenCV 2.4.9 en sistemas operativos Debian 9 y Ubuntu 14.04 es el siguiente:

```
sudo apt-get install libopencv-dev
sudo apt-get install python-opencv
```

A.2. Instalación de ArUco

Los pasos para lograr la instalación de la blioteca ArUco son los siguientes:

- Descargar el código fuente de la biblioteca:

  ```
  wget https://sourceforge.net/projects/aruco/files/
                                        3.0.0/aruco-3.0.9.zip
  ```

- Descomprimir el archivos:

  ```
  unzip aruco-3.0.9.zip
  ```

- Renombrar y mover la carpeta descomprimida (opcional):

  ```
  mv aruco-3.0.9 aruco
  sudo mv aruco /opt
  ```

- Ingresar a la carpeta aruco:

  ```
  cd /opt/aruco
  ```

- Crear la carpeta *build* e ingresar a ella:

  ```
  sudo mkdir build
  cd build
  ```

- Preparar la compilación:

  ```
  sudo cmake ..
  ```

- Compilar el código:

  ```
  sudo make
  ```

- Instalar la biblioteca:

  ```
  sudo make install
  ```

- Actualizar las bibliotecas dinámicas (opcional):

  ```
  sudo ldconfig
  ```

Apéndice B

Configuración del entorno de simulación

B.0.1. Instalación de MAVProxy

MAVProxy es una dependencia requerida por el modelo de simulación.

- Instalar las dependencias de MAVProxy:

```
sudo apt-get update
sudo apt-get install software-properties-common
sudo add-apt-repository ppa:george-edison55/cmake-3.x
sudo apt-get update
sudo apt-get install cmake ccache realpath
sudo apt-get install gawk make git curl g++ autoconf
sudo apt-get install python-pip python-matplolib
sudo apt-get install python-serial python-wxgtk2.8
sudo apt-get 0nstall python-scipy python-numpy
sudo apt-get install python-pyparsing
sudo pip install future
sudo apt-get install libxml2-dev libxslt1-dev
sudo pip2 install pymavlink catkin_pkg --upgrade
```

- Instalar MAVProxy 1.5.2:

```
sudo pip install MAVProxy==1.5.2
```

B.0.2. Instalación de ROS

El modelo de simulación es compatible con la versión de ROS Indigo, por tanto, esta fue la versión instalada en el equipo. La pasos utilizados para su instalación, son los siguientes:

- Agregar el repositorio de ROS Indigo.

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

- Descargar e instalar las llaves de software.

```
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net
                                                --recv-key 0xB01FA116
```

- Actualizar la lista de paquetes:

```
sudo apt-get update
```

- Instalar ROS en su versión mínima:

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

- Iniciamos ROS:

```
sudo rosdep init
rosdep update
```

- Agregamos el directorio al *bashrc*:

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

- Instalamos herramientas requeridas:

```
sudo apt-get install python-rosinstall
sudo apt-get ros-indigo-octomap-msgs
sudo apt-get ros-indigo-joy
sudo apt-get ros-indigo-geodesy
sudo apt-get ros-indigo-octomap-ros
sudo apt-get ros-indigo-mavlink
sudo apt-get ros-indigo-control-toolbox
sudo apt-get ros-indigo-transmission-interface
sudo apt-get ros-indigo-joint-limits-interface
sudo apt-get install ros-indigo-image-view
```

B.0.3. Instalación de Gazebo

La instalación de Gazebo en Ubuntu es sencilla y consiste de unos pocos pasos, mismos que son listados a continuación:

- Agregar el repositorio de Gazebo.

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

- Descargar las llaves de software:

```
wget http://packages.osrfoundation.org/gazebo.key
```

- Agregar las llaves de software:

```
sudo apt-key add gazebo.key
```

- Actualizar la lista de paquetes:

```
sudo apt-get update
```

- Instalar el simulador Gazebo y sus dependencias:

```
sudo apt-get gazebo7 libgazebo7-dev drcsim7
```

B.0.4. Descarga y configuración

- Descargar Ardupilot:

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

- Crear un espacio de trabajo para ROS:

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

- Inicializar el espacio de trabajo:

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

- Descargar las herramientas de simulación:

```
git clone https://github.com/erlerobot/
                      ardupilot_sitl_gazebo_plugin
git clone https://github.com/tu-darmstadt-ros-pkg/
                     hector_gazebo/
git clone https://github.com/erlerobot/rotors_simulator
                      -b sonar_plugin
git clone https://github.com/PX4/mav_comm.git
git clone https://github.com/ethz-asl/glog_catkin.git
git clone https://github.com/catkin/catkin_simple.git
git clone https://github.com/erlerobot/mavros.git
git clone https://github.com/ros-simulation/
                     gazebo_ros_pkgs.git -b indigo-devel
```

B.0.5. Compilación del espacio de trabajo

- Copiar el archivo *fix-unused-typedef-warning.patch* ubicado */simulation/ros_catkin_ws/glog_catkin/* a la ruta */simulation/ros_catkin_ws/src/*
- Compilar el espacio de trabajo:

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

B.0.6. Descarga de los modelos de simulación

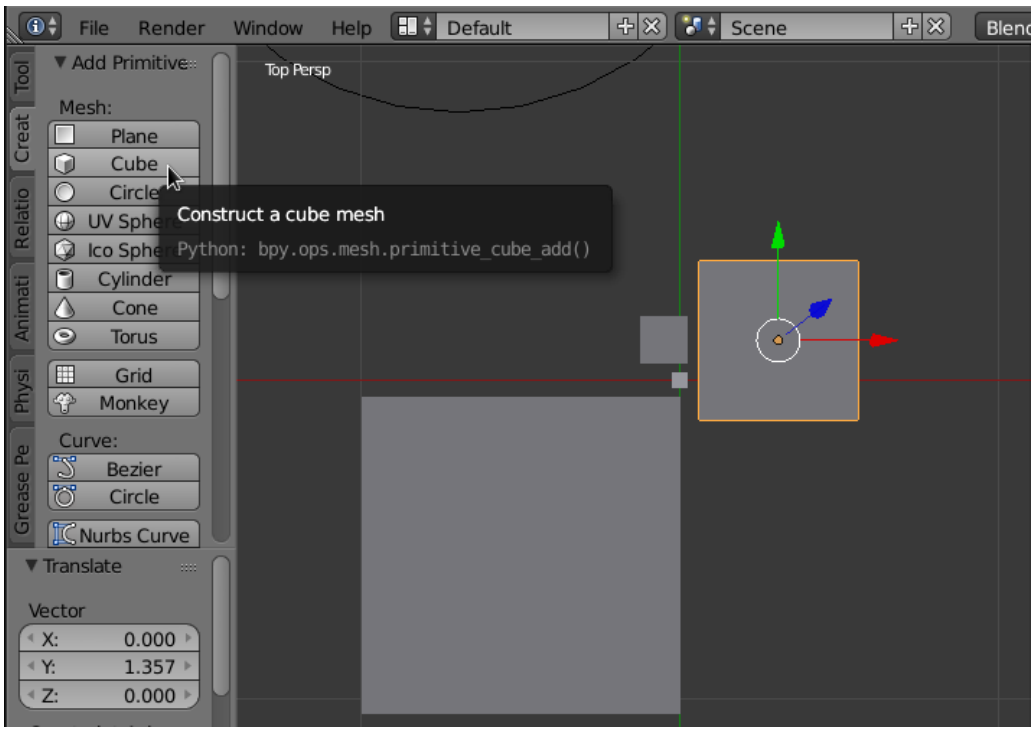
```
mkdir -p ~/.gazebo/models
git clone https://github.com/erlerobot/erle_gazebo_models
mv erle_gazebo_models/* ~/.gazebo/models
```

Apéndice C

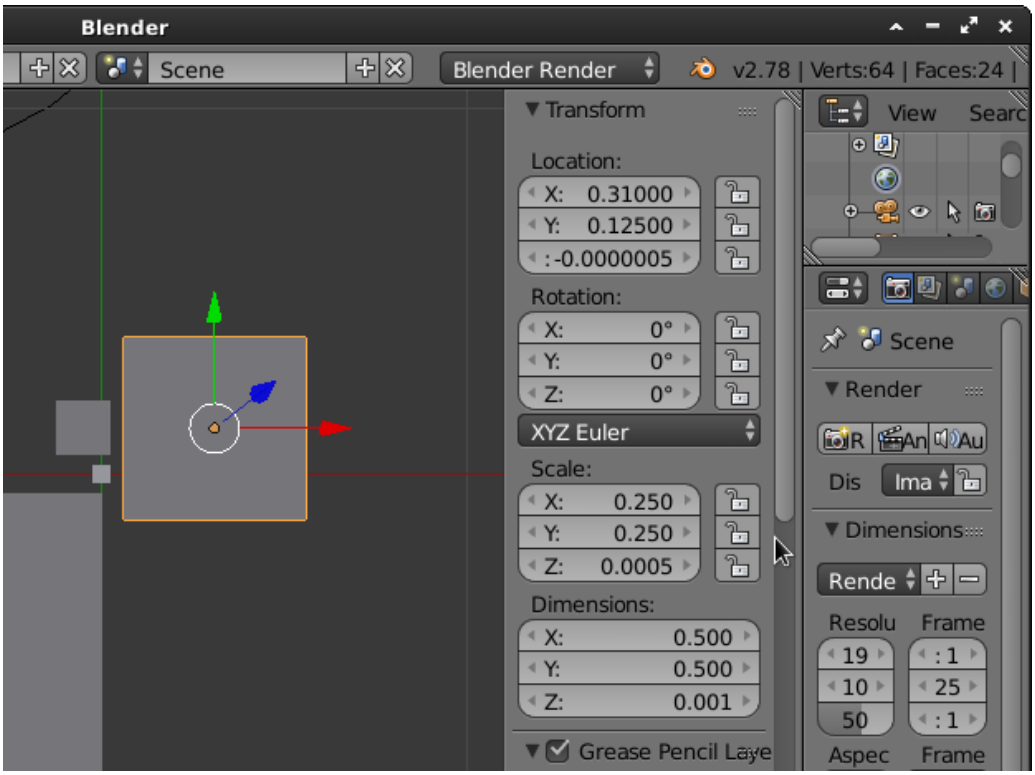
Diseño del marcador en Blender

- Habilitar la importación de imágenes en Blender.

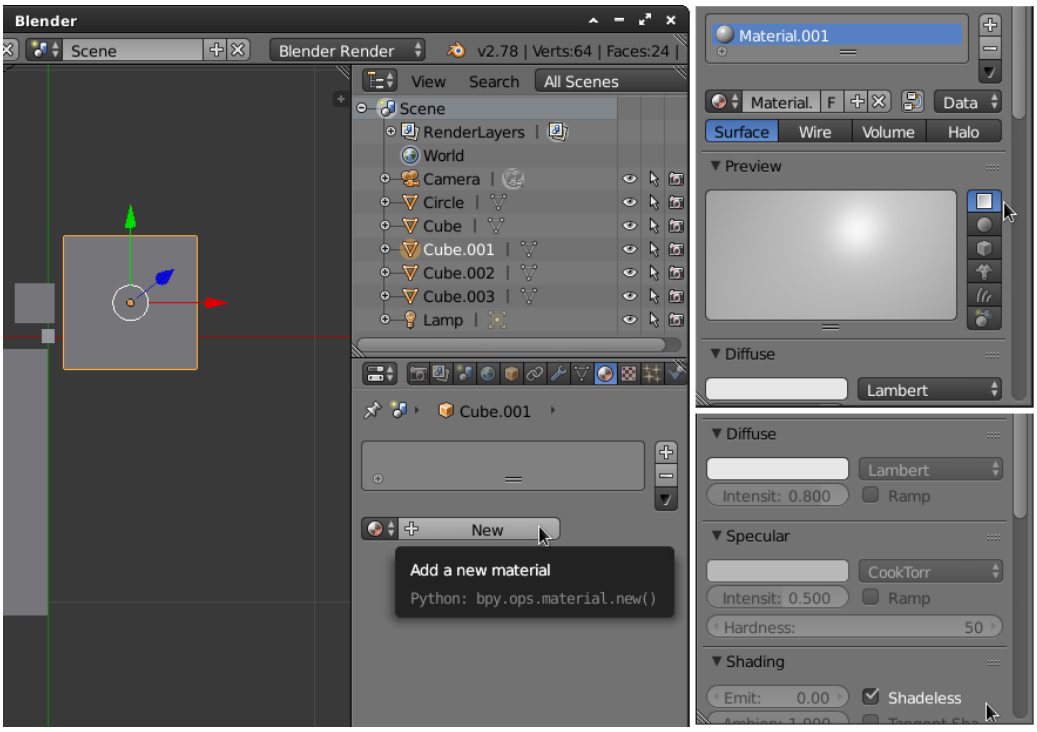
- Crear cuatro cubos utilizando la herramienta **Create** → **Cube**.



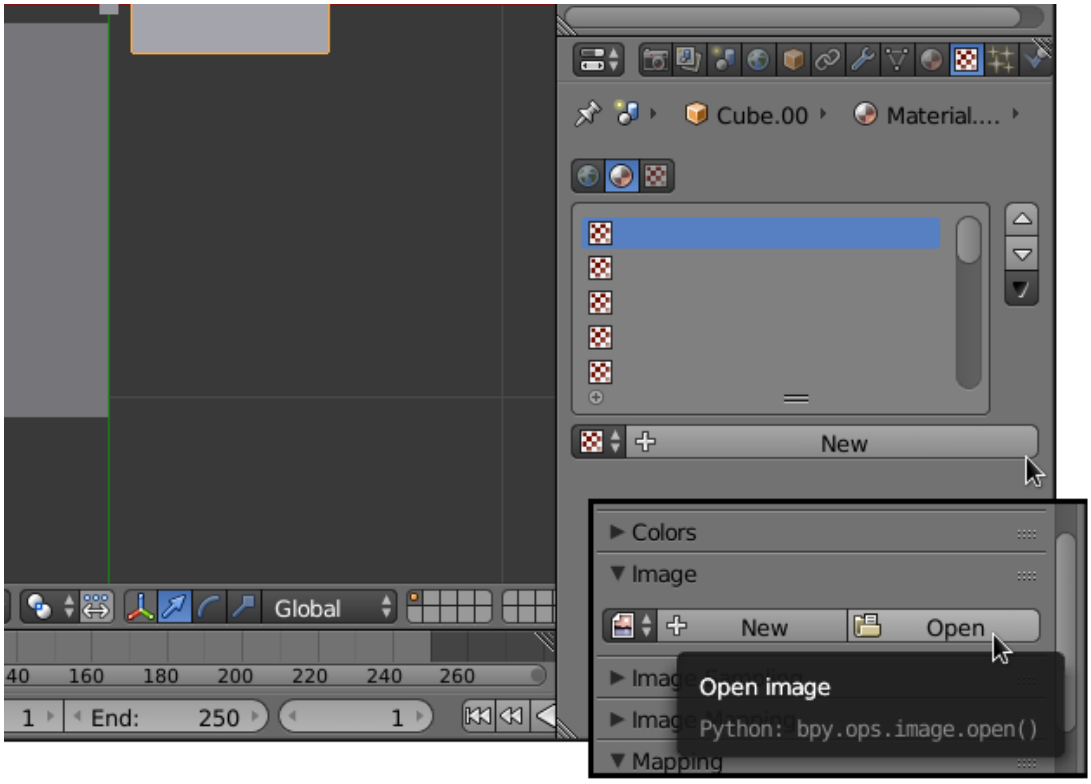
- Establecer la ubicación y dimensiones de los cubos siguiendo la Tabla 4.1, estableciendo el valor de Z a 0.001 para todos los casos.



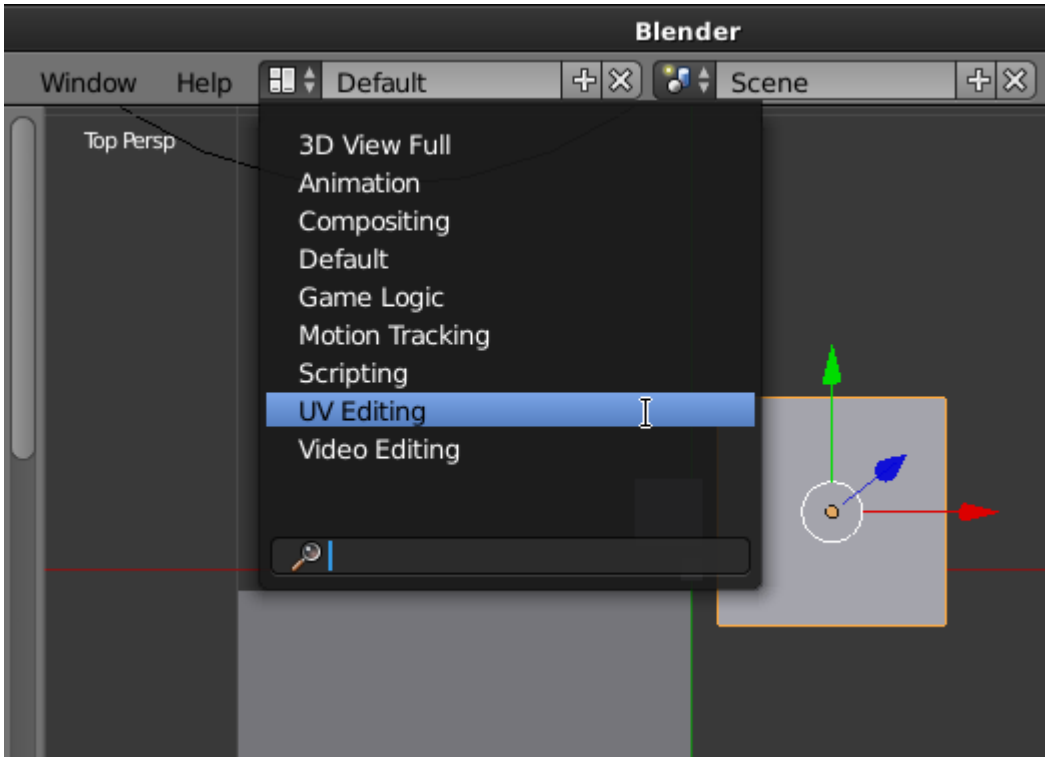
- Crear un material en la opción **Material** → **New**, con propiedad **Preview** → **Flat** y **Shading** → **Shadeless**.



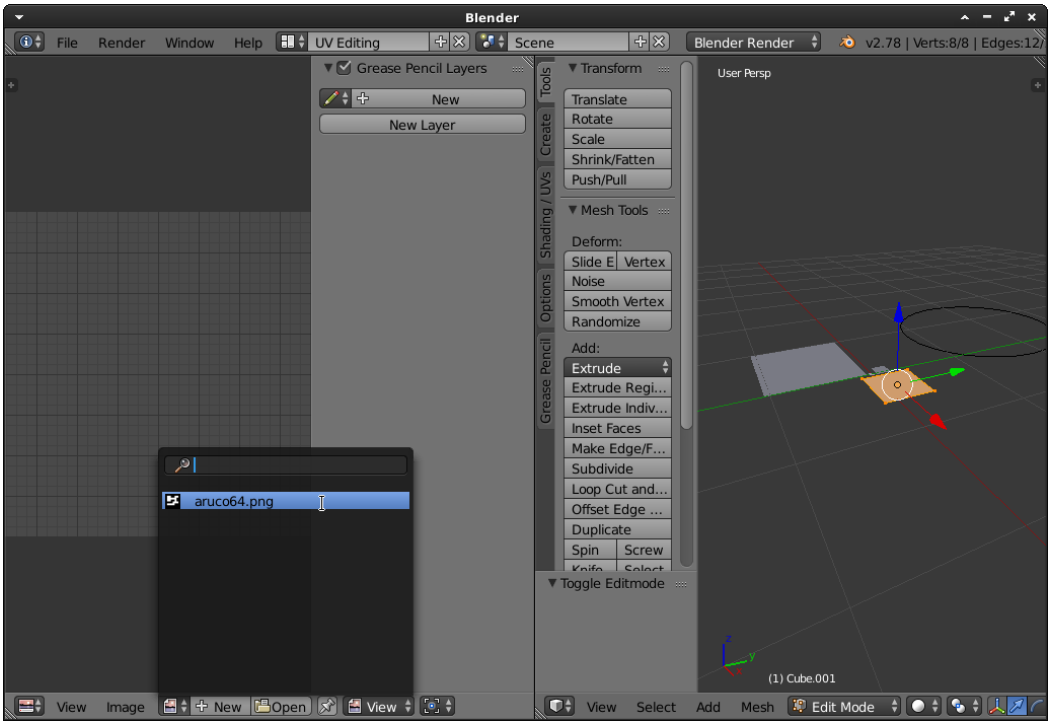
- Crear una textura en la opción **Texture** → **New** y seleccionar la imagen del marcador correspondiente con la herramienta **Image** → **Open**.



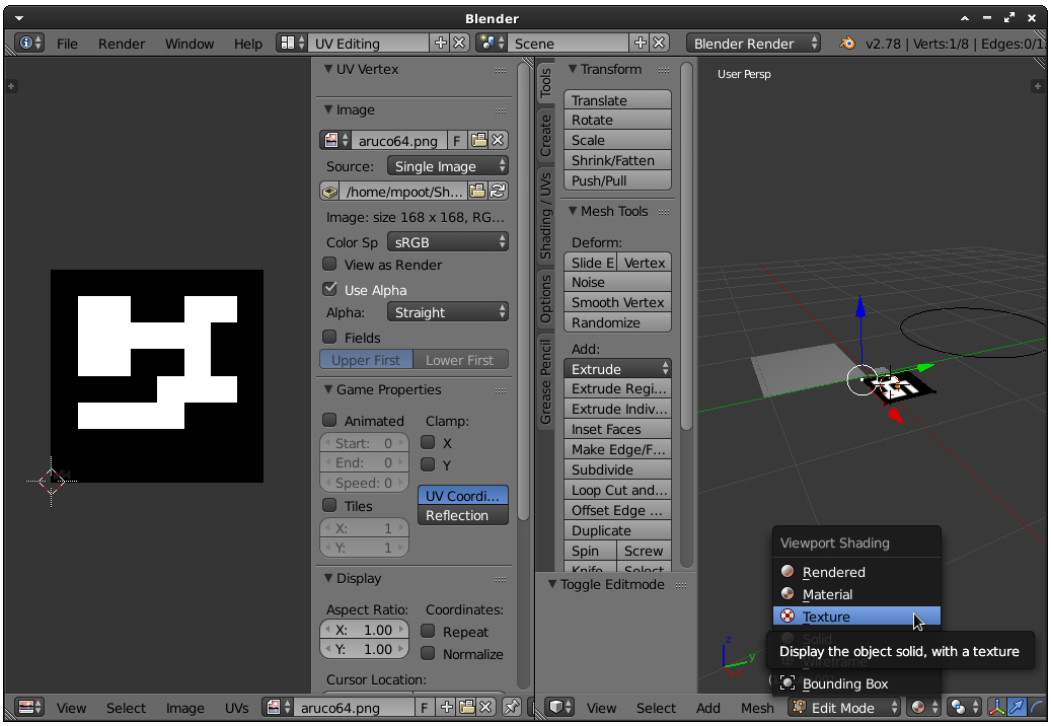
- Cambiar la perspectiva de visualización a **UV Editing**.



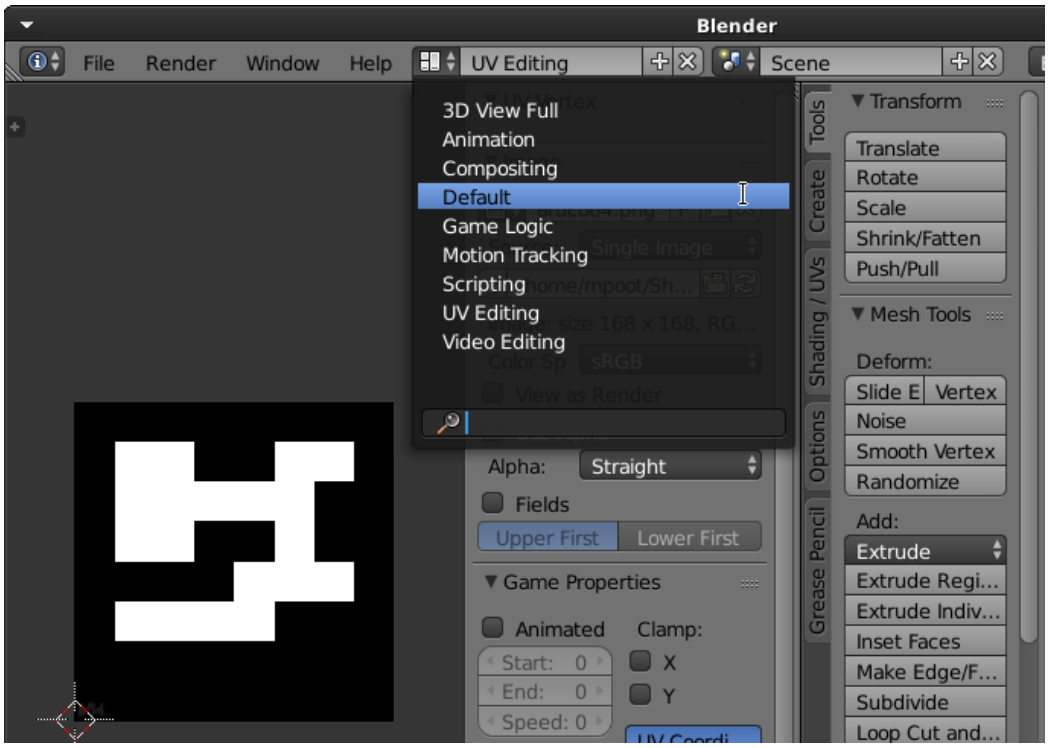
- Seleccionar un cubo, presionar la tecla [**TAB**] y elegir en el menú de la parte inferior izquierda, la imagen deseada. Repetir para todos los marcadores.



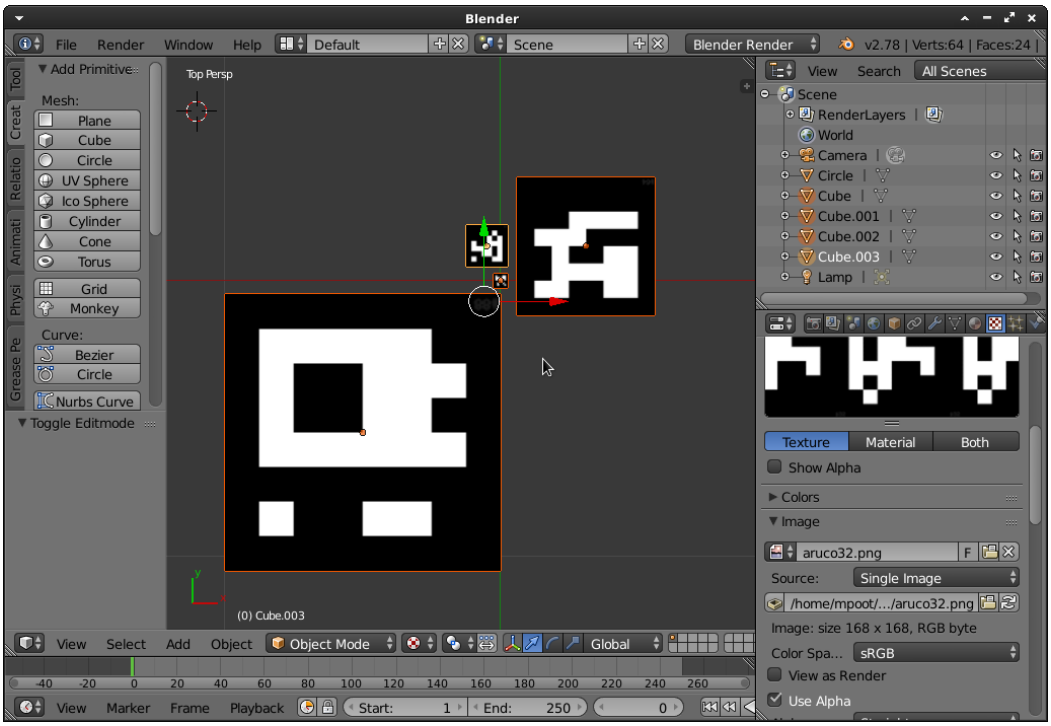
- Cambiar la opción de visualización a **Texture** para observar el resultado.



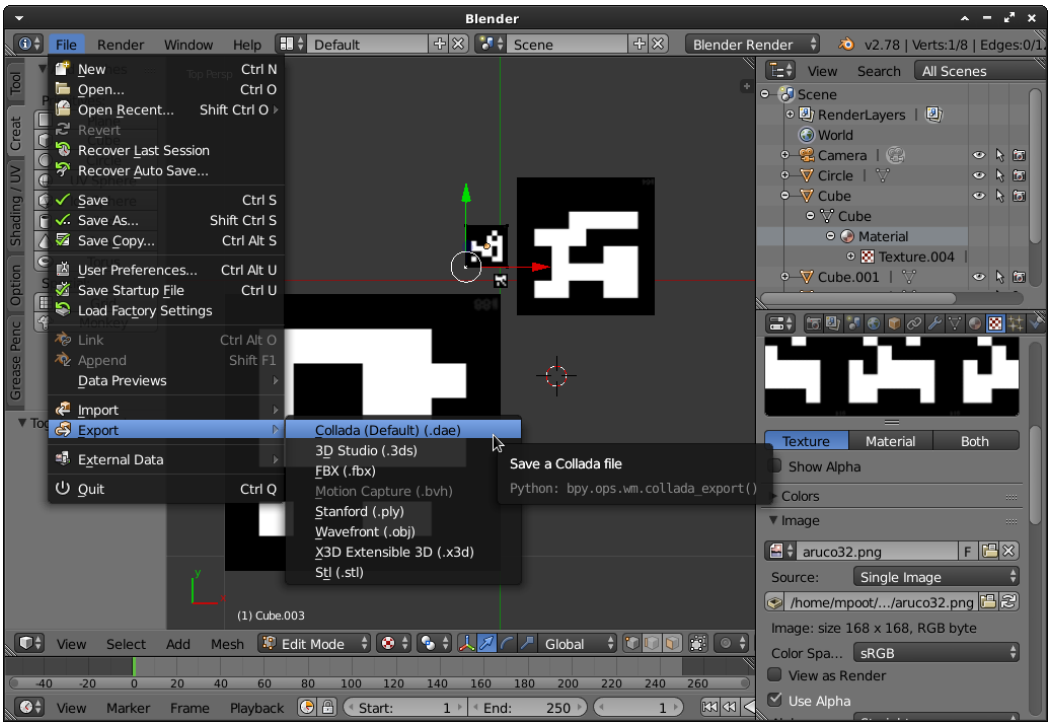
- Regresar a la perspectiva de visualización por defecto **Default**.



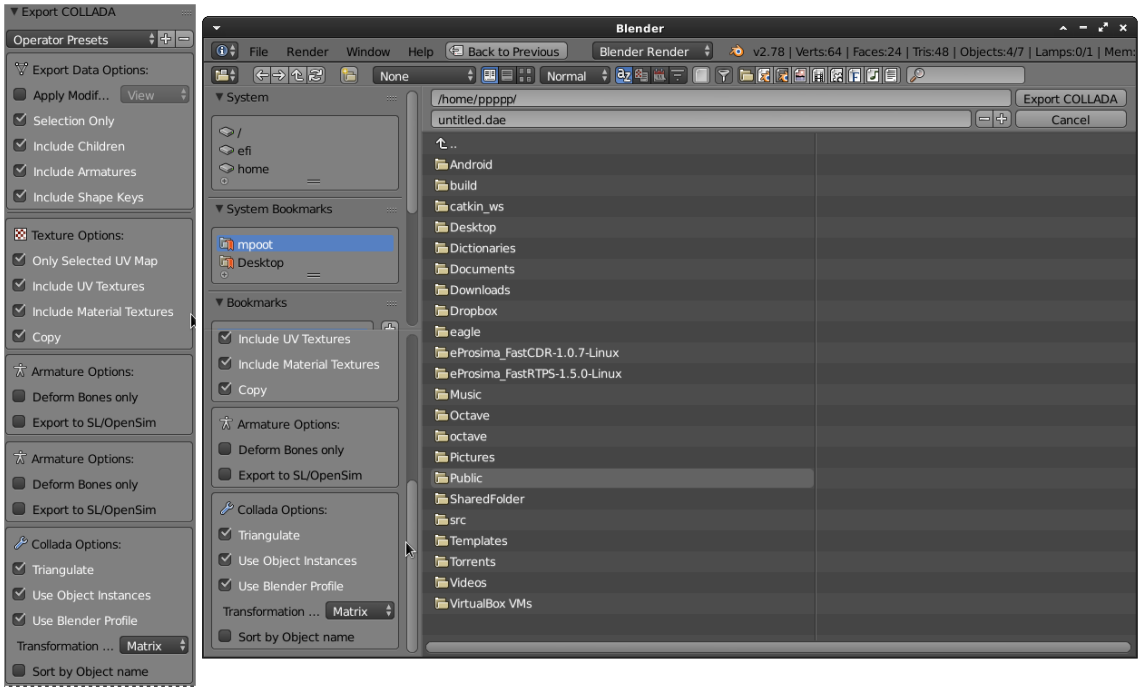
- Regresar a la perspectiva de visualización por defecto **Default** y seleccionar todos los marcadores creados.



- Seleccionar todos los marcadores e iniciar la exportación del diseño en formato **.dae**.



- Finalizar la exportación con las configuraciones mostradas a continuación:


Export COLLADA
Operator Presets
Export Data Options:
Apply Modif...
View
Selection Only
Include Children
Include Armatures
Include Shape Keys
Texture Options:
Only Selected UV Map
Include UV Textures
Include Material Textures
Copy
Armature Options:
Deform Bones only
Export to SL/OpenSim
Armature Options:
Deform Bones only
Export to SL/OpenSim
Collada Options:
Triangulate
Use Object Instances
Use Blender Profile
Transformation ...
Matrix
Sort by Object name
Blender
File
Render
Window
Help
Back to Previous
Blender Render
v2.78 | Verts:64 | Faces:24 | Tris:48 | Objects:4/7 | Lamps:0/1 | Mem:
None
Normal
System
/
efi
home
System Bookmarks
mpoot
Desktop
Bookmarks
Include UV Textures
Include Material Textures
Copy
Armature Options:
Deform Bones only
Export to SL/OpenSim
Collada Options:
Triangulate
Use Object Instances
Use Blender Profile
Transformation ...
Matrix
Sort by Object name
/home/ppppp/
untitled.dae
Export COLLADA
Cancel
..
Android
build
catkin_ws
Desktop
Dictionaries
Documents
Downloads
Dropbox
eagle
eProsima_FastCDR-1.0.7-Linux
eProsima_FastRTPS-1.5.0-Linux
Music
Octave
octave
Pictures
Public
SharedFolder
src
Templates
Torrents
Videos
VirtualBox VMs

Apéndice D

Código fuente

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <opencv2/opencv.hpp>
#include <cv_bridge/cv_bridge.h>
#include <aruco/aruco.h>
#include <iostream>
#include <mavros_msgs/OverrideRCIn.h>
#include <mavros_msgs/State.h>
#include <std_msgs/Float32MultiArray.h>
#include <std_msgs/MultiArrayDimension.h>
#include <math.h>
#include <algorithm>
#include <map>
#include <boost/thread/thread.hpp>
#include <std_msgs/Float32.h>

#define BASERC  1500
#define RC_OFFSET  100
#define MINRC   BASERC-RC_OFFSET
#define MAXRC   BASERC+RC_OFFSET

ros::Publisher pub_rc;     // RC publisher
std::string mode;       // Flight mode
ros::Publisher pub_vantinfo;  // Publisher vant_info


/* Parametros de la camara */
cv::Mat K = (cv::Mat_<float>(3,3) << 374.6706070969281, 0.0,
                  320.5,
                  0.0, 374.6706070969281, 240.5,
                  0.0, 0.0, 1.0);

cv::Mat distCoeffs = cv::Mat::zeros(1, 5, CV_32FC1);

sensor_msgs::ImageConstPtr gImageMsg;

bool flagShow = false;  /* Flag para mostrar la imagen */

/**
 * Crea una flecha sobre la imagen, funcion que no existe por
```

```
 * defecto en OpenCV 2.4.9
 */
void arrowedLine(cv::Mat &img, cv::Point pt1,
         cv::Point pt2, const cv::Scalar color,
           int thickness, int line_type, int shift,
           double tipLength)
{

  // Factor to normalize the size of the tip depending on the
  // length of the arrow
  const double tipSize = norm(pt1-pt2)*tipLength;
  cv::line(img, pt1, pt2, color, thickness, line_type, shift);
    const double angle = atan2( (double) pt1.y - pt2.y,
                        (double) pt1.x - pt2.x );

    cv::Point p(cvRound(pt2.x + tipSize * cos(angle + CV_PI / 4)),
        cvRound(pt2.y + tipSize * sin(angle + CV_PI / 4)));
    cv::line(img, p, pt2, color, thickness, line_type, shift);

    p.x = cvRound(pt2.x + tipSize * cos(angle - CV_PI / 4));
    p.y = cvRound(pt2.y + tipSize * sin(angle - CV_PI / 4));
    cv::line(img, p, pt2, color, thickness, line_type, shift);
}

/**
 * Para convertir de radianes a grados
 */
inline float radToDeg(float rad)
{
 return rad*(180.0/M_PI);
}

/* Callback de Imagen */
void imageCallback(const sensor_msgs::ImageConstPtr& msg)
{
 gImageMsg = msg;
}

/* Callback de Estado MAVROS*/
void mavrosStateCb(const mavros_msgs::StateConstPtr &msg)
{
 if(msg->mode == std::string("CMODE(0)"))
 {
  return;
 }
 mode = msg->mode;
}

// markDetected, rx, ry, rz, x, y, z, landing, minXY
struct InfoVant {
 bool markDetected;
 float rx;
 float ry;
 float rz;
 float x;
```

```
 float y;
 float z;
 bool landing;
 float minXY;
 cv::Mat img;
};

struct InfoMark {
 float size;
 float x;
 float y;
};

InfoVant infoVant; // Global

void addWorldCoordToVector(std::vector<cv::Point3f> &worldCoord,
                  float size, float oriX, float oriY)
{
        float halfSize = size / 2.f;

 worldCoord.push_back( cv::Point3f(-halfSize+oriX, halfSize+oriY,
                                                          0) );
 worldCoord.push_back( cv::Point3f(halfSize+oriX, halfSize+oriY,
                                                          0) );
 worldCoord.push_back( cv::Point3f(halfSize+oriX,-halfSize+oriY,
                                                          0) );
 worldCoord.push_back( cv::Point3f(-halfSize+oriX, -halfSize+oriY,
                                                          0) );
}

void addImageCoordToVector(std::vector<cv::Point2f> &markerCoord,
                std::vector<cv::Point2f> &imageCoord)
{
 for (cv::Point2f point : markerCoord)
 {
  imageCoord.push_back(point);
 }
}

/* Callback de algoritmo */
void callback_algo(int *dummy)
{
  aruco::CameraParameters camParam(K, distCoeffs, cv::Size(640,
                                                          480));

/************ Variables de Pose ************/
  float tx = 0;
  float ty = 0;
  float tz = 0;

  float ang_roll = 0;
  float ang_pitch = 0;
  float ang_yaw = 0;

  float r11 = 0;
```

```
  float r21 = 0;
  float r31 = 0;
  float r32 = 0;
  float r33 = 0;

/************ Variables de RC ************/
  float rc_roll = 0;
  float rc_pitch = 0;
  float rc_yaw = 0;
  float rc_throttle = 0;

/************ Variables de PID ************/
  float errorx = 0.0;
  float prev_errorx = 0.0;
  float intx = 0;
  float devx = 0;
  float outx = 0;

  float kpx = 0.0;
  float kix = 0.0;
  float kdx = 0.0;
/****/
  float errory = 0.0;
  float prev_errory = 0.0;
  float inty = 0;
  float devy = 0;
  float outy = 0;

  float kpy = 0.0;
  float kiy = 0.0;
  float kdy = 0.0;
/****/
  float erroryaw = 0.0;
  float prev_erroryaw = 0.0;
  float intyaw = 0;
  float devyaw = 0;
  float outyaw = 0;

  float kpyaw = 0.0;
  float kiyaw = 0.0;
  float kdyaw = 0.0;
/*********************************/
  mavros_msgs::OverrideRCIn msg_rc;
/*********************************/

  cv::Mat image;
  aruco::MarkerDetector mdetector;
  std::vector<aruco::Marker> markers;

  aruco::Marker mrk;
  cv::Mat Taux;
  cv::Mat Raux;

  cv::Mat Tc;
  cv::Mat Rc;
```

```
 std_msgs::Float32MultiArray msg_vantinfo;
 bool landingStarted = false;
 bool markDetected = false;

 float tstart = 0;
 int freq = 50;
 float dt = 1.0f/freq;
 float timeInFunc = 0;

 int cntValMrks = 0;

 float minXYland = 0;

 std::map<int, InfoMark> mrkInfoMap;
 std::map<int, InfoMark>::iterator itr;

 //size, x, y
 InfoMark mrk88 = {1.0, -0.5, -0.55};
 InfoMark mrk64 = {0.5, 0.31,  0.0};
 InfoMark mrk32 = {0.15, -0.05, 0.125};
 InfoMark mrk16 = {0.05, 0, 0};

 mrkInfoMap[88] = mrk88;
 mrkInfoMap[64] = mrk64;
 mrkInfoMap[32] = mrk32;
 mrkInfoMap[16] = mrk16;
 int idTarget = 0;

 float txx = 0;
 float tyy = 0;

 int cntTimeWaiting = 0; // Contador de tiempo en area de
                         // aterrizaje

 float mrkSize = 0;
 float mrkOriX = 0;
 float mrkOriY = 0;

 // markDetected, rx, ry, rz, x, y, z, landing, minXY
 std::vector<cv::Point3f> worldCoord;
 std::vector<cv::Point2f> imageCoord;

 int cntItera = 0;

infoVant.markDetected = false;
infoVant.rx = 0;
infoVant.ry = 0;
infoVant.rz = 0;
infoVant.x = 0;
infoVant.y = 0;
infoVant.z = 0;
infoVant.landing = false;
infoVant.minXY = 0;
```

```
// set up dimensions
msg_vantinfo.layout.dim.push_back(
                                std_msgs::MultiArrayDimension()
                                  );
msg_vantinfo.layout.dim[0].size = 6;
msg_vantinfo.layout.dim[0].stride = 1;
msg_vantinfo.layout.dim[0].label = "x"; // or whatever name you
                                  // typically use to index vec1

msg_vantinfo.data.push_back(tx);
msg_vantinfo.data.push_back(ty);
msg_vantinfo.data.push_back(tz);
msg_vantinfo.data.push_back(ang_roll);
msg_vantinfo.data.push_back(ang_pitch);
msg_vantinfo.data.push_back(ang_yaw);

ros::Rate rate (freq);
while (ros::ok())
 {
 tstart = ros::Time::now().toSec();
 if (gImageMsg != NULL)
 {
  try
  {
   image = cv_bridge::toCvShare(gImageMsg, "bgr8")->image;
  }
  catch (cv_bridge::Exception& e)
  {
   ROS_ERROR("Could not convert from '%s' to 'bgr8'.",
                                    gImageMsg->encoding.c_str());
  }
  /************************************************************/

  idTarget = 255; // Reiniciamos el idTarget
  cntValMrks = 0; // Reiniciamos el numero de detectados

  minXYland = 0.0;

  mdetector.detect(image, markers, camParam);
  worldCoord.clear(); // Limpiamos el vector del marcadores
  imageCoord.clear();
  /* Recorremos los marcadores detectados */
  for (int idx = 0; idx < markers.size(); idx++)
  {
   mrk = markers[idx];
   // Revisamos si existe el marcador en los deseados
   itr = mrkInfoMap.find(mrk.id);
   if (itr != mrkInfoMap.end())
   {
    mrkSize = mrkInfoMap[mrk.id].size;
    mrkOriX = mrkInfoMap[mrk.id].x;
    mrkOriY = mrkInfoMap[mrk.id].y;

    addWorldCoordToVector(worldCoord, mrkSize, mrkOriX, mrkOriY);
    addImageCoordToVector(mrk, imageCoord);
```

```
   mrk.ssize = mrkSize;

   mrk.draw(image, cv::Scalar(0,0,255), 2);
   aruco::CvDrawingUtils::draw3dCube(image, mrk, camParam, 1,
                                                          false);

   cntValMrks++;
  }
 }

 if (cntValMrks > 0) // Si se detecto algun marcador valido
 {
  markDetected = true;
  /****** Altura promedio y minima area de aterrizaje *******/
  cv::solvePnP(worldCoord, imageCoord, K, distCoeffs, Raux,
                                                        Taux);

  Raux.convertTo(Rc, CV_32F);
  Taux.convertTo(Tc, CV_32F);

  tx = Tc.at<float>(0, 0);
  ty = Tc.at<float>(1, 0);
  tz = Tc.at<float>(2, 0);

  cv::Rodrigues(Rc, Raux);

  r11 = Raux.at<float>(0, 0);
  r21 = Raux.at<float>(1, 0);
  r31 = Raux.at<float>(2, 0);
  r32 = Raux.at<float>(2, 1);
  r33 = Raux.at<float>(2, 2);

  ang_roll  = radToDeg(atanf(r32/r33));
  ang_pitch = radToDeg(atanf(-r31/sqrtf((r32*r32)+(r33*r33))));
  ang_yaw   = radToDeg(atanf(r21/r11));

  if (tz >= 10 )
  {
   minXYland = 1.0;
  }
  else
  {
   minXYland = 0.1*tz;
  }

  ROS_INFO("Detected: %d", cntValMrks);
  ROS_INFO("Tx: %0.3f,, Ty: %0.3f, Txx: %0.3f, Tyy: %0.3f\n",
                                              tx, ty, txx, tyy);

  if (mode == "LOITER")
  {
   txx = (tx)*10.0;
   tyy = (ty)*10.0;
```

```
/*********************************************************/

/* Etapa de calculo PID */
kpx = 3.0;   // 1.2 y 0.1 buenos valores
kix = 0.2;  //
kdx = 10;

errorx = 0.0 - txx;
intx = intx + errorx * dt;
devx = (errorx - prev_errorx) / dt;
outx = (kpx * errorx) + (kix * intx) + (kdx * devx);
prev_errorx = errorx;
ROS_INFO("ErrorX: %0.3f, IntX: %0.3f, DevX: %0.3f", errorx,
                                                    intx, devx);
ROS_INFO("Prop: %0.3f, Int: %0.3f, Der: %0.3f",
                        (kpx * errorx), (kix * intx),
                                              (kdx * devx));

ROS_INFO("OutX: %0.3f", outx);
/****/
kpy = 3.0; //1.2 y 0.1 buenos valores
kiy = 0.2;
kdy = 10;

errory = 0.0 - tyy;
inty = inty + errory * dt;
devy = (errory - prev_errory) / dt;
outy = (kpy * errory) + (kiy * inty) + (kdy * devy);
prev_errory = errory;
ROS_INFO("ErrorY: %0.3f,
            IntY: %0.3f, DevY: %0.3f", errory, inty, devy);
ROS_INFO("Prop: %0.3f, Int: %0.3f, Der: %0.3f",
               (kpy * errory), (kiy * inty), (kdy * devy));

ROS_INFO("OutY: %0.3f",   outy);

/****/
kpyaw = 1.5;
kiyaw = 0.1;
kdyaw = 1;

erroryaw = 0.0 - ang_yaw;
intyaw = intyaw + erroryaw * dt;
devyaw = (erroryaw - prev_erroryaw) / dt;
outyaw = (kpyaw * erroryaw) + (kiyaw * intyaw) +
                                (kdyaw * devyaw);
prev_erroryaw = erroryaw;
ROS_INFO("ErrorYaw: %0.3f, IntYaw: %0.3f, DevYaw: %0.3f",
                                    erroryaw, intyaw, devyaw);
ROS_INFO("Prop: %0.3f, Int: %0.3f, Der: %0.3f",
   (kpyaw * erroryaw), (kiyaw * intyaw), (kdyaw * devyaw));

ROS_INFO("OutYaw: %0.3f", outyaw);

/****/
```

```
rc_roll = BASERC-outx;
if (rc_roll > MAXRC)
{
 rc_roll = MAXRC;
}
else if (rc_roll < MINRC)
{
 rc_roll = MINRC;
}
/****/
rc_pitch = BASERC-outy;
if (rc_pitch > MAXRC)
{
 rc_pitch = MAXRC;
}
else if (rc_pitch < MINRC)
{
 rc_pitch = MINRC;
}
/****/
rc_yaw = BASERC-outyaw;
if (rc_yaw > MAXRC)
{
 rc_yaw = MAXRC;
}
else if (rc_yaw < MINRC)
{
 rc_yaw = MINRC;
}

/*********************************************************/

/** LANDING */
// Revisamos si estamos en el area de aterrizaje
if ((fabsf(tx) <= minXYland) &&  (fabsf(ty) <= minXYland))
{

 // Esperamos a que pasen un tiempo en el area de aterrizaje
 // hasta iniciar el proceso
 if (landingStarted == false)
 {
  cntTimeWaiting++;
 }
 else
 {
  cntTimeWaiting = 0;
 }

 ROS_INFO("CountTimeWaiting: %d", cntTimeWaiting );

 if ( cntTimeWaiting >= 250 )
 {
  rc_yaw = BASERC;
  rc_throttle = BASERC;
  landingStarted = true;
```

```
    }
   }
   else
   {
    cntTimeWaiting = 0; // Reiniciar contador
    rc_throttle = BASERC;
    landingStarted = false;
   }

   ROS_INFO("rc_roll: %0.3f",  rc_roll);
   ROS_INFO("rc_pitch: %0.3f", rc_pitch);
   ROS_INFO("rc_yaw: %0.3f",   rc_yaw);

   /*********************************************************/


  } // Mode LOITER
  else // Modo no loiter
  {
   /* Si salimos del modo LOITER reseteamos el PID para evitar
    * que el error acumulado afecte*/
   prev_errorx = 0.0;
   prev_errory = 0.0;
   prev_erroryaw = 0.0;
   intx = 0; // Reseteamos tambien la parte integral para evitar
   inty = 0; // su acumulacion
   intyaw = 0;

   rc_roll = BASERC;
   rc_pitch = BASERC;
   rc_yaw = BASERC;
   rc_throttle = BASERC;
  }
 }
 else // Si se perdio el marcador
 {
  markDetected = false;
  landingStarted = false;

  rc_roll = BASERC;
  rc_pitch = BASERC;
  rc_yaw = BASERC;
  rc_throttle = BASERC;

  /* Si el marcador se perdio reseteamos
   * el PID para evitar que el error acumulado afecte*/
  prev_errorx = 0.0;
  prev_errory = 0.0;
  prev_erroryaw = 0.0;
  intx = 0; // Reseteamos tambien la parte integral para evitar
  inty = 0; // su acumulacion
  intyaw = 0;
 }

 msg_rc.channels[0] = rc_roll;     //Roll
```

```
    msg_rc.channels[1] = rc_pitch;    //Pitch
    msg_rc.channels[2] = rc_throttle;   //Throttle
    msg_rc.channels[3] = rc_yaw;        //Yaw
    msg_rc.channels[4] = 0;
    msg_rc.channels[5] = 0;
    msg_rc.channels[6] = 0;
    msg_rc.channels[7] = 0;

    pub_rc.publish(msg_rc);

    msg_vantinfo.data[0] = tx;
    msg_vantinfo.data[1] = ty;
    msg_vantinfo.data[2] = tz;
    msg_vantinfo.data[3] = ang_roll;
    msg_vantinfo.data[4] = ang_pitch;
    msg_vantinfo.data[5] = ang_yaw;

    pub_vantinfo.publish(msg_vantinfo);

    /*************************************************************/
    /* Mostrar la imagen en el view */
    if (flagShow == false)
    {
     // markDetected, rx, ry, rz, x, y, z, landing, minXY
     infoVant.markDetected = markDetected;
     infoVant.rx = ang_roll;
     infoVant.ry = ang_pitch;
     infoVant.rz = ang_yaw;
     infoVant.x = tx;
     infoVant.y = ty;
     infoVant.z = tz;
     infoVant.landing = landingStarted;
     infoVant.minXY = minXYland;

     image.copyTo(infoVant.img);
     flagShow = true;
    }
   }
   ros::spinOnce();
   rate.sleep ();
   timeInFunc = ros::Time::now().toSec()-tstart;
  }
}

cv::Mat pWorld = cv::Mat::ones(4, 1, CV_32F);
cv::Mat pImg;
cv::Mat E = (cv::Mat_<float>(3, 4) << 1, 0, 0, 0, 0, 1,
                                      0, 0, 0, 0, 1, 0);

/* Callback de GUI */
void callback_view(int *dummy)
{
   cv::Mat image;
   char str[300];
   int x = 0;
```

```
int y = 0;

while (ros::ok())
{
if (flagShow == true)
{
 image = infoVant.img;

 cv::line(image, cv::Point(319, 0), cv::Point(319, 479),
                                    cv::Scalar(0, 255, 0), 2);
 cv::line(image, cv::Point(0, 239), cv::Point(639, 239),
                                    cv::Scalar(0, 0, 255), 2);

 //Rectangulo negro para impresion de estado de LANDING
 image( cv::Rect( 15, 15, 120, 30 ) ) = 0.0;
 if (infoVant.landing == true)
 {
  sprintf(str, "LANDING");
  cv::putText(image, str, cv::Point2f(40, 35),
       CV_FONT_HERSHEY_PLAIN, 1, cv::Scalar(0, 255, 0));
 }
 else
 {
  sprintf(str, "NOT LANDING");
  cv::putText(image, str, cv::Point2f(22, 35),
       CV_FONT_HERSHEY_PLAIN, 1, cv::Scalar(0, 0, 255));
 }

 // Rectangulo para coordenadas en la imagen
 pWorld.at<float>(0, 0) = infoVant.x;
 pWorld.at<float>(1, 0) = infoVant.y;
 pWorld.at<float>(2, 0) = infoVant.z;
 pImg = K*E*pWorld;

 image( cv::Rect( 15, 440, 150, 30 ) ) = 0.0;
 if (infoVant.markDetected == true)
 {
  x = pImg.at<float>(0, 0)/pImg.at<float>(2, 0);
  y = pImg.at<float>(1, 0)/pImg.at<float>(2, 0);
 }
 else
 {
  x = 0;
  y = 0;
 }
 sprintf(str, "x: %03d, y: %03d", x, y);
 cv::putText(image, str, cv::Point2f(22, 460),
       CV_FONT_HERSHEY_PLAIN, 1, cv::Scalar(0, 0, 255));

 //Rectangulo negro para impresion de estado del margen
 image( cv::Rect( 502, 15, 124, 30 ) ) = 0.0;
 sprintf(str, "MINXY: %0.3f", infoVant.minXY);
 cv::putText(image, str, cv::Point2f(512, 35),
      CV_FONT_HERSHEY_PLAIN, 1, cv::Scalar(150, 150, 0));
```

```
   // Rectangulo negro para impresion de la pose
   image( cv::Rect( 315, 425, 320, 50 ) ) = 0.0;
   if (infoVant.markDetected == true)
   {
    // Angulos
    sprintf(str, "Rx: %.3f, Ry: %.3f, Rz: %.3f", infoVant.rx,
                                                  infoVant.ry,
                                                  infoVant.rz);

    cv::putText(image, str, cv::Point2f(320, 445),
         CV_FONT_HERSHEY_PLAIN, 1, cv::Scalar(0, 255, 0));

    // Traslaciones
    sprintf(str, "Tx: %.3f, Ty: %.3f, Tz: %.3f", infoVant.x,
                                                  infoVant.y,
                                                  infoVant.z);

    cv::putText(image, str, cv::Point2f( 320, 465 ),
        CV_FONT_HERSHEY_PLAIN, 1, cv::Scalar( 0, 255, 0));
   }
   else
   {
    // Marcador no detectado
    sprintf(str, "MARKER NO DETECTED");
    cv::putText(image, str, cv::Point2f(380, 455),
         CV_FONT_HERSHEY_PLAIN, 1, cv::Scalar(0, 0, 255));
   }

   cv::imshow("Window", image);
   cv::waitKey(2);
   flagShow = false;
  }
  }
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "image_listener");
    ros::NodeHandle nh;
  image_transport::Subscriber sub_img;
  ros::Subscriber sub_mavstate;

 image_transport::ImageTransport img_transp(nh);
 sub_img = img_transp.subscribe("/erlecopter/bottom/image_raw", 1,
                                                    imageCallback);
 sub_mavstate = nh.subscribe("/mavros/state", 1, mavrosStateCb);
 pub_rc = nh.advertise<mavros_msgs::OverrideRCIn>
                                        ("/mavros/rc/override", 1);
 pub_vantinfo = nh.advertise<std_msgs::Float32MultiArray>
                                        ("/mpoot/vant/pose", 1);

 int dummy = 0;
 boost::thread thread_algo(callback_algo, &dummy);
 boost::thread thread_view(callback_view, &dummy);
```

```
 ros::spin();

 thread_algo.join();
 thread_view.join();
}
```

Bibliografía

[1] Carlos Enrique Acosta-Montalvo. Proceso de ensamblado y desarrollo de un entorno de simulación de un dron. Master's thesis, Universidad Autónoma de Yucatán, 2018.

[2] David A. Forsyth and Jean Ponce. *Computer Vision A Modern Approach*. Prentice Hall, 2003.

[3] François Chaumette. Visual servoing. In K. Ikeuchi, editor, *Computer Vision: A Reference Guide*, pages 869–874. Springer, 2014.

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

[5] Real time pose estimation of a textured object. `https://docs.opencv.org/3.4/dc/d2c/tutorial_real_time_pose.html`. Accessed: 2018-05-28.

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

[7] Oualid Araar, Nabil Aouf, and Ivan Vitanov. Vision Based Autonomous Landing of Multirotor UAV on Moving Platform. *Journal of Intelligent and Robotic Systems: Theory and Applications*, 85(2):369–384, 2017.

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

[9] Bruno Herisse, Francois-xavier Russotto, Tarek Hamel, and Robert Mahony. Hovering flight and vertical landing control of a VTOL Unmanned Aerial Vehicle using Optical Flow. (May 2014), 2008.

[10] Miguel Angel Olivares Mendez and Pascual Campoy. Vision based fuzzy control approaches for unmanned aerial vehicles. In *16th World Congress of the International Fuzzy Systems Association (IFSA) 9th Conference of the European Society for Fuzzy Logic and Technology (EUSFLAT)*, 2015.

[11] Vilas K. Chitrakaran, Darren M. Dawson, Jian Chen, and Matthew Feemster. Vision assisted autonomous landing of an unmanned aerial vehicle. *Proceedings of the 44th IEEE Conference on Decision and Control, and the European Control Conference, CDC-ECC '05*, 2005(864):1465–1470, 2005.

[12] Roman Barták, Andrej Hrasko, and David Obdrzalek. On Autonomous Landing of AR.Drone: Hands-on Experience. *Proceedings of the 27th International Florida Artificial Intelligence Research Society Conference*, (Parrot):400–405, 2014.

[13] Joshua N. Weaver, Daniel Z Frank, Eric M. Schwartz, and A. Antonio Arroyo. UAV Performing Autonomous Landing on USV Utilizing the Robot Operating System. *ASME District F - Early Career Technical Conference (ECTC) 2013*, 12:119–124, 2013.

[14] Jonathan Courbon, Youcef Mezouar, Nicolas Guenard, and Philippe Martinet. Visual navigation of a quadrotor aerial vehicle. *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2009*, pages 5315–5320, 2009.

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

[16] Rohan Kapoor, Subramanian Ramasamy, Alessandro Gardi, and Roberto Sabatini. UAV Navigation using Signals of Opportunity in Urban Environments: A Review. *Energy Procedia*, 110(December 2016):377–383, 2017.

[17] Zaher M Kassas, Joshua J Morales, Kimia Shamaei, and Joe Khalife. Lte steers uav.

[18] Riley Bauer, Shannon Nollet, and Saad Biaz. A Novel Approach to Non-GPS Navigation Using Infrasound. 2014.

[19] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

[20] Spatial transformations. `http://www.it.hiof.no/~borres/j3d/math/threed/p-threed.html`. Accessed: 2018-05-28.

[21] G.E. Zaikov. *Analysis and Performance of Engineering Materials: Key Research and Development*. Apple Academic Press, 2015.

[22] What is camera calibration? `https://www.mathworks.com/help/vision/ug/camera-calibration.html`. Accessed: 2018-05-24.

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

[24] Visual servoing. https://es.wikibooks.org/wiki/Visual_servoing. Accessed: 2018-05-28.

[25] History. http://gazebosim.org/. Accessed: 2018-05-28.

[26] History. https://www.osrfoundation.org/. Accessed: 2018-05-28.

[27] Plugins 101. http://gazebosim.org/tutorials/?tut=plugins_hello_world. Accessed: 2018-05-28.

[28] History. http://www.ros.org/history/. Accessed: 2018-05-28.

[29] About ros. http://www.ros.org/about-ros/. Accessed: 2018-05-28.

[30] What is ros? http://wiki.ros.org/ROS/Introduction. Accessed: 2018-05-28.

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

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

[33] solvepnp(). https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html#ga549c2075fac14829ff4a58bc931c033d. Accessed: 2018-06-14.

[34] Basic concepts of the homography explained with code. https://docs.opencv.org/3.4.1/d9/dab/tutorial_homography.html. Accessed: 2018-05-28.

[35] Rodrigues(). https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html?highlight=calib#rodrigues. Accessed: 2018-06-14.

[36] Karl Johan Astrom and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*, chapter PID Control. Princeton University Press, Princeton, NJ, USA, 2008.

[37] Erle robotics. http://erlerobotics.com. Accessed: 2018-05-28.

[38] Pattern follower. http://docs.erlerobotics.com/simulation/vehicles/erle_copter/tutorial_5. Accessed: 2018-05-28.

[39] What is sdf. http://sdformat.org/. Accessed: 2018-05-28.

[40] Pconversion from latitude/longitude to cartesian coordinates. abe-research.illinois.edu/courses/tsm352/lectures/Lat_Long_Conversion.pptx. Accessed: 2018-11-15.