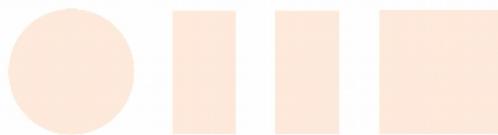


Universidad de Alcalá
Escuela Politécnica Superior

GRADO EN INGENIERÍA ELECTRÓNICA
DE COMUNICACIONES



Trabajo Fin de Grado

Desarrollo de aplicaciones basadas en visión con
entorno ROS para el *Drone Bebop 2*



ESCUELA POLITECNICA

Autor: Carlos Valero Lavid

SUPERIOR

Tutor/es: María Elena López Guillén

2017

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

Grado en Ingeniería Electrónica
de Comunicaciones



Trabajo Fin de Grado

“Desarrollo de aplicaciones basadas en visión con
entorno ROS para el dron Bebop 2”

Carlos Valero Lavid

2017

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

Grado en Ingeniería Electrónica
de Comunicaciones

Trabajo Fin de Grado

“Desarrollo de aplicaciones basadas en visión con
entorno ROS para el dron Bebop 2”

Autor: Carlos Valero Lavid

Tutor: María Elena López Guillén

Presidente: María Soledad Escudero Hernanz

Vocal 1º: Miguel Ángel García Garrido

Vocal 2º: María Elena López Guillén

Calificación:.....

Fecha:.....

*“Tanto si piensas que puedes, como si piensas
que no puedes, estás en lo cierto...”*

Henry Ford

Agradecimientos

En primer lugar, gracias a mi tutora y profesora Elena, que durante el tiempo que ha durado este proyecto me ha estado apoyando, guiándome por el mejor camino, sacando siempre un hueco para resolver problemas y dudas, y facilitándome lo necesario para poder llevar a cabo esta tarea. Agradecer también a mi familia, a mis amigos, y a todas aquellas personas que creyeron en mí, y que a lo largo de estos años han estado a mi lado, en los buenos y en los malos momentos, acompañándome hasta el final de esta aventura.

Contenido General

Resumen	1
Abstract	2
Resumen extendido	3
1 – Introducción	7
1.1 El auge de los drones	7
1.2 Objetivos del proyecto	10
1.3 Estructura de la memoria	12
2 – Herramientas Utilizadas	15
2.1 Bebop de Parrot	15
2.2 Hardware adicional	17
2.2.1 Antena <i>Wifi</i> Externa	18
2.3 Entorno de desarrollo ROS (Robotic Operating System)	20
2.3.1 ¿Qué es ROS?	20
2.3.2 Herramientas de ROS	22
2.3.3 Ejecución del ROS y comandos utilizados	25
2.4 Driver <i>bebop_autonomy</i>	31
2.4.1 Características Generales	32
2.4.2 <i>Topics</i> y mensajes utilizados	36
2.5 Librerías OpenCV	40
2.6 <i>Image Processing Toolbox</i>	41
2.7 <i>Robotics Systems Toolbox</i>	42
3 – Aplicaciones Desarrolladas	45
3.1 Introducción.....	45
3.2 Seguimiento autónomo de pasillos mediante puntos de fuga.....	47
3.2.1 Características extraídas de las imágenes recibidas.....	48
3.2.2 Programación de la aplicación.....	56

3.2.2.1 Nodo para la obtención del Punto de Fuga y del Punto de Luz.....	57
3.2.2.2 Nodo para el envío de velocidades al dron.....	59
3.2.2.3 Librerías y <i>Topics</i> utilizados.....	61
3.3 Control de movimiento mediante patrón visual.....	63
3.3.1 Programación de la aplicación.....	65
3.3.2 Obtención de información. Formación de la imagen recibida.....	67
3.3.3 Patrón visual y Segmentación.....	70
3.3.4 Velocidades en función de la posición del patrón.....	76
3.3.5 Interfaz gráfico con <i>AppDesigner</i>	79
3.3.6 Carga de <i>Bags</i> en Matlab.....	82
4 – Resultados, Conclusiones y Trabajos Futuros.....	87
4.1 Resultados.....	87
4.2 Conclusiones y Trabajos Futuros.....	88
5 – Presupuesto.....	93
6 – Pliego de Condiciones.....	99
7 – Manual de usuario.....	101
7.1 - Seguimiento autónomo de pasillo. Ejecución de la aplicación.....	101
7.2 - Seguimiento de patrón visual. Ejecución de la aplicación.....	102
8 – Planos.....	107
8.1 - Seguimiento autónomo de pasillo. Código de la aplicación.....	107
8.2 - Seguimiento de patrón visual. Código de la aplicación.....	120
9 – Bibliografía	129

Índice de Figuras

Figura 1: Dron Inspire 2 de la marca Dji.....	8
Figura 2: Dron Phantom 4 de la marca Dji.....	8
Figura 3: Dron de carreras.....	9
Figura 4: Dron Bebop 2 de la marca Parrot.....	9
Figura 5: Dron Bebop 2 de Parrot con su caja original.....	16
Figura 6: Batería del dron Bebop 2.....	16
Figura 7: Esquema de conexión con el dron.....	17
Figura 8: Antena externa USB Tp-Link TL-WN722N.....	18
Figura 9: Medida experimental de la potencia de señal recibida.....	19
Figura 10: Medida experimental de la relación señal a ruido recibida.....	19
Figura 11: Esquema de ROS como Meta-Sistema Operativo.....	21
Figura 12: Elementos de la red creada en ROS.....	22
Figura 13: Esquema del flujo de información en ROS.....	23
Figura 14: Esquema interno del Topic cmd_vel.....	24
Figura 15: Carpetas contenidas en el directorio de trabajo.....	32
Figura 16: Carpetas contenidas en el interior del driver bebop_autonomy.....	33
Figura 17: Archivo .yaml original, sin modificaciones.....	34
Figura 18: Parámetros añadidos al archivo .yaml.....	36
Figura 19: Esquema interno del Topic cmd_vel.....	37
Figura 20: Esquema de desplazamiento del Bebop 2 con velocidades positivas.....	38
Figura 21: Imagen captada por el Bebop 2 en posición horizontal.....	39
Figura 22: Imagen captada por el Bebop 2 en posición de avance.....	39
Figura 23: Esquema de uso de OpenCV en ROS.....	41
Figura 24: Esquema de la aplicación Seguimiento de pasillo mediante Puntos de Fuga.....	46
Figura 25: Esquema de la aplicación Seguimiento autónomo de Patrón Visual.....	46
Figura 26: Sensores incluidos en la parte inferior del Bebop 2.....	47
Figura 27: Cámara frontal del Bebop 2.....	48
Figura 28: Boceto ejemplo de un Punto de Fuga.....	49
Figura 29: Punto de Fuga desde distintas posiciones de un pasillo.....	49
Figura 30: Ubicación del Punto de Luz en una imagen.....	50
Figura 31: Esquema de posicionamiento del Punto de Luz en una imagen en función de la posición del dron en el pasillo.....	51

Figura 32: Recta obtenida al hacer la transformada de Hough a las luces de la imagen.	52
Figura 33: Esquema de orientación del Punto de Luz en función de la posición del dron en el pasillo.....	53
Figura 34: Esquema del ángulo formado por las luces de la imagen en función de la posición del dron en el pasillo.....	55
Figura 35: Esquema del flujo de información de la aplicación.....	56
Figura 36: Captura real de las imágenes captadas por el Bebop 2 en la ejecución del seguimiento autónomo de un pasillo.....	58
Figura 37: Esquema de posición valida para el Punto de Fuga dentro de la imagen.....	58
Figura 38: Imagen recibida (izquierda) e imagen filtrada (derecha).....	59
Figura 39: Captura del menú de inicio de la aplicación.....	59
Figura 40: Dirección de giro en función del Punto de Fuga.....	60
Figura 41: Desplazamiento lateral en función de la inclinación de las luces.....	61
Figura 42: Captura de código, librerías involucradas en la aplicación.....	62
Figura 43: Flujo de información del nodo bebop_autonomy.....	62
Figura 44: Flujo de información del nodo puntosfuga_bebop.....	63
Figura 45: Flujo de información del nodo control_puntosfuga_bebop.....	63
Figura 46: Patrón visual predeterminado.....	64
Figura 47: Patrón visual predeterminado, parte trasera.....	64
Figura 48: Captura del workspace de Matlab. Contenido del Topic image_raw.....	67
Figura 49: Esquema ejemplo de colocación de píxeles dentro de la componente G de una imagen RGB enviada desde Bebop 2.....	68
Figura 50: Componente G de una imagen RGB recibida desde el Bebop 2.....	69
Figura 51: Patrón visual diseñado para la aplicación.....	70
Figura 52: Imagen elegida para el entrenamiento del sistema.....	71
Figura 53: Imagen preprocesada para el entrenamiento del sistema.....	72
Figura 54: Fórmula de la circularidad.....	72
Figura 55: Clase 0 segmentada.....	73
Figura 56: Ejemplo del número de Euler como descriptor topológico.....	74
Figura 57: Clase 1 segmentada.....	75
Figura 58: Captura real de la aplicación en funcionamiento.....	75
Figura 59: Medida experimental de la variación del área del patrón visual con la distancia.....	77
Figura 60: Interfaz gráfico diseñado con <i>Appdesigner</i> en Matlab.....	80
Figura 61: Tabla de costes hardware y software.....	93

Figura 62: Tabla de costes de desarrollo.....	94
Figura 63: Tabla de costes totales.....	94
Figura 64: Interfaz gráfica diseñada con <i>Appdesigner</i> en Matlab.....	103
Figura 65: Figura para el entrenamiento del sistema.	120

Resumen

Este proyecto consiste en el desarrollo de dos aplicaciones para el control del dron Bebop 2 basadas en visión. Para ello se utilizan las imágenes recibidas de la cámara. La conexión con el dron se establece gracias al driver *bebop_autonomy*, el cual se ejecutará a través de ROS.

Para la ejecución de ambas aplicaciones se decidió implementarlas en entornos de programación diferentes, para así poder ser utilizadas en el futuro en docencia, abarcando dos herramientas distintas como son la programación en C con el uso de ROS nativo, y la programación en Matlab con el uso de la *Toolbox* de Robótica *Robotic System Toolbox*.

Las aplicaciones llevadas a cabo son por un lado el seguimiento autónomo de un pasillo por parte del dron, y por otro lado el reconocimiento y seguimiento de un patrón visual de manera autónoma.

Palabras clave

dron, visión artificial, ROS, seguimiento autónomo.

Abstract

This project consists in the development of two applications for the control of the dron Bebop 2 based on vision. For that the images that are received from the camera are used. The connection to the dron is established thanks to the *bebop_autonomy* driver, which will be executed through ROS.

For the implementation of both applications it was decided to implement them in different programming environments, to be used in the future in teaching, covering two different tools such as programming in C with the use of native ROS, and programming in Matlab with Use of the Robotic Toolbox Robotic System Toolbox.

The applications carried out are on the one hand the autonomous monitoring of a corridor by the dron, and on the other hand the recognition and monitoring of a visual pattern autonomously.

Keywords

dron, Artificial vision, ROS, autonomous tracking.

Resumen extendido

Este proyecto consiste en el desarrollo de dos aplicaciones para el control del dron Bebop 2 basadas en visión. Para ello se utilizan las imágenes recibidas de la cámara. La conexión con el dron se establece gracias al driver *bebop_autonomy*, el cual se ejecutará a través de ROS.

A finales de 2014 Parrot presentó el Bebop 1, un dron con un procesador interno de doble núcleo ARM Cortex A9 que trabaja sobre Linux. Más adelante publicó un *SDK (Software Developement Kit)* con las herramientas necesarias para poder interactuar con el dron, lo que dio lugar a la creación del driver *bebop_autonomy* que permite su ejecución en ROS.

A finales de 2015 Parrot lanzó al mercado el Bebop 2, un modelo mejorado y con más autonomía de vuelo que, sin embargo, se regía por el mismo sistema operativo haciendo también posible su control a través del driver *bebop_autonomy*.

Las aplicaciones desarrolladas se basan principalmente en la obtención de las imágenes captadas por el dron en una estación remota en tiempo real, para su tratamiento digital. Una vez se procesan las imágenes digitalmente y se consiguen extraer las características deseadas de la imagen se mandan comandos de velocidad en el eje x, eje y, eje z y *Yaw* para poder gobernar el dron.

La conexión entre la estación remota y el dron se realiza a través de *Wifi*. Se puede utilizar la propia antena *Wifi* del ordenador pero por seguridad se utilizará una antena externa de alta ganancia para aumentar el rango de acción y asegurar que la conexión no se interrumpa cuando el dron se aleje.

Para la ejecución de ambas aplicaciones se decidió implementarlas en entornos de programación diferentes, para así poder ser utilizadas en el futuro en docencia, abarcando dos herramientas distintas como son la programación en C con el uso de ROS nativo, y la programación en Matlab con el uso de la *Toolbox* de Robótica *Robotic System Toolbox*.

La primera aplicación desarrollada consiste en el seguimiento autónomo de un pasillo por parte del dron, a una altura fija y si colisionar en ningún momento con las paredes. Para ello se hace un tratamiento de las imágenes que recibimos del dron en C, donde se busca el Punto de Fuga y el Punto de Luz del pasillo. La aplicación se ejecutará en ROS nativo.

La segunda aplicación desarrollada consiste en el seguimiento autónomo por parte del dron de un patrón visual predeterminado, de tal forma que si el patrón se acerca al dron éste retroceda y si se aleja, el dron avance hacia él. Al desplazar el patrón hacia los lados el dron variará su posición en el Yaw rotando sobre sí mismo. En este caso, para el tratamiento de la imagen se utilizará Matlab, cuya *Toolbox* de tratamiento de imágenes proporciona las herramientas necesarias para la segmentación y cuya *Toolbox* de robótica permite la utilización de ROS, para levantar un nodo y recibir las imágenes directamente del dron.

Introducción

1 – Introducción

1.1 El auge de los drones

Los UAV (*Unmanned Aerial Vehicle*) sufren hoy en día una revolución con la creación de aeronaves cada vez más pequeñas y baratas. Tanto es así que en los últimos años han aparecido nuevas tecnologías que permiten el desarrollo y la comercialización de aeronaves de bajo coste para uso personal y profesional.

A pesar de que la palabra “dron” se define en la RAE como “aeronave no tripulada”, la mayoría de las personas asocian hoy en día la palabra dron al tipo de aeronave pequeña y ligera, que dispone de 4, 6 u 8 hélices que hacen posible un vuelo estable, gracias a una alimentación eléctrica que proviene de baterías, siendo los más comunes los cuadricópteros, o drones de 4 hélices.

Hoy en día, los drones existentes aún se encuentran muy limitados en ciertos aspectos como son las baterías o el peso que son capaces de levantar, debido a la potencia que necesitan los motores para vencer la gravedad y a la poca autonomía de las baterías eléctricas existentes.

Aún así existe toda una gama de aeronaves con diversos usos, desde los más profesionales como el Phantom 4 o el Inspire de Dji^[1], que cuentan con cámaras profesionales para la grabación y transmisión de video en directo, drones de carreras en los que prima la potencia y la velocidad frente a la autonomía de vuelo, y cuyos componentes son cuidadosamente seleccionados por cada piloto, o drones de uso lúdico como el Bebop 2 de Parrot^[2].



Figura 1: Dron Inspire 2 de la marca Dji.



Figura 2: Dron Phantom 4 de la marca Dji.



Figura 3: Dron de carreras.



Figura 4: Dron Bebop 2 de la marca Parrot.

Al ser un sector de muy rápido crecimiento, en los últimos años la legislación existente que hace referencia a los lugares permitidos para volar un dron, y a la cualificación necesaria por parte del piloto es bastante restrictiva, teniendo la formación un coste elevado para el uso de drones de manera no profesional y estando los lugares de vuelo muy restringidos. El vuelo está únicamente permitido bajo permisos temporales expedidos con días de antelación.

El auge de los drones también trae como consecuencia un creciente desarrollo de cámaras y medios de grabación y transmisión de imágenes, ya que los drones ponen a nuestro alcance la grabación de imágenes a vista de pájaro. Debido al bajo coste de las cámaras, hoy en día prácticamente todos los drones incorporan una, siendo por tanto muy importante la visión artificial y el tratamiento de imágenes.

1.2 Objetivos del proyecto

Los objetivos de este proyecto no se centran únicamente en el desarrollo de dos aplicaciones para el dron Bebop 2 basadas en visión, sino que también se ha llevado a cabo un estudio a fondo de las herramientas necesarias para implementar esta tarea.

Debido al auge de este sector en crecimiento decidimos desarrollar aplicaciones que puedan ser utilizadas en docencia utilizando un dron comercial.

Para que este sea un proyecto completo y pueda abarcar diferentes herramientas utilizadas en docencia, cada una de las aplicaciones se desarrollará en un entorno de programación distinto, utilizando para una de ellas ROS ^[3] nativo con nodos programados en C y para otra Matlab ^[4], con ayuda de las *Toolbox* de Robótica *Robotics System Toolbox* y de visión artificial *Image Processing Toolbox*, además de la herramienta de diseño de interfaces gráficos “*Appdesigner*” que provee Matlab a partir de la versión de 2016.

Para este proyecto, ha sido necesario por parte del alumno aprender cómo utilizar ROS, así como todos los conceptos y herramientas necesarios para su correcta ejecución.

Además se documenta lo aprendido sobre ROS y sobre el driver *bebop_autonomy* ^[5] que permite el intercambio de información con el Bebop 2, así como las *Toolbox* utilizadas en Matlab.

El proyecto ha constado de las siguientes fases, clasificadas por orden de ejecución:

1. Descarga e instalación del entorno ROS en sistema operativo Linux.
2. Estudio de las características y funcionamiento del entorno ROS a través de los tutoriales que proporciona la página oficial, siendo necesario familiarizarse por completo con el entorno.
3. Búsqueda, descarga y análisis del driver de control del Bebop 2 llamado *bebop_autonomy*. Se ha necesitado estudiar todas las posibilidades de control que ofrece el driver y si era compatible con el Bebop 2, ya que hasta el momento sólo se había utilizado en el modelo anterior, el Bebop 1.
4. Una vez conseguida la conexión con el dron con el uso del driver se comprobó que funcionaba correctamente haciendo una prueba de teleoperación desde la estación remota.
5. Reconocimiento de imágenes. Se ha decidido el diseño de un patrón visual para que el dron, mediante tratamiento digital y procesado de imágenes lo identifique y siga sus movimientos.
6. Estudio de las *Toolbox* de Matlab para establecer la conexión con ROS, así como el estudio de sus diferentes posibilidades de trabajo, como puede ser el hecho de trabajar con imágenes recibidas directamente del dron, o con imágenes que provienen de un archivo *.bag* previamente grabado.
7. Una vez se ha conseguido el control del dron y la detección óptima en las imágenes con la ayuda de las toolbox *Image Processing Toolbox* y *Robotics System Toolbox*, se ha procedido a la fusión de ambas para crear las dos aplicaciones finales. En esta parte del proceso se han realizado numerosas pruebas en distintos entornos, con las pertinentes correcciones de errores.
8. Por último, se ha redactado este documento que incluye toda la información del proyecto, del proceso llevado a cabo y de las herramientas utilizadas.

1.3 Estructura de la memoria

A continuación se muestra una breve estructura de la memoria de este trabajo.

En el primer apartado (en el que nos encontramos) se expresa brevemente la motivación de este proyecto debido al auge de los drones, los objetivos del proyecto y la estructura de la memoria.

En el segundo apartado se lista y explica cada una de las herramientas utilizadas en este proyecto. Éstas son el Bebop 2 de Parrot, la estación remota utilizada, el entorno de desarrollo ROS, el driver *bebop_autonomy*, las librerías *OpenCV*^[6], y las *Toolbox* de Matlab *Image Processing Toolbox* y *Robotics System Toolbox*.

A continuación, el tercer apartado explica en detalle cómo se ha llevado a cabo el desarrollo de las dos aplicaciones, así como la problemática que ha ido surgiendo a lo largo del proceso y cómo se ha solventado.

En el apartado cuatro se exponen todos los resultados obtenidos una vez finalizadas las aplicaciones, habiendo realizado pruebas en distintos entornos y con distintas condiciones de espacio y luminosidad, así como las conclusiones obtenidas a partir de todas ellas y los posibles trabajos futuros o campos de investigación.

El apartado cinco expone en detalle los costes totales del proyecto contando gastos hardware, software y de personal.

En el apartado número seis, pliego de condiciones, se especifica todo lo necesario para poder reproducir este proyecto, así como todas las versiones de los distintos elementos software y *Firmware* utilizados.

Por último, el séptimo y el octavo capítulo son un resumen de los pasos a seguir para la correcta ejecución de las aplicaciones y la bibliografía con todas las referencias consultadas para la obtención de información.

Herramientas Utilizadas

2 – Herramientas Utilizadas

2.1 Bebop de Parrot

A finales de 2014 Parrot presentó el Bebop 1, un dron con un procesador interno de doble núcleo ARM Cortex A9 que trabaja sobre Linux. Más adelante publicó un SDK (*Software Development Kit*), que posibilitó la creación del driver llamado *bebop_autonomy* el cual nos permite interactuar con el dron a través de Linux, utilizando ROS.

A finales de 2015 Parrot lanzó al mercado el Bebop 2, un modelo mejorado y con más autonomía de vuelo que, sin embargo, se rige por el mismo sistema operativo haciendo también posible su control a través del driver *bebop_autonomy*.

El Bebop 2 presenta una autonomía de hasta 25 minutos y permite la realización de vídeos en 1080p x 1920p y fotografías a 14 megapíxeles. Cuenta con un sistema de estabilización de imagen digital y a través de su aplicación *Freeflight*^[7] se pueden visualizar las imágenes captadas en tiempo real. Esta aplicación permite teleoperar el dron y ejecutar fotos y videos que se guardan en la tarjeta SD interna del dron.



Figura 5: Dron Bebop 2 de Parrot con su caja original.

Su velocidad máxima es de 60km/h y se estabiliza gracias a la acción de un acelerómetro, un giróscopo y un magnetómetro.

Las baterías que utiliza este dron son *plug and play* y su tiempo de carga es de hora y media.



Figura 6: Batería del dron Bebop 2

El Bebop 2 tiene un gran campo de imagen que posibilita la toma de imágenes casi de forma cenital, gracias a la posición oblicua de su cámara. Además, su lente de 180° permite que la imagen recibida no se incline o distorsione cuando el dron avanza, en otras palabras, el dron siempre mira al frente aunque nosotros lo inclinemos.

Este dron se puede pilotar de tres maneras distintas, todas ellas a través de *Wifi*:

1. Utilizando la aplicación *Freeflight* con un alcance máximo de 300m. Esta aplicación está disponible para dispositivos *Android* e *IOS*.
2. Utilizando el mando *Skycontroler* que aumenta su rango de acción a 2km.
3. A través del SDK publicado por Parrot donde el alcance dependerá del tipo de antena que utilicemos en nuestra estación remota.

Gracias a sus dos antenas internas el Bebop 2 emite *Wifi* en 2,4Ghz y en 5Ghz, y cuenta con una baliza de luz de color rojo en la parte trasera para localizarlo en la distancia.

2.2 Hardware adicional

Para que sea posible la conexión con el dron vía *Wifi* es necesario una estación de control remota, formada en este caso por un ordenador y una antena externa para mejorar la calidad de la señal enviada y recibida.



Figura 7: Esquema de conexión con el dron

2.2.1 Antena Wifi Externa

Al establecer la conexión es necesario que la antena utilizada tenga la ganancia suficiente para no perder la conexión cuando el dron se aleje. En caso de pérdida de la conexión con el dron prevalecerá la última orden de velocidad que le haya llegado en el momento anterior a la desconexión, pudiendo ser un movimiento de avance en cualquier dirección y siendo, por tanto, esto peligroso para el operador y para el propio dron.

Para las dos aplicaciones de este proyecto no se requiere un gran alcance de la señal y se podría utilizar la antena *Wifi* interna del portátil que se esté utilizando, pero para evitar riesgos utilizaremos la antena externa *Tp-Link TL-WN722N*^[8]con una ganancia de 6db y un *bit-rate* de 150Mbps en un USB 3.0.



Figura 8: Antena externa USB Tp-Link TL-WN722N

Para medir experimentalmente la calidad de la señal recibida con las distintas antenas, se puede usar el comando de Ubuntu *iwconfig*, de tal manera que se actualice cada segundo. Para ello se escribe en un terminal:

```
$ watch -n1 iwconfig
```

De todos los datos que aparecen hay que observar dos: *Link Quality* (relación señal / ruido) y *Signal Level* (potencia recibida).

El rango de acción en condiciones de visión directa, obtenido experimentalmente es el siguiente:

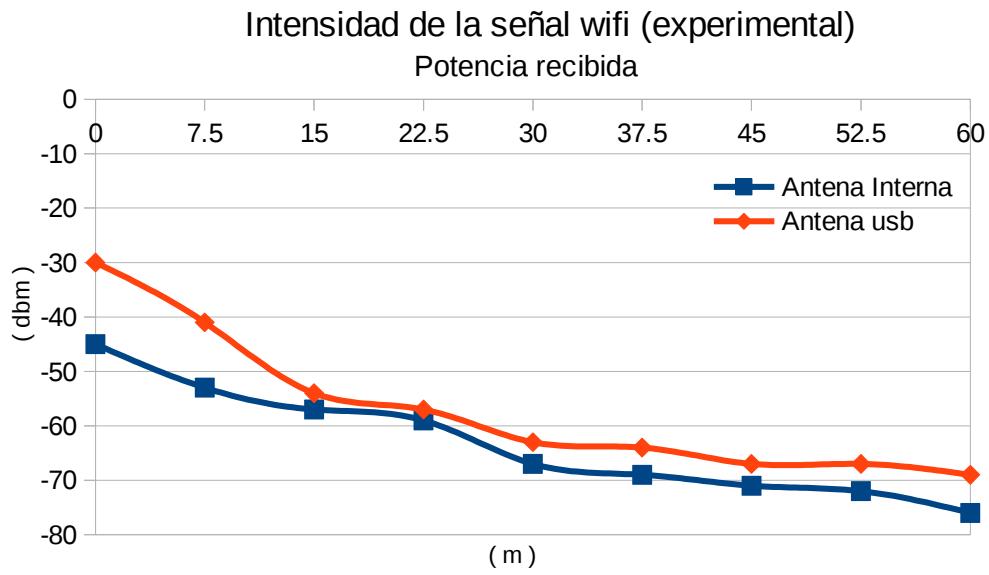


Figura 9: Medida experimental de la potencia de señal recibida

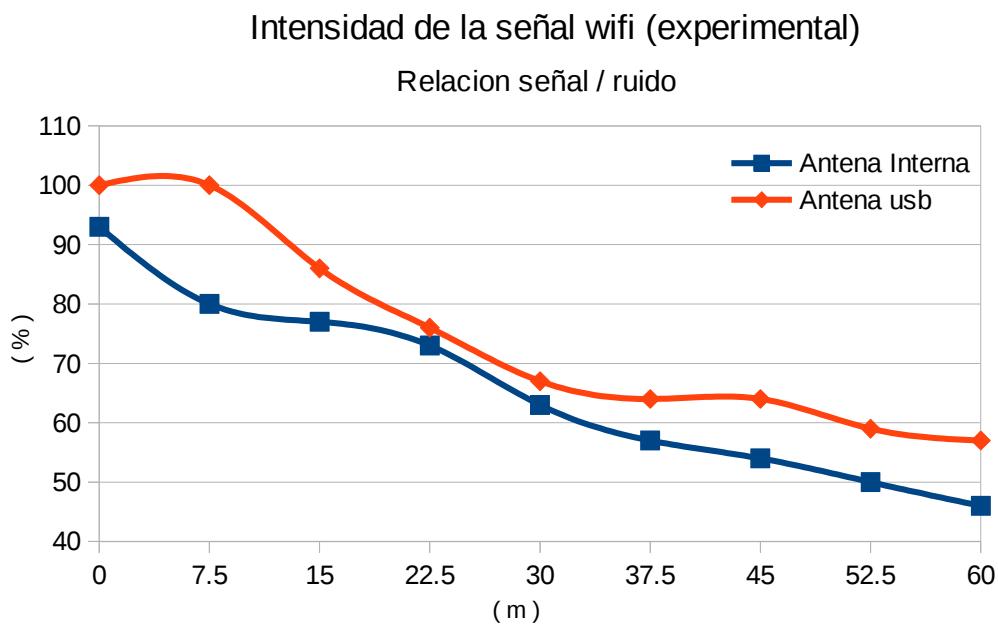


Figura 10: Medida experimental de la relación señal a ruido recibida

Se puede comprobar que a una distancia inferior a 15 metros la relación Señal a Ruido si se utiliza la antena externa USB está por encima del 85 % con una potencia recibida mayor de -55 dbm, calidad de la señal suficiente para la ejecución de las aplicaciones.

2.3 Entorno de desarrollo ROS (Robot Operating System)

2.3.1 ¿Qué es ROS?

Este sistema operativo para robótica es una herramienta flexible a la hora de escribir el software para un sistema robotizado. Se basa en un conjunto de herramientas, librerías y convenciones que simplifica la tarea de crear un software complejo y robusto para robótica que se pueda implementar en una gran variedad de plataformas.

ROS es un sistema de código abierto, construido desde cero para fomentar el desarrollo de software de robótica colaborativa. La idea era que cada grupo de expertos que trabajase con ROS pudiese aportar sus partes de software y que pudiesen ser utilizadas por cualquier otra persona en cualquier parte del mundo.

Este sistema fue diseñado para ser distribuido lo más modularmente posible, de modo que los usuarios pudiesen elegir qué partes utilizar en función de las necesidades de su aplicación.

ROS trabaja como si se tratase de un sistema operativo, ya que proporciona servicios estándares tales como abstracción del hardware, comunicación a través de mensajes entre procesos, mantenimiento de paquetes, control de dispositivos de bajo nivel e implementación de funcionalidad que son propios de éstos. Se habla de que trabaja como un sistema operativo pero en realidad no lo es, ya que se instala sobre otro, generalmente Linux, y por ello también recibe el nombre de Meta-Sistema Operativo.

ROS Overview: Meta-Sistema Operativo

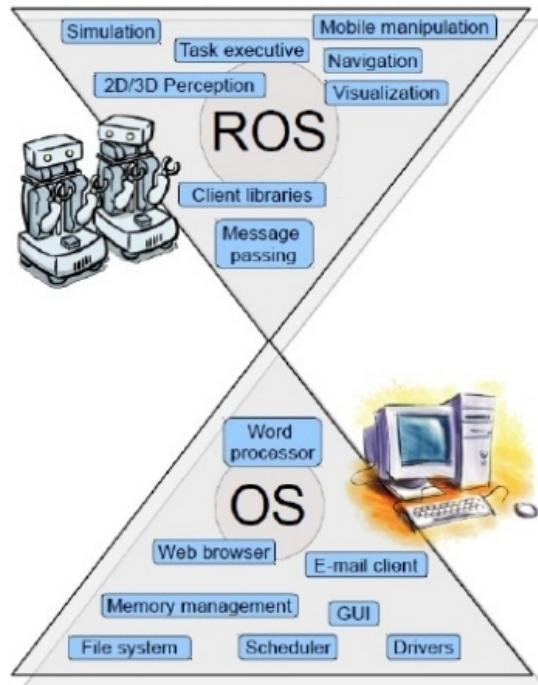


Figura 11: Esquema de ROS como Meta-Sistema Operativo

Además, ROS fomenta una gran comunidad de paquetes aportados por los usuarios, que cubren todo tipo de necesidades, desde implementación de pruebas de concepto hasta algoritmos con capacidades y calidades industriales.

La comunidad de usuarios de ROS se basa en una infraestructura común para proporcionar un punto de integración que ofrece acceso a controladores de hardware, capacidades de robot genéricas, herramientas de desarrollo, bibliotecas externas útiles y más.

Históricamente, la mayoría de los usuarios estaban en laboratorios de investigación, pero se está viendo cada vez más la adopción en el sector comercial, particularmente en la robótica industrial y de servicios.

2.3.2 Herramientas de ROS

ROS tiene mucho potencial debido a todas las herramientas que lo componen, sin embargo para este proyecto sólo se ha necesitado entender y dominar una serie de conceptos, que se explican a continuación con la ayuda de este gráfico:

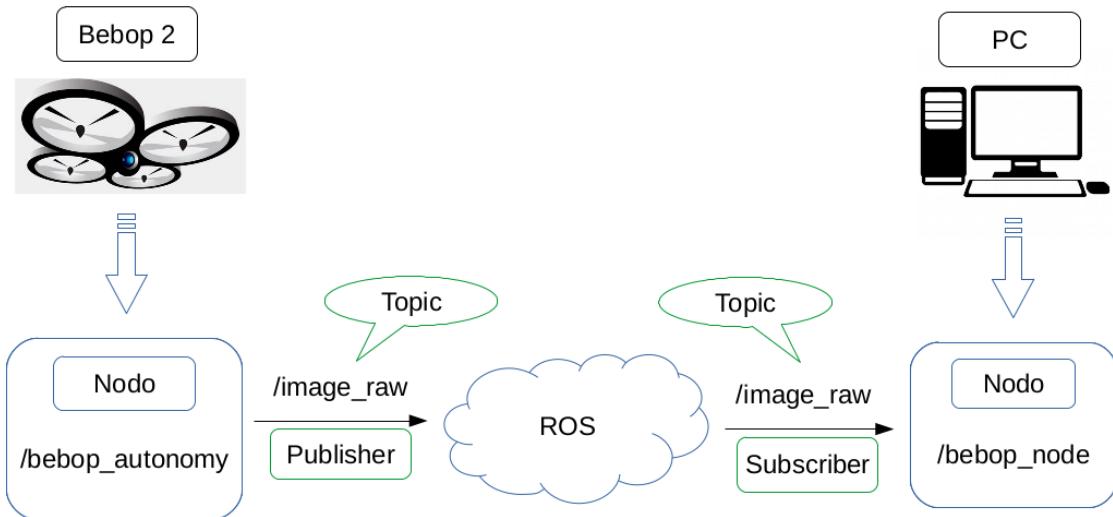


Figura 12: Elementos de la red creada en ROS

Los conceptos básicos que forman la red de procesos de ROS son los presentados a continuación:

- **Nodos (Node):** Los nodos son los encargados de realizar la computación propiamente dicha. ROS está estructurado modularmente de forma que un robot puede comprender muchos nodos que controlen distintas partes del robot, por ejemplo, un nodo específico para el movimiento del robot, otro para las distintas articulaciones del mismo y así sucesivamente. Un nodo de ROS está escrito usando una biblioteca específica de cliente, tales como *roscpp* (C++) o *Rospy* (Python).
- **Maestro (Master):** Es el nodo principal. El *Master* proporciona el registro de nombres y de consulta para la computación gráfica. Sin éste, los nodos son incapaces de trabajar ya que no encontrarían los mensajes necesarios o los

servicios que una aplicación requiera. El *Master*, en general, se ejecuta mediante el terminal con el comando *roscore*, aunque hay excepciones en las que se ejecuta sólo mediante la carga de archivos *.launch*.

- **Servidor de Parámetros (*Parameter Server*):** Permite el almacenamiento de datos en una localización central.
- **Mensajes (*Messages*):** Comunicación entre nodos. Se trata de una estructura de datos, la cual puede tener tipos numéricos y estructuras anidadas arbitrariamente.
- **Temas (*Topics*):** Los nodos comparten información mandando y recibiendo mensajes. Estos mensajes son publicados con un nombre especial llamado *Topics*. Por lo tanto, los nodos pueden publicar o suscribirse a un *Topic* específico para obtener los mensajes deseados. Incluso, un mismo nodo puede estar suscrito a distintos *Topics*.
- **Servicios (*Services*):** Los servicios proporcionan una comunicación mediante un método de pregunta/respuesta. Es decir, está formado por dos partes: un mensaje que es el que pregunta y otro mensaje que es el que responde. Cuando un nodo quiere llamar a un servicio, tiene que llamarle y esperar la respuesta. A diferencia de la comunicación de los *Topics*, un nodo sólo puede llamar a un servicio bajo un nombre particular.
- **Bolsas (*Bags*):** Se trata de un formato especial que permite al usuario la opción de guardar o reproducir mensajes de datos de ROS. Por ello, los *bags* tienen una gran importancia a la hora de hacer pruebas para desarrollo y algoritmos.

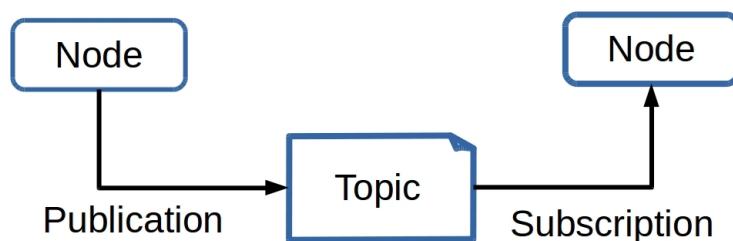


Figura 13: Esquema del flujo de información en ROS

ROS se basa en una arquitectura *Master-Slave*, en la que es necesario ejecutar ROS en un dispositivo (*Master*), y luego en todos los demás dispositivos en los que se ejecute un nodo se conectarán al *Master* (*Slaves*).

Los nodos se comunican entre sí transmitiendo parámetros en *streaming* mediante *Topics*, servicios *RPC* y un servidor de parámetros. Estos nodos están diseñados para operar a bajo nivel, por tanto los sistemas de control de un robot normalmente compondrán muchos nodos.

Por otro lado, a través de cada *Topic* se envía un tipo de información con un tipo de mensaje concreto. Así pues, se tendrá por ejemplo un *Topic* llamado *cmd_vel* en el que se publicarán las velocidades que se quieran dar al dron. Los mensajes son del tipo *geometry_msgs/Twist.msg* y los mensajes *Twist.msg* a su vez están formados por dos vectores de tres componentes cada uno que indican la velocidad lineal y angular en los ejes X, Y y Z.

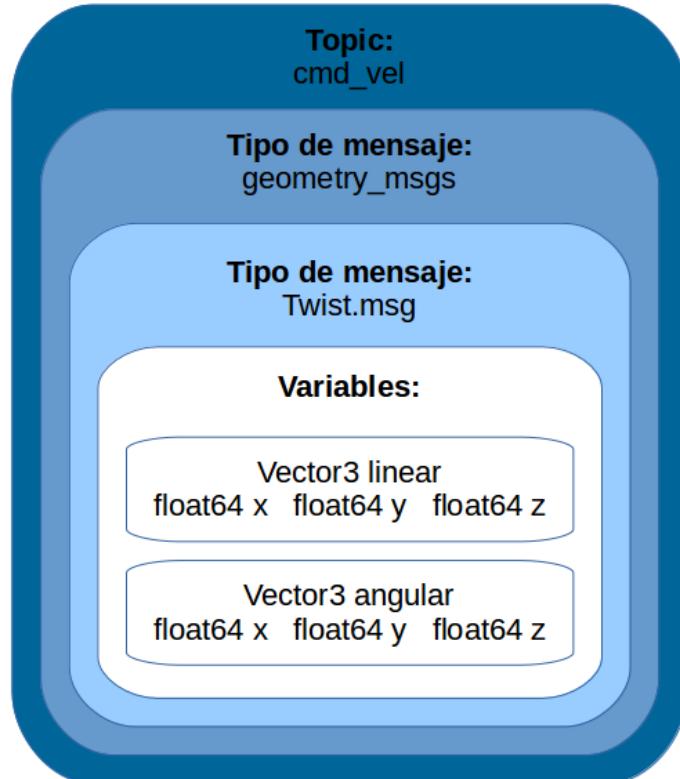


Figura 14: Esquema interno del Topic *cmd_vel*.

La información de cómo están formados los mensajes que ROS utiliza se encuentra fácilmente en su pagina web, en este caso en la documentación del driver *bebop_autonomy* [5].

2.3.3 Ejecución de ROS y comandos utilizados

ROS proporciona herramientas para facilitar la implementación de sistemas robotizados. Estas herramientas permiten a los usuarios simular, lanzar varios nodos a la vez y visualizar el flujo entre los nodos. Algunas de las herramientas que posee se presentan a continuación:

- **roscore:** Herramienta que ejecuta el núcleo de ROS. Este comando es muy importante y permite inicializar la comunicación entre nodos, servicios, parámetros, etc, mediante el nodo “maestro” o *master*.

```
$ roscore
```

- **roscreate-pkg:** Mediante esta herramienta el usuario puede comenzar la creación de los paquetes que necesite, pudiendo además especificar las dependencias de otros paquetes.

```
$ roscreate-pkg [nombre del paquete] [dependencias]
```

- **rospack:** Se puede obtener información de un paquete mediante esta herramienta.

```
$ rospack [comando] [paquete]
```

- **rosstack:** Información de la “pila” que se deseé.

```
$ rosstack [comando] [stack]
```

- **roscd:** Permite acceder a un paquete.

```
$ roscd [paquete]
```

- **rosls:** Herramienta que permite al usuario ver el contenido de un paquete.

```
$ rosls [paquete]
```

- **rosnode:** Muestra una lista los nodos en ejecución, información de un nodo o permite eliminar nodos de la ejecución.

```
$ rosnode [comando] [nodo]
```

- **rosdep:** Permite descargar dependencias de paquetes.

```
$ rosdep install [paquete]
```

- **rostopic:** Herramienta útil para controlar *Topics*. Se puede obtener una lista de todos los *Topics* en ejecución, así como publicar mensajes en ellos, obtener información o leer los datos de dichos *Topics*.

```
$ rostopic [comando] [Topic] [argumentos]
```

- **rosservice:** Al igual que los *Topics*, se puede obtener una lista de los distintos servicios que se están ejecutando, información y utilizar dichos servicios para un fin.

```
$ rosservice [comando] [servicio] [argumentos]
```

- **rosmsg:** Permite obtener la información de un tipo de mensaje.

```
$ rosmsg [comando] [tipo de mensaje]
```

- **rosrun:** Permite ejecutar un archivo ejecutable de un paquete específico.

```
$ rosrun [paquete] [ejecutable]
```

- **roslaunch:** ROS permite ejecutar varios nodos a la vez. Mediante esta herramienta se puede ejecutar archivos con formato *.launch*, en estos archivos se puede configurar varios nodos para poder ejecutarlos a la vez facilitando al usuario el uso del sistema global de ROS.

```
$ roslaunch [paquete] [archivo .launch]
```

Una vez conocidas las herramientas se procede a explicar los distintos tipos de posibilidades que existen a la hora de darles uso, y a exponer algunos ejemplos de ejecución de las más utilizadas.

En ROS existen una serie de comandos muy utilizados a la hora de obtener información de lo que se está ejecutando o de los *subscribers* y los *publishers* que están activos, así como los datos que se están intercambiando. Subscribirse a un *Topic* es la manera de recibir la información que circula por él, y publicar en un *Topic* es la manera que existe de enviar información a través de él.

Comando	Descripción
Rostopic bw	Muestra el ancho de banda consumido por el <i>Topic</i>
Rostopic delay	Retardo de visualización del <i>Topic</i>
Rostopic echo	Muestra los mensajes del <i>Topic</i> por pantalla
Rostopic find	Busca <i>Topics</i> por tipo
Rostopic hz	Muestra la velocidad de publicación de un <i>Topic</i>
Rostopic info	Muestra información de un <i>Topic</i> activo
Rostopic list	Lista de <i>Topics</i> activos
Rostopic pub	Publica un mensaje en un <i>Topic</i>
Rostopic type	Muestra un <i>Topic</i> o un tipo de campo

Para ver información acerca de los *Topics* se utiliza el comando *rostopic* seguido de una de las opciones descritas en la tabla anterior.

Por ejemplo, si una vez ejecutado el driver *bebop_autonomy* se ejecuta el comando *rostopic list*, se mostrará la lista de *Topics* activos en ese momento:

```
$ rostopic list
```

```
/bebop/autoflight/navigate_home  
/bebop/autoflight/pause  
/bebop/autoflight/start  
/bebop/autoflight/stop  
/bebop/bebop_driver/parameter_descriptions  
/bebop/bebop_driver/parameter_updates  
/bebop/camera_control  
/bebop/camera_info  
/bebop/cmd_vel  
/bebop/fix  
/bebop/flattrim  
/bebop/flip  
/bebop/image_raw  
/bebop/joint_states  
/bebop/land  
/bebop/odom  
(...)
```

Con el comando *rostopic info* seguido del nombre de un *Topic* nos mostrará la información del tipo de mensajes que maneja ese *Topic* y de los *publishers* y los *subscribers* que tiene.

```
$ rostopic info /bebop/cmd_vel
```

```
Type: geometry_msgs/Twist  
publishers: None  
subscribers:  
* /bebop/bebop_driver (http://valero-PC:37697/)
```

Para publicar un mensaje en un *Topic* se utiliza el comando *rostopic pub* seguido del *Topic* en el que se quiere publicar y del tipo de mensaje que utiliza dicho *Topic*, esto es útil si se tiene que aterrizar el dron publicando a través de línea de comandos, porque nuestra aplicación se ha colapsado y no responde. Si añadimos --once sólo publicará una vez el mensaje.

```
$ rostopic pub --once /bebop/land std_msgs/Empty
```

Para poder ver la información que se publica en un *Topic* se utiliza *rostopic echo*, útil para saber si se publica correctamente en un *Topic* los datos deseados, o para saber información del dron como la batería restante, como se ve a continuación.

```
$ rostopic echo /bebop/states/common/CommonState/BatteryStateChanged
```

```
header:  
seq: 6  
stamp:  
secs: 1493810131  
nsecs: 976993899  
frame_id: base_link  
percent: 62 (nivel de batería)
```

Para saber información de los nodos activos se utilizan los comandos *rosnode*.

Comando	Descripción
Rosnode cleanup	Borra toda la info de nodos que no están activos
Rosnode info	Muestra la info de un nodo
Rosnode kill	Detiene un nodo
Rosnode list	Lista los nodos activos
Rosnode machine	Muestra los <i>host</i> donde hay nodos activos
Rosnode ping	Test de conectividad con un nodo

Por ejemplo *rosnode list* muestra la lista de nodos activos.

```
$ rosnode list
```

```
bebop/bebop_driver  
/bebop/robot_state_publisher  
/rosout
```

Rosnode info seguido del nombre de un nodo da información de los *publishers*, los *subscribers* y los servicios que provee ese nodo.

```
$ rosnode info /bebop/robot_state_publisher
```

```
Node [/bebop/robot_state_publisher]  
Publications:  
* /tf [tf2_msgs/TFMessage]  
* /tf_static [tf2_msgs/TFMessage]  
* /rosout [rosgraph_msgs/Log]  
  
Subscriptions:  
* /bebop/joint_states [sensor_msgs/JointState]  
Services:  
* /bebop/robot_state_publisher/set_logger_level  
* /bebop/robot_state_publisher/get_loggers
```

```
contacting node http://valero-PC:33263/ ...
Pid: 10974
Connections:
  * Topic: /rosout
    * to: /rosout
    * direction: outbound
    * transport: TCPROS
  * Topic: /bebop/joint_states
    * to: /bebop/bebop_driver (http://valero-PC:37697/)
    * direction: inbound
    * transport: TCPROS
```

Cuando se está ejecutando un código, a la hora de depurarlo también son importantes los valores de las variables. Para poder saber los valores o poder modificar las variables desde línea de comandos se utilizan los comandos *rosparam*.

Comando	Descripción
Rosparam set	Dar valor a un parámetro
Rosparam get	Conocer el valor de un parámetro
Rosparam load	Cargar el valor de un parámetro de un archivo
Rosparam dump	Guardar el valor del parámetro en un archivo
Rosparam delete	Borrar el valor de un parámetro
Rosparam list	Lista de todos los parámetros

Para saber todos los comandos de ROS y sus posibilidades de uso toda la información se encuentra en la *Wiki online* [3].

2.4 Driver *bebop_autonomy*

A continuación, se va a explicar el driver *bebop_autonomy*, la utilidad que tiene y cómo se ha empleado para llevar a cabo este proyecto.

2.4.1 Características Generales

El *bebop_autonomy* es un driver de ROS para los drones Bebop 1 y Bebop 2, basado en el SDK ARDroneSDK3 oficial de Parrot. Este driver ha sido desarrollado por el *autonomy lab* en la universidad Simon Fraser por Mani Monajjemi además de otros colaboradores¹.

A continuación, se va a explicar la manera de ejecutar el driver *bebop_autonomy* para poder interactuar con el dron. Una vez ejecutado se crearán una serie de *Topics* que permiten la creación de *subscribers* para recibir la información enviada por el dron y *publishers* para enviar comandos al dron.

Para ello se accede al directorio de trabajo que se haya creado a la hora de instalar ROS, y dentro del cual deberá estar la carpeta que contenga el driver. En este caso el directorio de trabajo tiene como nombre *TFG* y la carpeta contenedora del driver se llama *bebop_autonomy*. El archivo de ejecución del driver es un archivo *.launch*, por tanto hemos de acceder a la carpeta *launch* utilizando el comando *cd* seguido de la ruta completa².

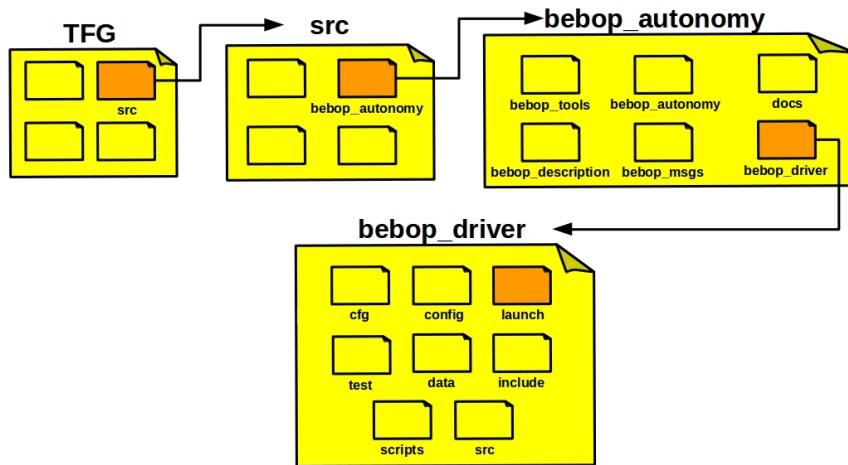


Figura 15: Carpetas contenidas en el directorio de trabajo.

1 - Lista de colaboradores:

<http://bebop-autonomy.readthedocs.io/en/latest/contribute.html#sec-contribs>

2 - Al utilizar el comando *rosrun* se ejecutan por defecto los archivos contenidos en la carpeta *launch* sin necesidad de ir a su ruta, al igual que se ejecuta el ROS *master* si no se estaba ejecutando en el momento de lanzar el comando.

Para ejecutar el archivo *.launch* es necesario utilizar el comando *roslaunch* seguido del paquete donde se encuentra (la carpeta que lo contiene) y del nombre del archivo (apartado 2.3.3 Ejecución de ROS y comandos utilizados).

```
$ rosrun [paquete] [archivo .launch]
```

```
$ rosrun bebop_driver bebop_node.launch
```

Una vez lanzado el driver se generan una serie de *Topics* que permiten conocer y manejar todas las variables del dron, desde la orientación de la cámara, la intensidad de la señal o el nivel de batería hasta el envío de velocidades para teleoperarlo o la recepción en nuestro ordenador de las imágenes captadas por la cámara.

El driver explica con todo detalle en la *wiki online* del *bebop_autonomy* [5]. Para este proyecto se ha estudiado y extraído la información más relevante que permita llevar a cabo el objetivo de las aplicaciones.

Una vez se instala, el driver consta de una serie de carpetas entre las que se encuentran la carpeta *launch* y la carpeta *config*.

Al ejecutar el archivo *.launch*, lo primero que ejecuta el driver internamente es el fichero de configuración *.yaml* que ejecuta un *setup* de todos los parámetros iniciales del dron y que se encuentra dentro de la carpeta *config*. Los parámetros que aparecen en ese archivo son parámetros que no se pueden cambiar una vez ejecutado el driver. Para modificarlos habría que detenerlo y volverlo a lanzar con los nuevos valores. Esos parámetros hacen referencia a la altitud máxima que adquiere el dron, altitud mínima, máxima distancia que se aleja de la emisora, grados máximos de inclinación a la hora de avanzar, etc.

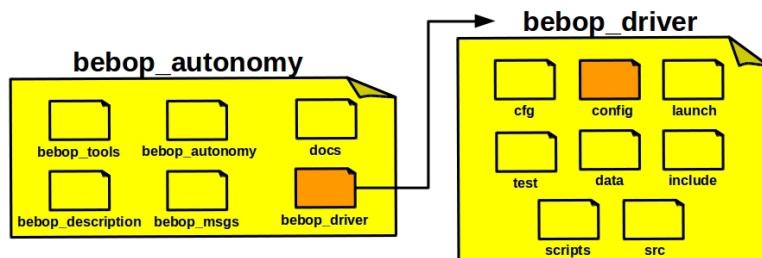


Figura 16: Carpetas contenidas en el interior del driver *bebop_autonomy*

Los valores no tienen por qué aparecer en ese archivo, adquiriendo las variables entonces sus valores predeterminados. El archivo *.yaml* original es el siguiente:

```
states:  
    enable_commonstate_batterystatechanged: true  
    enable_commonstate_wifisignalchanged: true  
    enable_overheatstate_overheatchanged: true  
    enable_controllerstate_ispilotingchanged: true  
    enable_mavlinkstate_mavlinkfileplayingstatechanged: true  
    enable_mavlinkstate_mavlinkplayerrorstatechanged: true  
    enable_flightplanstate_availabilitystatechanged: true  
    enable_flightplanstate_componentstatelistchanged: true  
    enable_pilotingstate_altitudechanged: true  
    enable_pilotingstate_flattrimchanged: true  
    enable_pilotingstate_flyingstatechanged: true  
    enable_pilotingstate_navigatehomestatechanged: true  
    enable_pilotingstate_positionchanged: true  
    enable_pilotingstate_speedchanged: true  
    enable_pilotingstate_attitudechanged: true  
    enable_pilotingstate_positionchanged: true  
    enable_altitudechanged: true  
    enable_autotakeoffmodechanged: true  
    enable_mediastreamingstate_videoenablechanged: true  
    enable_camerastate_orientation: true  
    enable_gpsstate_numberofsatellitechanged: true  
    enable_numberofsatellitechanged: true  
  
reset_settings: true  
publish_odom_tf: true  
odom_frame_id: "odom"  
cmd_vel_timeout: 0.2
```

Figura 17: Archivo *.yaml* original, sin modificaciones.

Para este proyecto se ha definido un archivo *.yaml*, con los valores que hemos considerado más óptimos a la hora de asegurar la integridad del dron y la seguridad de las personas que lo rodean.

Los valores de los parámetros añadidos al archivo *.yaml* son los siguientes:

- Altitud máxima a la que se quiere mantener el dron estático, hasta que se le indique lo contrario. Unidades en metros.

PilotingSettingsMaxAltitudeCurrent: 1.5

- Máxima inclinación que tomará el dron a la hora de hacer desplazamientos en el eje X o en el eje Y. Unidades en grados.

PilotingSettingsMaxTiltCurrent: 5.0

- Distancia máxima que se puede alejar el dron del emisor que lo controla. Unidades en metros.

PilotingSettingsMaxDistanceValue: 10.0

- Este parámetro se pone a 1 si queremos habilitar el parámetro *PilotingSettingsMaxDistanceValue* o a 0 si queremos deshabilitarlo para permitir que el dron se aleje.

PilotingSettingsNoFlyOverMaxDistanceShouldnotflyover: 1

- Altitud mínima a la que se permite volar al dron. Unidades en metros.

PilotingSettingsMinAltitudeCurrent: 0.0

- Máxima velocidad de ascenso o descenso. Unidades entre 0 y 1, siendo 0 el mínimo y 1 el máximo.

SpeedSettingsMaxVerticalSpeedCurrent: 0.2

- Máxima velocidad de rotación en el *Yaw*. Unidades en grados por segundo.

SpeedSettingsMaxRotationSpeedCurrent: 10

- Parámetro que indica si el dron vuela *outdoor* con valor 1 o *indoor* con valor 0.

SpeedSettingsOutdoorOutdoor: 0

El resto de valores quedan fijados por defecto.

```
PilotingSettingsMaxAltitudeCurrent: 1.5  
PilotingSettingsMaxTiltCurrent: 5.0  
PilotingSettingsMaxDistanceValue: 10.0  
PilotingSettingsNoFlyOverMaxDistanceShouldnotflyover: 1  
PilotingSettingsMinAltitudeCurrent: 0.0  
SpeedSettingsMaxVerticalSpeedCurrent: 0.2  
SpeedSettingsMaxRotationSpeedCurrent: 10  
SpeedSettingsOutdoorOutdoor: 0
```

Figura 18: Parámetros añadidos al archivo .yaml.

2.4.2 Topics y mensajes utilizados

Las aplicaciones diseñadas se basan en el tratamiento de imágenes captadas por el dron, para posteriormente enviar los comandos de velocidad oportunos dependiendo de cuál sea el objetivo a realizar.

La secuencia de video de la cámara frontal del Bebop se publica en el *Topic image_raw*. El tipo de mensajes que utiliza ese *Topic* es *sensor_msgs/Image*. Para recibir las imágenes basta con suscribirse a dicho *Topic*.

El *bebop_autonomy* cumple con las especificaciones de interfaz de cámara ROS y publica la información de la cámara y los datos de calibración en el *Topic* llamado *camera_info*. La calidad del flujo de video está limitada a 640x368 a 30 Hz, por tanto se recibirán 30 fotogramas por segundo.

Para enviar la orden de despegue o aterrizaje se necesita publicar un mensaje vacío en los *Topics takeoff* y *land* respectivamente. Para que no se envíen estos comandos de manera continuada se añade --once enviando así el mensaje una sola vez.

```
$ rostopic pub --once /bebop/land std_msgs/Empty
```

```
$ rostopic pub --once /bebop/takeoff std_msgs/Empty
```

Para poder enviar los comandos de velocidad correspondientes a los ejes X, Y, Z y Yaw del dron se necesita publicar mensajes del tipo *geometry_msgs/Twist* en el *Topic cmd_vel*, una vez que el dron haya despegado.

El rango de las variables que componen el mensaje *Twist* está comprendido entre [-1 1], de manera que 0 es la mínima velocidad y 1 y -1 la máxima velocidad, una en sentido opuesto a la otra.

Este comando se restablece a cero cuando se recibe el comando despegue, aterrizaje o emergencia. Para que el Bebop mantenga la posición estática sin desplazarse se tiene que publicar en este *Topic* el mensaje con todas sus variables a cero.

En caso de pérdida de conexión en este campo prevalece el último comando recibido.

El formato completo del mensaje es el siguiente:

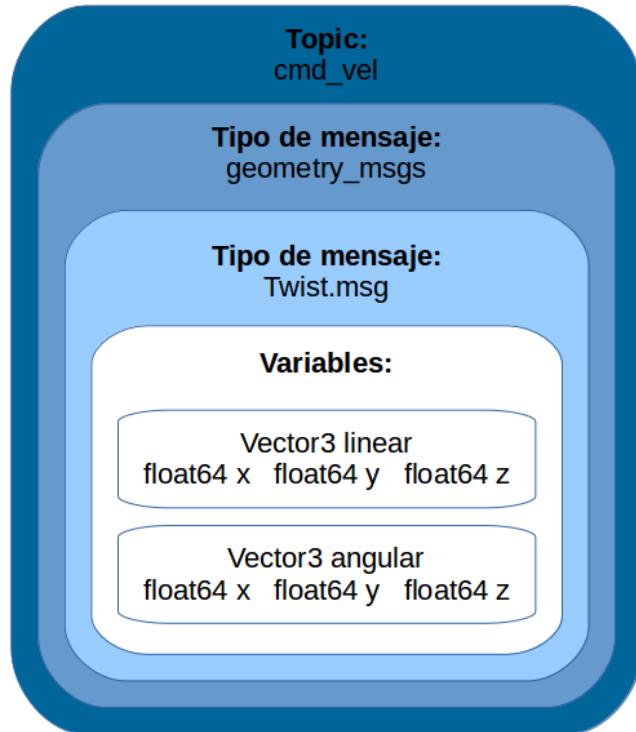


Figura 19: Esquema interno del Topic *cmd_vel*

Las velocidades del Bebop 2 en sentido positivo se muestran en la siguiente imagen:



Figura 20: Esquema de desplazamiento del Bebop 2 con velocidades positivas.

En cuanto a la cámara del Bebop 2, capta imágenes con un campo de visión de 180º, sin embargo sólo se muestra una parte de la imagen en función de la orientación del dron, de manera que la imagen que recibimos es aparentemente estática mientras que la imagen global captada por el dron varía.

En las siguientes imágenes se observa cómo el dron muestra una parte de la imagen captada (recuadro rojo) cuando está en posición horizontal (Figura 21), y cómo cuando se inclina hacia delante la imagen mostrada sigue siendo la misma, pero la imagen global captada es diferente, ya que capta más tierra y menos cielo (Figura 22).

El campo de visión que muestra el dron se puede variar dando valores a las variables de los mensajes *camera_control_msgsAngular.y* y *camera_control_msgsAngular.z*. Si estos dos valores son 0 la cámara se centrará en el centro de la imagen global, mientras que si *Angular.y* vale -30 grados la cámara enfocara 30 grados hacia abajo cuando el dron esté estabilizado horizontalmente.



Figura 21: Imagen captada por el Bebop 2 en posición horizontal.



Figura 22: Imagen captada por el Bebop 2 en posición de avance.

Los mensajes del *Topic camera_control_msgs* son del tipo *geometry_msgs/Twist.msg*, al igual que los del *Topic cmd_vel* explicado anteriormente. Por tanto, para cambiar los valores de los campos es necesario mandar dos vectores de 3 variables, las variables correspondientes con *Linear.x*, *Linear.y*, *Linear.z* y *Angular.x* con valor 0 y *Angular.y* y *Angular.z* con el valor de la orientación de la cámara deseado.

Para ver una lista de todos los *Topics* se puede escribir en línea de comandos *rostopic list*. Para saber información del dron como el nivel de batería, intensidad de la señal recibida, etc, se puede ejecutar en línea de comandos *rostopic echo*.

```
$ rostopic [comando] [Topic] [argumentos]
```

Para conocer todos los comandos disponibles consultar el 2.3.3 Ejecución de ROS y comandos utilizados o la información de ROS *online* [3].

2.5 Librerías OpenCV

OpenCV es una librería libre de visión artificial originalmente desarrollada por Intel. Desde que apareció su primera versión alfa en el mes de enero de 1999, se ha utilizado en infinidad de aplicaciones. Desde sistemas de seguridad con detección de movimiento, hasta aplicaciones de control de procesos donde se requiere reconocimiento de objetos. Esto se debe a que su publicación se da bajo licencia BSD, que permite que sea usada libremente para propósitos comerciales y de investigación con las condiciones en ella expresadas.

OpenCV es multiplataforma, existiendo versiones para GNU/Linux, Mac OS X y Windows. Contiene más de 500 funciones que abarcan una gran gama de áreas en el proceso de visión, como reconocimiento de objetos (reconocimiento facial), calibración de cámaras, visión estéreo y visión robótica.

ROS incorpora, a través de la librería *cv_bridge*, la posibilidad de utilizar de manera rápida y sencilla las librerías *OpenCV* en el formato que proporciona ROS sus imágenes, es decir utilizando mensajes del tipo *sensor_msgs/image*.

Se ha utilizado *OpenCV* a través de *cv_bridge* para el tratamiento y segmentación de las imágenes recibidas desde el dron Bebop 2, en un nodo programado en C que comparte información a través de ROS.

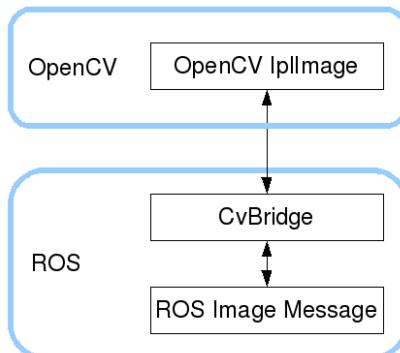


Figura 23: Esquema de uso de OpenCV en ROS

2.6 *Image Processing Toolbox*

Matlab es una potente herramienta de cálculo que, con ayuda del amplio conjunto de *Toolboxes* que proporciona, permite la realización de determinadas acciones con una programación sencilla y eficiente.

En este proyecto se ha utilizado la *Toolbox* de visión artificial *Image Processing Toolbox* para el tratamiento de imágenes.

Image Processing Toolbox proporciona un conjunto completo de algoritmos, funciones y aplicaciones de referencia estándar para procesamiento de imágenes, análisis, visualización y desarrollo de algoritmos. Puede realizar análisis de imagen, segmentación, mejora de imagen, reducción de ruido, transformaciones geométricas y registro de imágenes. Muchas funciones de esta *Toolbox* admiten procesamiento multinúcleo, *GPUs* y generación de código C.

Image Processing Toolbox soporta un conjunto diverso de tipos de imágenes, incluyendo rango dinámico alto, resolución de *gigapixels*, perfil *ICC* integrado y tomografía. Las funciones y aplicaciones de visualización le permiten explorar imágenes y videos, examinar una región concreta de la imagen, ajustar el color y el contraste, crear contornos o histogramas y manipular regiones de interés (*ROIs*). La *Toolbox* admite flujos de trabajo para el procesamiento, visualización y navegación de imágenes grandes.

2.7 *Robotics Systems Toolbox*

Robot System Toolbox es una *Toolbox* de Matlab 2015 que proporciona algoritmos y conectividad con hardware para el desarrollo de aplicaciones autónomas de robótica móvil. Permite representar mapas, planificar trayectorias y seguirlas de forma eficiente con robots autónomos utilizando Matlab o Simulink e integrarlos con los algoritmos de *Robotics System Toolbox*.

Esta *Toolbox* proporciona una interfaz entre Matlab/Simulink y ROS que permite probar y verificar las aplicaciones de los robots incluidos en ROS y utilizar sus simuladores. Es compatible con la generación de código C++, lo que permite generar un nodo ROS a partir de Matlab e introducirlo en la red de ROS, trabajar con los mensajes de ROS, suscribirse y publicar en *Topics*, acceder a los servidores de parámetros de ROS o al árbol de transformadas. En definitiva, poder manejar ROS desde Matlab.

Robotics System Toolbox incluye ejemplos que muestran cómo trabajar con robots virtuales en Gazebo y robots reales habilitados para ROS.

Para poder realizar la conexión entre Matlab y ROS, al ejecutar las aplicaciones desarrolladas sobre Linux directamente, en concreto sobre la versión de Linux 16.04 LTS, y al ejecutar en la misma estación remota ROS y Matlab, no tendremos más que ejecutar el ROS *Master*, y lanzar el nodo de Matlab ejecutando el comando *rosinit*.

De no ser así se tiene que especificar al inicializar el nodo en Matlab la dirección *IP* de la estación remota donde se ejecute el ROS *Master*.

Aplicaciones Desarrolladas

3 – Aplicaciones Desarrolladas

3.1 Introducción

Se ha desarrollado un proyecto lo más completo posible, utilizando distintas herramientas que son habituales hoy en día en el ámbito docente. Por ello, las dos aplicaciones llevadas a cabo utilizan distintos entornos de programación.

La aplicación de Seguimiento de Pasillos mediante Puntos de Fuga se ha desarrollado utilizando el entorno de ROS nativo, con nodos programados en C. Para el tratamiento de imágenes se ha recurrido a las librerías *OpenCV* haciendo uso de ellas a través de las librerías *cv_bridge* que se encuentran integradas en la instalación de ROS.

Para el envío de los comandos de velocidad necesarios para el movimiento autónomo del dron se ha programado un segundo nodo. A continuación se muestra un esquema más visual del planteamiento de la aplicación (Figura 24).

La segunda aplicación, que tiene como objetivo la identificación de un patrón visual predeterminado para su posterior seguimiento mediante el envío de las velocidades correspondientes, se ha desarrollado utilizando el software Matlab, con la ayuda de la las *Toolbox Image Processing Toolbox* y *Robotics System Toolbox*.

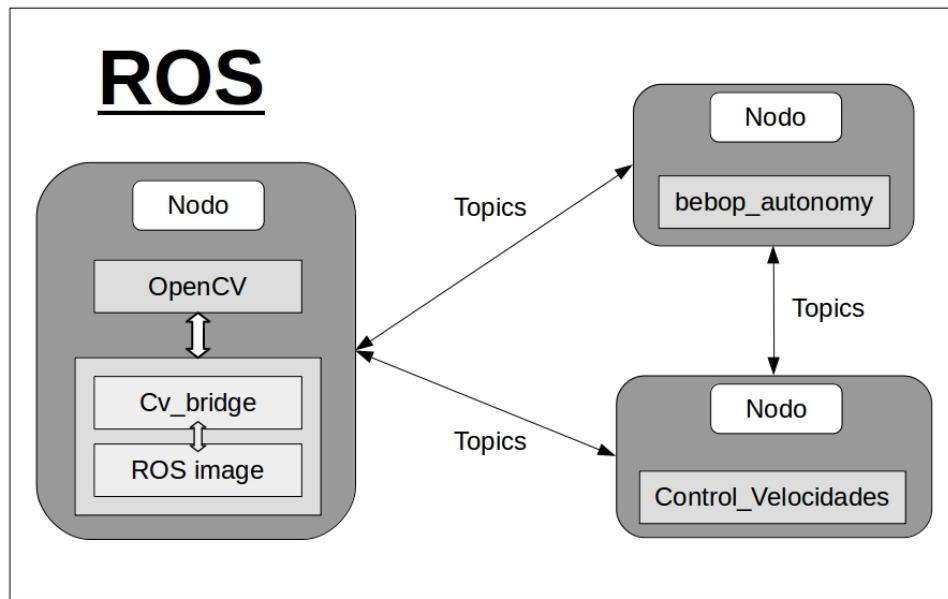


Figura 24: Esquema de la aplicación Seguimiento de Pasillo mediante Puntos de Fuga

Para el desarrollo de un interfaz gráfico que nos permita interactuar con la aplicación de manera rápida se ha utilizado también la herramienta *AppDesigner* incluida en Matlab a partir de su versión de 2016. El esquema de la aplicación es el siguiente:

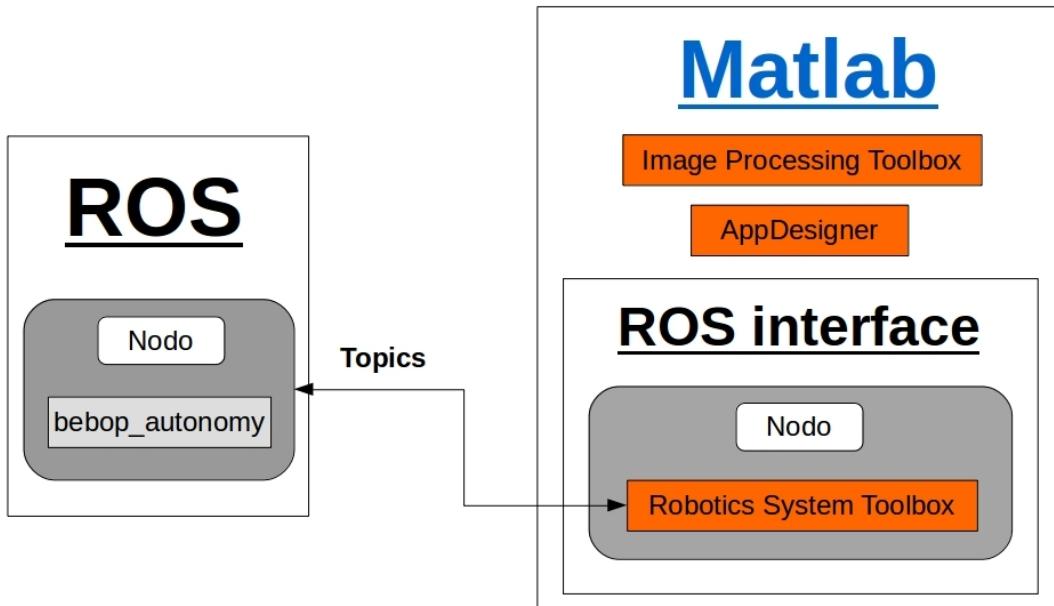


Figura 25: Esquema de la aplicación Seguimiento autónomo de Patrón Visual

3.2 Seguimiento autónomo de pasillos mediante Puntos de Fuga

La primera aplicación desarrollada consiste en el seguimiento autónomo de un pasillo por parte del dron, en la que deberá mantenerse a una altura fija con respecto al suelo y a través de las imágenes recibidas mantenerse centrado en el pasillo a la vez que avanza. La premisa es que la aplicación debe llevarse a cabo únicamente con los datos proporcionados por el dron, es decir, sin añadir sensores adicionales o externos.

La aplicación se basa únicamente en las imágenes que capta el dron, de las cuales, aplicando las correctas técnicas de procesado de imágenes, se extraen una serie de características que permitan posicionar el dron en el pasillo, permitiendo así su avance con seguridad.

Esta aplicación se ejecutará en ROS nativo, utilizando librerías *OpenCV* a través de la librería *cv_bridge* incluida en la instalación de ROS para el tratamiento de las imágenes.

Posicionar un elemento móvil como un dron en el espacio es una tarea compleja, ya que el driver *bebop_autonomy* en el momento de desarrollo de este proyecto no proporcionaba ningún tipo de odometría. Las únicas herramientas que utiliza el dron para mantenerse estático una vez ha despegado son un sensor de ultrasonidos y una cámara ubicadas en la parte inferior del aparato, a cuyas imágenes no tenemos acceso a través del driver.

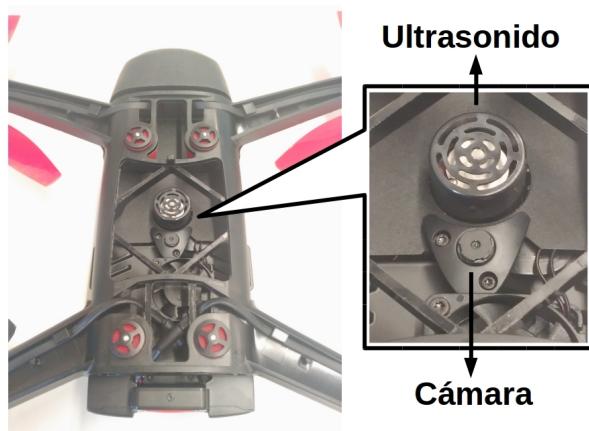


Figura 26: Sensores incluidos en la parte inferior del Bebop 2.

Se va a utilizar la cámara frontal para la extracción de dos características de las imágenes recibidas que permitan saber si el dron está en el centro del pasillo, y si el ángulo que tiene el dron con respecto a la dirección del pasillo es el adecuado, es decir, si está mirando al frente o ha variado el Yaw.



Figura 27: Cámara frontal del Bebop 2.

3.2.1 Características extraídas de las imágenes recibidas

Las dos características extraídas son el Punto de Fuga y la posición de las luces del pasillo en el techo a la que se hace referencia como Punto de Luz.

El Punto de Fuga es el lugar geométrico en el cual las proyecciones de las rectas paralelas a una dirección dada en el espacio convergen. En este caso la dirección es la del pasillo y el Punto de Fuga visto en perspectiva debe quedar en el centro del final del pasillo.

Para entenderlo mejor, en la siguiente imagen del boceto de un pasillo se aprecia claramente cómo el Punto de Fuga se sitúa al fondo del pasillo, donde convergen todas las líneas rectas que van en la dirección del pasillo.

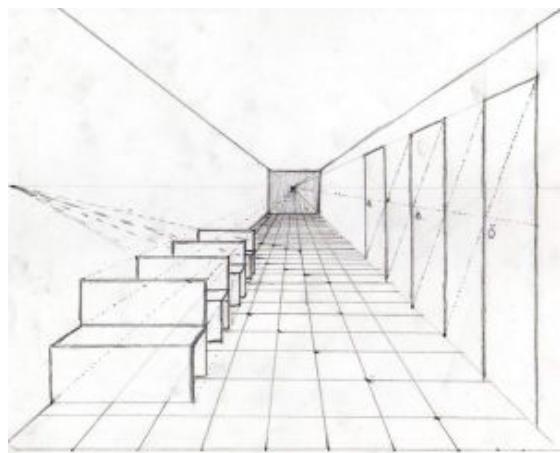


Figura 28: Boceto ejemplo de un Punto de Fuga.

El Punto de Fuga permite identificar la dirección del pasillo, por tanto permite al dron orientarse de manera adecuada variando el Yaw en función de la posición del punto de fuga dentro de la imagen captada, ya que si el dron está bien orientado el Punto de Fuga se encontrará en el centro de la imagen, en cuanto al eje X de la misma se refiere.

Sin embargo, una característica a tener en cuenta del Punto de Fuga es que es independiente de la posición que ocupa el dron dentro del pasillo, es decir, el punto de fuga aparecerá en el centro de la imagen siempre y cuando el dron esté bien orientado, sin importar que esté en la parte derecha del pasillo o en la parte izquierda.

Debido a que esta característica del Punto de Fuga puede parecer poco intuitiva, la siguiente imagen muestra tres fotografías tomadas en un pasillo, orientando la cámara al frente pero posicionándola en la parte izquierda, parte central y parte derecha del pasillo respectivamente.

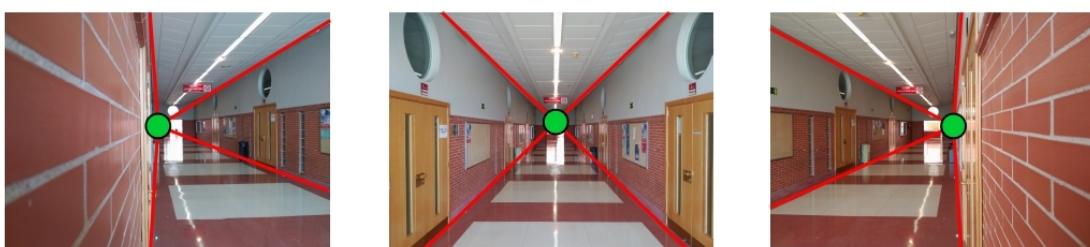


Figura 29: Punto de Fuga desde distintas posiciones de un pasillo.

Se observa que la posición del Punto de Fuga es prácticamente la misma en las tres imágenes mientras que la posición del dron varía de un extremo a otro del pasillo.

Para poder extraer el Punto de Fuga de la imagen, ésta se umbraliza y se hace la transformada de Hough, el Punto de Fuga es el lugar donde convergen las líneas rectas encontradas por la transformada dentro de la imagen.

Es necesario extraer de las imágenes alguna característica que nos aporte información de la posición del dron en el pasillo. En un primer momento se decidió buscar el punto de la imagen donde intersectaban las luces ubicadas en el techo con la parte superior de la imagen, y al que se llamó Punto de Luz.

Punto de Luz

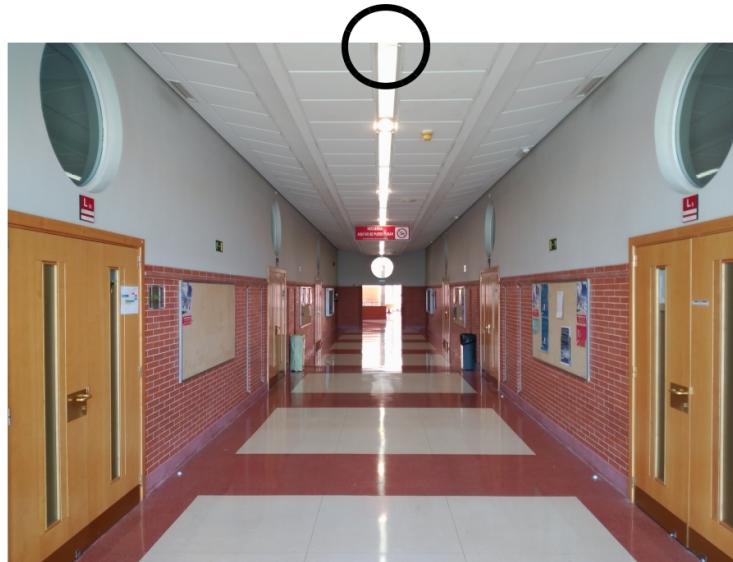


Figura 30: Ubicación del Punto de Luz en una imagen.

Aparentemente con el desplazamiento de este Punto de Luz a lo largo de la parte superior de la imagen debería bastar para saber si el dron se ha desplazado a izquierda o derecha, pero, teniendo en cuenta que el dron varía la posición en el Yaw a la vez que en el eje Y, se da la problemática que se explica a continuación.

Analizando las posibles posiciones del dron en el pasillo y dónde queda el Punto de Luz se observa lo siguiente:

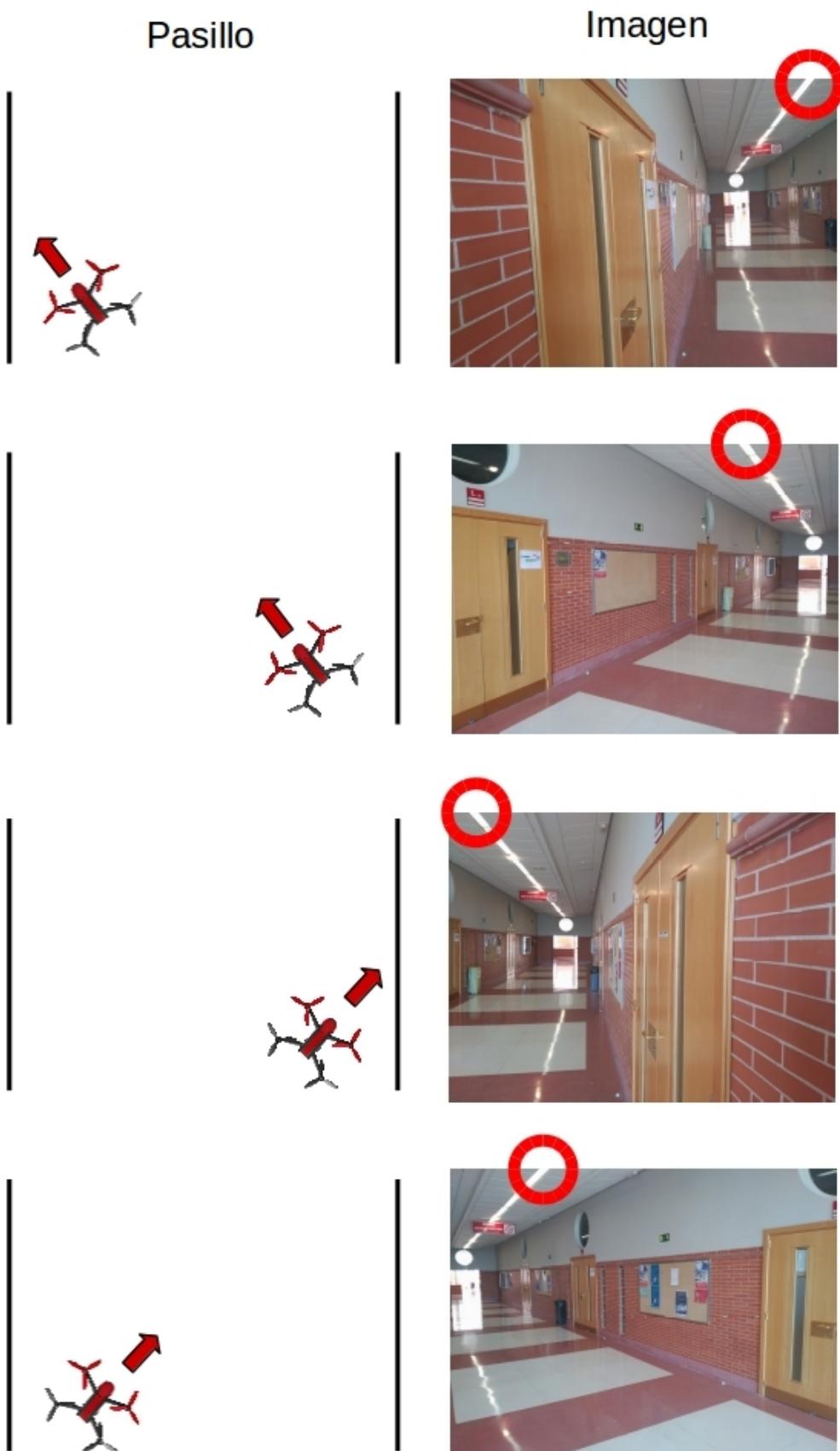


Figura 31: Esquema de posicionamiento del Punto de Luz en una imagen en función de la posición del dron en el pasillo.

En la imagen anterior se observa en el lado izquierdo la posición del dron en el pasillo y en el lado derecho la posición que ocupa el Punto de Luz en la imagen captada por el dron, remarcada por un círculo rojo.

Se puede observar que en el primer caso, estando el dron en la parte izquierda del pasillo y orientado hacia la izquierda, el Punto de Luz queda a la derecha de la imagen. Sin embargo, cuando el dron se encuentra en la parte derecha del pasillo, pero su orientación sigue siendo hacia la izquierda, el Punto de Luz se sigue encontrando en la parte derecha de la imagen.

Esto no aporta información relevante para controlar el dron ya que ante una misma posición del Punto de Luz se deberían enviar en un caso velocidades que posicionasen el dron más a la derecha, como es el primer caso, y en el otro velocidades que posicionasen el dron más a la izquierda, como es el segundo caso.

De igual manera que ocurre en los casos 1 y 2 ocurre en los casos 3 y 4.

Para solventar estas situaciones se ha prestado mas atención a la manera en la que se obtiene el Punto de Luz en la imagen.

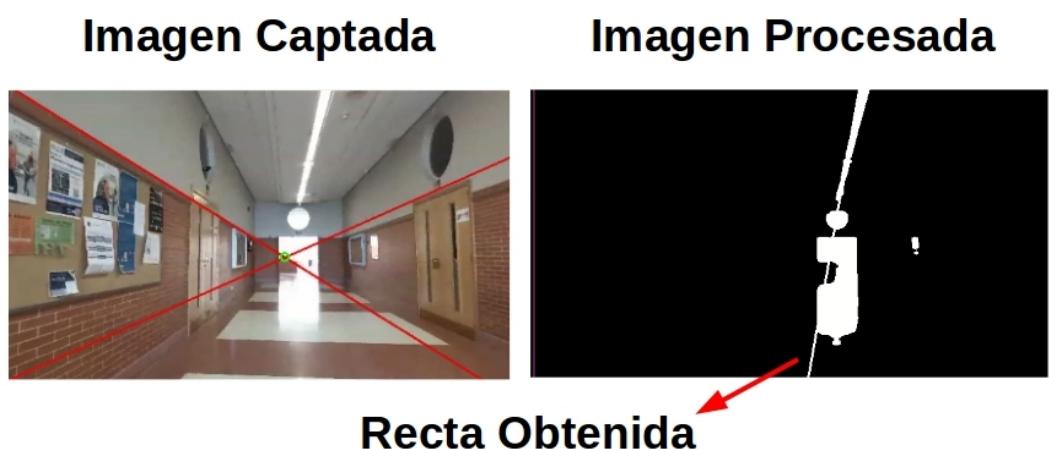


Figura 32: Recta obtenida al hacer la transformada de Hough a las luces de la imagen.

Una vez se binariza y preprocesa la imagen correctamente se hace la transformada de Hough, la cual da lugar a una recta que une todos los puntos en la imagen que componen las luces del techo.

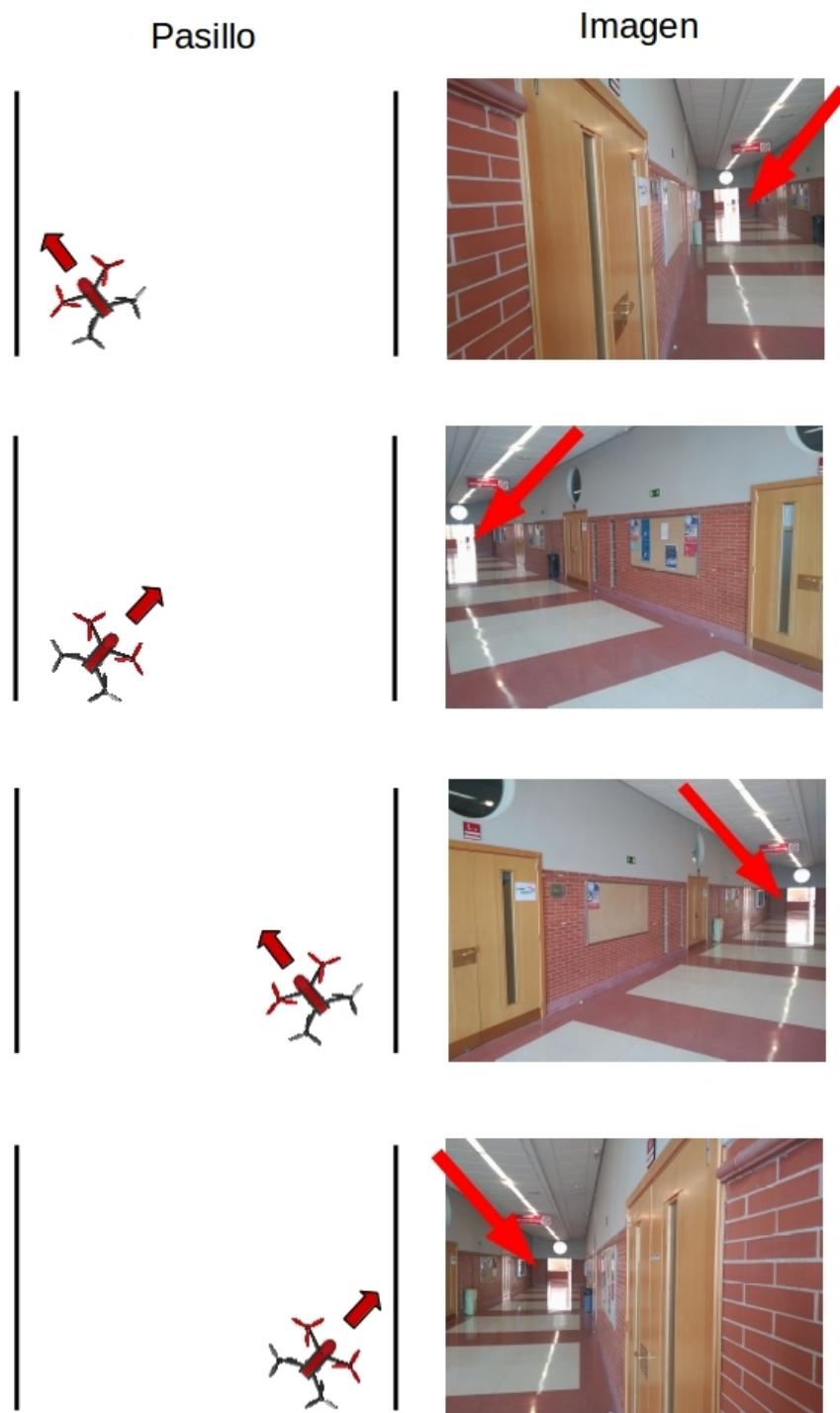


Figura 33: Esquema de orientación del Punto de Luz en función de la posición del dron en el pasillo.

La transformada de Hough, para decidir en qué dirección pinta dicha recta, se basa en una serie de acumuladores, dependiendo de cuál de estos acumuladores sea mayor (si el del lado izquierdo o el del lado derecho) decide con qué ángulo pintar la recta.

En otras palabras, dependiendo de la orientación de los puntos que forman la recta, la transformada de Hough pintará una recta inclinada hacia la derecha o hacia la izquierda.

La recta obtenida de esta manera tiene una peculiaridad, y es que su ángulo es independiente de la orientación del dron en el Yaw. Esto quiere decir que no importa que el dron rote sobre sí mismo, que si se encuentra en la parte izquierda del pasillo la orientación de la transformada de Hough no cambiará.

Esto permite dar velocidades en el eje Y sin cometer errores debido a la orientación.

Hay que tener en cuenta que la probabilidad de que el dron esté perfectamente centrado en el centro del pasillo y perfectamente orientado al frente, de tal forma que la transformada de Hough del Punto de Luz sea una recta totalmente vertical, tiende a cero. Esto es debido a la inercia del dron al volar, a las vibraciones del vuelo, al constante cambio del punto de fuga dentro de la imagen (por pequeño que sea), etc.

Para evitar una oscilación en los desplazamientos laterales del dron cuando está en el centro del pasillo, y teniendo en cuenta que el ángulo de la transformada de Hough del Punto de Luz aumenta cuanto más cerca está el dron de la pared, se ha programado de tal forma que cuando el dron esté en el centro del pasillo dentro de unos umbrales, la velocidad en el eje Y sea cero. Cuando el ángulo de la transformada de Hough del Punto de Luz sobrepasa cierta inclinación, se envían velocidades para volver a centrar el dron.

En la Figura 34 se observa de manera gráfica cómo varía la transformada de Hough del Punto de Luz según la posición del dron en el pasillo. Se observa cómo en efecto, cuando el dron está en el centro del pasillo el ángulo de inclinación de la recta que forma el Punto de Luz es pequeño, en comparación al ángulo que forma cuando está en uno de los extremos del pasillo.

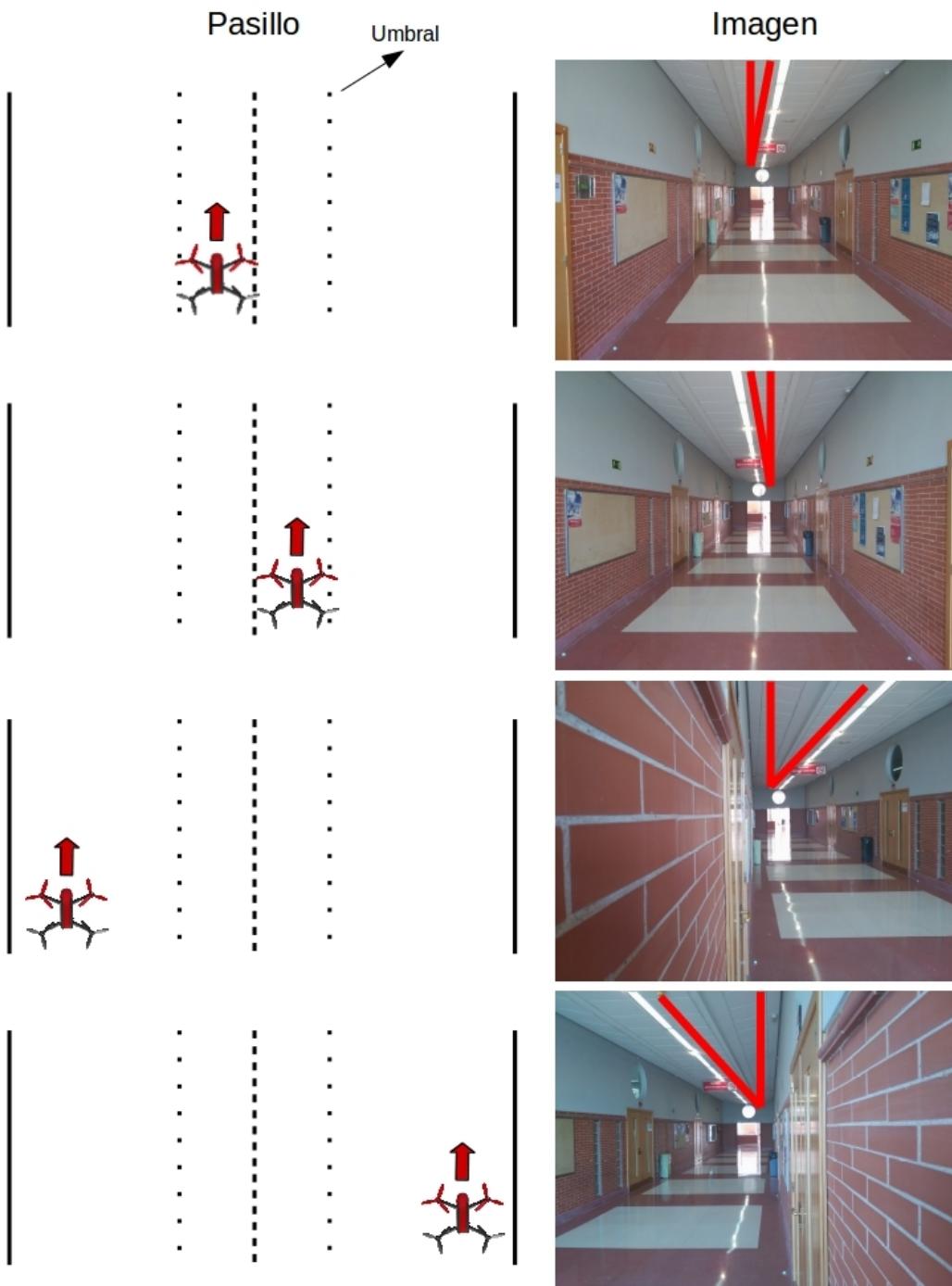


Figura 34: Esquema del ángulo formado por las luces de la imagen en función de la posición del dron en el pasillo.

3.2.2 Programación de la aplicación

Una vez decididas qué características de la imagen se van a extraer para el correcto posicionamiento del dron en el pasillo, se van a crear los nodos de ROS que lleven a cabo esta tarea.

Este proyecto parte de un nodo inicial que busca el Punto de Fuga y el Punto de Luz, y que no ha sido objeto de este proyecto de fin de carrera, con lo cual se explicará brevemente cómo está programado y la información que se ha necesitado extraer de él.

Para que la aplicación funcione es necesario lanzar en ROS tres nodos, el primero es el driver *bebop_autonomy*, el segundo es el nodo que detecta el Punto de Fuga y el Punto de Luz llamado *puntosfuga_bebop.cpp*, y el tercero es el que coge la información del Punto de Fuga y el Punto de Luz extraídos por el segundo nodo y decide qué velocidades enviar al dron. Este último contiene un menú desde el que despegar, aterrizar o comenzar el movimiento del aparato.

Este es un esquema de los nodos utilizados, así como de los *Topics* más importantes a tener en cuenta:

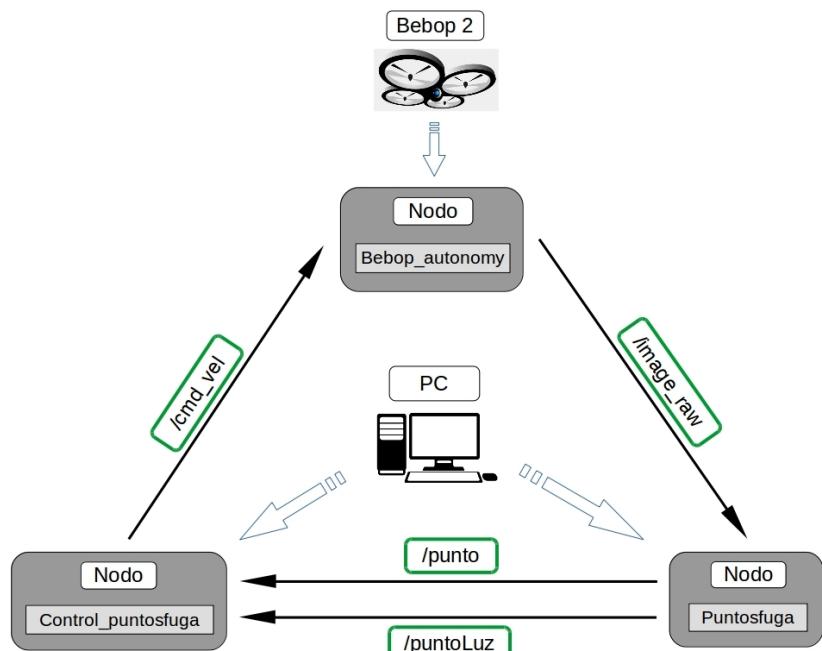


Figura 35: Esquema del flujo de información de la aplicación.

3.2.2.1 Nodo para la obtención del Punto de Fuga y del Punto de Luz

Para la extracción del Punto de Fuga de la imagen el nodo *puntosfuga_bebop.cpp* primero ejecuta un filtro Canny a la imagen recibida, obteniendo una imagen umbralizada donde se muestran todos los bordes captados en la imagen. A continuación, se hace la transformada de Hough y obtiene las rectas formadas por el mayor número de puntos en línea recta. El lugar de la imagen donde interseccionan esas dos rectas es el Punto de Fuga.

Para visualizarlo se ha pintado sobre la imagen real las rectas obtenidas por la transformada de Hough de color rojo y en la intersección se ha pintado un círculo verde que simboliza el Punto de Fuga.

Para la obtención del Punto de Luz se ha umbralizado la imagen y se ha ejecutado de nuevo la transformada de Hough de la imagen binarizada.

La Figura 36 es una captura real de la aplicación en funcionamiento. Vemos en primer lugar, en la parte superior izquierda la imagen recibida del dron con el Punto de Fuga superpuesto, en la parte inferior izquierda la obtención del Punto de Luz con la línea obtenida por la transformada de Hough, y en la parte inferior derecha la imagen que se obtiene al aplicar el filtro Canny a la imagen recibida.

En la Figura 36 la imagen mostrada en la parte superior derecha es la imagen resultante de ejecutar un filtrado a la obtención del Punto de Fuga. Esto es necesario porque, en ocasiones, cuando el dron está cerca de la pared y se desorienta mirando hacia ésta, puede llegar a confundir la textura de la pared con líneas rectas encontrando así Puntos de Fuga erróneos.

Estos Puntos de Fuga erróneos tienen en particular que todos ellos aparecen en la imagen por debajo de donde debería estar el Punto de Fuga correcto, por tanto, se ha establecido un umbral por debajo del cual no se tiene en cuenta el Punto de Fuga, publicando 0 en todas las velocidades para que el dron permanezca estático hasta encontrar el Punto de Fuga correcto (Figura 37).

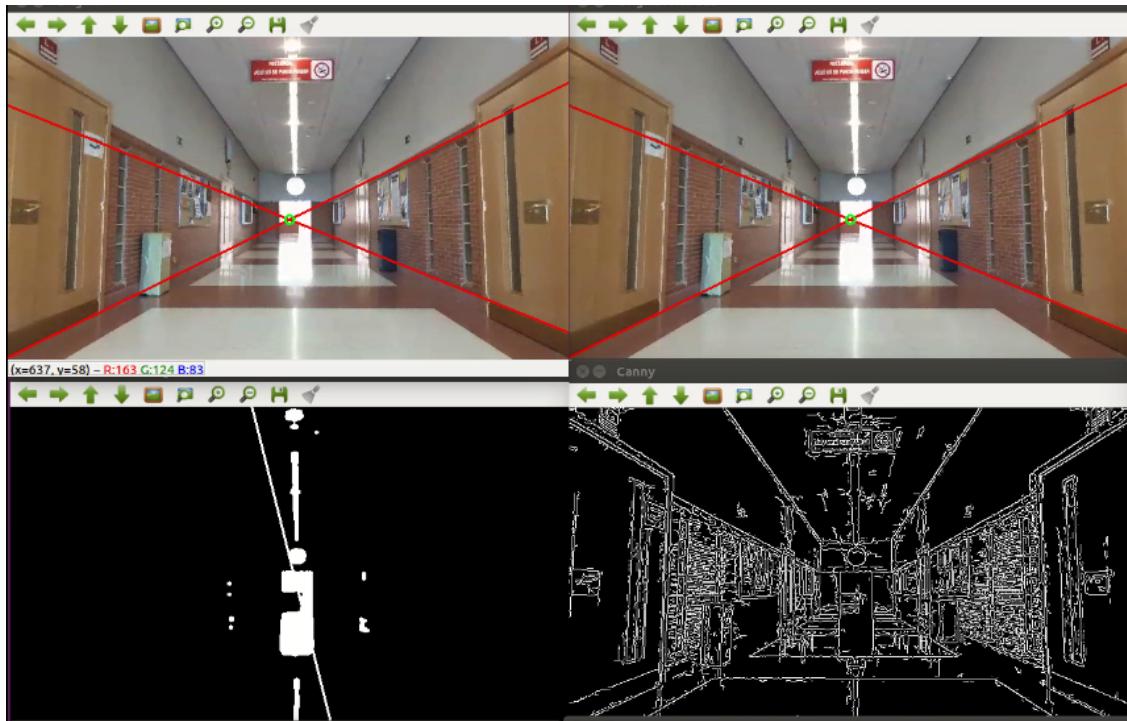


Figura 36: Captura real de las imágenes captadas por el Bebop 2 en la ejecución del seguimiento autónomo de un pasillo.

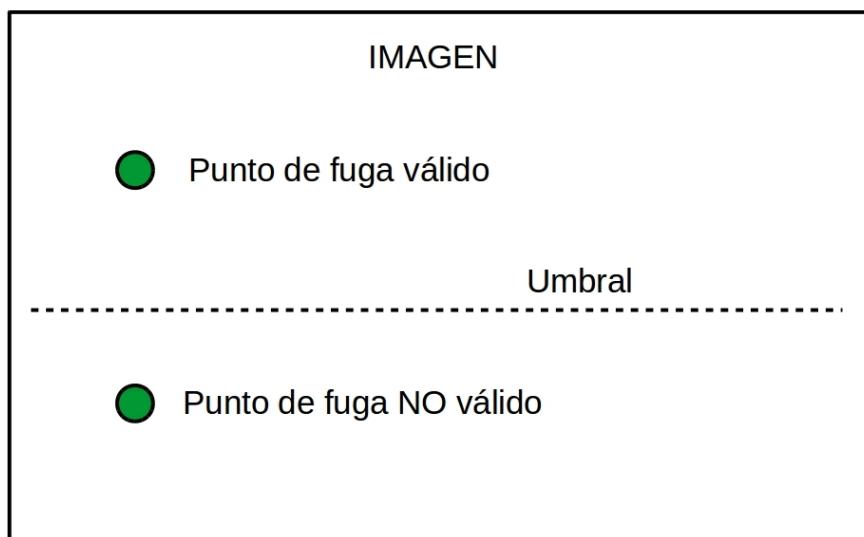


Figura 37: Esquema de posición válida para el Punto de Fuga dentro de la imagen.

En la Figura 38 se aprecia cómo en la imagen sin filtrar (imagen izquierda) se encuentra un Punto de Fuga que no corresponde con el Punto de Fuga del pasillo, y en la imagen filtrada (imagen derecha) prevalece el último Punto de Fuga válido encontrado.



Figura 38: Imagen recibida (izquierda) e imagen filtrada (derecha).

3.2.2.2 Nodo para el envío de velocidades al dron

Una vez obtenida la información necesaria de la imagen se va a explicar el nodo *control_puntosfuga_bebop.cpp* encargado del movimiento del dron.

Este nodo, una vez se ejecuta, lo primero que nos muestra es un pequeño menú, en el que podemos despegar el dron utilizando la tecla *Backspace*, volverlo a aterrizar utilizando la tecla *Enter* o darle permiso para comenzar a moverse utilizando la tecla espacio (*start*).

```
[ INFO] [1499714126.764153683]: Inicializaciones OK
Reading from keyboard
-----
Press 'Enter' to LAND
Press 'Backspace' to TAKEOFF
Press 'Space' to start
```

Figura 39: Captura del menú de inicio de la aplicación.

Si únicamente se despega el dron, este permanecerá inmóvil a una altura fija. Esta es una función propia del dron que se desactiva cuando comienza a recibir velocidades en el *Topic /bebop/cmd_vel*.

Las velocidades establecidas en el *Yaw* dependen de la posición del Punto de Fuga dentro de la imagen recibida, la cual se proporciona en píxeles. La velocidad está normalizada entre -1 y 1, de tal manera que si la componente X del Punto de Fuga se encuentra en el extremo izquierdo de la imagen, la velocidad enviada será 1 y si se encuentra en el extremo derecho será -1.

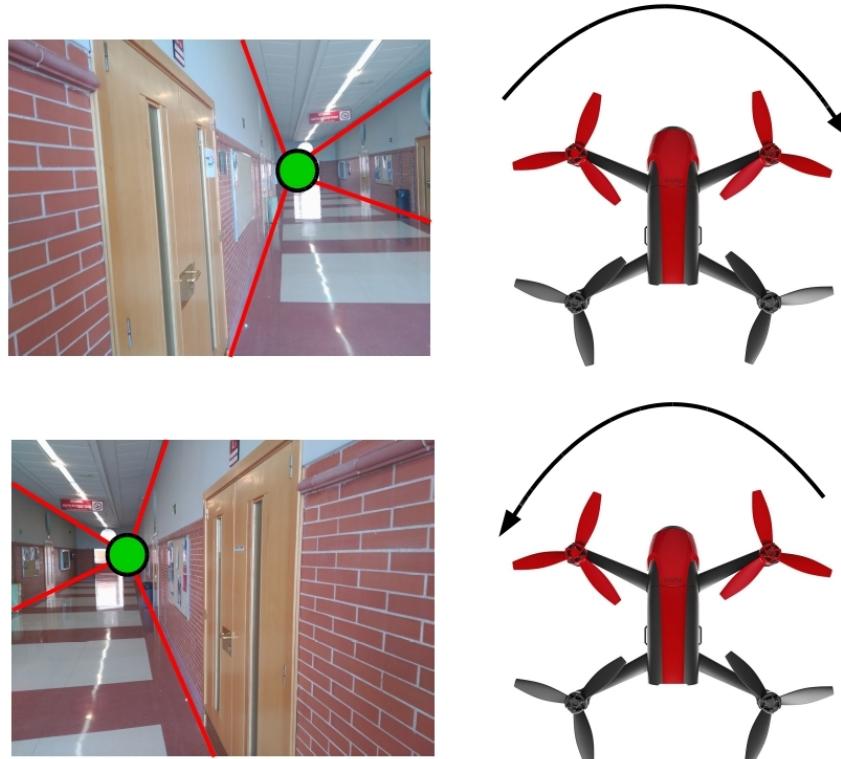


Figura 40: Dirección de giro en función del Punto de Fuga.

Los valores de las velocidades de desplazamiento en el eje Y del dron (desplazamiento lateral) se basan en el ángulo de inclinación de la transformada de Hough obtenida a partir de las luces del techo. La distinción que se hace en este caso se basa en el ángulo de inclinación de dicha recta, ya que si es positivo y supera un cierto valor el dron se moverá con una velocidad fija hacia un lado, y si es negativa y supera cierto valor el dron se moverá con velocidad fija hacia el lado contrario.

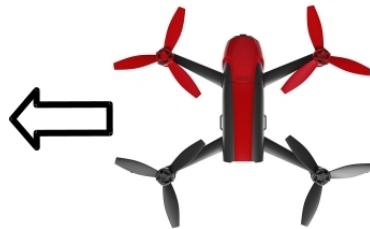


Figura 41: Desplazamiento lateral en función de la inclinación de las luces.

Una vez ejecutado el nodo, comienzan a calcularse las velocidades necesarias para el movimiento del dron, pero no se publican en ningún *Topic* manteniéndose, por tanto, las velocidades en *cmd_vel* a cero. Cuando se pulsa la tecla de la barra espaciadora (*start*) se activa la publicación de velocidades en el *Topic* y el dron comienza a moverse.

En el momento en que se pulsan las teclas de despegue o aterrizaje, se dejan de publicar velocidades en el *Topic cmd_vel* para evitar movimientos no deseados o choques, siendo la última publicación enviada con todas las velocidades a cero.

3.2.2.3 Librerías y *Topics* utilizados

Para ejecutar todo el tratamiento de imágenes necesario en esta aplicación así como el resto de operaciones llevadas a cabo en el nodo son necesarias las siguientes librerías:

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include "geometry_msgs/Pose.h"
#include "geometry_msgs/PoseArray.h"
#include "geometry_msgs/PoseStamped.h"
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
```

Figura 42: Captura de código, librerías involucradas en la aplicación.

Una vez explicado el funcionamiento general de la aplicación se van a explicar los *Topics* utilizados por cada nodo para compartir información entre ellos.

El nodo generado al lanzar el driver *bebop_autonomy* publica un mensaje del tipo */bebop/image_raw* con las imágenes que capta el Bebop 2 sin comprimir. La imagen obtenida es de 640x368 píxeles. Este nodo está suscrito al nodo *control_puntosfuga_bebop.cpp*, que es el encargado de enviarle los comandos de velocidad necesarios para su movimiento.

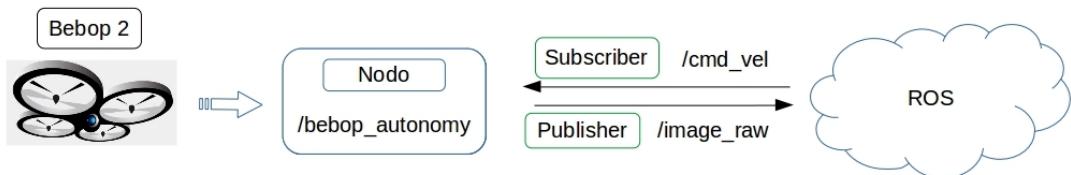


Figura 43: Flujo de información del nodo bebop_autonomy.

El nodo *puntosfuga_bebop.cpp* es el encargado de detectar el Punto de Fuga y el Punto de Luz de la imagen, por tanto está suscrito al *Topic* generado por el *bebop_autonomy* */bebop/image_raw*. Una vez detectados publica los datos en dos *Topics* llamados */puntosfuga/punto* para el Punto de Fuga y */puntosfuga/puntoLuz* para el Punto de Luz.

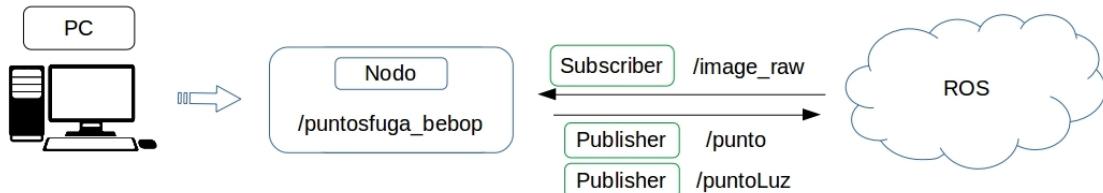


Figura 44: Flujo de información del nodo *puntosfuga_bebop*.

Por último, el nodo *control_puntosfuga_bebop.cpp* es el encargado de, a partir de los datos obtenidos del Punto de Fuga y el Punto de Luz, decidir qué valores de velocidad se publican en el *Topic cmd_vel* para permitir el movimiento del dron. Este nodo por tanto está suscrito a los *Topics* */puntosfuga/punto* y */puntosfuga/puntoLuz* y publica en el *Topic cmd_vel* generado por el *bebop_autonomy*.

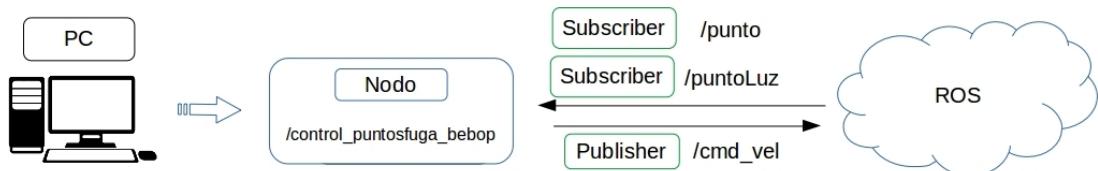


Figura 45: Flujo de información del nodo *control_puntosfuga_bebop*.

3.3 Control de movimiento mediante patrón visual

El objetivo de esta aplicación consiste en conseguir que el dron detecte y siga un patrón visual predeterminado, manteniendo en todo momento una distancia fija entre la persona que porta el patrón y el dron, de manera que, al mover el patrón, el dron reciba las velocidades correspondientes que le permitan seguirlo.

Esta aplicación está diseñada con el software Matlab tanto para el tratamiento de las imágenes obtenidas como para la ejecución del Nodo que controla las velocidades del dron.

El patrón visual está formado por un círculo negro y un círculo blanco más pequeño, ambos concéntricos. Sus medidas son de 21cm x 21cm y los materiales utilizados son cartulina negra y folios.

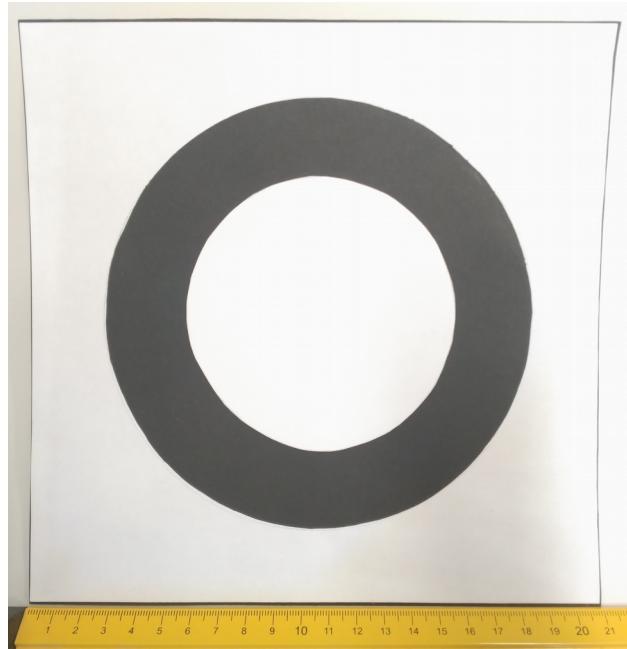


Figura 46: Patrón visual predeterminado.

El patrón consta en la parte trasera de un agarre, necesario para que los dedos no entren en contacto con los círculos concéntricos al sujetar el patrón dificultando la segmentación por parte del dron.

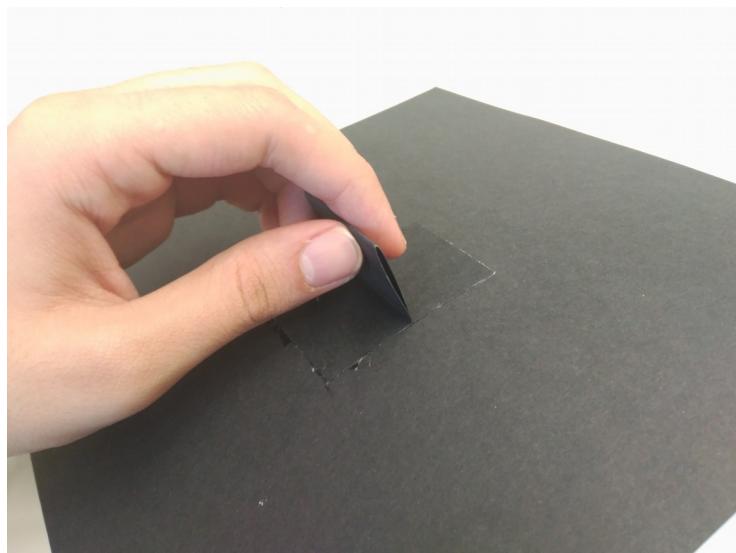


Figura 47: Patrón visual predeterminado, parte trasera.

3.3.1 Programación de la aplicación

Matlab provee, a partir de la versión de 2016a, la *Toolbox* llamada *Robotics System Toolbox*, la cual permite, a través de una serie de funciones, programar un nodo de ROS dentro de Matlab e interactuar con el resto de nodos conectados a un ROS *Master*.

Si el nodo que se ejecuta en Matlab es el primero, es decir no hay otro ROS *Master* ejecutándose, Matlab ejecutará el Nodo de ROS automáticamente como el ROS *Master* y, en caso de que haya otro ROS *Master* ejecutándose Matlab ejecutará el Nodo de ROS como *slave*.

Para ejecutar Ros en Matlab escribimos el comando:

```
rosinit;
```

Cuando se detiene un *script*, el cual estaba ejecutando ROS en Matlab, deja de funcionar pero el nodo de Matlab no se detiene, y no puede haber dos nodos a la vez funcionando en Matlab. Por tanto, cada vez que finalice un *script* o que se quiera ejecutar de nuevo un nodo, antes hay que utilizar la expresión:

```
rosshutdown;
```

Para crear un *subscriber* a un *Topic* determinado se iguala la función *rossubscriber* seguida del *Topic* al que se quiere realizar la subscripción. Es necesario poner después una pausa, aunque su valor sea muy pequeño, para que Matlab continúe la ejecución del *script* correctamente y no se detenga en ese punto.

```
sub = rossubscriber('/bebop/image_raw');  
pause(1);
```

Para crear un *publisher* se iguala la función *rospublisher* seguida del *Topic* en el que se quiere publicar y del tipo de mensaje que comprende ese *Topic*.

```
pub=rospublisher('/bebop/cmd_vel','geometry_msgs/Twist');  
cmd_vel_msgs=rosmessage(pub);
```

Para asignar valores a las variables que se publicarán en un *Topic* basta con mirar en la información de ROS *online* el nombre de dichas variables y de qué tipo son. En este caso, como se explicó en el apartado 2.4.2 Topics y mensajes utilizados, los mensajes del *Topic cmd_vel* son del tipo *Twist* y constan de dos vectores de tres componentes de tipo *float*. Para dar por ejemplo valor a la variable *linear.X* se utilizaría la expresión:

```
cmd_vel_msgs.Linear.X = 0.5;
```

Con la expresión *robotics.Rate* se fija el periodo con el que se quiere actualizar comunicación con ROS, expresado en milisegundos.

```
r=robotics.Rate(20);
```

Con la función *receive* seguida de una variable asociada a un *Topic* al que se esté suscrito, se recibe la información que pasa por dicho *Topic*. Se puede poner como parámetro cada cuántos milisegundos quieras recibir dicha información.

```
msg1 = receive(sub,20);
```

Para publicar información en un *Topic* se utiliza la función *send* seguida de una variable asociada a un *publisher* y del tipo de mensajes que utiliza el *Topic* en cuestión.

```
send(pub,cmd_vel_msgs);
```

La cámara del dron por defecto está inclinada con un cierto ángulo hacia abajo, se requiere que para esta aplicación la cámara esté orientada al frente, por tanto se publicará en el *Topic /bebop/camera_control* los siguientes comandos:

```

pubCamera=rospublisher('/bebop/camera_control','geometry_msgs/Twist');
camera_control_msgs=rosmessage(pubCamera);

camera_control_msgs.Angular.Y=0;
camera_control_msgs.Angular.Z=0;

send(pubCamera,camera_control_msgs);

```

Estos son todos los comandos utilizados en esta aplicación en cuando al uso de ROS se refiere.

3.3.2 Obtención de información. Formación de la imagen recibida

Una vez se está suscrito al *Topic* generado por el *bebop_autonomy* llamado *image_raw* se podría pensar que cada vez que se recibe la información de ese *Topic*, si se guarda en una variable y se utiliza una función que muestre la imagen recibida, como por ejemplo *imshow*, se podrá ver lo que capta el dron, pero la realidad es algo más compleja.

Los datos recibidos del *Topic* *image_raw* en Matlab tienen los siguientes campos:

Property	Value
MessageType	'sensor_msgs/Image'
Header	1x1 Header
Height	368
Width	640
Encoding	'rgb8'
IsBigEndian	0
Step	1920
Data	706560x1 uint8

Figura 48: Captura del workspace de Matlab.
Contenido del Topic *image_raw*.

El campo que contiene todos los píxeles de la imagen es el campo *Data* del mensaje, este campo es una matriz columna de 706560 valores, correspondientes a cada uno de los píxeles. El Bebop codifica los datos de la imagen en formato RGB8, y la imagen tiene un tamaño de 640x368, es decir de 235520 píxeles. Sabiendo esto, se intuye que la información de cada una de las componentes R, G y B de la imagen está en la matriz columna *Data*, ya que $706560/3=235520$. Solo se necesita reordenar los píxeles correctamente.

Los píxeles 1, 2 y 3 corresponden a los primeros píxeles de las componentes RGB, los píxeles 4, 5 y 6 corresponden a los segundos píxeles de las componentes RGB, y así sucesivamente.

Para formar la imagen completa de la componente G por ejemplo, basta con recorrer la matriz *Data* de tres en tres comenzando por el píxel número dos. De esta manera se obtiene la *ImagenG1*.

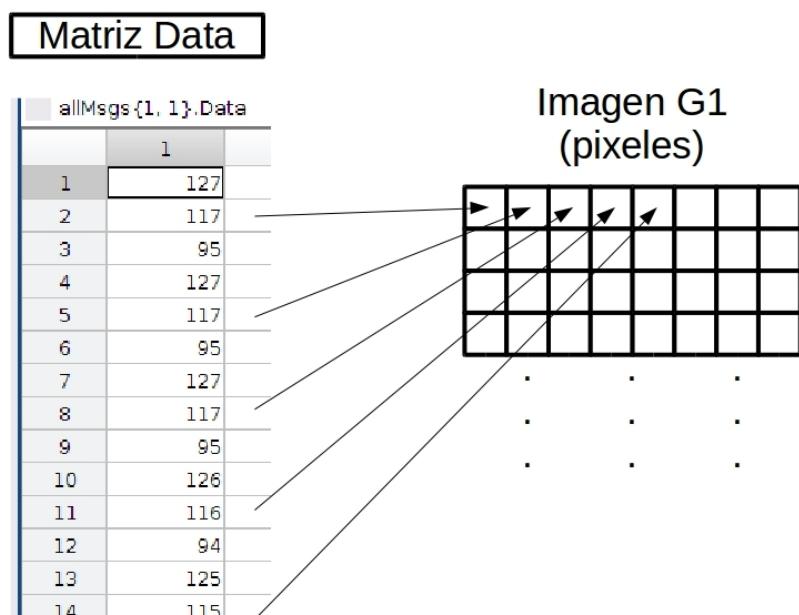


Figura 49: Esquema ejemplo de colocación de píxeles dentro de la componente G de una imagen RGB enviada desde Bebop 2.

El tiempo de procesado de esta aplicación es importante, ya que el dron tiene que responder en tiempo real. Como la segmentación de la imagen se hará en escala de grises no es necesario montar las tres componentes RGB, con obtener una vale. Recorreremos la matriz *Data* para obtener todos los píxeles de la componente G y los guardaremos en la matriz *imagenG*.

```
for indice0 = 2:3:706559  
    imagenG(indiceG,1)=imagen1(indice0,1);  
    indiceG=indiceG+1;  
end
```

A continuación, se ordenan todos los píxeles, primero llenaremos todas las columnas de la fila 1, de izquierda a derecha, a continuación todas las columnas de la fila 2, etc.

```
for fila = 1:1:368  
    for col = 1:1:640  
        imagenG1(fila,col)=imagenG(indice1,1);  
        indice1=indice1+1;  
    end  
end
```

Si mostramos con el comando *imshow* la imagen formada *imagenG1* se puede ver que los píxeles están correctamente ordenados.



Figura 50: Componente G de una imagen RGB recibida desde el Bebop 2.

3.3.3 Patrón visual y Segmentación

El tipo de patrón que debe seguir el dron estaba abierto a la elección del diseñador. Tras hacer pruebas de detección con una pelota roja e intentar segmentaciones por color y forma con resultados poco alentadores, debido a los cambios de iluminación que se producen en la pelota a medida que te desplazas, se decidió el siguiente modelo de patrón.

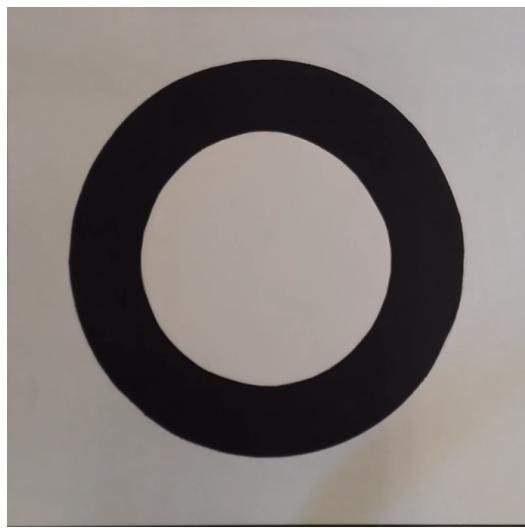


Figura 51: Patrón visual diseñado para la aplicación.

El patrón elegido consta de dos círculos concéntricos, uno de color blanco y uno de color negro. Este patrón está especialmente elegido para optimizar la segmentación que se va a llevar a cabo.

Debido a los problemas encontrados con la iluminación y para favorecer una buena umbralización de la imagen se decidieron los colores blanco y negro, ya que son los colores más opuestos entre sí del espectro visible. Además, si se escogen estos colores la componente RGB a segmentar de la imagen es independiente, ya que en todas obtendremos el mismo resultado.

Para poder encontrar el patrón dentro de la imagen se van a realizar dos búsquedas basadas en dos características distintas y a comparar los resultados.

La primera búsqueda será la de un punto circular, y para ello la búsqueda se basa en la característica de la circularidad. Para ello, Matlab en su *Toolbox* de visión artificial llamada *image processing Toolbox* provee una serie de herramientas muy útiles.

Para detectar un objeto en una imagen y saber si es circular o no, antes hay que tener un valor de circularidad previamente calculado con el que poder comparar. Para calcular este umbral de circularidad hay que entrenar el sistema, y para ello utilizaremos la siguiente imagen.

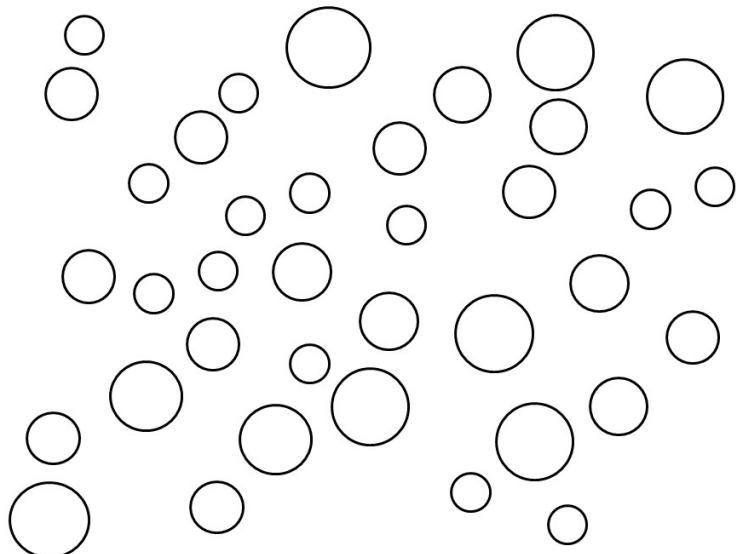


Figura 52: Imagen elegida para el entrenamiento del sistema.

En esta imagen se ven una serie de círculos de diferentes tamaños y que no son perfectamente redondos. Se va a calcular la circularidad de todos ellos y a obtener el valor medio. Este valor medio será el que posteriormente se utilizará para comparar con la circularidad del objeto que capte el dron con la cámara.

El círculo del patrón visual es blanco, y no contiene ningún hueco o agujero en su interior, por tanto primero se adapta la imagen del entrenamiento binarizándola y rellenando los huecos con ayuda de la función *bwfill*. La imagen queda de la siguiente manera:

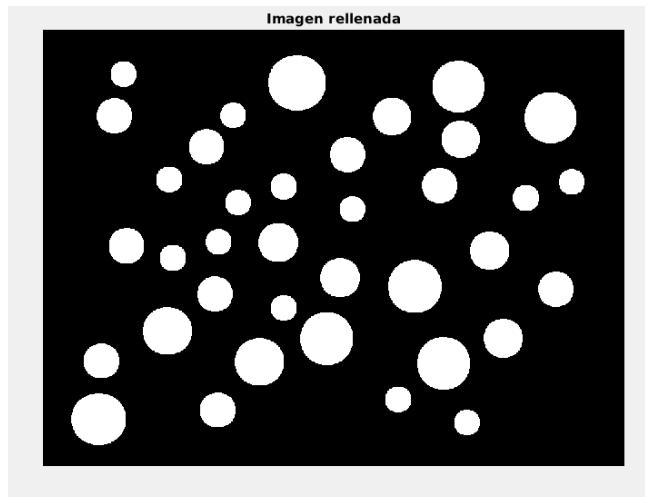


Figura 53: Imagen preprocesada para el entrenamiento del sistema.

Para buscar todos los objetos en la imagen se utilizan las funciones *bwareaopen* y *bwlabel*, que permiten separar uno a uno los elementos encontrados, y, a continuación, se extraen las características de los objetos encontrados con la función *regionprops*.

De todas las características de cada objeto que proporciona la función *regionprops* solo se necesitan dos, el área y el perímetro de cada uno de los círculos. Para calcular la circularidad se utiliza la fórmula:

$$\text{Circularidad} = \frac{\text{Perímetro}^2}{4 * \pi * \text{Área}}$$

Figura 54: Fórmula de la circularidad

Por último, se suman todos los valores de la circularidad obtenidos y se dividen entre el número de objetos totales, obteniendo así la circularidad media. El valor suele ser mayor que 0.9 y menor que 1, ya que obtener un 1 en la media de la circularidad querría decir que todos los círculos del entrenamiento eran perfectamente redondos, y no servirían para este propósito ya que los círculos que va a detectar el dron no van a ser perfectamente redondos.

Ahora que ya se tiene un umbral con el que comparar la circularidad, se hace un tratamiento de la imagen recibida del dron, fotograma a fotograma.

Primero se binariza la imagen y se utiliza un filtro de mediana para eliminar ruido. Hay que tener en cuenta que la velocidad de ejecución de este código no puede ser elevada, ya que, si cuando llega el siguiente fotograma aún se está procesando el anterior, la aplicación comenzará a ralentizarse e incluso llegará a colapsarse, algo que hay que evitar. Por esa razón el preprocesado es simple, pero rápido y efectivo.

Para eliminar ruido y pequeños píxeles u objetos detectados, a la hora de pintar los objetos en la clase 0, se eliminan todos aquellos que son más pequeños que un número de píxeles determinado, porque el patrón aunque se aleje nunca llegará a hacerse muy pequeño en la imagen, ya que el dron tiene que seguirlo.

Al igual que en el entrenamiento, se utiliza la función *regionprops* para extraer el área y el perímetro de los objetos encontrados en la imagen y se calcula la circularidad.

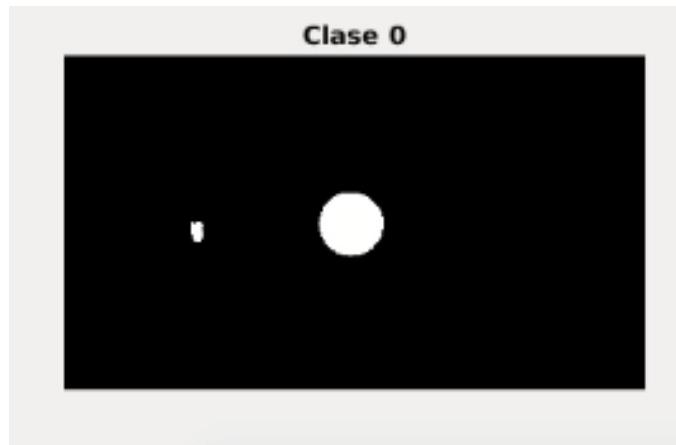


Figura 55: Clase 0 segmentada.

De los objetos que tengan una circularidad igual o mayor a la obtenida en el entrenamiento se extraen los centroides, y se pintan en una clase llamada clase 0 para poder visualizarlos.

La segunda característica que se busca en la imagen recibida se basa en el número de Euler de los objetos encontrados. El número de Euler es un descriptor topológico que informa del número de huecos que tiene un objeto. Este se calcula con la diferencia del número de componentes conectados entre sí en una determinada región menos el número de agujeros de ésta.

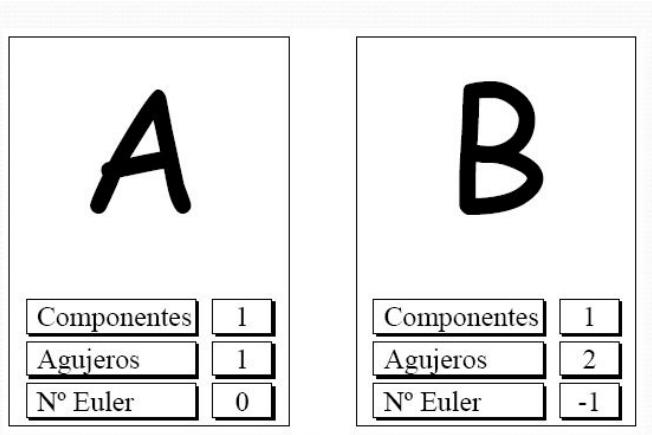


Figura 56: Ejemplo del número de Euler como descriptor topológico.

En este caso se busca un círculo con un solo hueco, por tanto el número de Euler buscado sera 0.

El patrón visual consta de un círculo negro, que tiene en su interior un círculo blanco, si se umbraliza la imagen de tal forma que se invierten los colores nos quedará un círculo blanco que consta en su interior de un círculo negro, es decir, un círculo con un hueco en su interior lo suficientemente grande como para poder detectarlo sin errores.

Una vez encontrados los objetos con el número de Euler adecuado se pintan en una nueva clase llamada clase 1 y se extraen los centroides.

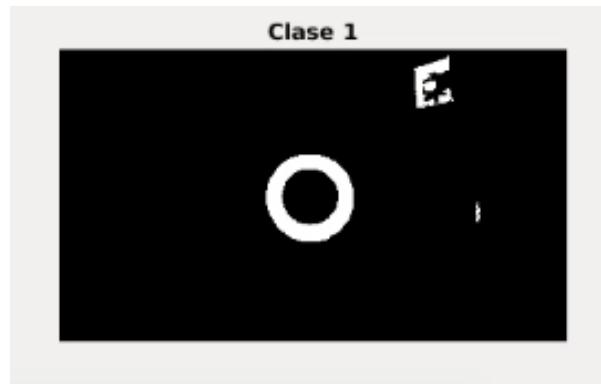


Figura 57: Clase 1 segmentada.

Si uno de los centroides extraído en la clase 0 coincide con uno de los centroides extraído de la clase 1, quiere decir que esos dos objetos son concéntricos, por tanto querrá decir que ese centroide es el que tiene que seguir nuestro dron ya que pertenece a nuestro patrón visual.

En la siguiente imagen se ve en la parte superior derecha la imagen recibida sin tratar, en la parte superior izquierda la imagen umbralizada y filtrada con el centroide encontrado remarcado con un círculo rojo, y en las dos imágenes inferiores las clases 0 y 1 segmentadas.

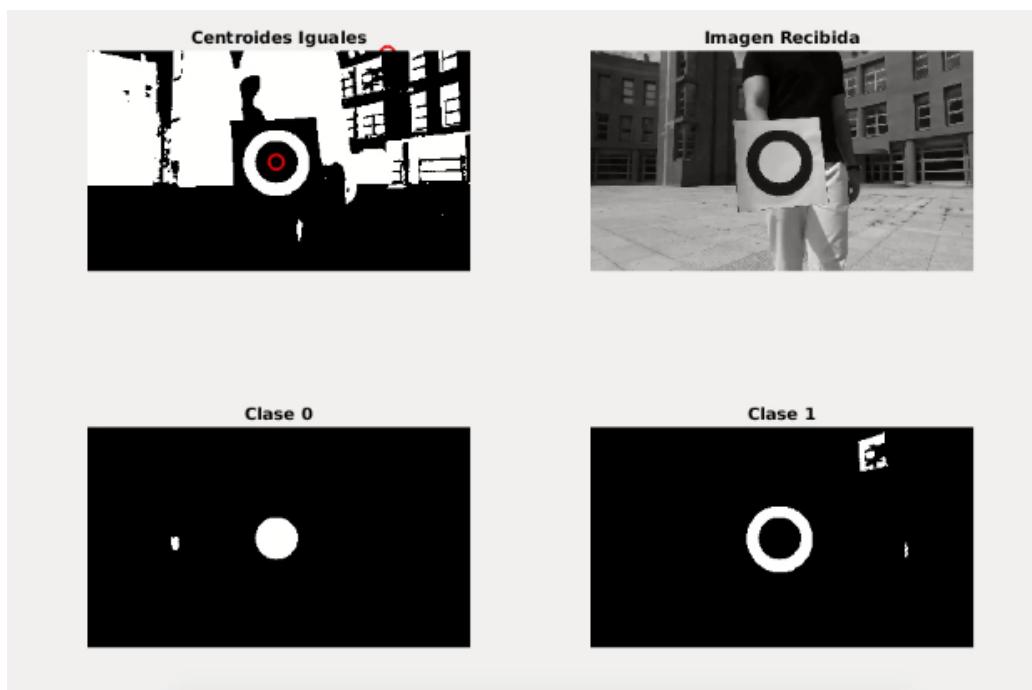


Figura 58: Captura real de la aplicación en funcionamiento.

Los centroides de la clase 0 y la clase 1 es difícil que coincidan en el mismo píxel exacto, por tanto a la hora de decidir o no si ambos centroides son el mismo hay un pequeño umbral, fijado en 20 píxeles. Por tanto, si el centroide de la clase 0 se encuentra en el píxel (124, 210) y el centroide de la clase 1 se encuentra en el píxel (130, 208) el programa interpretará que es el mismo y que pertenece al patrón.

3.3.4 Velocidades en función de la posición del patrón

Las velocidades que se pueden enviar al Bebop 2 en el *Topic cmd_vel* están comprendidas entre 0 y 1, siendo 0 el mínimo, 1 el máximo y -1 el máximo en dirección opuesta.

Hay que tener en cuenta que si se manda una velocidad elevada, pero en el archivo de configuración *.yaml* tenemos limitados los ángulos de inclinación a la hora de desplazarse en el eje X o en el eje Y, prevalecerá lo que imponga dicho archivo, y no la velocidad que se publique en el *Topic*.

Para que el dron siga el patrón visual se publicarán velocidades en el eje X para desplazamientos adelante y atrás, en el Yaw para que rote sobre sí mismo, y en el eje Z para que varíe la altura.

a) Variación de la velocidad lineal en el eje X:

Para dar velocidades en el eje X se ha utilizado el área de uno de los dos círculos buscados en la segmentación, el perteneciente a la clase 1 (véase 3.3.3 Patrón visual y Segmentación), ya que tiene un área mayor y se distinguirá mejor cuando nos alejemos de la cámara.

Para tener una idea de la cantidad de píxeles que comprenden ese área y de cómo varía el área con la distancia se hizo una medida experimental, tomando fotografías del patrón cada 20cm desde la cámara hasta una distancia de 3 metros y se obtuvo el área del círculo segmentado en cada una de ellas.

Los resultados obtenidos se muestran en la Figura 59.

Se observa que la variación del área no es lineal, y que aumenta mucho más rápido en distancias cercanas al dron que en distancias lejanas. Esto a la hora de pilotar el dron se traduce en que si únicamente se multiplica el área del patrón por una constante para que esté entre -1 y 1 y poder publicarla en el *Topic cmd_vel*, al acercarse al dron, este reacciona de forma brusca hacia atrás. Sin embargo, cuando te alejas del dron este se acerca hacia el patrón muy despacio.

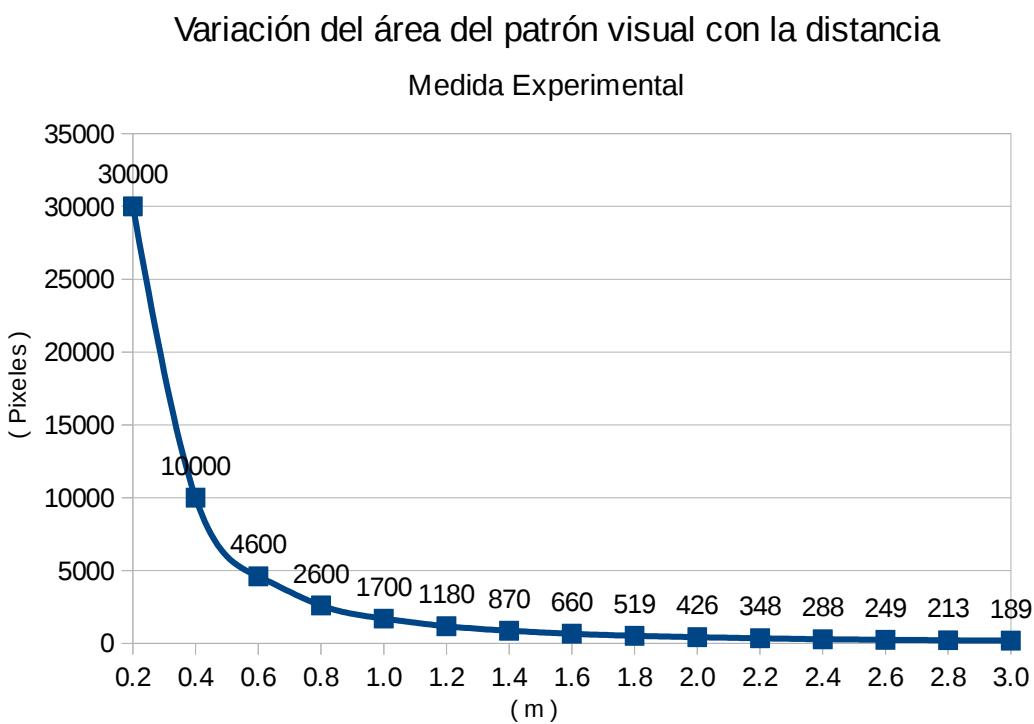


Figura 59: Medida experimental de la variación del área del patrón visual con la distancia.

En nuestro caso, como en el archivo de configuración *.yaml* tenemos la velocidad limitada (los grados que se inclina el dron a la hora de desplazarse), aunque la velocidad aumente rápidamente al acercarnos al dron, este ejecutará un movimiento lento y controlado, ya que el archivo *.yaml* limita el fondo de escala de la velocidad.

El umbral de distancia elegido para que mantenga el dron con el patrón visual es de aproximadamente 1.5 metros, lo que se traduce a un área detectada de unos 600 píxeles.

Para normalizar la velocidad se utilizará por tanto una constante de valor -1/600.

Otra posible opción sería utilizar constantes distintas si el umbral es menor que 600 píxeles y si el umbral es mayor.

b) Variación de la velocidad angular entorno al Yaw:

Para que el dron rote sobre sí mismo se utiliza la ubicación del centroide. Si el centroide se encuentra en la mitad izquierda de la imagen la velocidad del Yaw deberá ser positiva, haciendo al dron girar hacia la izquierda. De igual manera si se encuentra en la mitad derecha de la imagen la velocidad del Yaw deberá ser negativa, haciendo girar al dron a la derecha.

Si se toma como origen el centro de la imagen, el máximo valor que puede tomar el centroide en el eje X de la imagen es de 320 píxeles, ya que la imagen tiene de ancho 640 píxeles. Por tanto, para normalizar la velocidad se utilizará una constante de valor -1/320.

c) Variación de la velocidad lineal en el eje Z:

Para que el dron ascienda o descienda se utiliza la ubicación del centroide. Si el centroide se encuentra en la mitad superior de la imagen la velocidad del eje Z deberá ser positiva, haciendo al dron ascender. De igual manera si se encuentra en la mitad inferior de la imagen la velocidad del eje Z deberá ser negativa, haciendo al dron descender.

Si se toma como origen el centro de la imagen, el máximo valor que puede tomar el centroide en el eje y de la imagen es de 184 píxeles, ya que la imagen tiene de alto 368 píxeles. Por tanto para normalizar la velocidad se utilizará una constante de valor -1/184.

El fragmento de código que da valores a las velocidades es el siguiente:

```
Kvel = -(1/600);
Kx=-(1/320); %referida a la x de la imagen
Ky=-(1/184); %referida a la y de la imagen

(...)

cmd_vel_msgs.Linear.X=Kvel*(Centroides1(j).Area(1,1)-600);
cmd_vel_msgs.Angular.Z=Kx*(Centroides1(j).Centroid(1,1)-320); % Yaw
cmd_vel_msgs.Linear.Z=Ky*(Centroides1(j).Centroid(1,2)-184);

if cmd_vel_msgs.Linear.X > 1
    cmd_vel_msgs.Linear.X = 1;
end
```

3.3.5 Interfaz gráfico con *AppDesigner*

A partir de Matlab 2016a se dispone de la herramienta *AppDesigner* para la creación de interfaces gráficos. Se dispone de una serie de botones, indicadores, *sliders*, etc, que se pueden colocar en la posición deseada para luego completar las funciones predefinidas que cada uno de ellos crea.

El interfaz creado es la siguiente:

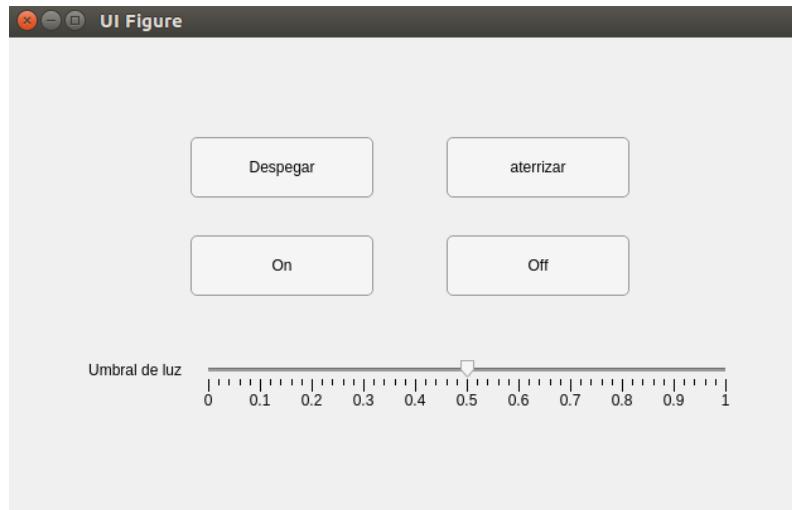


Figura 60: Interfaz gráfico diseñado con Appdesigner en Matlab.

Las rutinas de atención a la interrupción de cada uno de los botones únicamente cambian el valor de una variable, la cual se lee en el *script* de Matlab para hacer lo que deseemos. Para que funcione correctamente dichas variables tienen que ser globales.

```
global start;
global land;
global takeOff;
global umbral;
umbral=0.5;
```

Si se pulsa despegar la variable *takeoff* se pondrá a 1, y se publicará en el *Topic* /bebop/takeoff un mensaje vacío para que el dron despegue.

```
if takeOff == 1
    send(pubTakeoff,takeoff_msgs);
    takeOff = 0;
end
```

Si se pulsa aterrizar la variable *land* se pondrá a 1, y se publicará en el *Topic* */bebop/land* un mensaje vacío para que el dron aterrice.

```
if land == 1
    start=0;
    cmd_vel_msgs.Linear.X=0;
    cmd_vel_msgs.Linear.Y=0;
    cmd_vel_msgs.Linear.Z=0;
    cmd_vel_msgs.Angular.Z=0;
    send(pub,cmd_vel_msgs);

    send(pubLand,land_msgs);
    land = 0;
end
```

Si se pulsa *On* se pondrá a 1 la variable *start*, y se permitirá la publicación de velocidades en el *Topic* */bebop/cmd_vel*, por el contrario, si se pulsa *Off* se inhabilitará la publicación de velocidades en dicho *Topic*, siendo las últimas velocidades publicadas 0, para que el dron permanezca inmóvil.

```
if start==1
    send(pub,cmd_vel_msgs);
end

if start==0
    cmd_vel_msgs.Linear.X=0;
    cmd_vel_msgs.Linear.Y=0;
    cmd_vel_msgs.Linear.Z=0;
    cmd_vel_msgs.Angular.Z=0;
    send(pub,cmd_vel_msgs);
end
```

Si se varía el *slide* umbral nos permite variar el umbral con el que binariza la imagen recibida del dron, así podemos ajustar en tiempo real la visualización en caso de que las condiciones de luz cambien o no sean favorables.

```
Z=im2double(imagenG1);  
bin1=Z>umbral;  
bin2=not(bin1);
```

3.3.6 Carga de *Bags* en Matlab

Es habitual cuando se trabaja con ROS la grabación de *bags* de pruebas sobre el terreno, para después estudiar y procesar los datos obtenidos. En este caso, Matlab también nos da la opción de trabajar con ellos de una manera alternativa a la que se trabaja con ROS nativo.

En ROS nativo, al grabar un *bag* que contiene datos de un *Topic*, el cual contiene imágenes (como las enviadas por el dron en el *Topic image_raw*), basta con reproducir el *bag* y suscribirse al *Topic* para visualizarlo. Matlab también ofrece la posibilidad de suscribirse al *Topic* de ROS para reproducir imágenes, pero también da otra alternativa en el caso de que no se tenga instalado ROS y no se pueda reproducir el *bag*.

Matlab permite guardar una serie de fotogramas en la memoria Ram del ordenador, para después reproducirlos recorriendo el array de datos que se genera. Cuanto mejor sea el equipo utilizado y mayor capacidad de memoria Ram se tenga disponible, más fotogramas se podrán reproducir. Hay que tener en cuenta que si se llena la memoria Ram con fotogramas, la estación de trabajo puede verse afectada ralentizando su funcionamiento.

Para poder extraer los datos de un *bag* en Matlab hay que utilizar las siguientes instrucciones, en el orden que se indica y sin omitir ninguna, tal y como explica Matlab en su ayuda oficial.

En la siguiente expresión se utiliza la función *fullfile* para especificar la ruta donde se encuentra el *bag* dentro del equipo.

```
filePath = fullfile('/home','valero','TFG','bags','pelota','circulo_doble.bag');  
bagselect=rosbag(filePath)
```

La función *select* permite decidir cuántos segundos del *bag* se quieren cargar en la memoria para reproducir en Matlab. En la siguiente expresión se carga solo un segundo del *Topic /bebop/image_raw*, desde “*bagselect.startTime*” que corresponde al inicio del *bag* hasta *bagselect.StartTime + 1*.

```
bagselect2=select(bagselect, 'Time', [bagselect.StartTime bagselect.StartTime + 1], 'Topic',  
'/bebop/image_raw');  
allMsgs = readMessages(bagselect2);  
firstMsgs=readMessages(bagselect2,1:10);
```

Para reproducir el *bag* basta con cargar los datos que nos interesen con un bucle *for*. En este caso es la matriz *Data* que se encuentra dentro del array de *allMsgs*, como hemos cargado un solo segundo hay 30 fotogramas, por tanto el bucle tiene que ir de 1 a 30.

```
for celda = 1:1:30 % Ultimo número de este for es el número de fotogramas  
Imagen{1,1}=allMsgs{celda,1}.Data;  
Imagen1(:,1)=Imagen{1,1};
```


**Resultados,
Conclusiones y
Trabajos Futuros**

4 – Resultados, Conclusiones y

Trabajos Futuros

4.1 Resultados

Se ha conseguido desarrollar las dos aplicaciones de tal manera que cumplen correctamente los objetivos iniciales.

Por un lado, el seguimiento autónomo de un pasillo, que si bien es cierto que los puntos de fuga y los puntos de luz se detectan perfectamente en cualquier tipo de pasillo, a la hora de ejecutar la aplicación es conveniente hacerlo en un pasillo no demasiado estrecho, ya que las velocidades de respuesta del dron son pequeñas para asegurar su integridad y esto hace que adquiera una pequeña inercia a la hora de volar. Con velocidades de respuesta pequeñas el dron ejecutará movimientos amplios y controlados para corregir su trayectoria.

Esta aplicación ha sido probada con éxito en pasillos de 3 metros de ancho, sin embargo en pasillos de 1.5 metros de ancho no se le permitía al dron corregir correctamente su

trayectoria por falta de espacio y chocaba con la pared en el momento en el que adquiría inercia.

El tiempo de ejecución de cada iteracción del código, desde que se obtiene la imagen del dron hasta que se procesa por completo, es de 40ms.

Por otro lado, la aplicación de seguimiento de un patrón visual ha sido probada en condiciones de interior con escasa luz y en condiciones de exterior sin encontrar fallos. El dron seguía correctamente el patrón diseñado manteniendo una distancia de seguridad de 1.5 metros. Si las condiciones de luz cambian repentinamente puede ocurrir una pérdida puntual de la detección del patrón, que se puede ajustar rápidamente en tiempo real variando el umbral de detección desde la interfaz gráfica diseñada.

En este caso, el tiempo de ejecución para cada iteracción del código, desde que se reciben los datos de la imagen hasta que se envía la velocidad correspondiente al dron, oscila alrededor de los 310ms.

Debido a la complejidad de poder observar los resultados de este tipo de aplicaciones en un documento escrito, y a que en el momento del desarrollo de este proyecto el driver *bebop_autonomy* no proporciona ningún tipo de odometría, se han dispuesto en youtube dos videos con una muestra real de dichos resultados.

El canal en el que se podrán encontrar los videos recibe el nombre de “Projectronic” y los nombres y los enlaces para dichos videos son:

- Aplicación para el dron Bebop 2. Seguimiento autónomo de un pasillo mediante puntos de fuga.

<https://www.youtube.com/watch?v=mgsnHm4fsws>

- Aplicación para el dron Bebop 2. Seguimiento autónomo de un patrón visual. Matlab-ROS.

<https://www.youtube.com/watch?v=qEExcvqUgkk>

Además, se ha documentado todo lo utilizado en cuanto a ROS y al driver *bebop_autonomy*, quedando todo el material al servicio del Departamento de Electrónica de la Universidad de Alcalá para su uso futuro en docencia.

4.2 Conclusiones y Trabajos Futuros

El hecho de poder trabajar con un dron, el cual permite el intercambio de información a través de ROS, tiene un gran potencial, no solo como uso lúdico sino para el desarrollo de aplicaciones a nivel profesional.

La capacidad de vuelo permitida por el dron utilizado para este proyecto, el Bebop 2 de Parrot, nos permite no solo la programación de aplicaciones *indoor*, sino que con la antena y el equipo adecuados se puede aumentar su rango de acción permitiendo su uso en labores de reconocimiento de grandes superficies, revisión de infraestructuras, agronomía, vigilancia, etc.

El Departamento de Electrónica de la Universidad de Alcalá apoya el progreso de este sector en auge investigando con varias plataformas pioneras en el desarrollo de aplicaciones a medida con drones, así como su uso en docencia para formar a nuevos profesionales de este sector tan demandado hoy en día.

En el futuro las dos aplicaciones desarrolladas en este proyecto de fin de carrera van a ser utilizadas en cursos de formación de profesionales y como prácticas en asignaturas de robótica.

Presupuesto

5 – Presupuesto

En este capítulo se proporciona información detallada sobre los costes teóricos del desarrollo del proyecto, incluyendo los costes de materiales y las tasas profesionales.

<i>Objeto</i>		<i>Cantidad</i>	<i>Precio unidad (€)</i>	<i>Precio total (€)</i>
Hardware	Portátil MSI	1	1289€	1289€
	Bebop 2	1	450€	450€
	Adaptador Wifi USB	1	25€	25€
Software	Ubuntu 16.04 LTS	1	0,00€	0,00€
	Matlab R2016b	1	0,00€	0,00€
	ROS	1	0,00€	0,00€
	Libre Office	1	0,00€	0,00€
TOTAL				1764,00€

Figura 61: Tabla de costes hardware y software.

Las tasas profesionales incluyen los costes de realización del proyecto correspondientes al tiempo dedicado a su desarrollo. En este caso el equivalente a un Ingeniero en Electrónica de Comunicaciones contratado a media jornada.

<i>Objeto</i>	<i>Tiempo (meses)</i>	<i>Coste Unidad (€/mes)</i>	<i>Coste Total (€)</i>
Ingeniería	3	900€	2700€
Escritura	1	900€	900€
TOTAL			3600€

Figura 62: Tabla de costes de desarrollo.

Los costes totales del proyecto han sido obtenidos sumando los costes materiales y profesionales aplicando el IVA.

<i>Objeto</i>		<i>Costes totales</i>
Costes materiales		1764,00 €
Costes profesionales		3600,00 €
Proyecto	Impresión	60 €
	Encuadernación	40 €
Subtotal		5464 €
IVA (21%)		1147 €
TOTAL		6611,44 €

Figura 63: Tabla de costes totales.

Pliego de condiciones

6 – Pliego de Condiciones

En este apartado se exponen todos los software utilizados y sus versiones, así como el hardware utilizado con todas sus características principales.

- Sistema operativo Ubuntu. Versión 16.04 LTS.
- Meta-Sistema Operativo ROS. Versión Kinetic.
- Software Matlab de MathWorks. Versión R2016b con Licencia Académica.
- Dron Bebop 2 de Parrot. Versión de software 3.0.2. Versión de hardware HW_00. Versión de GPS 2.01F.
- Driver *blop_autonomy*
- Ordenador Portátil:
 - Procesador Intel® Core i7-7700HQ
 - Memoria RAM 16GB DDR4 2133MHz SODIMM
 - Disco duro 1TB (7200 rpm S-ATA) + 256 GB SSD (256GB *1 M.2 SATA)
 - Controlador gráfico Nvidia GeForce GTX 1050Ti 4GB GDDR5
- Antena *Wifi* USB TP LINK AC600.

Manual de Usuario

7 – Manual de Usuario

7.1 - Seguimiento autónomo de pasillo. Ejecución de la aplicación

1- Conectar el dron y establecer la conexión con el ordenador a través de la red *Wifi*.

2- Preparar un terminal con la orden de aterrizaje para casos de emergencia.

Terminal Nuevo

```
$ rostopic pub --once /bebop/land std_msgs/Empty
```

3- Ejecutar el ROS *Master*.

Terminal nuevo

```
$ roscore
```

4- Lanzar el driver *bebop_autonomy* ubicado en la carpeta *bebop_driver – launch*.

Terminal nuevo

```
$ rosrun bebop_driver bebop_node_miconfig.launch
```

5- Lanzar el nodo encargado de buscar los puntos de fuga de nombre *puntosfuga_bebop*.

Terminal nuevo

```
$ rosrun puntosfuga puntosfuga_bebop
```

6- Lanzar el nodo de control de las velocidades del dron de nombre *control_puntosfuga_bebop*.

Terminal nuevo

```
$ rosrun puntosfuga control_puntosfuga_bebop
```

Mantener este terminal seleccionado, ya que la lectura de teclado para las ordenes *start*, despegue y aterrizaje se hace a través de él.

7.2 - Seguimiento de patrón visual. Ejecución de la aplicación

1- Conectar el dron y establecer la conexión con el ordenador a través de la red *Wifi*.

2- Preparar un terminal con la orden de aterrizaje para casos de emergencia.

Terminal Nuevo

```
$ rostopic pub --once /bebop/land std_msgs/Empty
```

3- El ROS *Master* se puede ejecutar desde ROS nativo, o al ejecutar la aplicación de Matlab se ejecutará automáticamente. Para ejecutarlo en ROS nativo:

Terminal nuevo

```
$ roscore
```

4- Lanzar el driver *bebop_autonomy* ubicado en la carpeta *bebop_driver – launch*.

Terminal nuevo

```
$ rosrun bebop_driver bebop_node_miconfig.launch
```

5- Ejecutar Matlab. Ejecutar la aplicación.

Hay que tener en cuenta que la carpeta seleccionada en Matlab debe contener los siguientes archivos para que la aplicación se ejecute correctamente:

- Deteccion_patron_visual.m – Script principal de la aplicación.
- CircunfTrain.jpg – Imagen para el entrenamiento del sistema.
- bebop_app.mlapp – Interfaz gráfico diseñado con *Appdesigner*.

Ejecutar el script *Detección_patron_visual.m* y utilizar el interfaz gráfico que aparecerá para controlar la aplicación. El interfaz cuenta con un botón para Despegar, un botón para Aterrizar, dos botones, *On* y *Off*, para que el dron comience a seguir el patrón visual o lo deje de seguir respectivamente, y un *slide* con el umbral de segmentación de la imagen, que debemos variar si no se detecta correctamente el patrón por condiciones de iluminación.

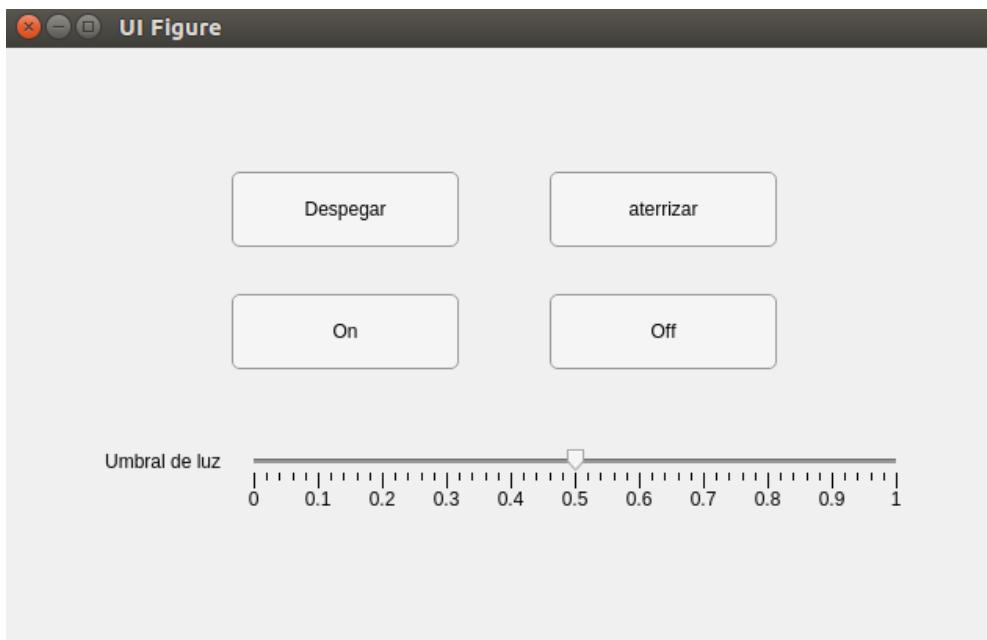


Figura 64: Interfaz gráfica diseñada con *Appdesigner* en Matlab

Planos

8 – Planos

A continuación se muestran todos los códigos involucrados en el desarrollo de las aplicaciones.

8.1 – Seguimiento autónomo de pasillo. Código de la aplicación

El siguiente nodo, programado en C, es el encargado de recibir la imagen captada por el dron y extraer de ella las características del Punto de Fuga y de la posición de las luces en el techo mediante segmentación.

Puntosfuga_bebop.cpp:

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include "geometry_msgs/Pose.h"
#include "geometry_msgs/PoseArray.h"
#include "geometry_msgs/PoseStamped.h"
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
```

```
static const std::string OPENCV_WINDOW1 = "Image window RGB";
static const std::string OPENCV_WINDOW2 = "Image window 32FC1";
static const std::string OPENCV_WINDOW3 = "Image window BN";
static const std::string OPENCV_WINDOW4 = "IXOR";
static const std::string OPENCV_WINDOW5 = "Image segmentation";
static const std::string OPENCV_WINDOW_BN = "Image BN";
static const std::string OPENCV_WINDOW_BN_AUX = "Image BN AUX";

#define PI 3.1415926
#define THMIN    10.0
#define THMAX    70.0

#define WIN_C  40
#define WIN_F  1
#define WIN_ANCHO 240
#define WIN_ALTO 95

struct miRecta
{
int c, f;
float m;
};

void Transformada_Hough (cv::Mat* image, float rho, float theta, struct miRecta *r_izq, struct miRecta *r_der, int *Acumulador_izq, int *Acumulador_der);
void Pintar_recta(cv::Mat* image, struct miRecta recta);
void Calculo_PuntoLuz(cv::Mat* image, struct miRecta recta);
void Calculo_PuntoFuga(cv::Mat* image, struct miRecta r_izq, struct miRecta r_der);
void Calculo_PuntosCarretera(cv::Mat* image, struct miRecta r_izq, struct miRecta r_der);

//Rectas definitivas obtenidas para el punto de fuga

struct miRecta recta_izq;
struct miRecta recta_der;
int Acumulador_der;
int Acumulador_izq;

CvPoint punto1;
CvPoint punto2;

CvRect rectangulo;

geometry_msgs::PoseStamped pose;
geometry_msgs::Point pointPuntoFuga;
geometry_msgs::Point pointLuz;
geometry_msgs::PoseArray poseArray;

ros::Publisher poseArrayPub;
ros::Publisher pointPuntoFugaPub;
ros::Publisher pointLuzPub;
```

```

using namespace cv;
using namespace std;

//PARA GUARDAR DATOS CALIBRACION
FILE* fichero;
int guardar_datos=1;

//convertir pÃ¬xeles en distancia
float distancia_y[12]={2.5,5.0,7.5,10.0,12.5,15.0,17.5,20.0,22.5,25.0,27.5,30.0};
//con y=800,700,600,500 respectivamente
float distancia_x[12]; //cuando y=800,700,600,500 respectivamente
float x_dist[12];
float y_dist[12];

//Punto de fuga
float x_puntofuga;
float y_puntofuga;

//Punto de luz
float x_puntoluz;
float y_puntoluz;

class ImageConverter
{
    ros::NodeHandle nh_;
    image_transport::ImageTransport it_;
    image_transport::Subscriber image_sub_;
    image_transport::Subscriber image_sub_rgb_;
    image_transport::Publisher image_pub_;

    cv_bridge::CvImagePtr cv_depth;
    cv_bridge::CvImagePtr cv_rgb;
    cv_bridge::CvImagePtr cv_rgb_clone;

    cv::Mat image_color;
    cv::Mat image_color_threshold;
    cv::Mat imgThresholded;
    cv::Mat imgCanny;

    cv::Mat image_bn;
    cv::Mat image_bn_aux;
    cv::Mat imghsv;
    cv::Mat imgThresholded2;
    cv::Mat img_color;
    cv::Mat img_color_threshold;
    cv::Mat gray_image;

public:
    ImageConverter()
        : it_(nh_)
    {
        // Subscribe to input video feed and publish output video feed

```

```
    image_sub_rgb_ = it_.subscribe("/bebop/image_raw", 1,
&ImageConverter::imageCb_rgb, this);

//Publicadores
poseArrayPub = nh_.advertise<geometry_msgs::PoseArray>("/puntos_objetivos",
1,true);
    pointPuntoFugaPub =
nh_.advertise<geometry_msgs::Point>("/puntosfuga/puntoPrueba", 1,true);
    pointLuzPub = nh_.advertise<geometry_msgs::Point>("/puntosfuga/puntoLuz",
1,true);

cv::namedWindow(OPENCV_WINDOW1);
cv::namedWindow(OPENCV_WINDOW2);

//PARA GUARDAR DATOS CALIBRACION
/*
    fichero = fopen("datosalibracion.txt", "w");
*/
}

~ImageConverter()
{
    cv::destroyWindow(OPENCV_WINDOW1);
    cv::destroyWindow(OPENCV_WINDOW2);

}

void imageCb_rgb(const sensor_msgs::ImageConstPtr& msg)
{

//Valores para segmentar la carretera en BGR
int iLowB = 220;
int iHighB = 255;
int iLowG = 220;
int iHighG = 255;
int iLowR = 220;
int iHighR = 255;
int punto_actual=0;
int punto_anterior=0;
int resta_puntos=0;

try
{
    cv_rgb = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
    cv_rgb_clone = cv_bridge::toCvCopy(msg,
sensor_msgs::image_encodings::BGR8);

}
catch (cv_bridge::Exception& e)
{
    ROS_ERROR("cv_bridge exception: %s", e.what());
}
```

```

        return;
    }

//Threshold the image
inRange(cv_rgb_clone->image, Scalar(iLowB, iLowG, iLowR), Scalar(iHighB, iHighG,
iHighR), imgThresholded);

//morphological opening (remove small objects from the foreground)
erode(imgThresholded, imgThresholded, getStructuringElement(MORPH_ELLIPSE,
Size(5, 5)) );
dilate( imgThresholded, imgThresholded, getStructuringElement(MORPH_ELLIPSE,
Size(5, 5)) );

//morphological closing (fill small holes in the foreground)
dilate( imgThresholded, imgThresholded, getStructuringElement(MORPH_ELLIPSE,
Size(5, 5)) );
erode(imgThresholded, imgThresholded, getStructuringElement(MORPH_ELLIPSE,
Size(5, 5)) );


//Filtrado de bordes

Canny(cv_rgb_clone->image,imgCanny, 25, 50, 3 );
//imshow("gRAY", imgGray); //show the original image

/*
//CreaciÃ³n del ROI en que se realiza el anÃ¡lisis del pasillo sobre imagen gris
rectangulo=cvRect(WIN_C,WIN_F,WIN_ANCHO,WIN_ALTO);
cvSetImageROI(img_CV_src_gray,rectangulo);
*/
//RealizaciÃ³n de la transformada de Hough
Transformada_Hough(&imgCanny,1,0.0349, &recta_izq, &recta_der,
&Acumulador_izq, &Acumulador_der);

//Mostrar resultados sobre imagen original. TraslaciÃ³n del origen
recta_izq.c+=0;
recta_izq.f+=0;
Pintar_recta(&cv_rgb_clone->image,recta_izq);

recta_der.c+=0;
recta_der.f+=0;
Pintar_recta(&cv_rgb_clone->image,recta_der);

//CÃ¡lculo y dibujo del punto de fuga como intersecciÃ³n de las dos rectas
x_puntofuga=0;
y_puntofuga=0;
Calculo_PuntoFuga(&cv_rgb_clone->image,recta_izq,recta_der);

```

//Publicar punto fuga

```
pointPuntoFuga.x= imgCanny.cols/2-x_puntofuga;
pointPuntoFuga.y=y_puntofuga;
pointPuntoFuga.z=0;
```

//Calcular punto luz

//RealizaciÃ³n de la transformada de Hough

```
Transformada_Hough(&imgThresholded,1,0.0349, &recta_izq,
&recta_der,&Acumulador_izq, &Acumulador_der);
```

```
if (Acumulador_izq>Acumulador_der)
    //Pintar_recta(&imgThresholded,recta_izq);
    Calculo_PuntoLuz(&imgThresholded,recta_izq);
else //Pintar_recta(&imgThresholded,recta_der);
    Calculo_PuntoLuz(&imgThresholded,recta_der);
```

```
pointLuz.x= imgThresholded.cols/2-x_puntoluz;
```

```
pointLuzPub.publish(pointLuz);
```

//Visualizar imÃ¡genes

```
// imshow("Thresholded Image", imgThresholded); //show the thresholded image
imshow("Original", cv_rgb_clone->image); //show the original image
// imshow("Canny", imgCanny); //show the original image
```

```
punto_actual=pointPuntoFuga.y;
resta_puntos= punto_actual-punto_anterior;
if ((resta_puntos<180) && (resta_puntos>-180))
{
    pointPuntoFugaPub.publish(pointPuntoFuga);
    imshow("Original filtrada", cv_rgb_clone->image); //show the original image
    printf("\n X: %f Y: %f resta: %d", pointPuntoFuga.x, pointPuntoFuga.y,
resta_puntos);
    //printf("\nAcc izq: %d con angulo_izq: %f angulo1: %d", currMax,
theta_real*180.0/PI, angulo1);
    //printf("\nAcc der: %d con angulo_der: %f angulo2: %d", currMax,
theta_real*180.0/PI, angulo2);
}
//printf("\n X: %f Y: %f resta: %d", pointPuntoFuga.x, pointPuntoFuga.y,
resta_puntos);
punto_anterior=pointPuntoFuga.y;

cv::waitKey(3);
}
```

```
};
```

```
int main(int argc, char** argv)
{
    ros::init(argc, argv, "image_converter");
```

```

ImageConverter ic;
ros::spin();
return 0;
}

/*
-----DEFINICION DE FUNCIONES-----
*/

```

```

void Transformada_Hough (cv::Mat* image, float rho, float theta, struct miRecta
*r_izq, struct miRecta *r_der, int *Acumulador_izq, int *Acumulador_der)
{
    int width, height;
    int numangle;
    int longrho;
    int numrho;
    int *accum;
    float *tabSin;
    float *tabCos;
    int n;
    float ang;
    int i, j;
    int r;
    int currMax;
    int t;
    int base;
    int thmin;
    int thmax;
    char dato;
    int rhomax=0;
    int thetamax=0;
    float rho_real=0;
    float theta_real=0;
    float irho = 1 / rho;
    int angulo1=0;
    int angulo2=0;

    //obtencion del tamaño de la imagen GRAY
    width = image->cols;
    height = image->rows;

    //printf("\nwidth: %d height:%d", width,height);

    if( width < 0 || height < 0)
    {
        printf("Error de tamaño de imagen");
    }

    //Calculo de la rejilla de acumuladores

```

```
numangle = (int) (PI / theta);
longrho = (width + height) * 2 + 1; /* -(i+j) <= rho <= (i+j) */
numrho = (int) (longrho / rho);

// Reserva de memoria
accum = (int *) malloc( sizeof( int ) * numangle * numrho );
tabSin = (float *) malloc( sizeof( float ) * numangle );
tabCos = (float *) malloc( sizeof( float ) * numangle );

if( tabSin == NULL || tabCos == NULL || accum == NULL )
{
    if( tabSin != 0 )
        free( tabSin );
    if( tabCos != 0 )
        free( tabCos );
    if( accum != 0 )
        free( accum );
    printf("Error al reservar memoria");
}

for( i = 0; i < numangle*numrho; i++ )
{
    accum[i]=0;
}

// calculo de la tabla de senos y cosenos
for( ang = 0, n = 0; n < numangle; ang += theta, n++ )
{
    tabSin[n] = (float) sin( ang );
    tabCos[n] = (float) cos( ang );
}

// Algoritmo Hough incremento acumuladores
for( i = 0; i < width; i++ )
{
    for( j = 0; j < height; j++ )
    {
        dato=image->data[j*width+i]; /* Get (i,j) pixel from image */
        if( dato != 0 )
        {
            for( n = 0; n < numangle; n++ )
            {
                r = cvRound( (i * tabCos[n] + j * tabSin[n]) * irho );
                r += (numrho - 1) / 2;
                accum[n * numrho + r]++;
            }
        }
    }
}

// Obtención de la recta de la izquierda
currMax = 0;
```

```

thmin=(int)((180-THMAX)*PI/(180.0*theta)); //angulos de bÃ³squeda normal
thmax=(int)((180-THMIN)*PI/(180.0*theta));

for( t = thmin; t < thmax; t++ )
{
    for( r = 1; r < numrho-1; r++ )
    {
        base = t * numrho + r;

        if( accum[base] > currMax )
        {
            currMax=accum[base];
            thetamax=t;
            rhomax=r;
        }
    }
}

//rho y theta definitivas
rho_real= (rhomax - (numrho - 1) / 2) * rho;
theta_real = thetamax * theta;
angulo1=theta_real*180.0/PI;
*Acumulador_izq=currMax;

//Imprimo el acumulador de la izquierda
//printf("\nAcc izq: %d con angulo_izq: %f", currMax, theta_real*180.0/PI);

//printf("\nAcc izq: %d con angulo_izq: %f angulo1: %d", currMax,
theta_real*180.0/PI, angulo1);

//calculo de la recta izquierda. Punto respecto de la imagen (0,0) y pendiente
r_izq->m=(float)tan(theta_real+PI/2);
r_izq->c=(int)rho_real*cos(theta_real);
r_izq->f=(int)rho_real*sin(theta_real);

//ObtenciÃ³n de la recta de la derecha

currMax = 0;
thmin=(int)((THMIN)*PI/(180.0*theta)); //angulos de bÃ³squeda normal
thmax=(int)((THMAX)*PI/(180.0*theta));

for( t = thmin; t < thmax; t++ )
{
    for( r = 1; r < numrho-1; r++ )
    {
        base = t * numrho + r;

        if( accum[base] > currMax )
        {
            currMax=accum[base];
            thetamax=t;
            rhomax=r;
        }
    }
}

```

```
}

//rho y theta definitivas
rho_real= (rhomax - (numrho - 1) / 2) * rho;
theta_real = thetamax * theta;
angulo2=theta_real*180.0/PI;
*Acumulador_der=currMax;

//Imprimo el acumulador de la derecha
//printf("\nAcc der: %d con angulo_der: %f", currMax, theta_real*180.0/PI);

//printf("\nAcc der: %d con angulo_der: %f angulo2: %d", currMax,
theta_real*180.0/PI, angulo2);
//printf("\n X: %f Y: %f", pointPuntoFuga.x, pointPuntoFuga.y);

//calculo de la recta derecha. Punto respecto de la imagen (0,0) y pendiente

r_der->m=(float)tan(theta_real-PI/2);
r_der->c=(int)(-1)*rho_real*cos(PI-theta_real);
r_der->f=(int)rho_real*sin(PI-theta_real);

//liberar memoria
free(tabSin );
free(tabCos );
free(accum );

}

void Pintar_recta(cv::Mat* image, struct miRecta recta)
{
    //variables par pintar
    float m,c0,f0;
    int c1,f1,c2,f2;
    int width, height;

    //obencion del tamaÑ±o de la imagen GRAY
    width = image->cols;
    height = image->rows;

    //Imprimo ancho y alto
    //printf("\nwidth: %d height:%d", width,height);

    m=recta.m;
    c0=recta.c;
    f0=recta.f;

    if(m>=0) // recta de la izquierda pendiente negativa
    {
        c1=0;
        f1=f0+m*(c1-c0);

        if(f1>=height)
        {
            f1=height-1;
            c1=((f1-f0)/m)+c0;
        }
    }
}
```

```

}

//c2=width/2; para pintar sÃ³lo hasta la mitad
c2=width;
f2=f0+m*(c2-c0);

if(f2<0)
{
    f2=0;
    c2=((f2-f0)/m)+c0;
}
else //recta de la derecha , pendiente negativa
{
//c1=width/2; Para pintar sÃ³lo hasta la mitad
c1=0;
f1=f0+m*(c1-c0);

if(f1<0)
{
    f1=0;
    c1=((f1-f0)/m)+c0;
}

c2=width;
f2=f0+m*(c2-c0);

if(f2>=height)
{
    f2=height-1;
    c2=((f2-f0)/m)+c0;
}
}

//Imprimo el valor de los puntos
//printf("\nc1: %d f1: %d c2: %d f2: %d", c1,f1,c2,f2);

/*
c1=100;
f1=100;
c2=300;
f2=300;
*/
punto1=cvPoint( c1,f1);
punto2=cvPoint( c2,f2);

cv::line(*image,punto1,punto2,CV_RGB(255,0,0),2,8,0); //color
//cv::line(*image,punto1,punto2,255,2,8); //blanco y negro
/*
if(image->nChannels==1)
    cvLine(image,punto1,punto2,128,2,8);
else
    cvLine(image,punto1,punto2,CV_RGB(255,0,0),2,8);
*/
}

```

```
void Calculo_PuntoFuga(cv::Mat* image, struct miRecta r_izq,struct miRecta r_der)
{
    int x=0,y=0;

    x= (r_der.f-r_izq.f+r_izq.m*r_izq.c-r_der.m*r_der.c)/(r_izq.m-r_der.m);
    y= r_izq.f+r_izq.m*x-r_izq.m*r_izq.c;

    //printf("\n x=%d y=%d", x,y);

    punto1=cvPoint(x,y);
    cv::circle(*image,punto1,5,CV_RGB(0,255,0),2);

    x_puntofuga=x;
    y_puntofuga=y;

}

void Calculo_PuntoLuz(cv::Mat* image, struct miRecta recta)
{
    //variables par pintar
    float m,c0,f0;
    int c1,f1,c2,f2;
    int width, height;

    //obtencion del tamaÑo de la imagen GRAY
    width = image->cols;
    height = image->rows;

    //Imprimo ancho y alto
    //printf("\nwidth: %d height:%d", width,height);

    m=recta.m;
    c0=recta.c;
    f0=recta.f;

    if(m>=0)  // recta de la izquierda pendiente negativa
    {

        f1=0;
        c1=c0+(f1-f0)/m;

        //c2=width/2; para pintar sÃ³lo hasta la mitad
        f2=height;
        c2=c0+(f2-f0)/m;

    }
    else      //recta de la deracha , pendiente negativa
    {
```

```

//c1=width/2; Para pintar sÃ³lo hasta la mitad
f1=0;
c1=c0+(f1-f0)/m;

f2=height;
c2=c0+(f2-f0)/m;

}

//Imprimo el valor de los puntos
printf("\nc1_luz: %d f1_luz: %d c2_luz: %d f2_luz: %d", c1,f1,c2,f2);

/*
c1=100;
f1=100;
c2=300;
f2=300;
*/
punto1=cvPoint( c1,f1);
punto2=cvPoint( c2,f2);

//cv::line(*image,punto1,punto2,CV_RGB(255,0,0),2,8,0); //color
cv::line(*image,punto1,punto2,255,2,8); //blanco y negro
/*
if(image->nChannels==1)
    cvLine(image,punto1,punto2,128,2,8);
else
    cvLine(image,punto1,punto2,CV_RGB(255,0,0),2,8);
*/
x_puntoluz=c1;

}

void Calculo_PuntosCarretera(cv::Mat* image, struct miRecta r_izq,struct miRecta
r_der)
{
    int x_fuga=0,y_fuga=0;
    int x2[12];
    int y2[12];

    int x_distancia[12];
    int y_distancia[12];
}

```

```
float m;

//Primero calcular el punto de fuga

x_fuga= (r_der.f-r_izq.f+r_izq.m*r_izq.c-r_der.m*r_der.c)/(r_izq.m-r_der.m);
y_fuga= r_izq.f+r_izq.m*x_fuga-r_izq.m*r_izq.c;

//printf("\n x_fuga=%d y_fuga=%d", x_fuga,y_fuga);

punto1=cvPoint(x_fuga,y_fuga);
cv::circle(*image,punto1,5,CV_RGB(0,255,0),2);

//calcular los puntos de la carretera
int filas_puntos[]={800,650,589,558,540,528,520,513,508,504,501,300};
float
distancia_filas_puntos[]={2.5,5.0,7.5,10.0,12.5,15.0,17.5,20.0,22.5,25.0,27.5,30.0};

if(x_fuga!=400)
{
    m=(800.0-y_fuga)/(x_fuga-400.0);

    for (int indice=0;indice<12;indice++)
    {
        y2[indice]=filas_puntos[indice];
        x2[indice]=(int)(400+(800-y2[indice])/m);
    }
}
else
{
    for (int indice=0;indice<4;indice++)
    {
        y2[indice]=filas_puntos[indice];
        x2[indice]=y2[indice];
    }
}

//comprobar que los puntos son menores que el punto de fuga
for (int indice=0;indice<12;indice++)
{
    if (y2[indice]<y_fuga)
    {
        x2[indice]=0;
        y2[indice]=0;
    }
}

//Pintar puntos sólo si son mayores del punto de fuga

for (int indice=0;indice<12;indice++)
{
    //printf("\n m=%f x2[%d]=%d y2[%d]=%d",
    m,indice,x2[indice],indice,y2[indice]);
```

```

if (y2[indice]>y_fuga)
{
    punto1=cvPoint(x2[indice],y2[indice]);
    cv::circle(*image,punto1,5,CV_RGB(0,0,255),2);
}
}

float x_30m=25.2;

for (int indice=0;indice<12;indice++)
{
    distancia_x[indice]=x_30m/12.0*(indice+1);

}

for (int indice=0;indice<12;indice++)
{
    if(y2[indice]!=0.0)
    {
        y_dist[indice]=distancia_y[indice];
        x_dist[indice]=(x2[indice]-400)*distancia_x[indice]/400.0;
    }
    else
    {
        y_dist[indice]=0;
        x_dist[indice]=0;
    }
}
/*
//Cuando y=600;

    y_dist[2]=distancia_y[2];
    x_dist[2]=(x2[2]-400)*distancia_x[2]/400.0;

//Cuando y=700;

    y_dist[1]=distancia_y[1];
    x_dist[1]=(x2[1]-400)*distancia_x[1]/400.0;

//Cuando y=800;

    y_dist[0]=distancia_y[0];
    x_dist[0]=(x2[0]-400)*distancia_x[0]/400.0;
}

for (int indice=0;indice<12;indice++)
{
    //printf("\n x_dist[%d]=%f y_dist[%d]=%f distancia_x[%d]=%f distancia_y[%d]=%f",
    indice,x_dist[indice],indice,y_dist[indice],indice,distancia_x[indice],indice,distancia_y[indice]);
}

```

El código que se expone a continuación, es el encargado de enviar las velocidades correspondientes al dron, basándose en la información extraída por el nodo puntosfuga_bebop.cpp. Este nodo también incluye la lectura por teclado de tres teclas para permitir el funcionamiento del menú de la aplicación.

Control_puntosfuga_bebop.cpp

```
//////////  
//  
// Programa: control_puntosfuga_bebop  
//  
//  
// Descripcion: A partir del punto de fuga publicado por el programa  
//               puntosfuga.cpp hacemos el control del bebop 2 mandando  
//               los comandos de velocidad apropiados en el topic cmd_vel  
//  
// input: /bebop/punto    topic del punto de fuga  
//        /bebop/puntoluz   topic del punto de luz  
//  
// output   /bebop/cmd_vel   topic que controla la velocidad del drone  
//  
//  
//////////  
  
//DECLARACION DE LIBRERIAS  
#include <ros/ros.h>  
  
//DECLARACION DE TIPOS DE DATOS  
#include <image_transport/image_transport.h>  
#include <cv_bridge/cv_bridge.h>  
#include <sensor_msgs/image_encodings.h>  
#include "geometry_msgs/Twist.h"  
#include "geometry_msgs/Pose.h"  
#include "geometry_msgs/PoseArray.h"  
#include "geometry_msgs/PoseStamped.h"  
#include <opencv2/imgproc/imgproc.hpp>  
#include <opencv2/highgui/highgui.hpp>  
#include <stdio.h>  
#include <stdlib.h>  
#include <iostream>  
#include <time.h>  
  
#include <termios.h>  
#include <signal.h>  
#include <math.h>  
#include <fcntl.h>  
#include <std_msgs/Empty.h>  
  
#define KEYCODE_SPACE 0x20 //SPACE  
#define KEYCODE_ENTER 0xa  
#define KEYCODE_RETURN 0x7f
```

```

//FUNCION PARA CONTAR EL TIEMPO
unsigned long GetMillisecondsTime()
{
    struct timeval tv;
    if(gettimeofday(&tv,NULL !=0))
        return 0;
    else
        return (unsigned long)((tv.tv_sec*1000ul)+(tv.tv_usec/1000ul));
}

//DECLARACION DE CLASES
class control_puntosfuga_bebop
{
public:
    //DECLARACION DEL CONSTRUCTOR
    control_puntosfuga_bebop();

    //DECLARACION DE CALLBACK PUNTO, aqui entras cada vez que hay un dato
    //nuevo del punto de fuga
    void puntoCallback(const boost::shared_ptr<geometry_msgs::Point const>&
msg);

    //DECLARACION DE CALLBACK PUNTOLUZ, aqui entras cada vez que hay un dato
    //nuevo del punto de luz
    void puntoluzCallback(const boost::shared_ptr<geometry_msgs::Point const>&
msg);

    //DECLARACION DEL TIMER
    void periodico(const ros::TimerEvent& e);

    // ~TeleopPR2Keyboard() { }
    void keyboardLoop();

    //int despega;
    //int aterriza;
    //int start;

private:
    //DECLARAR SUBSCRIBERS, PUBLISHERS, TIMERS etc
    ros::Subscriber punto_sub_;
    ros::Subscriber puntoluz_sub_;

    geometry_msgs::Point actualPunto;
    //geometry_msgs::Point initPunto;
    geometry_msgs::Point actualPuntoluz;
    //geometry_msgs::Point initPuntoluz;

    int initPunto;
    int initPuntoluz;
    int anteriorPunto;
    float restaPuntos;
    int init;
}

```

```
int UmbralCentro;
float velocidadAvance;
float K;           //constante de multiplicacion de las velocidades linearY y angular
unsigned long t1;
unsigned long t2;

int despegaa;
int aterriza;
int start;

ros::Publisher cmd_vel_pub_;
geometry_msgs::Twist velocidades;

ros::Timer timer_;

ros::Publisher land_pub_;
ros::Publisher takeoff_pub_;
std_msgs::Empty land;
std_msgs::Empty takeoff;

//Movimiento
//int movimiento;

};

int kfd = 0;
struct termios cooked, raw;

void quit(int sig)
{
    tcsetattr(kfd, TCSANOW, &cooked);
    exit(0);
}

//CALLBACK PUNTO
void control_puntosfuga_bebop::puntoCallback(const
boost::shared_ptr<geometry_msgs::Point const>& msg)
{
    //SALVAR LA ULTIMA ODOMETRIA MEDIDA
    actualPunto=*msg;
    //ROS_INFO("Callback punto");
    //CUANDO LA PRIMERA ODOMETRIA HAYA SIDO LEIDA ACTIVAMOS EL FLAG
    if (initPunto==0)
    {
        //ROS_INFO("initPunto=1");
        initPunto=1;
    }
}

//CALLBACK PUNTOLUZ
void control_puntosfuga_bebop::puntoluzCallback(const
boost::shared_ptr<geometry_msgs::Point const>& msg)
{
    //SALVAR LA ULTIMA ODOMETRIA MEDIDA
    actualPuntoluz=*msg;
```

```

//ROS_INFO("Callback puntoluz");
//CUANDO LA PRIMERA ODOMETRIA HAYA SIDO LEIDA ACTIVAMOS EL FLAG
if (initPuntoluz==0)
{
    //ROS_INFO("initPuntoluz=1");
    initPuntoluz=1;
}
}

//CALLBACK DEL TIMER
void control_puntosfuga_bebop::periodico(const ros::TimerEvent& e)
{
    //keyboardLoop();

char c;
bool dirty=false;
unsigned long t1;
int bytes_read=0;

int despega=0;
int aterriza=0;
int start=0;
int contador=0;

// int contador=0

// get the console in raw mode
tcgetattr(kfd, &cooked);
memcpy(&raw, &cooked, sizeof(struct termios));
raw.c_iflag &= ~ (ICANON | ECHO);
// Setting a new line, then end of file
raw.c_cc[VEOL] = 1;
raw.c_cc[VEOF] = 2;
//Terminal configuration parameters: (loop each 0.1s -> VTIME=1);(start the timer
in function call ->VMIN=0)
raw.c_cc[VTIME] = 1;
raw.c_cc[VMIN] = 0;
tcsetattr(kfd, TCSANOW, &raw);
//With this config, the keyboard event is set as asynchronous. If the keyboard is not
pressed, the data is sent each 10*0.1 = 1s.

// get the next event from the keyboard
bytes_read=read(kfd,&c,1);
//printf("Valor leido: %x\n",c);
if( bytes_read < 0)
{
    perror("read():");
    exit(-1);
}
else if (bytes_read>0)
    tcflush(kfd,TCIFLUSH); //Clear terminal input buffer

switch(c)
{

case KEYCODE_SPACE:
ROS_INFO("Leo el SPACE");
}
}

```

```
start=1;

t1=GetMillisecondsTime();
dirty=true;
break;

case KEYCODE_ENTER:
ROS_INFO("Leo el ENTER");
//aterriiza=1;
init=0;
//start=0;
land_pub_.publish(land);
dirty=true;
break;

case KEYCODE_RETURN:
ROS_INFO("Leo el RETURN");
//despega=1;
init=0;
takeoff_pub_.publish(takeoff);
dirty=true;
break;

}

//ROS_INFO("Callback timer");
if (initPunto==1 && initPuntoluz==1 && start==1)
{
    //ROS_INFO("Callback timer if");
    init=1;
}

if (init==1)
{
    restaPuntos = actualPunto.x - actualPuntoluz.x;
    ROS_INFO("puntoFugaX %f PuntoLuzX %f restaPuntos %f \n", actualPunto.x,
actualPuntoluz.x, restaPuntos);

//    velocidades.linear.x=0.2;
    velocidades.linear.x=(1)/(actualPunto.x+actualPuntoluz.x);

    if (velocidades.linear.x>velocidadAvance)
    {
        velocidades.linear.x=velocidadAvance;
    }

    if (velocidades.linear.x<0)
    {
        velocidades.linear.x=(-1)*velocidades.linear.x;
    }

//if (abs(actualPunto.x<40))
if ((actualPunto.x<40) && (actualPunto.x>-40))
{
    velocidades.linear.y=K*actualPuntoluz.x;
```

```

    }
else
{
velocidades.linear.y=0;
}

velocidades.angular.z=K*actualPunto.x;

if(abs(actualPunto.x-anteriorPunto)>20)
{
velocidades.linear.y=0;
velocidades.angular.z=0;
}

t2=GetMillisecondsTime();

ROS_INFO("velX %f velY %f velYaw %f Time %lu \n", velocidades.linear.x,
velocidades.linear.y,           velocidades.angular.z, t2);

//cmd_vel_pub_.publish(velocidades);

//land_pub_.publish(land);

anteriorPunto=actualPunto.x;

contador=1;
initPunto=0;
initPuntoluz=0;
ROS_INFO("CONTADOR= %d \n", contador);
}

//CONSTRUCTOR DE LA CLASE
control_puntosfuga_bebop::control_puntosfuga_bebop()
{
//MANEJADOR DE NODOS DE ROS, SE PONE SIEMPRE
ros::NodeHandle n("~");

//SUBSCRIBERS
punto_sub_=n.subscribe<geometry_msgs::Point>
("/puntosfuga/puntoPrueba",1,&control_puntosfuga_bebop::puntoCallback, this);
puntoluz_sub_=n.subscribe<geometry_msgs::Point>
("/puntosfuga/puntoLuz",1,&control_puntosfuga_bebop::puntoluzCallback, this);

land_pub_=n.advertise<std_msgs::Empty>("/bebop/land",1);
//PUBLISHERS
takeoff_pub_=n.advertise<std_msgs::Empty>("/bebop/takeoff",1);
//PUBLISHERS

//PUBLISHERS
cmd_vel_pub_=
n.advertise<geometry_msgs::Twist>("/bebop/cmd_vel",1,true);

//TIMER

```

```
timer_ = n.createTimer (ros::Duration(0.1),
&control_puntosfuga_bebop::periodico,this);

//INICIALIZACI“N DE VARIABLES
init=0;
initPunto=0;
initPuntoluz=0;
anteriorPunto=0;
restaPuntos=0;

despegue=0;
aterriza=0;
start=0;

velocidades.linear.x=0;
velocidades.linear.y=0;
velocidades.linear.z=0;

velocidades.angular.x=0;
velocidades.angular.y=0;
velocidades.angular.z=0;

UmbralCentro=20;
velocidadAvance=1;
//K=1/320;
//K=1;
// K=0.05;
K=0.003;
//movimiento=0;
ROS_INFO("Inicializaciones OK");
}
```

```
//PROGRAMA PRINCIPAL
int main(int argc, char **argv)
{
    //INICIALIZAR ROS
    ros::init(argc,argv,"control_puntosfuga_bebop");

    //CARGAR LA CLASE
    control_puntosfuga_bebop control_puntosfuga_node;

    puts("Reading from keyboard");
    puts("-----");
    puts("Press 'Enter' to LAND");
    puts("Press 'Backspace' to TAKEOFF");
    puts("Press 'Space' to start");

    //TeleopPR2Keyboard tpk;
    //tpk.init();
    //tpk.keyboardLoop();

    //LANZAR EL NODO
    ros::spin();
}
```

8.2 – Seguimiento de patrón visual. Código de la aplicación

El siguiente código corresponde al *script* principal de Matlab que permite la ejecución de la aplicación, pero antes, para el correcto funcionamiento del entrenamiento de la aplicación, se ha utilizado la siguiente imagen:

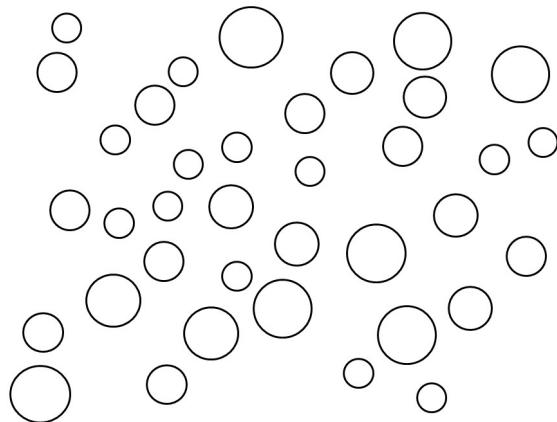


Figura 65: Figura para el entrenamiento del sistema.

```
%% ENTRENAMIENTO CIRCULOS
```

```
close all;
clear all;

I1=imread('CircunfTrain.jpg'); %Carga de la imagen
I2=rgb2gray(I1); %RGB a escala de grises
I3=not(I2>192); %UmbralizaciÃ³n de la imagen
I4=bwfill(I3,'holes'); %Se rellenan los huecos
I5 = bwareaopen(I4,5,4); %DetecciÃ³n de los objetos de la imagen
I6=bwlabel(I5,8); %Etiquetado de los objetos
stats = regionprops(I6,'Perimeter', 'Area', 'EulerNumber', 'PixelIdxList');
%ExtracciÃ³n de caracteristicas
allPerimetersCrellenos = [stats.Perimeter]; %Perimetros de los objetos
allAreasCrellenos = [stats.Area]; %Areas de los objetos
allCircularitiesCrellenos = allPerimetersCrellenos .^ 2 ./ (4 * pi* allAreasCrellenos);
%Circularidad de los objetos
```

```
mediaCircularitiesCrellenos=mean(allCircularitiesCrellenos); %Media de la circularidad
```

```
%% Deteccion en tiempo real
```

```
close all;  
rosshutdown; %Cierre de conexiones de ROS
```

```
%-----%  
% VARIABLES  
%-----%
```

```
global start;  
global land;  
global takeOff;  
global umbral;  
umbral=0.5;
```

```
Kvel = -(1/600); % Constante de velocidad  
Kx = -(1/320); %Referida a la x de la imagen  
Ky = -(1/184); %Referida a la y de la iamgen
```

```
media = 0;
```

```
%-----%  
% INICIALIZACIONES  
%-----%
```

```
rosinit; %inicializacion de ROS en Matlab
```

```
% SET UP CAMARA  
pubCamera=rospublisher('/bebop/camera_control','geometry_msgs/Twist');  
camera_control_msgs=rosmessage(pubCamera);
```

```
camera_control_msgs.Angular.Y=0;  
camera_control_msgs.Angular.Z=0;
```

```
send(pubCamera,camera_control_msgs);
```

```
% SET UP PUBLISHERS DESPEGUE, ATERRIZAJE Y VELOCIDADES
```

```
pubLand=rospublisher('/bebop/land','std_msgs/Empty');  
land_msgs=rosmessage(pubLand);
```

```
pubTakeoff=rospublisher('/bebop/takeoff','std_msgs/Empty');  
takeoff_msgs=rosmessage(pubTakeoff);
```

```
pub=rospublisher('/bebop/cmd_vel','geometry_msgs/Twist');  
cmd_vel_msgs=rosmessage(pub);
```

```
% SET UP SUBSCRIBER IMAGENES DEL DRON
```

```
sub = rossubscriber('/bebop/image_raw');  
pause(1);
```

```
r=robotics.Rate(20); %Periodo de conexión con ROS
```

```
run('bebop_app') %Ejecución del interfaz gráfico
```

```

while(1)

%-----%
%      RECEPCION Y FORMACION DE LA IMAGEN
%-----%

t=cputime; %Para medir el tiempo de ejecucion de cada iteracion

msg1 = receive(sub,20); %Recibimiento de datos desde ROS
imagen1=msg1.Data; %Extraccion de la imagen de los datos recibidos

indiceG=1;
fila=1;
col=1;
indice1=1;

%Extraccion de los pixeles que forman la componente G
for indice0 = 2:3:706559
    imagenG(indiceG,1)=imagen1(indice0,1);
    indiceG=indiceG+1;
end

%Formacion de la componente G de la imagen
for fila = 1:1:368
    for col = 1:1:640
        imagenG1(fila,col)=imagenG(indice1,1);
        indice1=indice1+1;
    end
end

Z=im2double(imagenG1); %Componente G de la imagen RGB

%-----%
%      SEGMENTACION POR CIRCULARIDAD
%-----%

bin1=Z>umbral;
bin2=not(bin1);

L1=medfilt2(bin1,[3 3]); %Filtro de mediana con entorno de vecindad 3x3

stats = regionprops(L1,'Perimeter', 'Area', 'EulerNumber', 'PixelIdxList');

allPerimetersTestCT = [stats.Perimeter];
allAreasTestCT = [stats.Area];
allCircularitiesTestCT = allPerimetersTestCT .^ 2 ./ (4 * pi* allAreasTestCT);

numEulerCrelenos1 = [stats.EulerNumber]';

clase0=zeros(368,640); % tamaÑo de la imagen que tratamos, si no descolocara
los pixeles

for i=1:1:size(numEulerCrelenos1)
    if allCircularitiesTestCT(i) >= mediaCircularitiesCrelenos - 0.3 &&
    allCircularitiesTestCT(i) <= mediaCircularitiesCrelenos + 0.3 && stats(i).Area > 100
    %[0.92 3.08]
        clase0(stats(i).PixelIdxList) = ones(stats(i).Area,1);
    end
end

```

```
    end
end

clase0bin=clase0>0.5;
Centroides0 = regionprops(clase0bin,'Centroid', 'Area', 'PixelIdxList'); % SACAR
SOLO AREA Y CENTROIDE

%-----%
%      SEGMENTACION POR NUMERO DE EULER
%-----%

L=medfilt2(bin2,[3 3]); %Filtro de mediana con entorno de vecindad 3x3

stats1 = regionprops(L,'Perimeter', 'Area', 'EulerNumber', 'PixelIdxList');

numEulerCrellenos = [stats1.EulerNumber];
clase1=zeros(368,640);

for i=1:1:size(numEulerCrellenos)
    if stats1(i).EulerNumber == 0 && stats1(i).Area > 100
        clase1(stats1(i).PixelIdxList) = ones(stats1(i).Area,1);
    end
    AreaAnterior=stats1(i).Area;
end

clase1bin=clase1>0.5;
Centroides1 = regionprops(clase1bin,'Centroid', 'Area', 'PixelIdxList'); % SACAR
SOLO AREA Y CENTROIDE

%-----%
%      SELECCION DEL CENTROIDE DEL PATRON VISUAL
%-----%

error = 20;
objeto=0;
cmd_vel_msgs.Linear.X=0;
cmd_vel_msgs.Linear.Y=0;
cmd_vel_msgs.Linear.Z=0;
cmd_vel_msgs.Angular.Z=0;

for i=1:1:size(Centroides0)
    for j=1:1:size(Centroides1)
        if Centroides0(i).Centroid(1,1) >= Centroides1(j).Centroid(1,1) - error &&
Centroides0(i).Centroid(1,1) <= Centroides1(j).Centroid(1,1) + error
            if Centroides0(i).Centroid(1,2) >= Centroides1(j).Centroid(1,2) - error &&
Centroides0(i).Centroid(1,2) <= Centroides1(j).Centroid(1,2) + error
                subplot(2,2,1), imshow(L); title('Centroides Iguales');
                hold on;

plot(Centroides0(i).Centroid(1,1),Centroides0(i).Centroid(1,2),'or','LineWidth',2,'MarkerSize',10); %pinta circulo alrededor del centroide

                objeto=i;
                cmd_vel_msgs.Linear.X=Kvel*(Centroides1(j).Area(1,1)-600);
%Desplazamiento frontal
                cmd_vel_msgs.Angular.Z=Kx*(Centroides1(j).Centroid(1,1)-320); % Yaw
                cmd_vel_msgs.Linear.Z=Ky*(Centroides1(j).Centroid(1,2)-184);
%Desplazamiento lateral
            end
    end
end
```

```

        end
    end
end

if cmd_vel_msgs.Linear.X > 1
    cmd_vel_msgs.Linear.X = 1;
end

%-----%
%          VISUALIZACION
%-----%

subplot(2,2,2), imshow(imagenG1); title('Imagen Recibida');
subplot(2,2,3), imshow(clase0); title('Clase 0');
subplot(2,2,4), imshow(clase1); title('Clase 1');

%-----%
%          PUBLICACION DE MENSAJES EN ROS
%-----%

if start==1
    send(pub,cmd_vel_msgs);
end

if start==0
    cmd_vel_msgs.Linear.X=0;
    cmd_vel_msgs.Linear.Y=0;
    cmd_vel_msgs.Linear.Z=0;
    cmd_vel_msgs.Angular.Z=0;
    send(pub,cmd_vel_msgs);
end

if land == 1
    start=0;
    cmd_vel_msgs.Linear.X=0;
    cmd_vel_msgs.Linear.Y=0;
    cmd_vel_msgs.Linear.Z=0;
    cmd_vel_msgs.Angular.Z=0;
    send(pub,cmd_vel_msgs);

    send(pubLand,land_msgs);
    land = 0;
end

if takeOff == 1
    send(pubTakeoff,takeoff_msgs);
    takeOff = 0;
end

e=cputime-t % Fin de la iteracion, medicion de tiempos de cpu
end

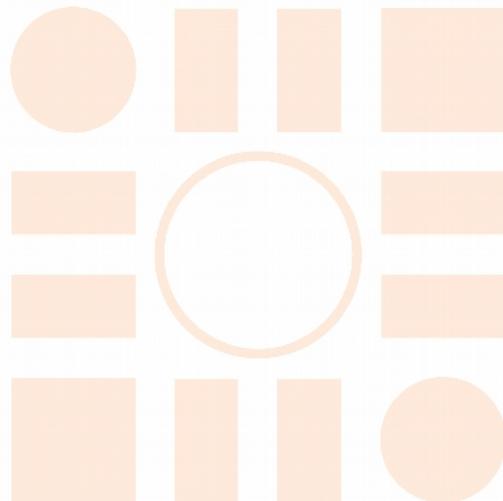
```


Bibliografía

9 – Bibliografía

- [1] Página oficial de DJI - <http://www.dji.com/es>
- [2] Página oficial de Parrot - <https://www.parrot.com/es/drones/parrot-bebop-2>
- [3] Página oficial de ROS - <http://wiki.ros.org/>
- [4] Página oficial de Matlab - <https://es.mathworks.com/>
- [5] Documentación *online* del driver *bebop_autonomy* – <http://bebop-autonomy.readthedocs.io/en/latest/>
- [6] Pagina oficial de OpenCV - <http://opencv.org/>
- [7] Aplicaciones de Parrot - <http://global.parrot.com/mx/apps/>
- [8] Adaptador *Wifi* externo - <http://www.tp-link.com/ar/products/details/TL-WN722N.html>

Universidad de Alcalá
Escuela Politécnica Superior



ESCUELA POLITECNICA
SUPERIOR



Universidad
de Alcalá