Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Navegación autónoma mediante raspberry PI y la plataforma OpenCV

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Alejandro Fuster Baggetto

Tutor: Ángel Rodas Jordá

2016-2017

Quiero agradecer especialmente:

A Ángel Rodas Jordá, tutor de este trabajo, por su tiempo, comprensión y dedicación hacia mi en cada una de las etapas del proyecto.

A mi hermana Paula y mi sobrina Bulldi por respetar el espacio de trabajo que teníamos compartido.

Y por supuesto a mis padres que han sido, como siempre, un apoyo constante e incondicional.

Resumen

Se utilizan la librería *OpenCV* y el lenguaje *Python* para desarrollar distintos algoritmos de guiado sobre un robot móvil autónomo de fabricación propia, que incluye una cámara conectada a una *Raspberry PI*. Dichos algoritmos tienen como objetivo resolver cinco pruebas mediante la extracción de parámetros de guiado a partir del procesamiento de la realimentación visual que ofrece la cámara, así como usar estos parámetros para controlar el robot. Cada una de las pruebas se acompaña con una demostración de los resultados en forma de vídeo.

Palabras clave: Raspberry PI, OpenCV, visión artificial, robot móvil, Python.

Resum

S'utilitzen la llibrería *OpenCV* i el llenguatge *Python* per a desenvolupar diferents algoritmes de guiat sobre un robot mòbil autònom de fabricació pròpia, que inclou una càmera connectada a una *Raspberry PI*. Aquests algoritmes tenen com a objetiu resoldre cinc proves mitjançant la extracció de paràmetres de guiat a partir del procesament de la realimentació visual que ofereix la càmera, així com utilitzar aquestos paràmetres per a controlar el robot. Cadascuna de les proves s'acompanya amb una demostració dels resultats en forma de vídeo.

Praules calu: Raspberry PI, OpenCV, visió artificial, robot mòbil, Python.

Abstract

We use the *OpenCV* library and the *Python* language in order to develop different guiding algorithms on a mobile autonomous own-manufactured robot, which includes a camera connected to a *Raspberry PI*. The purpose of these algorithms is to solve five challenges through the extraction of guiding parameters from the analysis of the visual feedback offered by the camera, as well as use these parameters to control the robot. The results of each of the challenges are accompanied by a vídeo demo.

Keywords: Raspberry PI, OpenCV, artificial vision, mobile robot, Python.

Tabla de contenidos

1. Introducción

1.1 Motivación

Como estudiante de Ingeniería Informática en la rama de Computación, he adquirido conocimientos sobre diversas técnicas de inteligencia artificial y reconocimiento de formas. Desde que empecé a estudiar aprendizaje automático, esta materia me resultó muy interesante hasta el punto de considerarla esencial en la informática actual. Sin embargo, el aprendizaje automático depende directamente de los resultados obtenidos por la fase de extracción de características, la cual se imparte solo durante la primera mitad de la asignatura de Percepción. Además, de ese contenido, únicamente una unidad trata las imágenes y lo hace desde una perspectiva teórica. Tras cursar esa asignatura, empecé a sentir curiosidad por aplicar lo que había aprendido al reconocimiento de imágenes y descubrí que *OpenCV* era la herramienta más adecuada para ello.

Durante ese periodo, me presenté a un concurso llamado ORC (Olympic Robotic Challenge) organizado por la asociación Makers UPV. Éste consistía en construir un robot que se enfrentara a diferentes pruebas de la mejor forma posible. Leyendo las normas del concurso vi que estaba totalmente prohibido el uso de cualquier tipo de visión artificial, de lo cual deduje que el motivo de esa norma era que ésta confería al robot una enorme ventaja sobre el resto, desequilibrando así la competición. Ésto aumentó mi ya existente curiosidad por descubrir el potencial de la visión artificial y aplicarlo a un robot.

Por otro lado, tenía cierta experiencia con *Arduino*, pero algunos compañeros me habían recomendado probar *Raspberry*, hablándome muy bien de este dispositivo y de la cantidad de usos que se le pueden dar.

Por estos motivos, cuando vi un proyecto que proponía utilizar la librería *OpenCV* para ofrecer visión artificial a un robot basado en *Raspberry*, decidí, sin duda alguna, que ese sería mi trabajo de fin de grado. Supe desde el principio que sería un gran reto y que tendría que adquirir una cantidad enorme de conocimientos, pero esto en lugar de suponer un obstáculo, fue lo que me hizo decantarme por él. Era una oportunidad de aprender cosas nuevas, que además considero muy útiles y que me servirían para completar mi formación como informático. No solo eso, sino que además valoré mucho que, en este proyecto, tendría la ocasión de aplicar en mayor o menor medida diversos aspectos de la informática.

En el presente trabajo se han utilizado conocimientos de mecánica y electrónica para la construcción del robot, de redes para la comunicación entre agentes, de computación paralela para la ejecución de varios procesos simultáneamente, de *Python* y *Arduino* para toda la parte de programación y cómo no, de *OpenCV* para la visión artificial.

1.2 Objetivos

Una vez explicada la motivación para realizar el presente trabajo pasamos a definir los objetivos.

- Construir y desarrollar un sistema completo que conste de un *host* y un robot móvil intercomunicados. El *host* supervisará al robot, que incluirá necesariamente una *Raspberry Pi* con cámara para la ejecución de algoritmos de visión artificial mediante los que decidirá su rumbo. Además, la estructura del sistema debe ser flexible y ampliable sin necesidad de modificar nada más que la parte reemplazada y representar una plataforma genérica sobre la que se puedan desarrollar trabajos futuros.

- Plantear cinco retos de guiado diferentes, así como sus resoluciones por parte del sistema sin más sensorización que una cámara. Con estos retos se pretende resaltar el potencial de la visión artificial como alternativa para la recopilación y análisis de datos en los sistemas físicos, a la vez que se ponen en práctica los conocimientos adquiridos sobre la librería *OpenCV*. Concretamente, se pretende que las soluciones a los retos cubran lo mejor posible las utilidades de *OpenCV*, es decir, que nos permitan tener una amplia perspectiva sobre esta librería.

- Estudiar la herramienta *OpenCV* debido a que es la librería de código abierto más utilizada en ámbitos como la visión artificial y el procesamiento de imágenes. Se busca aprender su utilización y comprender su alcance para poder sacarle el máximo partido.

1.3 Estructura de la memoria

El contenido de este trabajo se puede dividir semánticamente en dos partes fundamentales.

- Una primera parte, que comprendería los capítulos del dos al cinco, trata de dar una visión completa del sistema. En ella se presentan los diferentes agentes de los que consta, las comunicaciones entre ellos, la construcción del robot, la instalación de las dependencias, la estructura del software y el control de motores. Con todo ello, se pretende dar al lector una comprensión de la totalidad del sistema, lo cual es muy importante, pues los contenidos explicados en esta parte se obvian durante la siguiente.

- Una segunda parte, que comprendería los capítulos del seis al diez, plantea y resuelve las distintas pruebas (o retos) que se han elegido para el sistema. Cada una de las cinco pruebas consta de un capítulo individual en el que se profundiza en los problemas experimentados y las soluciones halladas. Además, se ha grabado y subido a *Youtube* un vídeo de cada prueba, cuyos enlaces se encuentran al final de los capítulos correspondientes.

2. Herramientas e instalación

Antes de entrar en los aspectos de *hardware* y *software* que conforman el proyecto, es oportuno ofrecer una visión general del sistema, de forma que se entiendan mejor los puntos posteriores.

2.1 Agentes del sistema

Como se puede apreciar en la Figura 2.1, el sistema consta de tres agentes: La *Raspberry* (con la cámara) , el *Arduino* (con los actuadores) y el *Host*.

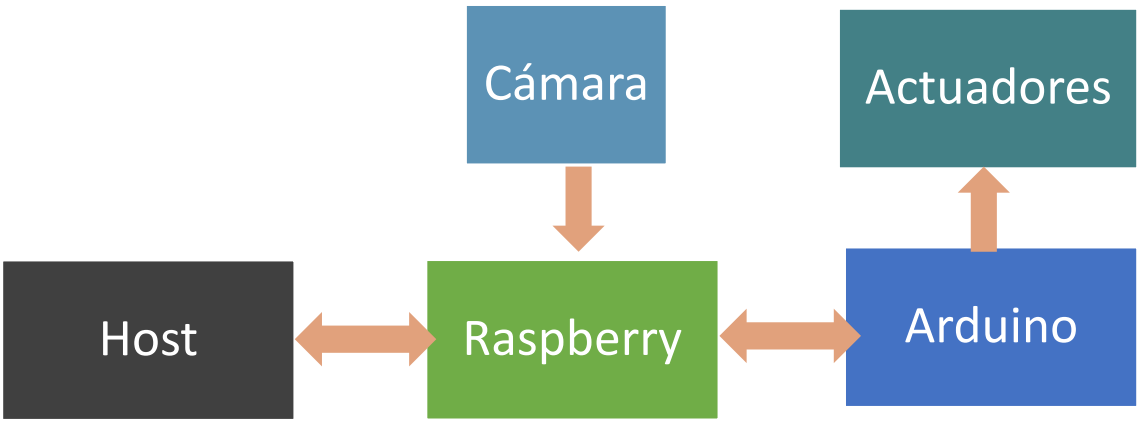


Figura 2.1 Los agentes del sistema

2.1.1 La *Raspberry*

El programa ejecutado por la *Raspberry* es capaz de recibir órdenes del usuario a la vez que, gracias a la cámara que lleva conectada, capta imágenes, las procesa y da las órdenes pertinentes al *Arduino*, indicándole cómo mover los motores. La *Raspberry* es el cerebro del sistema, (también la llamamos agente controlador) teniendo además conectada la cámara, que son los ojos del sistema.

Python 2.7

El programa de la *Raspberry* esta escrito en *Python 2.7*. Se eligió este lenguaje porque, pese a no ser tan eficiente como *c++*, por ejemplo, presenta innumerables ventajas sobre otros lenguajes de programación:

- Es muy compacto: Con relativamente pocas lineas de código pueden construirse algoritmos complejos. La Figura 2.2 ejemplifica este hecho comparando, para un programa sencillo, su código fuente en *Python* y en *Java*.

- Es muy flexible: Tiene rasgos del paradigma imperativo y del funcional, es un lenguaje orientado a objetos, soporta *multithreading* y *multiproceso*, etc.

- Tiene un manejo muy cómodo de las estructuras de datos (como las listas, las tuplas, los diccionarios,...) y una sintaxis muy ergonómica en general. En la Figura 2.2 vemos cómo hemos inicializado una sublista en una sola linea.

No se ha escogido *Python 3* porque, aunque existen adaptaciones de *OpenCV* para este lenguaje, ambos no son compatibles de base. Por este motivo se descartó utilizarlo para evitar futuros problemas.

```python
texto='Esta es una frase de prueba'
print(' '.join([w for w in texto.split(' ')if 'a' in w]))
```

```java
public class Ejemplo{
    public static void main(String[] args){
        String texto = "Esta es una frase de prueba";
        String[] palabras = texto.split(" ");
        texto="";
        for(int i=0;i<palabras.length;i++)
            if(palabras[i].contains("a"))
                texto += palabras[i]+" ";
        System.out.println(texto);
    }
}
```

Figura 2.2 Ejemplo de código en Python y Java

OpenCV

Para la visión artificial se ha escogido *OpenCV* porque es una de las bibliotecas más completas y utilizadas en este campo. No es una simple librería para el reconocimiento de imágenes. Incluye un amplio abanico de funciones en el ámbito gráfico que van desde el procesamiento de imágenes hasta el seguimiento de objetos en tiempo real, pasando por algoritmos de detección de esquinas, de eliminación de ruido, de aprendizaje automático, etc. Otro motivo que nos ha impulsado a usar *OpenCV* es que está disponible y bien optimizada para *Python*.

2.1.2 El *Arduino*

Es el agente encargado de controlar el movimiento del robot, limitándose a ejecutar las órdenes que recibe de la *Raspberry*. Por ello lo llamaremos ejecutor. Está conectado a través de sus pines digitales a diferentes actuadores (motores y servo) a los que manda las señales pertinentes. Utiliza lenguaje *Arduino*, que es el único compatible con este microcontrolador y que además es prácticamente idéntico al lenguaje *C*.

2.1.3 El *Host*

Desde su ordenador, el programador puede, mediante una aplicación de escritorio, mandar órdenes a la *Raspberry* para que las ejecute o se las reenvíe al *Arduino*.

Aunque se trate de un robot autónomo, es necesario que el usuario tenga control sobre el mismo, por ejemplo, para ocasiones en las que se quiera mover el robot a voluntad, hacer paradas de emergencia o utilizar directamente algunas funcionalidades como las de hacer fotos o vídeos.

En definitiva, el *Host* sirve para supervisar y controlar el sistema. Por este motivo lo llamaremos supervisor. Nótese que este agente no constaría únicamente del programa ejecutado en el *Host*, sino también del usuario físico, que es, en este caso, quien toma las decisiones. El programa está también escrito en *Python 2.7* por motivos similares

a los expuestos anteriormente, no obstante, no habría problemas de compatibilidad con el programa de la *Raspberry* aunque para el de usuario se utilizara un lenguaje distinto o se ejecutara en otra plataforma o sistema operativo. Lo único que se necesita del lenguaje es que sea capaz de conectarse vía *socket*. Así que, en lugar de una aplicación de escritorio en *Python*, podríamos haber desarrollado, por ejemplo, una aplicación móvil en *Android*, que ofrecería ,probablemente, una forma más cómoda de control para el usuario externo al proyecto.

2.2 Instalación de *OpenCV*

La instalación de la librería *OpenCV* en la *Raspberry* es un proceso complejo, así que vamos a hacer un breve recorrido por los distintos pasos de la misma:

1) **Expansión del sistema de archivos**

   Se trata de modificar la configuración de la *Raspberry* para tener acceso a todo el espacio de la tarjeta *SD* (en nuestro caso 16GB). Es un paso opcional, pero deseable, ya que, de no hacerlo, esta instalación dejaría muy poco espacio disponible.

2) **Instalación de dependencias**

   En primer lugar, se descargan algunas herramientas de desarrollador, como *Cmake*, que hacen posible la compilación de *OpenCV* (paso cinco). A continuación, se instalan librerías que son necesarias como complemento a *OpenCV*. En este grupo se encontrarían las encargadas de leer y escribir imágenes y vídeos en distintos formatos, o las que se utilizan para crear ventanas e interfaces de usuario con *OpenCV*. Por último están las dependencias que no son tan obligatorias, sino que optimizan algunas funciones de *OpenCV*.

3) **Descarga de *OpenCV***

   Es necesario descargar el código fuente de *OpenCV 3.1.0* desde el repositorio https://github.com/Itseez/*OpenCV*/archive/3.1.0.zip. Se debe descargar también un segundo código llamado *OpenCV-contrib*, que completa el primero,https://github.com/Itseez/*OpenCV*_contrib/archive/3.1.0.zip.

4) **Creación de un entorno virtual**

   Un entorno virtual es una herramienta cuyo cometido es mantener separadas las dependencias de los distintos proyectos. Está considerado en *Raspberry* como buenas prácticas el hecho de crear un entorno virtual por cada proyecto. Puede ocurrir que un proyecto requiera una librería, mientras que otro use una versión distinta, de forma que ambas versiones colisionen. Es de este problema del que precisamente nos previenen los entornos virtuales aislando los proyectos entre si. A pesar de ser muy recomendable, no es estrictamente necesario para la instalación de *OpenCV*.

   Al ser la primera vez que se utiliza en esta *Raspberry*, hay que instalar las herramientas *Virtualenv* y *Virtualenvwrapper*. Posteriormente creamos nuestro entorno virtual de *Python*. Una vez creado, podremos saber que nos encontarmos en él si vemos su nombre entre paréntesis al principio de cada linea de la terminal . Cada vez que abrimos una terminal nueva y queremos entrar en nuestro entorno, tenemos que escribir source *~/.profile* y *workon cv* (donde *cv* es el nombre del entorno virtual).

Por último, en este paso se instala ,ya en el entorno virtual, una dependencia más para *OpenCV*: *Numpy*, la cual es una librería de matemáticas y cálculo matricial muy útil para *Python* e imprescindible para *OpenCV*. Entre otras cosas, las imágenes utilizadas por *OpenCV* son internamente matrices *Numpy*.

5) Compilación de *OpenCV*

En primer lugar, se prepara la compilación con *Cmake* y posteriormente se usa *Make* para compilar. Recordemos que este paso de compilación ha de realizarse porque lo que tenemos descargado es el código fuente de *OpenCV*. Cuando se descarga *OpenCV* para *Windows* o *Ubuntu*, ya está precompilado. Para otros sistemas operativos menos comunes ,como *Raspbian* (una variante de *Debian*), que es el sistema operativo de *Raspberry*, no tenemos esta opción. Así que debemos compilar nosotros la librería.

Este proceso dura casi una hora y media, aunque finaliza antes si se indica la opción de utilizar los cuatro núcleos en paralelo. La paralelización evidentemente acelera la compilación, pero también la pone en riesgo ante posibles condiciones de carrera.

Tras la instalación, símplemente queda enlazar mediante un link simbólico el código compilado de *OpenCV* generado en el paso anterior con el entorno virtual.

3. Construcción del robot

En esta sección pasamos a explicar los objetivos que han guíado el ensamblaje del robot, así como los resultados obtenidos.

En primer lugar, vamos a enumerar las dos características principales requeridas en el robot, las cuales justifican muchas de las decisiones tomadas en relación a la construcción del mismo:

Sencillez:

- Un robot sencillo surge de un proceso de construcción simple, lo cual ha resultado muy conveniente debido a la escasez de materiales, herramientas y tiempo.
- Un robot sencillo es fácil de reparar. Si falla físicamente alguna pieza, un diseño simple permite encontrar y reemplazar la pieza causante rápidamente sin afectar al resto del robot.

Funcionalidad:

- Es imprescindible que el robot esté preparado para afrontar lo mejor posible las funcionalidades planeadas para él.

3.1 Especificaciones del robot

Tras esta breve mención a nuestras dos ideas fundamentales, vamos a concretar más las especificaciones del robot.

3.1.1 Disposición de las partes

Se trata de un robot móvil diferencial de fabricación propia que combina la potencia de una *Raspberry PI 3B* con la versatilidad de un *Arduino UNO*.

Desde el inicio de este proyecto, se ha decidido que tanto la captación (cámara) como el procesamiento (*Raspberry*) de las imágenes se encuentren empotrados en el robot. Es decir, éste no depende de ningún agente externo más que del agente supervisor, siendo completamente autónomo. Había otras opciones de configuración en este sentido que no ofrecían ninguna ventaja para este proyecto y planteaban distintos problemas, algunos de los cuales se consideran a continuación:

Configuraciones con agente controlador (*Raspberry*) externo al robot:

La comunicación entre la *Raspberry* y el *Arduino* tendría que hacerse de forma inalámbrica. En este caso lo más conveniente sería utilizar *Bluetooth*, aunque se necesitaría un *shield* adicional para el *Arduino*.

Configuraciones con cámara externa:

La cámara estaría fija y tendría un determinado rango de visión. Cuando el robot se alejara demasiado o tuviera que percibir algo fuera de este rango, fallaría. Además, una cámara externa añade la necesidad de transformaciones matemáticas para adecuar la percepción de la cámara a la perspectiva del robot

Configuraciones con cámara y agente controlador (*Raspberry*) separados (Cámara empotrada, *Raspberry* externa/ Cámara externa, *Raspberry* empotrada):

La cámara específica para *Raspberry* debe ir obligatoriamente conectada a la misma. Incluso en el supuesto de que se utilizara otra cámara, habría que transmitir por *Wifi,* vídeo en tiempo real desde la cámara a la *Raspberry*, lo que no solo representa un reto, sino que además, la comunicación añadiría un inevitable lastre a la frecuencia de muestreo.

3.1.2 Utilización de *Arduino*.

Procede hacer una breve puntualización de por qué se ha decidido utilizar *Arduino* en el sistema.

Se podría pensar que el *Arduino* no es necesario, ya que la *Raspberry* tiene salidas digitales y es perfectamente capaz de controlar los motores y el servo. En efecto, el *Arduino* no era necesario, pero se considera una mejora:

- El *Arduino* ha permitido disponer en capas separadas la *Raspberry* y los actuadores, haciéndolos independientes entre sí. Cualquier cambio en los actuadores no afectará a la *Raspberry* y cualquier cambio en la *Raspberry* no afectará al *Arduino* ni a los actuadores. Por ello, el *Arduino* ha sido un importante añadido que ha contribuido a conseguir la flexibilidad del sistema de la que se ha hablado en los objetivos.

- *Arduino* utiliza una estrategia completamente distinta a *Raspberry*. Tiene muchísima menos potencia de cómputo. Sin embargo su programación a bajo nivel (lenguaje *C*) le permite una fácil interacción con todo tipo de componentes. Este hecho combinado con la gran cantidad de *shields* que existen para *Arduino*, lo convierten en una herramienta muy versátil y adaptable.

- La comunicación *Raspberry-Arduino* (por serial) supone un valor añadido para el proyecto y es por supuesto reutilizable en cualquier ámbito en el que deseen combinarse estos dos dispositivos. Además, la presencia del *Arduino* apenas añade peso, coste o complicación al robot.

Por estos motivos se ha tomado el *Arduino* como brazo ejecutor (conectado directamente a los actuadores) mientras que se ha mantenido a la *Raspberry* como cerebro (conectada sólo al *Arduino* y a la cámara)

3.2 Diseño del robot

Tras este breve inciso, vamos a ahondar en el diseño del robot. Se han construido partiendo de cero dos robots (un prototipo experimental y el robot final). Ambos tienen en común los **componentes fundamentales** (Figura 3.1)

- *Raspberry* Pi 3B
- Cámara Pi V2
- *Arduino* UNO
- Controlador L298n
- 2X Motor DC
- Jumper wires
- Cables puente H
- Batería 7.5 V , 650 mAh
- Cable *serial* de *Arduino*
- *Switch*
- Soldadura de estaño
- Bridas
- 2X Rueda motriz
- Rueda fija

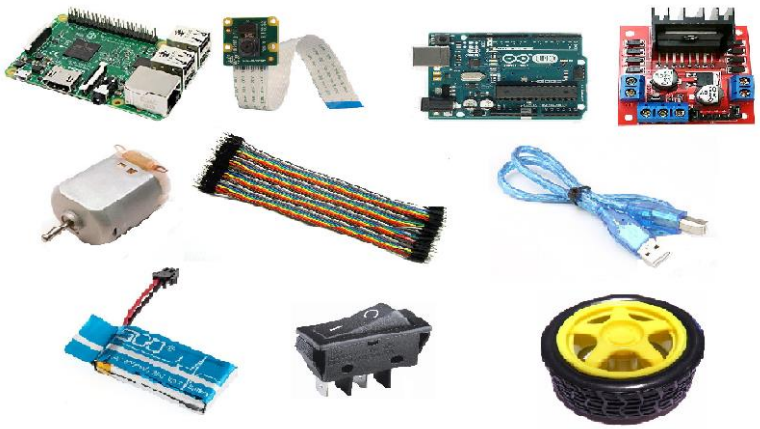


Figura 3.1 Componentes básicos del robot

Conexión del controlador de motores

La única conexión que podría necesitar explicación es la del *Arduino* con el controlador l298n.

Este controlador tiene dos señales de habilitación o *Enable signals*, cuatro entradas y cuatro salidas. Cada motor de corriente continua utiliza dos salidas, que el controlador obtiene a partir de dos entradas y una señal de habilitación. Las entradas solamente controlan el sentido del movimiento del motor (Tabla 3.1), mientras que la velocidad viene dada por el *enable*, que estará obligatoriamente conectado a uno de lo pines *PWM* (3,5,6,9,10,11) del *Arduino*. La distribución de pines de un *Arduino UNO* se muestra a continuación en la Figura 3.2

IN1	IN2	MOTOR
0	0	Parado
0	1	Se mueve en un sentido
1	0	Se mueve en sentido contrario
1	1	Parado (no recomendable)

Tabla 3.1 Entradas y salidas del controlador de motores

3.2.1 Prototipo

Materiales propios

- Chasis de plástico
- Regulador de corriente
- Adaptador pines a micro usb
- *Mini board*
- *Buzzer*
- 2X Led amarillo
- 3X Resistencia

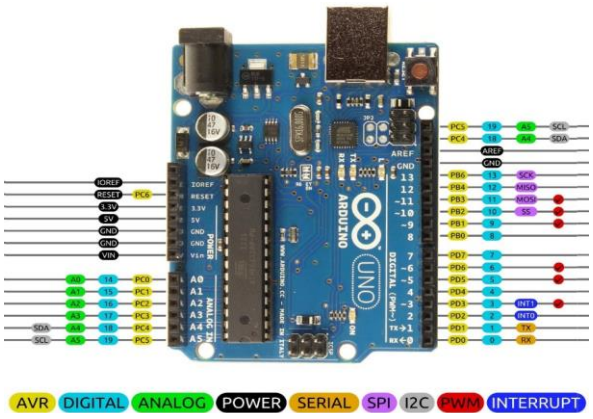


Figura 3.2 Distribución de pines de Arduino UNO

Probemas del prototipo

- La cámara estaba en un ángulo fijo, lo cual no se adaptaba a las necesidades de las pruebas, que toman imágenes desde distintos ángulos en el plano XY (distintas alturas).

- La batería duraba muy poco. En este robot una sola batería de 650 mAh tenía que alimentar el sistema completo. Sólamente la *Raspberry* puede consumir más de 1A. No es de extrañar que la batería apenas durase veinte minutos.

- Debido al diseño y a la falta de espacio, el cable serial pasaba por delante de la cámara, cuya visión se veía nublada tanto por el cable como por la sombra que proyectaba el mismo.

- Carecía de sencillez por culpa de añadidos que no aportaban nada al proyecto. Una maraña de cables resistencias y leds y un *buzzer* es lo que podemos hallar en la *mini board* presente en este robot.

- La *Raspberry* se reiniciaba o congelaba asiduamente por problemas de alimentación, probablemente debido a la deficiencia en alguna de las soldaduras con el regulador de corriente.

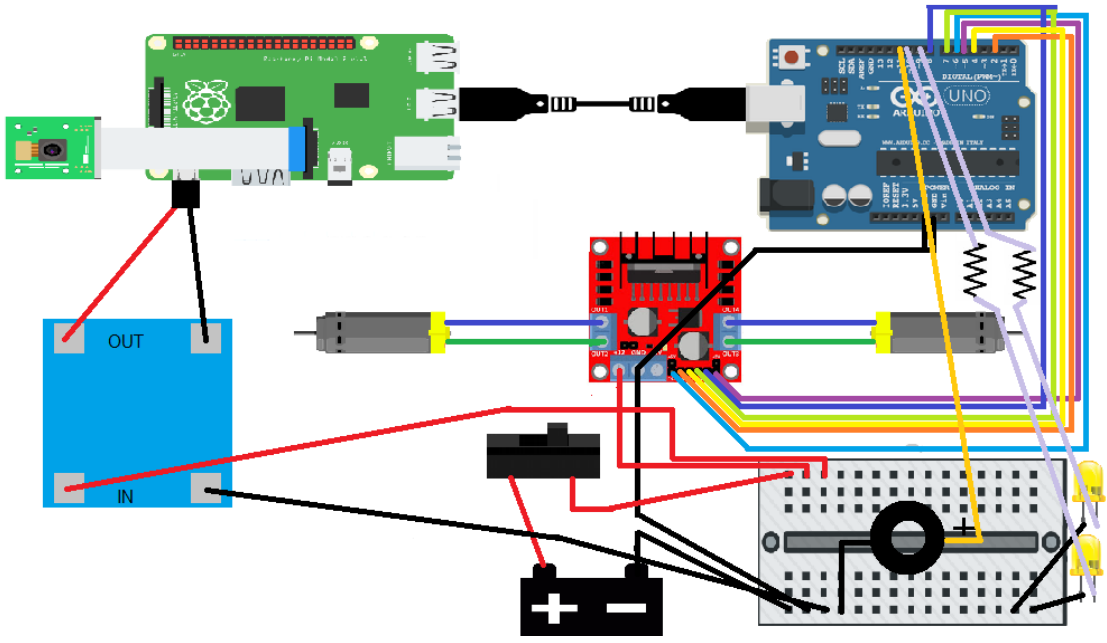


Figura 3.3 Esquema electrónico del prototipo

Todos estos problemas nos llevaron a construir otro robot, que fue el que finalmente se utilizó para el proyecto.

3.2.2 **Robot final** (Figura 3.4)

Materiales propios

- Mini servo
- Chasis de madera
- Batería Amazon 5600 mAh
- Cable usb-micro usb

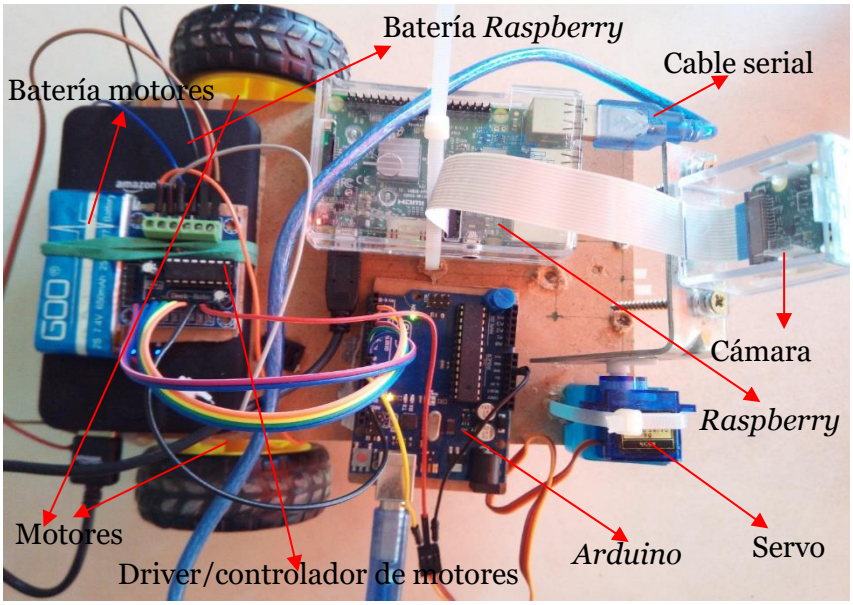


Figura 3.4 El robot final

Mejoras que presenta el robot

Este robot se hizo con la idea de solucionar los problemas que surgieron en el anterior:

- Gracias al servo, podemos cambiar el ángulo de la cámara en el plano *XY* según sea necesario. El servo tiene un rango de 180 º, así que podemos tener la cámara mirando hacia arriba, hacia abajo o a cualquier dirección entre estas dos.

- Como solución al problema de la batería, se decidió utilizar dos. La antigua, con una capacidad de 650 mAh, alimenta ahora sólo el controlador de motores (y por tanto los motores) y el servo. La batería nueva, con una capacidad de 5600 mAh, se encarga de alimentar la *Raspberry*, que a su vez alimenta al *Arduino* por el puerto serie.

- En este robot, ambas baterías duran varias horas, lo cual permite el desarrollo de código directamente en la *Raspberry*.

- El cable serial se ha pasado por detrás de forma que no suponga una molestia para la cámara

- Se han eliminado los añadidos innecesarios que perjudican la simplicidad del robot sin añadir funcionalidad alguna.

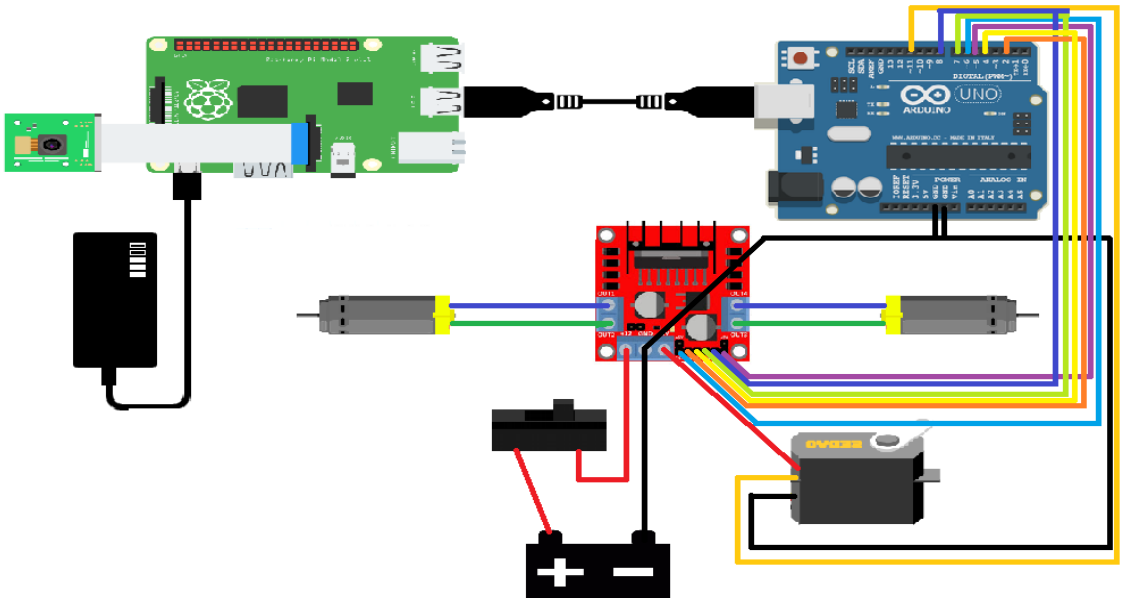


Figura 3.5 Esquema electrónico del robot 2

El resultado de este proceso es un robot apto para el desarrollo del presente trabajo de fin de grado.

4. Infraestructura y comunicaciones

4.1 Estructura Física de las comunicaciones

Como se muestra en la Figura 4.1, conectar los tres agentes del sistema, ha supuesto establecer dos comunicaciones.

Comunicación entre *Raspberry* y *Arduino*

Se utiliza el cable serial del *Arduino* conectado a un puerto *usb* de la *Raspberry*. Este cable alimenta el *Arduino* y permite la transmisión de datos en ambos sentidos. En nuestro caso, la comunicación es unidireccional, ya que solo la *Raspberry* envía datos. El *Arduino* podría sin ningún problema ni modificación adicional enviar datos a la *Raspberry* si fuera necesario, por ejemplo para el control de errores.

La comunicación se podría haber hecho por *Bluetooth*, pero como ya se ha dicho, haría falta un *shield* adicional para *Arduino*. Además, el cable presenta la ventaja de que a la vez alimenta el *Arduino*

También podríamos haber realizado la comunicación mediante *rs-232*, utilizando los puertos *RX* y *TX* (0 y 1) del *Arduino*. Sin embargo, esta alternativa no presenta ninguna ventaja adicional al cable serial, el cual además incluye alimentación.

Comunicación entre *Host* y *Raspberry*

La comunicación del *Host* (en un ordenador personal) y la *Raspberry*, se efectúa vía *Wifi* siguiendo una arquitectura cliente(ordenador) servidor(*Raspberry*). Ésta comunicación también es unidireccional, ya que la *Raspberry* no envía datos al *Host*, aunque también podría responder al *Host* a través de este canal si fuera necesario. El tema de la realimentación se tratará posteriormente.

Esta comunicación también se podría haber hecho por *Bluetooth*, sin embargo, existiría un límite de distancia entre el ordenador y la *Raspberry*, que obviamente no existe con red.

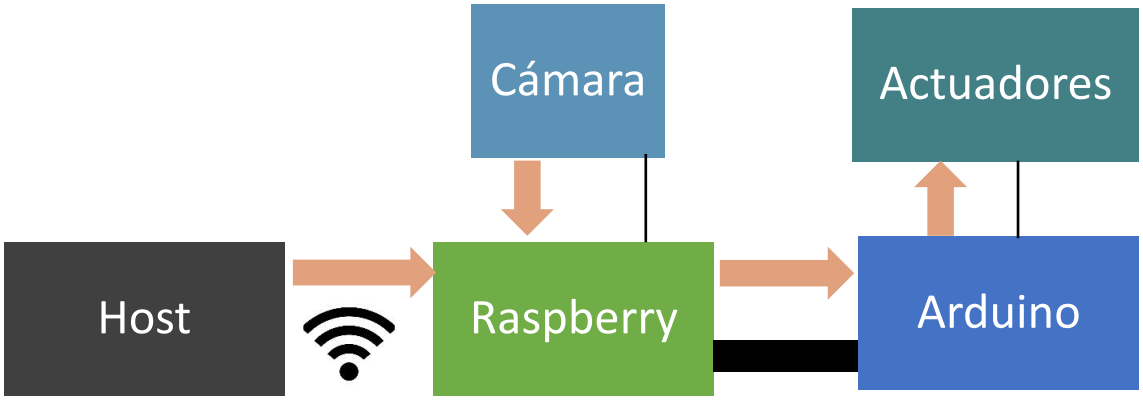


Figura 4.1 Comunicación entre los agentes del sistema

4.2 Protocolos y órdenes

Teniendo clara la estructura física de las comunicaciones, en este apartado se trata la parte *software* de las mismas.

4.2.1 Tipos de órdenes

La *Raspberry*, como servidor, acepta la conexión del cliente. Las órdenes que ésta puede recibir o enviar están predefinidas y se pueden agrupar en tres conjuntos según el agente al que se dirigen:

Tipo 1: Órdenes dirigidas al *Arduino*

Son órdenes sin significado para la *Raspberry*. Cuando el *Host* las envía a la *Raspberry*, su intención es que ésta las reenvíe al *Arduino*. Además, dichas órdenes no solo las genera el cliente, sino también la *Raspberry*, siendo éstas la salida de la mayoría de los algoritmos de visión desarrollados en el proyecto y las relacionadas con el movimiento de los motores o del servo y, por lo tanto, las detonantes de la navegación autónoma del vehículo.

Tipo 2: Órdenes dirigidas a la *Raspberry*

Son órdenes sin significado para el *Arduino*. Cuando el cliente las envía a la *Raspberry*, su intención es que ésta actúe en consecuencia. Ejemplos de estas órdenes son las de hacer fotos o vídeos o las que modifican el comportamiento del programa de la *Raspberry*.

Tipo 3: Órdenes dirigidas a *Raspberry* y *Arduino*

Son órdenes entendibles por ambos. Cuando el cliente las envía a la *Raspberry*, su intención es que ésta las reenvíe al *Arduino* y que además las ejecute. Un ejemplo sería la de finalizar el programa en ambos dispositivos.

4.2.2 Protocolos de comunicación

Cuando la *Raspberry* recibe una orden

La reenvía al *Arduino* inmediatamente independientemente de su tipo. De esta forma evitamos retardos en la respuesta del *Arduino* (en caso de que la orden vaya dirigida a él). Tras reenviarla, la *Raspberry* examina la orden para ver si va dirigida a ella y en caso afirmativo, actuar en consecuencia. Se ahondará en ésto posteriormente cuando entremos en la estructura del programa ejecutándose en la *Raspberry*.

Las órdenes que recibe el *Arduino*

Pueden haber sido emitidas en su origen por el usuario o por la *Raspberry*. Sin embargo, no están etiquetadas y, por lo tanto, el *Arduino* no conoce su procedencia. Por ejemplo, si recibe la orden de seguir recto, no sabe si ha sido idea de la *Raspberry* que, basándose en su visión, ha tomado esa decisión o si ha sido el usuario el que lo ha decidido, mandándola él mismo a la *Raspberry* para que ésta la reenvíe.

Formato de las órdenes

Por simplicidad y cohesión, los tres agentes comparten un único formato de órdenes.

Nombre:; (si no hay parámetros)

Nombre:param1,param2,...paramn; (si hay parámetros)

Por ejemplo: *recto:;* hará que el robot avance con las velocidades

por defecto, mientras que *recto:120,100;* hará que el robot avance con los parámetros especificados como velocidades de las ruedas izquierda y derecha respectivamente.

La siguiente tabla es una lista completa de las órdenes registradas.

Nombre	**Tipo**	**Descripción**	**Paráme tros**	**Significado de los parámetros**
recto	1	Avanza hacia delante	2	Velocidades de las ruedas
izq	1	Gira a la izquierda (rueda izquierda parada)	1	Velocidad de la rueda derecha
der	1	Gira a la derecha (rueda derecha parada)	1	Velocidad de la rueda izquierda
izq2	1	Gira a la izquierda sobre su propio eje (rueda izquierda hacia atrás)	1	Velocidad de las ruedas (derecha en positivo, izquierda en negativo)
der2	1	Gira a la derecha sobre su propio eje (rueda derecha hacia atrás)	1	Velocidad de las ruedas (izquierda en positivo, derecha en negativo)
para	1	Detiene el robot	0	
l	1	Aumenta la velocidad de la rueda izquierda	0	
r	1	Aumenta la velocidad de la rueda derecha	0	
atras	1	Avanza marcha atrás	2	Velocidades de las ruedas (ambas en negativo)
reclinar	1	Reclina la cámara	1	Decremento del ángulo de inclinación
inclinar	1	Inclina la cámara	1	Incremento del ángulo de inclinación
servo	1	Mueve la cámara al ángulo deseado	1	Ángulo de inclinación
p	1	Detiene el robot y hace que el *Arduino* ignore todas las órdenes que le lleguen a partir de ese momento excepto *escuchar* y *reset*	0	
escuchar	1	Hace que el *Arduino* atienda todas las órdenes que le lleguen a partir de ese momento	0	

Nombre	Tipo	Descripción	paráme tros	Significado de los parámetros
foto	2	Hace una captura del fotograma actual y lo almacena con formato *png*	1	Nombre de la foto
vídeo	2	Inicia/termina un vídeo. Cuando acaba se almacena con formato *avi*	1	Nombre del vídeo
realimen tación	2	Inicia/termina la realimentación visual. Cuando está activa, la *Raspberry* muestra lo que ve la cámara. En ocasiones conviene desactivarlo por eficiencia	0	
mode	2	Cambia al modo especificado. El modo activo marcará el comportamiento de la *Raspberry* y depende de la prueba que se esté realizando	1	Número de modo. Existen modos del cero al cinco
reset	3	Cambia el modo a cero, detiene el robot y pone el servo en su ángulo por defecto	0	
exit	3	Cierra el cliente y el servidor, detiene el robot y pone el servo en su ángulo por defecto	0	

Tipos de giro

Hay que destacar la diferencia entre los giros producidos por las órdenes *izq* y *der* (tipo 1) y los producidos por las órdenes *izq2* y *der2* (tipo 2).

- Los de tipo uno pivotan sobre una rueda. Es decir, mueven una rueda hacia delante mientras dejan la otra fija. Los giros de tipo uno son los que utilizamos para los algoritmos de control de motores.
- Los giros de tipo dos giran respecto a su eje. Es decir, mueven una rueda hacia delante mientras mueven la otra hacia atrás. Los giros de tipo dos los utilizamos concretamente en la prueba cuatro. Ambos tipos de giro hallan su explicación visual en la Figura 4.2, que es una representación sencilla del robot, tras realizar cada una de las maniobras de giro definidas partiendo desde un estado de reposo.

Es decir, los de tipo uno giran y avanzan mientras que los de tipo dos rotan sobre el sitio.

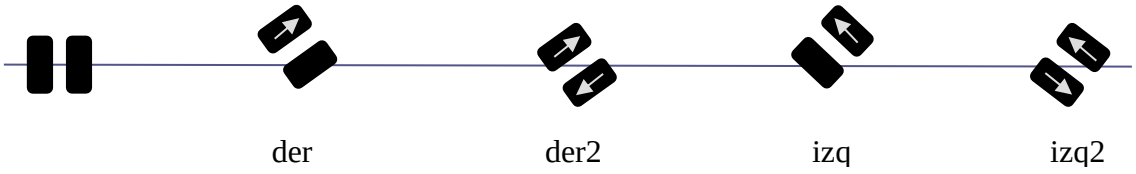


Figura 4.2 Tipos de giro

4.3 Estructura del software

Cuando el sistema está funcionando, hay tres programas ejecutándose a la vez, uno en cada uno de los tres agentes que conforman el sistema. Ya hemos explicado la comunicación entre dichos programas. En este subapartado vamos a profundizar en el funcionamiento, las dependencias y la estructura de cada uno de ellos.

4.3.1 Programa del *Host*

Dependencias

- *socket* para la conexión con el servidor
- *Tkinter* para la interfaz gráfica (solo en la versión gráfica)
- *sys* para finalizar el programa

Funcionamiento

Los pasos de este algoritmo están expresados, en forma de diagrama de flujo simplificado, en la Figura 4.3.

Lo primero que hace este programa es intentar conectarse mediante un *socket* a la *Raspberry* (puerto 8000). Cada vez que recibe una orden del usuario, la envía al servidor. Si es la de salir, el programa se cierra.

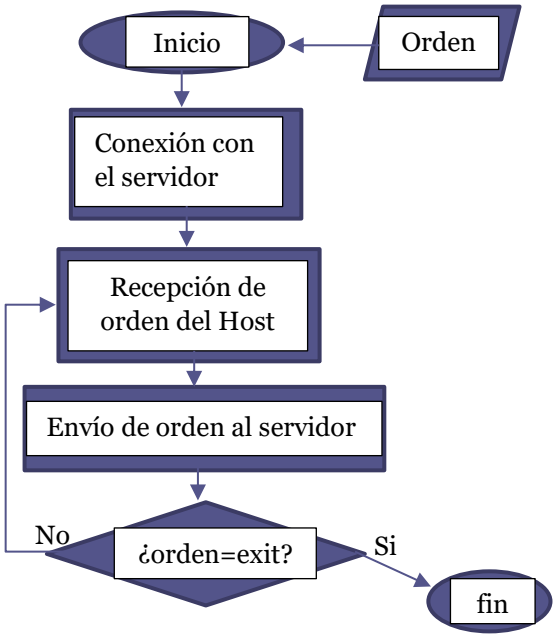


Figura 4.3 Flujo del host

Se han realizado dos versiones de este programa que difieren en la forma en la que el usuario introduce la orden:

Versión por linea de comandos

Las órdenes que envía son las que el usuario introduce por linea de comandos (con el formato especificado arriba).

Versión gráfica

El programa contiene una interfaz gráfica en *Tkinter* que consiste en una matriz de botones con diferentes iconos. Las órdenes que envía son las asociadas a los botones que pulsa el usuario.

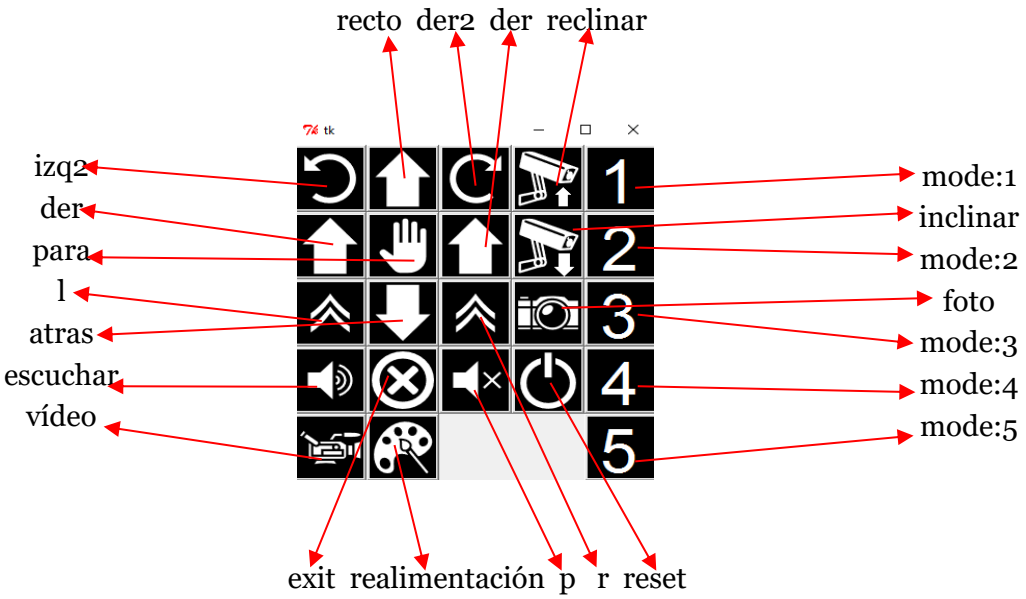


Figura 4.4 Interfaz gráfica del host

La interfaz así como las órdenes asociadas a los botones se encuentran en la Figura 4.4. Nótese que la interfaz no es nada atractiva ni está diseñada pensando en el usuario externo al proyecto. Se trata símplemente de un atajo para el programador, una alternativa más rápida a escribir los comandos cada vez. No obstante, si fuera necesario, se podría rediseñar la interfaz para acercarla más al usuario externo.

4.3.2 Programa de la *Raspberry*

El código se compone de dos procesos (presentes en el mismo código) que se comunican haciendo uso de colas. Se ha tenido que aplicar esta solución porque un servidor, que permanece ocioso a la espera de órdenes, no se puede combinar en el mismo hilo de ejecución con el programa principal, que requiere un bucle que tome imágenes continuamente a un determinado *framerate*. De combinarse, el servidor bloquearía el bucle hasta que recibiera órdenes.

Arquitectura Multiproceso vs multihilo

La solución más obvia a este problema era utilizar varios hilos. Sin embargo, se decidió en su lugar utilizar varios procesos por distintos motivos:

- Se experimentaron problemas a la hora de terminar los hilos en *Python* (para cerrar el programa)
- Los hilos se comunican mediante variables globales, lo cual requeriría protocolos de exclusión mutua y sincronización para evitar posibles condiciones de carrera inherentes a la concurrencia.

  Por contra, los procesos se comunican entre ellos usando una cola compartida cuya sincronización es ya transparente al programador. Esta comparativa se representa en la Figura 4.5.

Un aspecto general que, aunque no influye en este caso, hace que la concurrencia con hilos no se utilice en *Python* es el *GIL*(*Global Interpreter Lock*). *GIL* impide que se ejecuten varios hilos al mismo tiempo. Con esta limitación, los hilos apenas sirven para nada. En este caso, no hubiéramos tenido ese problema porque el servidor está casi todo el tiempo esperando órdenes y por lo tanto su hilo no estaría activo ocupando el *GIL*, sino que se hallaría en suspensión hasta recibir una orden. Se volverá a hablar del *GIL* más adelante.

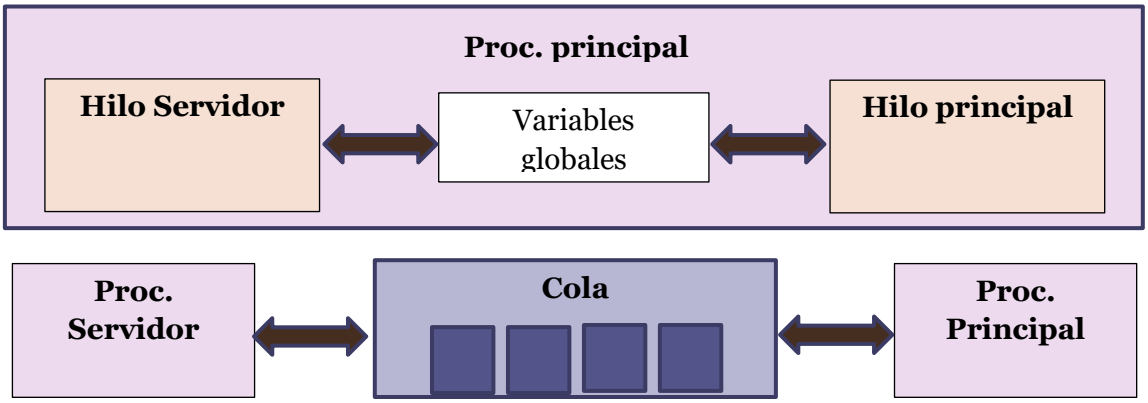


Figura 4.5 Comunicación entre hilos VS comunicación entre procesos

Dependencias

- *socket*: Se utiliza para crear el servidor, aceptar conexiones y leer órdenes del cliente.
- *select*: Se utiliza para gestionar las conexiones.
- *multiprocessing.Process*: Tipo de datos que define un proceso hijo.
- *multiprocessing.Queue*: Cola compartida que comunica el proceso padre con el hijo.
- *multiprocessing.Pool*: Sirve para distribuir varias tareas entre distintos procesos. Lo utilizaremos para paralelizar la prueba cinco.
- *functools.partial*: Nos permite crear funciones parciales a partir de otras funciones.
- *sys*: Para finalizar ambos procesos.
- *numpy*: Librería matemática para *Python*. Muy útil para el cálculo matricial y necesaria para *OpenCV*. Las imágenes con las que trabaja *OpenCV* son matrices *Numpy*.
- *cv2*: La librería de *OpenCV*, que se utiliza en todo el programa para procesar, reconocer y mostrar imágenes.
- *time*: Se utiliza para medir la eficiencia del programa (frecuencia de muestreo).
- *serial*: Se usa para la comunicación con el *Arduino*.
- *copy*: Sirve para copiar objetos. Concretamente nosotros la utilizamos para hacer copias de imágenes.
- *Picamera.array.PiRGBArray*: Se necesita para realizar las capturas con la cámara.
- *piCamera.PiCamera*: También necesaria para hacer las capturas. Permite cambiar los valores de la cámara(brillo, saturación ...) o añadir efectos (negativo, blanco y negro...).

Proceso Servidor

Proceso hijo creado por el proceso principal. Recibe órdenes del cliente que reenvía directamente al *Arduino* y añade a la cola compartida para que sean recogidas por el proceso principal.

Proceso principal

Crea el otro proceso (servidor) como proceso hijo. Contiene una inicialización previa de variables y después el bucle principal en el que se capturan las imágenes. El procesado y las posteriores órdenes enviadas al *Arduino* dependen del modo que esté activo. Por ejemplo, si hablamos del modo uno, la imagen tomada será convertida a escala de grises y umbralizada, de cuyo resultado dependerá el control de motores. Sin embargo, si hablamos del modo 4, se aplicarán algoritmos de detección de esquinas y se moverá el robot hasta centrar el objeto deseado.

Posteriormente, en caso de que esté activa la opción de realimentación, se imprimen las imágenes en una ventana para poder seguir el vídeo en tiempo real. Por último se recogen de la cola compartida las órdenes que ha recibido el servidor, se analizan y se aplican las acciones pertinentes (cambio de modo, guardar captura de cámara, etc). Tanto el funcionamiento de éste proceso como el del servidor descrito anteriormente, se encuentran esquematizados en el diagrama de flujo de la Figura 4.6.

Los modos

Existen seis modos de ejecución. Como puede apreciarse en la Figura 4.6, el modo activo es el que marca el código que se ejecuta durante cada iteración del bucle del proceso principal. Con el modo cero no se hace nada, mientras que con los otros cinco se ejecutan los códigos correspondientes a cada una de las pruebas. Ésto sirve para poder incluir todas las pruebas dentro del mismo código.

Al inicio del proyecto se decidió que se quería incluir todas las funcionalidades en el mismo programa, en lugar de tener un programa separado por cada prueba. La utilización de los modos es símplemente la opción más lógica para este propósito.

La frecuencia de muestreo

Un aspecto clave relacionado con este programa es la frecuencia de muestreo. Es un concepto de mucha importancia porque se va a mencionar en las distintas pruebas como un factor limitante a tener en cuenta al escribir cualquier algoritmo.

Llamamos frecuencia de muestreo al número de iteraciones por segundo que hace el bucle principal del programa de la *Raspberry*. Siempre es deseable conseguir una frecuencia de muestreo lo más alta posible, porque imaginemos lo que puede pasar si es baja., por ejemplo, de un ciclo por segundo (que es extremadamente baja). Eso querría decir que, dado que en cada ciclo se capta y procesa una imagen, solo se captaría una imagen cada segundo. Si, por ejemplo, el robot fuera recto, y tuviera que girar debido a una curva, tardaría un segundo en detectarla y por tanto en reaccionar. Aunque pudiera no parecer tan grave, un segundo es clave.

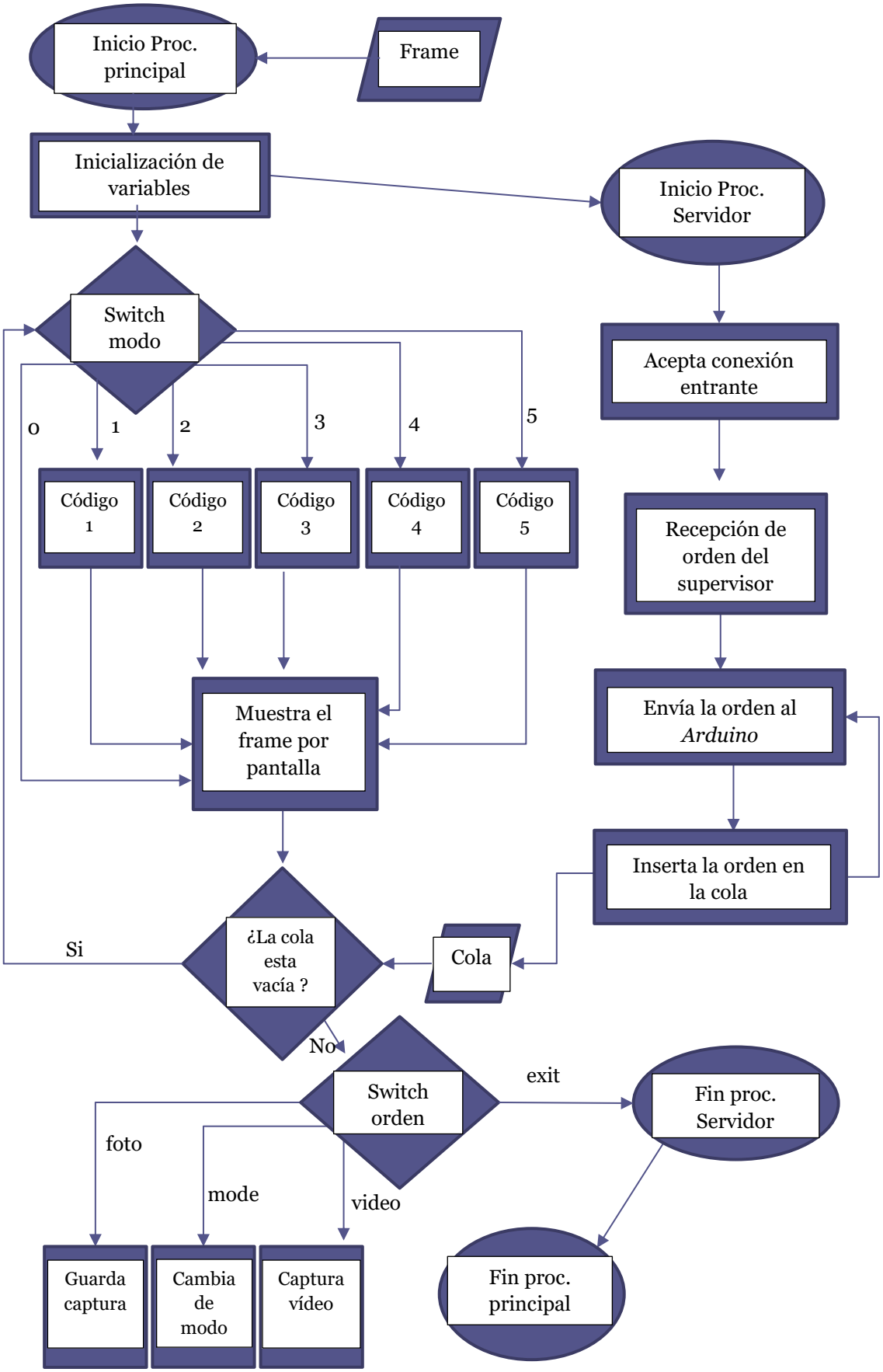


Figura 4.6 Flujo del proceso principal del la Raspberry

Una frecuencia de muestreo de un ciclo por segundo equivaldría a que el conductor de un coche cerrara los ojos y solo los abriera una vez cada segundo, lo que sería un auténtico desastre.

Afortunadamente, no es el caso, ya que en cada prueba se han podido conseguir frecuencias de muestreo que se adecúan a las mismas. Aun así, se busca siempre el algoritmo más eficiente porque cada décima de segundo cuenta. Se ha comprobado en las distintas pruebas. Por ejemplo, en la primera, relacionada con el seguimiento de una línea, una frecuencia de muestreo media menor a diez ciclos por segundo era inadmisible y hacía que el robot se saliera de la pista, es decir, que si el robot tarda más de una décima de segundo en captar y procesar la imagen, se sale de las curvas. Y no se trata de un problema del control de motores, como se creyó al principio, sino de una característica inherente al sistema con la que hay que lidiar, intentando caer lo menos posible en los errores que provoca.

La frecuencia de muestreo depende del tiempo de ejecución del algoritmo, el cual se puede mejorar optimizando el algoritmo o ejecutándolo en un dispositivo más rápido.

4.3.3 Programa del *Arduino*

Dependencias

- *servo.h*: Lo utilizamos para controlar nuestro servo conectado a un pin *PWM*

Estructura

Tiene una estructura muy simple, marcada por el propio microcontrolador. En primer lugar, se declaran las variables y constantes globales. Seguidamente se ejecuta la función *setup*, que se encarga de la inicialización de las variables y la preparación para la función *loop*, que se ejecuta en bucle. En nuestro caso *loop* comprueba si puede leer una orden del puerto serie y lo hace en caso afirmativo.Conocedor del formato de las órdenes, separa el comando y los parámetros, analiza el contenido de la orden y actúa en consecuencia.

Para las órdenes que implican el movimiento de los motores, se ha definido una función que les transmite las velocidades dadas mediante los pines conectados al controlador *L298n*.

4.4 Detalles del proceso de desarrollo

Otro aspecto que se incluye en este punto es la forma en la que se ha desarrollado el código. Concretamente, nuestro protocolo de desarrollo se caracteriza por tres parámetros:

- Existe la continua necesidad de **transferir datos** (código, capturas de la cámara, etc) entre el *PC* y la *Raspberry*, lo cual puede hacerse de diferentes formas. Podemos utilizar un *usb* o subir los datos a la nube, sin embargo, ésto requiere acciones por nuestra parte desde la *Raspberry*, lo que puede no sonar problemático, pero lo es teniendo en cuenta que para ello hace falta que tengamos acceso a la *Raspberry*, conectándole cada vez un monitor, un ratón y un teclado o accediendo a ella de forma remota. Como alternativa, se pensó utilizar el comando *scp* desde

la terminal del *PC* que sirve para realizar una copia remota desde o hasta otro dispositivo, es decir, ejecutando este comando en un *PC*, podemos traernos datos desde la *Raspberry* hasta el *PC* o copiarlos desde el *PC* hasta la *Raspberry*. En la Figura 4.7 podemos ver un ejemplo de la utilización de este comando.

```
C:\Users\Alex\Desktop\TFG>scp -pr pi@192.168.1.8:/home/pi/tfg/fotos/capturas* fotos
pi@192.168.1.8's password:
captura_97.png                              100% 3650     3.6KB/s   00:00
captura_99.png                              100% 3169     3.1KB/s   00:00
captura_104.png                             100% 5524     5.4KB/s   00:00
captura_91.png                              100% 2839     2.8KB/s   00:00
captura_101.png                             100% 5873     5.7KB/s   00:00
captura_87.png                              100%   25KB  24.6KB/s   00:00
```

Figura 4.7 Transmisión de datos por scp

- Otro punto clave es la **ejecución del programa de la *Raspberry***. Para iniciar cualquier funcionalidad del robot, éste tiene que estar activo. En un principio se pensó en incluir el comando de la ejecución en el fichero *local.rc* de la *Raspberry*, pues el contenido de este archivo se ejecuta cada vez que se enciende el dispositivo.

  El problema con esta alternativa era que cada vez que se quería ejecutar el programa (por ejemplo porque se habían hecho cambios) era necesario reiniciar la *Raspberry*.

  La alternativa que se escogió fue la de acceder a la misma de forma remota. Para ello, era suficiente con conectarnos mediante el comando *ssh*. No obstante, esta solución dejaba algunos problemas sin cubrir, ya que *ssh* no transmite elementos gráficos, lo que resultaba un impedimento a la hora de desarrollar el código del Arduino directamente desde la Raspberry y sobre todo para acceder en el *PC* a la realimentación visual de la cámara.

  Por estos motivos, se cambió la conexión por *ssh* por un cliente de escritorio remoto, que nos da acceso a la Raspberry y a su interfaz gráfica. Para ello hay que instalar previamente el programa *xdrp* (hay otros clientes de escritorio remoto, como *VNC*) en la *Raspberry*. La Figura 4.8 es una captura de la *Raspberry* que, accedida por escriotrio remoto, está ejecutando el programa y mostrando en una ventana el vídeo captado.

  Podemos conectarnos fácilmente si el *PC* y la *Raspberry* están en la misma red. Basta con utilizar la dirección *ip* local de la *Raspberry* (en nuestro caso 192.168.1.8). Si, por contra, están conectados a distintas redes, es algo más complicado. Se tiene que utilizar la *ip* pública (que es la del *router*) y un puerto.Ésto no funciona a menos que configuremos el *router* para que redireccione a la *Raspberry* el tráfico del puerto que usamos .

  Por último, recalcar que, aunque se ha hablado de conectarnos a la *Raspberry* desde el *PC*, todo lo que se ha dicho es totalmente transferible a otros dispositivos, como *smartphones* o *tablets*. Existen, por ejemplo, muchas aplicaciones *Android* tanto de clientes *ssh* como de escritorio remoto.

- La **realimentación visual** es otro punto necesario que ya se ha mencionado anteriormente. Es muy útil poder ver en tiempo real los *frames* captados por la cámara para la corrección de los algoritmos de visión artificial desarrollados. Para este fin se intentó transmitir las imágenes en tiempo real mediante los *sockets* que comunican la *Raspberry* con el *Host*. Sin embargo, el tamaño del *buffer* de la *Raspberry* no es lo suficientemente grande como para enviar una imagen completa de una sola vez. Por este motivo, la secuencia de bytes, que era la imagen, se tenía

que dividir en varias partes que después se recomponían en el *host*.

Este proceso acabó funcionando pero era demasiado ineficiente. Esta idea se descartó completamente, pasando a utilizar la opción del escritorio remoto que ya se ha descrito en el punto anterior.

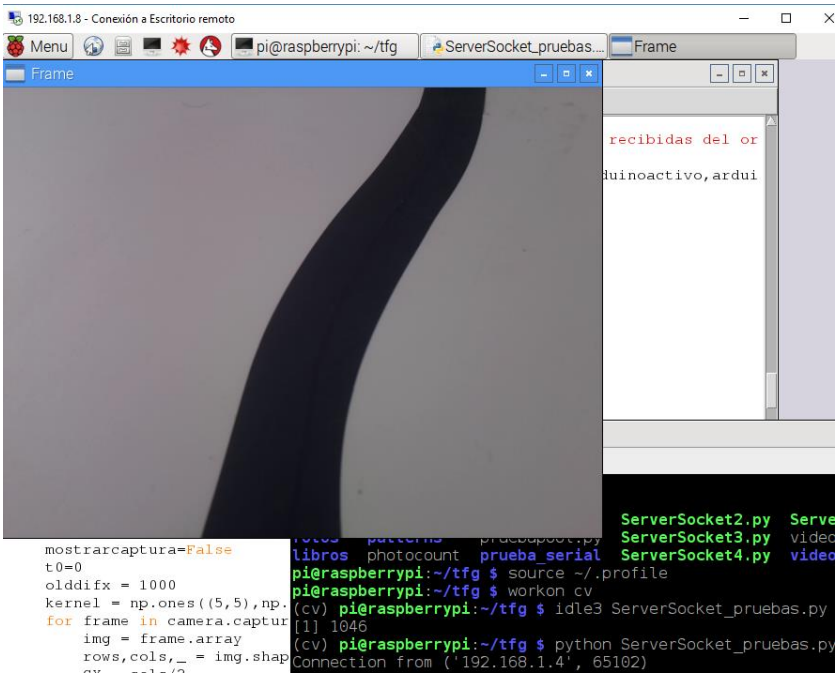


Figura 4.8 Escritorio remoto a la Raspberry con xdrp

5. Control de motores

El control de motores se ejecuta en cada iteración del bucle principal del programa de la *Raspberry* concretamente en los modos uno, dos y tres.

Su objetivo en cada ejecución es calcular la orden de movimiento a enviar al *Arduino*

5.1 Entrada

El control de motores utiliza como entrada la distancia entre dos puntos. Ambos se representan en la Figura 5.1.

Figura 5.1 Entrada del control de motores

Punto1

Es el punto al cual el robot está mirando. Si visualizamos lo que la cámara percibe, este punto se halla siempre en el centro . Su valor constante es la mitad de la anchura de la imagen.

Punto2

Es el punto al cual el robot debería estar mirando. También lo llamaremos punto de interés. Es una variable y la forma en la que se calcula su valor depende de la prueba. Por ejemplo, si tenemos que seguir un determinado objeto, este punto podría ser el centro del objeto.

5.2 Métodos de control

En cada iteración calculamos el punto de interés y la distancia horizontal entre ambos puntos restando las componentes *X* del punto dos y el punto uno. Si la distancia es igual a cero, quiere decir que los puntos coinciden y que por lo tanto el robot está yendo exactamente a donde debe ir, en caso contrario, deberá corregir su rumbo. Si la distancia es menor que cero, quiere decir que el punto de interés está a la izquierda y que el robot tendrá que corregir a la izquierda. Si la distancia es mayor que cero, ocurre lo contrario: El punto de interés está a la derecha y ,por lo tanto, el robot debe corregir a la derecha.
Se han desarrollado diferentes algoritmos de control

5.2.1 Basados en la distancia actual

Toman como única información de entrada la distancia horizontal de ambos puntos en la iteración actual del bucle.

Control por signo

Es el algoritmo de control más básico que existe. La respuesta del robot solo depende del signo de la distancia y no de su valor. Si la distancia es nula, seguimos recto. Si es negativa, giramos a la izquierda (movemos hacia delante la rueda derecha mientras dejamos fija la izquierda). Si es positiva,

giramos a la derecha (movemos hacia delante la rueda izquierda mientras dejamos fija la derecha). Solo hay estas tres posibles opciones y todas se hacen con una velocidad constante e independiente del valor de la distancia. Más distancia no significa más velocidad de giro. Este algoritmo es excesivamente simplista y no se adapta a las necesidades del proyecto.

Control por intervalo

Es una ampliación del control por signo, puesto que define el sentido de giro igual que éste, sin embargo, obtiene la velocidad de giro en función del valor de la distancia. Si la distancia pertenece a un intervalo, se corresponderá con una velocidad. Si la distancia pertenece a un intervalo mayor, se corresponderá una velocidad mayor. Aunque mejora ligeramente el anterior, hay que añadir muchos intervalos (obtenidos empíricamente) para conseguir resultados aceptables.

Control proporcional

La velocidad de giro se obtiene a partir de una función lineal de la forma $v = v0 + k * abs(difx)$ en la que v es la velocidad resultante, v0 es la velocidad base (en nuestro caso 120), *difx* es la distancia de entrada y k es una constante $k = incv/dmax$. *Incv* es el incremento máximo de la velocidad en función de la distancia. *Dmax* es la distancia máxima que podría haber entre los dos puntos y típicamente se corresponde con la mitad de la anchura de la imagen. Si por ejemplo, en nuestro caso, utilizamos un incv de 60 y una dmax de 320, si la distancia es 320, $v = 12 + 60 = 180$.

Control proporcional modificado

Al control anterior se añaden algunas mejoras para hacerlo más realista. Imaginemos con el anterior que el robot detecta una distancia de -5, gira a la izquierda y a continuación detecta una distancia de +5 y gira a la derecha. Y así sucesivamente. Si establecemos una distancia mínima por debajo de la cual el robot no gira(por ejemplo veinte), seguirá recto, evitando oscilar continuamente entre dos distancias pequeñas en lugar de seguir recto.

Otra mejora que se ha hecho es a la hora de girar. No todos los giros son tan cerrados como para necesitar que una de las ruedas esté fija. Muchos de ellos se pueden hacer de forma más natural moviendo lentamente esa rueda en vez de dejarla parada. A esta rueda que estaría parada en el giro proporcional, la llamamos rueda secundaria.

Por ejemplo, si tenemos una suave corrección a la izquierda, conviene mover la rueda derecha según la velocidad calculada por el control proporcional pero también un poco la rueda secundaria (izquierda) para que la maniobra sea menos brusca. De esta forma, establecemos un límite de distancia por debajo del cual los giros son considerados abiertos (y por lo tanto se da una velocidad constante a la rueda secundaria). Por encima de ese límite, los giros son considerados cerrados y se actúa igual que en el algoritmo de control anterior (mantienen quieta la rueda secundaria).

5.2.2 Basados en la distancia actual y la anterior

Nos dimos cuenta de que, el control proporcional no funcionaba siempre bien. Cuando el objeto que seguíamos se encontraba cerca pero se alejaba rápidamente,

el control proporcional era incapaz de reaccionar suficientemente rápido, obteniendo una velocidad de giro demasiado lenta y dejando escapar al objeto.

Por otra parte, cuando éste se encontraba lejos pero no se movía o se alejaba muy lentamente, el control proporcional obtenía una velocidad de giro excesiva, haciendo que el robot se pasase de largo. Estas anomalías se deben a que el control proporcional solo considera la distancia y no la variación de la misma.

En el primer caso, percibe el objeto como cercano, así que calcula una velocidad baja. En el segundo caso, el objeto está lejos y por eso calcula una velocidad alta. Sin embargo, en estos dos casos, el factor que realmente debería tener en cuenta no es la distancia sino la variación de la misma a lo largo del tiempo. Por ello, además de la distancia horizontal entre ambos puntos en la iteración actual del bucle, estos algoritmos toman también como entrada la distancia correspondiente a la iteración anterior. Por supuesto, ambos incluyen todos los añadidos del control proporcional modificado.

Control proporcional con derivativo por signo

Este algoritmo resta la distancia actual y la distancia de la iteración anterior. Si la distancia actual es menor, significa que vamos bien y que nos estamos acercando. Si la distancia es mayor, quiere decir que nos estamos alejando (El objeto se aleja del robot más rápido de lo que el robot se acerca a él), así que se aumenta en una constante la velocidad obtenida por el control proporcional.

Control proporcional derivativo

La relación entre este algoritmo y el anterior es la misma que la que hay entre el control proporcional y el control por signo.

El valor que se suma a la velocidad debido a la variación de la distancia ya no es una constante, sino una variable calculada a partir de una función lineal. La ecuación queda de la siguiente forma *v = v0 k*abs(difx) + k2*max(0,incdistancia)* donde el último término es el que se añade en este algoritmo. *K2* es una constante que calculamos de la siguiente forma: *k2 = incv2/maxvar.*

Incv2 es el máximo valor en el que incrementamos la velocidad debido a la variación de la distancia.

Maxvar es la variación máxima de la distancia que empíricamente puede observarse.

Incdistancia es el incremento de la distancia que se ha producido en la última iteración.

Si el incremento es negativo (nos hemos acercado a nuestro objetivo desde la iteración anterior), al término se le asigna valor nulo para que no disminuya la velocidad.

6. Prueba 1: Seguimiento de linea

Esta prueba consiste en el seguimiento por el robot de una linea negra sobre fondo blanco. Para llevar a cabo esta prueba, se han fabricado varios circuitos con cinta aislante negra sobre una plancha blanca de cartón pluma. Por supuesto, no se han utilizado sensores infrarrojos ni de color, tan comunes en los robots que siguen lineas. Como en el resto de pruebas, el único sensor es la cámara, que, en esta prueba se dispone mirando directamente hacia el suelo, con el servo, a un ángulo de 180º.

6.1 Blanco y negro en *OpenCV*

Antes de entrar en los circuitos y los resultados de la prueba, vamos a hablar de cómo detectamos negro sobre blanco en *OpenCV*.

En *OpenCV* hay muchos y diversos métodos para diferenciar colores. En este caso en concreto, en el que lo único que necesitamos es detectar negro sobre blanco, vamos a centrarnos en tres métodos simples, los cuales ofrecen resultados sobradamente buenos con la eficiencia que sólo los algoritmos sencillos tienen:

6.1.1 Detección de contornos por gradiente

Estos algoritmos toman como entrada una imagen en escala de grises. La salida es una imagen con el fondo negro sobre la que están dibujados los contornos hallados en la imagen de entrada. *OpenCV* ofrece dos alternativas diferentes de gradiente: *Sobel*, que se puede usar en el eje *X* o en el *Y*, especificando la dirección de los gradientes y por tanto obteniendo los contornos horizontales o verticales respectivamente. El otro método es el de Laplace, que calcula los contornos a partir de una relación que incluye las derivadas de *Sobel* en *X* e *Y*.

6.1.2 Detector de contornos *Canny*

Es una evolución de los métodos anteriores y el que realmente se recomienda para la detección de contornos con *OpenCV*. Los formatos de entrada (escala de grises) y salida (blanco y negro) son los mismos que para la detección por gradiente.

Se trata de un algoritmo de cuatro pasos. En primer lugar se aplica un filtro Gaussiano a la imagen para eliminar ruido. En segundo lugar se calculan los gradientes a partir de las derivadas de *Sobel* en *X* e *Y*. A partir de este paso, ya tenemos los candidatos a contornos. En tercer lugar, se suprimen los contornos cuyos gradientes no representan un máximo local (ya que ésto suele indicar que se trata de ruido en lugar de contornos reales). El cuarto y último paso también se centra en la eliminación de los contornos no aptos. En este caso utiliza una técnica llamada umbralización por histéresis. lo que significa que se eliminan los contornos cuyos gradientes estén por debajo de un mínimo y también los que son demasiado cortos o no están conectados a un contorno con gradiente alto.

6.1.3 Umbralización

Es un conjunto de métodos cuyo objetivo es convertir una imagen de entrada (en escala de grises) a blanco y negro a partir de un límite o umbral.

Ésto solo es exactamente así cuando hablamos de umbralización binaria, que es la que nos interesa. Concretamente usaremos umbralización binaria inversa. Ésto quiere decir que los píxeles que estén por encima del umbral, serán pintados de negro, mientras que los que estén por debajo del umbral, serán pintados en blanco (si no fuera inversa, funcionaría al revés), ya que para pasos posteriores necesitamos que el objeto a seguir (la cinta aislante negra) quede de color blanco en nuestra imagen umbralizada. Hay tres métodos de umbralización que se diferencian básicamente en cómo obtienen el umbral:

Umbralización simple

El umbral es el dado por el usuario. Símplemente aplica la opción especificada (en nuestro caso umbralización binaria inversa).

Umbralización adaptativa

Este método calcula un umbral distinto para cada región de la imagen (o vecindario) cuyo tamaño es dado por el usuario. Además, se tiene que especificar la forma en la que se calculan los umbrales. En *OpenCV* podemos calcularlos como la media de los valores del vecindario, o como su media ponderada utilizando una ventana Gaussiana.

La umbralización adaptativa es otra forma de detección de contornos. Al obtener umbrales locales para cada vecindario, las zonas oscuras tendrán umbrales bajos y las zonas iluminadas tendrán umbrales altos, de forma que ambas serán iguales en el resultado. Los vecindarios que contengan a la vez una zona clara y otra oscura, serán los únicos cuyo umbral separe el negro del blanco, lo que se traduce en una detección de contornos.

Umbralización de *Otsu*

Es un método que calcula automáticamente el umbral óptimo, lo cual evita tener que hallarlo empíricamente, como se hace en la umbralización simple.

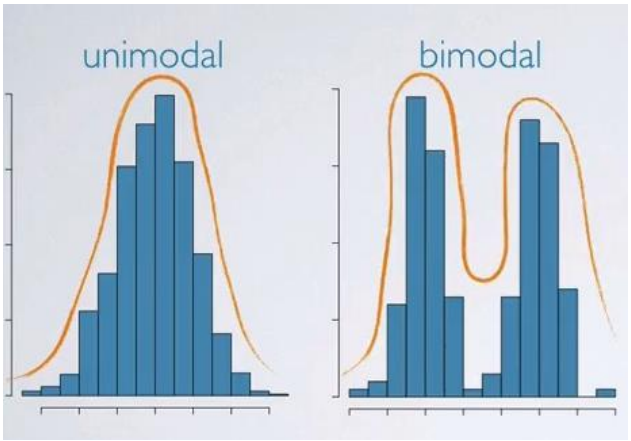


Figura 6.1 Histograma unimodal VS bimodal

Cuando el histograma de una imagen, que es la gráfica que representa en el eje *X* el rango de valores y en el eje *Y* la cantidad de píxeles de cada valor que contiene la imagen en escala de grises, tiene dos máximos locales bien separados, a ésta se le llama imagen bimodal. Ésto quiere decir que la imagen consta de dos polos opuestos en cuanto a brillo: Una parte oscura y una parte clara.

Este tipo de imágenes son aptas para la binarización de *Otsu*, que obtiene el umbral como un valor entre los dos montículos del histograma, separando la imagen de forma óptima. La Figura 6.1 muestra una comparativa entre un histograma unimodal y otro bimodal.

6.2 Ejecución de la prueba

Centrándonos ya en la prueba, vamos a razonar qué opción hemos elegido para la detección del circuito y por qué es la más adecuada.

6.2.1 Técnicas consideradas

Derivadas de *Sobel*

Se decidió prescindir de los detectores de contorno por gradiente porque tanto el método de *Sobel* como el de Laplace, generaban ruido en la imagen (Figura 6.4.B).

Umbralización adaptativa

Este algoritmo representa una mejora importante respecto a *Sobel*. Los contornos hallados están mucho más definidos y el ruido que genera es muy pequeño y por lo tanto fácilmente eliminable con una transformación morfológica sin poner el riesgo el resultado (Figura 6.4.D).

Canny edge detection

Canny ha demostrado que en este campo no tiene rival. Nos ha ofrecido muy buenos resultados. Un contorno completo, totalmente definido y sin ruido (Figura 6.4.C).

La umbralización simple

Tras hallar empíricamente el umbral, funcionaba muy bien y distinguía el circuito y sólo el circuito (Figura 6.4.E). El problema venía cuando las condiciones cambiaban. Por ejemplo, si se intentaba probar con el mismo umbral a otra hora del día, la posición del sol era distinta y por tanto se reflejaba más luz en la cinta aislante (Figura 6.2.A). En este caso podía pasar que parte del circuito estuviera por encima del umbral y fuera percibido como blanco aún a pesar de ser negro (Figura 6.2.B).

También ocurría al revés. Debido a la hora del día o a la orientación del robot en determinadas partes del circuito, las sombras proyectadas por el mismo podían llegar a ser lo suficientemente oscuras como para estar por debajo del umbral y ser percibidas como negras, es decir, como parte del circuito.

En resumen, la umbralización simple funciona muy bien bajo condiciones fijas. Los problemas vienen cuando cambian las condiciones sin cambiar el umbral.

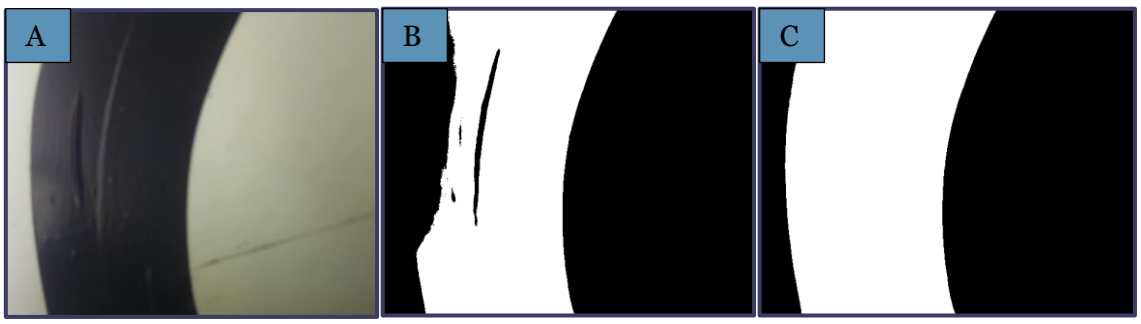


Figura 6.2 Umbralización simple VS Otsu en iluminación no uniforme

La umbralización de *Otsu*

La Umbralización de *Otsu* es perfecta para esta prueba. En cada iteración está hallando el umbral óptimo y por ello funciona tan bien como la umbralización simple (Figura 6.4.F) con la diferencia de que se adapta a los cambios (Figura 6.2.C). Si cambian las condiciones de iluminación, también lo hará el umbral óptimo y por tanto se seguirá percibiendo el circuito y solo el circuito, que es al final el objetivo de la prueba.

6.2.2 Problema con *Otsu*

El problema con este algoritmo se da cuando la imagen que toma no es bimodal (no tiene dos montículos bien diferenciados). Durante la prueba, ésto ocurre solo cuando el robot no está viendo el circuito (Figura 6.3.A). Debido a las oscilaciones del control de motores, los giros bruscos y la frecuencia de muestreo, es imposible que el robot vea siempre el circuito. En algunos momentos, aun sin salirse, ocurre que el robot pierde momentaneamente de vista el circuito. Lo importante en estos casos es que el robot sepa diferenciar cuando lo está viendo y cuando no, de forma que, cuando no lo vea, descarte la imagen.

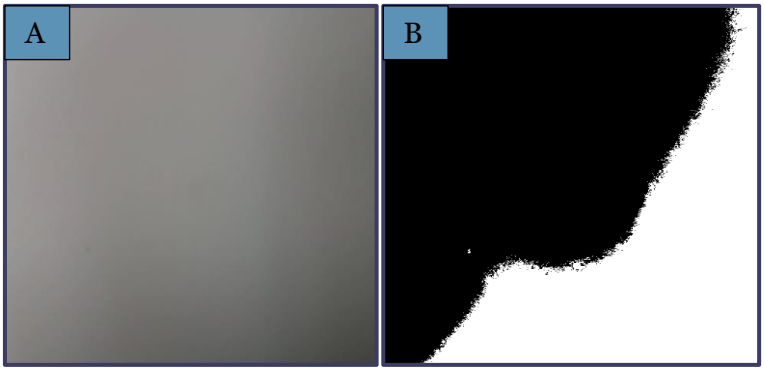


Figura 6.3 Umbralización de Otsu con un histograma unimodal

Por ejemplo, imaginemos un giro muy brusco a la izquierda.El robot percibe que el circuito está a la izquierda y que por lo tanto tiene que girar hacia allí. El programa envía la orden al *Arduino*, que empieza a girar, sin embargo, no lo hace lo suficientemente rápido y su cámara pierde brevemente de vista el circuito. Si usa estas imágenes sin circuito, la umbralización de *Otsu* será un caos (Figura 6.3.B) porque la imagen no es bimodal . El programa calculará una maniobra errónea y se la mandará al *Arduino*, que la ejecutará, sacando definitivamente al robot del circuito. Si por contra, ignora estas imágenes en las que no ve el circuito, el *Arduino* no recibirá nuevas órdenes y seguirá girando a la izquierda, lo que hará que el robot vuelva a encontrar el circuito.

Canny* vs *Otsu

Llegados a este punto, la elección era entre *Canny* y *Otsu*. Como hemos apuntado, el segundo presenta un problema grave. Desde este punto de vista parecería más adecuado utilizar *Canny*. No obstante, si se conseguía soslayar el problema, *Otsu* representaba ventajas respecto a *Canny*. Para empezar, aunque *Canny* es eficiente y su ejecución en tiempo real no es ninguna imposibilidad, *Otsu* es un algoritmo más simple y computacionalmente menos pesado. Y como ya se ha apuntado anteriormente, la eficiencia es un aspecto clave que se ha tenido en cuenta constantemente.

Otro factor que ha influido en la decisión es que el control de motores necesario para las pruebas dos y tres es uno pensado para el seguimiento de objetos y no de contornos. Así pues, utilizar *Canny* supondría la necesidad de realizar una versión alternativa de control que se adaptara mejor a la detección de

contornos. Sin embargo, umbralizando con *Otsu*, no hay que hacer absolutamente ningún cambio en ese sentido.

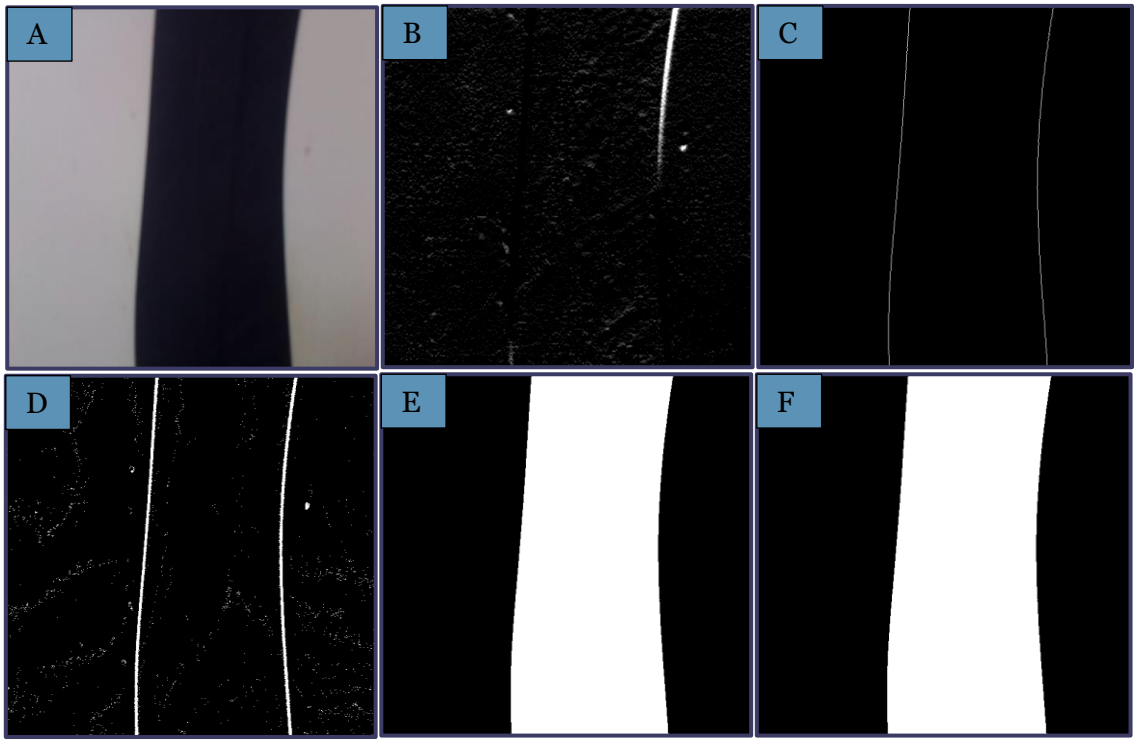


Figura 6.4 Resultados de los algoritmos en condiciones estándar

6.2.3 Posibles soluciones al problema

A continuación, se plantean algunas alternativas para solucionar el problema de utilizar *Otsu* en esta prueba, es decir, la necesidad de distinguir una imagen con circuito, o válida, de una imagen sin circuito o inválida.

Machine learning

Se intentó entrenar un clasificador lineal para determinar mediante el aprendizaje automático a partir de qué umbral hallado por el algoritmo de *Otsu* es óptimo considerar que la imagen es válida. Se tomaron varias muestras de los umbrales de *Otsu* de imágenes válidas e inválidas.

El resultado fue una gran decepción. Los umbrales hallados en las imágenes válidas eran muy similares, pero los de las imágenes inválidas eran prácticamente aleatorios (y muchos se asemejaban a los de las imágenes válidas). No era posible diferenciar ambas clases en base al valor de su umbral de *Otsu*.

Cálculo de bimodalidad

Otra estrategia que se siguió fue hallar el histograma de la imagen y medir su bimodalidad. Es viable distinguir los histogramas de las imágenes válidas(con dos montículos bien separados) de las inválidas(con un solo montículo).

Umbral óptimo inicial

Una estrategia que podría haber sido útil pero que no llegó a probarse fue la de hallar al principio del circuito el umbral óptimo de *Otsu* y usarlo como umbral simple el resto del circuito. Así se eliminaría el problema de las imágenes inválidas de *Otsu*, ya que la umbralización simple las vería vacías y el de la adaptabilidad a los cambios de luz a lo largo del tiempo, ya que al principio de la prueba se calcularía el umbral óptimo. El problema que sí podría tener esta etrategia, por el cual decidimos no usarla, es que seguiría teniendo los mismos problemas de adaptabilidad que la umbralización simple cuando cambien las condiciones de luz en distintas partes del circuito.

Número de objetos detectados

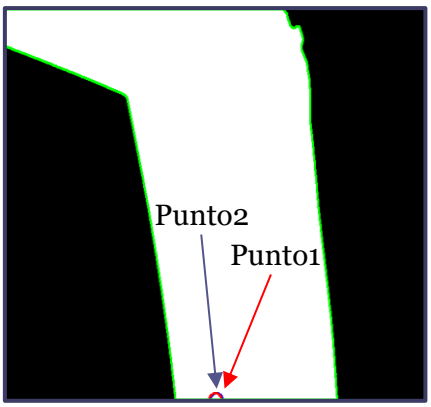
Posteriormente se pensó otra solución al problema que era tremendamente sencilla y eficiente. La comprensión de esta solución requiere la exposición del paso siguiente a la umbralización.

Tras la umbralización, se detectan los objetos de color blanco. Para ello se utiliza la funcion *findcontours* de *OpenCV*. Obviamente, en las imágenes válidas se detectan muy pocos objetos,entre los cuales destaca el fragmento de circuito que estamos viendo. Por contra, las imágenes inválidas estan llenas de ruido y en ellas se detectan muchos objetos, la mayoría de ellos pequeños.

Para intentar vaciar las imágenes inválidas, se utilizaron técnicas de eliminación de ruido, como los filtros gaussianos o las transformaciones morfológicas, que veremos posteriormente en otras pruebas. En esta prueba en concreto, no fueron útiles, puesto que no consiguieron eliminar la totalidad del ruido en las imágenes inválidas sin alienar demasiado las válidas. Por este motivo, se prescindió de la eliminación de ruido y se ideó la que sería la estrategia definitiva para la distinción entre imágenes válidas e inválidas. Como ya se ha explicado, en las primeras se detectan muy pocos objetos (menos de 50), mientras que en las segundas se detectan muchos (más de 150). Por consiguiente establecemos un límite de 100 objetos detectados, por encima de los cuales, la imagen pasa a ser inválida y descartada.

6.2.4 Detección del circuito

Dentro de las imágenes válidas, de entre todos los objetos obtenidos, debemos seleccionar y seguir siempre el que se corresponde con el circuito, ya que el resto son ruido proveniente principalmente de sombras. Para este fin, seleccionamos el objeto con mayor área, que va a ser siempre el circuito.



Una vez seleccionado el objeto a seguir, tenemos que obtener el punto de interés que necesitamos para el control de motores. Recordemos que el punto de interés es el que va a marcar la maniobra que realicemos en esa iteración. Dado el objeto, se han probado diferentes formas de hallar el punto de interés:

Figura 6.5 Problema al usar el centro de gravedad

Centro de gravedad

Gracias a *OpenCV* podemos obtener los momentos de un objeto, lo que nos permite calcular el centro de gravedad. El centro de gravedad funciona bien como punto de interés en rectas y curvas suaves. En curvas cerradas puede llegar a dar resultados opuestos a los esperados. Como podemos ver en la siguiente curva hacia la izquierda de la Figura 6.5, cuyo centro de gravedad está en medio.

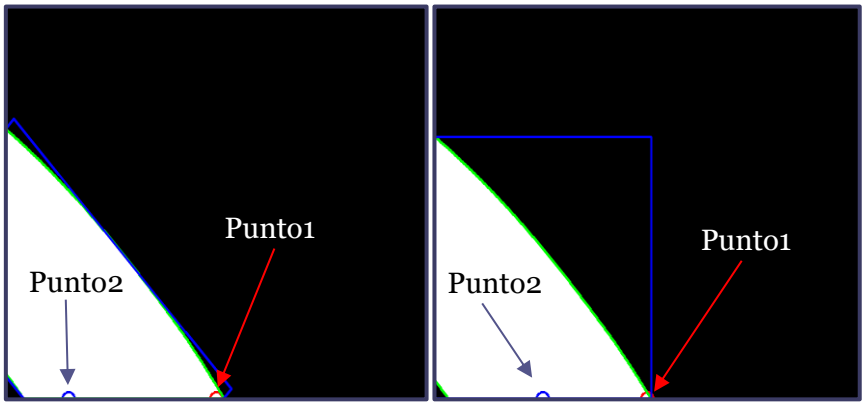


Figura 6.6 Problema al usar el centro del rectángulo rotado

Centro del rectángulo rotado que contiene el objeto

Funciona bien en la mayoría de casos, pero como la posición del centro depende del ángulo de rotación del rectángulo, en algunas curvas cerradas, el ángulo de rotación cambia demasiado deprisa haciendo que el punto de interés oscile (Figura 6.6) y confunda al robot.

Centro del rectángulo que contiene el objeto

Los únicos parámetros del rectángulo son las coordenadas del vértice superior izquierdo, la anchura y la altura. No tiene ángulo de rotación. Funciona bien en prácticamente todos los casos (Figura 6.7). Otro ejemplo más, como todos los que se están dando en este trabajo, de que, a menudo, la mejor solución puede ser también la más sencilla.

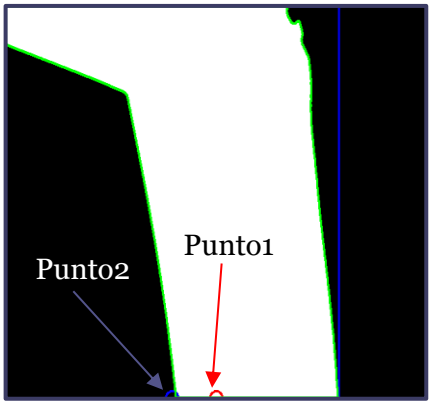


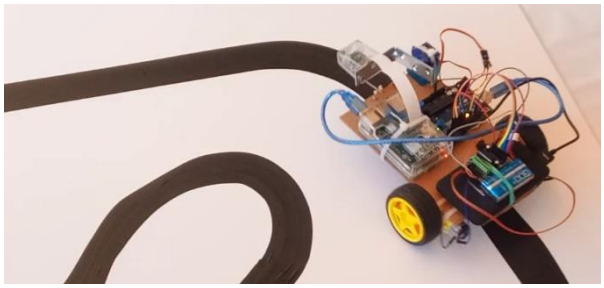
Figura 6.7 Resultado correcto

6.3 Resumen del algoritmo

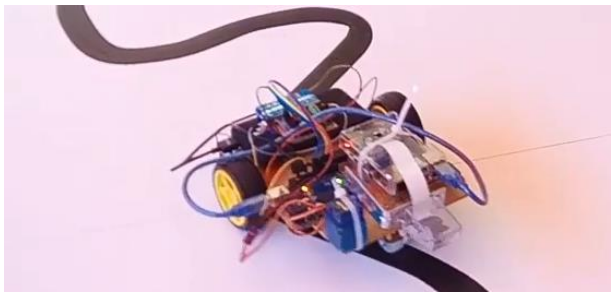
En primer lugar, se convierte la imagen a escala de grises. Posteriormente se binariza a blanco y negro mediante la umbralización de *Otsu*. Luego se detecta los objetos blancos de la imagen binarizada con *findcontours*. Si el número de objetos detectados es mayor que cien, se descarta la imagen por ser inválida. De lo contrario, se escoge el objeto con mayor área y se halla el centro del rectángulo sin rotar que lo contiene. La componente horizontal de dicho centro se pasa al control de motores como punto de interés. El algoritmo se encuentra representado en el diagrama de flujo de la Figura 6.8.

6.4 Resultados

Una vez explicado todo el algoritmo, pasemos a los resultados que se han obtenido en esta prueba. Se han diseñado dos circuitos, siendo el segundo más complejo que el primero. En el primer circuito, se probó el control de motores proporcional modificado y en el segundo el proporcional derivativo. Como se puede observar en los dos vídeos, se han obtenido resultados satisfactorios.



https://youtu.be/uHoMIRX7jwU



https://youtu.be/x_iU9WkqbxY

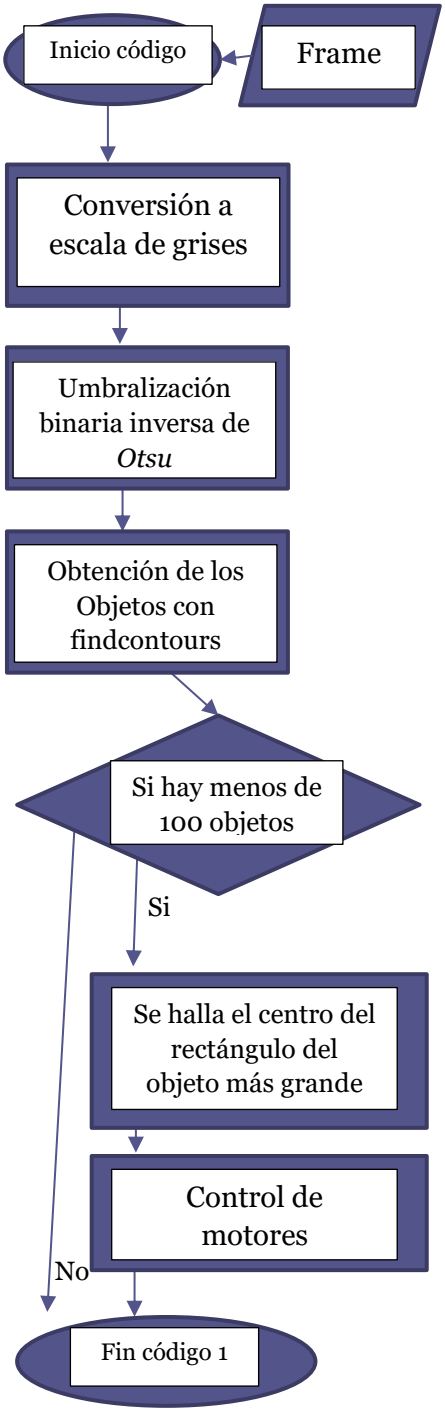


Figura 6.1 Flujo del código específico del modo 1

7. Prueba 2: Seguimiento por color

El objetivo de la segunda prueba es que el robot sea capaz de seguir un objeto simple de un determinado color. Se eligió la forma esférica por la comodidad que supone el poder hacerla rodar para que el robot la siga.

Como se trata de un seguimiento por color, se eligieron para el objeto a seguir colores vivos que destacan sobre el entorno en el que se hace la prueba, con el fin de reducir el ruido durante la detección. Se probó con pelotas de distintos colores y se escogió arbitrariamente la verde. Por otra parte, la cámara se encuentra inicialmente mirando al frente (con el servo en un ángulo de 90 º), sin embargo, para que ésta pueda adaptarse a la distancia de la pelota, se ha escrito una función que sube la cámara si la pelota está lejos (o arriba) y la baja si está cerca (o abajo). Se entrará más en profundidad sobre este algoritmo durante la prueba.

7.1 Detección del objeto

El algoritmo de esta prueba es muy sencillo. En cada iteración del bucle, se convierte la imagen de *bgr* a *hsv* para posteriormente aplicarle el método *imrange* de *OpenCV*. Al llamar al método, el usuario tiene que establecer un rango de color, otro de saturación y otro de brillo. Los píxeles que se encuentran en los tres rangos a la vez, se dibujan de color blanco y los que no de color negro. De esa forma, el algoritmo devuelve una imagen en blanco y negro (nótese que existe cierta semejanza con la umbralización simple). El principal problema de este algoritmo es dar con los rangos adecuados.

La pelota, como esfera, tiene distinta iluminación en sus diferentes partes, por lo que un umbral de brillo demasiado restrictivo puede dejar fuera parte de la pelota. Sin embargo, un umbral de brillo demasiado laxo puede hacer que se incluya en el resultado ruido proveniente de sombras o reflejos. Tras la fase del *imrange*, se utiliza, al igual que en la prueba anterior, la función *findcontours* para hallar los objetos blancos de la imagen binaria.

En nuestro caso, con los rangos adecuados obtenidos experimentalmente, esta función devolvía el objeto que representa la pelota, además de otros objetos de inevitable ruido. A priori basta con la técnica ya mencionada de quedarnos con el objeto detectado más grande, que normalmente es la pelota. Digo normalmente porque puede ocurrir que la pelota se encuentre tan lejos que sea del mismo tamaño que el ruido.

7.1.1 El problema del ruido

El problema, una vez más, viene dado por los giros bruscos y la frecuencia de muestreo. Si apartamos lateralmente la pelota demasiado rápido, puede ocurrir que el robot deje momentáneamente de verla, solo vea ruido, y se ponga a seguirlo.

Al igual que en la prueba anterior, el objetivo aquí es ignorar las imágenes que carecen del objeto de interés para que el *Arduino* siga haciendo lo último que se le ha ordenado, que es lo correcto. En la prueba uno, para distinguir cuándo estábamos viendo la línea de cuándo no, es decir, las imágenes válidas de las inválidas (términos que se utilizarán también en esta prueba), poníamos un límite en los objetos que se detectaban, ya que, cuando la umbralización de *Otsu* se realizaba sobre una imagen válida, no había apenas ruido. Esta solución no es aplicable en nuestra prueba. Con *imrange* se genera el mismo ruido en las imágenes válidas que en las inválidas.

7.1.2 Eliminación de ruido en *OpenCV*

Se intentó mediante filtros y transformaciones morfológicas eliminar todo el ruido para que las imágenes inválidas quedaran vacías. Vamos a ahondar en estas técnicas de eliminación de ruido que ya se han mencionado anteriormente:

Los filtros Gaussianos

Utilizan una ventana de Gauss para difuminar la imagen. El valor de cada píxel se actualiza en función del suyo y los de sus vecinos contenidos en la ventana. Como resultado, se obtiene una imagen más borrosa, pero con menos ruido.

Las transformaciones morfológicas

Las operaciones morfológicas actúan sobre imágenes binarias. Dependen del *kernel* que se les pasa como parámetro, ya que todos ellos deslizan dicho *kernel* por toda la imagen. Hay cuatro tipos de transformaciones:

- Erosión: Un píxel pasa a ser blanco si y solo si ya era blanco y si, cuando es el centro del *kernel*, todos los píxeles contenidos en el mismo son también blancos. Ésto se traduce, como el propio nombre del método indica en una erosión de las partes externas de los objetos de color blanco. Obviamente, cuanto más grande sea el *kernel*, mayor será la erosión y cuando un objeto es de menor tamaño que el *kernel*, es borrado. De aquí que se use para la eliminación de ruido.

- Dilatación: Es la operación opuesta a la erosión. Un píxel pasa a ser blanco si, cuando es el centro del *kernel*, hay algún píxel blanco dentro del mismo. Se utiliza para eliminar pequeños píxeles negros que forman ruido dentro de los objetos.

- Apertura: Es una erosión seguida de una dilatación. La erosión elimina el ruido blanco y la dilatación devuelve a su tamaño normal los objetos que han pasado la erosión.

- Cierre: Es la operación opuesta a la apertura. Consiste en una dilatación seguida de una erosión. La dilatación elimina el ruido negro y la erosión devuelve los objetos a su tamaño normal.

7.1.3 Aplicación de técnicas de eliminación de ruido a la prueba

Para la eliminación de ruido se aplicaba un filtro Gaussiano sobre la imagen sin procesar, tras el cual se hacía una apertura y un cierre sobre la imagen binaria para eliminar el ruido blanco y el negro respectivamente. Todo este proceso no solo no conseguía eliminar la totalidad del ruido, sino que además añadía un lastre más al bucle principal, disminuyendo la frecuencia de muestreo.

El ruido negro estaba presente en el interior de la pelota, pero únicamente molestaba a nivel visual, así que se decidió eliminar el cierre para ahorrar ciclos de reloj. La apertura hace una dilatación al final para devolver los objetos a su tamaño natural. Sin embargo, se trata también de una parte prescindible, pues la relación de aspecto y la forma de los objetos va a ser la misma tras una erosión, que solo afecta al tamaño de los mismos.

No nos importa que los objetos queden más pequeños siempre que se elimine el ruido, así que se sustituyó la apertura por una erosión. Finalmente, también acabó eliminándose el filtro Gaussiano por no ser determinante en este caso. El ruido que eliminaba el filtro podía ser perfectamente eliminado directamente por la erosión. Gracias a estos cambios, obtuvimos un algoritmo eficiente que producía un resultado relativamente libre de ruido.

Sin embargo, no lo estaba completamente y seguía estando el problema de que no se podían distinguir las imágenes válidas de las inválidas. Por ello el robot se desviaba cada vez que perdía brevemente de vista la pelota y percibía algo de ruido.

7.1.4 La Transformada de Hough

OpenCV incorpora un algoritmo llamado transformada de Hough, cuyo fin es hallar círculos en una imagen. Internamente es un algoritmo costoso computacionalmente (sobre todo cuando es aplicado a imágenes complejas) dependiente de varios parámetros. En resumen, el algoritmo encuentra los círculos basándose en su representación matemática. Además, es un proceso muy tedioso ajustar los parámetros para obtener resultados aceptables, es decir, no detectar ni demasiados círculos (Figura 7.1.B) ni demasiado pocos (Figura 7.1.C).

Transformada de Hough sobre escala de grises

Inicialmente, se pensó en aplicar este algoritmo sobre la imagen en escala de grises y cruzar los resultados con los del *imrange*. De esta forma sería como hacer una consulta a la imagen de los objetos que fueran verdes y redondos, que es mucho más restrictiva que solo el color y que eliminaría mucho ruido.

El problema es que, debido a la iluminación no uniforme del volumen, la transformada de Hough funciona mejor con círculos planos que con esferas, así que detectaba otras formas redondas antes que la pelota (Figura 7.1.C).

Transformada de Hough sobre imagen binaria

Para intentar soslayar el problema de utilizar Hough con la imagen en grises, se decidió aplicar el algoritmo sobre la imagen ya binarizada por el *imrange*.

No obstante, la pelota no siempre era redonda en la imagen binarizada, porque el *imrange*, a veces, no la detectaba completa (Figura 7.1.D) y (Figura 7.1.E) debido a las sombras o reflejos en la misma, que se hallaban fuera del rango de brillo por abajo o por arriba respectivamente.

Transformada de Hough sobre contornos *Canny*

Se ideó, como alternativa, utilizar el método *Canny*, que ya se ha mencionado anteriormente, para detectar los contornos de la imagen en escala de grises y aplicar sobre el resultado la transformada de Hough.

El problema en este caso era que la pelota en escala de grises no siempre destacaba tanto sobre el fondo como para que *Canny* detectara su contorno completo, sin el cual, la transformada de Hough no funciona bien (Figura 7.1.F).

Además de no ofrecernos buenos resultados, este método es ineficiente en los tres casos expuestos, lo cual afectaba negativamente a la frecuencia de muestreo. Por todo ello, fue definitivamente descartado para esta prueba.

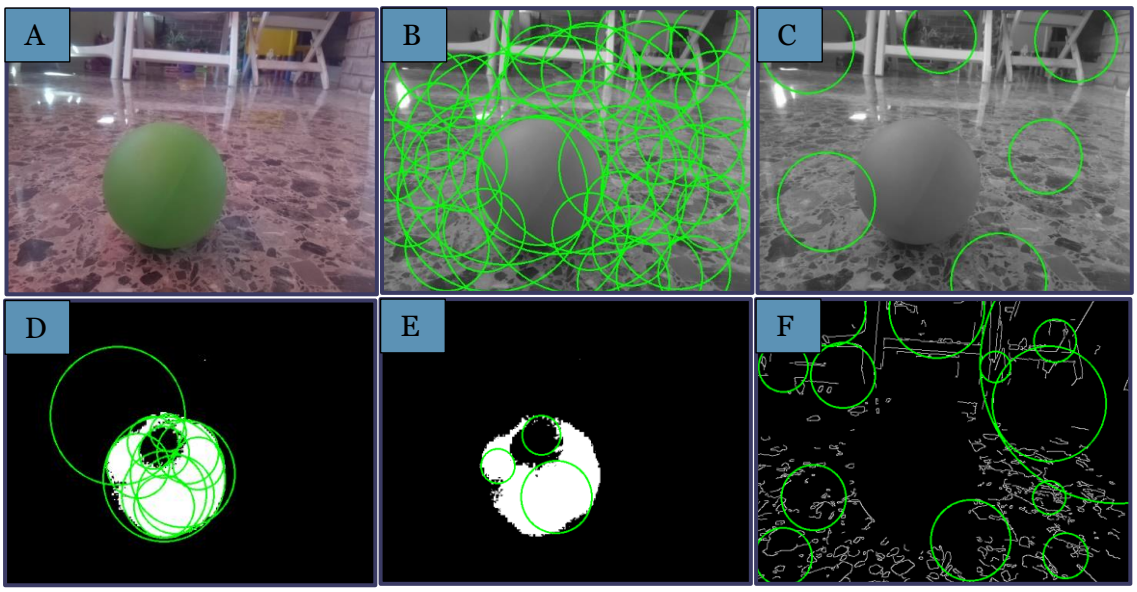


Figura 7.1 Comparación de aplicar la transformada de Hough de distintas formas

7.1.5 Cambio de parámetros de la cámara

Figura 7.2 Resultado de subir el contraste y la saturación

La solución que se adoptó finalmente fue la de subir el contraste y la saturación de la cámara.

La librería *Picamera* ofrece ambas variables con un valor predeterminado de cero. Las dos son modificables y se les puede dar un valor entre -100 y 100. Podemos hacer dichas modificaciones durante la captura de vídeo (como de hecho se hace), de forma que los *frames* posteriores a los cambios, las incorporan.

Se comprobó que con cien de contraste y cincuenta de saturación, la pelota se veía muy nítida y destacaba del entorno.

Efecto de las modificaciones de los parámetros de la cámara en la detección del objeto

El objeto a detectar era de un color vivo, por lo que, el hecho de subir el contraste y la saturación consiguió que en las capturas apareciera de color verde chillón, lo que la diferenciaba mucho del resto de la imagen (Figura 7.2.A).

Se tuvieron que volver a ajustar los rangos del *imrange*. El mayor cambio respecto a los rangos anteriores fue el de la saturación. Como la pelota (de un color muy vivo) tenía mucha saturación, se pudo ser muy restrictivo con este rango, gracias a lo cual se eliminó una gran cantidad de ruido. De hecho, el poco ruido que queda es tan pequeño que puede ser eliminado fácilmente con una erosión morfológica.

No solo eso, sino que, con el contraste y la saturación altos, las condiciones de luz afectan menos a la imagen y permiten que se vea la pelota completa y sin ruido negro en su interior (Figura 7.2.B). Ahora ya no hay ningún problema con las imágenes inválidas, pues como carecen de cualquier objeto detectado, se ignoran por completo siguiendo el rumbo anterior. También se ha conseguido una frecuencia de muestreo muy buena gracias a la eficiencia de los algoritmos utilizados.

Asimismo, se comprobó que los problemas experimentados al intentar hacer la transformada de Hough sobre la imagen binarizada o los contornos, desaparecían gracias a que la pelota destacaba mucho más sobre el fondo. Sin embargo, ya se disponía de una solución buena y eficiente y se decidió seguir prescindiendo de la transformada de Hough, que, en este caso solo añadía ciclos de reloj.

7.2 Ajuste de la inclinación de la cámara

Hemos explicado la detección de la pelota, pero falta comentar el algoritmo de inclinación de la cámara.

Imaginemos que estamos viendo un objeto. Si está muy cerca lo vemos a su verdadera altura (muy baja, a ras de suelo), pero si se va alejando, vamos viéndolo más arriba hasta que converge en el horizonte que sería el punto de fuga de la perspectiva cónica en la que se basa nuestra percepción (y también la de la cámara).

Tanto si el objeto está subiendo físicamente (aumentando su altura respecto al suelo) como si su posición aparente se debe al efecto visual de estar alejándose, la cámara ha de subir.

Al detectar el objeto con *findcontours*, se obtiene el centro del rectángulo rotado que lo contiene. Se divide la imagen verticalmente en cuatro partes de arriba abajo.

Si el centro del rectángulo se encuentra en la primera parte por arriba, lo que quiere decir que el objeto está subiendo, o que, en cualquier caso, no está centrado, se sube la cámara un ángulo prudencial de diez grados, de forma que queda en las partes centrales. Hacia abajo funciona igual. Si el centro se encuentra en la primera parte por abajo, la cámara se inclina diez grados hacia abajo.

7.3 Resumen de la prueba

En primer lugar, se aumentan de antemano el contraste y la saturación. En el bucle, cada imagen captada se convierte a *hsv,* se binariza con *imrange* y se elimina el ruido blanco con la erosión morfológica. Se detectan los objetos blancos de la imagen con *findcontours*. En caso de haber detectado alguno, se escoge como objeto de interés el de mayor área y se obtiene el centro del rectángulo rotado que contiene el dicho objeto. La componente vertical del centro se utiliza para rotar la cámara arriba y abajo, mientras que la componente horizontal se le pasa al control de motores como punto de interés. Todo este proceso se esquematiza en la Figura 7.3.

7.4 Resultados

Los resultados de esta prueba fueron muy positivos, como se puede comprobar en los vídeos a continuación.

Se evaluó tanto con el control proporcional modificado como con el proporcional derivativo.

Aunque el robot seguía la pelota en ambos casos, evidentemente se adaptaba mejor a la trayectoria de la misma con el control proporcional derivativo.

Además, el algoritmo de rotación de la cámara hace que el robot se adapte mejor a los cambios de altura y distancia de la pelota.

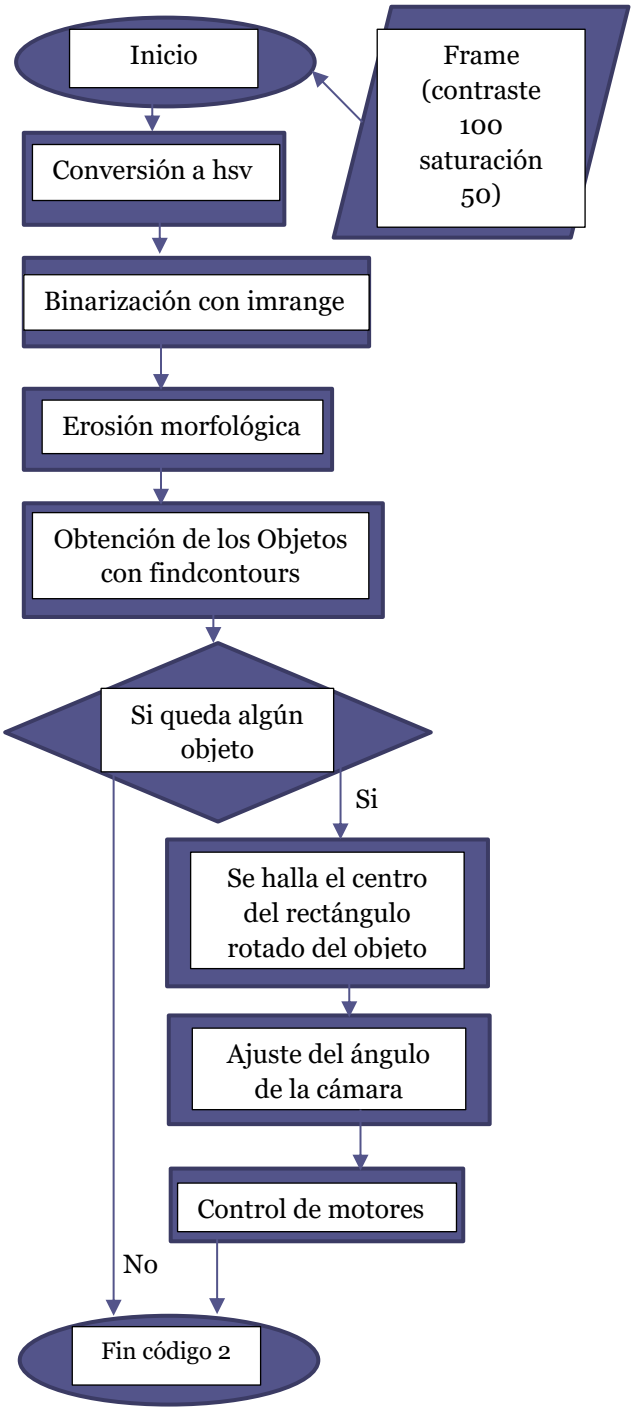


Figura 7.3 Flujo del código específico del modo 2

https://youtu.be/7WnmwKGj0IM

https://youtu.be/2Ch6x9FYiTI

8. Prueba 3: Seguimiento por histograma

Esta prueba, como la anterior, consiste en el seguimiento de un objeto por color. Sin embargo, se trata de una prueba más ambiciosa y que utiliza unas funcionalidades más avanzadas de *OpenCV*. Para empezar, no es el seguimiento de un objeto de un solo color, como sí ocurría en la prueba dos, sino que puede ser uno cualquiera con varios colores.

Sigue siendo recomendable que el objeto resalte sobre el fondo para evitar el ruido. Para este fin, del mismo modo que en la anterior prueba, se ha subido el contraste a cien y la saturación a cincuenta. Para las pruebas se ha utilizado una muñeca *Matryoshka* cuyos colores se diferencian del fondo en gran medida. Para la detección se han utilizado técnicas de *tracking* de objetos basadas en color.

8.1 *Backprojection Histogram*

El *Backprojection Histogram* es probablemente la técnica de *OpenCV* más avanzada y útil para la detección por color. Ya hemos hablado de los histogramas durante la prueba uno.

Backprojection Histogram toma como entrada una imagen y el histograma del objeto a buscar en la imagen. La función devuelve otra imagen de las mismas dimensiones que la original, en la que cada píxel se corresponde con la probabilidad de que pertenezca al objeto cuyo histograma se ha dado.

Si por ejemplo se quiere buscar un elemento verde y se aplica este algoritmo, al representar el resultado, veremos que las zonas más claras (con valores más altos) son las que se corresponden en la imagen con tonos de verde muy similares al del objeto que buscamos (o directamente con el objeto, si es que está presente en la imagen).

Para este método, conviene utilizar tanto la imagen como el histograma en color, pues los objetos son más reconocibles y diferenciables por su color que únicamente por su valor en escala de grises.

8.2 *Meanshift*

8.2.1 Introducción teórica

Imaginemos que tenemos una ventana rectangular sobre una imagen. La ventana es de un tamaño dado que no cambia (es una constante). El único parámetro variable en la ventana es su posición en la imagen, que viene dada por las coordenadas *X* e *Y* de su centro. La ventana contiene un determinado número de puntos. Podemos hallar el centroide de dichos puntos contenidos en ella sacando la media de sus coordenadas *X* e Y (Figura 8.1.A).

Llegados a este punto, hay dos opciones: Si el centro de la ventana coincide con el centroide calculado, el algoritmo ha terminado y podríamos decir que ha convergido. Si, de lo contrario, el centro es distinto del centroide, movemos la ventana a la posición del centroide (8.1.B). El centro ahora sería el antiguo centroide y el nuevo

centroide se calcularía de la misma forma en que se calculó el antiguo, es decir, sacando la media de las posiciones de los puntos contenidos en la ventana. El algoritmo continuaría haciendo ésto hasta que alcanzara alguna de las condiciones de parada.

Como se ha mencionado antes, el hecho de que coincidan el centro y el centroide es una condición de parada, pero no es la única. También lo es el que la distancia entre ambos sea menor que un límite pasado como parámetro.

Además, se pueden establecer condiciones de parada en forma de límites de iteraciones. Este tipo de límites se podrían establecer en caso de necesitarse a toda costa un algoritmo rápido y que fuera suficiente con obtener buenas aproximaciones de la solución óptima. Efectivamente, este es un problema de optimización donde la función objetivo a maximizar es el número de puntos contenidos en la ventana y el coste del algoritmo descrito es, en el peor caso, cuadrático con las dimensiones de la imagen. Este algoritmo se llama *Meanshift*.

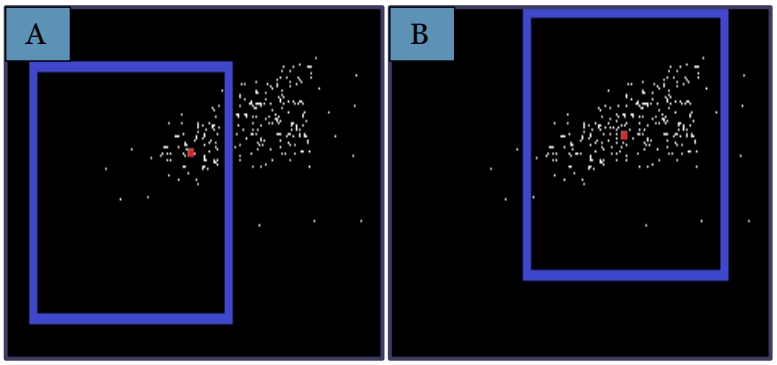


Figura 8.1 Esquema explicativo de Meanshift

8.2.2 Consideraciones sobre *Meanshift*

Aplicación de *Meanshift* a la prueba

Meanshift, toma como entrada una imagen en escala de grises, la cual, en este caso, viene dada por la salida del *Backprojection Histogram* y mueve la ventana hasta que el algoritmo converge en una zona. Ésto es, en este caso, hasta que el algoritmo converge porque el límite de iteraciones especificado es muy alto y la diferencia mínima entre centro y centroide, muy pequeña. Con estas condiciones de parada se ha buscado obtener la solución óptima, tras comprobar que es adecuado en términos de eficiencia.

De un parámetro de entrada del algoritmo *Meanshift* que no hemos hablado es de la ventana inicial. En efecto, *Meanshift* necesita una ventana que mover durante sus iteraciones. Nosotros utilizamos como ventana de entrada la que se devolvió como resultado la ejecución de *Meanshift* en la iteración anterior del bucle principal.

Si durante el bucle principal estamos realizando esta prueba, en cada iteración se calculará el *Backprojection Histogram* de la imagen capturada y el objeto a buscar. Posteriormente se utilizará *Meanshift*, que devolverá la ventana (incluida su posición). Esa ventana es donde debería hallarse el objeto si todo ha salido bien. Durante la siguiente iteración del bucle, con la siguiente captura de la cámara, ocurre lo mismo y como ventana inicial del *Meanshift*, se utiliza el resultado de la iteración anterior.

***Meanshift* es una búsqueda local**

El algoritmo mueve la ventana desde una posición inicial a otra final en la que converge. Sin embargo, lo normal es que no recorra toda la imagen,

sino que converja (para bien o para mal) tras algunas iteraciones. Teniendo en cuenta que estamos analizando vídeo, si en un *frame* tenemos detectado por *Meanshift* un objeto en una determinada posición, en el próximo *frame*, *Meanshift* partirá de dicha posición. De un *frame* a otro el objeto puede haberse movido. La distancia que recorre el objeto depende de la velocidad relativa entre el mismo y la cámara y del tiempo entre *frames*. Es decir, de la frecuencia de muestreo.

8.2.3 Problema con *Meanshift* y la frecuencia de muestreo

Para cada frecuencia de muestreo y objeto hay una velocidad a la que llamaremos velocidad de escape, que sería la velocidad del objeto a partir de la cual, pasa de estar localizado durante un *frame* a dejar de encontrarse en las inmediaciones de la ventana durante el siguiente *frame*. Pasada esta velocidad, la búsqueda local del *Meanshift* resulta inútil, ya que simplemente convergería en algún pequeño máximo local debido al ruido. En ausencia de ruido, *Meanshift* terminaría inmediatamente al no encontrar ningún punto en la ventana.

Por este motivo es tan importante la frecuencia de muestreo también en esta prueba. De hecho, lo es más que en las dos pruebas anteriores, en las que si el objeto se hallaba en la imagen, era detectado. En ésta, solo se detecta si se halla cerca de su posición en el *frame* anterior.

Se han conseguido una frecuencia de muestreo media de diez ciclos por segundo. Por supuesto es insuficiente cuando el robot está en movimiento y éste pierde su objetivo en repetidas ocasiones (en giros en los que la velocidad relativa supera a la de escape).

8.2.4 Soluciones al problema con *Meanshift* y la frecuencia de muestreo

Eliminación de ruido

Lo primero que se ha utilizado, viendo los buenos resultados que ya nos dio en el apartado anterior es la erosión morfológica para la eliminación de ruido, en este caso sobre el resultado de *Backprojection Histogram*.

Para eliminar el ruido proveniente de zonas en las que coincide el color, pero están poco saturadas o tienen poco brillo (tonos de gris, sombras, etc.) se ha usado el *imrange* sobre la imagen original en *hsv* y el resultado se ha aplicado como máscara sobre el resultado de *Backprojection*.

Distinción de las diferentes situaciones posibles

Tras conseguir una imagen relativamente libre de ruido, se ha construido un algoritmo considerando los tres casos que pueden darse:

1. El objeto está presente en la imagen y será detectado por *Meanshift* por estar en las inmediaciones de la ventana.
2. El objeto está presente en la imagen, pero no será detectado por *Meanshift* por encontrarse lejos de la ventana.
3. El objeto no está presente en la imagen. Se ha perdido de vista brevemente como ya pasaba en las pruebas anteriores

Lo importante es que el algoritmo sepa distinguir cuando se encuentra en cada uno de los casos para poder actuar en consecuencia. Al utilizar la salida del *Backprojection*, que es una imagen en escala de grises, es fácil saber

si el objeto está presente o no en la original. Si el máximo valor de la salida del *Backprojection*, es menor que un límite, quiere decir que el objeto no está en la imagen.

Recordemos que el valor de un píxel es la probabilidad de que pertenezca al objeto que se busca. Si la probabilidad máxima es baja, significa que el objeto no está presente.

Si dicha probabilidad es alta, todavía nos queda distinguir entre los dos primeros casos. Para ello, además del máximo global, obtenemos el máximo dentro de la ventana. Si éste es igual o similar al global, quiere decir que parte del objeto se encuentra en la ventana y que, por lo tanto, nos hallamos en la opción uno. En ese caso aplicamos *Meanshift* sin más, con la certeza de que hallará el objeto. Si, por el contrario, el máximo de la ventana es mucho menor que el global, quiere decir que el objeto se encontrará en otro sitio fuera de la ventana y que, aun estando presente en la imagen, no será hallado por *Meanshift*. Es la opción dos. En este caso, movemos la ventana a la posición en la que se encuentra el máximo y llamamos al *Meanshift*, de esta forma, forzamos a que el algoritmo se corrija a si mismo cuando es necesario.

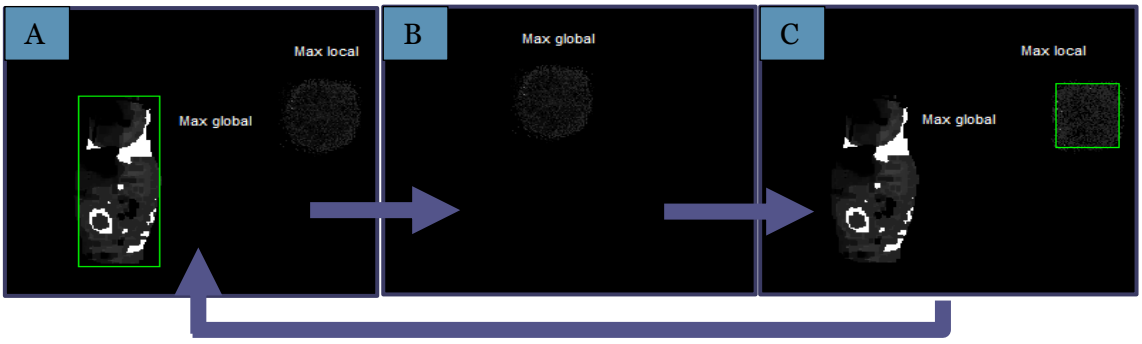


Figura 8.2 Solución al problema de perder de vista el objeto

La Figura 8.2 muestra una de las situaciones más comunes que requieren de la solución que acabamos de explicar. En un principio, la ventana está encima de la muñeca, por lo que se está detectando (Figura 8.2.A).

A continuación, se aparta la muñeca bruscamente hacia un lado. El robot empieza a girar, pero pierde de vista brevemente a la muñeca (Figura 8.2.B). En este momento, como el máximo global es pequeño (el perteneciente al ruido), no aplica *Meanshift* y descarta las capturas.

Cuando el robot por fin ha girado lo suficiente como para volver a ver la muñeca, el máximo global es alto, así que, en este caso, habría que aplicar *Meanshift*. Sin embargo, en las inmediaciones de la ventana solo hay ruido, por lo que, si se aplicara *Meanshift* en este momento, convergería en el ruido (Figura 8.2.C).

Aun así, se comprueba que el máximo global, que se corresponde con la muñeca, es mucho mayor que el local, que se corresponde con el ruido. Por ello se mueve la ventana inicial a la posición del máximo global, tras lo cual se aplica *Meanshift*, obteniendo una vez más la situación deseada (Figura 8.2.A).

8.2.5 Problema con *Meanshift* y la ventana inicial

Meanshift toma como entrada la ventana de salida de la iteración anterior del bucle principal, pero para la primera iteración, necesitamos definir una ventana inicial, o lo que es lo mismo, una noción aproximada de dónde va a estar el objeto inicialmente.

Además, necesitamos una imagen modelo del objeto a buscar, para poder aplicar el *Backprojection Histogram*.

8.2.6 Posibles soluciones al problema con *Meanshift* y la ventana inicial

Reconocimiento previo con *BRISK*

El inicio del proceso consistía en cargar una foto en color del objeto a buscar y hallar su histograma. Seguidamente, se convertía a escala de grises y se buscaba en el primer *frame* de la cámara con métodos de detección de esquinas. Concretamente se calculaban sobre ambas los puntos clave y los descriptores con *BRISK* para obtener las coincidencias con un *BruteForceMatcher*.

Estas técnicas se explican en profundidad en la siguiente prueba. Aplicando este algoritmo, se detectaba la posición inicial del objeto sin ningún problema, que era el objetivo.

Introducción manual de la ventana inicial

Se pensó que sería más interesante no cerrarnos al seguimiento de un objeto, sino que éste lo especificara el usuario al principio de la ejecución.

Para ello, lo ideal no era usar detección de esquinas sobre fotos ya cargadas, sino que el usuario pudiera especificar él mismo la ventana inicial sobre el primer *frame*. Definitivamente se ha hecho así. El primer *frame* justo después de cambiar de modo al de esta prueba, se queda congelado esperando a que el usuario, mediante tres clics de ratón (que se corresponden con la posición de la esquina superior izquierda, anchura y altura respectivamente) defina la ventana inicial. Para ello se ha programado una función que salta cada vez que ocurre el evento de que el usuario hace clic sobre el primer *frame*.

8.3 *Meanshift* vs *Camshift*

Aunque durante todo el apartado se ha estado hablando de *Meanshift*, lo cierto es que no es el algoritmo que se ha utilizado finalmente. *Meanshift* tiene dos pequeñas debilidades: El tamaño y la rotación. No importa lo cerca o lejos que esté el objeto de la cámara, si usamos *Meanshift*, la ventana siempre va a tener el mismo tamaño (Figura 8.3.A). Ésto no es adecuado, ya que buscamos que la ventana se ajuste al tamaño del objeto, sin ser demasiado grande cuando el objeto está lejos (Figura 8.3.B) ni demasiado pequeña cuando está cerca (Figura 8.3.C). Por otra parte, con la rotación ocurre lo mismo. El objeto puede rotar, pero la ventana de *Meanshift* nunca lo hará.

Camshift es la mejora de *Meanshift* que soluciona estos dos inconvenientes. *Camshift* empieza aplicando *Meanshift*, tras lo cual escala la ventana y la rota, para luego volver a usar *Meanshift*, y a escalar y rotar la ventana. Así hasta que el algoritmo converge y termina sus iteraciones. El resultado es una ventana que se ajusta en posición, tamaño y ángulo a nuestro objeto (Figura 8.3.D), (Figura 8.3.E), (Figura 8.3.F).

En cuanto a la finalización del algoritmo, en caso de que el objeto estuviera presente en el *frame* y que, por lo tanto, se hubiera ejecutado el *Camshift*, se obtendría el centro de la ventana rotada que devuelve. Con la coordenada *Y* de ese centro, se utiliza la función para subir y bajar la cámara y adecuarla a la altura o distancia del objeto. La coordenada *X* del centro se pasa como punto de interés al control de motores.

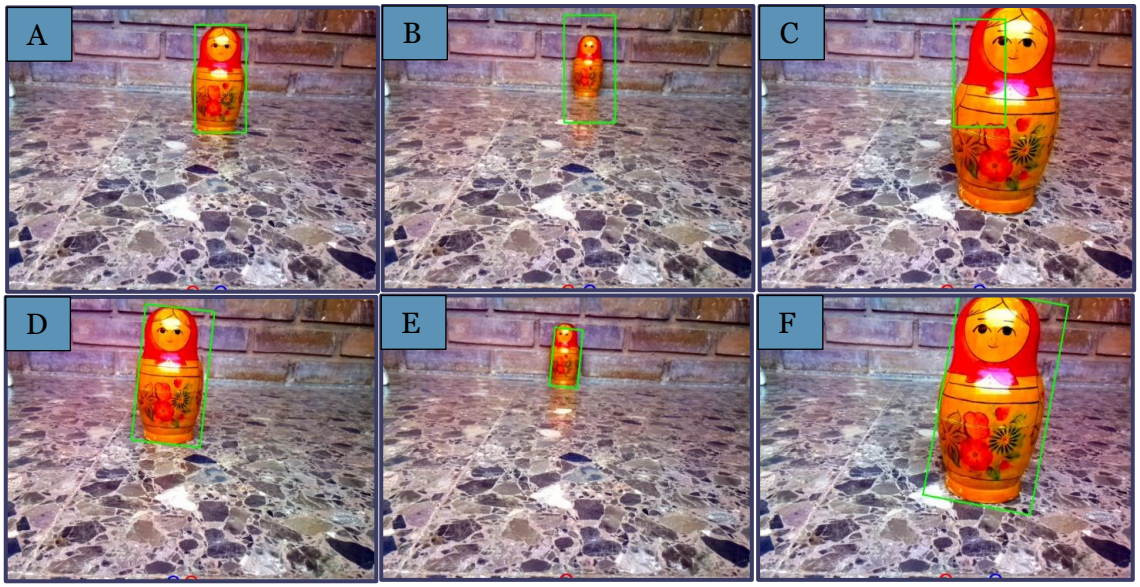


Figura 8.3 Meanshift VS Camshift

8.4 Resumen del algoritmo (Figura 8.5)

Al inicio de la ejecución, se sube el contraste y la saturación y el usuario selecciona la ventana que contiene el objeto a seguir. Se obtiene el histograma en color de dicha ventana. Para cada iteración del bucle principal, se calcula el *Backprojection Histogram* sobre la imagen y el histograma del objeto. Se aplica *imrange* en la imagen sin procesar y se utiliza el resultado como máscara sobre la imagen en escala de grises que devuelve *Backprojection Histogram*. Sobre ella se aplica una erosión morfológica para eliminar ruido y se obtiene el máximo global y de la ventana. Si el máximo global es menor que un valor, el *frame* se ignora. Si no, en caso de que el máximo global sea similar al de la ventana, se aplica *Camshift* directamente. De lo contrario, se mueve la ventana y se aplica *Camshift*. El centro de la ventana resultado de *Camshift* se utiliza para controlar los motores y la inclinación de la cámara.

8.5 Resultados

Figura 8.4 Resultado final

Los resultados obtenidos han sido buenos, como puede apreciarse en el vídeo. Del mismo modo que en el resto de pruebas, se probó con dos algoritmos de control: El control proporcional modificado y el proporcional derivativo.

Sin embargo, en este caso, el control proporcional no funcionaba bien. Esta prueba, debido a la localidad del *Camshift*, era más sensible a los cambios bruscos y las oscilaciones que a veces produce el control proporcional, haciendo que el robot tuviera que relocalizar continuamente el objeto en la imagen.

El control proporcional derivativo, mucho más suave y adaptativo, ha conseguido reducir estos casos en los que hay que mover la ventana al máximo global a giros bruscos y momentos puntuales.

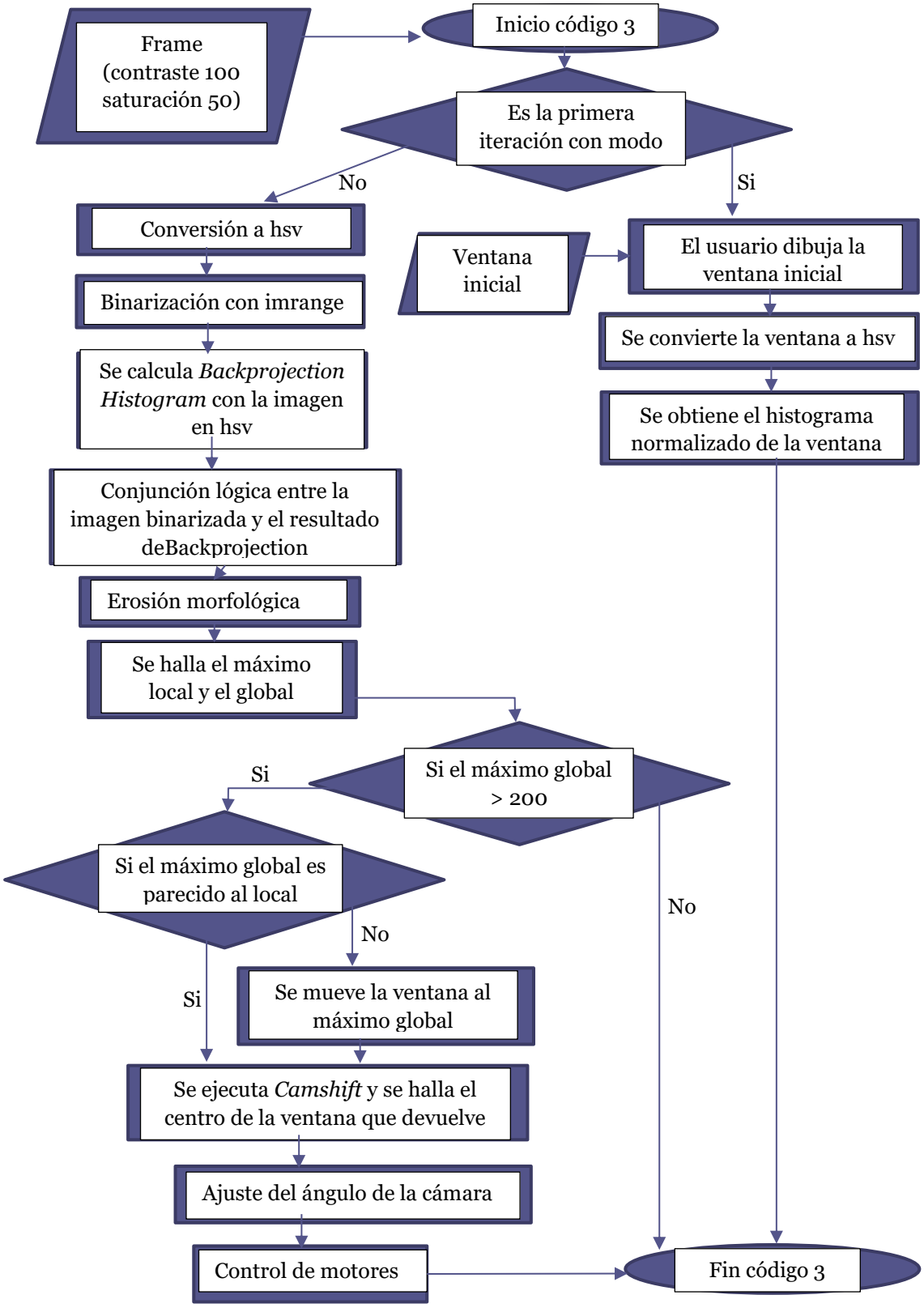


Figura 8.5 Flujo del código específico del modo 3

https://youtu.be/gwdqmX1SpHw

9. Prueba 4: Distinción de objetos por detección de características

En esta prueba se colocan en circulo varias carátulas de películas en formato de vídeo para que el robot gire y detecte una de ellas. Tras reconocerla, el robot se dirigirá a ella. Se han elegido arbitrariamente cinco películas, aunque el algoritmo funcionaría igualmente con cualquier otro conjunto de objetos.

La *Raspberry* tiene almacenada una foto de cada portada. Al iniciar el modo, la película a buscar puede ser seleccionada por el usuario o por el propio programa. Si se decide que la elección la haga el programa, este simplemente decidirá aleatoriamente. Posteriormente se carga en escala de grises la foto correspondiente a la película elegida.

Desde este momento, que ya tenemos una imagen a buscar (la portada que hemos cargado), empieza realmente el algoritmo de esta prueba.

Al estar las películas en círculo, el robot tiene que ir rotando para encontrarla. Hay que recalcar que todos los giros realizados en esta prueba son sobre su propio eje. Es decir, giros de tipo dos.

- En primer lugar, el robot gira entre uno y dos segundos hacia la derecha o la izquierda. Se trata de una maniobra aleatoria que sirve para que el punto de partida no sea siempre el mismo.

- En cada iteración del bucle, el robot toma una imagen, la analiza y gira (hacia la izquierda) durante un tiempo previamente establecido. El ángulo de giro es aproximadamente de 45 º, el cual, teniendo en cuenta que el ángulo de apertura horizontal de la cámara es de 62.2 º, es suficientemente pequeño como para no dejar direcciones sin cubrir entre dos iteraciones.

9.1 La detección de objetos por características en *OpenCV*

La visión se ha realizado mediante métodos de detección por esquinas, que se utilizan cuando se tienen dos imágenes y se quiere encontrar una de ellas en la otra.

Aunque no es nuestro caso, también se puede usar para hallar una imagen en un vídeo en tiempo real, aplicando el método para cada *frame*. Se dividen en tres fases: detección de puntos clave (o esquinas), cálculo de los descriptores y cálculo de los *matches*. La primera y la segunda se hacen de forma separada para las dos imágenes. La tercera, se hace una vez de forma conjunta para ambas. Obviamente los tres pasos son algoritmos que pueden aplicarse de forma separada, aunque en gran medida están pensados para actuar de forma conjunta en este tipo de situaciones.

9.1.1 La detección de esquinas

Parte de la idea de que la mejor forma de distinguir una imagen es por sus esquinas. Si por ejemplo tomamos un fragmento de la frente de una persona, éste puede ser uniforme y común a muchas otras, es decir, no será un aspecto distintivo, que es lo que buscamos. Si, sin embargo, utilizamos las esquinas, como los ojos o la boca, será mucho más sencillo distinguir a la persona.

Los algoritmos de detección de esquinas tienen como salida un conjunto de puntos clave, en los que se han detectado las esquinas.

Algoritmo de Harris

Formalmente, consideramos esquina a un fragmento de una imagen, cuando en éste hay una variación alta de gradiente en todas direcciones.

Esta definición fue la que Harris expresó de forma matemática en uno de los primeros y más conocidos algoritmos de detección de esquinas, que toma una imagen en escala de grises, aplica las derivadas de *Sobel* para hallar el gradiente en cada uno de los fragmentos, seleccionando aquellos que se ajustan a la definición anterior. El algoritmo de Harris es el más sencillo en el ámbito de la detección de esquinas, y, de hecho, se utiliza mucho como parte de otros algoritmos más elaborados. Sin embargo, no se adapta bien a cambios de tamaño, dando resultados distintos para una imagen y para una transformación de escalado de la misma.

Algoritmo *FAST*

Uno de los algoritmos más eficientes e utilizados es el *FAST* (*Features from Accelerated Segment Test*). Lo que hace al algoritmo tan rápido es la sencillez de sus cálculos, ya que trabaja únicamente con las intensidades de los píxeles del vecindario de cada píxel, sin necesidad de aplicar gradientes. Además, está pensado para poder (de forma completamente opcional) estimar sus parámetros mediante el uso del aprendizaje automático, adecuándolos al ámbito de las imágenes que se van a tratar.

9.1.2 El cálculo de descriptores

Aunque los puntos clave son aquellos que definen las imágenes y que por lo tanto utilizaremos para hallar los *matches* de ambas imágenes, su estructura es la de un punto cualquiera. Solo contienen su posición, por lo que los puntos en si no ofrecen la información necesaria para calcular los *matches*. Para ello necesitamos calcular los descriptores, es decir, las características de las regiones próximas a los puntos clave. Cada descriptor calculado se estructura como un vector de características cuya longitud depende del algoritmo. Todos estos vectores se devuelven como filas dentro de una misma matriz, que es el resultado de cualquier algoritmo de este tipo.

Con los descriptores, sí que se podrá decidir qué puntos clave entre ambas imágenes tienen parecido y por lo tanto calcular los *matches*. Los diferentes algoritmos para el cálculo de descriptores se diferencian entre ellos por las dimensiones de sus vectores de características, de lo cual, además, depende el espacio ocupado en memoria, así como de la información que contienen dichos vectores y el tamaño del vecindario que usa alrededor de cada punto clave para hallar su descriptor.

Algoritmo *BRIEF*

Uno de los algoritmos dedicados exclusivamente al cálculo de descriptores y que además es de los más conocidos y eficientes es el *BRIEF* (*Binary Robust Independent Elementary Features*). Los descriptores de *BRIE*F están en la compacta forma de una cadena binaria. Para reducir el coste en tiempo del algoritmo, las cadenas no son halladas aplicando algún algoritmo de compresión a los descriptores en coma flotante, sino que *BRIEF* las calcula directamente, eliminando aquellas características inservibles.

9.1.3 Los detectores-descriptores

Dada la frecuencia con la que la detección de puntos clave y el cálculo de descriptores se realizan juntos, existen toda una gama de métodos detectores-descriptores que nos permiten hacer ambos pasos. De hecho, todos ellos contienen la función *detectandcompute* que permite ejecutar los dos pasos directamente. Vamos a mencionar varios de estos métodos, parándonos solamente en los que hemos utilizado.

Como se ha comentado antes, el detector de esquinas de Harris es deficiente con los cambios de escala. Lowe, pensando en un algoritmo que subsanara dicho problema, creó *SIFT* (*Scale-Invariant, feature detection*). Sin embargo, *SIFT* no es eficiente en tiempo, ya que hace cálculos pesados durante la detección. Es por ello que apareció *SURF* (*Speeded-Up Robust Features*) como alternativa más ligera. *SIFT* tampoco es eficiente en espacio y *SURF* mejora también a *SIFT* en este sentido. No obstante, los descriptores de ambos tienen tamaños innecesariamente grandes, que pueden contener características irrelevantes para la detección, ocupando espacio en balde.

Aunque tanto *SIFT* como *SURF* ofrecen buenos resultados, es cierto que ambos utilizan filtros Gaussianos a diferentes escalas, lo cual hace que la imagen entera se difumine y pueda perderse exactitud en la detección. Para soslayar este inconveniente se creó el algoritmo *KAZE*, que utiliza filtros Gaussianos locales, es decir, aplica los filtros de forma selectiva eliminando el ruido a la vez que mantiene definidos los contornos. Sin embargo, *KAZE* es muy ineficiente.

Desde la aparición de dispositivos móviles con cámara, se ha disparado el número de aplicaciones que utilizan la detección de objetos en tiempo real. A medida que fue surgiendo esta necesidad, lo hicieron también distintos algoritmos que trataban de cubrirla. *AKAZE* (*Accelerated KAZE*) nace como una versión eficiente de *KAZE*. Por otra parte, tanto *ORB* (*Oriented FAST and Rotated BRIEF*) como *BRISK* (*Binary Robust Invariant Scalable Keypoints*), combinan el detector *FAST* y el descriptor *BRIEF* mencionados anteriormente, manteniendo la tolerancia a escalados y rotaciones.

9.1.4 El cálculo de *matches*

Una vez dados los vectores de características (o descriptores) de ambas imágenes, debemos hallar los *matches* entre ellas. De forma simplificada, consiste en calcular las distancias entre los vectores de características de ambas imágenes, es decir, se compara cada descriptor de la primera imagen (imagen A) con todos los de la segunda (imagen B). Esto es lo que se llama cálculo de *matches* por fuerza bruta. El matcher es un objeto que tiene dos funciones:

Match

Para cada punto de la imagen A calcula el mejor *match* con la imagen B. Además, para eliminar los falsos positivos, *Match* contiene un parámetro llamado *crosscheck*. Cuando se pone a *true* este parámetro, se calculan los *matches* de A con B y los de B con A y solo se toman aquellos que coinciden para ambos casos.

Por ejemplo, si tomamos un descriptor 1 de A y obtenemos el descriptor 2 de B como resultado al mejor *match* del descriptor 1 con B, con el *crosscheck* activado, tendremos que calcular también el mejor *match* del descriptor 2 con A. Si el resultado de este segundo cálculo fuera el descriptor 1, querría decir que el *match* entre ambos descriptores es correspondido y que por tanto se incluye en el resultado. De lo contrario se descartaría.

KnnMatch

La función Knnmatch calcula los K mejores *matches* de cada punto clave de A con B.

Concretamente utilizamos *K=2* para hallar los dos *matches* de menor distancia de cada descriptor, ya que, utilizar una *K>2* hace el algoritmo menos eficiente sin aportar nada, mientras que una *K=1* no permite aplicar la técnica de eliminación de falsos positivos que describimos a continuación.

Knnmatch no ofrece crosscheck, por lo que la eliminación de falsos positivos se hace otra forma. La idea se basa en que un match verdadero es especial y único y debería diferenciarse mucho de otros posibles *matches* peores de ese descriptor. De esta forma, solo se admitirá el mejor de los dos *matches* si su distancia es considerablemente más pequeña que la del segundo mejor match.

FlannBasedMatcher

Otro matcher, que vamos simplemente a mencionar y que ofrece resultados similares al de fuerza bruta, resultando más eficiente en situaciones con muchos descriptores, es el *FlannBasedMatcher*.

9.2 Herramientas escogidas para la prueba

9.2.1 Comparación entre los detectores-descriptores

De entre los diferentes detectores-descriptores expuestos (*SIFT*, *SURF*, *KAZE*, *AKAZE*, *ORB* y *BRISK*), se decidió directamente no utilizar *SIFT* ni *SURF* debido a que ambos están patentados y se necesita pagar para poder utilizarlos. Tras descartar estos dos algoritmos, nos quedan los otros cuatro. Hay que recalcar que la eficiencia del algoritmo de visión utilizado en esta prueba no es una prioridad, pues, de hecho, no se trata de detección en tiempo real porque se toma únicamente un *frame* tras cada giro, lo cual es tiempo de sobra para cualquiera de los detectores-descriptores mencionados anteriormente.

Para poder escoger adecuadamente el algoritmo, se han probado en las mismas condiciones los cuatro utilizando para ello la portada de un libro.

KAZE nos ofreció 112 *matches* (Figura 9.1.A). Puede parecer una cifra alta, sobre todo teniendo en cuenta que muchos de estos *matches* son reales. No obstante, también se puede apreciar que gran cantidad de ellos proceden de puntos clave situados sobre las letras o sobre el contorno de la portada. La mayoría de puntos clave no se han detectado en el cuerpo de la portada, sino en partes secundarias, lo cual sería una diferenciación muy pobre frente a otros libros negros con letras blancas.

ORB obtuvo únicamente 9 *matches* (Figura 9.1.B). Es una cantidad minúscula si la comparamos con la obtenida por el resto de algoritmos. Como podemos ver, *ORB* apenas ha detectado puntos clave y los pocos que ha detectado difieren en ambas imágenes o se corresponden, una vez más, con las letras.

AKAZE se encuentra en un término medio con 65 *matches* (Figura 9.1.C). Como se puede observar, los puntos clave detectados por este algoritmo son parecidos a los de *KAZE*. Sin embargo, ha detectado muchos menos en las letras y los bordes de la portada. Por ello, pese a tener menos *matches*, podríamos decir que estos son de mejor calidad que los de *KAZE*. Recordemos, además, que *AKAZE* es un algoritmo más eficiente que *KAZE*.

BRISK ha sido claramente el mejor de los cuatro. Ha hallado 140 *matches*, de los cuales, la cantidad de falsos positivos es despreciable respecto al número total.

Hay que recalcar que los *matches* obtenidos por *BRISK* son de calidad, ya que provienen en su mayoría de una gran cantidad de puntos clave hallados en el cuerpo de la portada. Solo unos pocos se corresponden con las letras y el contorno.

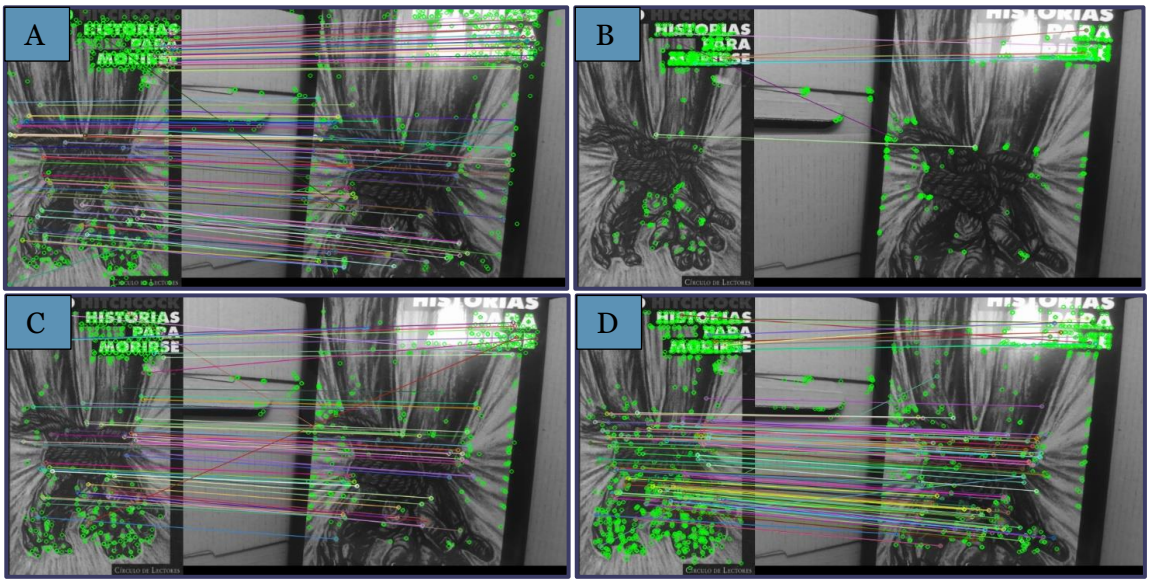


Figura 9.1 Comparación entre los detectores-descriptores

9.2.2 Comparación entre las funciones de *BFMatcher*

Para empezar, elegimos el matcher de fuerza bruta sobre el *Flann*. El *Flann* es un método aproximado y solo ofrece ventajas en términos de eficiencia, lo que no se encuentra entre los requisitos de esta prueba. Dentro del *matcher* de fuerza bruta, se realizaron cuatro pruebas en exactamente las mismas condiciones partiendo del mismo conjunto de puntos clave y descriptores.

En primer lugar, se ejecutó un matcher de fuerza bruta sin filtrar de ninguna forma los *matches* (Figura 9.2.A). Se obtuvieron 1147 *matches*, es decir, una cantidad excesiva, siendo la mayoría de ellos falsos.

Para limitar el número de *matches*, se aplicó la técnica del *crosscheck*, que resultó ser una mejora sustancial pero insuficiente, obteniendo 427 *matches* (Figura 9.2.B), muchos de los cuales seguían siendo falsos.

Otra de las pruebas, fue aplicar la función *Knnmatch* con *K=2* y un ratio de 0.75. Los resultados fueron muy positivos, obteniendo 140 *matches*, la mayoría de ellos reales (Figura 9.2.D). De los *matches* eliminados por el ratio, un 95% son falsos positivos, mientras que solo un 5% son reales.

Por último, se decidió probar *crosscheck* y quedarnos con los 140 mejores *matches*, es decir, de menor distancia, (Figura 9.2.C), simplemente para compararlo en igualdad de condiciones con *Knnmatch*. Se obtuvieron resultados similares, aunque claramente detecta más *matches* falsos que *Knnmatch* con ratio.

En vista de los resultados obtenidos, la decisión ha sido sencilla. La mejor opción es *Knnmatch* con ratio.

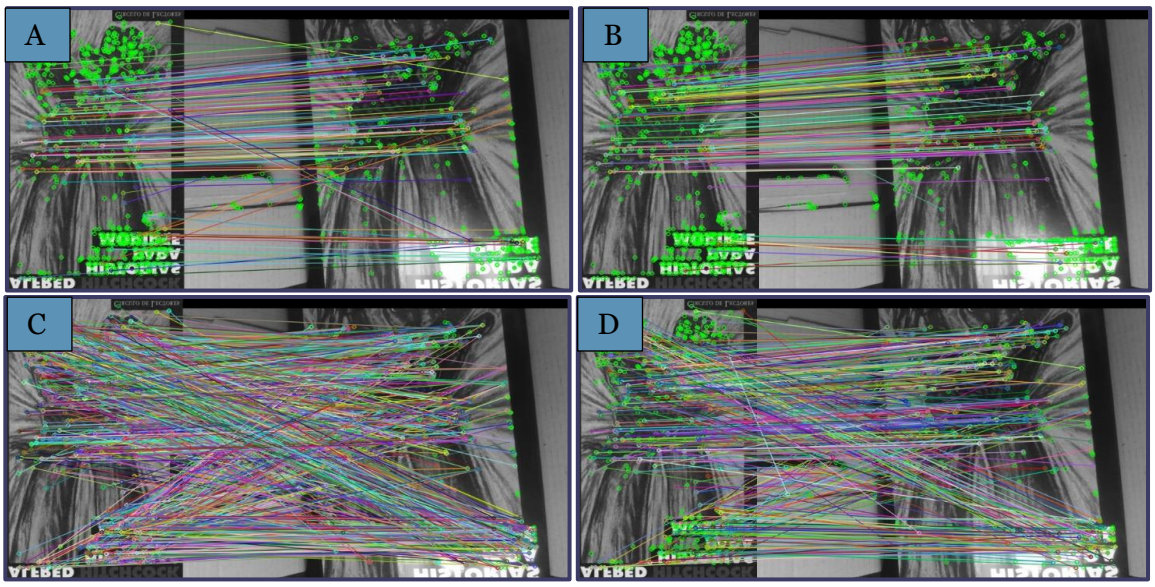


Figura 9.2 Comparación entre las funciones de BFMatcher

9.3 Estrategias para buscar el objeto

Obviamente, el robot debe detenerse en algún momento frente a la película detectada. Para esto hemos probado dos estrategias.

9.3.1 Búsqueda por mejor captura

El robot va girando hasta que da una vuelta completa, es decir, hasta que termina ocho iteraciones, pues en cada una gira 45 º. A cada una de las ocho capturas, le asigna una puntuación (cuanto más alta mejor). Dicha puntuación es directamente proporcional a la cantidad de *matches* entre ambas imágenes e inversamente proporcional a la media de las distancias de los *matches*.

Tras la vuelta completa, el algoritmo escoge la mejor puntuación y gira hacia la izquierda lo necesario para dar con ella. Es decir, si la captura ganadora se había tomado tras tres giros de 45º desde la posición inicial, tras dar la vuelta completa, el robot vuelve a estar en la posición inicial, así que solo tiene que volver a dar tres giros para hallarse de frente con su elección.

9.3.2 Búsqueda por primera captura válida

El robot va girando hasta que se cumple una determinada condición que hace que tome el objetivo actual como solución y se detenga frente a él. En nuestro caso, la condición se basa solo en superar un determinado número de *matches*. Nos dimos cuenta de que cuando aplicábamos la visión sobre una imagen que no contiene el objeto que buscamos, el número de *matches* es bajo, siempre menor a 30 (Figura 9.3.A), (Figura 9.3.B), (Figura 9.3.C), (Figura 9.3.D), (Figura 9.3.E), (Figura 9.3.F), (Figura 9.3.G), mientras que cuando estamos viendo el objeto correcto, hay una cantidad alta de *matches*, siempre superior a 70 (Figura 9.3.H).

Es por ello que nuestro límite es 50, que sería el separador de margen máximo. Evidentemente, este algoritmo no siempre da una vuelta completa, lo cual tiene ventajas y desventajas.

Lo positivo es que puede completar la misma prueba que el anterior en menos tiempo.

Lo negativo es que bajo determinadas condiciones en las que los objetos a distinguir se asemejen mucho entre ellos, podríamos dar con un falso positivo. El hecho de poder comparar todos los objetos para elegir el mejor (como sí hace el anterior), puede ser determinante a la hora de acertar en la elección del objeto.

Se ha utilizado este segundo algoritmo para distinguir películas de la misma colección y hemos obtenido muy buenos resultados. Eso quiere decir, que para que el algoritmo fallase, los objetos tendrían que ser muy parecidos. Aún en ese caso, podría modificarse la condición, adecuando el límite o dividiéndolo entre la distancia media, lo que lo convertiría en un límite más informado y que debería ofrecer mejores resultados.

Con todo esto, en determinadas situaciones en las que haya que distinguir entre objetos prácticamente iguales, puede ser más conveniente utilizar el primer algoritmo.

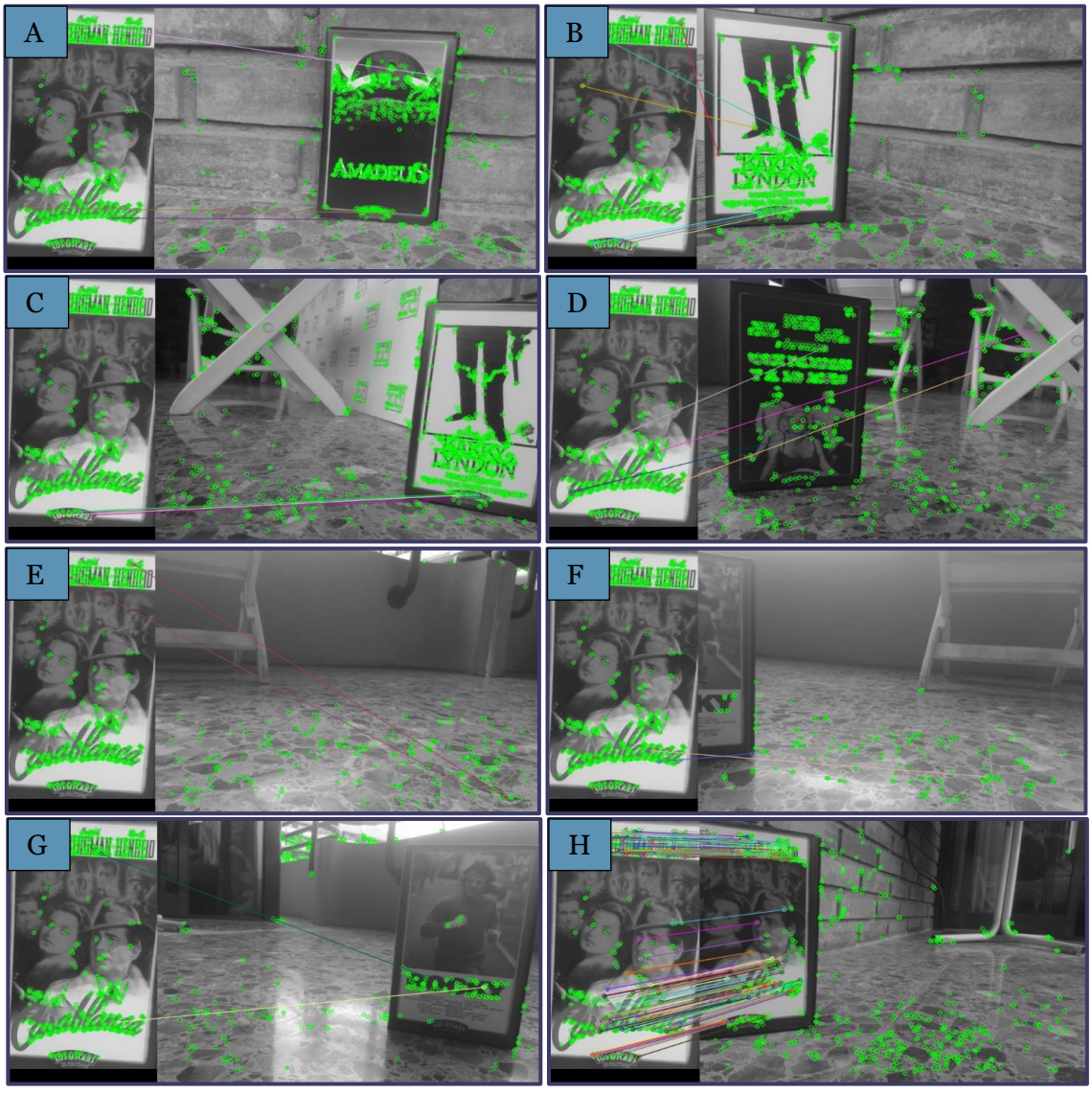


Figura 9.3 Ejemplo de resultado con vuelta completa

9.4 Centrado de la imagen y fin de la prueba

9.4.1 Obtención del punto de interés

Una vez el robot se ha detenido delante del objetivo, tenemos que centrarlo para que se dirija recto hacia él. Como en otras pruebas, obtenemos el punto de interés como el centro del rectángulo rotado que contiene el objeto. En este caso, definimos el objeto por sus puntos clave que además han hecho *match* con la imagen original.

Simplemente se obtiene el rectángulo rotado que engloba dichos puntos. Se pensó también en utilizar, como es muy común tras los algoritmos de detección de esquinas, la homografía inversa para hallar un contorno detallado del objeto a partir de los *matches*. Sin embargo, este método es más costoso e ineficiente y en nuestro caso no necesitamos un resultado tan preciso. Solo necesitamos el centro del objeto y ,para ello, el centro del rectángulo rotado es una más que buena aproximación.

9.4.2 Control a partir del punto de interés

Si la distancia horizontal de nuestro centro al punto de interés es menor que un límite, quiere decir que estamos justo enfrente del objeto y seguimos recto. Si dicha distancia es mayor que ese límite, debemos girar hacia el lado adecuado dependiendo de si el objeto está a la izquierda o a la derecha.

De esta forma se configura una especie de control de motores que se ha diseñado específicamente para esta prueba. Cada giro, al igual que todos los de esta prueba, es de tipo dos y se hace durante un determinado tiempo.

Si tras el giro, el objeto sigue estando hacia el mismo lado (Figura 9.4.B), (Figura 9.4.C), volvemos a girar. Si, en cambio, nos hemos pasado de largo y hallamos el objeto al otro lado (Figura 9.4.D), giramos hacia el otro lado, pero durante la mitad de tiempo que lo hacíamos anteriormente para no pasarnos como antes. El algoritmo va haciendo giros cada vez más pequeños hasta que converge con el objetivo centrado y se dirige recto hacia él.

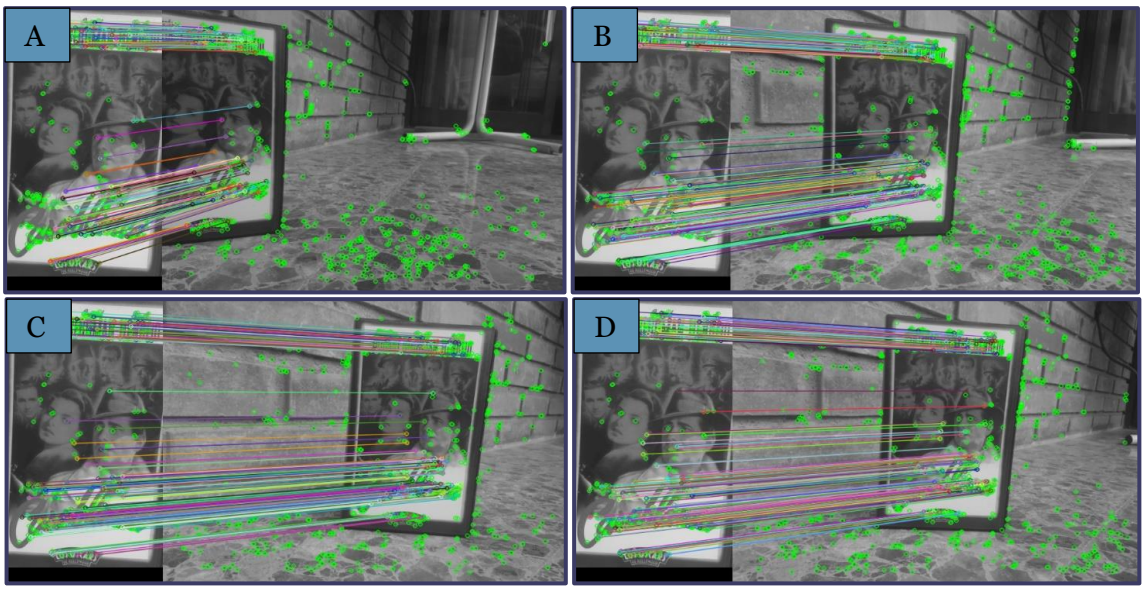


Figura 9.4 Centrado del objetivo

9.5 Resumen de la prueba

Al empezar la prueba se elige y carga en escala de grises la foto de una de las cinco películas disponibles. Se da una vuelta aleatoria para empezar desde cualquier sitio. En cada iteración se transforma el *frame* captado a escala de grises, se detectan los puntos clave y se calculan los descriptores en la imagen cargada y en la captada y se hallan los *matches* entre los descriptores de ambas imágenes. Si no hay suficientes *matches*, giramos 45 º y pasamos a la siguiente iteración. Este proceso se repite hasta que se detecta el objeto. Cuando eso ocurre, el robot maniobra hasta centrar dicho objeto en la imagen y posteriormente se dirige hacia él. La Figura 9.5 muestra el funcionamiento del algoritmo de forma esquemática mediante un diagrama de flujo

9.6 Resultados

Finalmente, los resultados de esta prueba han sido también positivos, como se puede apreciar en el vídeo, para el cual se ha deshabilitado la vuelta aleatoria que se hace al principio y se ha elegido la película Casablanca, con el objetivo de asegurarnos de que el robot de toda la vuelta. El objetivo de esto es poder mostrar capturas de los *matches* con todas las películas, demostrando el contraste de la cantidad de *matches* entre la película buscada y las otras cuatro.

Las capturas tomadas por el robot durante el video de esta prueba, se corresponden con las Figuras 9.3 y 9.4

https://youtu.be/eh0sbGLFp1Q

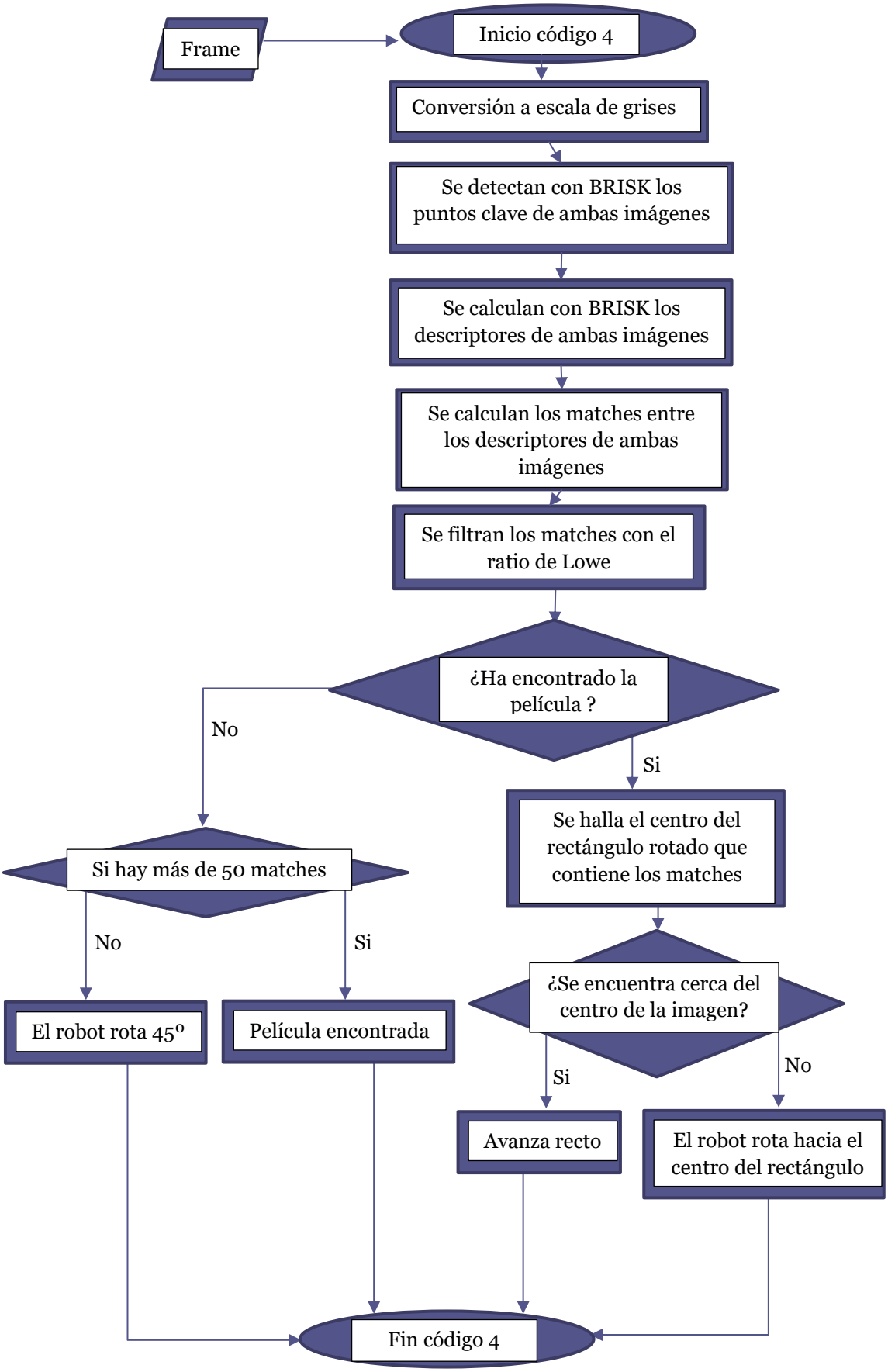


Figura 9.5 Flujo del código específico del modo 4

10. Prueba 5: Distinción de señales por *Template Matching*

Esta prueba consiste en el reconocimiento de señales gracias a la técnica del *Template Matching*. Además, el robot tiene que reaccionar adecuadamente a lo que detecta. En este caso hemos establecido cuatro tipos de señales de tráfico: cono (obstáculo que debe esquivar), giro a la derecha, giro a la izquierda y stop.

10.1 *Template Matching*

10.1.1 Introducción teórica

El *Template Matching* tiene un funcionamiento relativamente sencillo en comparación con otros algoritmos que hemos visto. Las entradas son: la imagen, el *template* (que es otra imagen) a buscar en la imagen y el método de *Template Matching* que se quiere utilizar. Hay seis métodos que básicamente se diferencian en la fórmula matemática que utilizan para calcular la similitud entre los píxeles del *template* y la imagen. Estos son: *CCOEFF, CCOEFF NORMED, CCORR, CCORR NORMED, SQDIFF y SQDIFF NORMED*. El *Template Matching* desliza la ventana, que es el *template*, por toda la imagen aplicando en cada posición el método especificado.

Si llamamos W y H a la anchura y la altura de la imagen y w y h a la anchura y la altura del *template* respectivamente, la cantidad de ventanas posibles sería de $(W-w+1)*(H-h+1)$, que es la cantidad exacta de píxeles que contiene la imagen que devuelve el *Template Matching*. Dicha imagen de salida está en escala de grises y cada uno de sus píxeles contiene un valor que da una medida de la probabilidad de que ese píxel sea el vértice superior izquierdo del template. Cuanto más grande sea el valor de un píxel, mayor es la probabilidad. Excepto en *SQDIFF* y *SQDIFF NORMED*, que funciona al revés: Cuanto menor es el valor, mayor es la probabilidad.

10.1.2 Método de matching escogido

Los algoritmos que dan mejores resultados son *CCOEFF* y *CCOEFF NORMED*, mientras que *CCORR* es el más eficiente y también el que da peores resultados.

De los seis métodos se ha elegido *CCOEFF NORMED* por resultar ser el mejor empíricamente y porque, frente a *CCOEFF*, presenta la ventaja de que sus valores se encuentran acotados de cero a uno, que es directamente la probabilidad y por lo tanto una medida entendible de la misma.

10.1.3 El problema de eficiencia con *Template Matching*

Tras algunas ejecuciones de esta prueba, reparamos en que la frecuencia de muestreo era inaceptablemente baja. El *Template Matching* es relativamente pesado (incluso con *CCORR* que es el método más simple). Además, en cada iteración del bucle principal teníamos que aplicarlo cuatro veces (una por cada señal), obteniendo una frecuencia de muestreo de 0.75 frames por segundo, o lo que es lo mismo, un *frame* cada 1.33 s. Con esta frecuencia el robot no era capaz de reaccionar y la prueba no era factible.

10.1.4 Alternativas para aumentar la frecuencia de muestreo

Conversión a escala de grises

Lo primero que se hizo fue usar un *template* y una imagen en escala de grises. Como una imagen en escala de grises tiene un canal, mientras que una en color tiene tres (rojo, verde y azul), conseguimos el triple de frecuencia de muestreo. Es decir, 2.25 fps, lo cual seguía siendo poco, sobre todo, teniendo en cuenta que en las tres primeras pruebas conseguimos más de 10 fps.

Cambio del tamaño del template

Se probó a agrandar y a reducir el template. El resultado era evidentemente siempre el mismo: Si lo agrandamos se reduce el número de ventanas posibles, pero se incrementan las iteraciones por ventana (porque hay más píxeles en cada una). Si lo reducimos, reducimos también las iteraciones por ventana, pero aumentamos las ventanas posibles en la imagen.

De esta forma, el coste se mantiene constante independientemente del tamaño del *template*.

Escalado de la imagen

Lo que sí que ha conseguido mejorar el rendimiento del algoritmo es reducir el tamaño de la imagen, pues en este caso, hay menos ventanas posibles mientras que no aumentan las iteraciones por ventana.

Para conseguirlo, no podemos captar directamente imágenes con menos resolución porque ya se está utilizando la resolución mínima, que es 640x480. Por ello, una vez captada la imagen de 640x480, la reducimos con un escalado. Se ha escalado a 200x150, pues se ha comprobado que para dichos valores la frecuencia de muestreo se hace aceptable y vale alrededor de 8 fps. Además, para no alterar la imagen captada, tras el escalado, se ha mantenido el mismo *aspect ratio*, es decir, 4:3.

Se podría pensar que añadir a cada *frame* una transformación de escalado, resulta ineficiente y reduce significativamente la mejora del algoritmo que se obtiene utilizando una imagen pequeña para el *Template Matching*. Sin embargo, lo cierto es que el escalado es un proceso muy sencillo que apenas añade coste.

10.1.5 Alternativas para conseguir escalabilidad

Aunque se han utilizado cuatro señales, lo interesante es que sea escalable. Es decir que podamos aumentar el número de señales (aunque sea hasta un valor razonable) sin que disminuya demasiado la frecuencia de muestreo. Gracias a las mejoras utilizadas hasta el momento, hemos conseguido una frecuencia de muestreo aceptable, sin embargo, en la medida en que vayamos aumentando el número de señales, el algoritmo se irá haciendo proporcionalmente más costoso y habrá que ir reduciendo cada vez más la imagen para mantener la frecuencia de muestreo.

Reducir la imagen hasta cierto punto es una mejora, mientras que reducirla demasiado es un error, sobre todo, si es una medida para contrarrestar el aumento de señales, ya que, cuando escalamos la imagen, perdemos píxeles y por lo tanto capacidad de distinción. Si vamos perdiendo capacidad de distinción mientras vamos añadiendo señales, el algoritmo será cada vez más impreciso y cometerá más errores. No obstante, reducir la imagen no es la única medida que podemos aplicar.

Paralelización

Hasta ahora, aunque la *Raspberry* tiene cuatro núcleos, las pruebas se han ejecutado en un solo proceso, hilo de ejecución y núcleo. Ésto quiere decir que solo hemos estado aprovechando un 25% de la potencia de cómputo de la *Raspberry*. Una mejora de eficiencia sería paralelizar el programa para que se ejecutara en los cuatro núcleos a la vez. Sin embargo, no todos los algoritmos son paralelizables, ya que muchos deben ejecutarse de forma secuencial porque existen dependencias de entrada o salida entre sus diferentes fases. Dentro de nuestro programa, solo algunos fragmentos de código son paralelizables.

Concretamente, el bucle de esta prueba que ejecuta el *Template Matching* para cada una de las señales, es paralelizable. Así que, aunque en este caso no representa una mejora significativa porque se ha escalado la imagen para que la fase del *Template Matching* sea ligera, se ha decidido paralelizar dicho bucle.

La paralelización significará dividir el bucle entre los cuatro núcleos, lo que dividirá su tiempo de ejecución entre cuatro. Sin embargo, cómo afecte esto al tiempo de ejecución global (inversa de la frecuencia de muestreo) depende del porcentaje de dicho tiempo global que represente el bucle paralelizado. Cuanto mayor sea este porcentaje, mayor será la mejora global, que se puede calcular a partir de la mejora parcial con la ley de Amdahl.

Si se va aumentando la cantidad de señales, este bucle se irá haciendo más pesado, sacando más rentabilidad a la paralelización. Para este proceso se ha utilizado *multiprocessing.Pool*, que implementa de forma sencilla una piscina de procesos. No se han utilizado hilos por el problema del *GIL* explicado anteriormente. Cada proceso se ejecuta en un intérprete (o *GIL*) y no puede haber más de un hilo de ejecución a la vez en el mismo *GIL*. Esto, en la mayoría de los casos se traduce en que no existe paralelización, pues un hilo se encuentra dentro del *GIL* mientras el resto esperan (Figura 10.1.A). Es decir, no se saca ningún partido a la paralelización. Si, de lo contrario, utilizamos varios procesos, cada uno tendrá su propio *GIL* y podrán ejecutarse en paralelo (Figura 10.1.B).

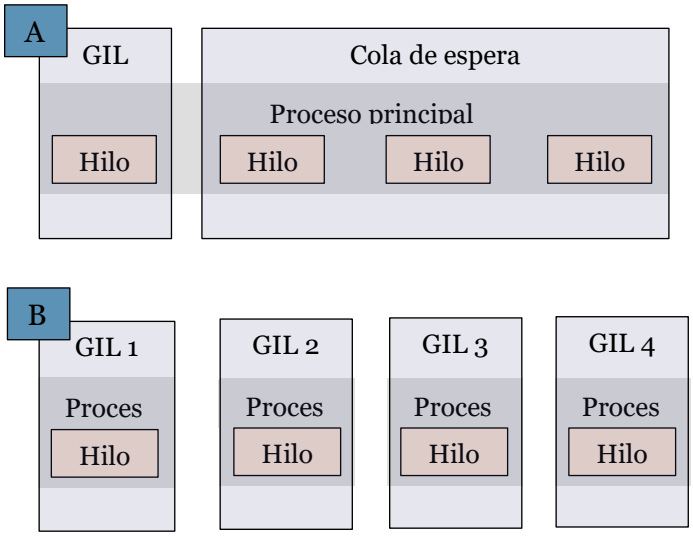


Figura 10.1 Ejecución de cuatro hilos VS cuatro

Localización de la búsqueda

Otra mejora en términos de eficiencia que se podría hacer al algoritmo sería utilizar en el *frame* técnicas de detección por color como las que se han visto en las pruebas dos y tres para reducir el rango de búsqueda del *Template Matching*, convirtiendo en local una búsqueda global. Por ejemplo, podríamos hacer el *Backprojection Histogram* del *template* en la captura y después aplicar el *Template Matching* solo sobre la ventana con mayor densidad de puntos.

10.2 Detección y maniobras tras el *Template Matching*

10.2.1 Información extraída del *Template Matching*

Tras aplicar el *Template Matching* una vez para cada señal (o *template*), el algoritmo halla el píxel con mayor valor para cada imagen devuelta. Estos son, como ya se ha mencionado, los píxeles con mayor probabilidad de ser las esquinas superiores izquierdas de las señales en la imagen. Se selecciona la señal con máxima probabilidad y si ésta es mayor que un límite, se considera que la señal está presente en la imagen.

Además, el resultado del *Template Matching* es totalmente dependiente del tamaño del *template*, lo que en nuestro caso es muy conveniente. Imaginemos que situamos el robot a un metro de distancia de un cono para hacerle una captura y emplearla posteriormente como *template*. Si en cada iteración posterior aplicamos el *Template Matching* para intentar hallar la captura del cono en los *frames* que va captando la cámara, solo se detectará el cono con una probabilidad alta cuando se vea éste con aproximadamente el mismo tamaño que el *template*. O lo que es lo mismo, cuando el robot se encuentre a un metro del cono.

Esto, en nuestra prueba significa que, una vez hecha la captura del *template* de una señal, el robot siempre la detectará a la misma distancia, por lo que ésta se puede calibrar para cada señal según convenga.

10.2.2 Maniobras tras la detección de una señal

Por último, una vez detectada la señal, el robot se detiene y hace la maniobra pertinente. En el caso del cono, por ejemplo, hace un giro de tipo dos en una dirección, a continuación, sigue recto, efectúa un giro de tipo dos en el sentido contrario al anterior y sigue recto. Una vez realizados estos pasos se considera que el robot ya ha sobrepasado el obstáculo y se realizan las maniobras opuestas para volver al carril inicial (la línea que habría seguido de haber ido recto y sin tener que esquivar ningún obstáculo).

10.3 Resumen del algoritmo

Como se puede apreciar en el diagrama de flujo de la Figura 10.2, al principio de la prueba se cargan en escala de grises los *templates* de las cuatro señales y el robot avanza recto. En cada iteración se escala la imagen captada a 200x150 píxeles y se convierte a escala de grises. Se aplica *Template Matching* para la imagen con cada *template* y se halla el máximo valor de cada resultado. De entre todos los máximos, se obtiene el máximo, que corresponde a la señal más probable. Si la probabilidad es mayor que un límite, se considera que se ha detectado la señal y se mandan las órdenes correspondientes al *Arduino*. Después de que el *Arduino* aplique las órdenes, el robot sigue recto y vuelve a empezar el algoritmo.

10.4 Resultados

Como se puede comprobar en los dos vídeos, se han obtenido resultados satisfactorios. Concretamente, para cada vídeo se han encadenado dos señales. Aunque se han utilizado cuatro señales, su número es escalable, teniendo, por supuesto, siempre en cuenta la frecuencia de muestreo.

https://youtu.be/oYFje6FL7Mc

https://youtu.be/hbAwfveO_iA

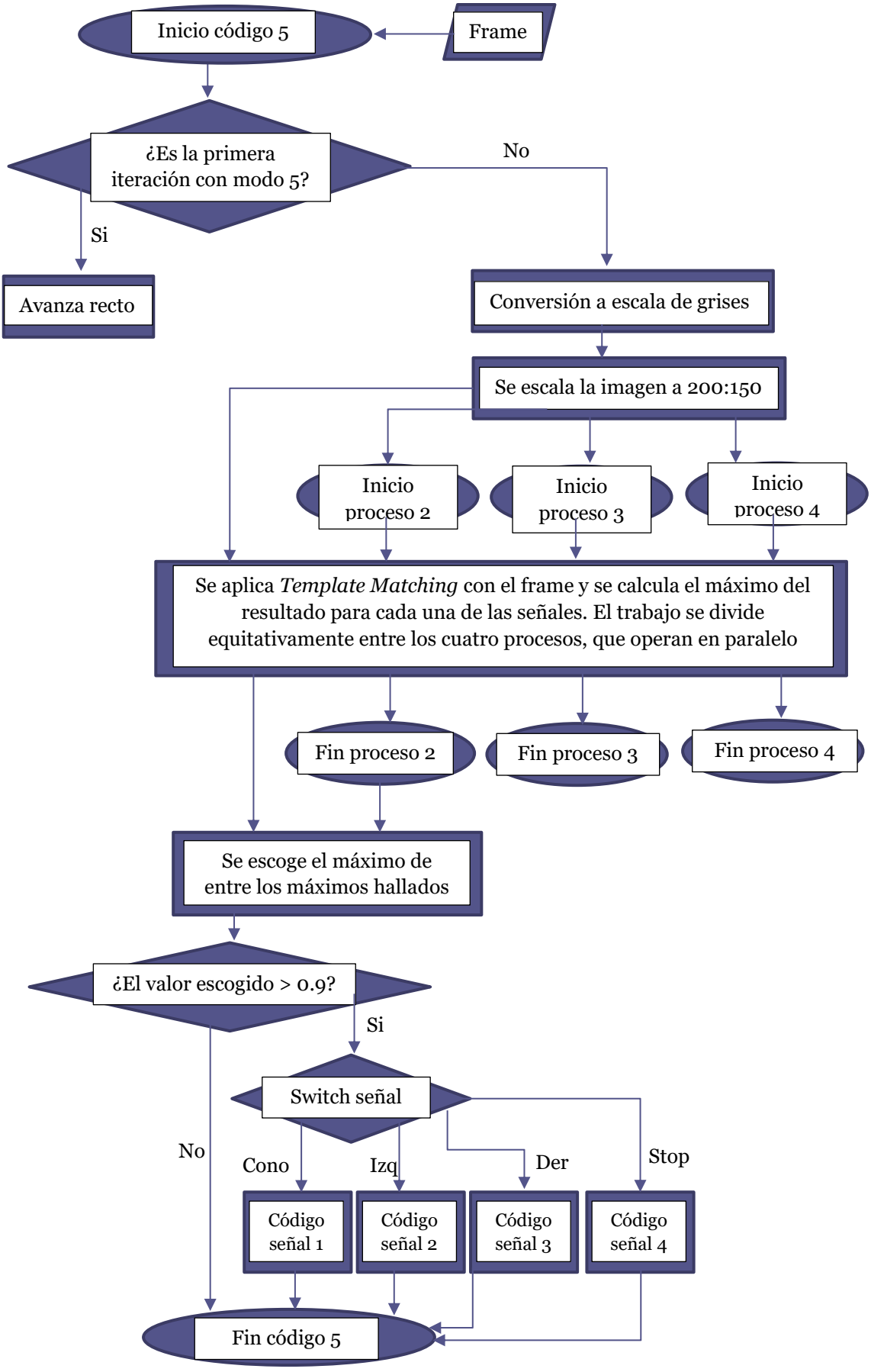


Figura 10.2 Flujo del código específico del modo 5

11. Conclusiones

Tras la realización del trabajo hemos llegado a las siguientes conclusions:

- Se ha hecho un análisis completo de la librería *OpenCV*, estudiando el funcionamiento, la finalidad y el rendimiento de cada uno de los algoritmos que nos ofrece esta herramienta.

  Se trata de un conocimiento adquirido de forma completamente autónoma y que, además es un complemento perfecto para mi formación como ingeniero informático.

- El sistema se ha desarrollado con éxito, desde la construcción del robot hasta la estructuración de los comportamientos propios de cada una de las partes del sistema, pasando por los protocolos que se han establecido para la comunicación entre las mismas.

  A pesar de las dificultades técnicas que, como en todo proyecto, se han presentado, se ha conseguido un sistema funcional y flexible totalmente adecuado para el desarrollo de este trabajo y de trabajos futuros.

- Se han planteado y resuelto cinco retos diferentes. Como se ha mencionado anteriormente, en todos ellos, la información obtenida por el robot proviene en su totalidad del análisis de las capturas tomadas por una cámara, que es la clave de la generalidad del Sistema, ya que, únicamente con la visión artificial se han suplido el sensor de infrarrojos, el sensor de color y el de ultrasonidos para distinguir entre blanco y negro, percibir un color y detectar un obstáculo respectivamente.

  Los retos han implicado el desarrollo de un código complejo, así como la necesidad de soslayar diversos problemas, sobre todo físicos y de eficiencia, los cuales han puesto a prueba y afianzado los conocimientos sobre *OpenCV* adquiridos durante el trabajo.

  Además, aun pudiendo repetir algoritmos, se ha utilizado una forma distinta de reconocimiento de imágenes en cada prueba, obteniendo una visión global de *OpenCV*.

- Los resultados de las cinco pruebas, como se ha mostrado en los vídeos, han sido altamente satisfactorios, demostrando el poder de la visión artificial y sus aplicaciones al guiado de dispositivos móviles.

  No obstante, es necesario resaltar que las condiciones de iluminación son determinantes en la mayoría de algoritmos de visión artificial y, sin duda, deben ser una de las principales preocupaciones para quien pretenda profundizar en este campo.

  Otro factor importantísimo en el guiado autónomo de dispositivos móviles es la eficiencia, de la cual depende la frecuencia de muestreo, a la que se le ha dado mucha relevancia en este trabajo.

Por experiencia, podemos afirmar que tanto la iluminación como eficiencia son prioritarias cuando hablamos de visión artificial en tiempo real.

11.1 Trabajos futuros

Como ya se ha comentado en apartados anteriores, se ha diseñado un sistema cuya capacidad no se limita únicamente a realizar las pruebas planteadas en este proyecto, sino que es flexible y sobre él pueden desarrollarse multitud de trabajos. En este apartado se ofrecen algunas ideas de trabajos futuros:

- Añadir al robot un micrófono y un reconocedor de voz con el objetivo de que el usuario pueda dar sus órdenes por voz. Ésto no solo simplificaría la comunicación entre el usuario y el robot, sino que además eliminaría la necesidad de un *host* y, por lo tanto, de una red *Wifi*.

- Rediseñar, modelar e imprimir un chasis para el robot que, además, incluya una cubierta que proteja los componentes. Otra mejora que se le podría hacer sería añadir otro servo a la cámara para que ésta tenga un grado más de libertad y pueda girar también en horizontal.

- Conectar la *Raspberry* con geolocalización. Ésto permitiría que el usuario conociese la posición del robot en todo momento (por ejemplo para controlarlo remotamente) y que fijase rutas para el mismo.

- Convertir el sistema en un producto comercial, como un juguete robot programable o una mascota robot. En cualquier caso, se necesitaría un rediseñado completo del robot y una adaptación del *software* para que éste cumpliera con los requisitos funcionales del producto.

Evidentemente, estas mejoras no se han aplicado por ser ajenas a los objetivos del presente proyecto.

12. Bibliografía

Manual oficial de *OpenCV Python*:

http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_tutorials.html

Ejemplo básico de cliente y servidor en *Python*:

http://www.pythondiario.com/2015/01/simple-programa-clienteservidor-socket.html

Tutorial para conectar *Raspberry* y *Arduino* a través del Puerto serie

https://geekytheory.com/arduino-raspberry-pi-raspduino/

Tutorial para la instalación de OpenCV:

http://www.pyimagesearch.com/2016/04/18/install-guide-raspberry-pi-3-raspbian-jessie-opencv-3/

Tutorial para la conexión a una *Raspberry* por escritorio remote:

http://www.circuitbasics.com/access-raspberry-pi-desktop-remote-connection/

Documentación de la librería *Picamera*:

https://picamera.readthedocs.io/en/release-1.13/

Documentación de *OpenCV*:

http://docs.opencv.org/2.4/modules/refman.html

Documentación de la librería *multiprocessing*:

https://docs.python.org/2/library/multiprocessing.html

Manual official de *OpenCV*:

http://docs.opencv.org/3.0-beta/doc/tutorials/tutorials.html

Tutorial para controlar motores de corriente continua con un l298n:

https://electronilab.co/tutoriales/tutorial-de-uso-driver-dual-l298n-para-motores-dc-y-paso-a-paso-con-arduino/