Trabajo Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Localización de productos en entorno comercial mediante códigos de barras usando un robot móvil

Autor: Juan de Dios Herrera Hurtado
Tutor: Miguel Ángel Ridao Carlini

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

Sevilla, 2023

Proyecto Fin de Grado
Grado en Ingeniería Electrónica Robótica y Mecatrónica

Localización de productos en entorno comercial mediante códigos de barras usando un robot móvil

Autor:
Juan de Dios Herrera Hurtado

Tutor:
Miguel Ángel Ridao Carlini
Catedrático de Universidad

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2023

Trabajo Fin de Grado: Localización de productos en entorno comercial mediante códigos de barras usando un robot móvil

Autor: Juan de Dios Herrera Hurtado

Tutor: Miguel Ángel Ridao Carlini

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocal/es:

Secretario:

Acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

En primer lugar, me gustaría darle las gracias a los tres establecimientos donde he tomado fotos por su amabilidad: la gasolinera BP A-49 Pilas y los supermercados Super REK2 y Mercadona.

También quiero darle las gracias a mi familia por su apoyo en todo momento a lo largo de esta etapa.

No puedo olvidarme de mis compañeros del Club de fans de Manolo: Javi, José Antonio y Arturo que, de no ser por ellos, estos cuatro años habrían sido mucho más duros de lo que ya de por sí han sido.

Por último, me gustaría agradecerle a mi tutor Miguel Ángel toda la ayuda que me ha ofrecido durante la elaboración de este proyecto y la libertad que me ha dado a la hora de enfocarlo.

Juan de Dios Herrera Hurtado

Sevilla, 2023

Resumen

El trabajo que se presenta es una continuación de una serie de Trabajos Fin de Grado ya realizados sobre un proyecto acordado entre la empresa Tier1 y la ETSI.

La idea del proyecto en su conjunto es, mediante un robot móvil, realizar un mapeo del comercio en cuestión identificando pasillos que serán las celdas libres del mapa y estanterías que serán los obstáculos, localizar las coordenadas dentro de ese mapa de cada producto y, finalmente, planificar una ruta de coste mínimo y realizarla pasando por los diferentes productos que el cliente haya indicado que desea comprar.

En este trabajo nos centraremos en la segunda parte, es decir, la localización de los productos en el mapa que es una parte puramente basada en visión por computador, para ello, el robot deberá recorrer todas las estanterías del comercio identificando los productos por su código de barras y asignándoles unas coordenadas (x, y, z) en el mapa previamente calculado. Tomaremos como referencia [1], el trabajo de Pedro Tito Macías Roselló, que fue el último compañero en intervenir en el trabajo y en modificar esta segunda parte concretamente.

Antes de intervenir en el proyecto, el robot era capaz de identificar códigos de barras sobre una pared de azulejos, pero a veces identificaba falsos positivos, esto significa que el programa indicaba que en un lugar de la imagen existía un código de barras cuando realmente no era así. El otro problema que presentaba la propuesta previa era que estábamos trabajando sobre una pared prácticamente lisa, luego en caso de introducir algún elemento más en la imagen, la efectividad del algoritmo se vería drásticamente mermada.

Por ello, los dos objetivos que se proponen cumplir con este trabajo son resolver ambos problemas expuestos en el párrafo anterior. En primer lugar, lograr que el programa identifique códigos de barras en un entorno comercial real, que es donde queremos implementar este proyecto y, en segundo lugar, reducir las probabilidades de obtener falsos positivos.

En la práctica, no se dispone del robot móvil en un entorno comercial real, sería complicado llevarlo y es susceptible de recibir algún golpe accidentalmente por parte de una persona. Además, el trabajo propuesto ya de por sí se considera que es bastante completo. No obstante, se ha tratado de que los experimentos realizados emulen lo más fielmente posible las condiciones reales en las que se encontraría el robot. Dado que no dispondremos del robot, tomaremos imágenes con la cámara del móvil y serán estas imágenes las que se procesen.

Abstract

The work presented here is a continuation of a series of Final Degree Projects already carried out on a project agreed between the company Tier1 and the ETSI.

The idea of the project as a whole is, by means of a mobile robot, to map the shop in question by identifying aisles that will be the free cells on the map and shelves that will be the obstacles, to locate the coordinates of each product on the map and, finally, to plan a minimum-cost route and carry it out by passing through the different products that the customer has indicated he wishes to buy.

In this work we will focus on the second part, that is, the location of the products on the map, which is a part purely based on computer vision. To do this, the robot must go through all the shelves of the shop identifying the products by their barcode and assigning them coordinates (x, y, z) on the previously calculated map. We will take as a reference [1], the work of Pedro Tito Macías Roselló, who was the last colleague to intervene in the work and to modify this second part specifically.

Before intervening in the project, the robot was able to identify barcodes on a tiled wall, but sometimes it identified false positives, meaning that the programme would indicate that there was a barcode in one place in the image when there really wasn't. The other problem with the previous proposal was that we were working on a practically smooth wall, so if we introduced any other element into the image, the effectiveness of the algorithm would be drastically reduced.

Therefore, the two objectives of this work are to solve both problems outlined in the previous paragraph. Firstly, to get the program to identify barcodes in a real commercial environment, which is where we want to implement this project, and secondly, to reduce the probability of obtaining false positives.

In practice, the mobile robot is not available in a real commercial environment, it would be difficult to carry and is susceptible to being accidentally hit by a person. Moreover, the proposed work is already considered to be quite complete. However, we have tried to ensure that the experiments carried out emulate as closely as possible the real conditions in which the robot would find itself. Since we do not have the robot at our disposal, we will take images with the mobile phone camera and these images will be processed.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

ÍNDICE DE CÓDIGOS

Notación

ETSI	Escuela Técnica Superior de Ingeniería
G	Módulo del gradiente
Gx	Gradiente horizontal
Gy	Gradiente vertical
x	Coordenada horizontal en la imagen
y	Coordenada vertical en la imagen
ρ	Distancia en la representación polar de una recta
θ	Ángulo en la representación polar de una recta
sen	Función seno
cos	Función coseno
Xcm	Coordenada horizontal del centro de masas
Ycm	Coordenada vertical del centro de masas
$<$	Menor que
$>$	Mayor que
$\leq$	Menor o igual que
$\geq$	Mayor o igual que
$\Rightarrow$	Entonces

1 INTRODUCCIÓN

1.1 Motivación

Como se ha introducido en el resumen, este proyecto continúa una serie de trabajos ya realizados sobre una propuesta acordada entre la empresa Tier1 y la ETSI. En concreto, nosotros continuaremos el último trabajo realizado sobre el proyecto perteneciente al alumno Pedro Tito Macías Roselló y que denominaremos como "TFG de referencia" de aquí en adelante.

1.2 El código de barras

El código de barras [2] fue creado en 1952 por los inventores Joseph Woodland, Jordin Johanson y Bernard Silver, pero no fue posible su uso comercial hasta el año 1966. Se trata de un código basado en la representación de un conjunto de líneas paralelas de distinto grosor, color y espaciado que en su conjunto contienen una determinada información, es decir, las barras y espacios del código representan pequeñas cadenas de caracteres. De esta forma, los códigos permiten identificar un artículo de manera única y no ambigua debido a que el código de barras asociado a una determinada cadena de caracteres es único. Dicho de otro modo, cada código de barras representa una determinada cadena de caracteres y solo una cadena de caracteres.

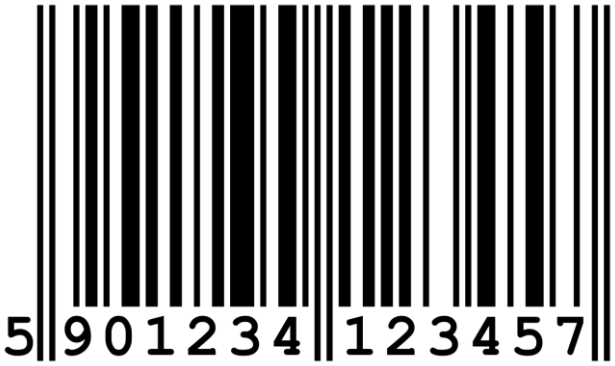

Figura 1-1. Ejemplo de código de barras EAN 13 [2].

Existen varios tipos de códigos de barras, aunque los más extendidos son los del tipo EAN13. Los códigos de barras del tipo EAN (European Article Number) [3], son un tipo de código de barras de carácter numérico adoptado por más de cien países y cerca de un millón de empresas. El 13 indica el número de caracteres que posee la cadena numérica, en la imagen anterior se pueden ver los 13 dígitos de la cadena correspondiente a ese código de barras.

A pesar de que los códigos de barras llamados matriciales o bidimensionales como los códigos QR poseen numerosas ventajas frente a los códigos de barras unidimensionales, no son estos los que encontramos en los comercios hoy en día, sino los convencionales de una dimensión. De ahí el motivo de que se quiera trabajar con códigos de barras unidimensionales y no con códigos QR por ejemplo, simplemente porque es lo que está presente actualmente en los comercios.

1.3 Descripción del problema

El trabajo que se presenta es un trabajo puramente de visión por computador, ya que se basa en obtener información, en nuestro caso identificar los posibles códigos de barras presentes en la imagen realizada con la cámara del robot, aunque en nuestro caso al no disponer del mismo en un entorno comercial real, se usará como ya se ha comentado la cámara del móvil para obtener las imágenes.

A continuación, se muestra un esquema de lo que se pretende que el algoritmo realice:

Figura 1-2. Esquematización del problema que se pretende resolver.

De los capítulos 5 al 10, desarrollaremos el algoritmo que por ahora hemos identificado como una "caja negra".

Dado que para este trabajo no se hace uso del robot real, no se considera oportuno mencionar las características y elementos de este, aunque si resulta de interés, se puede consultar en el TFG de referencia [1] en el capítulo "Hardware".

Asimismo, también se puede consultar más en detalle las funcionalidades adicionales previamente mencionadas en el resumen, como la creación del mapa o la creación de trayectorias una vez se recibe el listado de productos que el cliente desean comprar. Todo ello se puede consultar en el capítulo "Marco teórico" en la sección "Alcance" [1].

1.4 Objetivos del proyecto

La meta de este proyecto es mejorar el sistema de visión ya existente haciéndolo más robusto y flexible al entorno en el que se aplique. Es por ello por lo que se marcan dos objetivos claros y que detallaremos a continuación.

1.4.1 Posibilidad de aplicar el algoritmo de visión en un entorno real

En primer lugar y más importante, la plataforma robótica como producto final es algo que va a estar funcionando en un comercio real, luego debemos de elaborar un algoritmo que sea capaz de atajar todas las dificultades que ello plantea con la mayor efectividad posible.

Por este motivo, desde el comienzo de este trabajo se ha estado trabajando con imágenes tomadas a estanterías de comercios reales para asegurarnos que este objetivo se cumpliese en mayor o menor medida.

1.4.2 Robustez frente a la detección de falsos positivos

Por otro lado, se pretende reducir el número de falsos positivos, ya que esto ralentiza directamente el funcionamiento de esta parte de localización de productos como se explicará más adelante.

No obstante, se avanza que este mayor tiempo de ejecución se debe a que una vez identificamos dónde se encuentran los potenciales códigos de barras, debemos hacer zoom a dichas zonas de la imagen para que un algoritmo de decodificación de códigos de barras sea capaz de decodificarlo y obtener el código numérico correspondiente a ese producto. Luego, si hacemos zoom a un lugar donde no existe ningún código de barras, el algoritmo no será capaz de decodificar nada y se habrá perdido un tiempo que se podría haber empleado identificando un código de barras que sí existía.

2 SOFTWARE UTILIZADO

En este capítulo comentaremos los programas utilizados para llevar a cabo el trabajo.

2.1 Visual Studio Code (versión 1.78.2)

Visual Studio Code [4] es un editor de código fuente desarrollado por Microsoft. Se encuentra disponible para Windows, macOS y Linux. Contiene infinidad de extensiones para lenguajes de programación como C#, C++, Java o Python entre otros.

Figura 2-1. Logo de Visual Studio Code [4].

2.2 Python (versión 3.10.4)

Python [5] es un lenguaje de programación interpretado, interactivo y orientado a objetos. Incluye módulos, excepciones, tipos dinámicos, tipos de datos dinámicos de muy alto nivel y clases. Admite múltiples paradigmas de programación más allá de la programación orientada a objetos, como la programación procedimental y funcional. Además, combina una potencia notable con una sintaxis muy clara. Se ejecuta en muchas variantes de Unix, incluidos Linux y macOS y en Windows.

Figura 2-2. Logo de Python [5].

2.3 OpenCV (versión 4.5.5)

OpenCV (Open Source Computer Vision Library) [6] es una biblioteca de software de aprendizaje automático y visión artificial de código abierto. OpenCV se creó para proporcionar una infraestructura común para las aplicaciones de visión por computador y para acelerar el uso de la percepción de la máquina en los productos comerciales. La biblioteca dispone de más de 2500 algoritmos optimizados que van desde identificar contornos en una imagen, hasta cosas tan sofisticadas como por ejemplo clasificar acciones humanas en vídeos o extraer modelos 3D de objetos.

Figura 2-3. Logo de OpenCV [6].

2.4 GitHub

GitHub [7] es una herramienta para compartir proyectos entre la comunidad y es excelente para realizar un control de versiones de los diferentes códigos que se van desarrollando. Además, permite recuperar versiones antiguas del código en caso de que sea necesario.

Figura 2-4. Logo de GitHub [7]

3 HARDWARE UTILIZADO

Para tomar las imágenes se ha usado la cámara de un Xiaomi Redmi Note 10 Pro. Se recoge ahora en una tabla las características de la cámara trasera que es la que usaremos y que consta de un total de tres cámaras y un sensor de profundidad. Toda esta información se ha extraído de la página oficial de Xiaomi [8].

Tabla 3-1. Características de la cámara.

Tipo	Tamaño del sensor (megapíxeles)	Características
Cámara ultra gran angular	108	Tamaño de píxel de 0,7 µm, superpíxel 9 en 1 de 2,1 µm Tamaño del sensor de 1/1,52” f / 1,9
Cámara ultra gran angular	8	Campo de visión de 118° f / 2,2
Cámara telemacro	5	f / 2,4 AF (autofocus)
Sensor de profundidad	2	f / 2,4

Se muestra ahora una imagen de la cámara y la correspondencia con cada uno de los elementos mencionados en la tabla anterior.

Figura 3-1. Cámara Xiaomi Redmi Note 10 Pro [9].

4 CONSIDERACIONES A LA HORA DE TOMAR LA IMAGEN

4.1 Posición horizontal

En primer lugar, la posición del móvil será horizontal. De esta manera, logramos una imagen que abarca más estantería horizontalmente respecto a si la posición del móvil fuese vertical, todo ello considerando que se toman las imágenes a la misma distancia respecto a la estantería.

Figura 4-1. Imagen tomada con el móvil en posición vertical.

Figura 4-2. Imagen tomada con el móvil en posición horizontal.

Como se puede apreciar, la imagen tomada con el móvil en posición horizontal abarca más estantería que la tomada con el móvil en posición vertical.

Por otro lado, mientras más alejados estemos de la estantería, más estantería abarcaremos dentro de la imagen. A continuación, se muestra esto más claramente en un dibujo:

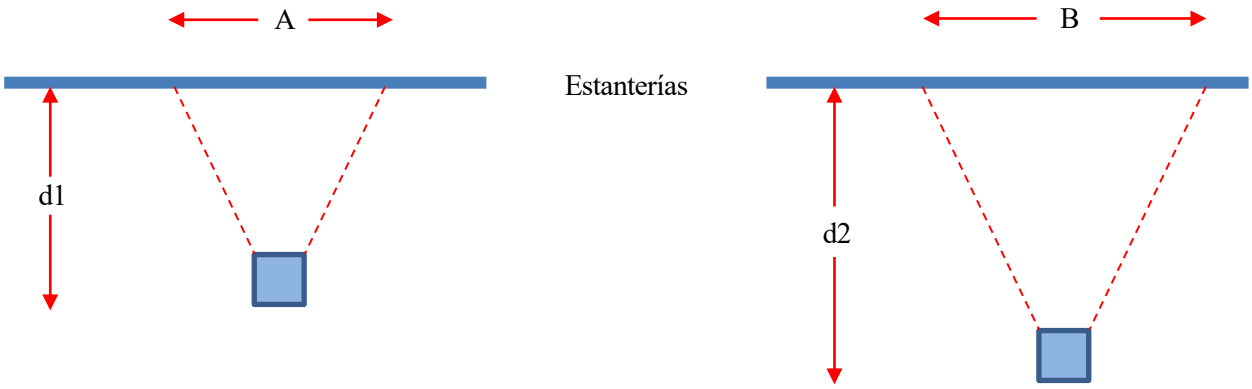

Figura 4-3. Relación distancia respecto a la estantería y amplitud de la estantería que entra en la imagen.

Por lo tanto:

$$\text{Si } d_1 < d_2 \Rightarrow A < B \tag{4-1}$$

4.2 Tamaño de la imagen

Usando el móvil de manera horizontal, se obtienen imágenes de 3000 píxeles de alto y 4000 píxeles de ancho tal y como se muestra en la imagen.

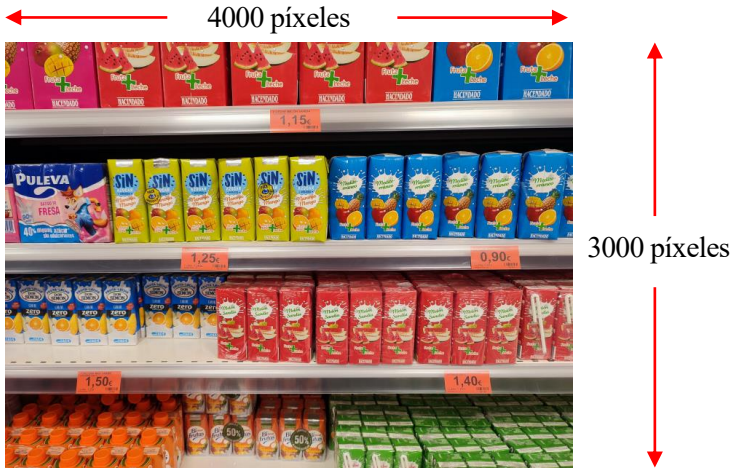

Figura 4-4. Tamaño de las imágenes obtenidas.

Esta información se obtiene fácilmente haciendo uso del atributo **.shape** o simplemente consultando las propiedades de algunas de las imágenes tomadas.

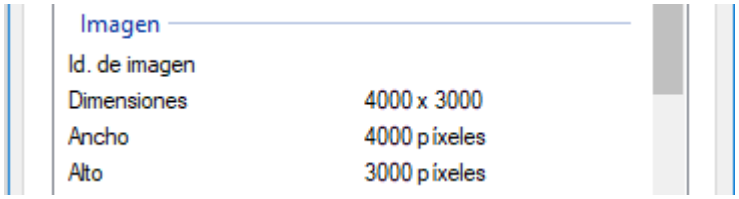

Figura 4-5. Consulta de las dimensiones de la imagen en sus propiedades.

Conociendo esta información ya se ve claramente por qué cuando usamos el móvil en horizontal abarcamos más estantería al tener 4000 píxeles de ancho, mientras que con el móvil en vertical tendríamos 3000 píxeles de ancho.

4.3 Zoom

4.3.1 ¿Qué es el zoom?

En primer lugar, definamos qué es el zoom. Es bien conocido por todos que cuando hablamos del concepto de zoom nos referimos a captar un objeto a un tamaño mayor al que realmente lo veríamos desde esa posición, es decir, el efecto de que parezca que nos hemos acercado a ese objeto para verlo más grande, pero sin realmente habernos movido.

4.3.2 Importancia del zoom en el algoritmo

El zoom es una parte importante del algoritmo como se verá una vez lo expliquemos, en él es necesario aplicar un primer zoom para identificar los códigos de barras en la imagen y, luego, un segundo zoom para decodificar la información de esos códigos de barras identificados.

La imagen que se tomará de la estantería se hará en modo normal y sin aplicar ningún tipo de zoom, es decir, zoom 1X (tamaño real). Pero como se ha comentado, el algoritmo requiere de aplicar zoom y es aquí cuando se nos presenta un problema: el zoom de la cámara del móvil que se va a utilizar es digital. Para entender por qué supone esto un problema hablaremos de qué es el zoom digital y en contraposición del zoom óptico.

4.3.3 Zoom digital

[10] Cuando tenemos una lente fija, la única forma de ampliar un objeto es quedarnos solo con la parte del sensor que capta ese objeto y realizar un proceso de interpolación para "inventarnos" el resto de la fotografía. Suponiendo por ejemplo un sensor de 8x8 píxeles que produce imágenes de 64 píxeles, si queremos hacer zoom nos quedamos por ejemplo con un recuadro de 4x4 píxeles del interior del sensor que produce una imagen de 16 píxeles, e interpolamos el resto de los píxeles recuperando la imagen de tamaño 64 píxeles.

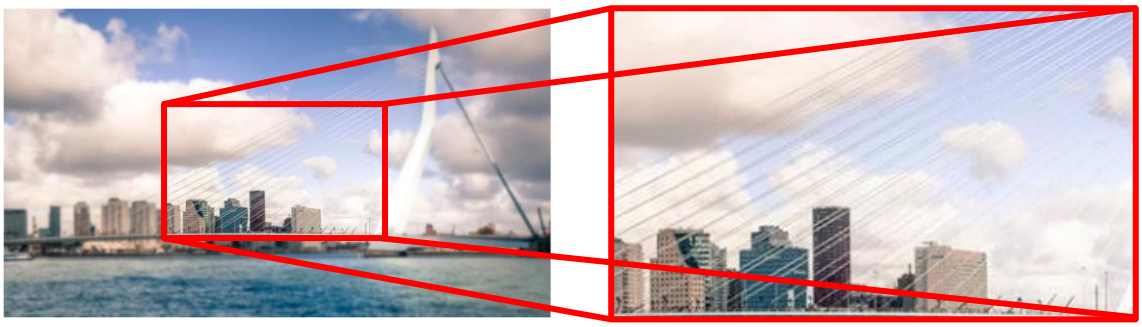

Figura 4-6. Ejemplo de zoom digital [10].

Como se puede apreciar, la imagen obtenida no refleja fielmente la realidad ya que nos hemos "inventado" el valor de ciertos píxeles usando el valor de los píxeles vecinos a ese que queríamos rellenar, por todo ello, una imagen a la que se le ha aplicado zoom digital pierde calidad.

4.3.4 Zoom óptico

En contraposición al zoom digital tenemos el zoom óptico [10]. En este caso la lente no está fija, se mueve logrando modificar la distancia focal. La distancia focal es la distancia que existe entre la lente y el sensor de la cámara. A mayor distancia focal se logrará un mayor acercamiento, es decir, una imagen del objeto a mayor tamaño.

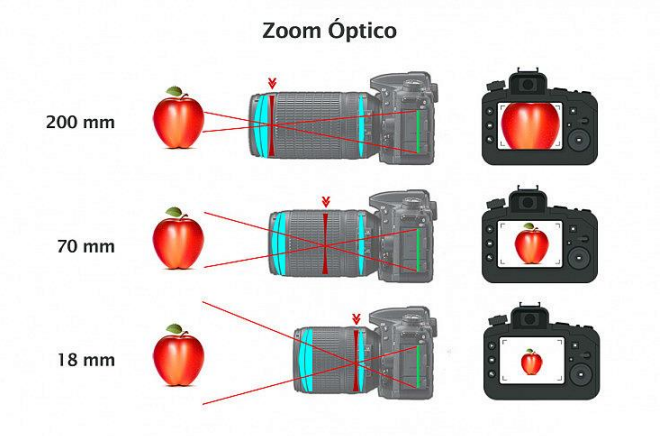

Figura 4-7. Funcionamiento del zoom óptico [11].

Aquí podemos apreciar que mientras más separados estén lente y sensor, más grande observaremos la manzana en la imagen y viceversa.

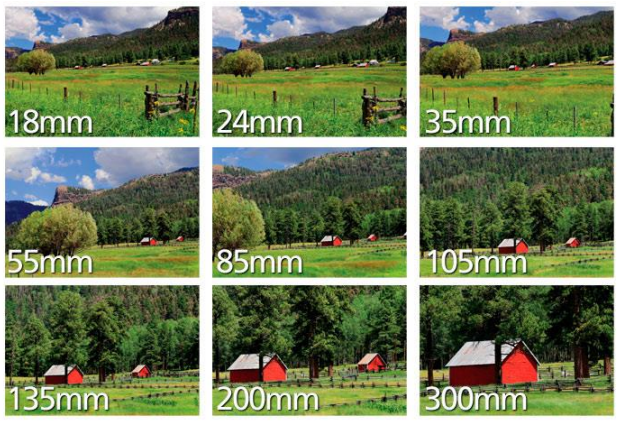

Figura 4-8. Ejemplo de zoom óptico [12].

Es importante recalcar que en este caso no nos "inventamos" valores de píxeles, se mantiene la información fielmente, de ahí el motivo de que una imagen con zoom óptico no pierda calidad.

4.3.5 Solución adoptada para el problema del zoom

Ya podemos entender que, si usamos el zoom digital de la cámara del móvil, al perder información la imagen, puede que el algoritmo obtenga ciertos resultados que han sido conducidos por esos errores consecuencia del proceso de interpolación o que por ejemplo no podamos decodificar un código de barras por falta de nitidez en la imagen al realizar el zoom. En la práctica, la cámara con la que cuenta el robot real goza de zoom óptico, luego este problema desaparecería.

En nuestro caso, para resolver este problema y poder emular el zoom óptico de la cámara real, se ha optado por tomar imágenes más de cerca y así obtener esa información que nos proporciona el zoom óptico.

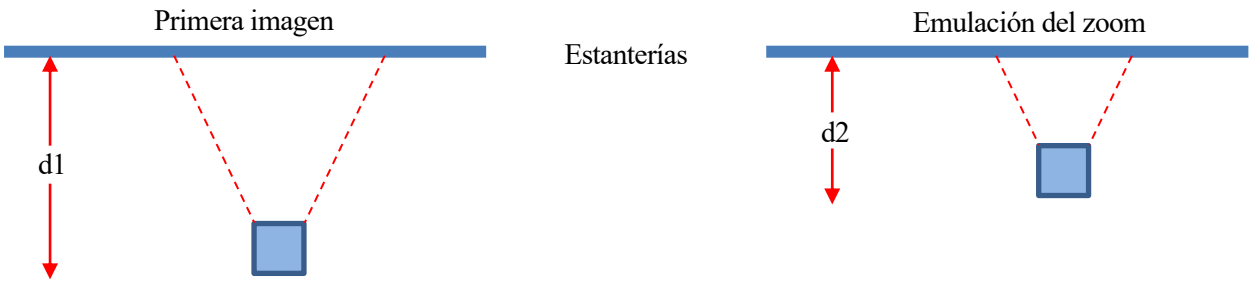

Figura 4-9. Solución adoptada para resolver el problema del zoom.

De esta manera, logramos el mismo efecto de un zoom óptico, es decir, un aumento del tamaño de los objetos a la vez que evitar pérdida de información sobre ese objeto. Vamos a mostrarlo con un ejemplo:

Figura 4-10. Imagen de partida para ilustrar la diferencia entre el zoom óptico y el digital.

Ahora haciendo zoom al código rodeado por el rectángulo de la imagen anterior desde la misma posición en la que hemos tomado dicha imagen se obtiene lo siguiente:

Figura 4-11. Zoom digital de la cámara usada a 10X (valor máximo).

Y si tomamos la imagen del mismo código, pero sin hacer zoom (1X) a una distancia más cercana:

Figura 4-12. Emulación del zoom óptico acercándonos a la estantería.

Comparando las figuras 4-11 y 4-12 vemos que las barras se ven mucho más nítidas en la segunda (aumentando la escala de este documento se aprecia mucho mejor que a simple vista), mientras que en la primera imagen al "inventarnos" valores nuevos pierde calidad.

4.4 Bandas superior e inferior cerca de los límites de arriba y de debajo de la imagen respectivamente

El tercer paso de la etapa 1 del algoritmo, como se verá en el capítulo 6, es dividir la imagen en franjas horizontales para facilitar la búsqueda de líneas horizontales que serán los límites superior e inferior de las bandas donde se encuentran los códigos de barras. Para ello suponemos que las bandas son equidistantes entre sí. Si por ejemplo la imagen contiene tres bandas de códigos de barras, la imagen se dividirá en tres franjas horizontales del mismo tamaño.

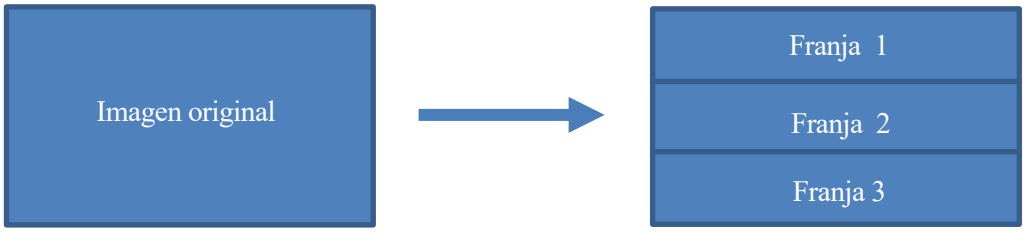

Figura 4-13. División de la imagen en franjas horizontales iguales.

La idea de este paso del algoritmo es aislar cada banda horizontal donde se encuentran los códigos de barras en cada una de las franjas en que se ha dividido la imagen. De ahí el motivo de que sea tan importante que la banda que está más arriba esté cerca del límite superior de la imagen y, que la banda que está más abajo esté cerca del límite inferior de la imagen, si es así, nos aseguraremos de que la división de las bandas en cada una de las franjas se realiza correctamente. Ahora se muestra un ejemplo de la división realizada erróneamente y realizada correctamente continuando con un ejemplo tres bandas.

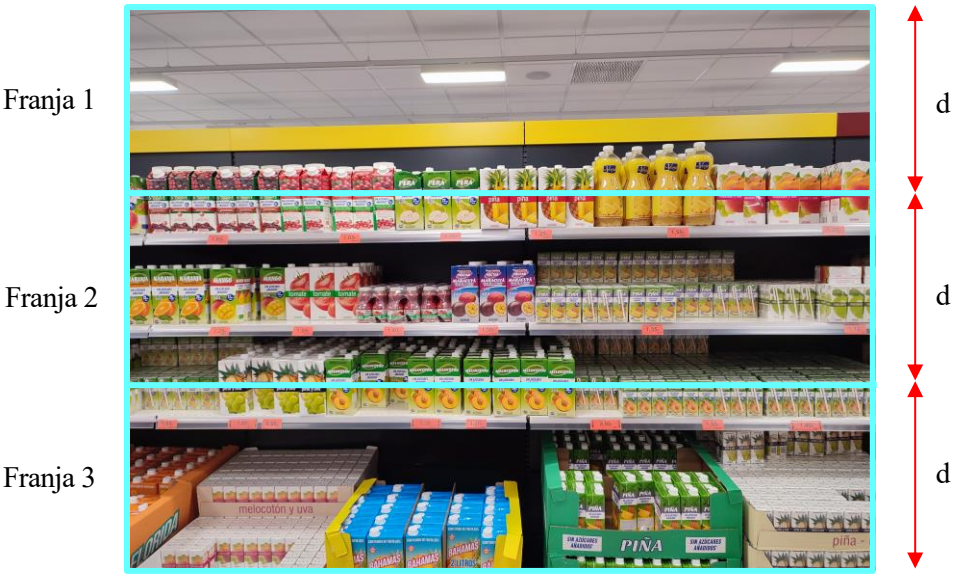

Figura 4-14. División en franjas errónea.

Como podemos ver en la figura anterior, al no encontrarse la banda de más arriba y la banda de más abajo cercanas a los límites superior e inferior de la imagen respectivamente, al dividir la imagen en franjas horizontales iguales, en la primera franja no aislamos ninguna banda y, en cambio, en la segunda tenemos dos bandas, luego la división no ha podido hacer lo que se busca con ella que es aislar las bandas.

Figura 4-15. División en franjas correcta.

En cambio aquí, vemos como cada banda queda aislada en una franja diferente.

Realmente lo que ocurre es que, a partir de la imagen original, se crean otras tres en el caso de tres bandas a identificar. La primera imagen tendrá la primera franja y el resto de la imagen serán píxeles negros, la segunda imagen tendrá solo la banda central y el resto negro, y la tercera imagen la última banda. En el capítulo 6 se verá esto que se acaba de comentar, pero aquí se han unido las tres imágenes en una sola.

4.5 Perpendicularidad y altura del móvil respecto al suelo a la hora de tomar la imagen

A la hora de tomar la imagen, se tratará de que el móvil esté lo más perpendicular al suelo posible para así tratar de evitar ciertas perspectivas que puedan por ejemplo hacer que en la imagen las bandas no se perciban como equidistantes que es una de las suposiciones que haremos en la siguiente sección. Además, para seguir preservando la equidistancia lo máximo posible, el móvil se situará a una altura igual a la altura media de las bandas, si por ejemplo tenemos tres bandas que identificar, situaremos el móvil a la altura de la banda central. Se muestra ahora una imagen de perfil de la idea que se ha comentado.

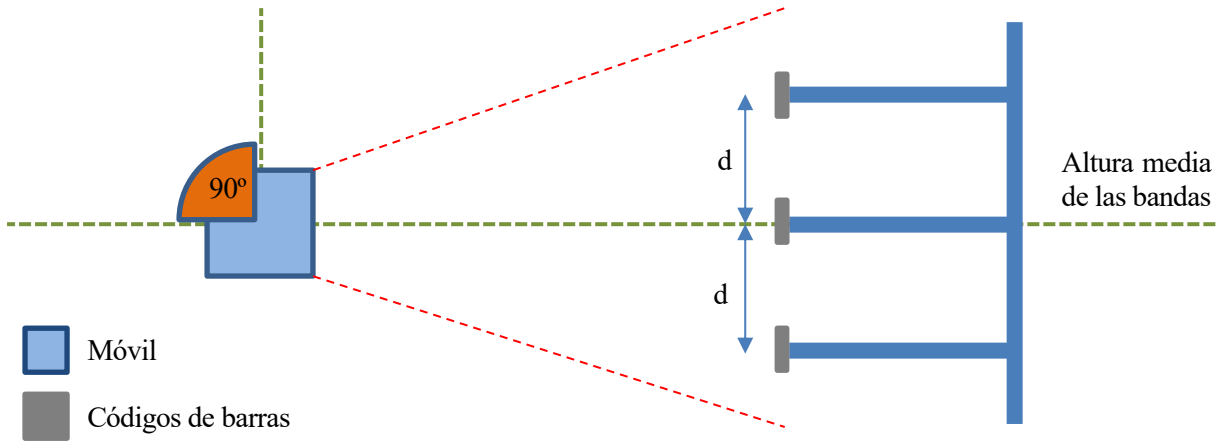

Figura 4-16. Colocación del móvil a la hora de tomar las imágenes.

A pesar de lo comentado en esta sección, no se dispone de un trípode por ejemplo para facilitar la tarea, así que, aunque se tomen a pulso las imágenes, se tratará de cumplir esto en la medida de lo posible.

También es necesario comentar que, en caso de usar el robot, evidentemente no vamos a poder situarlo de tal manera que la cámara se encuentre a una altura igual a la altura media de las bandas a identificar, pero dado que es un primer acercamiento a una funcionalidad en el mundo real, se hará así para facilitar la tarea.

4.6 Suposiciones a la hora de tomar la imagen

4.6.1 Bandas equidistantes

En primer lugar, el algoritmo cómo se ha visto en las secciones previas, considera que las bandas están separadas a la misma distancia unas de otras. En la práctica, esto se cumple en la mayoría de las estanterías de comercios, pero no en todas. Por ejemplo, en estanterías dónde hay envases grandes como packs de cajas de leche esto no se cumple.

Figura 4-17. Ejemplo de bandas no equidistantes.

Aquí se aprecia como las tres primeras bandas son equidistantes pero la última, no mantiene esa relación en la separación. Se ha tomado la imagen con el móvil en posición vertical, pero era simplemente para que entrasen todas las bandas bien, dado que esta imagen solo se va a usar para ilustrar un caso en el que no existe equidistancia, no tiene importancia la posición del móvil.

4.6.2 Banda superior e inferior cercanas a los límites de la imagen

También hemos comentado que, el primer paso es aislar cada banda en una franja diferente, luego si queremos que este paso cumpla esa función, debemos asegurarnos de que las bandas de los extremos, es decir, la que está más arriba y la que está más abajo, se encuentren cercanas al límite superior e inferior de la imagen respectivamente.

4.6.3 Número de bandas a identificar conocido

Es el único dato que el algoritmo necesita saber antes de ejecutarse y se le pasará al programa por la línea de comandos. Una vez se conoce el dato, el programa da valor a algunos parámetros en función del número de bandas que se le ha indicado que debe identificar.

4.6.4 Condiciones de luz adecuadas

Lógicamente, para que el algoritmo funcione correctamente, debemos tener unas condiciones de iluminación lo más favorables posibles. No obstante, el algoritmo considera, por ejemplo, el reflejo de la luz que puede producirse sobre la tira plástica que se pone sobre los códigos de barras para protegerlos y evitar que se caigan, o que aparezcan códigos de barras cortados por los límites laterales de la imagen.

4.6.5 Códigos de barras del mismo tamaño

En un paso del algoritmo, se utilizan transformaciones morfológicas para aislar los códigos de barras del resto de la imagen y se usan plantillas o también conocidas como kernels. Estas plantillas son de un cierto tamaño y en caso de que los códigos de barras no fuesen todos del mismo tamaño, podríamos eliminar algunos códigos de barras de los que debíamos identificar o al revés, es decir, identificar elementos de la imagen como códigos de barras cuando realmente no lo son, esto es lo que denominamos “falsos positivos”. En los siguientes capítulos explicaremos todos estos conceptos mencionados.

Aunque cabe mencionar que en la realidad sí que suelen ser todos los códigos del mismo tamaño en un mismo comercio, luego esto no supone ningún problema. Cuando hablemos de los resultados en el capítulo 11 comprobaremos como esto es así.

5 ALGORITMO DE RECONOCIMIENTO DE CÓDIGOS DE BARRAS

Antes de comenzar a explicar el algoritmo, es necesario comentar que la mayoría de los conocimientos aplicados, han sido extraídos de las diapositivas de la asignatura del cuarto curso "Sistemas de Percepción" y que denotaremos como [13].

5.1 Etapas del algoritmo

El algoritmo al completo lo podríamos dividir en 5 etapas:

- Etapa 1: identificación de las bandas horizontales → una primera etapa donde identificamos dónde se encuentran las bandas horizontales que contienen los códigos de barras.
- Etapa 2: zoom hacia cada una de las bandas identificadas → seguidamente se realiza un primer zoom a cada banda y se hace un barrido de la banda en cuestión en busca de códigos de barras.
- Etapa 3: identificación de los potenciales códigos de barras → consiste en identificar potenciales códigos de barras a partir de las imágenes obtenidas del barrido.
- Etapa 4: zoom hacia los posibles códigos de barras → realizamos un segundo zoom a los potenciales códigos de barras para decodificar la secuencia numérica correspondiente.
- Etapa 5: algoritmo de decodificación → finalmente, se aplica un algoritmo de decodificación de OpenCV para obtener las secuencias numéricas y a cada código se les asignan unas coordenadas (x, y, z) en el mapa.

En los siguientes capítulos, explicaremos en profundidad cada una de las etapas mencionadas, pero antes vamos a ver un esquema del algoritmo al completo.

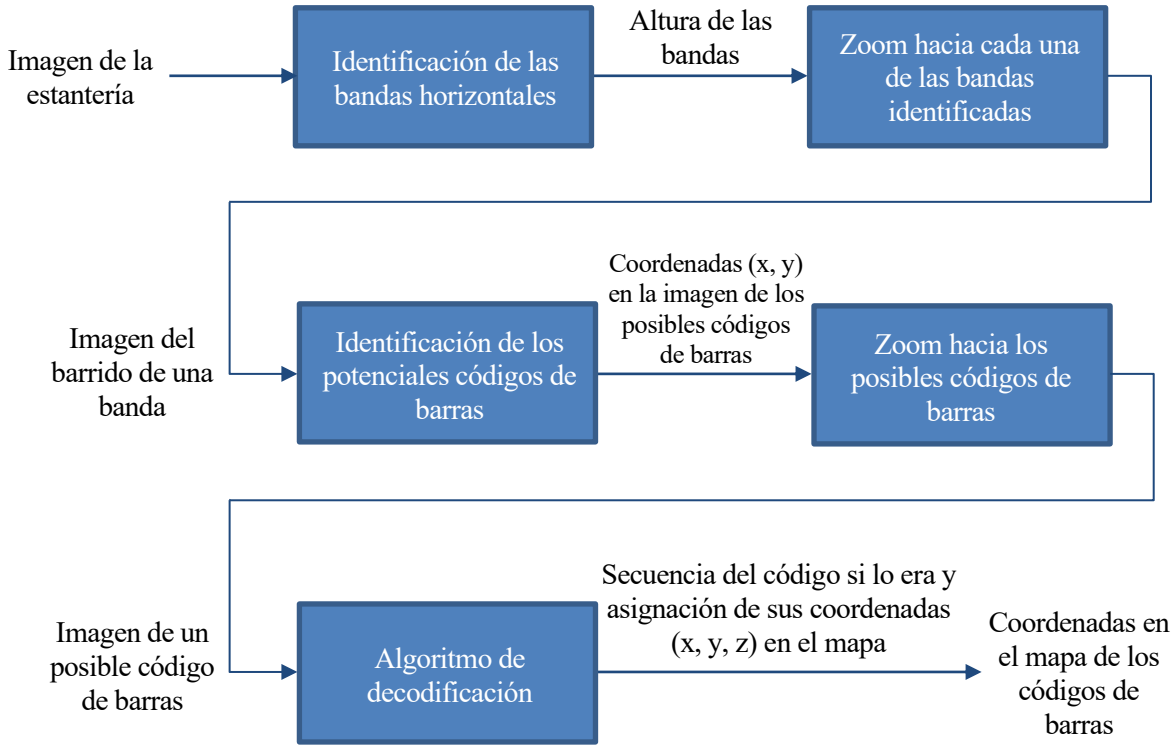

Figura 5-1. Esquema del algoritmo completo.

6 ETAPA 1: IDENTIFICACIÓN DE LAS BANDAS HORIZONTALES

6.1 Introducción

Como ya se ha introducido, esta primera etapa se encarga de identificar las bandas donde se encuentran los códigos de barras. Para ello, lo que se hace es obtener dos alturas de cada banda, una del límite superior y otra del límite inferior. Ahora se muestra una imagen de lo que se busca idealmente en esta etapa.

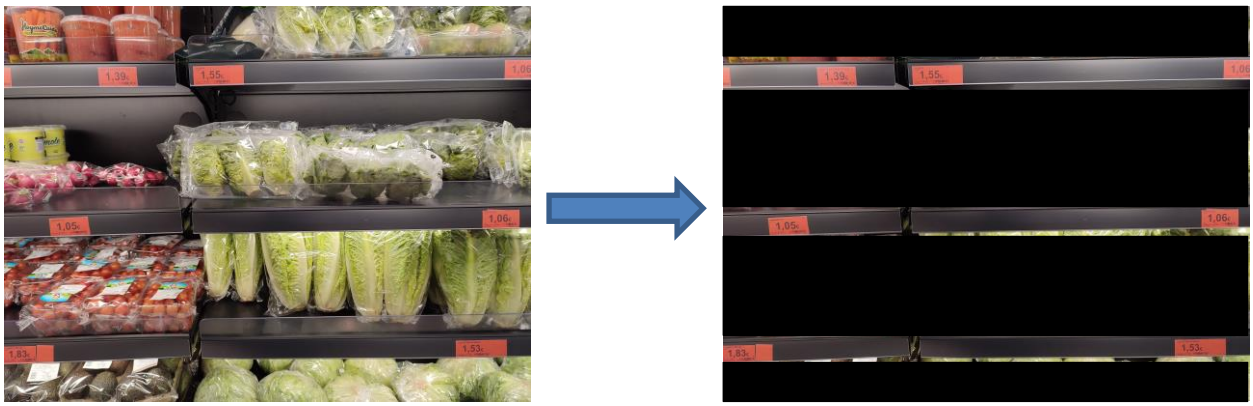

Figura 6-1. Objetivo de la etapa 1.

6.2 Diagrama de flujo

Comenzaremos mostrando un diagrama de flujo del algoritmo de la etapa uno.

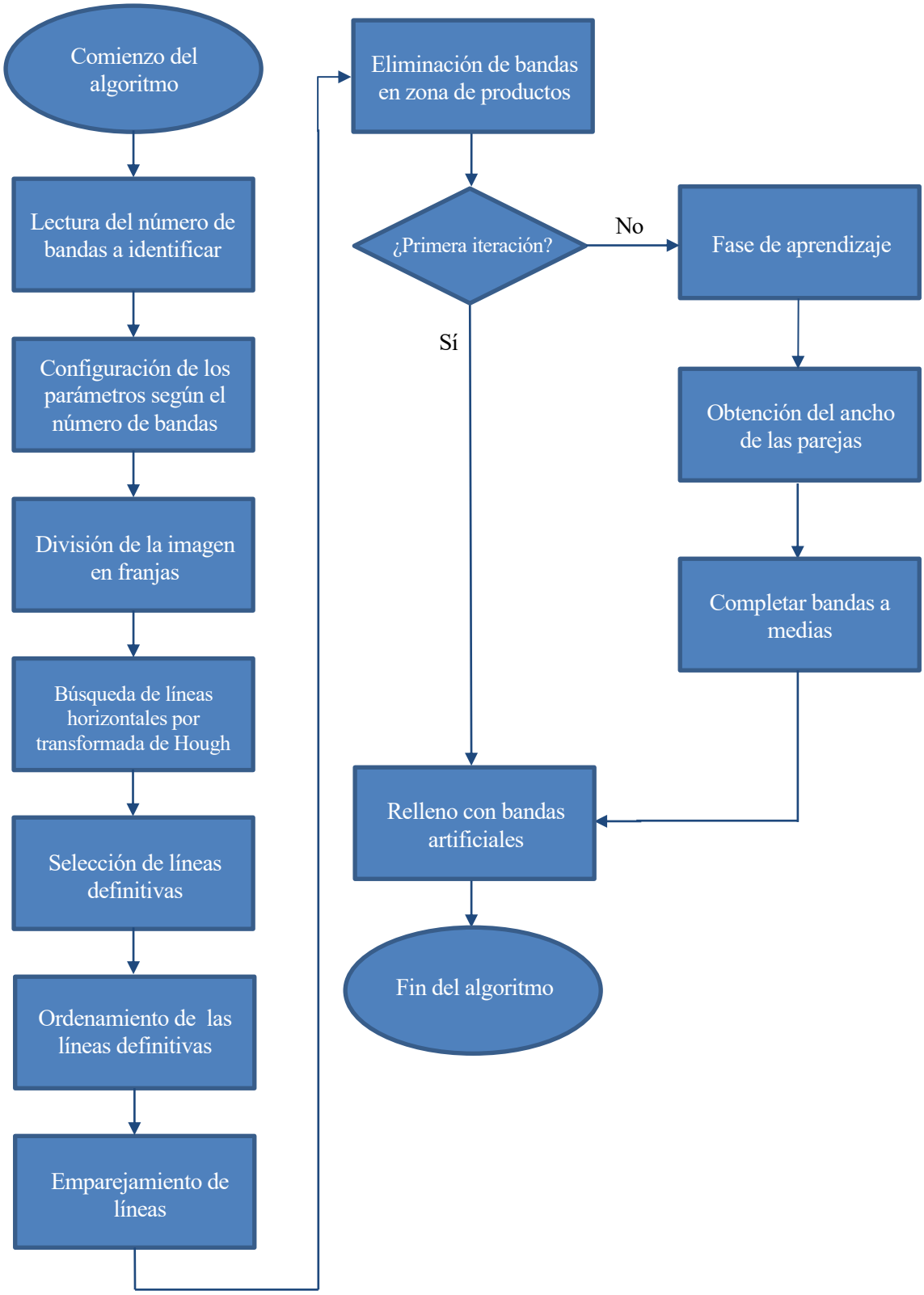

Figura 6-2. Diagrama de flujo de la etapa 1.

6.3 Explicación del algoritmo

En esta sección, iremos comentado los pasos que se han llevado a cabo para lograr aislar las bandas del resto de la imagen siguiendo el diagrama de flujo anterior.

6.3.1 Lectura del número de bandas a identificar

El primer paso será ejecutar el programa. Para ello, se pasará por la línea de comandos el número de bandas que se deben identificar en la imagen. El comando que se debe introducir consiste en cinco partes:

python + etapa1.py + número de bandas a identificar + número de comercio + número de secuencia

Tanto el número de comercio como el número de secuencia no influyen ahora, luego los ignoraremos por ahora y ya los veremos más adelante.

Suponiendo por ejemplo que se deben identificar tres bandas en el comercio 1 de la secuencia 1, el comando quedaría como sigue:

python etapa1.py 3 1 1

6.3.2 Configuración de los parámetros según el número de bandas

En este paso se llama a la función ***configuracion_numero_bandas*** que recibe como único parámetro el número de bandas a identificar y que, parametriza ciertas variables que serán necesarias en los pasos posteriores pero cuyos valores dependen del número de bandas.

Código 6-1. Función para configurar las variables según el número de bandas a identificar.

```
#Configuramos ciertas variables según el número de bandas que se deben identificar
def configuracion_numero_bandas(numero_bandas):
    print('------------------------------------------------------------
Configuración parámetros según número de bandas ---------------------------------
----------------------')
    if numero_bandas == 2:
        separacion = 175
        separacion3 = 450            #separacion2 + 100
        separacion2 = 350
        distancia_eliminar = 400

    elif numero_bandas == 3:
        separacion = 150
        separacion3 = 400            #separacion2 + 100
        separacion2 = 300
        distancia_eliminar = 350

    elif numero_bandas == 4:
        separacion = 115
        separacion3 = 275            #separacion2 + 100
        separacion2 = 175
        distancia_eliminar = 275

    print('Se configuran los parámetros para', numero_bandas, 'bandas')
    print('Separación para unir líneas:', separacion)
    print('Separación para formar pareja grande:', separacion3)
    print('Separación para formar pareja pequeña:', separacion2)
    print('Separación para eliminar parejas muy próximas:', distancia_eliminar)
    print('------------------------------------------------------------------------
------------------------------------------------------------------------
--------')
    print('')

    return separacion, separacion3, separacion2, distancia_eliminar
```

6.3.3 División en franjas y búsqueda de líneas horizontales por transformada de Hough

Como ya se comentó en la sección 4.4, se trata de dividir la imagen en franjas horizontales iguales aislando cada banda en cada una de las franjas y facilitando así, la identificación de las líneas horizontales que delimitan por arriba y por debajo cada una de las bandas. Veremos ahora cómo funciona siguiendo con un ejemplo de tres bandas.

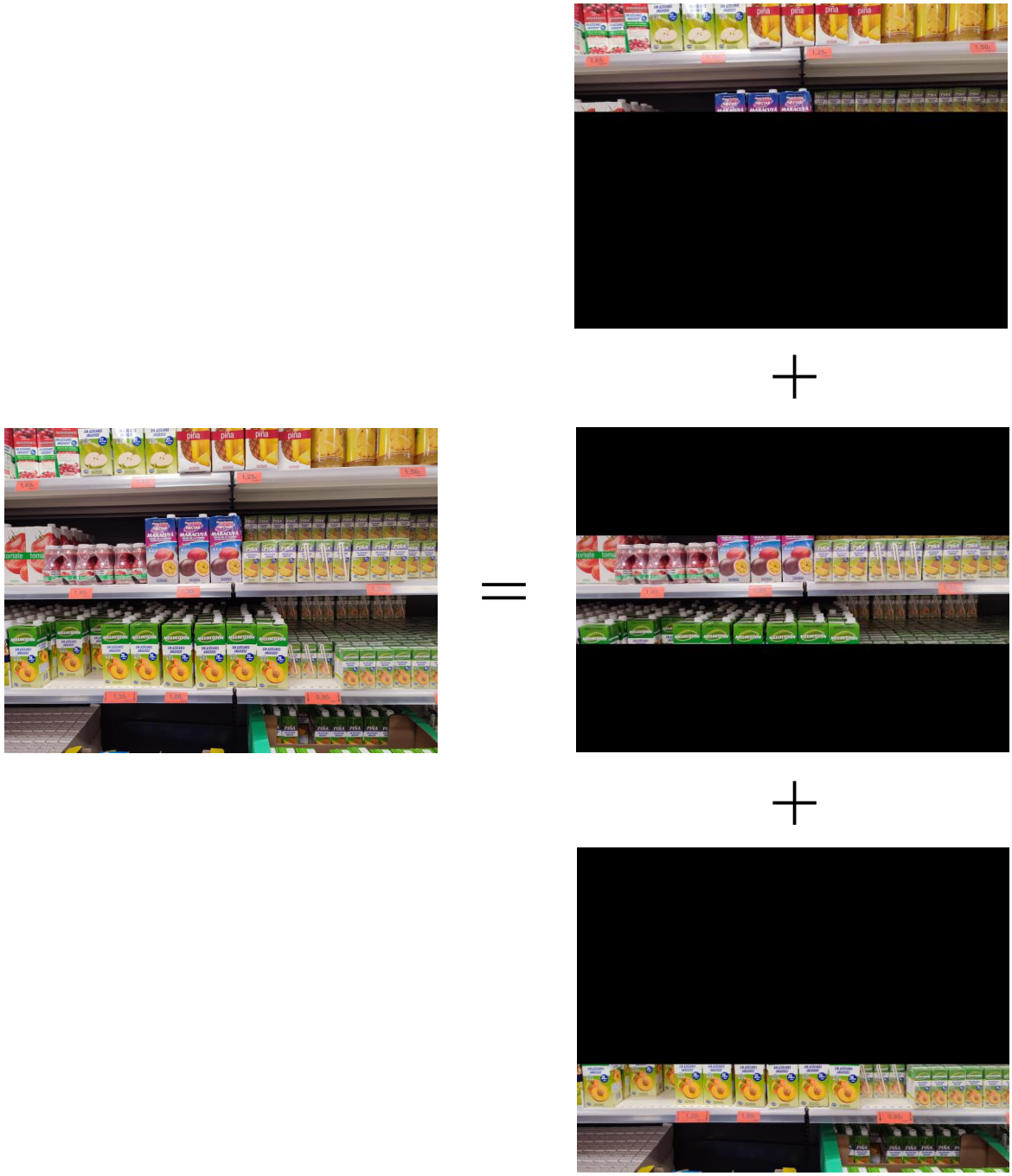

Figura 6-3. Ejemplo de división de la imagen en franjas con tres bandas.

Pero antes de pasar al código es necesario explicar el operador Canny y la transformada de Hough. La transformada de Hough [13] recibe como entrada una imagen binaria que contiene los contornos de la imagen. Una forma de obtener estos contornos de manera sencilla es usando el operador Canny al cual se le pasan dos umbrales. El operador Canny calcula para cada píxel el módulo del gradiente y evalúa si está por encima del umbral superior, en ese caso, se clasifica como contorno seguro. Por otro lado, si tenemos un píxel que está entre el límite superior e inferior, pero a la vez está conectado a un contorno seguro, también se incluye al

contorno, pero si no está conectado a un contorno seguro se descarta al igual que los píxeles que están por debajo del límite inferior. El módulo del gradiente se calcula como:

$$G = \sqrt{G_x^2 + G_y^2} \tag{6-1}$$

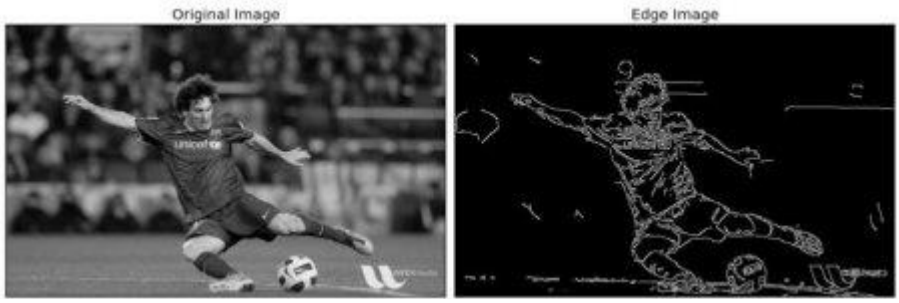

Figura 6-4. Ejemplo de obtención de contornos mediante el operador Canny [6].

Aplicando este operador a la imagen que usaremos de aquí en adelante obtenemos lo siguiente:

Figura 6-5. Imagen de partida.

Figura 6-6. Operador Canny aplicado a la figura 6-5.

En cuanto a la transformada de Hough [13], sirve para identificar formas en la imagen binaria de contornos como por ejemplo líneas, aunque también puede usarse para identificar otras formas como circunferencias. La idea es que se recorre la imagen binaria y en cada píxel del contorno, se evalúan todas las posibles rectas que pasan por ese píxel, cada recta que se evalúe votará a un par (ρ, θ).

La recta evaluada se representa en coordenadas polares con los parámetros (ρ, θ) donde el primero representa la distancia mínima de la esquina superior izquierda a la recta, y el segundo, el ángulo en sentido horario entre el borde superior de la imagen y la recta normal a la recta que se está evaluando. La ecuación de una recta en coordenadas polares es:

$$x \cdot \cos(\theta) + y \cdot \sin(\theta) = \rho \qquad (6\text{-}2)$$

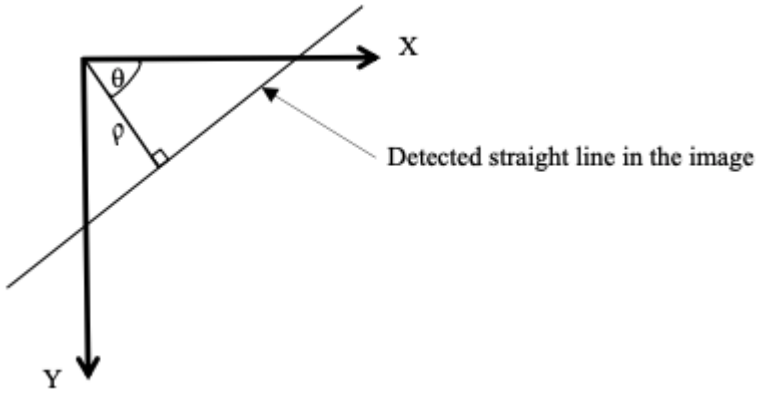

Figura 6-7. Parámetros ρ y θ para la transformada de Hough [14]

En nuestro caso, al tratarse de líneas horizontales, el parámetro ρ coincide con la altura de la línea horizontal (coordenada "y"), y θ serán 90º.

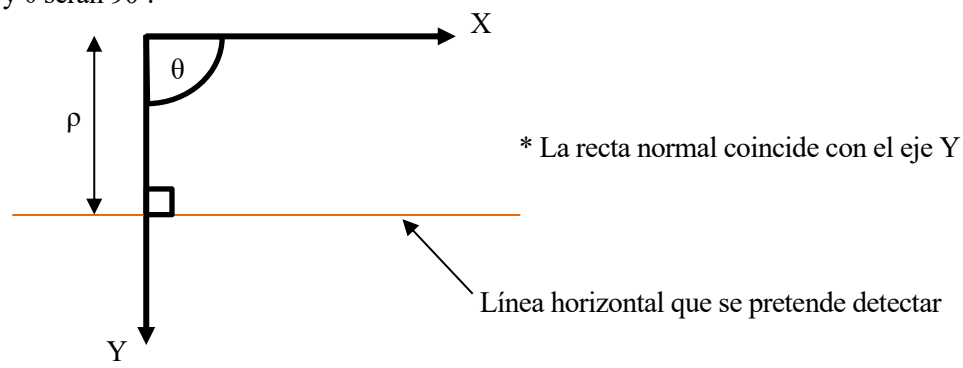

Figura 6-8. Parámetros ρ y θ en el caso de líneas horizontales.

Tras recorrer la imagen completa, obtenemos lo que se llama matriz de votos de coordenadas (ρ, θ, número de votos), los pares (ρ, θ) con más votos, serán las líneas de la imagen. La matriz tendría la siguiente estructura:

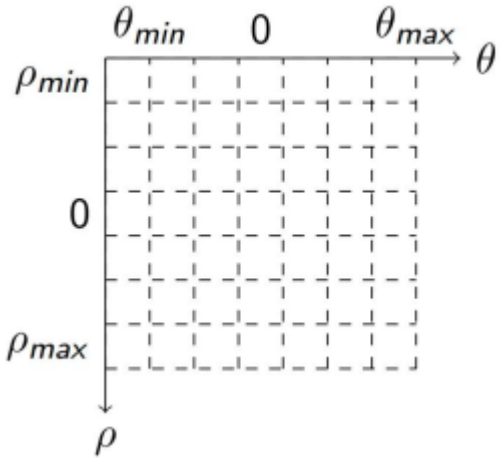

Figura 6-9. Matriz de votos de la transformade de Hough [13].

Una vez explicado esto podemos volver al código. La función que divide la imagen en franjas es ***Hough_franjas***. Esta función dividirá la imagen en tantas partes iguales como bandas tengamos que identificar. La variable **vector_alturas_unidas** devuelta por la función, es un vector que agrupa todas las alturas (componente ρ) de las líneas horizontales detectadas en todas las franjas en que se ha dividido la imagen. También se devuelven los ángulos correspondientes a esas líneas.

Se mostrará el código para el caso de dos bandas, ya que, el resto es lo mismo, pero extrapolando para un número distinto de bandas. El programa contempla el caso de 2, 3 y 4 bandas a identificar que es lo común que se encuentra en un comercio.

El argumento **numero_lineas_a_detectar** que se le pasa a la función tiene un valor de 10 para cualquier número de bandas.

Código 6-2. Función para dividir la imagen en franjas y que a su vez llama la función Hough.

```
#Divide la imagen en tantas franjas horizontales como número de bandas a
identificar. Además, le calcula la transformada de Hough a cada banda en busca de
líneas horizontales
def Hough_franjas(numero_bandas, height, width, image, numero_lineas_a_detectar):
    edges, lines = Hough(image, 30)
    plot_franjas = 0
    plot_edges = 0

    if numero_bandas == 2:
        cuarto1 = numpy.zeros((height, width),numpy.uint8)
        cuarto1[0 : int(height/2), :] = 1

        cuarto2 = numpy.zeros((height, width),numpy.uint8)
        cuarto2[int(height/2) : height, :] = 1

        image1 = cv2.bitwise_and(image, image, mask=cuarto1)
        image2 = cv2.bitwise_and(image, image, mask=cuarto2)

        if plot_franjas == 1:
            plt.subplot(221),plt.imshow(image,cmap = 'gray')
            plt.title('Imagen original'), plt.xticks([]), plt.yticks([])

            plt.subplot(222),plt.imshow(image1,cmap = 'gray')
            plt.title('1ª franja'), plt.xticks([]), plt.yticks([])

            plt.subplot(223),plt.imshow(image2,cmap = 'gray')
            plt.title('2ª franja'), plt.xticks([]), plt.yticks([])
            plt.show()

        edges1, lines1 = Hough(image1, numero_lineas_a_detectar)
        img_copy11 = pintar_lineas(image, height, width, lines1, None, None)
#Para pintar todas las líneas detectadas en la franja 1

        edges2, lines2 = Hough(image2, numero_lineas_a_detectar)
        img_copy12 = pintar_lineas(image, height, width, lines2, None, None)
#Para pintar todas las líneas detectadas en la franja 2

        #Hay que unir los 2 elementos "lineX"
        vector_alturas_unidas = []
        vector_angulos_unidos = []
        for i in lines1:
            vector_alturas_unidas.append(i[0][0])
            vector_angulos_unidos.append(i[0][1])
        for i in lines2:
            vector_alturas_unidas.append(i[0][0])
            vector_angulos_unidos.append(i[0][1])

        if plot_edges == 1:
            plt.subplot(221),plt.imshow(img_copy11)
            plt.title('Cuarto 1'), plt.xticks([]), plt.yticks([])

            plt.subplot(222),plt.imshow(img_copy12)
            plt.title('Cuarto 2'), plt.xticks([]), plt.yticks([])
            plt.show()

    return edges, vector_alturas_unidas, vector_angulos_unidos
```

Esta función llama a su vez a la función ***Hough,*** que como su nombre indica, simplemente calcula la transformada de Hough en busca de líneas horizontales. Lo que se ha hecho para filtrar las líneas y quedarnos solo con las más votadas ha sido ir incrementando el parámetro **threshold** de la función de OpenCV **HoughLines** de 25 en 25 votos hasta que el número de líneas detectadas sea igual o inferior a 10. Este parámetro es el número de votos mínimo que debe tener un determinado par (ρ, θ) para que la función lo devuelva, si tiene menos votos, no aparecerá en las líneas detectadas que devuelve la función.

Además, esta función permite buscar rectas con un θ determinado, que en nuestro caso se ha visto en la figura 6-8 que serán 90º, aunque se ha dejado un margen de ± 2,5º para contemplar posibles errores al tomar la imagen a pulso. Cabe mencionar que se llama a la función ***Hough*** y se le pasa cada franja de manera aislada, es decir, si tenemos tres bandas por ejemplo, se llama tres veces a la función, una con cada franja.

Código 6-3. Función que calcula la transformada de Hough.

```
#Función que calcula la transformada de Hough en busca de líneas horizontales (rho
resolución 1 píxel, theta = 90° +- 2.5°)
def Hough(imagen, numero_lineas_detectadas):
    print('-------------------------------------------------------------------
Búsqueda de líneas horizontales ---------------------------------------------
-----------------')
    edges = cv2.Canny(imagen,125,225,apertureSize=3,L2gradient=True)
    #Búsqueda de líneas horizontales: en el rango [87.5°,92.5°] -> aumento el
umbral (threshold) de 25 en 25 hasta quedarme con menos de 10 líneas detectadas
    lineas_detectadas = 2000
    umbral = 100
    while(lineas_detectadas>numero_lineas_detectadas):
        lines =
cv2.HoughLines(edges,rho=1,theta=numpy.pi/180,threshold=umbral,srn=0,stn=0,min_thet
a=numpy.pi/2-2.5*numpy.pi/180,max_theta=numpy.pi/2+2.5*numpy.pi/180)
        lineas_detectadas = len(lines)
        print('Líneas detectadas:',len(lines),'---','Umbral:', umbral)
        umbral += 25
    print('-------------------------------------------------------------------
-------------------------------------------------------------------------------
--------')
    print('')

    return edges, lines
```

En el caso de que no dividiésemos la imagen en franjas, la transformada de Hough no nos daría unos resultados tan buenos. Para demostrarlo, vamos a mostrar ahora una imagen de la transformada de Hough calculada a la imagen completa directamente y, cuando la calculamos a cada una de las franjas de manera independiente.

Figura 6-10. Transformada de Hough aplicada a la imagen completa directamente.

Figura 6-11. Transformada de Hough aplicada a la imagen por franjas.

Para esta última imagen, realmente se obtienen 3 imágenes, pero se ha condensado en una para evitar poner tantas imágenes de nuevo. Se obtendrían 3 imágenes, cada una contendría una franja y las líneas horizontales detectadas en esa franja.

Tanto en la figura 6-10 como en la figura 6-11 se calcula más o menos el mismo número de líneas, dependerá de las iteraciones cuando se vaya aumentando el número de votos. Será un máximo de 30 líneas en el caso de aplicar la transformada de Hough directamente a toda la imagen, y de 10 en cada franja lo que resulta en otras 30 líneas en total. Pero como se puede ver, a pesar de detectar aproximadamente el mismo número de líneas en ambas imágenes, el segundo resultado es mejor porque identificamos los límites superiores e inferiores de las tres bandas, mientras que al aplicar Hough directamente a la imagen original las líneas se concentran en la franja central.

Esta idea de buscar los límites superiores e inferiores de cada banda surge de [15], donde se propone identificar las diferentes líneas verticales blancas y negras que componen el código de barras usando la transformada de Hough para identificar líneas verticales. Pero como es de esperar, en nuestro caso donde los códigos de barras se ven tan pequeños en la imagen de partida, no podemos hacer esto, y es cuando surge la idea de primero aislar las bandas donde se encuentran los códigos de barras para reducir considerablemente la zona de búsqueda.

Como podemos ver, para una misma línea horizontal en la imagen tenemos varias líneas detectadas, por ello el siguiente paso será agrupar todos aquellos pares (ρ, θ) que representan la misma línea horizontal en uno solo.

6.3.4 Selección de líneas definitivas

Recordemos que para cada franja calculábamos un máximo de 10 parejas de valores (ρ, θ), pero estas parejas pueden representar la misma línea, luego para facilitar la tarea y reducir el número de líneas con las que trabajar en los pasos posteriores, se agrupan usando la función ***seleccion_lineas_definitivas***.

Es en este momento cuando se usa una de las variables parametrizadas en la función ***configuracion_numero_bandas***. Dicha variable es **separacion**, esta variable indica que, si dos rectas tienen una separación o, mejor dicho, diferencia de altura (componente "y" / ρ) inferior a esos píxeles, entonces consideraremos que ambos pares (ρ, θ) representan la misma línea horizontal en la imagen y, por tanto, nos quedaremos solo con uno de los pares.

Código 6-4. Función para agrupar los pares (ρ, θ) que representan la misma línea horizontal.

```
#Función para agrupar los pares (rho, theta) que representan la misma horizontal
def seleccion_lineas_definitivas(vector_alturas_unidas, vector_angulos_unidos,
separacion):
    vector_alturas = []
    vector_angulos = []
    primera_iter = 1
    distinto = 1

    print('-------------------------------------------------------------------
Selección de líneas definitivas ----------------------------------------------
-----------------')

    for i in range(len(vector_alturas_unidas)):
        print('rho:',vector_alturas_unidas[i])
        distinto = 1
        if primera_iter == 1:
            vector_alturas.append(vector_alturas_unidas[i])
            vector_angulos.append(vector_angulos_unidos[i])
            primera_iter = 0

        if primera_iter == 0:
            for j in vector_alturas:
                print('Elemento vector:',j)
                if abs(vector_alturas_unidas[i] - j) < separacion:
#Líneas separadas menos de "separacion"" píxeles se considera que representan la
misma horizontal
                    print('Misma línea------------------------------')
                    print('Vector_alturas:',vector_alturas)
                    print('Vector_angulos:',vector_angulos)
                    print('')
                    distinto = 0
                    break

            if distinto == 1:                                    #Una vez
comprobado que es distinto a todos los elementos ya existentes en "vector_alturas"
                vector_alturas.append(vector_alturas_unidas[i])        #lo añadimos
                vector_angulos.append(vector_angulos_unidos[i])
                print('Añadimos línea---------------------------------')
                print('Vector_alturas:',vector_alturas)
                print('Vector_angulos:',vector_angulos)
                print('')
    print('-------------------------------------------------------------------
-------------------------------------------------------------------------------
--------')

    return vector_alturas, vector_angulos
```

Básicamente se inserta en los vectores **vector_alturas** y **vector_angulos** el primer par (ρ, θ), ρ en el primer vector y θ en el segundo. Una vez hecho eso, con el resto de los pares lo que se hace es comprobar si alguno de los pares ya añadidos a **vector_alturas**, está cercano al par que estamos comprobando. Si tras comprobar todos los elementos de **vector_alturas** no es cercano a ninguno, se añade el par a ambos vectores, en caso contrario, se desecha porque ya hay un par que representa la misma línea que el par que estamos estudiando. El resultado obtenido por ejemplo para la figura 6-11 es el siguiente.

Figura 6-12. Selección de las líneas horizontales definitivas.

6.3.5 Ordenamiento de las líneas definitivas

Este paso sencillamente ordena los pares (ρ, θ), o también podemos decir las líneas horizontales definitivas, de la que está más arriba en la imagen (la que tenga un ρ o componente "y" más bajo), a la que está más abajo. Esto será necesario para facilitar el paso siguiente cuando tratemos de emparejar dos líneas que estén próximas entre sí para formar bandas con su límite superior e inferior.

La función encargada de esta tarea es ***ordena_alturas*** y se muestra a continuación.

Código 6-5. Función para ordenar las líneas definitivas de menor a mayor altura.

```
#Función que ordena los pares (rho, theta) definitivos de par con menor rho
a par con mayor rho
def ordena_alturas(vector_alturas, vector_angulos):
    tam_vector = len(vector_alturas)
    print('Tamaño vector:',tam_vector)

    alturas_ordenadas, angulos_ordenados = zip(*sorted(zip(vector_alturas,
vector_angulos)))
    print('Vector alturas ordenados:',alturas_ordenadas)
    print('Vector ángulos ordenados:',angulos_ordenados)
    print('')

    return tam_vector, alturas_ordenadas, angulos_ordenados
```

De igual manera que las alturas se ordenan los ángulos, es decir, si por ejemplo una altura que estaba en la tercera posición y ahora es la primera, su ángulo correspondiente pasa ahora a estar también en la primera posición, pero del vector de ángulos ordenados. Se ilustra mejor ahora con una tabla.

Tabla 6-1. Ejemplo de la función ordena_alturas.

vector_alturas (píxeles)	vector_angulos (radianes)	ordena_alturas (vector_alturas, vector_angulos)	alturas_ordenadas (píxeles)	angulos_ordenados (radianes)
1500	1.562		250	1.579
2000	1.562		1500	1.562
250	1.579		2000	1.562

Los ángulos como veremos en el capítulo 11 de los resultados son todos prácticamente 90º como es de esperar y, realmente no se usan, pero se ha visto conveniente guardarlos y ordenarlos también en un vector para mantener la idea de los pares (ρ, θ).

6.3.6 Emparejamiento de líneas

Una vez seleccionadas y ordenadas las líneas definitivas con las que trabajaremos, es el momento de agruparlas para formas parejas. Para ello se utiliza la función ***emparejamiento_lineas*** que utiliza dos de las variables configuradas en la función ***configuracion_numero_bandas***. Estas dos variables son **separacion2** y **separacion3**.

Como las alturas ya están ordenadas gracias a la función ***ordena_alturas***, simplemente iremos recorriendo el vector de la línea más alta (la que tenga menos ρ), e iremos viendo si la podemos emparejar con la línea siguiente o dos posiciones por delante de la actual que estamos intentando emparejar.

Separacion2 es un umbral para emparejar las líneas, es decir, si dos líneas están a una separación menor a **separacion2**, entonces formamos una pareja con esas dos líneas. **Separacion3** actúa igual que **separacion2**, pero sirve para contemplar el caso en que tres líneas estén próximas entre sí, en ese caso nos quedamos con la de más arriba y la de más abajo ignorando la que está en medio.

Antes de pasar al código es necesario hablar de la variable **vector_mascara**, es la más importante del algoritmo. Contiene las alturas de las parejas que se van formando ya sea en este paso o en pasos siguientes. Para obtener una imagen como la figura 6-1, se pasa **vector_mascara** como argumento de la función ***creacion_mascara*** obteniendo una matriz binaria del mismo tamaño que la imagen original. Dicha matriz solo contiene unos y ceros. Los unos estarán en aquellos píxeles cuya altura esté entre el límite superior y el límite inferior de alguna de las parejas formadas.

Código 6-6. Función para crear una máscara binaria.

```
#Función que recibe "vector_mascara" con las alturas de los límites de las
bandas y crea una máscara que al aplicarla a la imagen, nos deja solo las
bandas y el resto de la imagen a (0, 0, 0) que es el color negro
def creacion_mascara(height, width, vector_mascara, flag):
    mascara = numpy.zeros((height, width),numpy.uint8)
    for h0, h1 in vector_mascara:
        lim_inferior = int(h0)
        lim_superior = int(h1)
        #Caso para las parejas detectadas directamente a través de las
línea de Hough
        if flag == 1:    #Límite superior - 20 píxeles y límite inferior +
20 píxeles para en caso de que las líneas detectadas no sean completamente
horizontales, asegurarnos que en la banda entra todo el código de barras
            mascara[lim_inferior - 20 : lim_superior + 20, :] = 1
        else:    #Para la creación de bandas artificiales
            mascara[lim_inferior : lim_superior, :] = 1

    return mascara
```

Como se aprecia, se sube la altura del límite superior en 20 píxeles y se baja la altura del límite inferior en otros 20 píxeles para crear un margen de 40 píxeles en total, que contemple cuando las líneas identificadas no sean completamente horizontales y asegurarnos de esta manera, que en la banda creada entra todo el código de barras. Se muestra ahora un ejemplo en el que se realizan todos los pasos explicados hasta el emparejamiento de líneas.

Figura 6-13. Imagen para aplicar la máscara binaria obtenida de vector_mascara tras emparejar líneas.

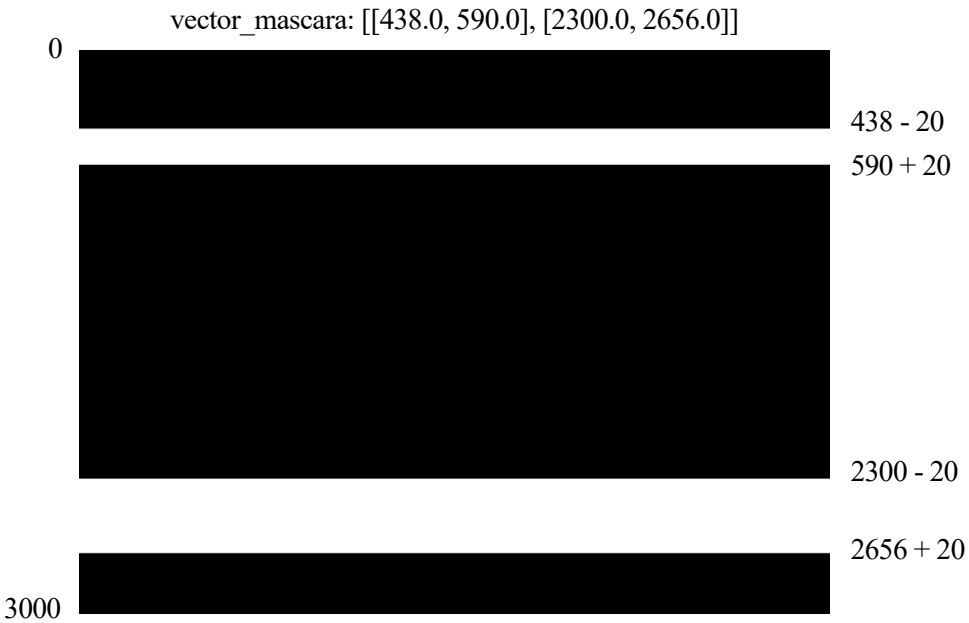

Figura 6-14. Creación de máscara binaria a partir de vector_mascara.

En definitiva, los píxeles que no estén entre alguna de las alturas de las parejas serán píxeles negros con valor 0, y los píxeles que estén entre alguna de las alturas de las parejas serán píxeles blancos con valor 1. Finalmente, aplicando la operación AND de esta máscara con la imagen original obtenemos lo siguiente.

Figura 6-15. Resultado de aplicar la máscara con las parejas formadas a la imagen original.

Código 6-7. Función para formar parejas de líneas (I).

```
#Miramos si dos pares están cerca, si es así, los agrupamos formando una banda
def emparejamiento_lineas(tam_vector, alturas_ordenadas, height, numero_bandas,
separacion3, separacion2):
    print('-------------------------------------------------------------------
Emparejamiento de líneas ------------------------------------------------------
----------')
    vector_mascara = []             #Vector de parejas de alturas
    alturas = []                  #Vector de alturas definitivas ordenadas
    vector_desechadas = []          #Vector de líneas desechadas (las que no se
emparejan y se quedan solas)
    indice_pareja = 0               #Para saber en qué índice de "vector_mascara"
debo meter las líneas desechadas una vez las rellene
    vector_indices = []           #Vector que contiene los índices en los que se
deben meter las líneas desechadas en "vector_mascara"
    i = 0
    while i < tam_vector - 1:                #Ejemplo: 6 líneas -> i va [0, 4] < 6
- 1 = 5
        altura1 = alturas_ordenadas[i]         #Si "i" tiene valor correspondiente
al último elemento de la lista, no lo podemos emparejar con ninguno,
        altura2 = alturas_ordenadas[i+1]     #luego el máximo valor de "i" es el
penúltimo elemento
        if i < tam_vector - 2:                 #Ejemplo: 6 líneas -> i va [0, 3] < 6
- 2 = 4
            altura3 = alturas_ordenadas[i+2]
            print('Alturas:', altura1, '-', altura2, '-', altura3)
        else:
            print('Alturas:', altura1, '-', altura2)

        if abs(altura3 - altura1) <= separacion3:#Caso de 3 líneas muy juntas,
nos quedamos con la dos más exteriores
            vector_mascara.append([altura1, altura3])
            alturas.append(altura1)
            alturas.append(altura3)
            i = i + 3
            print('Líneas emparejadas:', [altura1, altura3])
            print(vector_mascara)
            print('')
        elif abs(altura2 - altura1) <= separacion2:         #Líneas separadas menos
de "separacion2" píxeles -> pareja de líneas
            vector_mascara.append([altura1, altura2])
            alturas.append(altura1)                        # "altura1" es la línea
de arriba y "altura2" es la línea de abajo
            alturas.append(altura2)
            i = i + 2
            print('Líneas emparejadas:', [altura1, altura2])
            print(vector_mascara)                                  #Incrementamos el
índice de las líneas desechadas solo cuando formamos
            print('')                                             #pareja, ya sea
altura3-altura1 o altura2-altura1
        else:
            vector_desechadas.append(altura1)
            vector_indices.append(indice_pareja)
            i = i + 1
            print('Línea desechada:', altura1)
            print('Vector desechadas:', vector_desechadas)
            print('')
        indice_pareja = indice_pareja + 1
```

Código 6-8. Función para formar parejas de líneas (II).

```
        print('Valor de "i" al salir del bucle de emparejamiento:', i)
        print('')

        #Si i es el índice del último elemento del vector, no lo podemos emparejar
    con otra línea, luego añadimos esa línea al resto de líneas desechadas
        if i == tam_vector - 1:
            altura = alturas_ordenadas[i]
            vector_desechadas.append(altura)
            vector_indices.append(indice_pareja)
            print('Última línea se queda sola para emparejar -> se añade a las
    líneas desechadas')
            print('')

        numero_de_parejas = len(vector_mascara)
        numero_lineas_desechadas = len(vector_desechadas)

        separacion_teorica = int(height / numero_bandas)
        vector_ocupacion = numpy.zeros(numero_bandas)    #Para saber qué posiciones
    están ocupadas (0 libre - 1 ocupado)

        for i in range(numero_de_parejas):        #Ubico cada pareja en
    "vector_ocupacion" para saber qué bandas son las que hay que crear
    artificialmente
            ubicado = 0
            indice_pareja = 1
            altura_de_abajo = vector_mascara[i][1]               #Límite inferior
    de las parejas ya formadas

            #Averiguo qué posición tiene cada pareja en el "vector_ocupacion"
            while ubicado == 0:
                factor = indice_pareja * separacion_teorica
                if altura_de_abajo <= factor:
                    vector_ocupacion[indice_pareja - 1] = 1
                    ubicado = 1
                indice_pareja = indice_pareja + 1

        print('Vector máscara:', vector_mascara)
        print('Vector alturas:', alturas)
        print('Número de parejas:', numero_de_parejas)
        print('Vector ocupación:', vector_ocupacion)

        print('Vector líneas desechadas:', vector_desechadas)
        print('Índices líneas desechadas:', vector_indices)
        print('Número de líneas desechadas:', numero_lineas_desechadas)
        print('')
        print('--------------------------------------------------------------------
    --------------------------------------------------------------------------
    ----------------')

        return vector_mascara, alturas, vector_ocupacion, vector_desechadas,
    vector_indices, numero_de_parejas, numero_lineas_desechadas
```

Las líneas que no han sido emparejadas se añaden a **vector_desechadas**, en pasos posteriores volveremos a este vector para ver si podemos utilizar alguna de las líneas de este vector.

El último bucle for de la función sirve para darle valor a **vector_ocupacion**, se trata de un vector muy importante del algoritmo y este nos indica qué bandas faltan por detectar. Al igual que ocurría a la hora de dividir la imagen en franjas, para rellenar el **vector_ocupacion** es también necesario que la banda de más arriba y la de más abajo estén próximas a los límites superior e inferior de la imagen respectivamente. Si por ejemplo tenemos tres bandas a identificar y tenemos parejas formadas solo en la primera y la última franja, el vector tendría valor [1, 0, 1], donde 1 indica que hay una pareja formada en esa franja y 0 que aún no. El primer valor de **vector_ocupacion** corresponde a la franja de más arriba y así sucesivamente hacia abajo.

Figura 6-16. Ejemplo de cómo se rellena el vector_ocupacion.

Básicamente lo que hacemos es mirar qué altura tienen las parejas formadas y ubicarlas en las franjas correspondientes para saber en qué franjas no hay parejas formadas aún. Es importante aclarar que aquí no se comprueba si las bandas detectadas son correctas o no, es decir, si contienen las bandas donde están los códigos de barras, solo se comprueba en qué franjas tenemos parejas formadas.

6.3.7 Eliminación de bandas en zona de productos

La finalidad de este paso es eliminar una de las bandas en el caso de que existan dos parejas formadas que sean próximas entre sí. Tan solo se ha dado una vez en todas las imágenes que se han probado, pero como mecanismo de seguridad se deja. Para ello se utiliza la función ***eliminacion_bandas_productos***. Esta función utiliza la última variable parametrizada según el número de bandas a identificar en ***configuracion_numero_bandas*** y que es **distancia_eliminar**. Esta variable representa la distancia mínima a la que deben estar separadas dos bandas para no eliminar una de ellas. Para realizar esta tarea, se calcula la diferencia de altura entre el límite inferior de la banda de arriba y el límite superior de la banda de abajo.

Figura 6-17. Cálculo de la distancia entre bandas para eliminar una de ellas.

$$\text{Si } d_1 < distancia\_eliminar \Rightarrow \text{Se elimina la banda 2} \quad (6\text{-}3)$$

$$\text{Si } d_2 < distancia\_eliminar \Rightarrow \text{Se elimina la banda 3} \quad (6\text{-}4)$$

En este caso, aunque no tengamos valores cuantitativos, se ve que la banda dos está muy próxima a la banda uno, luego se elimina la banda dos.

Figura 6-18. Eliminación de banda en zona de productos.

Y el código de la función es el siguiente.

Código 6-9. Función para eliminar la banda de abajo cuando dos bandas están próximas entre sí.

```
#Si al formar parejas en la función "emparejamiento_lineas()", tenemos dos bandas
muy próximas, eliminamos la banda inferior. En la memoria explicaremos por qué se
ha decidido eliminar la banda que está más abajo de las dos
def eliminacion_bandas_productos(height, numero_bandas, vector_mascara,
vector_ocupacion, numero_de_parejas, distancia_eliminar):
    print('------------------------------------------------------- Eliminación
bandas en productos ------------------------------------------------------------
-----')
    print('Vector máscara:', vector_mascara)
    print('')
    valor_auxiliar = 0

    #Bandas a una separación inferior a "distancia_eliminar" píxeles, se elimina la
banda de abajo
    if len(vector_mascara) > 1:
        for i in range(len(vector_mascara) - 1):    #Ejemplo: 4 bandas detectadas -
> i va [0, 2] < 4 - 1 = 3
            print('Índice:',i,'---',vector_mascara[i-valor_auxiliar][1],'---
',vector_mascara[i+1-valor_auxiliar][0])
            if abs(vector_mascara[i-valor_auxiliar][1]-vector_mascara[i+1-
valor_auxiliar][0]) < distancia_eliminar:    #Parejas muy próximas -> eliminamos la
de abajo
                print('Pareja eliminada:', vector_mascara[i+1-valor_auxiliar])
                vector_mascara.pop(i+1-valor_auxiliar)
                print('Vector máscara tras eliminar pareja en zona de productos:',
vector_mascara)
                valor_auxiliar = valor_auxiliar + 1
                numero_de_parejas = numero_de_parejas - 1
            print('')

    #Recalculamos el "vector_ocupacion"
    separacion_teorica = int(height / numero_bandas)
    print('Numero de parejas', numero_de_parejas)
    vector_ocupacion = numpy.zeros(numero_bandas)
    for i in range(numero_de_parejas):          #Ubico cada pareja en
"vector_ocupacion"
        ubicado = 0
        indice_pareja = 1
        altura_de_abajo = vector_mascara[i][1]                #Límite inferior de
las parejas ya formadas

        #Averiguo qué posición tiene cada pareja en el "vector_ocupacion"
        while ubicado == 0:
            factor = indice_pareja * separacion_teorica
            if altura_de_abajo <= factor:
                vector_ocupacion[indice_pareja - 1] = 1
                ubicado = 1
            indice_pareja = indice_pareja + 1

    print('Vector máscara:', vector_mascara)
    print('Vector ocupación:', vector_ocupacion)
    print('Número de parejas actuales:', numero_de_parejas)
    print('------------------------------------------------------------------------
------------------------------------------------------------------------
--------')
    print('')

    return vector_mascara, numero_de_parejas, vector_ocupacion
```

Una vez eliminadas las bandas necesarias, recalculamos de nuevo el **vector_ocupacion**.

La idea de eliminar la banda de debajo de las dos que estamos comparando se debe a que, las líneas detectadas por la transformada de Hough que no corresponden a límites superiores o inferiores de alguna banda, suelen ser líneas que se detectan en la parte superior de los envases de los productos. En la siguiente imagen marcaremos los posibles sitios donde se pueden detectar líneas horizontales que no corresponden con límites de las bandas para ejemplificar lo que acabamos de comentar.

Figura 6-19. Ejemplo de posibles líneas detectadas en zona de productos.

Es por ello que, si tenemos dos parejas próximas, eliminamos la que está abajo que probablemente se haya formado en la parte de arriba de un producto cuando termina su envase. A raíz de esto podemos inducir que, dependiendo de la geometría de los productos, el algoritmo será más o menos propenso a identificar líneas en zona de productos. Si por ejemplo estamos en un caso como la figura 6-13 que se trata de lechugas donde directamente no existen envases que puedan provocar la detección de líneas horizontales en esa zona, la detección de límites de bandas mediante Hough será más efectiva.

6.3.8 Fase de aprendizaje

El objetivo de este paso es, utilizar información adquirida en iteraciones previas, de esta manera, no empezamos siempre de cero con cada imagen. Dicha información se almacena en **vector_aprendizaje** que contiene la misma información que **vector_mascara**, es decir, las alturas de las bandas, pero en este caso, ya sabemos que se trata seguro de alturas donde hay bandas, no como cuando le dábamos valor en ***emparejamiento_lineas*** a **vector_mascara** que no teníamos seguridad de que fuesen parejas (ρ, θ) que contuviesen bandas.

La manera de comprobar que realmente contienen bandas se realiza en la etapa tres aplicando gradientes y métodos morfológicos. Si resulta que hay códigos de barras en la teórica banda, se guarda en **vector_aprendizaje** la altura de los límites de esa banda, en caso de que no se detecten códigos de barras no se almacena. Lo veremos más en profundidad en la tercera etapa.

La información de **vector_aprendizaje** se utiliza para compararla con las bandas que nos quedan tras pasar por ***eliminacion_bandas_productos*** contenidas en **vector_mascara**. Si alguna pareja de **vector_mascara** tiene valores similares a alguna de las parejas contenidas en **vector_aprendizaje**, (se han tomado 100 píxeles de margen entre el límite superior de alguna de las bandas de **vector_mascara** y el límite superior de algunas de las bandas de **vector_aprendizaje**, lo mismo para el límite inferior), se pasa esa pareja de **vector_mascara** a **vector_mascara_def**.

En caso de que alguna de las parejas de **vector_mascara** no esté próxima a ninguna de las contenidas en **vector_aprendizaje**, se añaden ambos límites de la pareja a **vector_desechadas**, que se unirán a las líneas ya desechadas que no pudimos emparejar en la función ***emparejamiento_lineas***.

Una vez hecho esto, vemos si podemos añadir de manera individual algunas de las líneas contenidas en **vector_desechadas** a **vector_mascara_def**, es decir, ya no completar una banda entera sino añadir solo el límite superior o el límite inferior. Se vuelve a usar un margen de 100 píxeles.

Código 6-10. Función para realizar la fase de aprendizaje (I).

```
#Función que realiza la fase de aprendizaje
def fase_aprendizaje(vector_mascara, vector_desechadas, vector_aprendizaje,
numero_bandas):
    print('-------------------------------------------------------------
Emparejamiento usando vector aprendizaje ----------------------------------------
--------------------------')
    numero_de_parejas = 0
    vector_mascara_def = []
    for i in range(numero_bandas):
        vector_mascara_def.extend([[0,0]])

    vector_desechar_parejas = numpy.zeros(len(vector_mascara))

    print('Vector aprendizaje:', vector_aprendizaje)
    print('')

    #Bucle para recorrer el vector aprendizaje
    for i in range(len(vector_aprendizaje)):
        print('Elemento vector aprendizaje:', vector_aprendizaje[i])
        #Bucle para recorrer el vector máscara. Así comparo cada pareja formada en
"vector_mascara" con cada elemento del "vector_aprendizaje"
        for j in range(len(vector_mascara)):
            print('Elemento vector máscara:', vector_mascara[j])
            #Si la pareja estudiada de "vector_mascara" está a la misma altura que
la pareja que se está comprobando de "vector_aprendizaje", se añade a la máscara
definitiva
            if abs(vector_mascara[j][0] - vector_aprendizaje[i][0]) < 100 and
abs(vector_mascara[j][1] - vector_aprendizaje[i][1]) < 100 and
vector_aprendizaje[i] != [0, 0]:
                vector_mascara_def[i] = [vector_mascara[j][0],
vector_mascara[j][1]]
                print('Añadida pareja a vector mascara definitivo:',
vector_mascara_def)
                numero_de_parejas = numero_de_parejas  + 1
            else:
                vector_desechar_parejas[j] = vector_desechar_parejas[j] + 1
        print('')

    #Hay que añadir las parejas que no pasan al vector máscara definitivo a las
líneas desechadas para ver si alguna línea suelta, ya no como pareja, se puede
añadir al vector máscara definitivo
    for i in range(len(vector_mascara)):
        if vector_desechar_parejas[i] == len(vector_aprendizaje):
            print('Añadida pareja', vector_mascara[i], 'a las líneas desechadas')
            vector_desechadas.append(vector_mascara[i][0])
            vector_desechadas.append(vector_mascara[i][1])        #Añado la pareja
formada a las líneas desechadas

    print('Vector máscara tras añadir parejas completas:', vector_mascara_def)
    print('Vector desechadas después de añadir parejas desechadas:',
vector_desechadas)
    print('')
```

Código 6-11. Función para realizar la fase de aprendizaje (II).

```
    #Bucle para recorrer el vector aprendizaje
        for i in range(len(vector_aprendizaje)):
            print('Elemento vector aprendizaje:', vector_aprendizaje[i])
            #Bucle para recorrer el vector de líneas desechadas y ver si
    podemos añadir alguna en el vector máscara definitivo
            for j in range(len(vector_desechadas)):
                print('Línea desechada:', vector_desechadas[j])
                #Añado un límite superior
                if abs(vector_aprendizaje[i][0] - vector_desechadas[j]) <
    100 and vector_aprendizaje[i][0] != 0:
                    vector_mascara_def[i][0] = vector_desechadas[j]
                    print('Añadido límite superior a vector mascara
    definitivo:', vector_mascara_def)
                #Añado un límite inferior
                elif abs(vector_aprendizaje[i][1] - vector_desechadas[j]) <
    100 and vector_aprendizaje[i][1] != 0:
                    vector_mascara_def[i][1] = vector_desechadas[j]
                    print('Añadido límite inferior a vector mascara
    definitivo:', vector_mascara_def)
            print('')

        print('Vector máscara definitivo:', vector_mascara_def)
        print('Número de parejas completas:', numero_de_parejas)

        return vector_mascara_def, numero_de_parejas
```

Luego, después de pasar por esta función, se le da a **vector_mascara** en el programa principal el valor de **vector_mascara_def**. Y como hemos visto, **vector_mascara_def** puede tener identificado de algunas bandas solo uno de los límites, luego el siguiente paso será completar las posibles bandas que están a medias.

Es necesario comentar que, con el robot real las bandas van a estar siempre exactamente a la misma altura, luego se podrían asignar directamente en **vector_mascara** las bandas que ya sabemos seguro que son bandas reales contenidas en **vector_aprendizaje**. De esta manera, en cada iteración solo habría que buscar las bandas de las franjas que aún no están en **vector_aprendizaje**, pero las demás ya las conoceríamos al comienzo de cada iteración. Si se realizase de esta manera, al cabo de unas pocas iteraciones cuando ya se hayan identificado todas las bandas, simplemente se impondrían esas alturas del **vector_aprendizaje** a **vector_mascara** al comienzo de cada iteración, ahorrándonos todos los pasos previamente comentados y los que todavía quedan por explicar.

6.3.9 Obtención del ancho de las parejas

Para poder completar las bandas que solo tienen un límite identificado, debemos saber el ancho que tienen las bandas completas ya detectadas usando la función ***ancho_bandas***. Una vez conocida esa información y sabiendo qué limite tenemos identificado de las bandas en las que solo conocemos un límite, bastaría con añadir ese ancho en la dirección adecuada, esto es lo que haremos en el siguiente paso.

Código 6-12. Función para obtener el ancho de las bandas completas ya detectadas.

```
#Obtengo el ancho (diferencia en la altura) de las bandas. Se calcula como
altura del límite inferior - altura del límite superior
def ancho_bandas(numero_de_parejas, vector_mascara):
    print('-----------------------------------------------------------------
---- Cálculo ancho bandas ------------------------------------------------
------------------')
    vector_anchos = []
    ancho = 0
    if numero_de_parejas != 0:
        for i in range(len(vector_mascara)):
            if vector_mascara[i][0] != 0 and vector_mascara[i][1] != 0:
#Para asegurarme que es una pareja cuyos dos valores ya han sido dados ->
es decir no podría ser [1279,0]
                vector_anchos.append(int(vector_mascara[i][1] -
vector_mascara[i][0]))

        print('Vector anchos:', vector_anchos)
        ancho = max(vector_anchos)
        print('Ancho máximo:', ancho)
        print('')
    else:    #En caso de que no haya parejas formadas, se informa con un
mensaje
        print("No hay parejas formadas para obtener el ancho máximo")
        print('')
        print('-----------------------------------------------------------
----------------------------------------------------------------------------
---------------------------')

    return vector_anchos, ancho
```

Como podemos ver, de todas las bandas completas, nos quedamos con aquella que tenga más ancho para asegurarnos que al completar las bandas que están a medias, los códigos de barras queden dentro de la banda definida. Como su nombre indica, este ancho máximo se almacena en la variable **ancho**. En caso de que al llamar a esta función **vector_mascara** no tenga ninguna banda completa, se le asignará un valor por defecto como ya veremos en la función principal.

6.3.10 Completar bandas a medias

Como ya hemos comentado, ahora lo que haremos es usar el ancho que acabamos de calcular para completar las bandas que están identificadas a medias. La función que realiza esta tarea es ***completar_bandas_aprendizaje***. Recorre **vector_mascara**, si el límite que tenemos es el superior, rellenamos la banda hacia abajo, por el contrario, si el límite que tenemos es el inferior, rellenamos la banda hacia arriba. Para ello usamos **ancho** calculado previamente en la función ***ancho_bandas***.

Código 6-13. Función para completar las bandas a medias.

```
#Función que completa las bandas del vector definitivo que se han quedado a medias,
es decir, solo con el límite superior o solo con el límite inferior
def completar_bandas_aprendizaje(vector_mascara, ancho, height, numero_bandas,
numero_de_parejas):
    print('-------------------------------------------------------------------
Completar bandas a medias ------------------------------------------------------
-----------')
    print("Vector máscara:", vector_mascara)
    for i in range(len(vector_mascara)):
        #Tenemos el límite superior, pero no el inferior así que lo añadimos
sabiendo el ancho de las bandas
        if vector_mascara[i][0] != 0 and vector_mascara[i][1] == 0:
            vector_mascara[i][1] = vector_mascara[i][0] + ancho
            print("Completado límite inferior:", [vector_mascara[i][0],
vector_mascara[i][1]])
            print('Añadido límite inferior:', vector_mascara[i][1])
        #Tenemos el límite inferior, pero no el superior así que lo añadimos
sabiendo el ancho de las bandas
        elif vector_mascara[i][0] == 0 and vector_mascara[i][1] != 0:
            vector_mascara[i][0] = vector_mascara[i][1] - ancho
            print("Completado límite superior:", [vector_mascara[i][0],
vector_mascara[i][1]])

    print("Vector máscara tras completar bandas:", vector_mascara)

    separacion_teorica = int(height / numero_bandas)
    vector_ocupacion = numpy.zeros(numero_bandas)    #Para saber qué posiciones
están ocupadas (0 libre - 1 ocupado)

    for i in range(len(vector_ocupacion)):        #Ubico cada pareja en
"vector_ocupacion"
        ubicado = 0
        indice_pareja = 1
        altura_de_abajo = vector_mascara[i][1]               #Límite inferior de
las parejas ya formadas

        if vector_mascara[i][0] != 0 and vector_mascara[i][1] != 0:        #Por
asegurarnos, se comprueba que es una pareja formada al completo
            #Averiguo qué posición tiene cada pareja en el "vector_ocupacion"
            while ubicado == 0:
                factor = indice_pareja * separacion_teorica
                if altura_de_abajo <= factor:
                    vector_ocupacion[indice_pareja - 1] = 1
                    ubicado = 1
                indice_pareja = indice_pareja + 1

    print('Vector ocupación:', vector_ocupacion)

    numero_de_parejas = 0
    #Cuento el número de parejas formadas
    for i in range(len(vector_mascara)):
        if vector_mascara[i] != [0, 0]:
            numero_de_parejas = numero_de_parejas + 1
    print('Número de parejas formadas tras aprendizaje:', numero_de_parejas)
    print('')

    return vector_mascara, vector_ocupacion, numero_de_parejas
```

Una vez completadas las bandas que estaban a medias junto con las que ya estaban completas, volvemos a calcular el **vector_ocupacion**. En la siguiente función ya será cuando lo utilicemos, pero era necesario calcularlo en las otras funciones y no solo esta, porque en la primera iteración no realizamos ni la fase de aprendizaje ni esta función para rellenar bandas incompletas.

6.3.11 Relleno con bandas artificiales

Estamos ante el último paso de la etapa uno. Este paso solo se realiza en caso de que el número de bandas detectadas no sea igual al número de bandas a detectar introducido por la línea de comandos al ejecutar el programa. La función encargada de esta última tarea es ***bandas_artificiales***.

En primer lugar, calculamos la **separacion** que existe entre dos bandas consecutivas. Para ello haremos uso de **vector_ocupacion**, para saber la posición que ocupan las bandas que estamos comparando. Sabiendo la posición que ocupan las bandas ya detectadas en **vector_ocupacion** y los valores de dichas bandas obtendremos dicha separación.

Lo que se hace es comprobar cada banda detectada comenzando desde la que esté más arriba con las bandas posteriores. En el momento que seamos capaces de obtener la separación una vez, ya nos salimos del bucle. Esto es así porque hemos vuelto a aplicar la suposición de que las bandas son equidistantes. El cálculo se realiza tomando como referencia la altura media de las bandas, es decir, se hace la media entre la altura del límite superior e inferior y eso es lo que se usará para calcular la diferencia de altura entre dos bandas ya detectadas. Veamos un ejemplo para comprenderlo mejor.

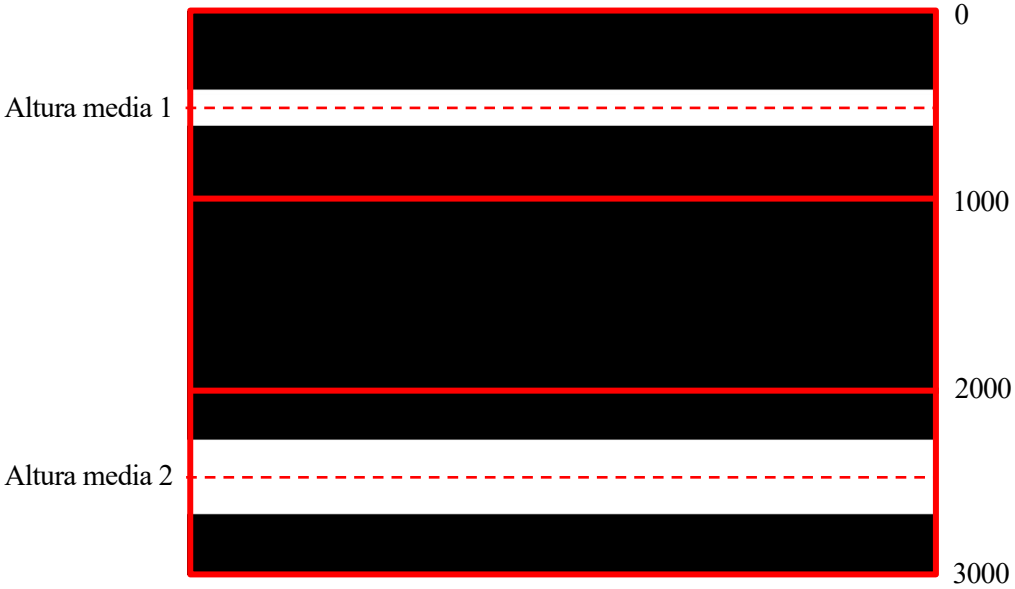

Figura 6-20. Ejemplo del cálculo de la separación entre bandas ya detectadas.

Como podemos ver, el **vector_ocupacion** tendrá valor [1, 0, 1]. Al saber esto cuando comparemos las dos bandas ya detectadas, sabremos que falta una banda por detectar entre las dos que ya tenemos, luego la separación que nosotros estemos calculando será el doble de la separación que estamos buscando. Para este ejemplo el cálculo sería el siguiente.

$$\text{separacion} = \frac{\text{Altura media 2} - \text{Altura media 1}}{2} \tag{6-5}$$

En caso de que no se pueda calcular la separación porque solo tengamos detectada una banda, se le asignará por defecto el valor de la variable **separacion_teorica**. Esta variable ya se ha usado en funciones previas para calcular el **vector_ocupacion** y no es más que el alto de la imagen, dividido entre el número de bandas a detectar. En la imagen vemos dos líneas continuas horizontales que dividen la imagen en tres partes iguales, esta es la **separacion_teorica** con valor 1000 píxeles. Vemos ahora el cálculo para este ejemplo.

$$\text{separacion\_teorica} = \frac{\text{alto de la imagen}}{\text{número de bandas a detectar}} = \frac{3000}{3} = 1000 \tag{6-6}$$

Si nos damos cuenta, es el mismo principio que se aplica para dividir la imagen en franjas iguales al comienzo de esta primera etapa en la función ***Hough_franjas***.

Código 6-14. Función para crear bandas artificiales (I).

```
#Función que crea bandas artificiales cuando no hemos sido capaces de
formar todas las parejas a partir de las líneas detectadas por Hough. Para
ello utiliza la suposición de que las bandas son equidistantes
def bandas_artificiales(height, numero_bandas, vector_mascara,
vector_ocupacion, numero_de_parejas):
    print('-----------------------------------------------------------------
---- Relleno bandas artificiales ---------------------------------------
-------------------------')
    #Miro la altura de las bandas ya detectadas y sabiendo que son
equidistantes creo artificialmente las que queden
    #hasta llegar a "numero_de_parejas == numero_bandas -> hasta completar
el vector de ocupación"
    i = 0
    separacion = 0

    #En cuanto sea capaz de calcular la separación entre bandas, dejo de
buscar ya que al suponer que son equisdistantes, el resto de resultados nos
dará una valor de separación similar
    while(i < len(vector_ocupacion) - 1):        #Ejemplo: 4 bandas -> i va
[0, 2] < 4 - 1 = 3
        #Miro si la banda que estoy mirando y la siguiente están ocupadas.
En ese casos tenemos dos bandas consecutivas ocupadas
        if vector_ocupacion[i] == 1 and vector_ocupacion[i+1] == 1:
            altura1_media = int((vector_mascara[i][0] +
vector_mascara[i][1]) / 2)
            altura2_media = int((vector_mascara[i+1][0] +
vector_mascara[i+1][1]) / 2)
            separacion = altura2_media - altura1_media
            break
        #Miro si la banda que estoy mirando y la que está dos posiciones
más por delante están ocupadas. En ese casos obtendríamos 2 * (separación
entre bandas) y se dividiría entre dos para obtener la separación entre
bandas consecutivas
        elif i < len(vector_ocupacion) - 2 and vector_ocupacion[i] == 1 and
vector_ocupacion[i+2] == 1:        #Banda ocupada, hueco, banda ocupada
            altura1_media = int((vector_mascara[i][0] +
vector_mascara[i][1]) / 2)
            altura2_media = int((vector_mascara[i+2][0] +
vector_mascara[i+2][1]) / 2)
            separacion = int((altura2_media - altura1_media) / 2)
            break
        #Miro si la banda que estoy mirando y la que está tres posiciones
más por delante están ocupadas. En ese casos obtendríamos 3 * (separación
entre bandas) y se dividiría entre tres para obtener la separación entre
bandas consecutivas
        elif i < len(vector_ocupacion) - 3 and vector_ocupacion[i] == 1 and
vector_ocupacion[i+3] == 1:        #Banda ocupada, hueco, banda ocupada
            altura1_media = int((vector_mascara[i][0] +
vector_mascara[i][1]) / 2)
            altura2_media = int((vector_mascara[i+3][0] +
vector_mascara[i+3][1]) / 2)
            separacion = int((altura2_media - altura1_media) / 3)
            break
        i = i + 1
```

Código 6-15. Función para crear bandas artificiales (II).

```
    #Separación teórica que se usa en caso de no haber podido calcular la
separación entre bandas
    separacion_teorica = int(height / numero_bandas)
    if separacion == 0:
        print('No hay dos bandas consecutivas para obtener la separación entre
ellas -> se usa la separación teórica')
        separacion = separacion_teorica

    print('Vector máscara:', vector_mascara)
    print('Separación teórica:', separacion_teorica)
    print('Vector ocupación:', vector_ocupacion)
    print('Separación entre bandas:', separacion)
    print('')

    #Conociendo ya la separación, relleno las bandas que están vacías usando el
ancho de las bandas calculado en la función "ancho_bandas()" y la separación entre
ellas que lo acabamos de calcular
    for i in range(len(vector_ocupacion)):    #Miro si la banda de arriba o de
abajo de la vacía está ocupada para usarla como referencia
        print('Índice vector ocupación:', i)
        if vector_ocupacion[i] == 0:
            if i > 0 and vector_ocupacion[i-1] == 1:
                print('i-1')
                print('Pareja usada:', [vector_mascara[i-1][0], vector_mascara[i-
1][1]])
                vector_mascara.pop(i)
                vector_mascara.insert(i, [vector_mascara[i-1][0]+separacion-50,
vector_mascara[i-1][1]+separacion+50])
                print('Añadida banda artificial:', [vector_mascara[i][0],
vector_mascara[i][1]])
                numero_de_parejas = numero_de_parejas + 1
                print('Vector máscara tras añadir banda artificial:',
vector_mascara)
                vector_ocupacion[i] = 1
                print('Vector ocupación:', vector_ocupacion)
            elif i < len(vector_ocupacion) - 1 and vector_ocupacion[i+1] == 1:
                print('i+1')
                print('Pareja usada:', [vector_mascara[i+1][0],
vector_mascara[i+1][1]])
                vector_mascara.pop(i)
                vector_mascara.insert(i, [vector_mascara[i][0]-separacion-50,
vector_mascara[i][1]-separacion+50])
                print('Añadida banda artificial:', [vector_mascara[i][0],
vector_mascara[i][1]])
                numero_de_parejas = numero_de_parejas + 1
                print('Vector máscara tras añadir banda artificial:',
vector_mascara)
                vector_ocupacion[i] = 1
                print('Vector ocupación:', vector_ocupacion)
            else:
                print('No se puede tomar ninguna banda de referencia')
        else:
            pass
        print('')
```

Código 6-16. Función para crear bandas artificiales (III).

```
    print('Vector máscara tras relleno artificial:', vector_mascara)
    print('Número de parejas tras bandas artificiales:', numero_de_parejas)
    print('------------------------------------------------------------------
------------------------------------------------------------------------
------------------------')
    print('')

    return vector_mascara, numero_de_parejas, vector_ocupacion, separacion
```

En el segundo trozo de código es donde creamos las bandas artificiales necesarias. Volviendo a hacer uso del **vector_ocupacion**, buscamos un elemento del vector que tenga valor 0 y que por tanto es una banda que hay que crear, luego buscamos una banda ya detectada, esto significa que el **vector_ocupacion** tenga valor 1, si es así, creamos la banda usando la **separacion** más 100 píxeles de margen, 50 hacia cada lado. Usemos el mismo ejemplo de la figura 6-20.

Figura 6-21. Ejemplo de creación de bandas artificiales.

En este caso no hace falta recalcular el **vector_ocupacion**, se puede rellenar directamente porque cuando creamos una banda artificial, esa posición del vector estaba a 0, luego simplemente al crear la banda la cambiamos a 1.

6.3.12 Función principal

Pasamos a comentar la función ***funcion_princpial***. Simplemente leemos en un bucle las imágenes a procesar usando la función **imread** de OpenCV, luego se va llamando a cada una de las funciones comentadas en este capítulo, se recogen los resultados devueltos por cada función, y se le pasan los argumentos necesarios a la función siguiente, así hasta completar todos los pasos comentados.

La función ***pintar_lineas*** sirve como su nombre indica, para pintar las líneas detectadas. Recibe como argumentos el vector de alturas/ρ y el vector de ángulos/θ entre otros e imprime las líneas sobre la imagen que estamos procesando. No profundizaremos más en esta función ya que carece de interés para el funcionamiento del algoritmo. El esqueleto de esta función se ha extraído de [6].

Se han eliminado los plots de la función principal para reducir el tamaño del código, pero en el archivo .py están y se pueden usar si se quiere.

Código 6-17. Función principal de la etapa 1 (I).

```
#Función principal de la etapa 1
def funcion_principal():
    #Se lee el número de bandas y se configuran las variables necesarias
    numero_bandas = int(sys.argv[1])
    separacion, separacion3, separacion2, distancia_eliminar =
configuracion_numero_bandas(numero_bandas)
    numero_lineas_a_detectar = 10

    vector_aprendizaje = []
    for i in range(numero_bandas):
        vector_aprendizaje.extend([[0,0]])

    indice_lectura_codigos = 0
    primera_iter = 1

    #Directorio donde se encuentran las imágenes de las estanterías al
completo
    mypath='E:\\Documents\\Juan de Dios\\TFG\\Fotos Mercadona\\Misma
altura\\Secuencia1'
    onlyfiles = [ f for f in listdir(mypath) if isfile(join(mypath,f)) ]
    onlyfiles = natsorted(onlyfiles)
    images = numpy.empty(len(onlyfiles), dtype=object)
    for n in range(0, len(onlyfiles)):
        images[n] = cv2.imread( join(mypath,onlyfiles[n]) )
        hsv = cv2.cvtColor(images[n], cv2.COLOR_BGR2HSV)
        images[n] = cv2.cvtColor(images[n], cv2.COLOR_BGR2RGB)

        height, width, channels = images[n].shape

        #Busco las bandas horizontales por transformada de Hough en cada
una de las franjas
        edges, vector_alturas_unidas, vector_angulos_unidos =
Hough_franjas(numero_bandas, height, width, images[n],
numero_lineas_a_detectar)

        #Si hemos detectado alguna línea
        if len(vector_alturas_unidas) != 0:
            #Para pintar todas las líneas detectadas por Hough
            img_copy1 = pintar_lineas(images[n], height, width, None,
vector_alturas_unidas, vector_angulos_unidos)

            #Miro la altura de las líneas y desecho las que representan la
misma línea horizontal para quedarme solo con una
            vector_alturas, vector_angulos =
seleccion_lineas_definitivas(vector_alturas_unidas, vector_angulos_unidos,
separacion)

            #Pinto las líneas definitivas
            img_copy2 = pintar_lineas(images[n], height, width, None,
vector_alturas, vector_angulos)
        else:
            print('Ninguna línea detectada')

        #Obtenemos el número de líneas a emparejar y ordenamos los vectores
de menor rho a mayor y los ángulos acorde a cómo se han ordenado las
distancias (rho)
        tam_vector, alturas_ordenadas, angulos_ordenados =
ordena_alturas(vector_alturas, vector_angulos)
```

Código 6-18. Función principal de la etapa 1 (II).

```
        #Emparejamos las líneas detectadas
        vector_mascara, alturas, vector_ocupacion, vector_desechadas,
vector_indices, numero_de_parejas, numero_lineas_desechadas =
emparejamiento_lineas(tam_vector, alturas_ordenadas, height, numero_bandas,
separacion3, separacion2)

        #Miramos si alguna de las parejas formadas está muy próxima a otra,
en ese caso, eliminamos la banda de abajo
        vector_mascara, numero_de_parejas, vector_ocupacion =
eliminacion_bandas_productos(height, numero_bandas, vector_mascara,
vector_ocupacion, numero_de_parejas, distancia_eliminar)

        #A partir de la segunda imagen, usamos el "vector_aprendizaje"
        if primera_iter == 0:
            vector_mascara, numero_de_parejas =
fase_aprendizaje(vector_mascara, vector_desechadas, vector_aprendizaje,
numero_bandas)

        #Obtenemos el ancho de las bandas completas ya detectadas
        if primera_iter == 0:
            vector_anchos, ancho = ancho_bandas(numero_de_parejas,
vector_mascara)
            #En caso de no existir ninguna banda completa, suponemos un
ancho de banda por defecto
            if ancho == 0:
                ancho = 350

        #Arreglo praa resolver un problema de dimensiones
        if primera_iter == 1 and len(vector_mascara) !=
len(vector_ocupacion):
            for i in range(len(vector_ocupacion)):
                if vector_ocupacion[i] == 0:
                    vector_mascara.insert(i, [0, 0])
                    print('Añadido [0, 0] en la posición', i, 'del vector
máscara')
        print('Vector máscara tras relleno con ceros:', vector_mascara)
        print('')

        #Completamos las bandas que tras la fase de aprendizaje están solo
con uno de los bordes
        if primera_iter == 0:
            vector_mascara, vector_ocupacion, numero_de_parejas =
completar_bandas_aprendizaje(vector_mascara, ancho, height, numero_bandas,
numero_de_parejas)

        #Creamos máscara para filtrar por alturas usando las parejas
formadas
        mascara = creacion_mascara(height, width, vector_mascara, flag = 1)

        #Resultado obtenido hasta ahora
        imagen_aprendizaje = cv2.bitwise_and(images[n],images[n],
mask=mascara)
```

Código 6-19. Función principal de la etapa 1 (III).

```
        #En caso de que falten bandas por detectar, las creo
artificialmente aplicando la suposición de que son equidistantes
        if numero_de_parejas < numero_bandas:
            vector_mascara, numero_de_parejas, vector_ocupacion,
separacion_bandas = bandas_artificiales(height, numero_bandas,
vector_mascara, vector_ocupacion, numero_de_parejas)

        primera_iter = 0

        #Copia del vector actual para comparar cuando eliminemos alguna
pareja
        vector_mascara_copia = vector_mascara

        #Creamos la máscara con las bandas artificiales
        mascara2 = creacion_mascara(height, width, vector_mascara, flag =
0)

        #Resultado final
        resultado = cv2.bitwise_and(images[n],images[n], mask=mascara2)
```

El resultado final con las bandas identificadas lo almacenamos en **resultado**.

La modificación de **vector_aprendizaje** se debe realizar en la etapa 3 que es cuando comprobamos si las bandas que hemos identificado en esta etapa son realmente bandas correctas que contienen códigos de barras o no. No obstante, para poder utilizar el **vector_aprendizaje** en esta etapa y modificarlo, se introducirá manualmente un vector por imagen donde se indique con valor **True** si es una banda correcta o con **False** si es una banda incorrecta.

De no haberlo hecho así, tendríamos que haber ejecutado el programa, ir viendo dónde ubica el programa las bandas y volver al comercio para tomar fotos como las que se usarán en la etapa 3 pero tratando de que solo salga la zona donde el programa ha ubicado las bandas, lo que sería bastante tedioso de realizar. Vemos ahora con un ejemplo cómo funciona lo comentado.

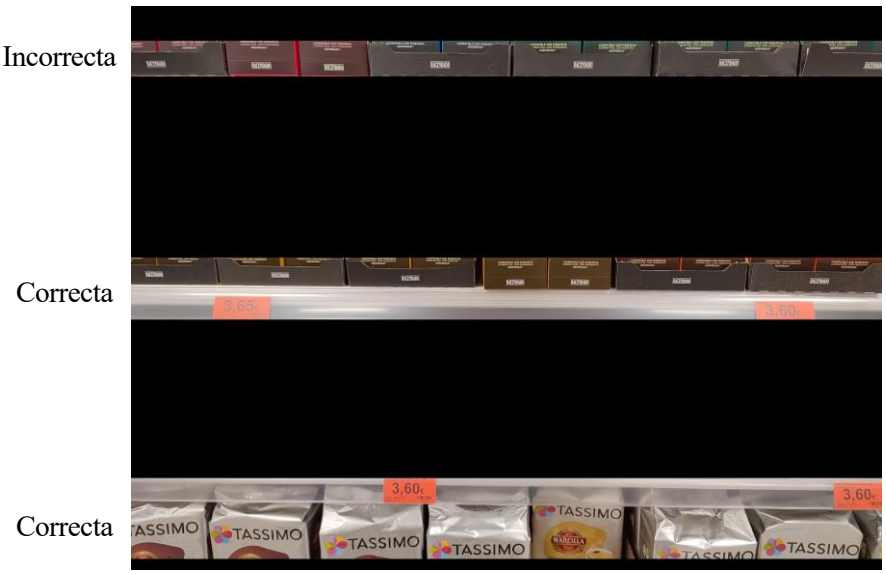

Figura 6-22. Ejemplo de la indicación manual de la validez de las bandas detectadas en la etapa 1.

En este ejemplo de tres bandas, tenemos que la primera banda es incorrecta ya que está en zona de productos, pero la segunda y la tercera banda contienen las etiquetas con los códigos de barras (no importa que se corte un poco algún código de barras porque el zoom que hacemos en la etapa 2, no es exactamente solo de la banda, sino que se deja un poco de margen por encima y por debajo de la banda como se verá).

Con todo esto dicho, el vector que indica la validez de las bandas identificadas en esta imagen quedaría como sigue:

vector_imagen = [False, True, True]

Cuando analicemos los resultados, usaremos secuencias de tres imágenes, luego tendremos tres vectores de este tipo agrupados en la variable **vector_imagenes**.

Para realizar la tarea de tomar el **vector_imagenes** adecuado, se usa la función ***modificacion_manual***, que conociendo de qué comercio se trata, qué secuencia de imágenes estamos usando y el número de bandas a identificar, selecciona el **vector_imagenes** correspondiente. Es aquí cuando cobra importancia el número de comercio y el número de secuencia que mencionamos en la subsección 6.3.1. El código de esta función carece de interés y se ha decidido no poner, aunque si se desea se puede consultar en el código de la etapa uno.

7 ETAPA 2: ZOOM HACIA CADA UNA DE LAS BANDAS IDENTIFICADAS

Tal y como se indicó en la sección 5.1, en esta etapa pasamos de enfocar toda la estantería, a realizar un zoom hacia una de las bandas que hemos identificado. En el TFG de referencia [1] se comenta que, el paquete de ROS que se usa para controlar la cámara del robot cuenta con un nodo al que se le pueden pasar las coordenadas de la imagen que se quiere encuadrar y el zoom a aplicar. Así se logra que el píxel central de la imagen sea el punto cuyas coordenadas son las que le hemos pasado como argumento al nodo. La componente "y" será la altura media de la banda, es decir, la media entre la altura del límite superior y la altura del límite inferior de la banda.

Una vez tenemos solo una banda en la imagen, realizamos un barrido en busca de potenciales códigos de barras.

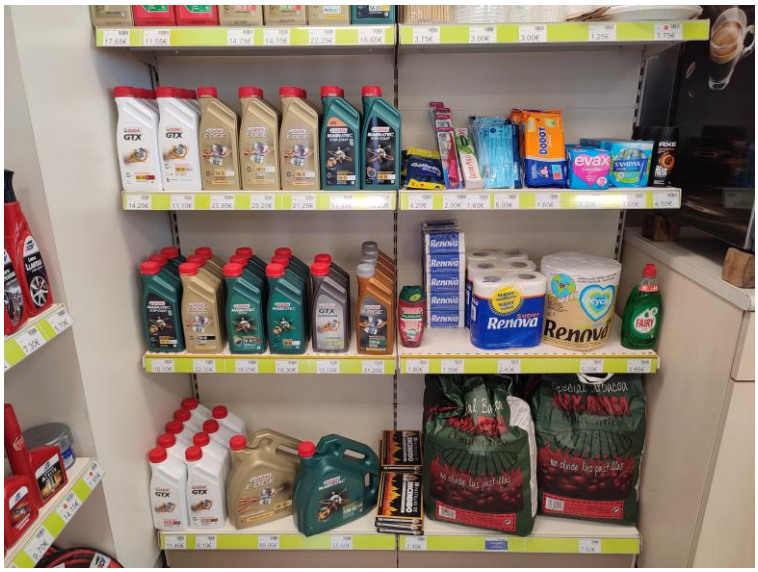

Figura 7-1. Ejemplo de imagen de la etapa 1.

De una imagen por ejemplo como la de la figura anterior, pasamos a una imagen del siguiente tipo.

Figura 7-2. Ejemplo de imagen obtenida tras realizar la etapa 2.

Aquí podemos ver lo que se comentó al final de la etapa 1, no se hace un zoom para solo dejar la banda en la imagen, sino que se deja también un poco de zona de productos tanto por arriba como por debajo de la banda.

En una imagen de este tipo ya sí tenemos los códigos de barras a un tamaño lo suficientemente grande como para identificarlos aplicando gradientes.

Es necesario volver a recordar que, usando el robot no se obtendrían imágenes como la figura 7-2 con la cámara enfrentada a la misma altura que la banda a la que estamos haciendo zoom, pero lo hacemos así para facilitar la tarea.

Por otra parte, recordemos que la manera de emular el zoom óptico de la cámara del robot dado que la cámara del móvil tiene zoom digital es, acercando la cámara del móvil hacia la zona que se quiere ampliar.

8 ETAPA 3: IDENTIFICACIÓN DE LOS POTENCIALES CÓDIGOS DE BARRAS

8.1 Introducción

Es el momento de, partiendo de una imagen como la figura 7-2, identificar los potenciales códigos de barras presentes en la imagen aplicando gradientes. Para ello usaremos la función ***calcula_codigos*** que a su vez llama a funciones ya comentadas en la etapa 1. Aunque los códigos de barras más comunes de encontrar son los EAN-13, probaremos también a identificar otros códigos de barras propios de algunos establecimientos. Partiendo de una imagen como la figura 7-2, se pretende idealmente obtener lo siguiente:

Figura 8-1. Ejemplo de lo que se busca en la etapa 3.

8.2 Diagrama de flujo

Mostramos ahora un diagrama de flujo de esta etapa.

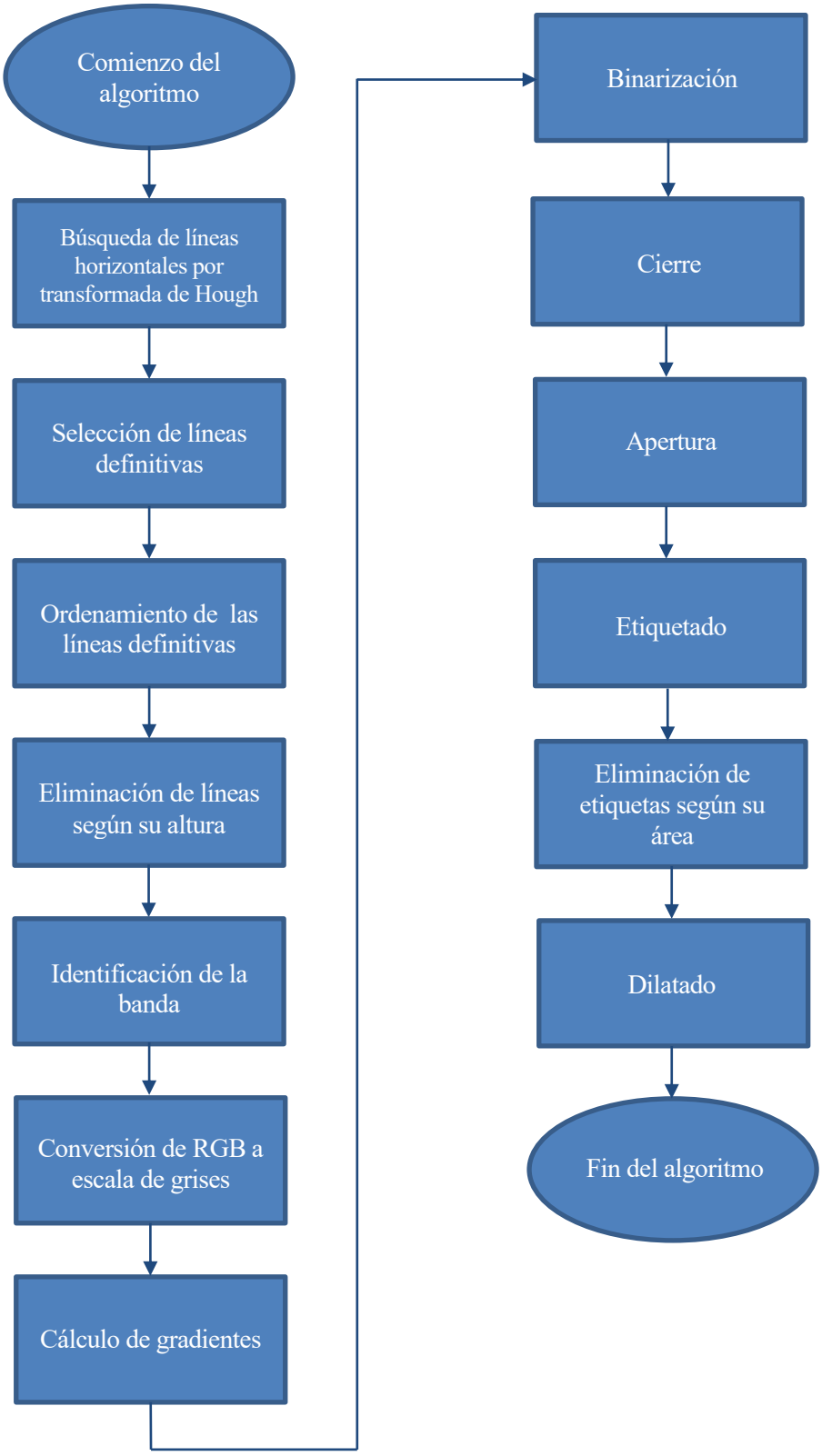

Figura 8-2. Diagrama de flujo de la etapa 3.

8.3 Identificación de líneas horizontales, agrupación y aplicación de la máscara binaria

Lo primero que haremos será identificar la única banda presente en la imagen para así reducir la zona de búsqueda de los códigos de barras. Esto con las funciones ya explicadas en la etapa 1 es una tarea fácil de realizar. Primero buscamos líneas horizontales usando ***Hough***, aquí no es necesario dividir la imagen en franjas usando ***Hough_franjas*** porque solo tenemos una banda que, además, está a una altura entre 1/3 y 2/3 de la altura de la imagen aproximadamente, ya que en la etapa 2 se encuadra la altura media de la banda.

Una vez identificadas las líneas, agrupamos las que representan la misma horizontal, ordenamos las que quedan como definitivas y las filtramos para quedarnos con las dos que se encuentran en la franja central de la imagen. Para ello, establecemos un límite superior de 850 y un límite inferior de 2150, toda línea que no tenga un ρ entre esos valores la desecharemos.

Figura 8-3. Identificación de líneas horizontales en imagen que emula el zoom a una banda.

Y las líneas definitivas resultan ser.

Figura 8-4. Líneas definitivas en la imagen que emula el zoom a una banda.

Luego ordenamos las líneas restantes y las filtramos para quedarnos con las que están en la franja central como se ha comentado. La imagen una vez aplicada la máscara con los límites de la banda y que será de la que partamos para calcular gradientes es la siguiente.

Figura 8-5. Banda aislada para el cálculo de gradientes en la etapa 3.

8.4 Aplicación de gradientes y binarización

En los pasos sucesivos nos hemos basado en [16] y [17]. Partiendo de esta imagen, la transformamos a escala de grises, calculamos gradientes horizontales y verticales usando una máscara de Sobel para identificar potenciales códigos de barras. Se obtiene lo siguiente aplicando valor absoluto a la diferencia entre el gradiente horizontal y el gradiente vertical, de esta manera destacamos zonas con gradientes horizontales altos y gradientes verticales bajos.

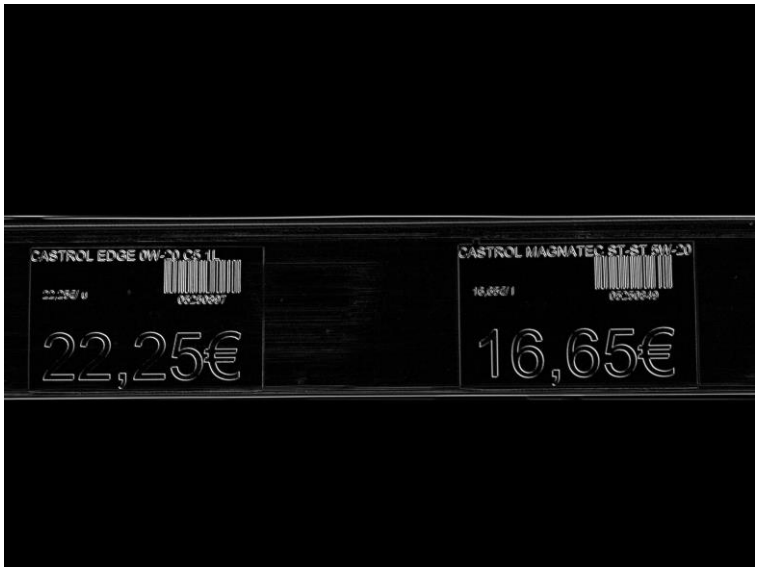

Figura 8-6. Resultado al aplicar gradientes a la imagen con la banda aislada.

Luego usamos la función de OpenCV **threshold** para binarizar la imagen. Como resultado obtendremos una imagen con solo dos valores, 0 o 255. Se establece como umbral un valor de 100. Todo píxel con valor en la figura anterior igual o superior a 100, pasará a tener valor 255 y, los píxeles que estén por debajo pasarán a tener valor 0.

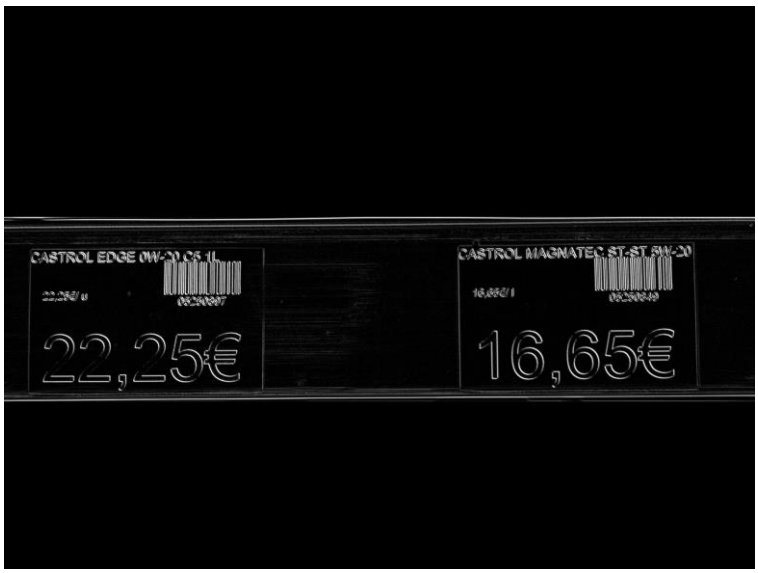

Figura 8-7. Binarización de la imagen de gradientes.

8.5 Transformaciones morfológicas y algoritmo de etiquetado

Antes de pasar a los siguientes pasos vamos a explicar lo que es un dilatado, una erosión, un cierre y una apertura [13]. Se tratan de transformaciones morfológicas y suelen aplicarse a imágenes binarias, que es el tipo de imagen que tenemos en la figura 8-7. El dilatado consiste en ensanchar las zonas que están a valor 255, mientras que la erosión es lo contrario.

Figura 8-8. a) Imagen original. b) Dilatado. c) Erosión [6].

Para estas transformaciones se necesita una plantilla o kernel como se mencionó en la subsección 4.6.5. Se trata de una matriz binaria, pero en este caso con valor 0 o 1. Veamos un ejemplo de un dilatado con una plantilla de tamaño 3x3.

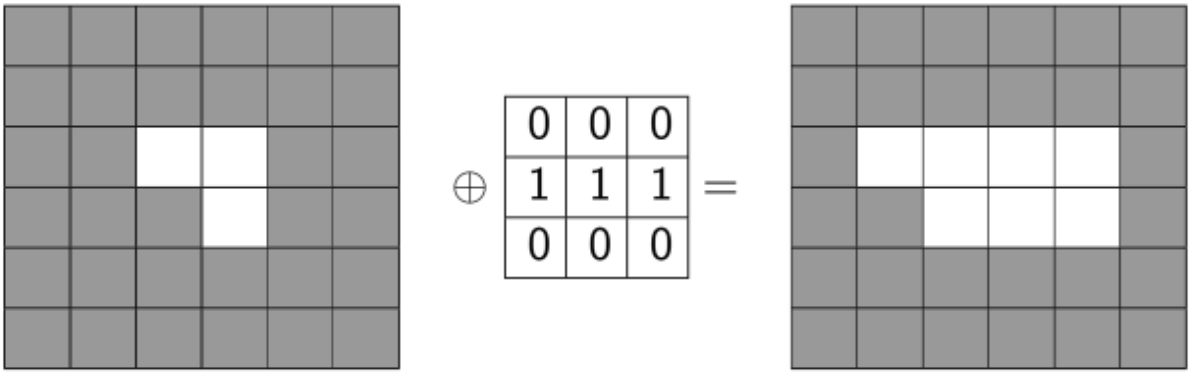

Figura 8-9. Ejemplo sencillo de un dilatado [13].

En este ejemplo, según la plantilla, habrá que comprobar si el píxel que está a la izquierda o a la derecha del que estamos comprobando está a 255, en ese caso el píxel pasa también a valor 255 si ya no lo estaba previamente. De lo contrario el píxel permanece a 0. Básicamente, si existe algún píxel con valor 255 que está en alguna posición donde la plantilla tiene un 1, obviando el píxel central, el píxel central pasa a valer 255 también.

En cuanto al cierre y la apertura, nos son más que la combinación de dilatado y erosión, pero en órdenes invertidos.

$$\text{Cierre} = \text{Dilatado} + \text{Erosión} \tag{8-1}$$

$$\text{Apertura} = \text{Erosión} + \text{Dilatado} \tag{8-2}$$

El cierre se usa para rellenar pequeños huecos negros en las zonas que están a valor 255 (píxel blanco). Por el contrario, la apertura sirve para eliminar pequeños puntos blancos que se encuentren sobre un fondo negro.

Figura 8-10. a) Cierre. b) Apertura [6].

Las plantillas que nosotros utilicemos tendrán forma rectangular, ya que es la forma que más se parece a los códigos de barras.

Con todo esto explicado podemos seguir con el algoritmo. A la imagen binaria como la de la figura 8-7, le realizamos un cierre para rellenar las zonas de los códigos de barras y así obtener rectángulos.

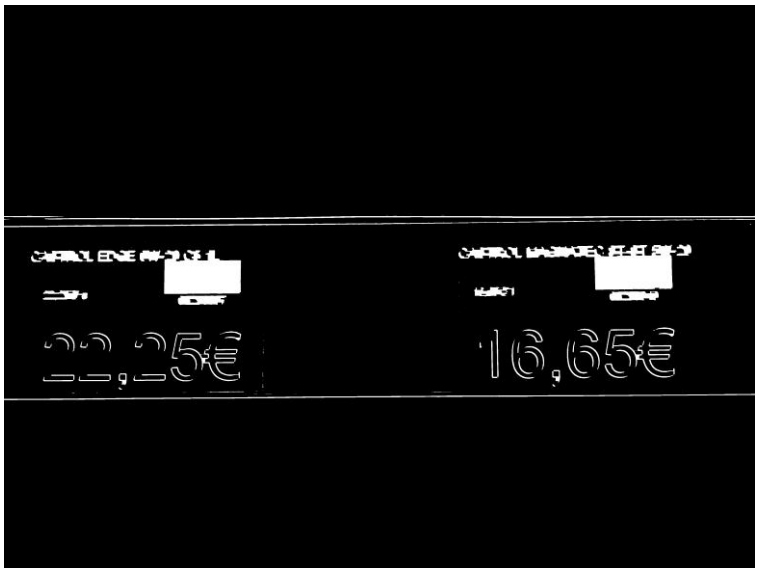

Figura 8-11. Cierre de la imagen binaria.

Luego aplicamos una apertura para eliminar la mayoría de los elementos que no son los rectángulos de los códigos, aunque pueden quedar algunos puntos sueltos.

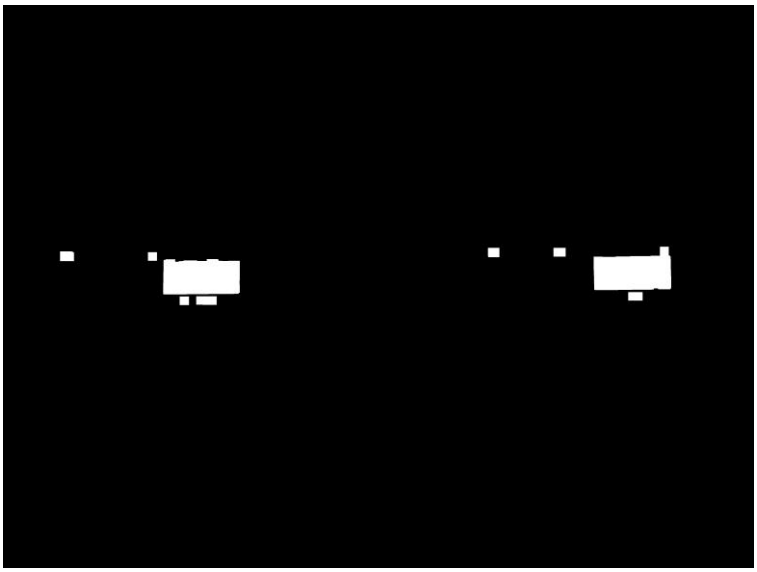

Figura 8-12. Apertura para eliminar la mayoría de las zonas no deseadas de la máscara binaria.

Ahora aplicando lo que se propone en [17], realizamos un algoritmo de etiquetado usando la función **connectedComponentsWithStats** de OpenCV que devuelve el área en píxeles de cada etiqueta entre otras cosas. Usando esta información, eliminamos aquellas etiquetas cuya área sea inferior a cierto umbral, 25000 píxeles en nuestro caso. De esta manera, logramos eliminar los pequeños puntos blancos que no corresponden con los rectángulos de los códigos de barras.

El etiquetado como su nombre indica, consiste en etiquetar los diferentes elementos de una imagen binaria. Cada región blanca que vemos en la imagen será una etiqueta distinta con valor entero de 1 hacia delante, la etiqueta 0 se reserva para el fondo negro. En nuestro caso, sin contar la etiqueta 0 del fondo, tendríamos nueve etiquetas porque tenemos nueve regiones blancas.

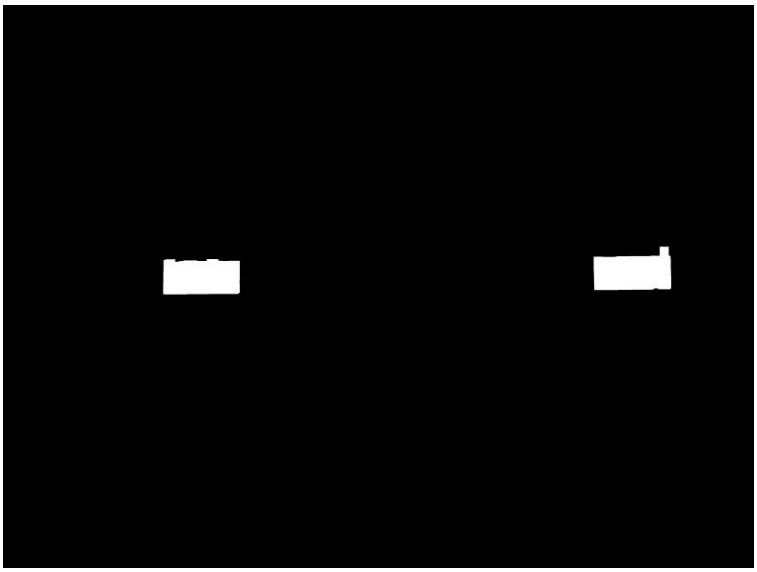

Figura 8-13. Filtro para eliminar etiquetas con área menor a un número de píxeles.

Como se puede apreciar, estableciendo un umbral para el área, logramos eliminar las siete etiquetas que no corresponden con los rectángulos de los códigos de barras. Esta máscara ya podríamos aplicarla a la imagen para obtener los códigos de barras, pero antes realizaremos un dilatado para asegurarnos que los códigos de barras quedan completamente circunscritos en los rectángulos.

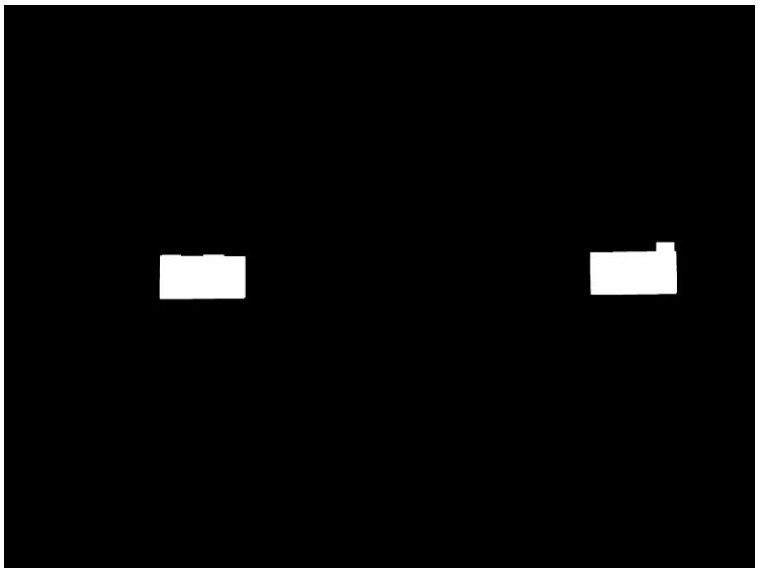

Figura 8-14. Dilatado para circunscribir los códigos completamente.

Esta sí es nuestra máscara definitiva, ya podemos aplicarla a la imagen original aplicando la operación AND. Aquellos píxeles donde la máscara esté a 255 mantendrán su valor de la imagen original, y el resto se pondrán a valor (0, 0, 0) que es el color negro.

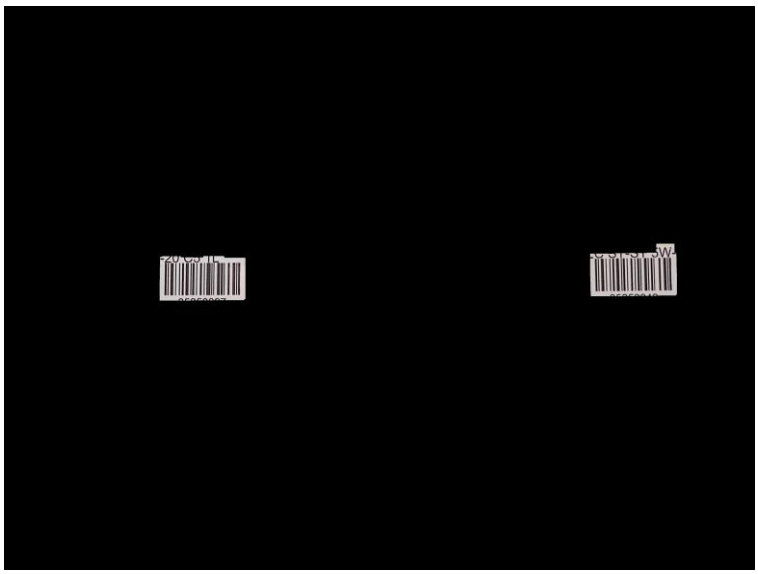

Figura 8-15. Resultado tras aplicar la máscara del dilatado a la imagen original.

Figura 8-16. Códigos de barras detectados.

Con el umbral de píxeles establecido nos aseguramos no detectar falsos positivos, tan solo aquellas regiones lo suficientemente grandes como para estar seguros de que son códigos de barras. Igualmente, si por ejemplo tenemos el reflejo de la luz sobre la tira plástica que protege y sostiene los papeles de las etiquetas, al calcular gradientes, binarizar, realizar el cierre y la apertura, el área de ese código será muy pequeña y se descartará al aplicar el filtro del área. Lo mismo ocurriría con un código que no saliese completo en la imagen. De esta manera, cuando el robot avance y el reflejo ya no se encuentre sobre ese código o el código ya sí aparezca al completo en la imagen, lo detectaremos correctamente ya que su área en la máscara binaria tras realizar la apertura será mucho mayor.

Figura 8-17. Código de barras con reflejo de la luz encima.

Figura 8-18. Ejemplo de código de barras con reflejo filtrado.

En la figura 8.18, vemos como el código de barras del centro de la figura 8-17 con el reflejo no se detecta, una vez avance el robot y el reflejo no esté sobre el código, se detectará correctamente.

Código 8-1. Función para identificar los posibles códigos de barras en la etapa 3 (I).

```
#Función de la etapa 3. Busca líneas horizontales en la mitad de la imagen, ya que
la imagen es solo la banda de la que estamos haciendo el barrido.
def calcula_codigos(image, height, width):
    plot_banda = 0
    plot_gradientes = 0
    plot_morfologia = 1
    edges, lines = Hough(image, 7)

    vector_alturas_unidas = []
    vector_angulos_unidos = []
    for i in range(len(lines)):
        vector_alturas_unidas.append(lines[i][0][0])
        vector_angulos_unidos.append(lines[i][0][1])

    print('rho:', vector_alturas_unidas)
    print('theta:', vector_angulos_unidos)

    img_copy = pintar_lineas(image, height, width, lines, None, None)

    #Miro la altura de las líneas y desecho las que representan la misma línea
horizontal para quedarme solo con una
    vector_alturas, vector_angulos =
seleccion_lineas_definitivas(vector_alturas_unidas, vector_angulos_unidos,
separacion = 300)

    #Ordenamos alturas para formar máscara
    tam_vector, alturas_ordenadas, angulos_ordenados =
ordena_alturas(vector_alturas, vector_angulos)

    #Formamos "vector_mascara" y creamos la máscara
    vector_mascara = []

    #Filtramos las líneas según sus alturas (nos quedamos con aquellas que están en
la franja central de la imagen)
    alturas_ordenadas = [x for x in alturas_ordenadas if x > 850 and x < 2150]

    print('Altura banda zoom:',alturas_ordenadas)
    print('')

    vector_mascara.append([alturas_ordenadas[0], alturas_ordenadas[1]])
    #Máscara que elimina todo menos la banda
    mascara = creacion_mascara(height, width, vector_mascara, flag = 1)

    #Aplicamos la máscara
    target = cv2.bitwise_and(image,image, mask=mascara)
    target_gray = cv2.cvtColor(target, cv2.COLOR_RGB2GRAY)

    #Suavizado previo a calcular los gradientes vertical y horizontal de la imagen
en escala de grises
    blurred = cv2.GaussianBlur(target_gray,(15,15),0)

    #Cálculo de gradientes y conversión a valores enteros positivos
    grad_x = cv2.Sobel(blurred, cv2.CV_16S, 1, 0, ksize=3, scale=1, delta=0,
borderType=cv2.BORDER_DEFAULT)
    abs_grad_x = cv2.convertScaleAbs(grad_x)

    grad_y = cv2.Sobel(blurred, cv2.CV_16S, 0, 1, ksize=3, scale=1, delta=0,
borderType=cv2.BORDER_DEFAULT)
    abs_grad_y = cv2.convertScaleAbs(grad_y)

    #Cálculo gradiente absoluto
    gradient1 = cv2.subtract(grad_x, grad_y)
    gradient2 = cv2.convertScaleAbs(gradient1)

     #Se binariza la imagen
    umbral,binaria = cv2.threshold(gradient2,100,255,cv2.THRESH_BINARY)
```

Código 8-2. Función para identificar los posibles códigos de barras en la etapa 3 (II).

```
    #Se hace un cierre para formarlos rectángulos que engloban a los códigos de
barras
    kernel1 = cv2.getStructuringElement(cv2.MORPH_RECT, (20, 5))           #
(Ancho, alto)
    closed = cv2.morphologyEx(binaria, cv2.MORPH_CLOSE, kernel1)

    #Luego se hace una apertura para limpiar la imagen y quedarnos solo con los
rectángulos de los códigos de barras
    kernel2 = cv2.getStructuringElement(cv2.MORPH_RECT, (45, 45))        # (Ancho,
alto)
    opened = cv2.morphologyEx(closed, cv2.MORPH_OPEN, kernel2)

    #Resultado con la máscara que idealmente solo tiene los rectángulos de los
códigos de barras, aunque puede tener puntos sueltos
    masked = cv2.bitwise_and(image, image, mask=opened)

    #Realizamos un etiquetado de la imagen binaria
    output = cv2.connectedComponentsWithStats(opened, 4, cv2.CV_32S)
    (numLabels, labels, stats, centroids) = output

    mascara_area = numpy.zeros((height, width), dtype="uint8")

    for i in range(1, numLabels):        #Obviamos el índice 0 que corresponde al
fondo negro
        x = stats[i, cv2.CC_STAT_LEFT]
        y = stats[i, cv2.CC_STAT_TOP]
        w = stats[i, cv2.CC_STAT_WIDTH]
        h = stats[i, cv2.CC_STAT_HEIGHT]
        area = stats[i, cv2.CC_STAT_AREA]

        #Umbral de píxeles que se aplica a cada etiqueta para eliminar los posibles
puntos blancos sueltos de la imagen y, quedarnos seguro solo con los rectángulos de
los códigos de barras
        ok_area = area > 25000

        #Si el área de esa etiqueta es superior, añadimos esa etiqueta a la nueva
máscara "mascara_area" que ya no contendrá los puntos sueltos
        if ok_area == True:
            componentMask = (labels == i).astype("uint8") * 255
            mascara_area = cv2.bitwise_or(mascara_area, componentMask)

    #Dilatado para asegurarnos que los rectángulos circunscriben a los códigos de
barras por completo
    kernel3 = cv2.getStructuringElement(cv2.MORPH_RECT, (50, 50))        # (Ancho,
alto)
    dilated = cv2.dilate(mascara_area, kernel3)

    #Resultado con la máscara que tiene los códigos de barras cirncunscritos por
completos
    masked2 = cv2.bitwise_and(image, image, mask=dilated)

    print('-------------------------------------------------------------------------
-------------------------------------------------------------------------------
--------')
    print('')

    return target, target_gray, blurred, binaria, closed, opened, masked,
mascara_area, dilated, masked2
```

En cuanto a la función principal, se calculan los contornos usando la función **findContours** de OpenCV de la imagen dilatada (variable **dilated)**, es decir, el contorno de los rectángulos que circunscriben a los códigos de barras, y se calcula también el centroide de dichos rectángulos. Para esta última tarea se recurre a la función **moments** de OpenCV, que como su nombre indica, calcula los momentos en este caso de la variable devuelta por **findContours** que es **contours**.

El momento de orden r, s [13] se define como la fórmula siguiente.

$$m_{r,s} = \sum_{x=1}^{N} \sum_{y=1}^{M} x^r \cdot y^s \cdot p(y,x) \tag{8-3}$$

Se trata de un sumatorio evaluado en toda la imagen, donde x es la coordenada horizontal del punto, y es la coordenada vertical del punto y p es la imagen binaria de valores 0 o 1. De esta forma si por ejemplo calculamos m00 tendríamos.

$$m_{r,s} = \sum_{x=1}^{N} \sum_{y=1}^{M} p(y,x) \tag{8-4}$$

Luego estaríamos calculando el número de píxeles blancos en la imagen binaria porque sería un sumatorio de todos los píxeles de valor 1, ya que los píxeles con valor 0 no modificarían el valor del sumatorio. En nuestro caso, los píxeles de valor 1 son aquellos píxeles blancos de la imagen **dilated**. Pero a la función **moments** se le pasa cada contorno de manera independiente, luego los cálculos se pueden realizar de manera aislada para cada rectángulo blanco. De igual manera, el centroide o centro de masas de cada región blanca se calcula como:

$$x_{cm} = \frac{m_{10}}{m_{00}} \tag{8-5}$$

$$y_{cm} = \frac{m_{01}}{m_{00}} \tag{8-6}$$

Donde Xcm es la coordenada horizontal del centro de masas e Ycm es la coordenada vertical del centro de masas de cada rectángulo.

Y la función principal de la etapa 3 es la siguiente.

Código 8-3. Función principal de la etapa 3.

```
#Función principal de la etapa 3
def funcion_principal():
    mypath='E:\\Documents\\Juan de Dios\\TFG\\Fotos gasolinera\\Zoom'
    onlyfiles = [ f for f in listdir(mypath) if isfile(join(mypath,f)) ]
    images = numpy.empty(len(onlyfiles), dtype=object)
    for n in range(0, len(onlyfiles)):
        images[n] = cv2.imread( join(mypath,onlyfiles[n]) )
        images[n] = cv2.cvtColor(images[n], cv2.COLOR_BGR2RGB)

        height, width, channels = images[n].shape

        target, target_gray, blurred, binaria, closed, opened, masked,
mascara_area, dilated, masked2 = calcula_codigos(images[n], height, width)

        # Se buscan los contornos de los códigos de barras (rectángulos) y se
pintan
        contours, hierarchy = cv2.findContours(dilated, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
        image_copy = images[n].copy()

        #Obtención del píxel central de los códigos de barras identificados
        for i in contours:
            M = cv2.moments(i)
            if M['m00'] != 0:
                cx = int(M['m10']/M['m00'])
                cy = int(M['m01']/M['m00'])
                cv2.drawContours(image_copy, [i], -1, (0, 255, 0), 6)
                cv2.circle(image_copy, (cx, cy), 40, (255, 0, 0), -1)

        plot_deteccion = 1
        if plot_deteccion == 1:
            plt.subplot(211),plt.imshow(images[n])
            plt.title('Imagen original'), plt.xticks([]), plt.yticks([])
            plt.subplot(212),plt.imshow(image_copy)
            plt.title('Códigos detectados'), plt.xticks([]), plt.yticks([])
            plt.show()
```

Sobre la imagen original pintamos tanto el contorno de los rectángulos como sus centroides, que aproximadamente coincidirán con los centroides de los códigos de barras identificados.

Figura 8-19. Códigos de barras identificados tras ejecutar la etapa 3.

Si se usasen imágenes de los otros dos comercios, simplemente habría que cambiar el tamaño de las máscaras para realizar el cierre, la apertura y el dilatado. Se ha considerado que no aporta nada adicional al trabajo y es por ello que solo se trabajan con imágenes de un comercio para esta etapa.

8.6 Modificación del vector de aprendizaje

Es el momento de modificar **vector_aprendizaje**, si no detectamos códigos de barras en la banda de la que estamos haciendo el barrido, significará que esa banda es errónea y que se encuentra en zona de productos. Si por el contrario hemos identificado algún código de barras, significará que es una banda correctamente identificada. La manera de modificar **vector_aprendizaje** según los valores de **vector_mascara** es la siguiente. Vamos a tomar un ejemplo de tres bandas.

8.6.1 Caso con el vector de aprendizaje no completo

En primer lugar, si tenemos una banda en **vector_aprendizaje** no definida aún, y la banda correspondiente en la misma posición en **vector_mascara** es una banda en la que hemos detectado código, automáticamente se le asigna el valor de la banda del **vector_mascara**. Veamos esto en un ejemplo.

vector_aprendizaje vector_mascara

Figura 8-20. Modificación del vector_aprendizaje cuando una de sus bandas no tiene valor aún.

Las bandas verdes indican que se han detectado códigos de barras en ellas, mientras que las bandas rojas indican que no se ha detectado. Si la banda de **vector_mascara** es incorrecta, es decir, no hemos detectado código, no hacemos nada, pero si la banda es correcta, se compara esa banda con su correspondiente del **vector_aprendizaje**, esto significa que, si por ejemplo la segunda banda es correcta, la compararemos con la segunda banda de **vector_aprendizaje**.

En este caso, al no tener definida la segunda banda aún, asignamos directamente el valor de la banda de **vector_mascara** ya que es correcta, si fuese incorrecta, la segunda banda de **vector_aprendizaje** seguiría sin estar definida.

Figura 8-21. Resultado del vector de aprendizaje tras completar banda que estaba vacía.

8.6.2 Caso con el vector de aprendizaje completo

En caso de que todas las bandas estén definidas en **vector_aprendizaje**, su modificación es diferente.

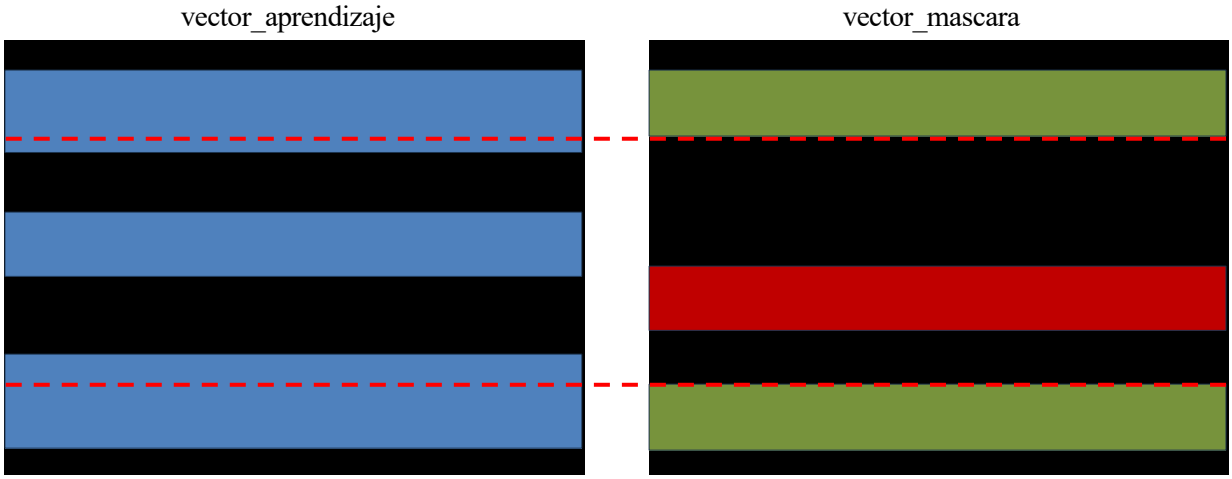

Figura 8-22. Modificación del vector_aprendizaje cuando todas sus bandas están definidas.

Para este caso, comparamos los límites de las bandas que ocupan la misma posición. Por ejemplo, la banda primera de **vector_mascara** es correcta, luego comparamos sus límites con la banda primera de **vector_aprendizaje**. El límite superior es el mismo en ambos casos luego no cambia, pero si nos fijamos vemos que el límite inferior de la banda de **vector_mascara** está más arriba, luego con una banda más estrecha seguimos identificando códigos de barras, luego el límite inferior de la banda de **vector_aprendizaje** pasará a ser el límite inferior de la banda de **vector_mascara**.

La segunda banda es errónea luego no modificamos la segunda banda de **vector_aprendizaje**. En cuanto a la tercera banda, el límite inferior de ambas es el mismo luego no cambia, si el límite inferior de la banda de **vector_mascara** estuviese por encima, modificaríamos el de **vector_aprendizaje** ya que acotaríamos aún más la tercera banda. Es lo que ocurre con el límite superior, es más bajo que el de **vector_aprendizaje** logrando una banda más estrecha en la que también se identifican códigos de barras.

El resultado final sería el siguiente.

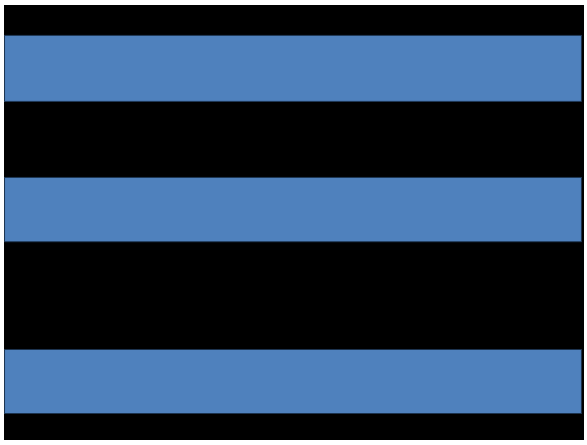

Figura 8-23. Resultado del vector de aprendizaje cuando ya tiene valor en todas las bandas.

Evidentemente se puede dar el caso en que se deba modificar tanto el límite superior como el inferior, simplemente se ha separado para facilitar la comprensión de lo que se ha planteado.

8.6.3 Código propuesto

Se deja un ejemplo de cómo podría ser el código para realizar esta parte de modificar el **vector_aprendizaje** y que es la misma idea que se ha usado al final de la etapa 1. Creamos un vector del mismo tamaño que **vector_mascara** y **vector_aprendizaje**, por ejemplo **banda_correcta**. En él guardamos un valor **False** en aquellas posiciones donde la banda sea correcta y se hayan podido identificar códigos de barras y **True** en caso contrario. Para ello podemos partir de la variable **dilated** que contenía los códigos detectados en caso de que hubiese. Era una imagen binaria con valores 0 o 255. Para comprobar si **dilated** solo contiene ceros podemos usar la librería **numpy** haciendo lo siguiente.

$$\text{Todo ceros} = \text{not}\ \ \text{numpy.any}(\text{dilated}) \tag{8-7}$$

Si obtenemos valor **True**, significa que **dilated** solo contiene ceros y por tanto no hemos identificado ningún código de barras. Por el contrario, si obtenemos valor **False**, significa que no todo son ceros y entonces sí hemos detectado algún código de barras.

Recorremos el vector y si el elemento es **False** y esa banda no está definida aún en **vector_aprendizaje**, se copia directamente la de **vector_mascara** ya que la banda es correcta. En caso de que ya tenga valor, se mira si el límite superior de la banda de **vector_mascara** está más hacia abajo, o si el límite inferior está más hacia arriba para así acotar más las bandas. Hay que recordar que las coordenadas de la imagen crecen hacia abajo verticalmente y hacia la derecha horizontalmente, luego que algo esté más hacia abajo significa que tiene una componente “y” mayor.

Código 8-4. Modificación del vector de aprendizaje.

```
for i in range(len(vector_mascara)):
    if banda_correcta[i] == False:     #Significa que no todo es '0' en
dilated y que por tanto hemos identificado códigos de barras
        #Primera vez que se le da valor a una de las posiciones del vector
de aprendizaje
        if vector_aprendizaje[i][1] == 0:
            vector_aprendizaje[i] = [vector_mascara[i][0],
vector_mascara[i][1]]
        #Si hemos conseguido leer código de una banda con el límite
superior más hacia abajo, es porque la banda la podemos estrechar más y
seguir leyendo
        if vector_aprendizaje[i][0] != 0 and vector_aprendizaje[i][0] <
vector_mascara[i][0]:
            vector_aprendizaje[i][0] = vector_mascara[i][0]
        #Si hemos conseguido leer código de una banda con el límite
inferior más hacia arriba, es porque la banda la podemos estrechar más y
seguir leyendo
        if vector_aprendizaje[i][1] > vector_mascara[i][1]:
            vector_aprendizaje[i][1] = vector_mascara[i][1]
```

Un código similar a este será el que usemos en la etapa 1 para poder modificar el **vector_aprendizaje** y así podremos ver cómo funciona esa etapa usando la información contenida en **vector_aprendizaje**. Aunque para hacerlo más intuitivo en la etapa 1 se identifica como **True** las bandas correctas y como **False** las bandas incorrectas.

9 ETAPA 4: ZOOM HACIA LOS POSIBLES CÓDIGOS DE BARRAS

Conociendo los centroides de los códigos de barras gracias a la etapa 3, simplemente se trata de encuadrarlos en la imagen, pero pasando en este caso como coordenadas los centroides de los diferentes códigos de barras identificados. Además, se aumentaría nuevamente el zoom para así facilitar la tarea a la función de OpenCV encargada de decodificar los códigos de barras en la siguiente etapa. Partimos de una imagen como la siguiente:

Figura 9-1. Ejemplo de imagen de la etapa 3.

Y pasaríamos a una imagen como la que se muestra a continuación.

Figura 9-2. Ejemplo de imagen obtenida tras realizar la etapa 4.

Para este paso, antes de hacer el encuadre a los centroides códigos de barras, es importante guardar la posición de la cámara antes de realizar dicho encuadre, para que una vez hagamos zoom a un código y realicemos lo que se comentará en la siguiente etapa, podamos volver a las condiciones anteriores de la cámara que es donde conocemos las coordenadas de los centroides de los demás códigos de barras identificados.

[18] Esta posición de la cámara son los tres grados de libertad que tiene la cámara real que es una PTZ (Pan Tilt Zoom). Las tres letras de su nombre son los tres grados de libertad: pan es la rotación, tilt es la inclinación para subir y bajar la cámara y zoom el aumento del tamaño de los objetos modificando la distancia focal.

10 ETAPA 5: ALGORITMO DE DECODIFICACIÓN

Llegamos a la última etapa del algoritmo. Una vez tenemos una imagen como la figura 9-2, creamos del módulo **barcode** de OpenCV [19], un objeto de tipo **BarcodeDetector** que asignamos a la variable **objeto**. Tras esto, usamos la función **.detectAndDecode** del objeto previamente creado para obtener la secuencia numérica del código de barras en cuestión.

Figura 10-1. Código de barras identificado por el algoritmo de OpenCV.

Y el código obtenido para este caso es el siguiente:

```
Código de barra decodificado: True
Decoded info: ('8414339766926',)
Decoded type: (2,)
Corners: [[[1490.702  1446.4498]
  [1491.7421 1270.6718]
  [2757.3447 1278.1595]
  [2756.3047 1453.9376]]]
Lista x: [1490, 1491, 2757, 2756]
Lista y: [1446, 1270, 1278, 1453]

Código de barra decodificado: True
Decoded info: ('8437013070027',)
Decoded type: (2,)
Corners: [[[1360.1848 1543.8779]
  [1358.6267 1397.4019]
  [2448.409  1385.8096]
  [2449.967  1532.2856]]]
Lista x: [1360, 1358, 2448, 2449]
Lista y: [1543, 1397, 1385, 1532]
```

Figura 10-2. Código de barras decodificado.

El tipo de código vemos que es EAN-13 que corresponde con el número dos de "Decoded type", y la secuencia numérica obtenida para el código de la figura 10-1 se muestra en "Decoded info" que vemos que es correcta.

Código 10-1. Función principal de la etapa 5.

```
objeto = cv2.barcode_BarcodeDetector()

def funcion_principal():
    mypath='E:\\Documents\\Juan de Dios\\TFG\\Dataset etapa 5'
    onlyfiles = [ f for f in listdir(mypath) if isfile(join(mypath,f)) ]
    images = numpy.empty(len(onlyfiles), dtype=object)
    for n in range(0, len(onlyfiles)):
        images[n] = cv2.imread( join(mypath,onlyfiles[n]) )
        images[n] = cv2.cvtColor(images[n], cv2.COLOR_BGR2RGB)
        height, width, channels = images[n].shape

        ok, decoded_info, decoded_type, corners =
objeto.detectAndDecode(images[n])

        # Detección, códigos decodificados, tipos de códigos y esquinas de
los códigos:
        print('Código de barra decodificado:', ok)
        print('Decoded info:',decoded_info)
        print('Decoded type:',decoded_type)
        print('Corners:', corners)                      #"Corners" desde
esquina inferior izquierda en sentido horario

        img_copy = images[n].copy()

        #No se ha detectado código de barras
        if ok == False:
            print('Código de barra no detectado')
            print('')

            plt.subplot(111),plt.imshow(images[n])
            plt.title('Original Image'), plt.xticks([]), plt.yticks([])
            plt.show()

        #Se ha detectado código de barras
        else:
            for i in range(len(corners)):  # Para todos los códigos de
barras
                lista_x = []
                lista_y = []
                for j in range(4):                              #Para un
código de barras concreto
                    x = int(corners[i][j][0])                #Desde esquina
inferior izquierda en sentido horario
                    y = int(corners[i][j][1])
                    lista_x.append(x)
                    lista_y.append(y)
                    cv2.circle(img_copy, (int(x),int(y)), 40, (255,0,0), -
1)
                print('Lista x:', lista_x)
                print('Lista y:', lista_y)
                print('')

            plt.subplot(121),plt.imshow(images[n])
            plt.title('Original Image'), plt.xticks([]), plt.yticks([])
            plt.subplot(122),plt.imshow(img_copy)
            plt.title('Barcode detection'), plt.xticks([]), plt.yticks([])
            plt.show()
```

El algoritmo [19] solo decodifica códigos de barras del tipo EAN-13, EAN-8 y UPC-A, por ello solo hemos podido probar con códigos de este comercio que son del tipo EAN-13, ya que los códigos de las estanterías de Mercadona y la gasolinera utilizados para las otras etapas son de otro tipo.

Con el código numérico ya obtenido, tendríamos que comprobar dicho código en la memoria del sistema para averiguar de qué producto se trata. Finalmente, una vez sabemos qué producto es, se le asignará a dicho producto las coordenadas (x, y, z) actuales del robot en el mapa en ese preciso instante. Recordemos que previamente a la identificación de los productos, hemos creado un mapa del supermercado y entonces sabemos las coordenadas actuales del robot en el mapa.

El motivo de hacer esto así es porque realmente, no necesitamos saber las coordenadas reales de los productos en el mapa. Como se comenta en [1], el sistema robótico final consiste en que el robot planifique y realice una ruta pasando por los diferentes productos que el cliente ha indicado que desea comprar. Luego si al planificador le pasamos directamente las coordenadas a las que debe ir el robot, nos ahorramos realizar algún procesamiento adicional para saber qué coordenadas tendría el robot sabiendo que el producto se encuentra en tales coordenadas. Nos ahorramos ese procesamiento y le pasamos directamente la posición a la que debe ir el robot que estará cerca de la estantería donde se encuentra ese producto. Lo vemos más claramente ahora en un esquema:

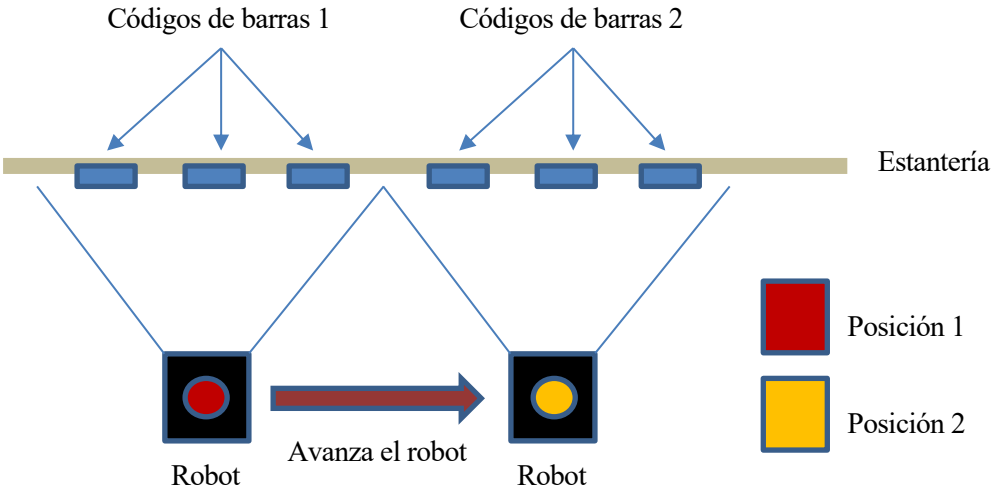

Figura 10-3. Asignación de las coordenadas (x, y, z) a los productos.

A los "Códigos de barras 1" se les asigna la "Posición 1" que será donde el robot va a tener que ir para llegar a esos productos. Lo mismo con los "Códigos de barras 2" y la "Posición 2". Cabe la posibilidad de que algún código de barras sea detectado en dos posiciones distintas del robot, dado que van a ser dos posiciones muy próximas entre sí, se puede simplemente asignar las nuevas coordenadas del robot, y "machacar" las antiguas coordenadas de la primera posición donde se identificó.

Recordemos que en la etapa 1 habremos identificado varias bandas, luego al acabar con la primera banda, habría que realizar todo el proceso otra vez, pero con la siguiente banda. Esto significa repetir el proceso desde la etapa 2 a la etapa 5, pero en este caso en la etapa 2 encuadraríamos la altura media de la segunda banda. Y así hasta acabar con todas las bandas identificadas en la etapa 1. Una vez hecho, el robot avanzaría paralelamente a la estantería una cierta distancia, y se repetiría el proceso desde la etapa 1.

11 RESULTADOS OBTENIDOS Y CONCLUSIONES

En este capítulo, estudiaremos el comportamiento de los diferentes programas desarrollados utilizando imágenes extraídas de tres comercios diferentes.

11.1 Etapa 1

Para el estudio de esta etapa usaremos secuencias de tres imágenes como ya se ha comentado.

11.1.1 Resultados impresos por terminal

En primer lugar, vamos a ver los resultados que imprime el programa por el terminal para una iteración en la que tenemos fase de aprendizaje en un caso de tres bandas a identificar. En algunos pasos pondremos solo algunas partes para reducir el tamaño de las imágenes que se mostrarán ahora, al ejecutar el programa se puede ver al completo si se desea.

Comenzábamos calculando la transformada de Hough a las franjas de manera independiente. Se iteraba aumentando el umbral de 25 en 25 hasta quedarnos con 10 líneas detectadas o menos.

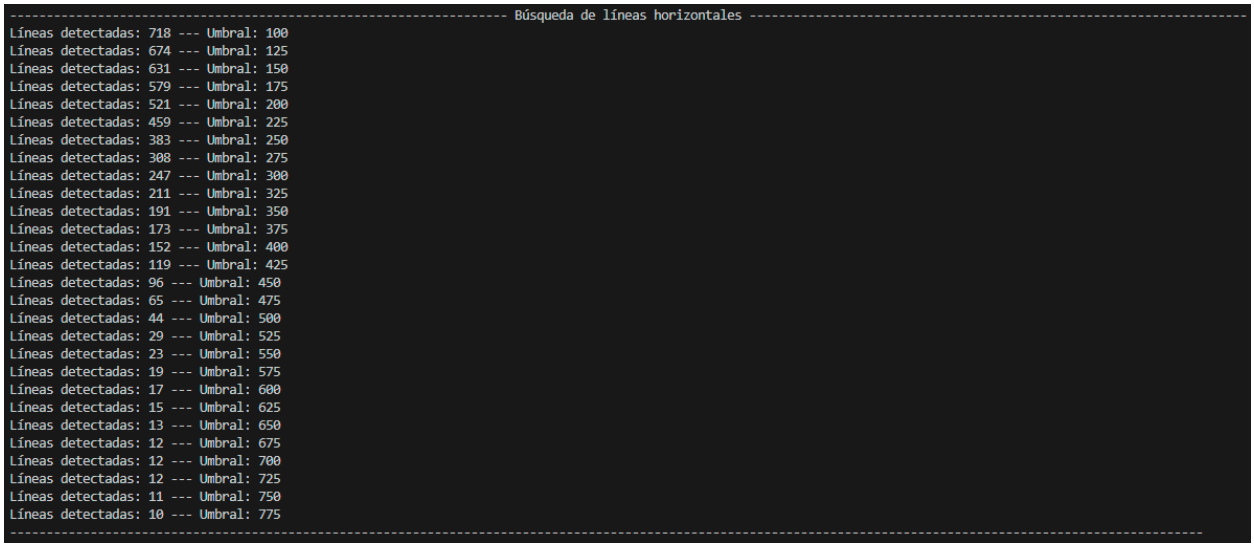

Figura 11-1. Ejemplo de transformada de Hough aplicada a una franja.

Luego veíamos qué parejas (ρ, θ) representaban la misma horizontal para agruparlas.

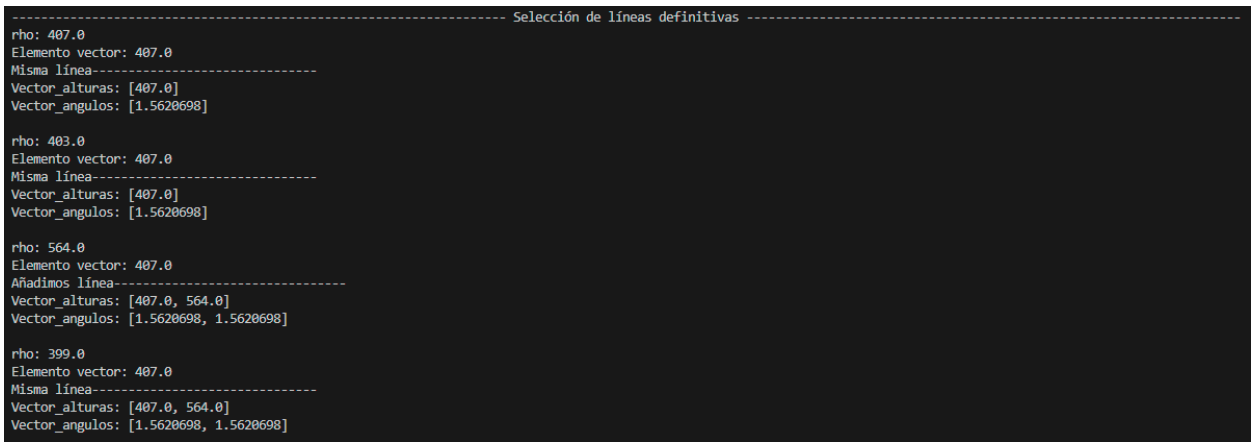

Figura 11-2. Ejemplo de agrupación de líneas.

Una vez seleccionadas las líneas definitivas las ordenábamos.

```
Vector_alturas: [407.0, 564.0, 209.0, 1644.0, 1483.0, 1312.0, 2575.0, 2302.0]
Vector_angulos: [1.5620698, 1.5620698, 1.5620698, 1.5620698, 1.5620698, 1.5620698, 1.5795231, 1.5620698]

--------------------------------------------------------------------------------------------------------------------------------
Tamaño vector: 8
Vector alturas ordenados: (209.0, 407.0, 564.0, 1312.0, 1483.0, 1644.0, 2302.0, 2575.0)
Vector ángulos ordenados: (1.5620698, 1.5620698, 1.5620698, 1.5620698, 1.5620698, 1.5620698, 1.5620698, 1.5795231)
```

Figura 11-3. Ordenamiento de las líneas definitivas.

Tras esto formamos las parejas.

```
------------------------------------------------------------------ Emparejamiento de líneas ------------------------------------------------------------------
Alturas: 209.0 - 407.0 - 564.0
Líneas emparejadas: [209.0, 564.0]
[[209.0, 564.0]]

Alturas: 1312.0 - 1483.0 - 1644.0
Líneas emparejadas: [1312.0, 1644.0]
[[209.0, 564.0], [1312.0, 1644.0]]

Alturas: 2302.0 - 2575.0
Líneas emparejadas: [2302.0, 2575.0]
[[209.0, 564.0], [1312.0, 1644.0], [2302.0, 2575.0]]

Valor de "i" al salir del bucle de emparejamiento: 8

Vector máscara: [[209.0, 564.0], [1312.0, 1644.0], [2302.0, 2575.0]]
Vector alturas: [209.0, 564.0, 1312.0, 1644.0, 2302.0, 2575.0]
Número de parejas: 3
Vector ocupación: [1. 1. 1.]
Vector líneas desechadas: []
Índices líneas desechadas: []
Número de lineas desechadas: 0

--------------------------------------------------------------------------------------------------------------------------------
```

Figura 11-4. Ejemplo de emparejamiento de líneas.

A continuación, miramos si es necesario eliminar alguna banda que se encuentre en zona de productos, en este caso no la hay.

```
------------------------------------------------------------ Eliminación bandas en productos -------------------------------------------------------------
Vector máscara: [[209.0, 564.0], [1312.0, 1644.0], [2302.0, 2575.0]]

Índice: 0 --- 564.0 --- 1312.0

Índice: 1 --- 1644.0 --- 2302.0

Numero de parejas 3
Vector máscara: [[209.0, 564.0], [1312.0, 1644.0], [2302.0, 2575.0]]
Vector ocupación: [1. 1. 1.]
Número de parejas actuales: 3
--------------------------------------------------------------------------------------------------------------------------------
```

Figura 11-5. Ejemplo de eliminación de banda en zona de productos.

Ahora formamos el **vector_mascara** definitivo usando la información contenida en el vector de aprendizaje.

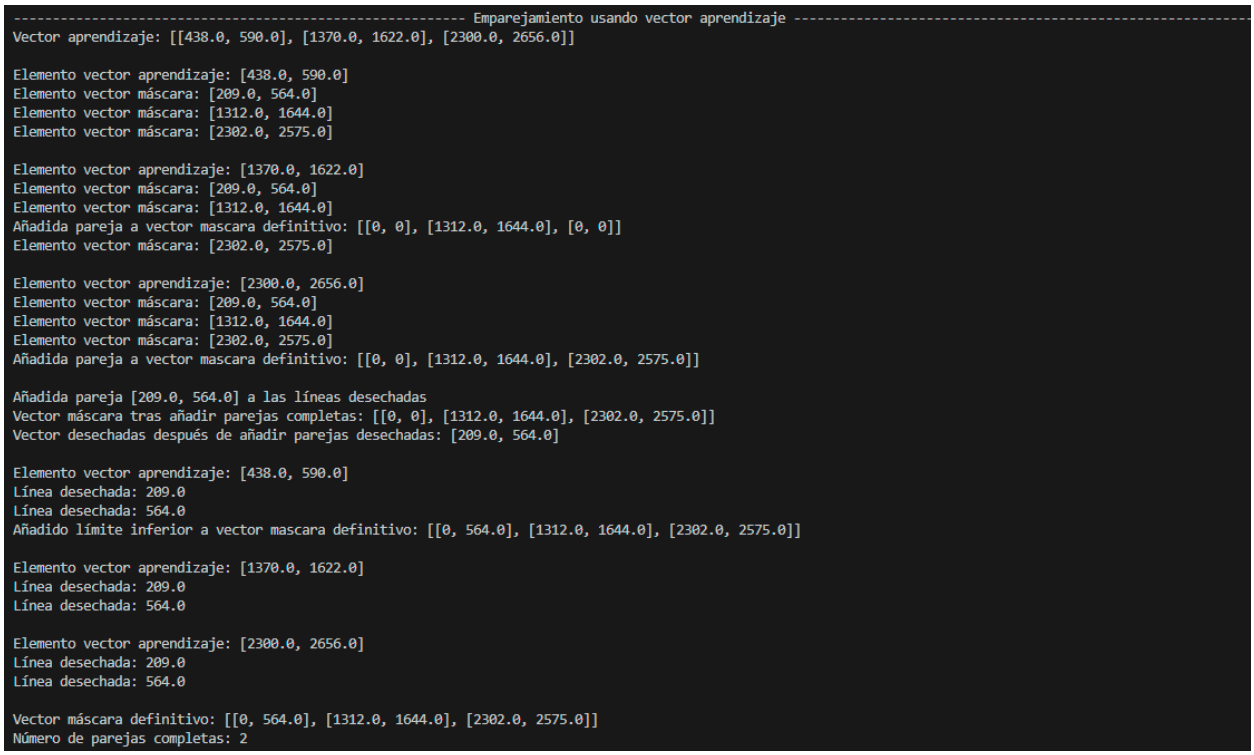

Figura 11-6. Ejemplo de fase de aprendizaje.

Calculamos el ancho y completamos las bandas que estén a medias.

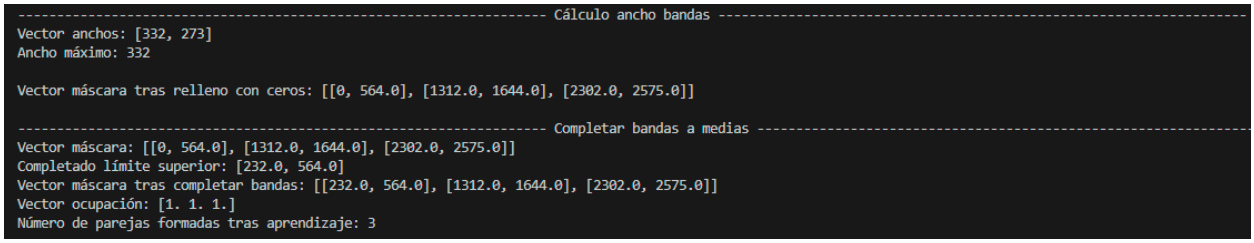

Figura 11-7. Ejemplo del cálculo del ancho de las bandas y completación de las bandas.

En este caso tenemos tres bandas detectadas, luego no hace falta rellenar con bandas artificiales, no obstante, pondremos ahora un ejemplo de otra imagen donde sí se realiza.

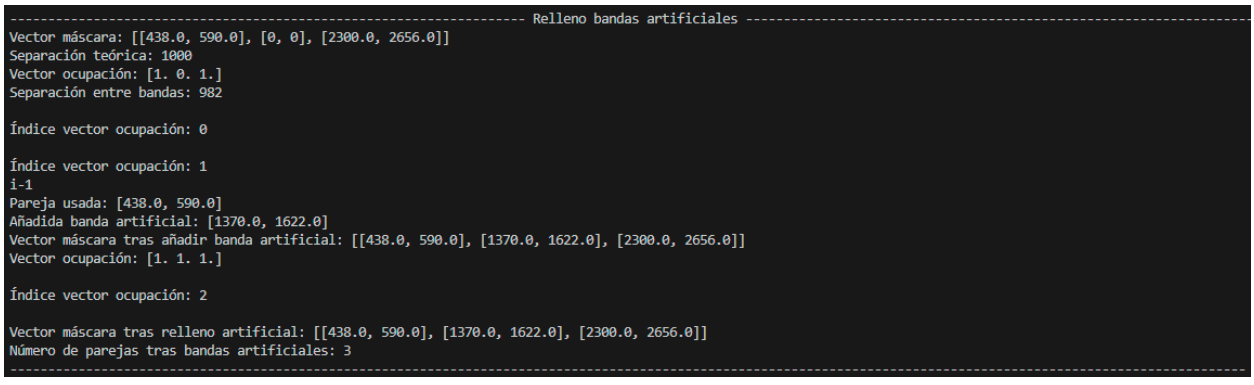

Figura 11-8. Ejemplo de relleno con bandas artificiales.

Finalmente, modificamos el **vector_aprendizaje** retomando la imagen que se ha usado para todas las figuras que llevamos menos la del relleno con bandas artificiales. Recordemos que se hace manualmente, aunque en realidad se haría en la etapa 3 cuando sepamos si realmente había códigos de barras en esa banda o no.

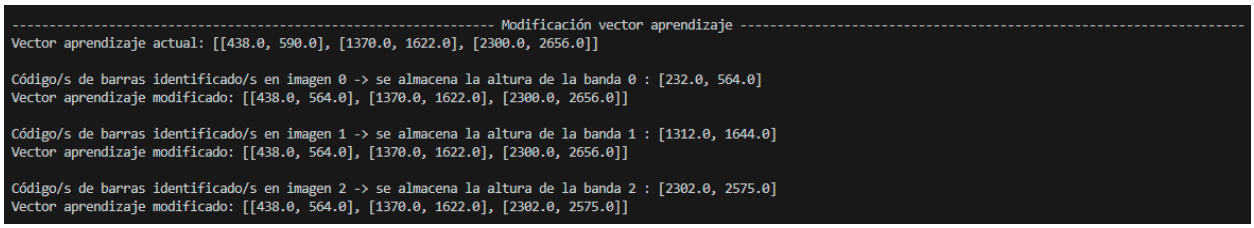

Figura 11-9. Ejemplo de modificación del vector de aprendizaje.

11.1.2 Efectividad del algoritmo

Ahora usaremos las secuencias de imágenes de las que disponemos para evaluar lo efectivo que es el programa identificando las bandas donde se encuentran los códigos de barras en las diferentes imágenes. Los resultados se recogen en la tabla siguiente:

Tabla 11-1. Efectividad de la etapa 1.

Secuencia	Número de bandas	Comercio 1	Comercio 2
1	2	16.67 %	83.33 %
2		0.00 %	66.67 %
1	3	100.00 %	66.67 %
2		44.44 %	77.78 %
3		77.78 %	-
1	4	41.67 %	0.00 %

El porcentaje se ha calculado como:

$$\text{Porcentaje}(\%) = \frac{\text{Bandas totales acertadas}}{\text{Número de bandas totales en la secuencia}} \cdot 100 \qquad (11\text{-}1)$$

Si por ejemplo tenemos 6 bandas acertadas en una secuencia de tres bandas el cálculo será:

$$\text{Porcentaje}(\%) = \frac{6}{3\ \text{bandas} \cdot 3\ \text{imágenes}} \cdot 100 = 66.67\ \% \qquad (11\text{-}2)$$

Para realizar esta tabla, primero ha sido necesario rellenar el **vector_imagenes** correctamente indicando si las bandas identificadas por el programa eran correctas o no.

Estas secuencias eran de tan solo tres imágenes, con secuencias mayores se le da más tiempo al programa a encontrar las bandas correcta y probablemente se obtengan mejores resultados.

Un hecho que se repite a lo largo de todos los resultados obtenidos es que, si en la primera iteración se detectan pocas bandas correctamente, al algoritmo le cuesta más encontrar las demás bandas. En caso de que en la primera iteración no detecte correctamente ninguna banda, al comparar en la siguiente iteración en la fase de aprendizaje con el **vector_aprendizaje** que estará vacío, se desecharán todas las parejas formadas y el algoritmo no devolverá ninguna banda. Este es el mayor problema que presenta el algoritmo.

Es por ejemplo lo que ocurre en las secuencias de dos bandas del comercio 1 o en la secuencia de cuatro bandas del comercio 2, se forman las parejas en la primera iteración en zonas de productos y la efectividad decae drásticamente. En cambio, si se detectan bastantes bandas correctamente en la primera iteración, el algoritmo funciona bastante bien.

También podemos extraer de la tabla que la efectividad depende mucho de la forma de los productos, por ejemplo, en la secuencia de efectividad 100%, se trata de una secuencia de fotos a una estantería de frutas y verduras que carecen de formas rectas y horizontales, luego es más complicado que se formen rectas en zonas de productos.

11.1.3 Tiempo de ejecución

Estudiaremos ahora el tiempo que tarde en ejecutarse el programa.

Tabla 11-2. Tiempo de ejecución de la etapa 1.

Secuencia	Número de bandas	Comercio 1	Comercio 2
1	2	1.67 segundos	0.90 segundos
2		1.50 segundos	1.61 segundos
1	3	2.07 segundos	1.54 segundos
2		1.53 segundos	1.54 segundos
3		1.76 segundos	-
1	4	2.03 segundos	1.77 segundos

Si comparamos con la tabla de la efectividad, en términos generales, aquellas secuencias con porcentaje de aciertos alto son aquellas que tardan más en ejecutarse, mientras que aquellas con porcentaje bajo tardan menos.

Es lógico, ya que por ejemplo si en la primera iteración no detectamos ninguna banda correctamente, el resto de las iteraciones no devolverá ninguna banda y se pasará por todas las funciones sin realizar cálculos ni comprobaciones.

11.2 Etapa 2 y 4

Para estas dos etapas no podemos comprobar nada ya que se trata de encuadrar un punto de la imagen a la vez que hacemos un zoom con la cámara.

11.3 Etapa 3

11.3.1 Resultados impresos por terminal

Al igual que en la etapa 1, veremos un ejemplo de lo que imprime el programa por el terminal.

Primero se identificaban líneas horizontales para quedarnos con las dos que estuviesen en la franja central de la imagen.

```
------------------------------------------------------------------ Búsqueda de líneas horizontales ------------------------------------------------------------------
Líneas detectadas: 924 --- Umbral: 100
Líneas detectadas: 689 --- Umbral: 125
Líneas detectadas: 490 --- Umbral: 150
Líneas detectadas: 357 --- Umbral: 175
Líneas detectadas: 236 --- Umbral: 200
Líneas detectadas: 162 --- Umbral: 225
Líneas detectadas: 130 --- Umbral: 250
Líneas detectadas: 96 --- Umbral: 275
Líneas detectadas: 68 --- Umbral: 300
Líneas detectadas: 54 --- Umbral: 325
Líneas detectadas: 49 --- Umbral: 350
Líneas detectadas: 42 --- Umbral: 375
Líneas detectadas: 36 --- Umbral: 400
Líneas detectadas: 30 --- Umbral: 425
Líneas detectadas: 27 --- Umbral: 450
Líneas detectadas: 22 --- Umbral: 475
Líneas detectadas: 18 --- Umbral: 500
Líneas detectadas: 15 --- Umbral: 525
Líneas detectadas: 15 --- Umbral: 550
Líneas detectadas: 15 --- Umbral: 575
Líneas detectadas: 14 --- Umbral: 600
Líneas detectadas: 12 --- Umbral: 625
Líneas detectadas: 11 --- Umbral: 650
Líneas detectadas: 10 --- Umbral: 675
Líneas detectadas: 8 --- Umbral: 700
Líneas detectadas: 8 --- Umbral: 725
Líneas detectadas: 8 --- Umbral: 750
Líneas detectadas: 7 --- Umbral: 775
------------------------------------------------------------------------------------------------------------------------------------------------------------------

rho: [1315.0, 1336.0, 2050.0, 2056.0, 2038.0, 619.0, 2046.0]
theta: [1.5620698, 1.5620698, 1.5620698, 1.5620698, 1.5620698, 1.5795231, 1.5620698]
```

Figura 11-10. Ejemplo de transformada de Hough para aislar banda en etapa 3.

Ahora agrupamos las líneas detectadas (no se muestran todas las iteraciones).

```
------------------------------------------------------------------ Selección de líneas definitivas ------------------------------------------------------------------
rho: 1315.0
Elemento vector: 1315.0
Misma línea------------------------------
Vector_alturas: [1315.0]
Vector_angulos: [1.5620698]

rho: 1336.0
Elemento vector: 1315.0
Misma línea------------------------------
Vector_alturas: [1315.0]
Vector_angulos: [1.5620698]

rho: 2050.0
Elemento vector: 1315.0
Añadimos línea------------------------------
Vector_alturas: [1315.0, 2050.0]
Vector_angulos: [1.5620698, 1.5620698]

rho: 2056.0
Elemento vector: 1315.0
Elemento vector: 2050.0
Misma línea------------------------------
Vector_alturas: [1315.0, 2050.0]
Vector_angulos: [1.5620698, 1.5620698]

rho: 2038.0
Elemento vector: 1315.0
Elemento vector: 2050.0
Misma línea------------------------------
Vector_alturas: [1315.0, 2050.0]
Vector_angulos: [1.5620698, 1.5620698]
```

Figura 11-11. Ejemplo de agrupación de líneas para aislar la banda.

Finalmente ordenamos el vector de las alturas y filtramos para quedarnos con las que están entre 850 y 2150.

```
Tamaño vector: 3
Vector alturas ordenados: (619.0, 1315.0, 2050.0)
Vector ángulos ordenados: (1.5795231, 1.5620698, 1.5620698)

Altura banda zoom: [1315.0, 2050.0]
```

Figura 11-12. Ejemplo de selección de líneas según su altura.

Con todo esto, es cuando realizamos todo el procesamiento como el cálculo de gradientes o el algoritmo de etiquetado hasta llegar al resultado final.

La imagen de la que hemos obtenido los resultados que acabamos de mostrar es la siguiente.

Figura 11-13. Imagen de partida de los resultados obtenidos.

La máscara obtenida con los códigos identificados se muestra a continuación.

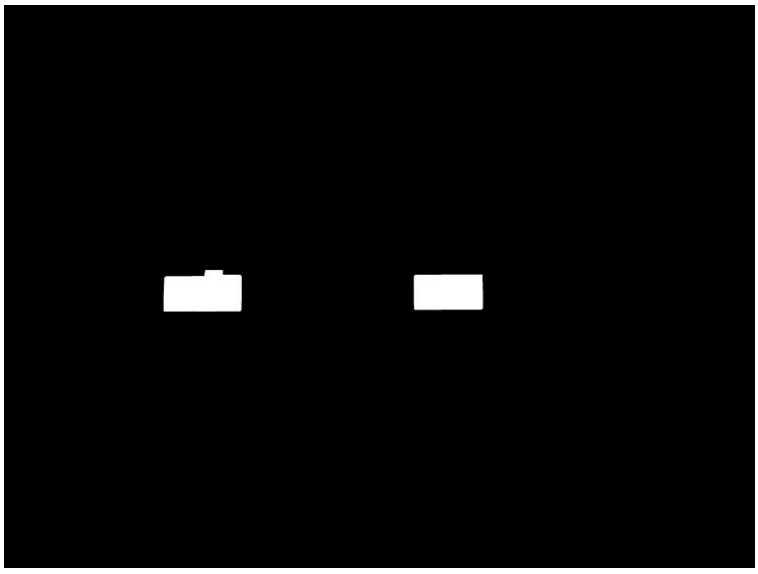

Figura 11-14. Máscara binaria obtenida al procesar la figura 11-13.

Y aplicando la máscara a la imagen original obtenemos los códigos identificados.

Figura 11-15. Códigos identificados.

11.3.2 Efectividad del algoritmo

Con las cuatro imágenes usadas para evaluar el programa de la etapa 3, se ha obtenido un porcentaje de acierto del 100 % para cada una de ellas.

Tabla 11-3. Efectividad de la etapa 3.

Imagen	Porcentaje de aciertos
1	100 %
2	100 %
3	100 %
4	100 %

El problema de esta etapa es que, si el tamaño de los códigos varía respecto al tamaño de los que estamos usando, el tamaño de las máscaras para realizar el cierre, la apertura y el dilatado ya no serían válidos, habría que cambiarlos, luego el programa funciona correctamente siempre y cuando se respete esta condición. Si usásemos fotos de los otros dos comercios donde el tamaño de los códigos de barras es distinto, las máscaras deberían ser modificadas.

También puede darse el caso de que no se aísle bien la banda, de las cinco imágenes probadas solo ha pasado en una ocasión y se ha decidido eliminar, ya que el resto de los pasos no se iban a poder realizar luego carecía de sentido dejarla en el dataset.

11.3.3 Tiempo de ejecución

El tiempo para identificar los códigos de barras en las imágenes han sido los siguientes:

Tabla 11-4. Tiempo de ejecución de la etapa 3.

Imagen	Tiempo de ejecución (segundos)	Número de códigos de barras
1	0.61	2
2	0.75	3
3	0.79	3
4	0.58	1

Viendo el número de códigos a identificar vemos que la última imagen solo tiene un código y es la que tarda menos en procesarse. La primera imagen tiene dos códigos y es la segunda imagen más rápida. En el caso de la segunda imagen tiene tres códigos, aunque uno se termina desechando porque sale cortado y la tercera imagen también tiene tres códigos, ambos son los que tardan más. Luego podemos ver que mientras más códigos se procesen en la imagen más tiempo tarda el programa en ejecutarse.

11.4 Etapa 5

11.4.1 Resultados impresos por terminal

Probaremos con la siguiente imagen del dataset de esta etapa.

Figura 11-16. Imagen de partida para ejemplo de la etapa 5.

Y por terminal se obtiene lo siguiente:

```
Código de barra decodificado: True
Decoded info: ('8414339766926',)
Decoded type: (2,)
Corners: [[[1490.702  1446.4498]
  [1491.7421 1270.6718]
  [2757.3447 1278.1595]
  [2756.3047 1453.9376]]]
Lista x: [1490, 1491, 2757, 2756]
Lista y: [1446, 1270, 1278, 1453]
```

Figura 11-17. Ejemplo de resultado obtenido en la etapa 5.

11.4.2 Efectividad del algoritmo

Esta etapa simplemente es utilizar la librería de OpenCV. Aunque sí es cierto que se ha detectado que para que el algoritmo decodifique correctamente el código de barras, la imagen debe ser muy nítida. Vamos a ver un ejemplo donde el algoritmo falla.

Figura 11-18. Ejemplo de código que el algoritmo de OpenCV decodifica erróneamente.

Y se obtiene el siguiente código:

```
Código de barra decodificado: True
Decoded info: ('80867111',)
Decoded type: (1,)
Corners: [[[1643.2635 1516.0157]
  [1641.9384 1313.961 ]
  [2810.7896 1306.2941]
  [2812.1147 1508.3488]]]
Lista x: [1643, 1641, 2810, 2812]
Lista y: [1516, 1313, 1306, 1508]
```

Figura 11-19. Código erróneo obtenido de la figura 11-18.

El problema es que los cuatro puntos que el algoritmo sitúa no engloban todo el ancho del código, la parte de más a la izquierda se corta.

Figura 11-20. Colocación errónea de las esquinas del algoritmo de decodificación de OpenCV.

Vemos que las primeras barras de la izquierda se quedan fuera del rectángulo definido por los cuatro puntos rojos. Lo importante para que el algoritmo funcione correctamente es que todo el código a lo ancho quede dentro de los puntos obtenidos, cosa que no ocurre en este ejemplo.

11.4.3 Tiempo de ejecución

Los tiempos de ejecución se recogen ahora para las imágenes del dataset:

Tabla 11-5. Tiempo de ejecución de la etapa 5.

Número de imagen	Tiempo de ejecución (segundos)
1	0.026
2	0.025
3	0.025
4	0.025

Podemos concluir que es un algoritmo que está bastante optimizado.

11.5 Conclusiones

Recordemos que los objetivos eran plantear una funcionalidad que se pudiese aplicar en un entorno real y reducir el número de falsos positivos.

11.5.1 Primer objetivo

En cuanto al primer objetivo, es cierto que no se ha llegado a poner a prueba el algoritmo implementado en el robot en un supermercado, pero debemos ser conscientes de la complejidad del problema que se está planteando resolver. No obstante, la solución que se propone a pesar de que dependa mucho de lo bien que vaya la primera iteración de la etapa 1, puede ofrecer resultados muy buenos en un entorno real donde los problemas y las complejidades que debe afrontar el algoritmo son numerosos.

11.5.2 Segundo objetivo

El segundo objetivo está resuelto en la etapa 3, principalmente el umbral de píxeles a las etiquetas en la imagen binaria permite quedarnos solo con lo que estamos seguros de que son códigos de barras. Además, previo a intervenir en el proyecto, la identificación de los códigos se hacía sobre una pared prácticamente lisa, ahora se realiza en una estantería real lo que supone una mejora importante.

12 LÍNEAS FUTURAS DE TRABAJO

12.1 Probar el algoritmo implementado en el robot en un supermercado real

Lo primero que se debería hacer para continuar este proyecto es lo más evidente, todo lo explicado es una solución que se ha comprobado que presenta un buen funcionamiento en cuanto a la parte de visión por computador, pero es necesario implementarlo en el robot y ponerlo a prueba con todo el sistema en su conjunto en un entorno real, y ver los diferentes problemas y dificultades que se presentan, así como comprobar que el funcionamiento del algoritmo de visión sigue siendo correcto.

Será necesario averiguar también cuál es el zoom adecuado a aplicar en la etapa 2 y 4 para obtener imágenes similares a las que se han mostrado para esas etapas, cuánto debe avanzar el robot paralelamente a la estantería entre una iteración y la siguiente, o cuántos grados debe rotar la cámara cuando haga el barrido de una banda para evitar que no aparezca el mismo código en dos iteraciones distintas y así ahorrar tiempo, pero tampoco pasarnos hasta tal punto que nos saltemos algún código por rotar demasiado entre una imagen y la siguiente.

12.2 Pasar el código a un lenguaje de programación más rápido

A pesar de que todo este proceso de localización de los productos en el mapa se va a realizar una única vez, es evidente que mientras más rápido se ejecute el algoritmo, menor tiempo tardará en realizarlo.

Un cambio por ejemplo de un lenguaje interpretado como es Python a un lenguaje compilado como es C++, puede disminuir el tiempo de ejecución de manera considerable. Ha sido la falta de conocimientos en este último lenguaje lo que no ha permitido desarrollar el proyecto en C++.

12.3 Hacer un algoritmo más eficiente

También es evidente que, se podría tratar de implementar la misma funcionalidad en menos líneas de códigos como cuando, por ejemplo, vamos comparando cada elemento de un vector con todos los elementos de otro vector, quizás se puede hacer de tal manera que no haga falta comparar todos los elementos de un vector con todos los elementos del otro vector, o también se podrían realizar ciertos pasos usando menos variables.

Sobre todo, también debemos contemplar que cada vez que se tome una imagen de la estantería, debemos hacer zoom a cada banda y hacer un barrido de cada banda identificada, luego pasar a la siguiente banda identificada y volver a hacer un barrido, así con todas las bandas identificadas. Esto se debe repetir con todo el supermercado, luego el tiempo para abarcar la tienda entera puede ser incluso de horas. Es cierto que este proceso solo se va a realizar una vez, pero si se puede reducir el tiempo mucho mejor.

12.4 Disminuir la dependencia de la efectividad del algoritmo a la primera iteración y a las líneas identificadas con Hough al principio

Si el algoritmo detecta pocas bandas correctamente en la primera iteración, luego cuando comparemos en la fase de aprendizaje el **vector_mascara** con el **vector_aprendizaje**, al estar este último con algunas posiciones sin definir aún desecharemos en la segunda iteración parejas que incluso pueden ser correctas, provocando que el programa tarde más iteraciones en encontrar las bandas correctas y en acotarlas bien. En el peor de los casos, si en la primera iteración no detectamos ninguna banda correcta, vamos a desechar todas las parejas en la siguiente iteración y el algoritmo no va a devolver ninguna banda.

Como se ha podido comprobar, el algoritmo depende mucho de las líneas identificadas por Hough al comienzo y de las parejas formadas con ellas, aunque como ya se comentó, en la aplicación real con el robot, una vez tengamos un **vector_aprendizaje** bien acotado después de varias iteraciones, se podría asignar

directamente esos valores al **vector_mascara** ya que las alturas de las bandas con el robot serían las mismas en todo momento. Una vez se tengan bien acotadas las bandas, se podría incluso realizar el barrido de la primera banda al completo recorriendo toda la estantería mientras el robot va avanzando paralelamente, luego de la segunda y así sucesivamente hasta barrer todas las bandas de esa estantería.

13 MANUAL DE INSTALACIÓN Y DE USO DEL ALGORITMO

En este último capítulo, comentaremos los pasos necesarios para ejecutar los programas. Los códigos se pueden descargar del siguiente enlace de GitHub:

Enlace códigos	https://github.com/JuandeDiosHerrera/TFG_JDDHH

Una vez descargados los códigos, necesitamos un editor de código donde podamos ejecutar ciertos comandos por terminal y ejecutar los programas como puede ser Visual Studio Code. Para ello podemos acudir al enlace [4]. Será necesario instalar el intérprete de Python que se puede descargar en el apartado "Extensiones".

Recordemos también que se ha usado Python 3.10.4 y OpenCV 4.5.5. Para descargar Pyhon nos vamos a la página oficial [5] a la sección "Downloads" seleccionando la versión comentada. Previo a la instalación de OpenCV, necesitamos instalar pip, que es el instalador de paquetes para Python. Para ello desde el terminal de Visual Studio Code ejecutamos el siguiente comando.

```
pip install pip
```

Y seguidamente para instalar OpenCV.

```
pip install opencv-python==4.5.5
```

Los datasets de imágenes se pueden descargar de los siguientes enlaces de Google Drive:

Dataset etapa 1	https://drive.google.com/drive/folders/1ARsP2UynwnHRz40lK9_xUVjbQCiRYtYp?usp=sharing
Dataset etapa 3	https://drive.google.com/drive/folders/1ToxVPzA2hqscWhkSXpOBAxF8uP9thg10?usp=sharing
Dataset etapa 5	https://drive.google.com/drive/folders/1Jf-e-bmCyzApwpbgf52mZfEoJUEoMtK3?usp=sharing

Es necesario cambiar el directorio donde se encuentran las imágenes que procesaremos y que se almacena en la variable **mypath** para todos los programas. Dicho directorio puede contener espacios en los nombres de las carpetas, pero no tildes. Se recomienda fijarse en el directorio que está puesto. También es muy importante el uso de la doble barra invertida "\\".

Las imágenes y los archivos de los códigos se pueden encontrar en carpetas distintas, basta con pasarle al código la dirección correcta de la carpeta donde se encuentran las imágenes.

Para ejecutar los programas es necesario tener el directorio de Visual Studio Code en la carpeta correspondiente donde se encuentran los códigos. Si queremos ejecutar el código correspondiente a la etapa 1 por ejemplo para identificar 4 bandas en el comercio 2 de la secuencia de imágenes número 1, ejecutaremos lo siguiente.

```
python etapa1.py 4 2 1
```

Donde el primer número indicará el número de bandas a identificar, el segundo el número de comercio y el tercer número la secuencia de imágenes utilizada. Es importante ser cuidadosos y procurar que el número que introduzcamos, coincida realmente con el número de bandas que contienen las imágenes de la secuencia que deseamos procesar, así como el número de secuencia y de comercio para asegurarnos que el programa usa el **vector_imagenes** correcto.

Tanto para el programa correspondiente a la etapa 3 como para el programa correspondiente a la etapa 5, basta con pulsar el botón “Run” para ejecutarlos ya que no precisan de introducir argumentos por la línea de comandos, pero si cambiar el directorio donde se encuentran los datasets.

REFERENCIAS

[1] P. T. Macías Roselló, «Robot móvil con ros para la lectura de códigos de barras mediante zoom óptico en un entorno comercial. Dirigido por Miguel Ángel Ridao Carlini,» 2021.

[2] *Web Wikipedia <<Código de barras>>,* [En línea]. [Último acceso: 2023]. Disponible en: https://es.wikipedia.org/wiki/C%C3%B3digo_de_barras.

[3] *Web Wikipedia <<European Article Number>>,* [En línea]. [Último acceso: 2023]. Disponible en: https://es.wikipedia.org/wiki/European_Article_Number.

[4] *Web Visual Studio Code,* [En línea]. [Último acceso: 2023]. Disponible en: https://code.visualstudio.com/.

[5] *Web Python,* [En línea]. [Último acceso: 2023]. Disponible en: https://www.python.org/.

[6] *Web OpenCV,* [En línea]. [Último acceso: 2023]. Disponible en: https://opencv.org/.

[7] *Web GitHub,* [En línea]. [Último acceso: 2023]. Disponible en: https://github.com/.

[8] *Web Xiaomi <<Redmi Note 10 Pro>>,* [En línea]. [Último acceso: 2023]. Disponible en: https://www.mi.com/es/product/redmi-note-10-pro/specs.

[9] *Web XiaomiAdictos <<Los Redmi Note 10, Note 10S y Note 10 Pro llegan a Europa a un precio de auténtico éxito>>,* [En línea]. [Último acceso: 2023]. Disponible en: https://www.xiaomiadictos.com/.

[10] *Web Xataka <<Móvil: zoom, zoom digital y zoom óptico>>,* [En línea]. [Último acceso: 2023]. Disponible en: https://www.xatakamovil.com/xatakamovil/todo-fotografia-movil-6-que-zoom-digital-zoom-optico-zoom-hibrido.

[11] *Web dZoom <<Zoom Óptico y Zoom Digital: ¿Cuál es Mejor?>>,* [En línea]. [Último acceso: 2023]. Disponible en: https://www.dzoom.org.es/zoom-optico-y-zoom-digital-cual-es-mejor/.

[12] *Web Nikon <<Entendiendo la distancia focal>>,* [En línea]. [Último acceso: 2023]. Disponible en: https://www.nikon.com.mx/learn-and-explore/a/tips-and-techniques/entendiendo-la-distancia-focal.html.

[13] M. V. Villanueva, *Diapositivas de la asignatura "Sistemas de Percepción".*

[14] *Web Chegg,* [En línea]. [Último acceso: 2023]. Disponible en: https://www.chegg.com/homework-help/questions-and-answers/hough-transform-commonly-used-computer-vision-find-straight-line-segments-image-matlab-hou-q85620314.

[15] P. Bodnár y L. G. Nyúl, «Improving Barcode Detection with Combination of Simple Detectors,» *2012 Eighth International Conference on Signal Image Technology and Internet Based Systems,* [En línea]. [Último acceso: 2023]. Disponible en: https://ieeexplore.ieee.org/document/6395110.

[16] *Web PyImageSearch <<Detecting Barcodes in Images with Python and OpenCV>>,* [En línea]. [Último acceso: 2023]. Disponible en: https://pyimagesearch.com/2014/11/24/detecting-barcodes-images-python-opencv/.

[17] *Web PyImageSearch <<OpenCV Connected Component Labeling and Analysis>>,* [En línea]. [Último acceso: 2023]. Disponible en: https://pyimagesearch.com/2021/02/22/opencv-connected-component-labeling-and-analysis/.

[18] *Web Avonic <<What is a PTZ camera?>>,* [En línea]. [Último acceso: 2023]. Disponible en: https://avonic.com/what-is-a-ptz-camera/.

[19] *Web OpenCV <<Recognizing one-dimensional barcode using OpenCV>>,* [En línea]. [Último acceso: 2023]. Disponible en: https://opencv.org/blog/2021/06/28/recognizing-one-dimensional-barcode-using-opencv/.

[20] J. Howse y J. Minichino, Learning OpenCV 4 Computer Vision with Python 3 Third Edition, Birmingham: Packt, 2020.

[21] *Web Stack Overflow para consulta de dudas de programación,* [En línea]. [Último acceso: 2023]. Disponible en: https://stackoverflow.com/.