ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA DEL SOFTWARE

SISTEMA DETECTOR Y CONTADOR DE TABLONES DE MADERA

Análisis y procesado de imágenes para el conteo de tablones de madera

DETECTOR AND COUNTER SYSTEM OF WOODEN PLANKS

Analisys and processed of image for count of wooden planks

Realizado por
Silvia Cuenca Ramos
Tutorizado por
Enrique Domínguez Merino
Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JUNIO 2017

Fecha defensa:
El Secretario del Tribunal

Resumen

Este Trabajo de Fin de Grado se desarrolla como solución a los problemas de las empresas madereras derivados del conteo de los tablones de madera por parte de los operarios. El algoritmo propuesto pretende ahorrar tiempo en este conteo, y evitar los errores derivados del factor humano.

En este proyecto se ha investigado y estudiado como realizar este conteo de forma automática. Para ello se ha investigado acerca de diferentes métodos de procesado de la imagen y de detección de rectángulos para posteriormente realizar un pequeño algoritmo que sirva de base para la posterior realización de una herramienta móvil por parte del resto de los integrantes del equipo.

El algoritmo propuesto ha sido implementado utilizando una interfaz simple, que permita comprobar la exactitud de dicha detección, pero dejando al usuario la tarea de introducir los parámetros correctos en función de la imagen que desea analizar. Estos parámetros son muy importantes, pues de ellos depende que la detección de rectángulos sea más o menos precisa.

Palabras clave

Conteo de rectángulos, tablones de madera, transformada de Hough, procesamiento de imágenes, detección de rectángulos, MATLAB

Abstract

This Final Year Project is developped as a solution for the problem that lumber businesses have, because human operators count wooden planks. The suggested algorithm expects to save time in wooden plank count, and also, avoid errors that are derivated of human factor.

In this project, we have been studied how to perform this task automatically. For that, we have researched about different methods of image processing and rectangle detector to develop a simple algorithm that works of the basis of the mobile application that has been developped by the rest of team members.

The proposed algorithm has been developped using a simple interface, that allows the user to check the accuracy of detection, but user must write parameters correctly, depending on the image that wants to analyze. Those parameters are really important, because they do more or less accurate detection.

Keywords

Rectangle counter, wooden plank, Hough transform, image processing, rectangle detector, MATLAB

Índice general

Índice de figuras

Índice de Códigos

Índice de tablas

1. Introducción

1.1. Motivación

Actualmente las empresas dedicadas al tratamiento o transporte de maderas realizan de forma manual el control del número de tablones, siendo un operario quien realiza este procedimiento, con el riesgo de error que esto conlleva. Esto puede ocasionar grandes pérdidas a la empresa debido al descontento de los clientes y a la pérdida de tiempo que supone realizar esta operación de forma manual, así como la comprobación de que no ha habido ningún error.

Por esta razón, se ha decidido realizar una herramienta móvil que posibilite la automatización de este procedimiento, lo que conllevaría a una disminución del tiempo del proceso manual. También podría suponer evitar las pérdidas asociadas.

1.2. Objetivos

Este trabajo de fin de grado pretende investigar, estudiar y desarrollar un algoritmo de procesado y análisis de la imagen, cuya finalidad es el desarrollo de una herramienta para la automatización del proceso de control del número de tablones de maderas pertenecientes a una misma imagen. Dicha investigación se centrará sobre todo en la calidad de la detección.

Este estudio desarrollará una aplicación MATLAB que permita observar los resultados finales del estudio utilizando para ello una interfaz simple.

Esta aplicación tendrá como entrada una imagen que será obtenida del sistema de archivos del dispositivo, además de diversos parámetros que serán los que permitan que la detección sea más o menos precisa. Por otro lado, devolverá como resultado una imagen, en la que se superpondrán rectángulos indicando la posición de los tablones detectados, así como el número de tablones detectados.

Otra de las finalidades de este estudio es servir como base para la creación de la herramienta móvil realizada por el resto de integrantes del grupo.

1.3. Metodología

La metodología seguida ha sido la siguiente:

- Se ha utilizado la herramienta web *Trello*, como muestra la figura 1.1, para una mejor planificación y control de las fases.
- Se han realizado estudios continuos mediante tablas y comparativas.
- Se han marcado fechas de finalización de cada fase y entregas internas de prototipos.
- Se han realizado planificaciones antes del inicio de cada fase para marcar los hitos y tareas a realizar.
- Se han realizado distintas reuniones con el tutor para supervisar el progreso de la aplicación.

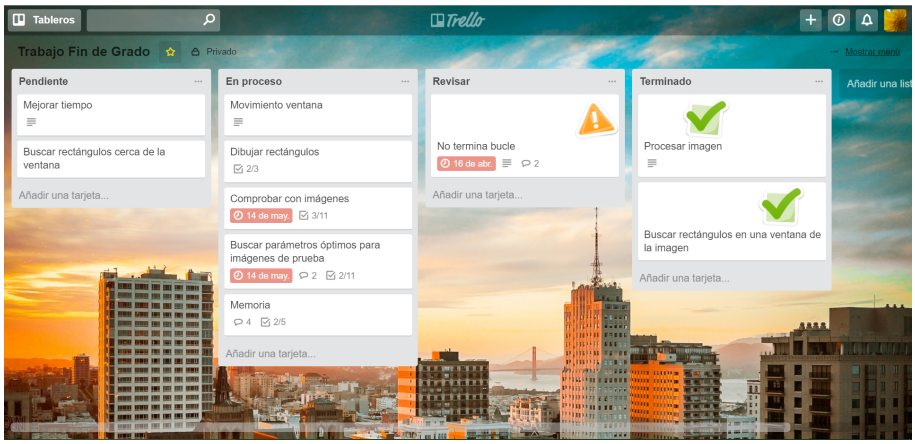


Figura 1.1: Planificación en la herramienta web *Trello*

1.4. Entorno tecnológico

A lo largo del desarrollo de este trabajo de fin de grado se han utilizado diversas tecnologías, como son MATLAB y LaTeX. Los recursos software utilizados han sido MATLAB, TeXstudio y MagicDraw. A continuación se detallan las características de cada uno de ellos.

1.4.1. MATLAB

MATLAB (**MAT***rix* **LAB***oratory*) es una herramienta software especializada en cálculo científico. Ofrece un *IDE*, entorno de desarrollo integrado, con su propio lenguaje de programación, el lenguaje M. Se trata de software propietario desarrollado por *MathWorks*. Tiene multitud de prestaciones, las más relevantes para este proyecto son sobre todo la manipulación de matrices y el procesado de la imagen.

Se ha escogido este software pues, como se ha mencionado anteriormente, facilita la manipulación de matrices e imágenes, ambas muy importantes en el desarrollo de este trabajo de fin de grado. Otra de las razones por las que se ha elegido MATLAB ha sido por que aunque es software propietario, la Universidad de Málaga ofrece licencias a sus alumnos.

Las versiones utilizada para el desarrollo de este proyecto han sido las versiones R2016a y R2016b. Inicialmente se empezó a desarrollar con la versión R2016a, pero para acceder a ciertas funcionalidades necesarias para la realización del proyecto fue necesaria la instalación de una versión más reciente, en concreto para poder acceder a ciertas características de App Designer.

App Designer

App designer es un entorno para crear interfaces de MATLAB de forma visual. Integra tanto el diseño de la interfaz como la programación del comportamiento del programa. Esta herramienta facilita y simplifica el proceso de creación de interfaces de usuario ofreciendo multitud de componentes para utilizar.

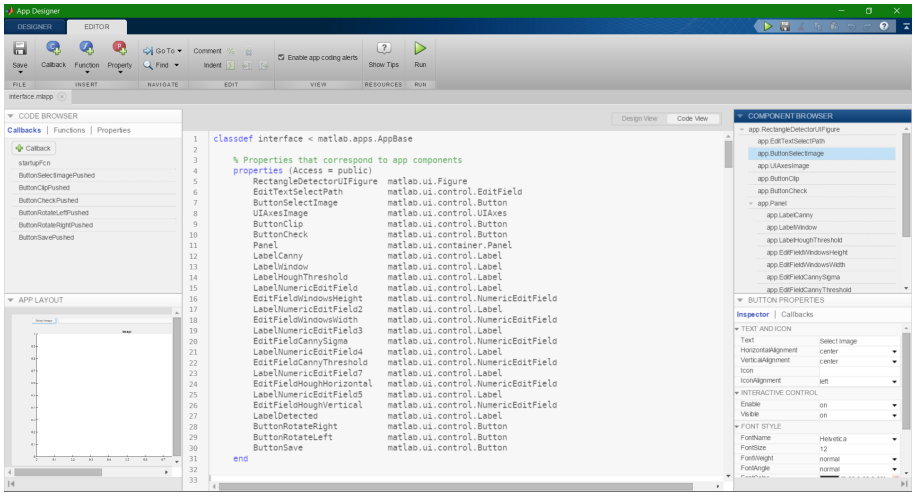


Figura 1.2: Interfaz de usuario de App Designer

1.4.2. MagicDraw

MagicDraw es una herramienta de modelado UML desarrollado por *No Magic, Inc*. Es una herramienta *CASE*, pues ayuda al ingeniero de software a desarrollar y mantener software. Con esta herramienta se pueden realizar multitud de diagramas, cuya finalidad es presentar diversas perspectivas del sistema para documentar el proyecto.

Esta herramienta ha sido escogida para la realización de los diagramas UML pues es la herramienta que se ha utilizado a lo largo de carrera en diversas asignaturas, y por tanto ya se conoce.

1.4.3. TeXstudio

TeXstudio es un entorno de desarrollo gratuito para LaTeX. Proporciona multitud de funcionalidades muy útiles a la hora de escribir textos en este lenguaje, como pueden ser auto completado de funciones, corrección ortográfica en el idioma seleccionado o resaltado de sintaxis.

Este software ha sido escogido para la realización de la documentación, pues además de las ventajas nombradas anteriormente es un editor de código abierto y multiplataforma.

2. Especificación y diseño

2.1. Análisis de Requisitos

El análisis de requisitos es la fase en la que se establece qué se espera que haga el sistema, y si para ello tiene alguna restricción. En esta sección se analizarán tanto los requisitos funcionales como los no funcionales.

2.1.1. Requisitos funcionales

Los requisitos funcionales expresan aquello que el sistema tiene que cumplir para producir un resultado útil.

RF1. Seleccionar imagen.

El usuario podrá seleccionar una imagen del gestor de archivos del sistema en el que se encuentre.

RF2. Recortar imagen.

El usuario podrá recortar una imagen.

RF3. Girar imagen

El usuario podrá girar una imagen.

RF4. Guardar imagen

El usuario podrá guardar una imagen.

RF5. Completar parámetros

El usuario podrá completar los parámetros de detección.

RF6. Buscar rectángulos

El usuario podrá buscar rectángulos en la imagen.

RF7. Mostrar imagen seleccionada

El sistema mostrará la imagen seleccionada por el usuario.

RF8. Mostrar imagen detectada

El sistema mostrará la imagen con los rectángulos superpuestos.

RF9. Mostrar número de rectángulos detectados

El sistema mostrará el número de rectángulos detectados.

2.1.2. Requisitos no funcionales

Los requisitos no funcionales son aquellos que presentan restricciones y características generales del sistema.

- **RNF1.** El sistema se asegurará que los parámetros de detección introducidos por el usuario son válidos.
- **RNF2.** La aplicación funcionará en un sistema que tenga instalado MATLAB R2016b o posterior.

2.2. Diseño

2.2.1. Fases del proyecto

Este proyecto se ha dividido en las siguientes fases:

- **Análisis de requisitos:** se han analizado las funcionalidades que iba a requerir el sistema, generando un documento de requisitos.
- **Búsqueda de información:** se ha realizado una tarea de investigación de las diferentes tecnologías utilizadas, así como los distintos algoritmos de procesamiento de la imagen para la detección de formas geométricas.
- **Preparación del entorno de trabajo:** se han descargado e instalado las herramientas software necesarias.
- **Diseño de la aplicación:** se han diseñado y modelado los distintos diagramas UML necesarios para la comprensión del sistema.
- **Diseño de la interfaz:** se ha diseñado la interfaz del sistema y la integración entre vistas.
- **Investigación y desarrollo en MATLAB:** se ha implementado una aplicación MATLAB, además de realizar múltiples comparativas sobre la capacidad de detección del algoritmo.
- **Fase de pruebas:** se han probado que los resultados en cada fase sean los esperados.

2.2.2. Diagrama de comunicación

En este apartado se presenta el diagrama de comunicación, que muestra las interacciones entre las distintas funciones del sistema.

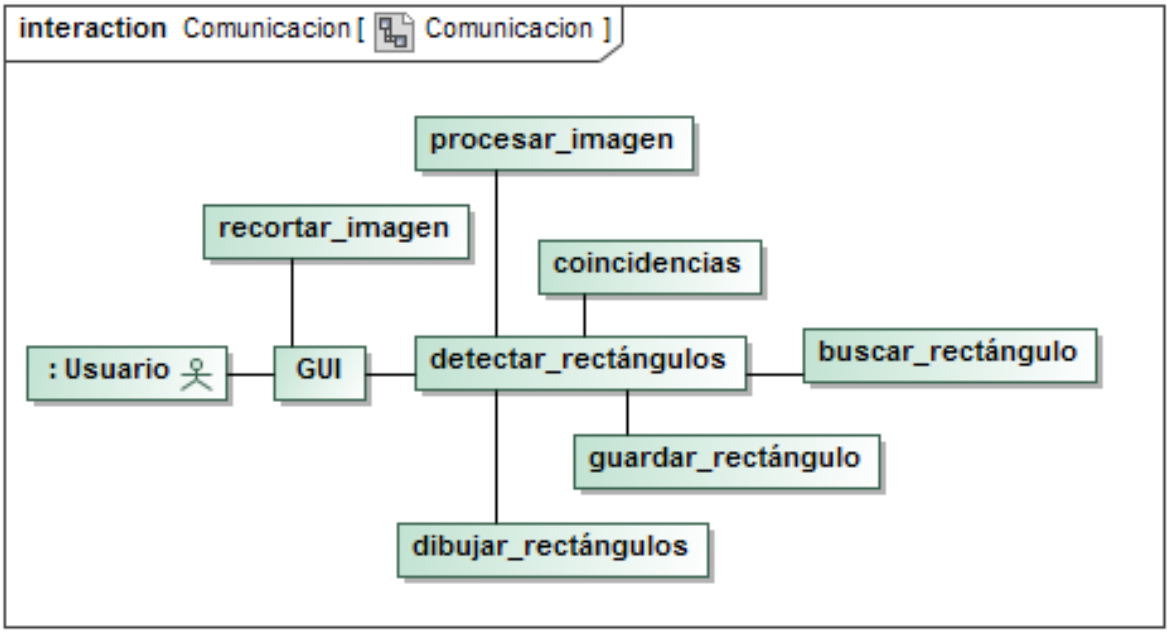


Figura 2.1: Diagrama de comunicación de la aplicación

Como puede observarse en la figura 2.1, el usuario interacciona con la interfaz gráfica de usuario, y esta se comunica con las funciones *recortar_imagen* y *detectar_rectángulos*. La función *recortar_imagen* se encargará de recortar la imagen de entrada y devolver dicho recorte. La función *detectar_rectángulos* detectará y dibujará rectángulos en la imagen pasada como argumento.

Esta última función se comunicará con ciertas funciones auxiliares para cumplir su tarea. Estas funciones auxiliares tendrán tareas independientes entre sí, serán: *procesar_imagen*, que se encargará de preparar la imagen para pasarla por el algoritmo; *coincidencias*, que comprobará si ya hay un rectángulo en el segmento de la imagen que se está analizando; *buscar_rectángulo*, que se encargará de buscar si hay un rectángulo en el segmento de la imagen a analizar; *guardar_rectángulo*, que almacenará el rectángulo en caso de que haya sido detectado; y *dibujar_rectángulos*, que dibujará todos los rectángulos detectados anteriormente sobre la imagen.

2.2.3. Diagrama de secuencia

En este apartado se presenta el diagrama de secuencia, que muestra la interacción entre las distintas funciones del sistema.

En la figura 2.2 se observan la secuencia de acciones entre el usuario y la aplicación, y las llamadas entre las diferentes funciones.

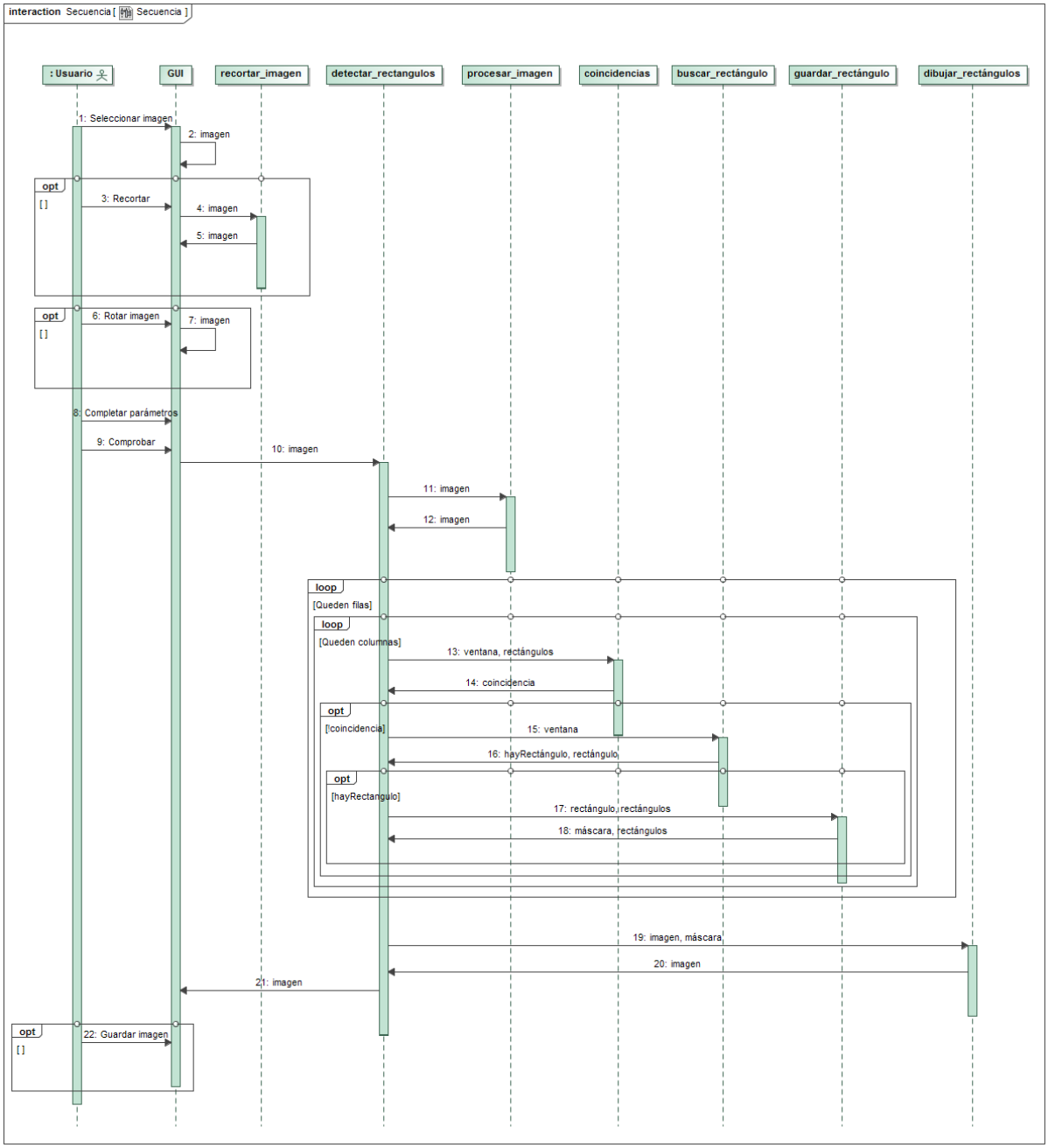


Figura 2.2: Diagrama de secuencia de la aplicación

Inicialmente, el usuario deberá seleccionar una imagen. Una vez seleccionada la imagen, podrá recortar o rotar la imagen. Para recortarla, la interfaz se comunica con la función *recortar_imagen*.

Posteriormente, el usuario completará los parámetros de la detección y comprobará si hay rectángulos en la imagen. Para la comprobación de los rectángulos la interfaz se comunica con la función *detectar_rectángulos*, que será la que realice la detección de rectángulos. Esta función se comunicará con la función *procesar_imagen* para la preparación previa de la imagen.

Una vez haya recibido la imagen procesada, se desplazará una ventana deslizante por toda la imagen. Para cada una de estas ventanas deslizantes, se comprobará mediante la función *coincidencias* si hay algún rectángulo lo suficientemente cercano a la ventana, como para que no quepa otro. En caso de no haberse encontrado rectángulos cercanos, se analiza la ventana buscando si hay algún rectángulo con la función *buscar_rectángulo*, en cuyo caso se almacena el nuevo rectángulo con la función *guardar_rectángulo*.

Una vez realizada la búsqueda anterior por toda la imagen, la función *detectar_rectángulos* realiza una llamada a la función *dibujar_rectángulos* para dibujar los rectángulos sobre la imagen, y esta imagen será devuelta al usuario. Finalmente, el usuario podrá guardar la imagen con los rectángulos detectados.

2.2.4. Diagrama de interfaz

La figura 2.3 muestra el diagrama de interfaz de la aplicación. Esta aplicación tendrá una serie de botones, que permitirán cargar una imagen del sistema, recortarla, o buscar rectángulos en dicha imagen. Además, tendrá una serie de campos de texto dónde el usuario podrá introducir los valores de los parámetros de la función. También dispondrá de una zona dónde se mostrará la imagen cargada por el usuario, y posteriormente con los rectángulos detectados, y una etiqueta con el número de rectángulos detectados.

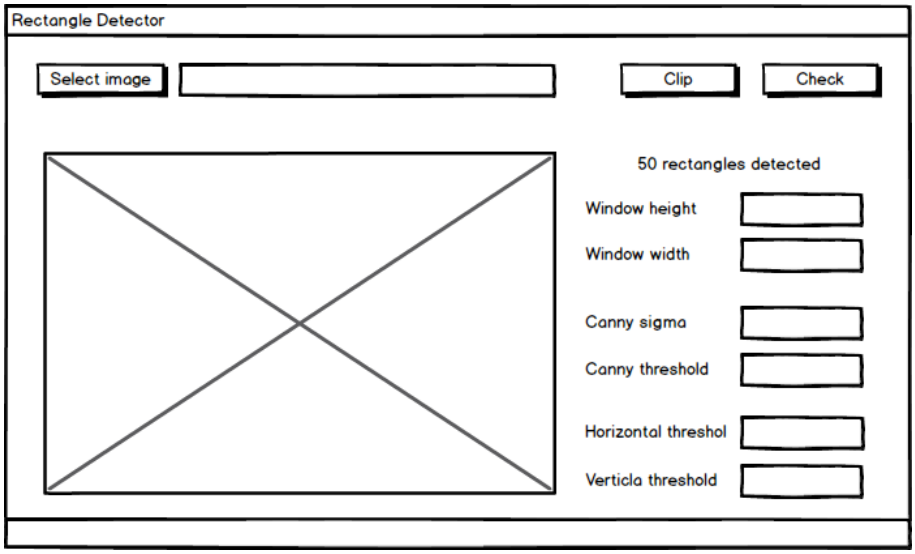


Figura 2.3: Diagrama de interfaz de la aplicación

2.3. Casos de uso

A partir de los requisitos mencionados anteriormente, se han extraído los siguientes casos de uso, los cuáles serán detallados posteriormente.

- **CU1:** Seleccionar imagen.
- **CU2:** Recortar imagen.
- **CU3:** Girar imagen.
- **CU4:** Guardar imagen.
- **CU5:** Modificar parámetros.
- **CU6:** Buscar rectángulos en la imagen.

Cómo se observa en la figura 2.4, el usuario podrá realizar una serie de acciones en la aplicación, sin embargo, para realizar algunas acciones debe de haber realizado otras con anterioridad. El usuario podrá recortar una imagen, girarla, modificar sus parámetros o buscar rectángulos en dicha imagen, siempre que previamente haya seleccionado un imagen. Para guardar una imagen será necesario que previamente haya buscado rectángulos en la imagen.

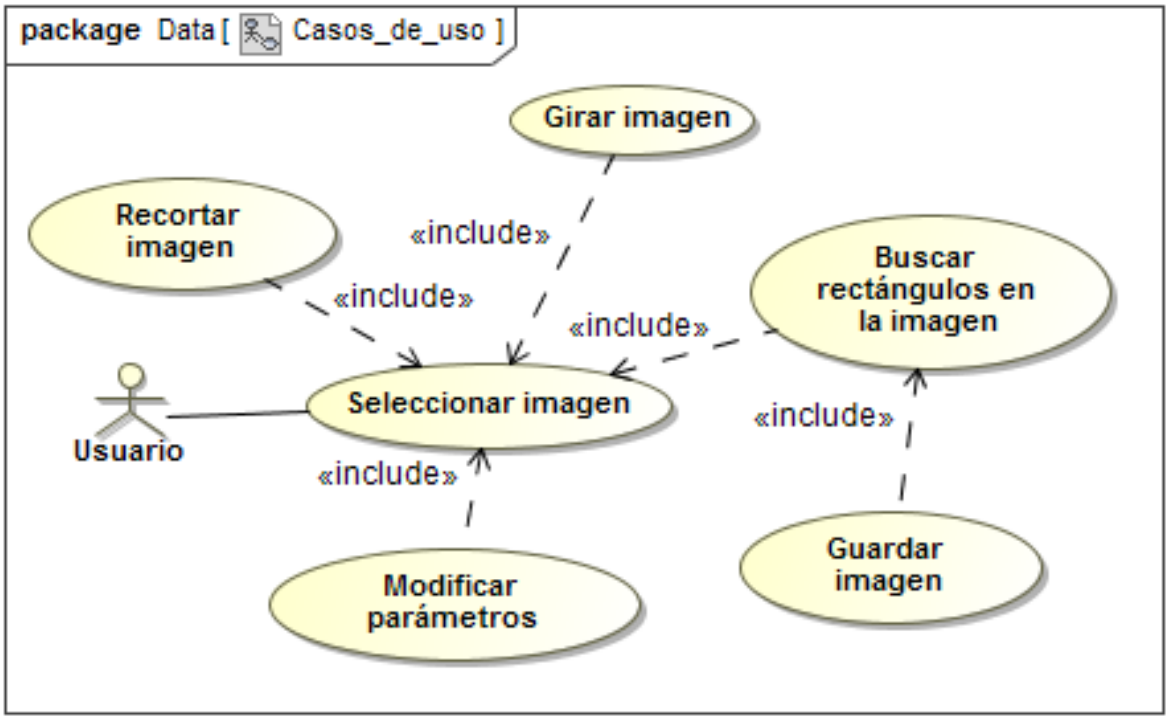


Figura 2.4: Diagrama de casos de uso de la aplicación

3. Implementación

En este capítulo se detallará el funcionamiento del sistema, tanto las funciones implementadas finalmente, como todo aquello que se ha investigado, pero que no se ha utilizado finalmente para este proyecto.

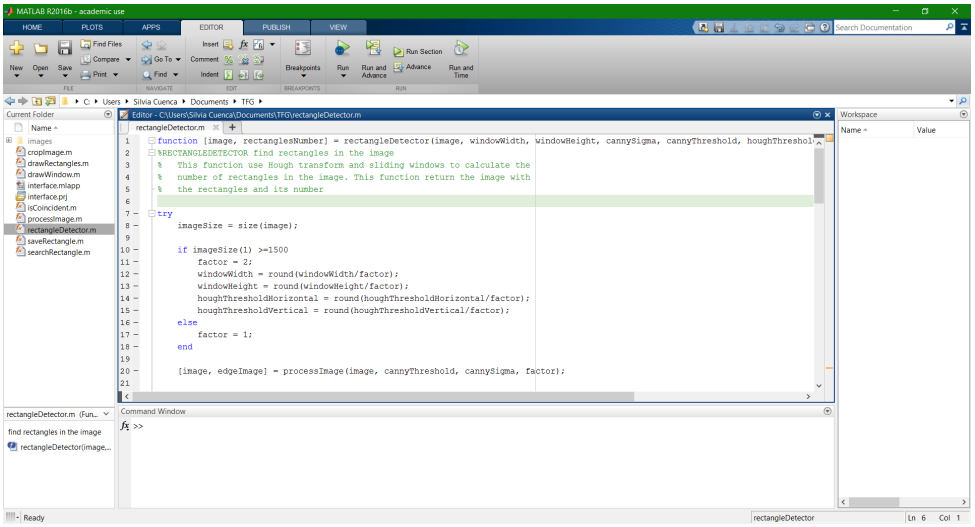


Figura 3.1: Espacio de trabajo de *MATLAB*

3.1. Investigación

Para el desarrollo de este proyecto se han utilizado diversos algoritmos que serán analizados en esta sección. La investigación ha sido dividida en diversas secciones para su mejor comprensión.

3.1.1. Procesado de la imagen

La primera parte de este proyecto ha consistido en procesar la imagen para poder ser analizada.

Segmentación por color

El primer paso para el procesado de la imagen ha consistido en realizar la segmentación por color. Esta segmentación se ha hecho atendiendo a los diferentes canales en un espacio de

color. MATLAB permite la segmentación por color de una forma fácil mediante una pequeña aplicación llamada *Color Thresholder*, la cual incluye los espacios de color mostrados en la figura 3.2. Estos espacios de color se encuentran representados en la figura 3.3.

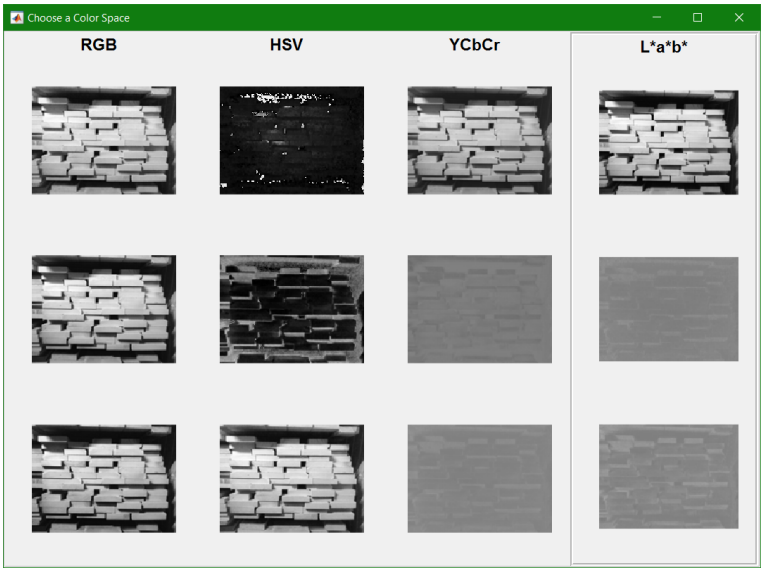


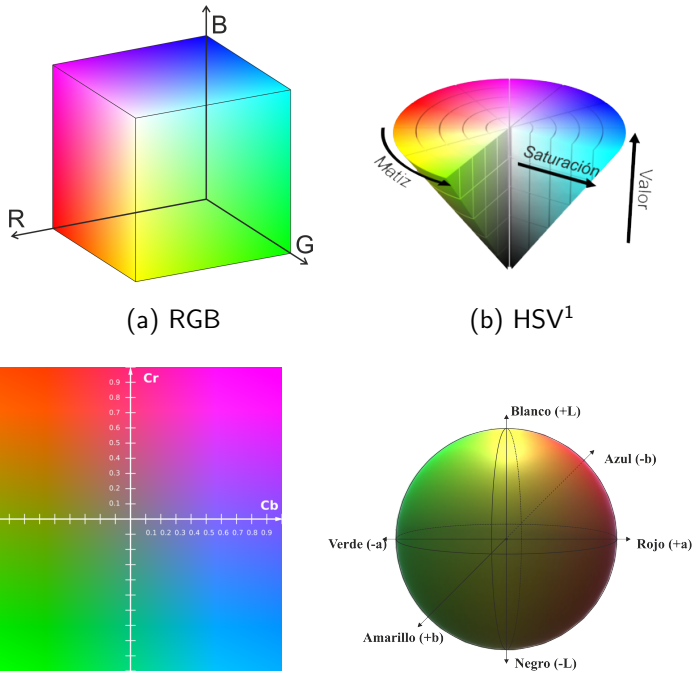
Figura 3.2: Espacios de color disponibles en MATLAB

- **Espacio de color RGB.** Una imagen RGB está formada por tres matrices del tamaño de la imagen, cada una de estas matrices representa cada uno de los componentes de color de la imagen. La primera matriz identifica a la componente roja, la segunda a la componente verde y la tercera a la componente azul. Para definir un color en este espacio de color es necesario especificar su valor de intensidad para cada una de los tres componentes. Figura 3.3a.

- **Espacio de color HSV.** Es un espacio de color definido por tres componentes: matiz, saturación y valor. Su nombre proviene del inglés *Hue*, *Saturation* y *Value*. La componente *matiz* está representada por un ángulo, el color rojo se encuentra en 0°, el verde en 120° y el azul en 240°. La componente *saturación* indica como de puro es el color, va desde el color blanco hasta el color totalmente saturado. La componente *valor* indica la altura en el eje blanco-negro. Figura 3.3b.

- **Espacio de color YCbCr.** Este espacio de color está definido por tres componentes. La componente Y indica la luminancia, que es la imagen en escala de grises. Las componentes Cb y Cr indican el tono. Cb indica el tono entre el azul y el amarillo, y Cr indica el tono entre el rojo y el verde. Figura 3.3c.

- **Espacio de color L*a*b.** Es un espacio de color definido por tres componentes. La componente L indica lo luminosa u oscura que es una imagen, la componente a indica la diferencia entre rojo y verde, y la componente b indica la diferencia entre amarillo y azul. Figura 3.3d.



(a) RGB

(b) HSV[1]

(c) Plano Cb-Cr con luminancia Y=0.5[2]

(d) L*a*b

Figura 3.3: Representación de los espacios de color

La imagen ha sido analizada en los espacios de color *HSV* y *L*a*b*, pues han sido considerados los más adecuados para este proyecto.

Para la segmentación *HSV* se observa en la figura 3.4 que en la imagen a color apenas se distinguen la imagen original (figura 3.4a) y la imagen segmentada en el canal *V* (figura 3.4c). Sin embargo, en las imágenes de bordes (figuras 3.4b y 3.4d) se aprecian diferencias, que aunque sutiles, afectan de forma negativa al algoritmo de búsqueda de rectángulos. Esto es así pues al tener la imagen segmentada, hay ciertas zonas que aparecen completamente en negro, lo que hace que los bordes con esas zonas sean más fuertes,y por tanto el algoritmo de búsqueda de bordes elegido detecta como bordes lo que en la imagen original no sería un borde, empeorando así el resultado del algoritmo de búsqueda de rectángulos.

Igualmente, en la figura 3.5, ocurre algo similar con la segmentación *L*a*b*. La imagen a color segmentada (figura 3.5a) se parece bastante a la original, pero en esta se aprecian más diferencias que respecto a la segmentación *HSV*. Sin embargo, en la imagen de bordes segmentada (figura 3.5b) se aprecian bastantes diferencias que afectan negativamente al algoritmo de búsqueda de rectángulos, pues al igual que en el caso anterior, la detección de bordes detecta como borde las zonas segmentadas aún cuando estas no son bordes realmente.

Finalmente, la imagen no ha sido segmentada por color, pues se comprobó que esto pro-

[1]https://es.wikipedia.org/wiki/Modelo_de_color_HSV#/media/File:Cono_de_la_coloraci%C3%B3n_HSV.png
[2]https://en.wikipedia.org/wiki/YCbCr#/media/File:YCbCr-CbCr_Scaled_Y50.png

(a) Imagen original

(b) Imagen de bordes sin segmentar por color

(c) Imagen segmentada en el canal V

(d) Imagen de bordes segmentada en el canal V

Figura 3.4: Segmentación de la imagen en espacio de color HSV

(a) Imagen segmentada

(b) Imagen de bordes segmentada

Figura 3.5: Segmentación de la imagen en espacio de color L*a*b

vocaba demasiado fallos en la búsqueda de rectángulos posterior.

Búsqueda de bordes

El siguiente paso en el procesado de la imagen ha consistido en la búsqueda de bordes, estos se definen como fuertes variaciones de intensidad, los cuales corresponden a las fronteras entre dos regiones. Para poder realizar esta búsqueda ha sido necesario pasar la imagen a escala de grises. Una vez que la imagen se encontraba en escala de grises, ha sido analizada con diversos algoritmos para comprobar las diferencias entre sí. Los algoritmos que han sido probados se detallan a continuación.

- **Operadores basados en la primera derivada**
  Los operadores basados en la primera derivada se basan en el gradiente, el cual mide la rapidez en que los valores de los píxeles cambian. El gradiente es un vector bidimensional que apunta a la dirección de tasa máxima de cambio. La fórmula del gradiente es la siguiente:

$$\nabla f(x,y) = \begin{bmatrix} \frac{\partial}{\partial x} f(x,y) \\ \frac{\partial}{\partial y} f(x,y) \end{bmatrix} = \begin{bmatrix} f_x(x,y) \\ f_y(x,y) \end{bmatrix}$$

  El módulo del gradiente es proporcional a la intensidad de la variación máxima de $f(x,y)$:

$$|\nabla f(x,y)| = \sqrt{(f_x(x,y))^2 + (f_y(x,y))^2} \approx |f_x(x,y)| + |f_y(x,y)|$$

  En las imágenes en escala de grises, los operadores basados en la primera derivada se basan en la diferencia entre los niveles de gris. Se implementan mediante la convolución de la imagen con máscaras que detectan cambios horizontales y cambios verticales. Para obtener la imagen de bordes completa es necesario sumar el resultado de estas convoluciones de forma absoluta y obtener aquellos píxeles que superan un umbral. Estas máscara son filtros de paso alto, pues son los que refuerzan los contrastes de la imagen. En las máscaras de paso alto la suma de sus valores es 0. El tamaño de esta máscara influye en la calidad de la detección: una máscara *pequeña* aporta una localización más precisa, pero se ve más afectado por el ruido; sin embargo, una máscara *grande* aporta una localización más precisa, es más robusta al ruido, pero requiere un coste computacional mayor.

  - **Algoritmo de Sobel** (Figura 3.10b)
    El algoritmo de Sobel se basa en el método mencionado en el párrafo anterior:

$$imagen\ de\ bordes = |f(x,y) \otimes H_h(x,y)| + |f(x,y) \otimes H_v(x,y)|$$

dónde $f(x, y)$ es la imagen, $H_h(x, y)$ y $H_v(x, y)$ las máscaras para la detección de líneas horizontales y verticales respectivamente y $\otimes$ es el operador convolución.

Las máscaras del algoritmo de Sobel son las siguientes siguientes:

$$Máscara\ de\ Sobel\ horizontal: \frac{1}{4}\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

$$Máscara\ de\ Sobel\ vertical: \frac{1}{4}\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

- **Operador DroG (Derivada de la Gaussiana)**
  Se considera ruido de una imagen a la variación aleatoria del color o el brillo en una imagen. Esto dificulta la detección de bordes en los filtros basados en la primera derivada. La solución a esto es suavizar la imagen antes de realizar la detección de bordes.

  El algoritmo DroG, *Derivative of the Gaussian*, combina suavizado y gradiente en una única operación. Se basa en aplicar el gradiente a la convolución de la imagen con una función de suavizado (Figura 3.6).

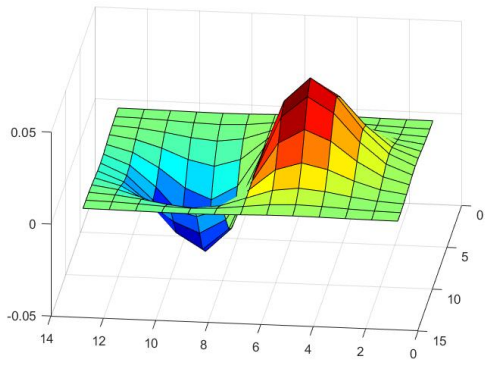


Figura 3.6: Máscara del operador *DroG*

Este operador no ha sido utilizado directamente para la búsqueda de bordes de la imagen, pero se ha decidido incluir aquí su descripción pues se utilizará junto con otro algoritmo de búsqueda de bordes.

- **Operadores basados en la segunda derivada** Los operadores basados en la segunda derivada son muy sensibles al ruido, por ello suele combinarse su aplicación con un filtro de paso bajo para suavizar el ruido. La ventaja de estos operadores es que son independientes de la orientación del borde.

  - **Operador Laplaciano**
    Este operador ofrece una detección precisa y además es independiente de la orientación del borde, sin embargo es muy sensible al ruido, por lo que en general aporta multitud de falsos positivos. Este operador aprovecha los cruces por cero de la segunda derivada para encontrar los bordes de los objetos. Puede ser definido como la convolución de la imagen con la máscara siguiente:

    $$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

    Como puede observarse, esta máscara es la suma de una máscara de bordes verticales con una máscara de bordes horizontales:

    $$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & -2 & 1 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 0 \\ 0 & -2 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

    La búsqueda de bordes mediante el operador laplaciano no suele utilizarse, pues es muy sensible al ruido y suele producir bordes dobles.

    Al igual que el operador *DroG*, este operador no ha sido utilizado directamente para la búsqueda de bordes de la imagen, pero se ha decidido incluir aquí su descripción pues se utilizará junto con otro algoritmo de búsqueda de bordes.

  - **Operador LoG (Laplaciano de la Gaussiana)** (Figura 3.10c)
    Este operador combina suavizado con el operador laplaciano: primero se aplica el suavizado y después el operador laplaciano.

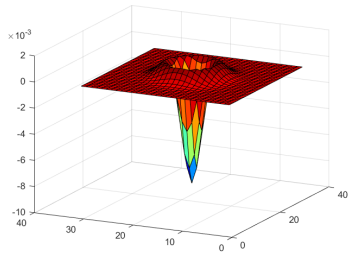


Figura 3.7: Máscara del operador *LoG*

Sin embargo, este algoritmo presenta algunos inconvenientes, como son: alto coste computacional, no proporciona la orientación del borde, la salida contiene valores

negativos y no enteros por lo que es necesaria su normalización y tiende a redondear las esquinas de los objetos (en función del σ aplicado).

Para aplicarlo, primero se ejecuta la convolución de la segunda derivada de la imagen con la gaussiana, o la convolución de la imagen con la segunda derivada de la gaussiana.

- **Algoritmo de Canny** (Figura 3.10d)
  Este método de detección de bordes está basado en tres criterios:

  - El criterio de *detección* establece que este método evite los falsos positivos y negativos.
  - El criterio de *localización* establece que la diferencia entre la posición del borde detectado y la posición del borde real sea mínima.
  - El criterio de *respuesta única* hace que se integren en un único borde las respuestas múltiples.

  Este algoritmo consiste en una serie de pasos:

  1. **Aplicación del operador DroG:** el primer paso es suavizar la imagen con un filtro gaussiando, calcular la direción y el módulo del gradiente estimar las magnitudes del gradiente y las direcciones de los bordes locales para cada píxel, y determinar los máximos locales de las magnitudes del gradiente.
  2. **Supresión de non-máxima:** el siguiente paso es aproximar la dirección del gradiente a una de las cuatro direcciones principales de una imagen (horizontal, vertical y las dos diagonales principales). Con esta dirección se calculan tres puntos a lo largo de esta, y se toma el máximo. La supresión de non-máxima hace que este algoritmo cumpla con el *criterio de respuesta única*. En la imagen 3.8 se muestran en azul las cuatro direcciones de la imagen, y en rojo un gradiente y su aproximación.

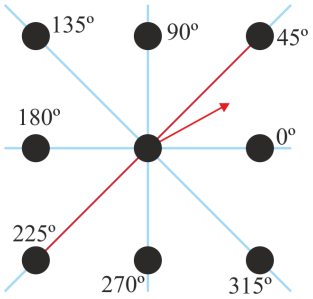


Figura 3.8: Representación de las cuatro direcciones principales de la imagen, y aproximación a una de estas direcciones

  3. **Umbralización mediante histéresis:** se tienen dos umbrales, un umbral superior y un umbral inferior. Los píxeles que superan el valor del umbral máximo son considerados bordes fuertes, los que encuentran por debajo del umbral inferior no son

considerados bordes, y los que se encuentran entre ambos umbrales son considerados bordes débiles. Serán considerados como bordes finales todos aquellos bordes fuertes, y bordes débiles que estén unidos a bordes fuertes.

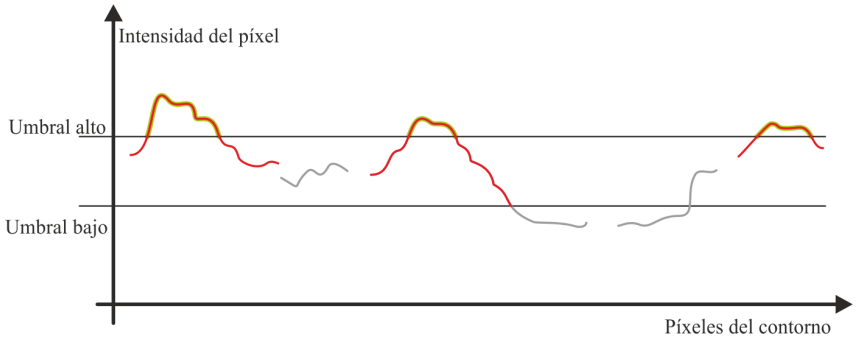
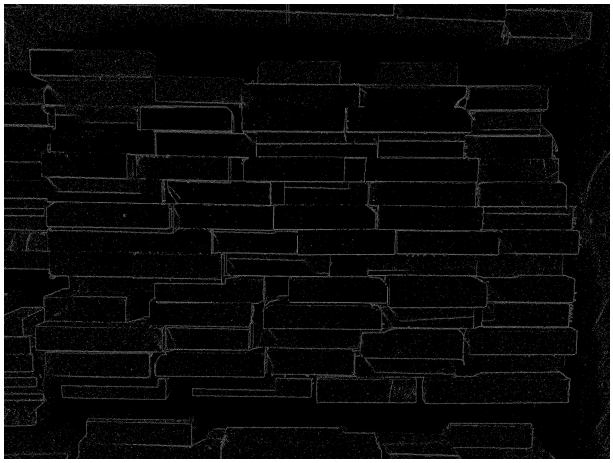


Figura 3.9: Intensidad de los píxeles del contorno de una imagen y los umbrales alto y bajo

(a) Imagen original



(b) Algoritmo de Sobel con threshold=0.045

(c) Operador LoG con threshold=0.005

(d) Algoritmo de Canny con threshold=0.08 y sigma=3

Figura 3.10: Detección de bordes mediante los diferentes algoritmos descritos anteriormente

El algoritmo utilizado finalmente en este proyecto ha sido el algoritmo de Canny pues es el que mayor precisión y menos falsos positivos y negativos aporta. Este algoritmo puede ser más lento que otros métodos, como el método de Sobel, pero al ser sus resultados más precisos agiliza el resto del proceso, lo cual es preferible, pues los bordes únicamente van a ser buscados una vez, mientras que sobre este resultado se realizarán múltitud de búsquedas de líneas.

3.1.2. Búsqueda de rectángulos en la imagen

Una vez obtenida la imagen de bordes, el siguiente paso ha sido buscar rectángulos en la imagen. Primero se ha intentado buscar rectángulos con los operadores morfológicos, después realizar segmentación por regiones, y finalmente se ha optado por el algoritmo de Hough.

Operadores morfológicos

Los operadores morfológicos permiten mejorar los resultados de una imagen segmentada, en este caso una imagen de bordes. Estos operadores permiten transformar el contorno de los objetos, eliminar objetos pequeños, rellenar huecos, identificar objetos e identificar objetos.

Estos operadores se basan en la imagen y un elemento estructurante, que será el que defina la forma del nuevo objeto. En este proyecto se analizarán los siguientes operadores:

- **Operador dilatación.** (Figura 3.11c)
  Este operador permite agrandar objetos de una imagen, por tanto se rellenan los pequeños huecos de la imagen, pero también se ensancha el contorno de los objetos.

$$[f \oplus h](i,j) = \begin{cases} 1 & si\ f * h(i,j) \geq 1 \\ 0 & si\ f * h(i,j) = 0 \end{cases}$$

  dónde f es la imagen y h es el elemento estructurante formado por 0's y 1's.

- **Operador erosión.** (Figura 3.11d)
  Este operador permite erosionar el contorno de los objetos, y por tanto eliminar pequeñas líneas y objetos de la imagen.

$$[f \ominus h](i,j) = \begin{cases} 1 & si\ f * h(i,j) = t \\ 0 & si\ f * h(i,j) < t \end{cases}$$

  dónde f es la imagen, h es el elemento estructurante formado por 0's y 1's y t es el número de 1's de h.

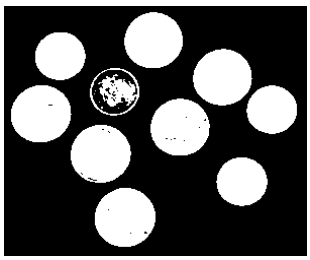
- **Operador cierre.** (Figura 3.11e)
  Este operador combina los operadores de dilatación y erosión para cerrar pequeños

huecos en los objetos. En este proyecto se ha analizado su funcionamiento para cerrar los contornos de los objetos después de la búsqueda de bordes.

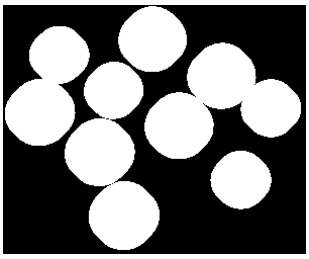
$$[f \bullet h](i,j) = [(f \oplus h) \ominus h](i,j)$$

dónde f es la imagen y h es el elemento estructurante formado por 0's y 1's.

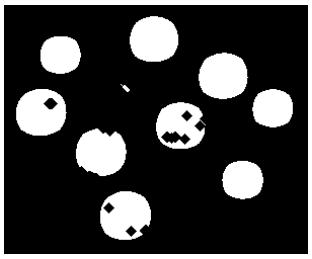
(a) Imagen original[3]



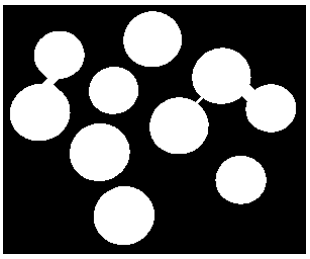
(b) Imagen binarizada



(c) Imagen con operador dilatación



(d) Imagen con operador erosión



(e) Imagen con operador cierre

Figura 3.11: Ejemplo de operadores morfológicos de dilatación, erosión y cierre

- **Operador acierto y fallo.** (Figura 3.12)
  Este operador permite identificar objetos con una forma dada. Este operador se puede expresar como combinación de los operadores erosión y dilatación.

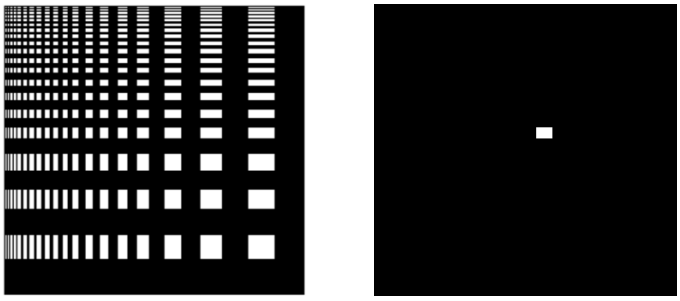
$$[f \otimes h](i,j) = (f \ominus h_1) \cap (f \ominus h_2)$$

dónde f es la imagen y h es el elemento estructurante formado por 0's y 1's.

Estos operadores finalmente no han sido utilizados para la búsqueda de rectángulos, pues al aplicar el *operador cierre* a los rectángulos detectados por el algoritmo de Canny, no solo cerraba el contorno de los objetos, sino que también unía las zonas dónde más ruido de la imagen había, y por tanto provocaba una mayor tasa de error en el algoritmo de detección de rectángulos.

El *operador de acierto y fallo* no ha sido utilizado, pues para ellos era necesario conocer el tamaño de los rectángulos con demasiada precisión. Otro inconveniente de este operador

[3] https://informatica.cv.uma.es/
[4] https://informatica.cv.uma.es/



(a) Imagen original[4]

(b) Objeto encontrado en la imagen tras aplicar el operador de acierto y fallo

Figura 3.12: Ejemplo de operador morfológico de acierto y fallo

ha sido que sería necesario aplicarlo una vez por cada tamaño y forma de rectángulo que se quisiera detectar, por tanto su uso quedó descartado.

Segmentación por regiones

Las técnicas de segmentación por regiones consisten en dividir una imagen en regiones agrupando los píxeles con características similares.

- **Crecimiento de regiones.**
  Este procedimiento consiste en ir formando regiones de píxeles según algún criterio. Consiste en tomar una serie de semillas que representan las regiones de la imagen (cada región debe tener al menos una semilla). Posteriormente se van analizando los vecinos de la región, y si cumplen el criterio de adhesión establecido, se unen a dicha región.

  Este método de segmentación presenta sin embargo ciertos problemas, como la elección de las semillas y sus posiciones, parte fundamental de este algoritmo, y la elección del criterio para la adhesión de un píxel a una región.

- **Algoritmo k-medias.**
  Este algoritmo es similar a crecimiento de regiones, sin embargo, en cada iteración del algoritmo se clasifican todos los píxeles de la imagen, no únicamente los vecinos (como pasaba en crecimiento de regiones), se calcula el píxel representativo de la región y se sigue ejecutando mientras los píxeles sigan cambiando de región.

Finalmente, no ha sido implementado ningún algoritmo de segmentación por regiones, pues para ello era necesario saber cuantas regiones había, además de posicionar las semillas.

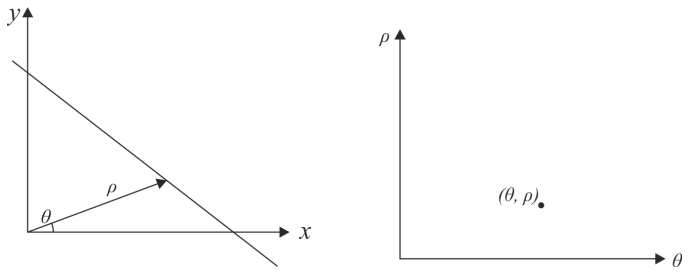
Transformada de Hough

La transformada de Hough es un método utilizado para la detección de formas en imágenes. Con este método pueden encontrarse rectas, circunferencias y elipses. Este proyecto está centrado en la detección de líneas, por tanto no se detallará el proceso de detección de circunferencias y elipses.

Hough propuso un método para la detección de rectas en una imagen. Este método se basa en la ecuación normal de una recta, cuya expresión es:

$$x\cos\theta + y\sin\theta = \rho$$

dónde ρ es la distancia de la recta al origen de coordenadas, y θ es el ángulo que forma la recta perpendicular a la recta que se quiere calcular, pasando por el eje de abcisas. $\theta \in [-\pi, \pi]$ y $\rho > 0$. (Figura 3.13)



(a) Línea en coordenadas XY (b) Línea en coordenadas polares

Figura 3.13: Representación de una línea en diferentes sistemas de coordenadas

En el caso de una imagen, x e y se corresponden con las coordenadas de un píxel. Al pasar a polar esta ecuación, se obtiene la relación entre los valores θ y r de todas las rectas que pasan por el punto (x, y), cuya representación gráfica es una curva sinusoidal. La *transformada de Hough* es conocida como la transformación punto a curva, que pasa de cada punto en el plano (X,Y) a una curva en el plano (θ, ρ). Todos los puntos pertenecientes a una misma recta en el plano (X,Y), se cortan en el mismo punto en el plano (θ, ρ).

El siguiente paso es definir un acumulador para cada par (θ, ρ), de forma que para cada punto (x, y) de la imagen se comprueban los posibles valores de θ, se calcula ρ, y si este valor es positivo, se incrementa el acumulador de ese punto.

Una vez realizado lo anterior sobre todos los puntos de la imagen, se obtienen los máximos locales que superan un umbral, los cuales corresponden a rectas en la imagen.

Las ventajas de la detección de líneas mediante la transformada de Hough son su insensibilidad al ruido, a segmentos que faltan en la línea, y a otros objetos no lineales en la imagen.

El principal problema que se ha encontrado al utilizar la *transformada de Hough* en este proyecto ha sido el emparejamiento de líneas, pues después de analizar la imagen y filtrar aquellas líneas verticales y horizontales consideradas válidas, no se podía saber que par de paralelas horizontales y verticales correspondían a cada rectángulo, tal y como se muestra en la figura 3.15.

La solución a esto ha sido el analizar segmentos de la imagen en vez de la imagen entera: sobre la imagen se desplaza una ventana deslizante, y a esta ventana será a la que se le

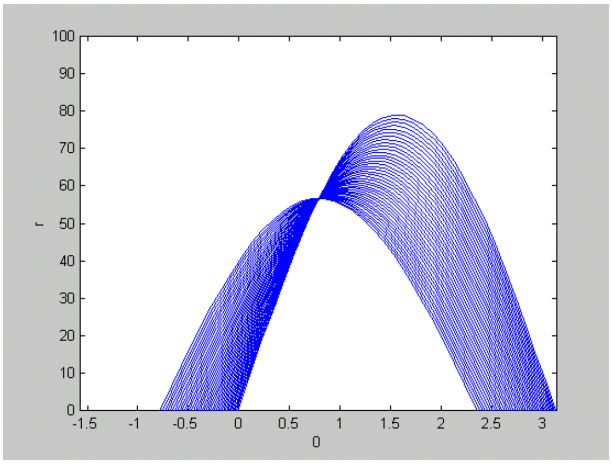


Figura 3.14: Puntos pertenecientes a una misma recta[5]

aplique la transformada de Hough. Dicha ventana deberá tener el tamaño suficiente para que no quepan dos rectángulos en ella, por tanto sabiendo que en cada ventana hay a lo sumo un rectángulo, es fácil realizar la correspondencia de líneas para obtener las cuatro rectas que forman un rectángulo (Figura 3.16).

3.1.3. Mejoras del algoritmo

La ejecución del algoritmo mencionado anteriormente (búsqueda de bordes con el algoritmo de Canny, y posterior búsqueda de rectángulos con el algoritmo de Hough) era demasiado lenta, pues había que realizar multitud de operaciones para cada imagen. Este algoritmo ha sido mejorado de forma que se ha conseguido reducir notablemente el tiempo de ejecución, pues para una misma imagen y unos mismos parámetros este se ha reducido de aproximadamente de unos 120 sg a unos 3 sg.

Las funcionalidades más importantes implementadas para mejorar el algoritmo serán explicadas a continuación.

Búsqueda de rectángulos cercanos

Una de las razones por las que este algoritmo tardaba en ejecutarse ha sido la ejecución de la búsqueda de rectángulos en cada ventana de la imagen. Para solucionar esto, se ha intentado ejecutar esta búsqueda en el menor número posible de ventanas de la imagen.

Para ello, se ha implementado un mecanismo de salto cuando se detecta un rectángulo, de forma que si en la ejecución de la ventana actual se ha encontrado un rectángulo, no se analiza la siguiente ventana, sino que se calcula dónde termina el rectángulo detectado y se continúan analizando ventanas de la imagen desde ese punto.

[5]https://informatica.cv.uma.es/

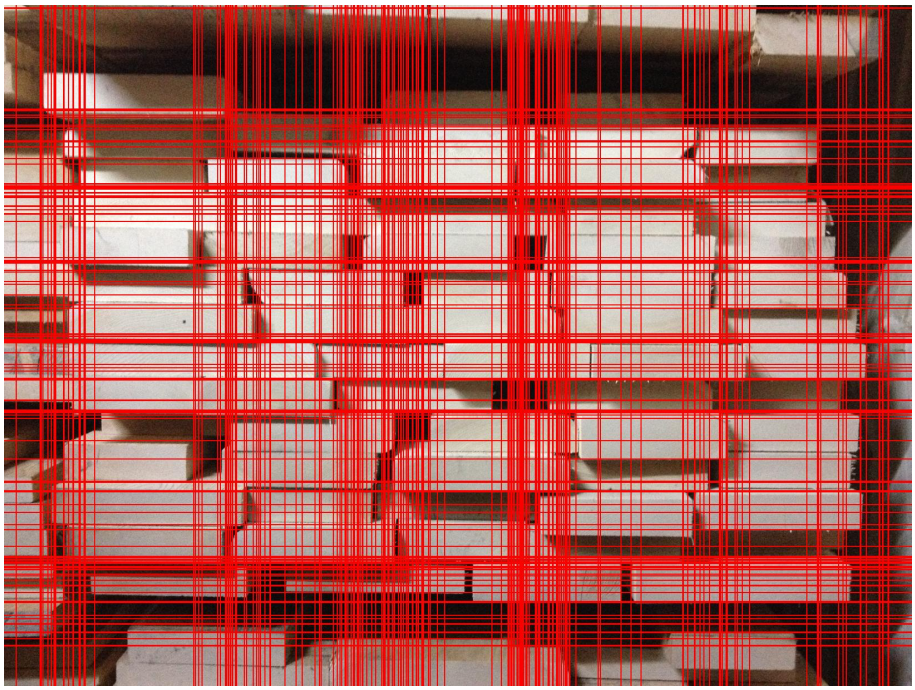


Figura 3.15: Imagen con líneas detectadas

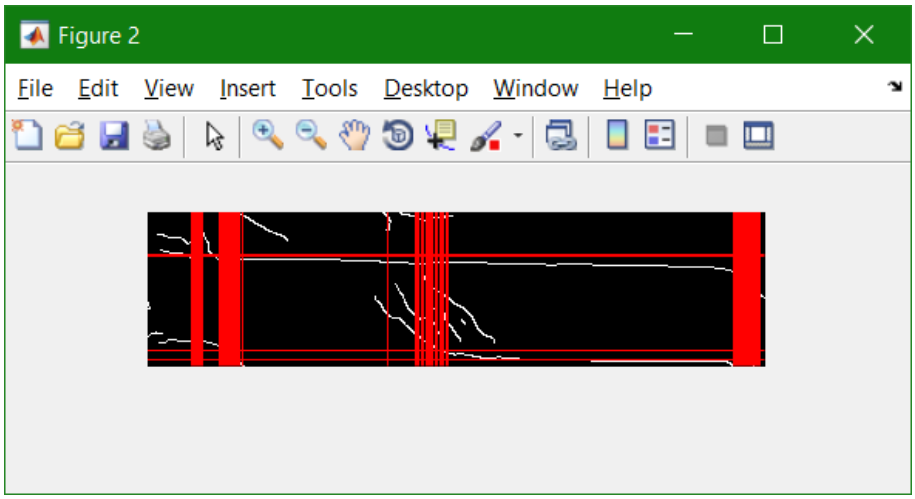


Figura 3.16: Ventana de la imagen con líneas detectadas

Otro de los mecanismos implementados para evitar la búsqueda de rectángulos en cada ventana de la imagen ha sido el de buscar rectángulos cercanos en la ventana antes de ejecutar esta búsqueda. De esta forma, una vez se conocen las coordenadas de la nueva ventana, se comprueba si ya se ha detectado algún rectángulo que se superponga con la ventana actual de forma que no quepa otro rectángulo en dicha ventana. En este caso, esa ventana no sería analizada, pues no detectaría ningún rectángulo.

Eliminación de líneas por intensidad

En la primera implementación del algoritmo, para filtrar y obtener las cuatro líneas pertenecientes a un rectángulo una vez que estaban detectadas todas las líneas horizontales y

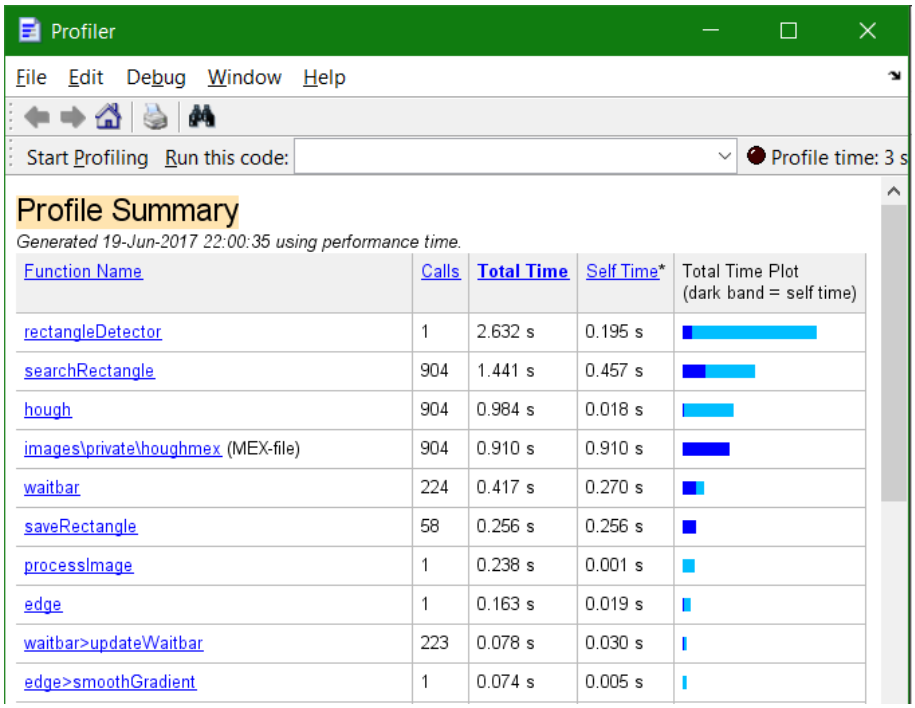


Figura 3.17: Tiempo de ejecución y tiempos de algunas funciones del algoritmo final

verticales se realizaba un doble bucle por cada conjunto de líneas. En este doble bucle se comprobaba cada línea con todas las siguientes para comprobar cuáles eran las dos líneas de mayor intensidad que se encontraban a cierta distancia.

Este doble bucle era muy ineficiente, para solucionarlo se ha implementado una solución alternativa: las líneas detectadas se ordenan por intensidad, la primera línea detectada será una de las dos que pertenecen a uno de los pares de paralelas, la siguiente línea que cumpla con el criterio de la distancia a la primera será la segunda línea del par de paralelas. De esta forma se ha pasado de una complejidad $O(n^2)$ a una complejidad $O(n)$.

3.2. Desarrollo del código

Este proyecto ha sido estructurado en diferentes funciones *MATLAB* que realizan llamadas entre sí. La función principal es *interface.mlapp*, que incluye la interfaz de usuario y el manejador de eventos. El funcionamiento de dichas funciones será explicado en detalle a continuación.

3.2.1. Interfaz de usuario

El archivo *interface.mlapp* es el que crea la interfaz que se muestra en la figura 3.19 y controla los eventos. Mediante las distintas funciones que manejan los eventos del sistema, se interactúa con las demás funciones.

Este archivo está compuesto por la definición de los componentes de la interfaz, diversas

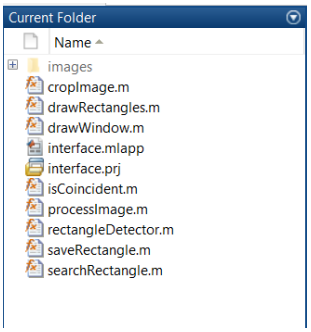


Figura 3.18: Estructura del proyecto en el espacio de trabajo

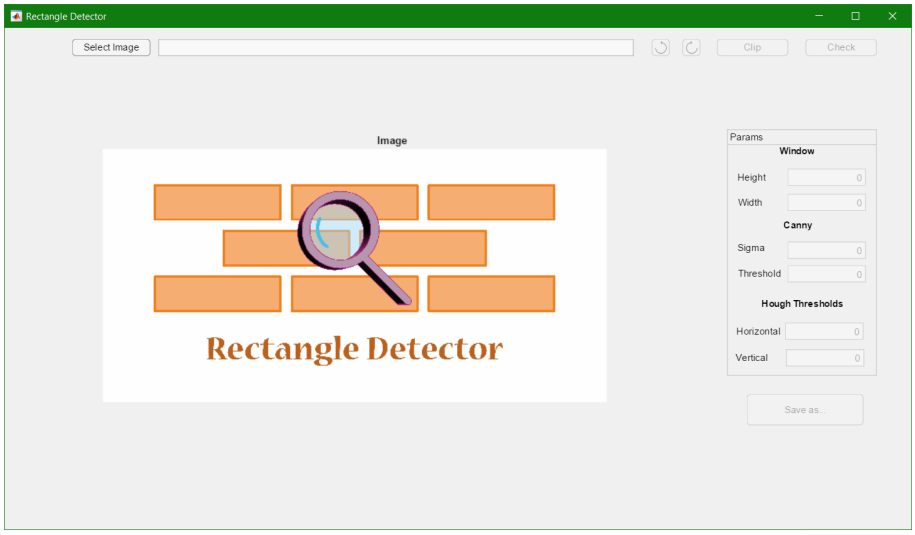


Figura 3.19: Interfaz de usuario de la aplicación

funciones privadas auxiliares y las funciones que se encargan de manejar los eventos. El sistema se asegura que el usuario únicamente pueda introducir valores válidos en los campos de entrada de texto.

3.2.2. Recortar imagen

La función *cropImage.m* se encarga de realizar un recorte de la imagen y devolverlo. Para ello lee la imagen de la ruta pasada como argumento, la muestra para que el usuario pueda realizar el recorte deseado, almacena dicho recorte y lo devuelve como argumento de salida de la función. Esta función además desactiva las advertencias del tipo "la imagen es demasiado grande para ser mostrada en pantalla".

Esta función recibe como argumentos de entrada la ruta de la imagen que se desea recortar, y el ángulo de la imagen. El argumento de salida de la imagen es la imagen ya recortada.

```
1 function [crop] = cropImage(filepath, degree)
2
```

```
3       warning('off', 'images:initSize:adjustingMag');
4
5       image = imread(filepath);
6       imshow(imrotate(image, degree));
7       crop = imcrop;
8       close all;
9   end
```

Código 3.1: Función cropImage.m

3.2.3. Detectar rectángulos

La función *rectangleDetector.m* se encarga de detectar rectángulos en la imagen pasada como argumento, y devolver dicha imagen con los rectángulos detectados dibujados en ella, y su número. Esta función recibe como parámetros de entrada el ancho y alto de la ventana deslizante, el umbral y sigma para realizar la detección de bordes mediante el método de Canny explicado anteriormente, y los umbrales horizontal y vertical para la detección de líneas con la función de Hough. Como argumentos de salida devuelve la imagen con los rectángulos dibujados, y su número.

Esta función primero comprueba si la imagen es demasiado grande, en cuyo caso reduce una serie de parámetros proporcionalmente. Posteriormente procesa la imagen de entrada con la función *processImage* e inicializa una serie de parámetros. La variable *rectanglesNumber* almacena el número de rectángulos detectados hasta el momento, la variable *rectanglesDetected* almacena los rectángulos detectados actualmente. Estos serán almacenados en forma de tabla con cuatro columnas, que representarán la coordenada y de la esquina superior izquierda del rectángulo detectado, la coordenada x de dicha esquina, la coordenada y del centro del rectángulo, y la coordenada x del centro del rectángulo. Otra variable importante es *mask*, que es una máscara donde se encuentran dibujados los rectángulos detectados hasta el momento.

Esta función utiliza dos bucles anidados para recorrer toda la imagen con una ventana deslizante cuyo tamaño ha sido decidido por el usuario. El bucle exterior se encarga de recorrer las filas, y el bucle interno de recorrer las columnas. Una vez en el bucle, primero se coge la nueva ventana a analizar, en función de la fila y columna actuales. Esta ventana deberá ser lo suficientemente grande para que quepa un rectángulo, pero debe estar lo suficientemente ajustada para que no quepa más de un rectángulo.

Posteriormente, se comprueba mediante la función *isCoincident* si ya hay un rectángulo en la ventana que se está analizando, o si hay uno ocupando un segmento de la ventana lo suficientemente grande como para que no quepa otro rectángulo. En caso de haber alguno, se pasa a calcular las coordenadas de la siguiente ventana deslizante; si no hay ninguno, se buscan un rectángulo en la ventana actual mediante la función *searchRectangle*. En caso de encontrar un rectángulo, este se guarda en la variable definida para ello mediante la función *saveRectangle*. Este proceso se muestra en el segmento de código 3.2.

```
...

[jump,maxIdx] = isCoincident(row,col,windowWidth,windowHeight,rectanglesDetected
    ,rectanglesNumber);

if ~jump
        slidingWindow = edgeImage(row:row+windowHeight-1,col:col+windowWidth-1);
        [detect, imageHorizontalRho, imageVerticalRho] = searchRectangle(
            slidingWindow, houghThresholdHorizontal, houghThresholdVertical);
        if detect
                [mask, rectanglesDetected, rectanglesNumber] = saveRectangle(
                    mask, imageHorizontalRho, imageVerticalRho,
                    rectanglesDetected, rectanglesNumber, row, col, factor);
        end
end

...
```

Código 3.2: Proceso de selección de ventana y búsqueda de rectángulo

El siguiente paso es calcular las coordenadas de la siguiente ventana, se comprueba si la ventana actual no se ha analizado por haber detectado ya un rectángulo en ella, o lo suficientemente cerca, en cuyo caso la nueva coordenada será la arista derecha de dicho rectángulo menos un margen; en caso de haber sido analizada la ventana, se comprueba si ha detectado un rectángulo, en cuyo caso la nueva coordenada será la arista derecha del rectángulo detectado menos un margen, o en caso de no haber detectado ninguno, simplemente aumentará su valor con un incremento predefinido. Este proceso se muestra en el segmento de código 3.3.

```
...

if jump
        col = 2*(rectanglesDetected(maxIdx,4)- rectanglesDetected(maxIdx,2)) +
            rectanglesDetected(maxIdx,2)-15;
elseif detect
        col=max(imageVerticalRho)-15;
else
        col=col+15;
end

...
```

Código 3.3: Proceso de cálculo de coordenadas de la ventana deslizante

Una vez terminado de analizar toda la imagen, se dibujan los rectángulos detectados en la imagen mediante la función *drawRectangles*. Durante todo el proceso de detección, se muestra al usuario una barra de progreso en la que se indican cuántos rectángulos han sido detectados hasta en el momento. Esta función también comprueba que el usuario no haya cancelado el proceso o que no haya ocurrido ningún problema.

3.2.4. Procesar imagen

La función *processImage.m* se encarga de la preparación previa de la imagen antes de la búsqueda de rectángulos. Para ello esta función redimensiona la imagen, la convierte a escala de grises y realiza la búsqueda bordes en la imagen utilizando el *método de Canny* explicado anteriormente. Como parámetros de entrada recibe la imagen a procesar, los valores del umbral y sigma para el método de Canny, y el factor de redimensión de la imagen. Esta función devuelve la imagen redimensionada y la imagen de bordes.

```
function [image, edgeImage]=processImage(image, threshold, sigma, factor)

imageSize = size(image);
image = imresize(image, [imageSize(1)/factor, imageSize(2)/factor]);
grayImage = rgb2gray(image);
edgeImage = edge(grayImage,'canny', threshold, sigma);
end
```

Código 3.4: Función processImage.m

3.2.5. Coincidencias

La función *isCoincident.m* se encarga de buscar si hay algún rectángulo lo suficientemente cerca de la ventana como para que no quepa otro rectángulo. Para ello calcula el centro de la imagen, y mediante la función *find* busca todos los rectángulos cuyo centro se encuentra lo suficientemente cerca horizontalmente del centro de la ventana actual.

Una vez que se tiene el conjunto de rectángulos que cumplen el requisito nombrado anteriormente, se busca en estos cuales tienen su centro lo suficientemente cerca verticalmente del centro de la ventana actual. En caso de haber algún rectángulo que cumpla esto, se calcula cual es el rectángulo cuya línea está mas superpuesta con el rectángulo.

Esta función recibe como argumentos de entrada la fila y columna de la ventana a analizar, el tamaño de la ventana, la tabla de rectángulos detectados hasta el momento y su número. Dicha función devuelve una variable booleana indicando si ha encontrado rectángulos superpuestos en la ventana actual, y por tanto no hace falta analizarla, y cuál es el rectángulo más próximo para poder encontrar la siguiente ventana a analizar.

3.2.6. Buscar rectángulo

La función *searchRectangle.m* es la más importante de este algoritmo, comprueba si hay un rectángulo en la ventana de la imagen pasada como argumento.

Esta función recibe como argumentos de entrada la imagen y los umbrales para la función de Hough. Esta función devuelve un valor indicando si se ha detectado algún rectángulo, y las

coordenadas horizontales y verticales de las líneas detectadas.

Esta función puede dividirse en una serie de pasos para su mejor comprensión:

1. **Función de Hough:** la ventana actual de la imagen es pasada por el algoritmo de Hough detallado anteriormente. Esto devuelve el conjunto de todas las líneas detectadas en el segmento de la imagen que se ha analizado. Estas líneas están representadas por sus valores de θ y ρ, y su intensidad está indicada en la matriz de Hough.

2. **Filtrado de líneas:** se eliminan todas aquellas líneas que no sean verticales ni horizontales, con cierto margen.

```
...

[H,~,R] = hough(image);

H(:,6:84) = 0;
H(:,96:175) = 0;

...
```

Código 3.5: Ejecución de la función de Hough y filtrado de líneas horizontales y verticales

3. **Obtención de líneas que superan un umbral de intensidad:** se eliminan todas aquellas líneas que no superan ciertos umbrales. Estos umbrales son definidos por el usuario. En el segmento de código 3.6 se muestra esta obtención de líneas para el conjunto de líneas verticales, para el conjunto de líneas horizontales ha sido omitido, pues es el mismo procedimiento.

4. **Conversión a coordenadas en la imagen:** se obtienen las coordenadas de los píxeles que forman las líneas en la imagen.

```
...

%Verticales
maxVertical = max(max(H(:,85:95)));
if maxVertical >= verticalThreshold
[verticalRho, verticalTheta] = find(H(:,85:95)>verticalThreshold);
        verticalTheta = verticalTheta + 84;
else
        verticalRho = [];
        verticalTheta = [];
end

%Coordenadas en la imagen
imageHorizontalRho = R(horizontalRho)';
imageVerticalRho = R(verticalRho)';

...
```

Código 3.6: Obtención de líneas verticales que superan cierto umbral y obtención de sus coordenadas en la imagen

5. **Obtención de los dos pares de paralelas que forman el rectángulo:** cada conjunto de líneas se ordena por intensidades y se seleccionan aquellas dos líneas de mayor intensidad que estén separadas cierta distancia. Las líneas horizontales forman un conjunto y las verticales otro, por tanto se obtendrán cuatro líneas o menos. En el segmento de código 3.7 se muestra este proceso para el conjunto de líneas horizontales, para el conjunto de líneas verticales sería el mismo proceso, por tanto se ha omitido.

```
...

%Horizontales
rho = [];

intensity = zeros(size(horizontalRho));
for i=1:(size(horizontalRho,1))
        intensity(i) = H(horizontalRho(i), horizontalTheta(i));
end

[horizontalIntensity, idx] = sort(intensity);
found = false;
cont = size(horizontalIntensity,1)-1;
while cont>0 && ~found
        if(abs(abs(imageHorizontalRho(idx(size(horizontalIntensity,1))))-
            abs(imageHorizontalRho(idx(cont)))) > threshMergeH)
                found = true;
                rho(2) = imageHorizontalRho(idx(cont));
        end
        cont=cont-1;
end

if(size(imageHorizontalRho,1)>0)
        rho(1) = imageHorizontalRho(idx(size(horizontalIntensity,1)));
end

imageHorizontalRho = rho';

...
```

Código 3.7: Obtención de las líneas horizontales que forman un rectángulo

6. **Comprobación de que se ha encontrado un rectángulo:** se comprueba que se hayan encontrado exactamente dos líneas horizontales y dos líneas verticales, en cuyo caso se entiende que se ha detectado un rectángulo en esa ventana de la imagen.

3.2.7. Guardar rectángulo

La función *saveRectangle.m* se encarga de almacenar el nuevo rectángulo detectado en la tabla de los rectángulos, añadirlo a la máscara e incrementar el contador de rectángulos.

Esta función recibe como argumentos de entrada la máscara sobre la que se dibujará el nuevo rectángulo, las coordenadas horizontales y verticales de las líneas pertenecientes al rectángulo detectado, la tabla con todos los rectángulos detectados, el contador de rectángulos, la fila y columna de la ventana que se está analizando actualmente, y el factor de redimensión de la imagen. Esta función devuelve como argumentos de salida la máscara con el nuevo rectángulo añadido, la tabla de los rectángulos detectados y el contador de rectángulos.

```
pointUpLeft = [min(abs(horizontalRho)) + row, min(abs(verticalRho)) + col];
width = abs(verticalRho(2) - verticalRho(1));
height = abs(abs(horizontalRho(2)) - abs(horizontalRho(1)));
pointCenter = [pointUpLeft(1)+round(height/2),pointUpLeft(2)+round(width/2)];

...

rectanglesNumber=rectanglesNumber+1;
rectanglesDetected(rectanglesNumber,:) = [pointUpLeft,pointCenter];

...
```

Código 3.8: Cálculo de las coordenadas a guardar en la tabla y guardado en la tabla de rectángulos

3.2.8. Dibujar rectángulos

La función *drawRectangles.m* se encarga de dibujar los rectángulos detectados en la imagen. Los rectángulos serán dibujados sin tener en cuenta su inclinación, es decir, todos los rectángulos serán dibujados con líneas completamente horizontales y completamente verticales.

Esta función recibe como argumentos de entrada la imagen que seleccionó el usuario, y una máscara con la posición de los rectángulos. Está máscara será una matriz booleana del tamaño de la imagen, indicando con un 1 los píxeles que pertenecen a los rectángulos, y 0 en otro caso. Esta función devuelve la imagen original con los rectángulos detectados en color rojo. Para cada rectángulo detectado se muestra su perímetro en color rojo y un punto marcando su centro.

```
...

im_r = uint8(image(:,:,1));
im_g = uint8(image(:,:,2));
im_b = uint8(image(:,:,3));
```

```
...

im_r(mask) = 255;
im_g(mask) = 0;
im_b(mask) = 0;

image(:,:,1) = im_r;
image(:,:,2) = im_g;
image(:,:,3) = im_b;

...
```

Código 3.9: Coloreado en rojo de los rectágulos sobre la imagen

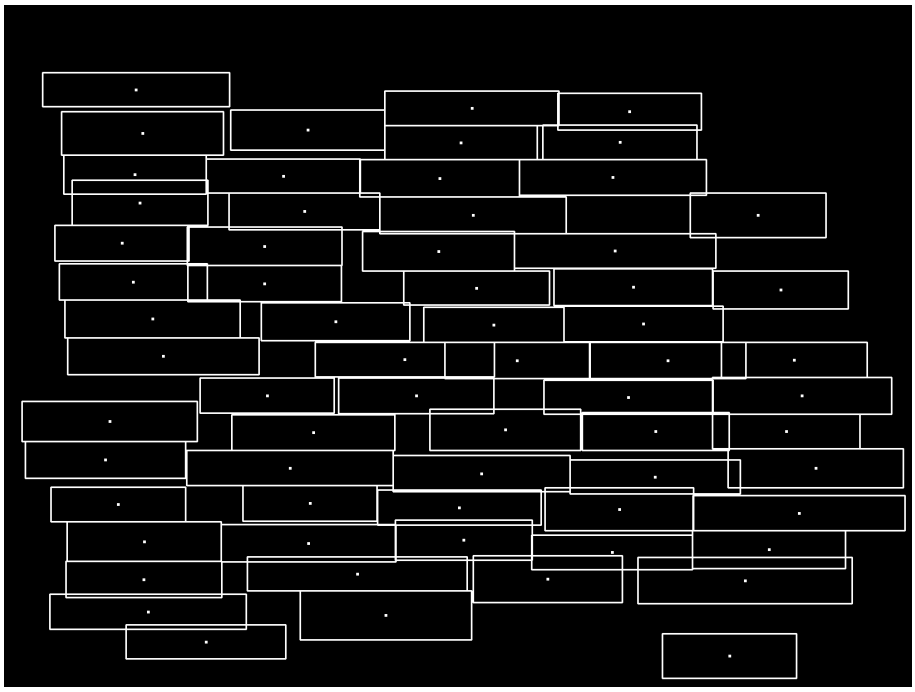


Figura 3.20: Máscara con los rectángulos detectados

3.2.9. Dibujar ventana

La función *drawWindow.m* no se ha utilizado en la implementación final, sólo ha sido utilizada con fines de desarrollo, pero debido a su gran utilidad para la depuración de código y detección de fallos se ha decidido su inclusión en este apartado.

Esta función se encarga de dibujar las líneas detectadas en la ventana de la imagen pasada como argumento. Únicamente puede ser ejecutada desde MATLAB, pues muestra las líneas sobre la imagen desde un elemento *figure*, y no superpuestas en la imagen. Dicha función recibe como argumentos de entrada la imagen sobre la que se desea dibujar las líneas, y las coordenadas de las líneas a dibujar. La imagen de entrada será habitualmente una imagen de bordes, pero también pueden dibujarse las líneas sobre la imagen a color.

La figura 3.21 muestra una ventana de la imagen de bordes con las líneas detectadas superpuestas. Este proceso para dibujar estas líneas en la ventana de la imagen puede observarse en el segmento de código 3.10

```
...

point1_horizontal = [zeros(size(imageHorizontalRho,1),1), abs(imageHorizontalRho
    )];
point2_horizontal = [ones(size(imageHorizontalRho,1),1)*size(image,2), abs(
    imageHorizontalRho)];

if(size(point1_horizontal,1)>0 && size(point1_horizontal,2) == size(
    point2_horizontal,2))
for i=1:size(point1_horizontal,1)
plot([point1_horizontal(i,1), point2_horizontal(i,1)],[point1_horizontal(i,2),
    point2_horizontal(i,2)], 'Color', 'red');
end
end

...
```

Código 3.10: Dibujo de líneas horizontales en una ventana de la imagen

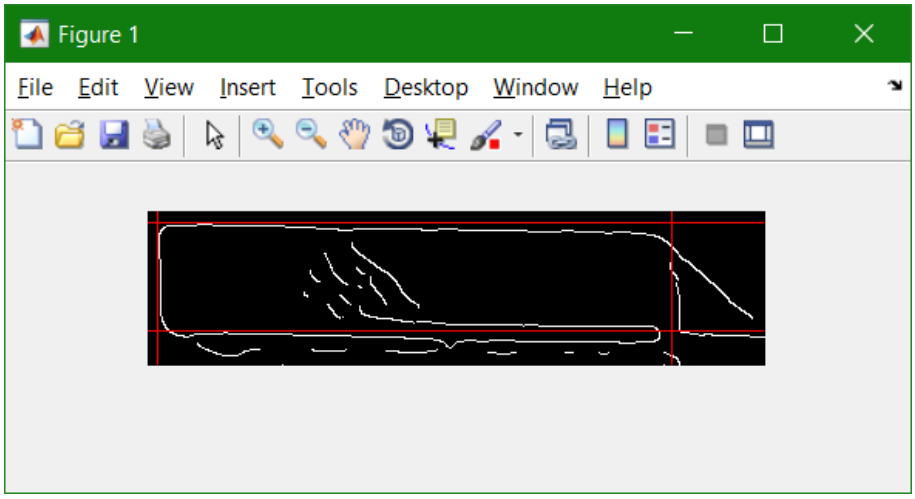


Figura 3.21: Ventana con las líneas dibujadas

4. Pruebas

Este proyecto ha sido desarrollado y probado en un equipo con las siguientes características:

- Windows 10 home
- Procesador Intel core i7-4700MQ
- 8GB de memoria RAM
- 1TB de HDD
- Tarjeta gráfica NVIDIA GeForce GT-745M de 4GB
- MATLAB R2016b

Durante el desarrollo de este proyecto se han realizado pruebas para verificar el correcto funcionamiento de las diferentes funciones. Estas pruebas han consistido en comprobar las salidas de las funciones conociendo su entrada. Estas pruebas han sido realizadas de forma visual, pues al trabajar con imágenes hubiera sido muy complicado automatizar este proceso.

Las pruebas han sido realizadas teniendo en cuenta tanto el número de rectángulos detectados, como el tiempo de ejecución. Para estas pruebas se han utilizado una serie de imágenes de las que se conocía el número de rectángulos que hay en dicha imagen. Estas imágenes han sido analizadas con diferentes parámetros de entrada para comprobar la diferencia en cuanto a tiempo y número de detecciones.

El algoritmo propuesto ha sido probado sobre diversas imágenes para observar su comportamiento con diferentes valores en los parámetros. Algunas de las imágenes que han sido probadas, y las conclusiones a las que se ha llegado con cada una de ellas se detallan a continuación.

Puede observarse en las imágenes siguientes que el tiempo de ejecución depende en gran medida del número de rectángulos detectados: a mayor número de rectángulos, menor tiempo de ejecución.

4.1. Primera imagen

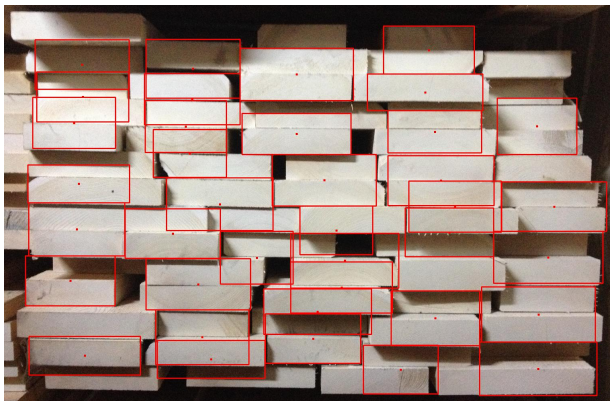
Alto de ventana	Ancho de ventana	Umbral de Canny	Sigma de ventana	Umbral horizontal de Hough	Umbral vertical de Hough	Detectados	Calidad de la detección	Tiempo de ejecución (sg)
200	800	3	0.08	140	35	61	Buena	2.632
100	400	3	0.075	10	5	103	Mala	10.577
200	800	2.5	0.1	50	10	64	Regular	1.862
300	600	3	0.075	200	50	38	Mala	7.730

Tabla 4.1: Parámetros de detección para la primera imagen

(a) Parámetros de la fila 1

(b) Parámetros de la fila 2

(c) Parámetros de la fila 3



(d) Parámetros de la fila 4

Figura 4.1: Imágenes con los parámetros de la tabla 4.1

La figura 4.1 muestra la primera imagen con los parámetros de la tabla 4.1. Como puede

observarse, el tamaño de la ventana es muy importanto, pues si no están bien ajustados no buscará los rectángulos de la imagen.

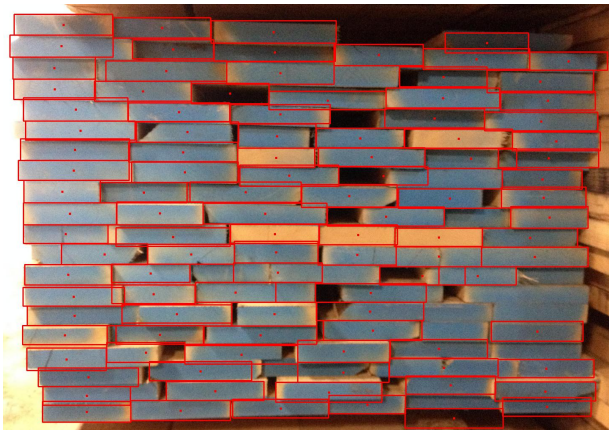
4.2. Segunda imagen

Alto de ventana	Ancho de ventana	Umbral de Canny	Sigma de ventana	Umbral horizontal de Hough	Umbral vertical de Hough	Detectados	Calidad de la detección	Tiempo de ejecución (sg)
140	600	3	0.08	150	50	51	Mala	10.548
140	600	2	0.08	100	25	105	Buena	3.073
140	600	4	0.08	200	80	3	Mala	17.738
140	600	4	0.035	20	5	112	Regular	1.769

Tabla 4.2: Parámetros de detección para la segunda imagen

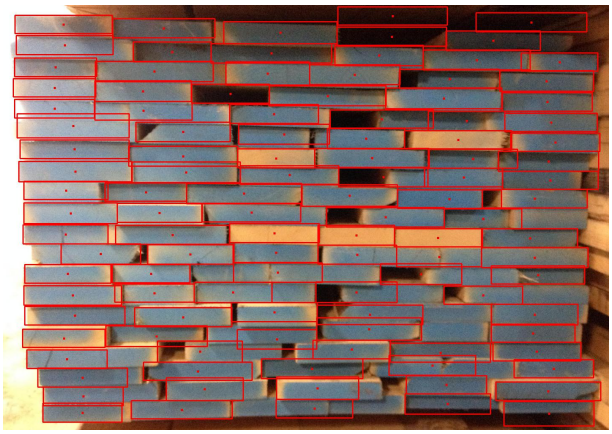
La figura 4.2 muestra la segunda imagen con los parámetros de la tabla 4.2. Como puede observarse, unos umbrales de Hough demasiado altos hacen que no se detecten apenas rectángulos, mientras que unos parámetros excesivamente bajos hace que aparezcan demasiados falsos positivos.

(a) Parámetros de la fila 1



(b) Parámetros de la fila 2

(c) Parámetros de la fila 3



(d) Parámetros de la fila 4

Figura 4.2: Imágenes con los parámetros de la tabla 4.2

4.3. Tercera imagen

La figura 4.3 muestra la tercera imagen con los parámetros de la tabla 4.3. Al igual que en la segunda imagen, puede observarse que unos umbrales de Hough demasiado altos hacen que no se detecten muchos rectángulos, mientras que con unos parámetros excesivamente bajos aparezcen demasiados falsos positivos.

Alto de ventana	Ancho de ventana	Umbral de Canny	Sigma de ventana	Umbral horizontal de Hough	Umbral vertical de Hough	Detectados	Calidad de la detección	Tiempo de ejecución (sg)
175	600	4	0.045	125	45	63	Buena	4.159
200	600	2.8	0.045	100	40	70	Regular	1.804
200	600	2.8	0.045	200	75	28	Mala	18.769
200	600	2.8	0.045	125	45	65	Regular	2.520

Tabla 4.3: Parámetros de detección para la tercera imagen

(a) Parámetros de la fila 1

(b) Parámetros de la fila 2

(c) Parámetros de la fila 3

(d) Parámetros de la fila 4

Figura 4.3: Imágenes con los parámetros de la tabla 4.3

5. Conclusiones

El procesamiento de imágenes y la visión por computador tienen multitud de aplicaciones en la actualidad. Sin embargo, para que los algoritmos basados en imágenes funcionen correctamente, es necesario que estas imágenes sean tomadas en entornos controlados de luz y sombras, pues realizar un algoritmo sin tener en cuenta estas limitaciones es muy complejo y lleva a una gran cantidad de errores.

Aún así, se ha intentado realizar un algoritmo lo más independiente posible de estas condiciones, sin embargo, en el apartado de pruebas puede observarse que las imágenes que mejor resultado obtienen son aquellas con unas condiciones determinadas: la imagen debe ser tomada evitando crear perspectiva, y con la fuente de luz de frente, de manera que se formen las menos sombras posibles.

Este proyecto ha cumplido con los objetivos propuestos, pues pretendía investigar, estudiar y desarrollar un algoritmo para la detección y conteo de tablones de maderas. Esto se ha conseguido para aquellos tablones que tengan forma rectangular, pues la aplicación propuesta detecta formas rectangulares en una imagen. Esta detección no es perfecta, debido a los problemas planteados anteriormente, pero si aporta una detección bastante precisa para las imágenes tomadas en las condiciones mencionadas anteriormente.

El algoritmo propuesto en este Trabajo de Fin de Grado tiene aspectos que podrían mejorarse, pero debido a su extensión han quedado fuera de los objetivos de este proyecto.

5.1. Posibles mejoras

Una de las posibles mejoras que podrían implementarse para mejorar este proyecto sería el desarrollo de una inteligencia artificial con aprendizaje que calculara automáticamente el valor de los parámetros necesarios para el algoritmo de Canny.

Además, también podría mejorarse la aplicación realizada añadiendo un selector de color para elegir el color de los rectángulos detectados, de forma que se permitiría al usuario cambiar el color de los rectángulos una vez que han sido detectados. Esto permitiría que si los rectángulos a detectar son de color parecido al rojo, pudieran visualizarse mejor.

Otra posible mejora que también podría implementarse sería una aplicación en un formato más usual, es decir, una aplicación que pueda ejecutar en la mayoría de equipos o dispositivos móviles sin tener que instalar nada en especial aparte de dicha aplicación, pues para ejecutar la aplicación realizada en MATLAB es necesario tener instalado su entorno de ejecución, y este no se encuentra instalado normalmente en los equipos de los usuarios que no utilizan MATLAB.

Bibliografía

[1] Mathworks Inc. Referencia de MATLAB.
https://es.mathworks.com, visitado por última vez el 27/05/17.

[2] González Jiménez, Javier. Apuntes de la asignatura *Visión por Computador*.
https://informatica.cv.uma.es, visitado por última vez el 20/05/17.

[3] Muñoz Pérez, José. Apuntes de la asignatura *Procesamiento de Imágenes y Video*.
https://informatica.cv.uma.es, visitado por última vez el 23/05/17.

[4] Jung, Claudio Rosito y Schramm, Rodrigo. Rectangle Detection Based on a Windowed Hough Transform. Editorial EEE Computer Society. 2004. ISBN: 0-7695-2227-0.
Descargado de http://dl.acm.org/citation.cfm?id=1025131.1026243.

[5] LaTex Stack Exchange. Referencia de Latex.
https://es.mathworks.com, visitado por última vez el 14/06/17.

[6] UML Web Site. Referencia de UML.
http://www.uml.org/, visitado por última vez el 17/03/17.

[7] MagicDraw. Referencia de MagicDraw.
https://www.nomagic.com/products/magicdraw, visitado por última vez el 11/03/17.

[8] Luis. Aprendiendo LaTex.
http://minisconlatex.blogspot.com.es/, visitado por última vez el 25/06/17.

[9] Kottwitz, Stefan. LaTex Begginer's Guide. Editorial Packt Publishing Ltd. Marzo 2011. ISBN: 978-1-847199-86-7.
Descargado de www.packtpub.com, visitado por última vez el 22/06/17.

[10] Stack Overflow. Referencias de MATLAB y LaTex.
https://stackoverflow.com/, visitado por última vez el 20/06/17.

Anexo I: Manual de usuario

En este apartado se incluye el manual de usuario que describe el funcionamiento de la app y las funcionalidades incluidas.

Rectangle Detector

Manual de Usuario

Índice general

Introducción

Rectangle Detector es una aplicación desarrollada con el objetivo de detectar rectángulos en imágenes. La calidad de esta detección depende de diversos factores, como son las condiciones en las que fue tomada la fotografía, y los parámetros elegidos por el usuario.

Las condiciones en las que fue tomada una imagen son importantes, pues una imagen borrosa o con muchas sombras no obtendrá buenos resultados, mientras que cuanto mejores sean estas condiciones, más precisa será la detección.

La importancia de los parámetros elegidos por el usuario reside en que estos parámetros serán los que definan el tamaño de los rectángulos a buscar, y la intensidad mínima de las líneas para que estas puedan ser consideradas como parte de un rectángulo.

Configuración

En este apartado se indican los requisitos que debe tener el sistema para que la aplicación funcione correctamente, y como ejecutar la aplicación.

2.1. Requisitos del sistema

Para que esta aplicación funcione correctamente es necesario que el sistema tenga instalado la versión R2016b o superior de *MATLAB*. Esta versión de *MATLAB* tiene los siguientes requerimientos mínimos.

En sistemas operativos *Windows*:

- Windows 7 o superior
- Procesador Intel o AMD x86-64
- 6GB de espacio en disco
- 2GB de RAM

2.2. Ejecución de la aplicación

Hay dos formas diferentes de ejecutar la aplicación: desde el directorio de la aplicación, o desde *MATLAB*.

2.2.1. Ejecución desde el directorio de la aplicación

Para la ejecución de la aplicación desde el directorio de la aplicación es necesario que las distintas funciones y la interfaz se encuentren en la misma carpeta. Una vez en la carpeta, sólo es necesario abrir el archivo *interface.mlapp*. La ejecución de este archivo abre *MATLAB*, y una vez que este ha cargado, se abre la aplicación *Rectangle Detector*.

2.2.2. Ejecución desde MATLAB

Para la ejecución de la aplicación desde *MATLAB* es necesario que tanto las funciones como la interfaz se encuentren en el espacio de trabajo de *MATLAB*.

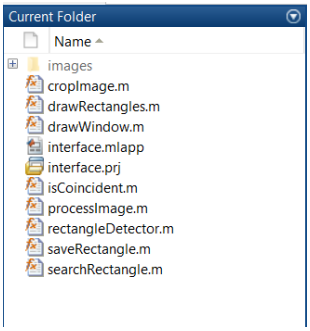


Figura 2.1: Estructura del espacio de trabajo de *MATLAB*

Desde la ventana de comandos de *MATLAB*, es necesario escribir el comando *interface*, hacer doble click sobre el archivo *interface.mmlap* o pulsar con el botón derecho en el archivo y dar a *run*.

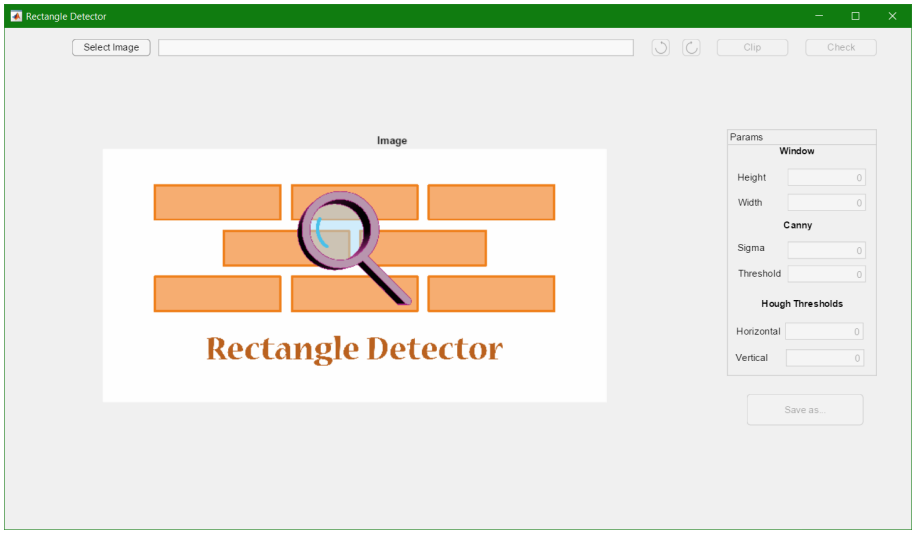


Figura 2.2: Aplicación *Rectangle Detector*

Funcionalidades de la aplicación

Esta aplicación cuenta con distintas funcionalidades que serán explicadas a continuación.

3.1. Cargar una imagen

Esta funcionalidad está disponible al abrir la aplicación en el botón *Select Image* (figura 3.1). Esta función se encarga de abrir un explorador del sistema para que el usuario seleccione una imagen. La imagen seleccionada será de tipo *.JPG* o *.PNG*.

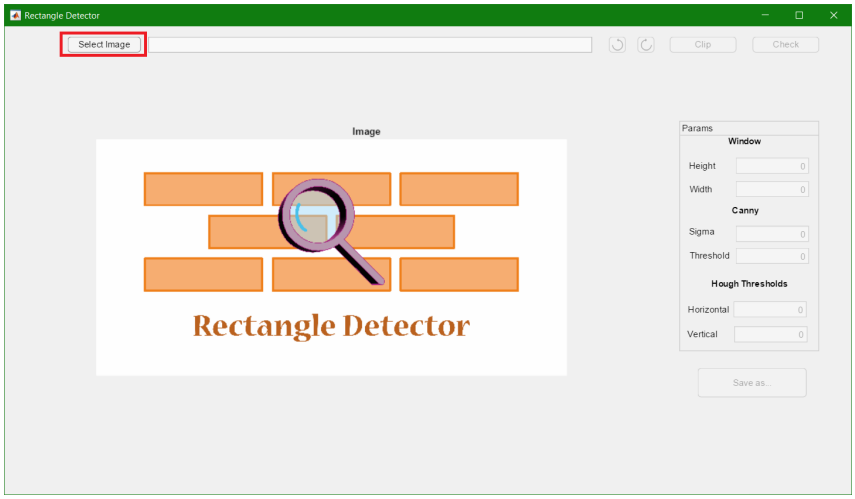


Figura 3.1: Función *cargar una imagen*

Una vez seleccionada una imagen, esta se carga en la interfaz para que el usuario pueda visualizarla.

3.2. Recortar una imagen

Esta funcionalidad está accesible una vez que el usuario ha cargado una imagen mediante la funcionalidad *cargar una imagen*.

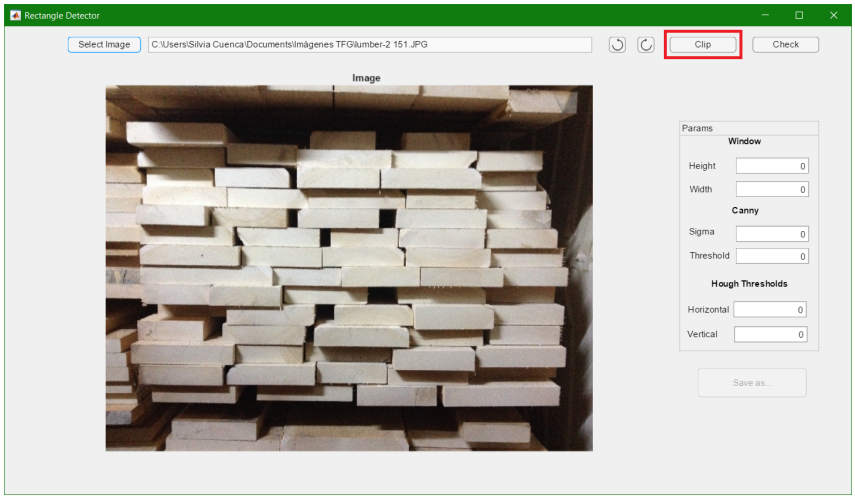


Figura 3.2: Función *recortar una imagen*

Al pulsar sobre el botón *Clip* (figura 3.3), se abre una ventana emergente en la que el usuario selecciona la zona que desea analizar.

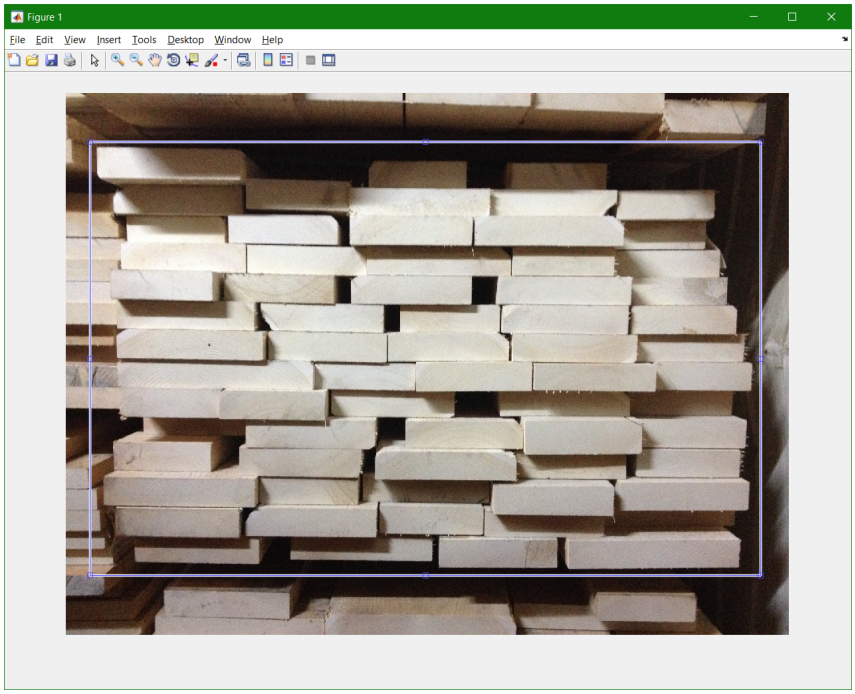


Figura 3.3: Selección de zona para analizar

Una vez seleccionada la zona a analizar, esta se carga en la interfaz para que el usuario pueda visualizarla.

3.3. Girar una imagen

Esta funcionalidad está accesible una vez que el usuario ha cargado una imagen mediante la funcionalidad *cargar una imagen*.

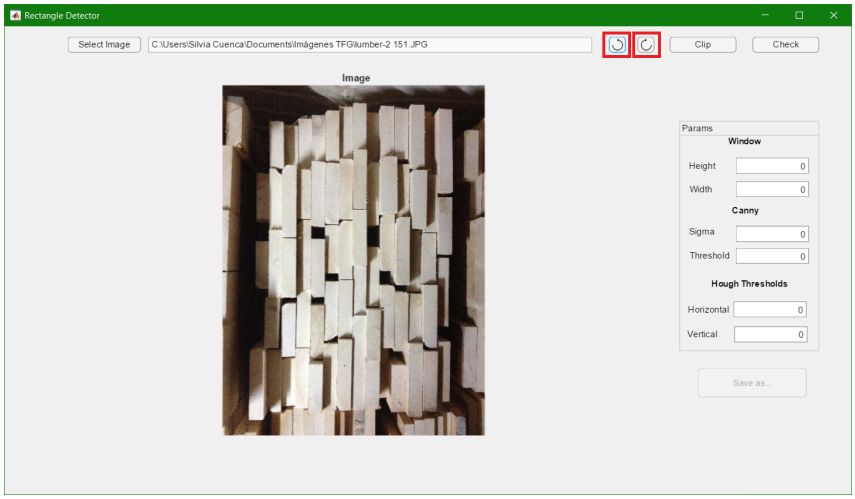


Figura 3.4: Función *girar una imagen*

Al pulsar sobre los botones de giro (figura 3.4), se rota la imagen que está en la interfaz. El sentido del giro depende del botón pulsado, puede girarse 90º en el sentido horario o antihorario.

3.4. Buscar rectángulos en una imagen

Esta funcionalidad está accesible una vez que el usuario ha cargado una imagen mediante la funcionalidad *cargar una imagen*.

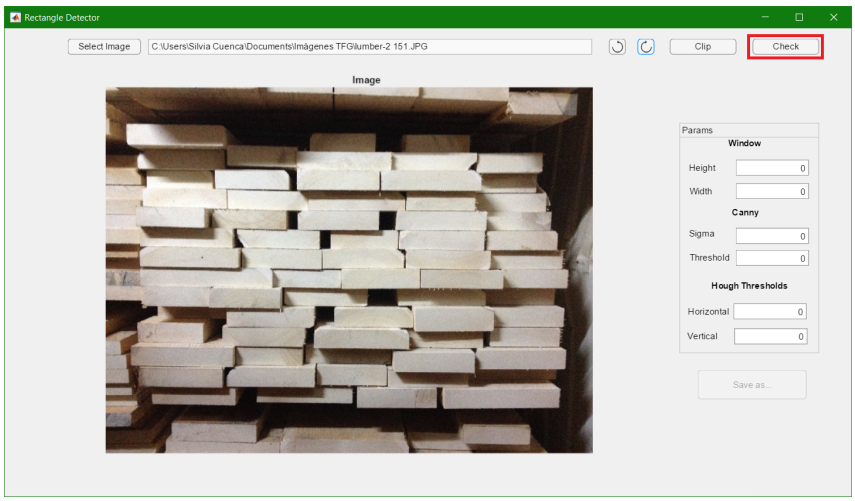


Figura 3.5: Función *buscar rectángulos en imagen*

Para que se busquen rectángulos, el usuario ha debido de completar los parámetros correctamente y pulsar sobre el botón *Check* (figura 3.5). En caso de que los parámetros no sean correctos, se mostrará el mensaje de error que se muestra en la figura 3.6.

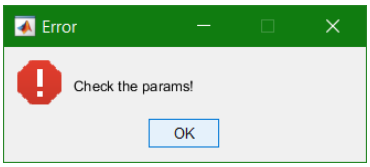


Figura 3.6: Ventana de error en caso de que algún parámetro no sea válido

Una vez que los parámetros introducidos sean correctos, se analiza la imagen buscando rectángulos. Mientras se está analizando se muestra una barra de progreso informando del progreso actual y del número de rectángulos detectados hasta el momento (figura 3.7).

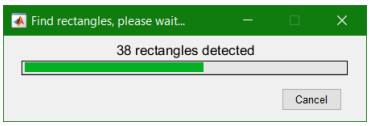


Figura 3.7: Barra de progreso en la búsqueda de rectángulos

Una vez completado el proceso de análisis, se muestra la imagen con los rectángulos detectados, y un mensaje indicando cuántos rectángulos han sido detectados, o si se ha producido algún error.

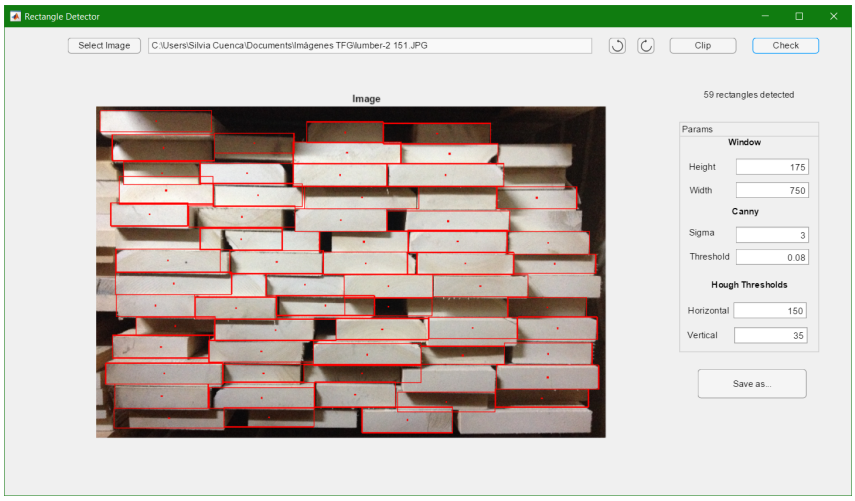


Figura 3.8: Rectángulos detectados en la imagen

3.5. Guardar una imagen

Esta funcionalidad está accesible una vez que el usuario ha detectado rectángulos en la imagen mediante la funcionalidad *buscar rectángulos en una imagen*.

Al pulsar en el botón *Save as...* (figura 3.9) se muestra el explorador del sistema, permitiendo al usuario seleccionar una ubicación donde guardar la imagen con los rectángulos detectados.

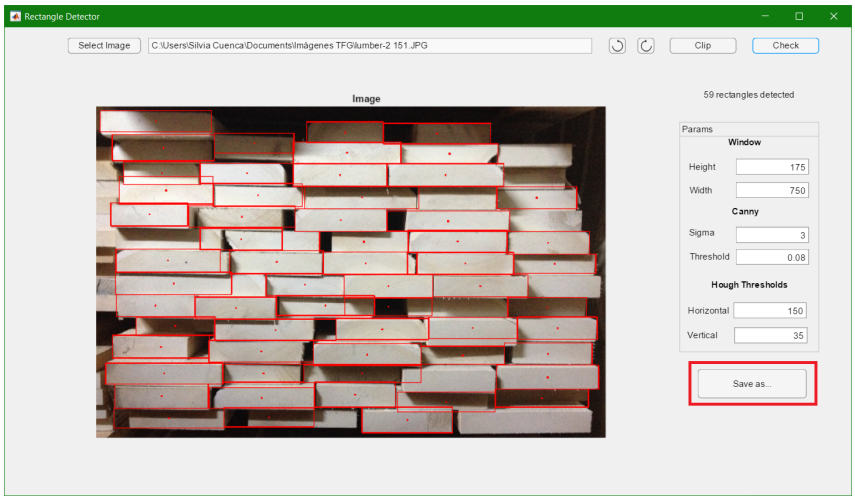


Figura 3.9: Función *guardar una imagen*