Grado en Ingeniería Electrónica Industrial

Escuela Politécnica Superior de Jaén

Universidad de Jaén

Escuela Politécnica Superior de Jaén

Trabajo Fin de Grado

JUPYTER NOTEBOOK COMO HERRAMIENTA DE APOYO A LA DOCENCIA EN LA ASIGNATURA VISIÓN POR COMPUTADOR

Alumno: Sandra Cano López

Tutor: Prof. Dª. Silvia María Satorres Martínez

Dpto: Ingeniería Electrónica y Automática

Septiembre, 2021

Universidad de Jaén
Escuela Politécnica Superior de Jaén
Departamento de Ingeniería Electrónica y Automática

Dª SILVIA MARÍA SATORRES MARTÍNEZ, tutor del Trabajo Fin de Grado titulado: JUPYTER NOTEBOOK COMO HERRAMIENTA DE APOYO A LA DOCENCIA EN LA ASIGNATURA VISIÓN POR COMPUTADOR, que presenta SANDRA CANO LÓPEZ, autoriza su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, SEPTIEMBRE de 2021

El alumno:

SANDRA CANO LÓPEZ

Los tutores:

SATORRES MARTINEZ SILVIA - 26741651B

Firmado digitalmente por SATORRES MARTINEZ SILVIA - 26741651B
Nombre de reconocimiento (DN): c=ES, serialNumber=IDCES-26741651B, givenName=SILVIA, sn=SATORRES MARTINEZ, cn=SATORRES MARTINEZ SILVIA - 26741651B
Fecha: 2021.09.06 12:09:43 +02'00'

SILVIA MARÍA SATORRES MARTÍNEZ

A mi padre, a mi madre y a mi hermano, por su apoyo incondicional y ser mi gran motor.

RESUMEN

El objetivo de este proyecto es el desarrollo de una herramienta informática formada por cuadernos realizados con Jupyter Notebook como apoyo a la docencia en la asignatura Visión por Computador que facilite el aprendizaje de la misma. La herramienta básicamente está basada en el desarrollo de una interfaz web que permite la ejecución de código Python en un navegador. De esta forma, se facilitará la docencia de la asignatura incrementando la motivación para su estudio, tanto en formato virtual como en un escenario mixto.

ABSTRACT

The objective of this project is the development of a computer tool made up of notebooks made with Jupyter Notebook to support teaching in the Computer Vision subject that facilitates its learning. The tool is basically based on the development of a web interface that allows the execution of Python code in a browser. In this way, the teaching of the subject will be facilitated by increasing the motivation for its study, both in virtual format and in a mixed setting.

AGRADECIMIENTOS

Quisiera dar las gracias a cada una de las personas que han formado parte de este trayecto. A mi familia y amigos, por apoyarme desde el primer día confiando ciegamente en mí. A mis compañeros, por vivir conmigo el día a día de este camino, por superarnos juntos y por celebrar cada mérito conseguido. A mi abuela, por darme las fuerzas y el coraje necesario de una mujer luchadora. A nuestro pequeño ángel, que estuvo a un paso de conseguir lo mismo que nosotros. A mi tutora, por su motivación y amabilidad. Y en especial, a mis padres y a mi hermano, por ser mi gran motor y por darme la oportunidad de perseguir mis sueños.

Sandra Cano López

ÍNDICE

RESUMEN ii
ABSTRACT iii
AGRADECIMIENTOS iv
1 - INTRODUCCIÓN 1
1.2 - Motivación 1
1.3 - Objetivos 2
1.4 - Estructura de la memoria 2
2 - ESTADO DEL ARTE 4
2.1 - Metodologías innovadoras 5
2.1.1 - Clase invertida 5
2.1.2 - Aprendizaje basado en proyectos 6
2.1.3 - Aprendizaje cooperativo 7
2.1.4 - Gamificación 7
2.1.5 - Aprendizaje basado en problemas 9
2.1.6 - Design thinking 9
2.1.7 - Aprendizaje basado en el pensamiento 10
2.1.8 - Aprendizaje basado en competencias 11
2.1.9 - Aprendizaje colaborativo 12
2.2 - Aprendizaje mixto 14
2.3 - Herramientas 15
2.3.1 - Matlab 15
2.3.2 - Jupyter Notebook 17
3 - JUPYTER NOTEBOOK 18
3.1 - Instalación 19
3.2 - Descripción del entorno 20
3.3 - Usos y aplicaciones 23
3.4 - Exportar a otros formatos 24
4 - PRÁCTICAS DE VISIÓN POR COMPUTADOR 25
4.1 - Manipulación de imágenes 25
4.1.1 - Leer y mostrar una imagen 25
4.1.2 - Añadir bordes a una imagen 25
4.1.3 - Rotación de una imagen 26
4.1.4 - Reflejo de una imagen 26
4.1.5 - Ecualización de una imagen 26
4.2 - Reducción de ruido en una imagen digital 26

4.2.1 - Filtros lineales 27
4.2.2 - Filtros no lineales. Filtro de la mediana 28
4.3 - Detección de bordes en una imagen 29
4.3.1 - Operadores basados en la primera derivada (Gradiente) 29
4.3.2 - Operadores basados en la segunda derivada (Laplaciana) 32
4.4 - Segmentación I. Transformada de Hough 34
4.5 - Segmentación II. Umbralización 36
4.6 - Segmentación III. Crecimiento de regiones 37
4.7 - Descripción de regiones basadas en el contorno 39
4.8 - Reconocimiento de formas geométricas y colores 41
4.8.1 - Reconocimiento de objetos 41
4.8.2 - Reconocimiento de colores 42
5 - RESULTADOS 44
5.1 - Manipulación de imágenes 45
5.2 - Reducción de ruido en una imagen digital 52
5.3 - Detección de bordes en una imagen 58
5.4 - Segmentación I. Transformada de Hough 63
5.5 - Segmentación II. Umbralización 67
5.6 - Segmentación III. Crecimiento de regiones 70
5.7 - Descripción de regiones basadas en el contorno 72
5.8 - Blob detection 76
5.9 - Reconocimiento de formas geométricas y colores 79
6 - CONCLUSIONES 84
7 - BIBLIOGRAFÍA 85

ÍNDICE DE FIGURAS

Figura 2.1 Modelo tradicional y clase invertida 6
Figura 2.2 Técnicas mecánicas de la gamificación 8
Figura 2.3 Técnicas dinámicas de la gamificación 8
Figura 2.4 Pasos del Design Thinking 10
Figura 2.5 Aprendizaje basado en competencias 12
Figura 3.6 Navegador de Anaconda 19
Figura 3.7 Entorno de Jupyter Notebook 20
Figura 3.8 Cuaderno de Jupyter Notebook 20
Figura 3.9 Celda en modo edición 21
Figura 3.10 Celda en modo comando 21
Figura 4.11 Máscaras de Roberts 30
Figura 4.12 Máscaras de convolución 31
Figura 4.13 Máscara Laplaciana 32
Figura 4.14 Máscara Laplaciana más genérica 33
Figura 4.15 Método de circularidad 42
Figura 4.16 Código para el reconocimiento de color 43
Figura 5.18 Lectura y representación de una imagen 46
Figura 5.19 Conversión a escala de grises 47
Figura 5.20 Añadir bordes a una imagen 48
Figura 5.21 Función para rotar una imagen 49
Figura 5.22 Imagen con diferentes rotaciones 50
Figura 5.23 Imagen invertida 50
Figura 5.24 Función para la ecualización 51
Figura 5.25 Resultados de la imagen ecualizada 51
Figura 5.26 Ruido de sal y pimienta 52
Figura 5.27 Ruido gaussiano 53
Figura 5.28 Filtro de la media 54
Figura 5.29 Creación de un kernel 54
Figura 5.30 Filtro de la mediana 54
Figura 5.31 Filtro gaussiano 54
Figura 5.32 Ruido de sal y pimienta y ruido gaussiano 55
Figura 5.33 Ruido gaussiano con distintos filtros 56
Figura 5.34 Ruido gaussiano con filtro de la media y mediana 56
Figura 5.35 Ruido de sal y pimienta con filtro de la media y mediana 57
Figura 5.36 Operador Roberts horizontal y vertical 58
Figura 5.37 Operador Roberts 59
Figura 5.38 Operador Prewitt horizontal y vertical 59
Figura 5.39 Operador Prewitt 60
Figura 5.40 Operador Sobel horizontal y vertical 60
Figura 5.41 Operador Sobel 61
Figura 5.42 Laplaciana 61
Figura 5.43 Operador Canny 62
Figura 5.44 Imagen con bordes detectados (I) 64
Figura 5.45 Transformada de Hough estándar 65
Figura 5.46 Imagen con bordes detectados (II) 66
Figura 5.47 Transformada probabilística de Hough 66

Figura 5.48 Umbralización binaria e invertida 68
Figura 5.49 Código para la umbralización de Otsu 69
Figura 5.50 Resultados umbralización global manual y Otsu 69
Figura 5.51 Código para el crecimiento de regiones (I) 70
Figura 5.52 Código para el crecimiento de regiones (II) 71
Figura 5.53 Crecimiento de regiones 71
Figura 5.54 Descripción de regiones basadas en el contorno 72
Figura 5.55 Búsqueda y representación de contornos 74
Figura 5.56 Momentos invariantes y descriptores topológicos 75
Figura 5.57 Figuras con centroides 75
Figura 5.58 Blob detection 77
Figura 5.59 Resultado de Blob detection 78
Figura 5.60 Código para el reconocimiento de figuras 80
Figura 5.61 Reconocimiento de figuras 81
Figura 5.62 Espacio de color HSV 81
Figura 5.63 Código para el reconocimiento de colores 82
Figura 5.64 Reconocimiento de colores 83

1 - INTRODUCCIÓN

Actualmente en la docencia se destaca la innovación como uno de los pilares principales para la mejora de las clases presenciales. En concreto, se hace hincapié en el uso de herramientas interactivas para favorecer el aprendizaje del alumno. El presente TFG se encuentra enmarcado en esta línea aportando el desarrollo de contenidos multimedia en la asignatura optativa Visión por Computador, impartida en el cuarto curso del Grado en Ingeniería Electrónica Industrial.

Por otro lado, siendo conscientes del año tan complicado y lleno de cambios que nos ha tocado vivir, hemos tenido que aprender a adaptarnos a una nueva metodología de enseñanza en la que muchos de los docentes se han visto perdidos. Es por ello que este proyecto puede tener gran importancia en años posteriores, puesto que es beneficioso tanto en la enseñanza virtual como en la presencial, mejorando esta última a gran nivel.

Los entornos de computación interactiva permiten llevar a cabo tareas de programación de manera intuitiva e inmediata. De esta forma los estudiantes pueden realizar cambios y ver los resultados al instante, haciendo que el flujo de interacción entre ellos y el medio sea ágil y sencillo. La herramienta para llevar a cabo este objetivo será Jupyter Notebook, un recurso muy valioso que está siendo adoptado en muchas instituciones educativas de todo el mundo.

1.2 - Motivación

La educación es una de las bases primordiales en el crecimiento de cualquier persona o sociedad y es por ello que se considera un derecho fundamental del ser humano. Aunque no para todos se encuentra al mismo alcance y en las mismas condiciones.

Generalmente, a la mayoría de los estudiantes no les resulta tan sencillo asimilar los conocimientos adquiridos en un aula y mucho menos ponerlos en práctica. Es por este motivo que se han ido desarrollando muchas herramientas interactivas a lo largo de los últimos años, para favorecer su aprendizaje y de esta forma incrementar el valor de sus notas. Así también a los docentes les

resulta mucho más simple ejercer su labor, ya sea en un aula o a través de la pantalla de un ordenador.

En definitiva y empleando los conocimientos adquiridos durante los estudios, la motivación de este proyecto es mejorar y favorecer la enseñanza de las clases, tanto en formato virtual como en un escenario mixto, concretamente en la asignatura optativa de Visión por Computador.

1.3 - Objetivos

El objetivo principal del presente proyecto es generar material para la asignatura de Visión por Computador, desarrollando un entorno colaborativo dentro del ámbito de la computación interactiva, donde los estudiantes podrán modificar a su antojo la resolución de los ejercicios planteados y ver los cambios en los resultados obtenidos en tiempo real.

Puesto que el aprendizaje real en las clases depende de la habilidad de los docentes para mantener y mejorar la motivación de los estudiantes, esto ayudará a comprender mejor el temario de la asignatura e incluso a incrementar el interés por la misma. Será útil y sencillo para trabajar tanto en clase como en casa, teniendo todo lo necesario a su disposición.

Dentro de este entorno colaborativo se podrá destacar la variedad en el uso de tecnología docente, la claridad del material y los objetivos de la asignatura, ideas estimulantes, la participación activa de los estudiantes, el uso de ejemplos apropiados, concretos y entendibles para la retención de los conocimientos, etc.

1.4 - Estructura de la memoria

Este proyecto consta de una estructura formada por un total de 7 capítulos. En primer lugar y en el que nos encontramos, está el capítulo *INTRODUCCIÓN*. En él se hace una breve introducción sobre el proyecto, su justificación o motivación de por qué se ha realizado, los objetivos que tiene y su estructura.

En segundo lugar, está el capítulo *ESTADO DEL ARTE*, donde se hace referencia a las metodologías innovadoras y a los diversos tipos empleados en

los centros educativos, el aprendizaje mixto y las herramientas mencionadas en este proyecto; Matlab y Jupyter Notebook.

En tercer lugar, se encuentra *JUPYTER NOTEBOOK*, un capítulo en el que se habla de la instalación de la herramienta, de su entorno, de sus diversos usos y aplicaciones y, además, de cómo exportar los documentos elaborados a otro tipo de formatos.

En cuarto lugar, tenemos el capítulo de *PRÁCTICAS DE VISIÓN POR COMPUTADOR*. En él se explica el contenido de cada práctica de la asignatura y una breve introducción de las funciones que se utilizan en Matlab.

El capítulo *RESULTADOS* se posiciona en el quinto lugar. En este capítulo se explica de nuevo cada práctica de la asignatura, esta vez haciendo uso de la herramienta de Jupyter Notebook. Se habla de forma detallada de las funciones utilizadas en cada caso y se adjuntan figuras donde se pueden ver mejor los resultados obtenidos.

En sexto lugar, está el capítulo de *CONCLUSIONES*, donde se hace referencia al trabajo realizado personalmente para el desarrollo de este proyecto. También, una breve opinión personal sobre cómo puede beneficiar este trabajo a los alumnos de los siguientes cursos.

En séptimo y último lugar, se encuentra la *BIBLIOGRAFÍA*, la cual ha sido empleada para facilitar la elaboración del presente proyecto.

2 - ESTADO DEL ARTE

La modernización de los sistemas educativos ha sido inevitable debido a las tendencias educativas surgidas a partir de los procesos de globalización, transformando así la visión de la educación. Esto ha generado cambios en los procesos de formación, dando mayor importancia a un conjunto de conocimientos, capacidades, actitudes y destrezas que son esenciales para responder a las necesidades educativas de la sociedad actual.

Por esta razón, se requiere de docentes que sean capaces de proporcionar mejores recursos de aprendizaje y de mayor eficacia mediante la implementación de herramientas tecnológicas, con el fin de permitir a los estudiantes algunas habilidades como el reconocimiento de sí mismos, la interacción con otros y, además, la comprensión de su entorno físico y social.

Por otro lado, con el estallido de la pandemia de COVID-19 se han tomado algunas medidas necesarias para mantener las clases en diferentes niveles de formación, como la adopción de estrategias para garantizar la continuidad de las clases basadas en estrategias sincrónicas y asincrónicas, próximas a la llamada educación a distancia o educación no presencial.

Es por estos motivos que el presente proyecto es un breve ejemplo de cómo el cambio en la docencia se ha visto reflejado. En la asignatura optativa de Visión por Computador la herramienta docente empleada es Matlab, y la herramienta que se pretende implementar para ser utilizada en años posteriores es Jupyter Notebook. Dos herramientas muy útiles para aprender y motivar a los estudiantes a querer saber más sobre la asignatura.

2.1 - Metodologías innovadoras

Frente a las metodologías tradicionales, con estudiantes cuya función es escuchar de forma pasiva para aprender las lecciones a base de memorización, se apuesta por el trabajo en equipo y la resolución de problemas basados en situaciones reales gracias a las metodologías activas.

Como resultado se obtiene un incremento en la motivación de los estudiantes que, además, participan de forma activa mejorando su comprensión y aprendizaje, además del desarrollo de sus habilidades y pensamientos críticos a base de compromiso, creatividad e investigación.

Algunas de las metodologías seleccionadas para este estudio son: clase invertida, aprendizaje basado en proyectos, aprendizaje cooperativo, gamificación, aprendizaje basado en problemas, design thinking, aprendizaje basado en el pensamiento, aprendizaje basado en competencias y, por último, aprendizaje colaborativo.

2.1.1 - Clase invertida

Con el modelo de la clase invertida o Flipped Classroom se consigue una diversidad en el aprendizaje semipresencial, teniendo como objetivo lograr que los estudiantes aprendan a gestionar su aprendizaje trabajando de manera colaborativa e incluso, interactuando con material audiovisual.

El modelo tradicional de enseñanza se basa en la trasmisión de la información desde el docente a los estudiantes, mientras que el objetivo del modelo de la clase invertida es proporcionar una experiencia de aprendizajes autónomos utilizando recursos de multimedia fuera de clase. En la *figura 2.1* vienen representados de forma esquemática los procesos de ambos modelos.

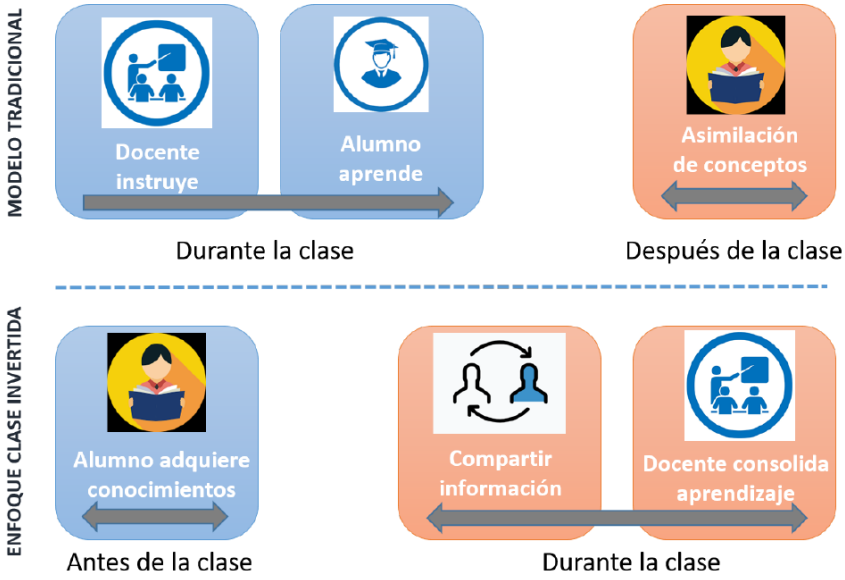


Figura 2.1 Modelo tradicional y clase invertida

El estudiante es quien gestiona su aprendizaje y, por ello, el tiempo de clase presencial es usado para debatir y trabajar puntos clave, así como cualquier cuestión o dificultad que pueda tener al respecto.

Mientras el profesor ejerce el papel de guía y tutor del grupo de estudiantes, ellos adoptan un rol activo en clase. Se requiere que los alumnos lean y asimilen en casa el temario o vean algunos videos que hayan sido previamente seleccionados por el docente, en su propio entorno y, además, que sean capaces de mantener una comunicación fluida con otros alumnos y profesores mediante redes sociales, foros o email.

2.1.2 - Aprendizaje basado en proyectos

El aprendizaje basado en proyectos (ABP) es un método pedagógico con el que se busca involucrar a los estudiantes de una manera activa en su aprendizaje. Tienen que investigar la respuesta a alguna pregunta o problema del mundo real y luego creer una solución determinada.

De esta forma se hacen más evidentes los beneficios en el alumnado en la capacidad de retener conocimiento y en la oportunidad de desarrollar competencias complejas como la comunicación, la colaboración o la capacidad de analizar y evaluar la consistencia de los razonamientos.

2.1.3 - Aprendizaje cooperativo

Los docentes utilizan esta metodología para agrupar a los estudiantes y, así, impactar en el aprendizaje de una manera positiva. Esto mejora entre otras cualidades, la atención, la adquisición de conocimientos y la implicación por parte de los alumnos.

Esta metodología se estructura en base a la formación de equipos de entre 3-6 alumnos, donde cada uno tiene un papel determinado y para alcanzar los objetivos deben interactuar y trabajar de forma conjunta y coordinada.

En el aprendizaje individual, el alumno se centra mayormente en conseguir sus objetivos sin tener que depender del resto de sus compañeros. En cambio, en el aprendizaje cooperativo el objetivo final es siempre común y se comprueba que cada uno de los miembros haya realizado correctamente sus tareas.

2.1.4 - Gamificación

Con esta técnica de aprendizaje se consigue transportar la mecánica de los juegos al entorno educativo-profesional. El objetivo es conseguir unos resultados más beneficiosos para aprender mejor algunos conocimientos o habilidades, o incluso, recompensar acciones concretas.

La gamificación gana terreno entre las metodologías de formación gracias a su carácter lúdico. Dicho carácter facilita la asimilación de conocimientos de una forma más divertida, lo que produce a los estudiantes una experiencia positiva.

Como recompensa por los objetivos alcanzados por el estudiante se utilizan una serie de técnicas mecánicas. Entre las más utilizadas podemos encontrar las mostradas en la *Figura 2.2*. Por otro lado, para hacer mención a la motivación del estudiante para jugar y seguir adelante en la obtención de sus objetivos se emplean unas técnicas dinámicas. Estas podemos visualizarlas en la *Figura 2.3*.

Es importante saber que dependiendo de la dinámica que se persiga, se deberán explotar unas técnicas mecánicas más que otras.

Figura 2.2 Técnicas mecánicas de la gamificación

Figura 2.3 Técnicas dinámicas de la gamificación

2.1.5 - Aprendizaje basado en problemas

Este proceso de aprendizaje, también denominado con las siglas ABP, es un ciclo formado por muchas etapas variadas, comenzando por la elaboración de preguntas y la adquisición de conocimientos que, a su vez, llevan a otras preguntas de mayor complejidad.

Algunas de las ventajas que tiene esta metodología pueden ser:

- El desarrollo de competencias creativas y pensamiento crítico.
- Habilidades más desarrolladas para la resolución de problemas.
- Una mayor motivación del alumno.
- Mayor capacidad para transferir conocimientos a otras situaciones.

El aprendizaje basado en problemas supone el ejercicio de investigación por parte de los alumnos y, además, convertirlo en datos e información útil.

2.1.6 - Design thinking

El Design Thinking (DT) o “Pensamiento de diseño”, permite identificar con mayor exactitud los problemas de cada alumno de manera individual y, además, generar en la experiencia educativa la creación y la innovación hacia la satisfacción de los demás, que más tarde se vuelve simbiótica[1].

Para integrar este método se deben de seguir los 5 pasos que vienen reflejados en la *Figura 2.4* Se tratan de un liderazgo con empatía, definir problemáticas y desafiar las suposiciones tradicionales, idear experimentos con consecuencias reales, prototipo y concretizar sobre las ideas planteadas y, por último, sacar conclusiones y comprobar resultados.

[1] Simbiosis. Relación de ayuda o apoyo mutuo que se establece entre dos personas o entidades, especialmente cuando trabajan o realizan algo en común.

Figura 2.4 Pasos del Design Thinking

2.1.7 - Aprendizaje basado en el pensamiento

Gracias a que es capaz de estimular en el alumnado la capacidad de llevar a cabo un aprendizaje más consciente y profundo que cambia la manera en la que aborda la información recibida, el Thinking based Learning (TBL) o Aprendizaje basado en el pensamiento, es una de las metodologías empleadas más populares en el ámbito educativo.

Los docentes tienen como labor animar a los alumnos a utilizar los nuevos hábitos mentales, sus habilidades de pensamiento y la metacognición[2]. Gracias a ello, los estudiantes son capaces de transformar su experiencia de aprendizaje, pasando de la memorización a la comprensión profunda de los conceptos, lo que les permite poder relacionar las ideas con mayor facilidad.

Además de desarrollar estas aptitudes de manera individual, lo hacen también de forma global. Esto conlleva a ciertas ventajas como mejorar su pensamiento crítico y creativo, su capacidad analítica e incluso, su inteligencia emocional cuando aprenden a escuchar de manera activa, a dominar sus propias emociones y a empatizar.

2.1.8 - Aprendizaje basado en competencias

El aprendizaje basado en competencias es una metodología que se centra en la demostración de los resultados de aprendizaje, basándose principalmente en la progresión del estudiante por medio de los planes de estudio a su correspondiente ritmo, profundidad, etc.

Los estudiantes no pueden continuar hasta que hayan demostrado los resultados de aprendizaje, lo que supone su enfoque en el dominio, la principal característica de este aprendizaje.

Por definición, el aprendizaje basado en competencias representa un conjunto de estrategias para lograr la adquisición de conocimiento, el desarrollo de habilidades y la solidificación de hábitos de trabajo. En la *Figura 2.5* viene representado en un esquema el funcionamiento de este tipo de aprendizaje.

[2] Metacognición. Se refiere a la capacidad de las personas para reflexionar sobre sus procesos de pensamiento y la forma en que aprenden.

Figura 2.5 Aprendizaje basado en competencias

2.1.9 - Aprendizaje colaborativo

Debido a que cada vez es más común el trabajo y las dinámicas de equipo, es muy útil aplicar el aprendizaje colaborativo en las clases. Este método consiste principalmente en el desarrollo cognitivo de los estudiantes haciendo así que se desarrolle favorablemente la interacción entre las personas. Además, es capaz de incrementar la integración entre los estudiantes de diferentes culturas, religiones y costumbres.

Entre sus beneficios podemos encontrar:

- Combate la ansiedad. Al permitir que los estudiantes se relajen y trabajen en un ambiente armonioso, pensando, ensayando y apoyándose entre ellos, se consigue que la ansiedad se reduzca.
- Permite optimizar la enseñanza. Los centros educativos son capaces de maximizar todos los recursos de los que están dotados gracias a este método, de forma que el proceso de enseñanza se ve optimizado.
- Desarrolla la independencia. Ante cualquier duda o problema que los estudiantes tengan, los compañeros se ofrecen a dar la ayuda que necesitan, por lo que la dependencia que tienen con el docente se reduce considerablemente.

- Potencia el pensamiento crítico. Cuando los estudiantes trabajan en un ambiente colaborativo, aprenden a proyectar mejor sus propios pensamientos e incluso, a reflexionar y desarrollar algunas habilidades metacognitivas.
- Responsabilidad individual. En un equipo, cada integrante es responsable de forma individual de perseguir los objetivos del grupo. Además, cada uno tiene las mismas oportunidades para participar y el mismo grado de responsabilidad y protagonismo que el resto de los componentes.
- Contribuye a la interdependencia positiva. En todos los colectivos hay estudiantes mejor preparados y otros que lo están, pero en menor medida. Es por ello que los segundos se pueden aprovechar y aprender del conocimiento de los que están mejor preparados y estos, a su vez, enriquecerse y fortalecer sus capacidades y habilidades.
- Responde a una sociedad heterogénea y multicultural. Se consigue gestionar de forma adecuada una clase con estudiantes de diferentes culturas, permitiendo de esta manera el desarrollo de las habilidades intelectuales e incrementando la capacidad de expresión y comunicación entre ellos.

2.2 - Aprendizaje mixto

Ya descritas algunas de las metodologías innovadoras que se han ido incorporando en los centros educativos para mejorar el desarrollo de las clases, daremos paso a hablar sobre el aprendizaje mixto, una combinación entre las tecnologías de aprendizaje moderno y los métodos de aprendizaje tradicionales.

La educación siempre ha estado en continuo crecimiento, dando lugar a cambios para favorecer tanto a los docentes como a los alumnos. Actualmente y de forma inevitable, ha pasado por una etapa que probablemente haya marcado un antes y un después en la historia de la enseñanza. Debido a la pandemia ocasionada por el COVID-19, todos los centros educativos se vieron obligados a cerrar sus puertas e impartir, en su lugar, las clases de forma virtual.

Este tipo de enseñanza ha tenido sus ventajas y desventajas. En el caso de los docentes, han tenido que adaptarse a una nueva forma de enseñar que, a algunos, les ha resultado un poco complejo. Estaban acostumbrados a la clase tradicional donde los alumnos se sientan ante ellos y escuchan, pero ahora, han tenido que adaptarse a enseñar las lecciones a través de una pantalla sin tener la seguridad de que los atienden. En el caso de los estudiantes, en ciertas situaciones el agobio y el estrés ha aumentado considerablemente por el hecho de tener que aprender nuevos conocimientos cuando los métodos docentes utilizados eran mínimos. Esto se ha visto agravado cuando la asignatura en concreto era muy práctica.

Muchos de los recursos utilizados han sido programas didácticos que han facilitado la enseñanza. Algunos de ellos tenían la particularidad de que tanto el docente como los estudiantes podían escribir en un mismo fondo en blanco, pudiendo cambiar el color del trazo para destacar los contenidos más importantes de las explicaciones.

Otros, en cambio, eran programas que utilizaban un lenguaje de programación para el análisis iterativo y los procesos de diseño, expresando las matemáticas de matrices y arrays directamente. Uno de ellos era Matlab, un programa muy utilizado en la asignatura de Visión por computador.

Aunque el uso de estos programas ha facilitado la impartición de las clases en un entorno virtual, con el progreso de la pandemia se tomó la medida del aprendizaje mixto. De esta forma, las clases virtuales se han adaptado para la teoría y las clases presenciales para las prácticas, consolidando así los conocimientos aprendidos.

2.3 - Herramientas

Como se ha mencionado anteriormente, uno de los programas utilizados en Visión por computador ha sido Matlab. Aunque es un programa muy útil para esta asignatura, con el objetivo de seguir mejorando y beneficiando los resultados de los estudiantes, se pondrá en práctica la implementación de la misma en Jupyter Notebook.

2.3.1 - Matlab

El nombre de Matlab viene de la abreviatura de Matrix Laboratory (laboratorio de matrices). Fue creado en 1984 por The MathWorks y nace para suministrar un fácil acceso al software matricial desarrollado por LINPACK y EISPACK[3].

Este programa ofrece un lenguaje de programación y un entorno de computación y desarrollo interactivo. Gracias a la integración de análisis numéricos, cálculo de matrices, procesos de señales y visualizaciones gráficas, es capaz de llevar a cabo proyectos cuyo objetivo son los cálculos matemáticos y la visualización gráfica de los mismos.

También dispone de programas de apoyo especializados que se denominan Toolboxes. Estos cubren la mayoría de áreas en el mundo de la ingeniería y la simulación, destacando el ‘toolbox’ de proceso de imágenes, redes neuronales, señal, diseño de sistemas de control, estadística, etc. De esta forma se proporciona a los estudiantes una ayuda a la hora de resolver los problemas más complejos con los que se puedan encontrar.

[3] LINPACK y EISPACK. Bibliotecas cuyo propósito era resolver sistemas de ecuaciones y problemas de valores propios.

En la asignatura de Visión por computador se trabaja principalmente con 3 de sus Toolboxes, las cuales son:

- Image Acquisition
- Image Processing
- GUIDE

Image Acquisition Toolbox proporciona bloques y funciones para enlazar cámaras a Matlab. Además de detectar y configurar de forma interactiva las propiedades del hardware, es capaz de generar código para automatizar la adquisición en futuras sesiones. Procesamiento in-the-loop[4], activación de hardware, adquisición en segundo plano y sincronización de la adquisición mediante varios dispositivos son algunos de sus modos de adquisición.

Image Processing Toolbox proporciona un conjunto de algoritmos estándar de referencia y apps de flujo de trabajo para el procesamiento, el análisis y la visualización de imágenes y el desarrollo de algoritmos. Es capaz de realizar la segmentación, mejora y registro de imágenes, transformaciones geométricas, reducción de ruido y procesamiento de imágenes 3D. Las apps además de procesar conjuntos de datos de forma interactiva, segmentan datos y comparan técnicas de registro de imágenes. Por otro lado, con el empleo de las funciones de visualización, exploran imágenes, volúmenes 3D y vídeos, ajustan el contraste, crean histogramas y manipulan regiones de interés (ROI).

GUIDE es el entorno de desarrollo de GUI o también, interfaces de usuario, y proporciona herramientas para su diseño. Es capaz de generar de forma automática el código para elaborar la interfaz, la cual se puede modificar posteriormente. Para tener un mayor control sobre el diseño y el desarrollo, es posible agregar controles de interfaz de usuario, como controles deslizantes o botones; y contenedores, como grupos de botones y paneles; y cuadros de diálogo.

[4] Hardware in-the-loop. Técnica en la que las señales reales de un controlador son conectadas a un sistema de pruebas que simula la realidad.

2.3.2 - Jupyter Notebook

Gracias al entorno de trabajo interactivo de Jupyter Notebook, se permite desarrollar código en Python de forma dinámica. Es posible integrar tanto bloques de código como texto, gráficas o imágenes en un mismo documento. Esto beneficia en gran nivel a los estudiantes porque ven y comprenden con mayor facilidad los resultados de los ejercicios propuestos en clase, además pueden ir variando los ejemplos para afianzar mucho mejor los conocimientos adquiridos.

Este entorno es muy útil para la asignatura de Visión por computador, para el aprendizaje automático y el procesamiento de imágenes. Esto es debido a que se puede trabajar con la biblioteca de código abierto OpenCV, utilizada tanto por empresas como por universidades. Su objetivo es procesar imágenes y videos para identificar rostros, objetos o incluso la escritura a mano de un humano. Además, es capaz de procesar la estructura de la matriz OpenCV para su análisis y, usando el espacio vectorial y realizando operaciones matemáticas, es posible identificar el patrón de la imagen y sus diferentes características.

Es por ello que este proyecto se basa en la implementación de la parte práctica de la asignatura Visión por computador con Jupyter Notebook, para favorecer el nivel de los estudiantes gracias a su facilidad y comodidad de trabajo.

3 - JUPYTER NOTEBOOK

Como se ha mencionado anteriormente, Jupyter Notebook es un entorno de código abierto que permite integrar texto, video, audio, imágenes y la ejecución de código a través del navegador en múltiples lenguajes. Esta ejecución se realiza mediante la comunicación con un Kernel o núcleo de cálculo. Aunque en un principio se utilizaba únicamente el núcleo de cálculo Python, se han ido incluyendo otros como por ejemplo Octave, Julia, R, Haskell, Ruby, C/C++, Fortran, Java, SageMath, Scala, Matlab y Mathematica. Así, su nombre aparece al unir 3 de los lenguajes más utilizados de programación de código abierto: Ju-lia, Py-thon y R.

Esta cualidad ha permitido un crecimiento tanto en el ámbito docente como investigador. Utilizando este tipo de cuadernos se pueden generar lecciones o material de apoyo ricos en contenido, así como material de trabajo colaborativo. Dado que resulta inevitable que un estudiante del área de ciencias necesite trabajar con algún tipo de programa de cálculo a lo largo de su carrera universitaria, o en su futuro laboral, la integración de este entorno permite a los estudiantes adquirir una competencia transversal de alto nivel para su formación. Además, proporciona otro nivel más de profundización en los contenidos de la asignatura, que puede ser explorado por aquellos estudiantes con una mayor motivación.

A lo largo de este capítulo veremos las características generales del entorno de Jupyter Notebook, su instalación y uso para proporcionar lecciones y ejercicios útiles en estudios científicos. Además, se describirán varias opciones para proporcionar un servidor de este tipo de documentos accesible por los estudiantes a través del ordenador.

3.1 - Instalación

Uno de los requisitos para instalar Jupyter Notebook es disponer de Python. Aunque los cuadernos permiten la ejecución en otros lenguajes de programación, algunas de las funciones de Jupyter están implementadas en este lenguaje. Dada su enorme popularidad en el ámbito científico, ha aparecido en los últimos años una gran cantidad de módulos de Python donde la más conocida es Anaconda. Este pack de módulos de Python dispone a su vez de un gestor de paquetes y facilita enormemente la instalación y gestión de librerías de Python relacionadas con aplicaciones científicas.

Jupyter Notebook es uno de los módulos que por defecto Anaconda instala, por lo que la instalación recomendada, independientemente del sistema operativo, es instalar dicha distribución de Python científico. Para ello, la página de descarga de Anaconda permite obtenerlo de forma gratuita. Siguiendo los correspondientes pasos de instalación, podemos tenerlo disponible para posteriormente ejecutarlo desde la consola del ordenador escribiendo *jupyter notebook* o directamente desde el navegador de Anaconda (*Figura 3.6*).

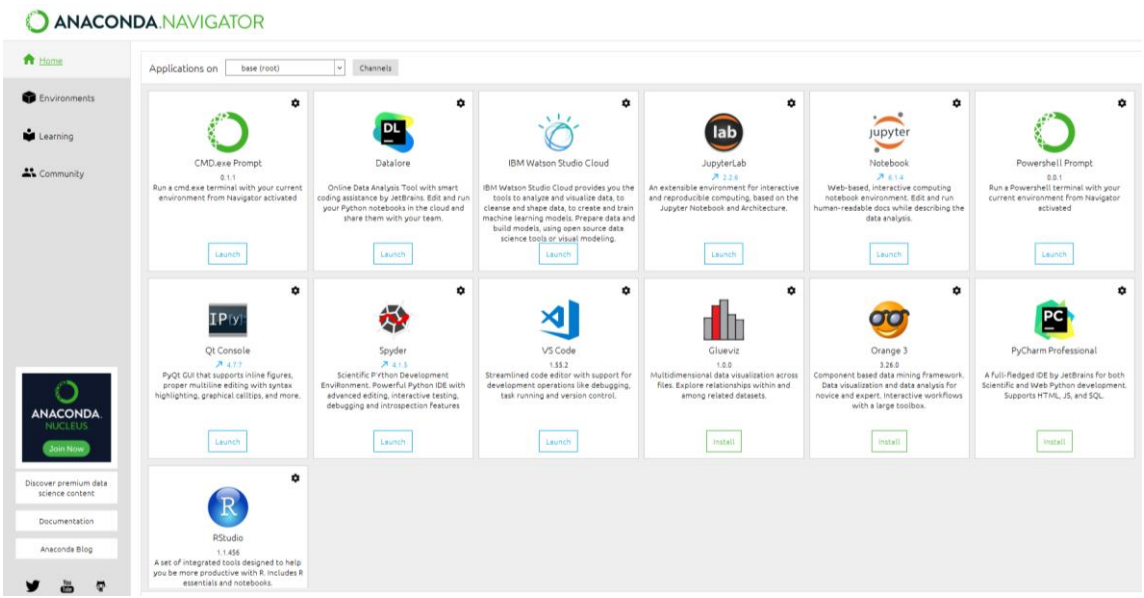


Figura 3.6 Navegador de Anaconda

3.2 - Descripción del entorno

Al ejecutar Jupyter Notebook en el ordenador, aparecerá en nuestro navegador una primera página denominada Jupyter Dashboard. Como podemos ver en la *Figura 3.7*, en esta página tenemos la lista de archivos disponibles en la carpeta en donde hayamos ejecutado nuestro programa. Un elemento importante es el botón para subir archivos al servidor, que aparece en la esquina superior derecha. En caso de ejecutarse Jupyter Notebook localmente, este botón no presenta una gran funcionalidad, pero sí para el caso en que accedamos a estos cuadernos desde otro ordenador y a través del navegador.

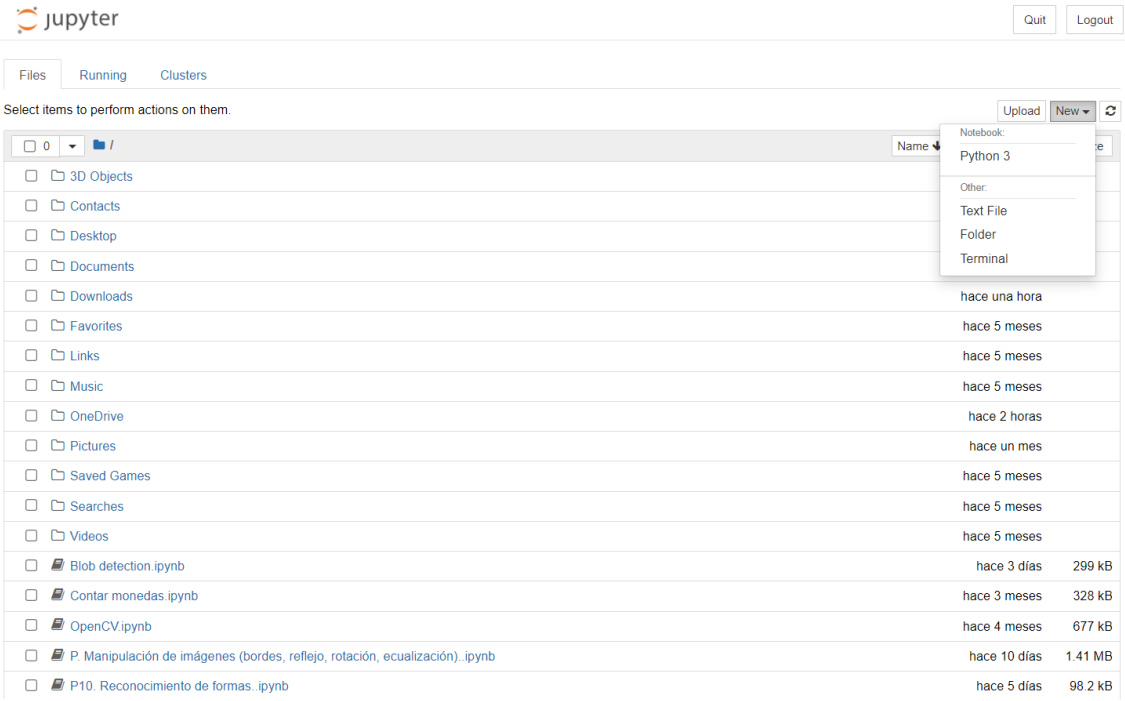


Figura 3.7 Entorno de Jupyter Notebook

Para generar un primer Jupyter Notebook, pinchamos en New y elegimos el núcleo de cálculo que queremos utilizar entre aquellos que tengamos instalados. Por defecto, únicamente tenemos Python. Un ejemplo de un documento tipo Jupyter Notebook puede visualizarse en la *Figura 3.8*.

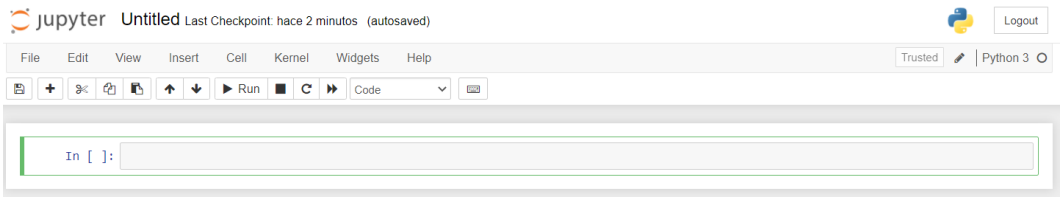


Figura 3.8 Cuaderno de Jupyter Notebook

En la parte superior del documento disponemos de los distintos menús para controlar tanto la edición, importando bibliotecas o incrustando widgets (elementos interactivos); como para descargar el documento en varios formatos (por ejemplo, PDF) así como insertar o borrar partes del documento. También incluye la elección y el control del núcleo de cálculo y un botón de ayuda que cubre gran parte de las características de Jupyter Notebook.

El cuaderno se divide en distintas celdas, cuyo comportamiento puede ser seleccionado entre las siguientes opciones: título, de diversos tamaños; texto, utilizando el lenguaje Markdown; o código. Por otro lado, las librerías específicas permiten importan imágenes o vídeos, incluir enlaces a recursos web o incluso incrustar una página web completa dentro del documento.

Es importante remarcar que en Jupyter tenemos dos modos de trabajo: edición y comando.

- Edición. La celda que tenemos seleccionada se muestra en verde (*Figura 3.9*) y, como si fuera un editor de texto, permite modificar el contenido de las celdas. Seleccionando una celda y pulsando *Enter* también podemos entrar en este modo de trabajo.

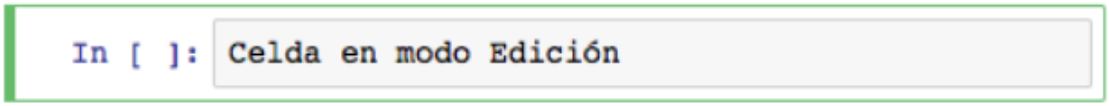


Figura 3.9 Celda en modo edición

- Comando. La celda seleccionada se muestra en azul (*Figura 3.10*) y, en este caso, nos permite modificar el cuaderno y su estructura o ejecutar celdas. Seleccionando una celda y pulsando *Esc* podemos volver a este modo de trabajo.

Figura 3.10 Celda en modo comando

Para ejecutar una celda podemos pulsar *Control* + *Enter* o, si deseamos pasar directamente a la siguiente celda después de ejecutarla, *Shift* + *Enter*. En el caso de ejecutar una celda e insertar una nueva, pulsamos *Alt* + *Enter*, y, si queremos insertar una nueva celda encima o debajo de la que tenemos seleccionada, pulsamos *a* y *b* respectivamente. Si queremos saber cuándo una celda se ha ejecutado, simplemente tenemos que mirar la cabecera de la misma.

- Si aparece **In []**, quiere decir que todavía no se ha ejecutado.
- Si tenemos **In [*]**, el proceso está en ejecución.
- Si entre los corchetes tenemos un número, por ejemplo: **In [1]**; quiere decir que ya ha terminado de ejecutarse.

Para insertar texto con formato, la opción elegida por Jupyter Notebook ya mencionada anteriormente, es utilizar el lenguaje Markdown. De este modo, se pueden incluir listas, texto en negrita o cursiva, tablas o imágenes, siendo las opciones de formato más utilizadas las siguientes:

- Texto en negrita/cursiva. Este tipo de texto se indica entre dos pares de asteriscos. De este modo ***cuaderno*** aparecerá como **cuaderno**. Por otro lado, el texto en cursiva se indica entre dos pares de asteriscos simples, así **cuaderno** aparecerá como *cuaderno*.
- Listas. Las listas se realizan indicando un asterisco o un número seguido de un punto si se desean listas numeradas. Su organización se realiza automáticamente por Markdown asignándole el número correcto a cada ítem.
- Inclusión de imágenes. La sintaxis para incluir imágenes es *! [nombre alternativo] (dirección de la imagen)* en donde el nombre alternativo aparecerá en caso de que no se pueda cargar la imagen y la dirección se puede referir a un enlace en Internet o a una imagen local.
- Inclusión de código HTML. En caso de necesitar un mayor control del formato, se puede incluir en HTML directamente, teniendo en cuenta que el lenguaje que utiliza Markdown es un subconjunto del lenguaje HTML.
- Enlaces. Los enlacen pueden derivar, tanto a otras partes del documento, como a otros archivos locales o incluso a páginas en internet. Su sintaxis es *[texto] (dirección del enlace)*.

- Fórmulas matemáticas. Para indicar cualquier tipo de fórmula y expresión matemática es posible, gracias al empleo de MathJax, incluir código en LaTeX. Escribiendo entre símbolos del dólar ($...$) tendremos las fórmulas dentro de una línea de texto, mientras que utilizando los símbolos del dólar dobles ($$...$$) tendremos las expresiones separadas del texto.

Estas posibilidades permiten utilizar celdas de texto documentando el código de otras celdas como, por ejemplo, explicando de forma detallada algún concepto teórico, del que después se puede presentar un ejercicio. Es importante destacar que el lenguaje seleccionado es el que se utiliza en todo el documento, ya que mezclar dos lenguajes de programación actualmente no es posible.

3.3 - Usos y aplicaciones

Jupyter Notebook, como ya se ha mencionado, proporciona un entorno para el flujo de trabajo de la simulación numérica y la ciencia de datos. En un mismo documento, los estudiantes pueden escribir, documentar y ejecutar código, visualizar datos, realizar problemas y comparar los resultados. Cabe destacar que, como el código se organiza en celdas independientes, es posible ejecutar y analizar bloques concretos de código de forma individual.

Dentro de los principales usos que tiene Jupyter Notebook, podemos recalcar los siguientes:

- Modelización estadística. Se trata de un método matemático capaz de estimar la probabilidad de distribución de una característica determinada.
- Depuración de datos. Al ejecutar un análisis de big data es posible distinguir entre los datos que son importantes y los que no lo son.
- Creación y preparación de modelos de aprendizaje automático. Diseño, programación y preparación de los modelos.
- Visualización de datos. Representaciones gráficas de los datos para visualizar con mayor claridad las tendencias, las interdependencias, los patrones, etc.

3.4 - Exportar a otros formatos

Aunque los documentos generados por Jupyter Notebook son fácilmente accesibles a través de un navegador, en ocasiones es necesario o útil disponer de otros formatos. Jupyter dispone de la herramienta **nbconvert**, que facilita la conversión permitiendo exportar los documentos hechos en Jupyter Notebook a otros formatos. Estos pueden ser: HTML, LaTeX, Markdown, reStructuredText, script de Python ejecutable y, presentación. Esta herramienta proporciona la conversión a documentos estáticos donde las celdas no pueden ejecutarse. El uso de nbconvert requiere la instalación de un paquete denominado Pandoc, el cual se puede instalar a través de su página web.

El funcionamiento de nbconvert se puede realizar mediante una consola, o bien directamente desde el documento, a través del menú *File -> Download as…*, aunque para aprovechar todas sus capacidades, la sintaxis para su uso en una terminal es la siguiente: *"$ jupyter nbconvert –to FORMAT notebook.ipynb"*, donde FORMAT es el formato que se quiere obtener.

Si la conversión que queremos hacer es a un archivo en LaTeX, tenemos que escribir: *"$ jupyter nbconvert --to latex notebook.ipynb"*. En el caso de querer convertirlo a formato pdf de forma directa, la instrucción quedaría de la siguiente manera: *"$ jupyter nbconvert --to latex notebook.ipynb –post PDF"*.

También es posible, añadiendo *--template book*, modificar las plantillas de un archivo LaTeX a formato libro. Es importante conocer que el formato presentación que se genera es Reveal.js en HTML, por lo que para su visualización se requiere que el ordenador muestre la presentación en el navegador. Esto se cumple si a la línea anterior añadimos *--post serve*. Este tipo de presentaciones dan un número más elevado de opciones que las presentaciones tradicionales y son, además, más sencillas de compartir a través de diversas plataformas y sistemas operativos.

4 - PRÁCTICAS DE VISIÓN POR COMPUTADOR

El análisis de imágenes o Visión por computador, es la capacidad de los ordenadores de analizar imágenes que han sido capturadas por una cámara con el objetivo de obtener información sobre los objetos que se encuentren presentes en la escena.

A continuación, se explicará cada una de las prácticas que conforman la asignatura optativa de Visión por computador y, además, se hará una breve introducción de las funciones empleadas en Matlab.

4.1 - Manipulación de imágenes

Cuando hablamos de manipulación de imágenes, nos referimos a las posibilidades que nos ofrecen determinados programas informáticos para trabajar con archivos de imagen y así, modificar algunas de sus características como el brillo, el tamaño, el contraste, la saturación de color, aplicar filtros, etc.

En esta práctica aprenderemos a añadir bordes, a rotar, a representar el reflejo y a ecualizar una imagen. Para ello, primero veremos cómo leer una imagen desde nuestro programa.

4.1.1 - Leer y mostrar una imagen

Con Matlab el comando utilizado para leer una imagen es **imread.**

```
>> A = imread ('*.tif');
```

Con esta función se lee una imagen en formato tiff, pudiendo ser otro el formato de la imagen, y la almacena en una matriz llamada A. Para mostrar la imagen se utiliza la instrucción **imshow(A)**.

4.1.2 - Añadir bordes a una imagen

Para añadir bordes a una imagen con Matlab, la función que se utiliza es **padarray**. Esta función tiene como parámetros de entrada la imagen a la que se le va a añadir los bordes, la cantidad de relleno (en píxeles) que se le va a añadir a cada dimensión especificada como un vector de enteros no negativos y la

dirección en la cual se va a añadir el borde. Por ejemplo, la siguiente línea de código añade a la imagen Im un borde de 76 píxeles en ambas direcciones:

```
>> bordes = padarray (Im, [76 76], 'both');
```

4.1.3 - Rotación de una imagen

En Matlab, para rotar una imagen simplemente hay que utilizar la función **imrotate**. Sus argumentos de entrada son la imagen que se desea rotar, la cantidad de rotación en grados y el método de interpolación, siendo el más utilizado el bilineal. Hay que tener en cuenta que esta función gira la imagen en sentido contrario a las agujas del reloj alrededor de su punto central, por lo que, si queremos girarla en el sentido de las agujas del reloj, hay que especificar un valor negativo para la rotación en grados. Por ejemplo, si queremos girar la imagen Im 45º en el sentido de las agujas del reloj, bastaría con escribir la siguiente línea de código:

```
>> rotar = imrotate (Im, -45, 'bilinear');
```

4.1.4 - Reflejo de una imagen

Para obtener el reflejo de una imagen con Matlab, se utiliza la función **fliplr** cuyo argumento de entrada es la imagen original. Esta función devuelve la imagen volteada en la dirección izquierda-derecha.

4.1.5 - Ecualización de una imagen

En Matlab, con un histograma se puede ver la distribución de intensidades en una imagen, solo hay que utilizar la función **imhist**. En el caso de que nos interese ecualizar el histograma de una imagen para mejorar el contraste y, así, distribuir los valores de intensidad por todo el rango, simplemente tendríamos que utilizar la función **histeq** para extender los valores de intensidad.

4.2 - Reducción de ruido en una imagen digital

Las operaciones de suavizado tienen por objeto reducir el ruido y/o efectos espurios que pueden presentarse en una imagen a consecuencia del proceso de captura, digitalización y transmisión. Normalmente es necesario utilizarla antes de aplicar un detector de bordes a una imagen.

Existen distintos tipos de algoritmos que permiten la reducción de ruido. A continuación, veremos algunos filtros lineales (convolución de una imagen con una máscara predefinida) como el filtro de la media y el filtro gaussiano; y no lineales (operación no lineal con los píxeles del entorno de vecindad) como el filtro de la mediana.

4.2.1 - Filtros lineales

Como se ha mencionado anteriormente, en los filtros lineales se realiza una operación de convolución entre la imagen que se quiere filtrar y una máscara. Tienen que ser utilizados con cierta precaución, ya que el mayor inconveniente que tienen estas técnicas es el enturbiamiento que producen en la imagen, provocando así un difuminado en los bordes.

Filtro de la media

El procedimiento se basa en dada una imagen f(i,j), generar una nueva imagen g(i,j) cuya intensidad de cada píxel se obtiene con el promedio de los valores de intensidad de los píxeles f(i,j) incluidos en un entorno de vecindad que ha sido definido con anterioridad.

En Matlab la función que permite realizar un filtro de la media es **imfilter**. Esta función tiene la siguiente estructura:

```
>> media = imfilter (A, h, option1, option2, …);
```

Su objetivo es filtrar el array A con el filtro multidimensional h. Por otro lado, los parámetros option1, option2, …, son opciones de tamaño del array de salida, de frontera, de convolución o de correlación.

La función de Matlab que permite generar el filtro h es **fspecial**. Esta función crea filtros bidimensionales del tipo especificado por type. El argumento type puede tener como valor un detector de bordes de sobel o prewitt, un filtro pasa baja gaussiano, un operador laplaciano, un filtro de la media, etc. Además, el segundo argumento de entrada depende del tipo de filtro.

```
>> h = fspecial (type, parameters);
```

Antes de aplicar el filtro de la media es necesario añadir ruido a la imagen. El ruido que se puede introducir es variado, pudiendo ser ruido aleatorio del tipo sal y pimienta, ruido gaussiano, ruido multiplicativo, etc.

Para añadir ruido a una imagen se utiliza la función **imnoise**.

```
>> pimienta = imnoise (Im, type, parameters);
```

Es fundamental conocer la importancia que tiene el tamaño de la máscara que se aplique, ya que cuanto mayor sea esta se consigue una mayor reducción del ruido, pero a la vez se produce una mayor difuminación de los bordes.

Filtro gaussiano

El resultado es el valor de cada punto obtenido de promediar con distintos pesos los valores vecinos a ambos lados de dicho punto. Este filtro también tiene el inconveniente del difuminado de los bordes, aunque no es tan evidente como en el caso de la media. Este tipo de filtro reduce especialmente el ruido de tipo gaussiano.

El ruido gaussiano tiene su origen en diferencias de ganancias del sensor, ruido en la digitalización, etc. y causa pequeñas variaciones en la imagen.

4.2.2 - Filtros no lineales. Filtro de la mediana

En la nueva imagen se generan los píxeles calculando la mediana del conjunto de píxeles del entorno de vecindad del píxel correspondiente en la imagen origen. De esta forma se homogeneizan los píxeles de intensidad muy diferente con respecto a la de los vecinos. Este tipo de filtro es bastante utilizado cuando se tiene ruido aleatorio.

En Matlab la instrucción empleada para realizar el filtro de la mediana es **medfilt2**, donde Im es la matriz de entrada a la que se le aplica el filtro de la mediana utilizando por defecto una vecindad de 3X3.

```
>> mediana = medfilt2 (Im);
```

4.3 - Detección de bordes en una imagen

Cuando hablamos de las transiciones entre dos regiones de niveles de gris significativamente distintos, nos referimos a los bordes de una imagen digital. Facilitan una información bastante importante sobre las fronteras de los objetos y puede ser utilizada para segmentar la imagen, reconocer objetos, etc.

La mayoría de las técnicas que se utilizan para detectar bordes emplean operadores locales basados en diferentes aproximaciones discretas de la primera y segunda derivada de los niveles de grises de la imagen.

4.3.1 - Operadores basados en la primera derivada (Gradiente)

La derivada de una señal continua proporciona las variaciones locales con respecto a la variable, de forma que cuanto más rápidas son estas variaciones, mayor es el valor de la derivada.

En el caso de funciones bidimensionales f(x,y), la derivada es un vector que apunta en la dirección de la máxima variación de f(x,y) y cuyo módulo es proporcional a dicha variación. Este vector se denomina gradiente y se define:

$$\nabla f(x,y) = \begin{bmatrix} \frac{\partial f(x,y)}{\partial x} \\ \frac{\partial f(x,y)}{\partial y} \end{bmatrix} \quad (1)$$

$$Mag[\nabla f(x,y)] = \sqrt{\left(\frac{\partial f(x,y)}{\partial x}\right)^2 + \left(\frac{\partial f(x,y)}{\partial y}\right)^2} \quad (2)$$

$$\theta = arctag \frac{\frac{\partial f(x,y)}{\partial x}}{\frac{\partial f(x,y)}{\partial y}} \quad (3)$$

En el caso bidimensional discreto, las diferentes aproximaciones del operador gradiente se basan en diferencias entre los niveles de grises de la imagen. La derivada parcial $f_x(x,y)$ (gradiente de fila $G_F(i,j)$) puede aproximarse por la diferencia de píxeles adyacentes de la misma fila.

$$\frac{\partial f(x,y)}{\partial x} \approx \nabla_x f(x,y) = f(x,y) - f(x-1,y) \quad (4)$$

La discretización del vector gradientes en el eje Y ($G_C(i,j)$), será:

$$\frac{\partial f(x,y)}{\partial y} \approx \nabla_y f(x,y) = f(x,y) - f(x,y-1) \quad (5)$$

Mediante la convolución de la imagen con las máscaras H_F y H_C obtenemos el gradiente de la fila G_F y de la columna G_C en cada punto:

$$G_F(i,j) = F(i,j) \otimes H_F(i,j) \quad (6)$$

$$G_C(i,j) = F(i,j) \otimes H_C(i,j) \quad (7)$$

La magnitud y orientación del vector gradiente suele aproximarse por la expresión:

$$|G(i,j)| = \sqrt{G_F^2 + G_C^2} \approx |G_F(i,j)| + |G_C(i,j)| \quad (8)$$

Roberts, Prewitt y Sobel son los operadores más utilizados.

Operador de Roberts

Las máscaras utilizadas en este operador son las mostradas en la *Figura 4.11*.

Gradiente fila

0	0	0
0	0	1
0	-1	0

Gradiente columna

-1	0	0
0	1	0
0	0	0

Figura 4.11 Máscaras de Roberts

Obtiene buena respuesta ante bordes diagonales. Ofrece buenas prestaciones en cuanto a localización. El mayor inconveniente de este operador es su excesiva sensibilidad al ruido y por tanto tiene escasas cualidades de detección.

Operadores de Prewitt y Sobel

Estos dos operadores pueden formularse de la misma manera con las máscaras de convolución que vienen en la *Figura 4.12*.

Gradiente fila

$$\frac{1}{2+K}\begin{bmatrix} 1 & 0 & -1 \\ K & 0 & -K \\ 1 & 0 & -1 \end{bmatrix}$$

Gradiente columna

$$\frac{1}{2+K}\begin{bmatrix} -1 & -K & -1 \\ 0 & 1 & 0 \\ 1 & K & 1 \end{bmatrix}$$

Figura 4.12 Máscaras de convolución

En el operador Prewitt (K = 1), para proporcionar mayor inmunidad al ruido, se involucran a los vecinos de filas/columnas adyacentes.

El operador Sobel (K = 2), se supone que es más sensible a los bordes diagonales que el de Prewitt, aunque en la práctica hay poca diferencia entre ellos.

En Matlab la función que detecta los bordes es **edge**. Con esta función se encuentran los bordes de una imagen con diferentes niveles de intensidad. Como resultado obtenemos una imagen binaria del mismo tamaño que la imagen original en la cual, “1” significa que ha detectado un borde y “0” es que no ha detectado ninguno.

```
>> bordes = edge (Im, ‘sobel’, thresh, direction);
```

El umbral de binarización se indica con el parámetro thresh. Si se elige el umbral de binarización, hay que ser consciente de que la función edge normaliza la imagen antes de procesarla, llevándola al intervalo [0,1].

Por otro lado, también se puede indicar el umbral utilizado en la detección de bordes con la función edge:

```
>> [bordes, umbral] = edge (Im, ‘sobel’, …);
```

También se podría aplicar el filtro de Sobel creando una máscara con la función **fspecial**, **imfilter**. De esta forma se obtiene una imagen de bordes en

escala de grises. Si queremos obtener tanto los bordes horizontales como los verticales simplemente tenemos que utilizar la función **imadd**.

Hay que tener en cuenta que antes de aplicar algoritmos de segmentación, la imagen de bordes ha de ser binaria. Para binarizar se pueden utilizar las funciones de Matlab **graythresh**, que halla el umbral de binarización y la función **im2bw**, que binariza la imagen.

4.3.2 - Operadores basados en la segunda derivada (Laplaciana)

Siempre que en la imagen se presente un cambio de intensidades a lo largo de una determinada dirección, existirá un máximo en la primera derivada a lo largo de dicha dirección y, consecuentemente, un paso por cero en la segunda derivada. La derivada segunda tiene la ventaja de facilitar la localización precisa del borde. Por tanto la Laplaciana (equivalente bidimensional de la segunda derivada) de una imagen f(x,y), denotado ∇^2 f(x,y), se define como:

$$\nabla^2 f(x,y) = \frac{\partial^2 f(x,y)}{\partial x^2} + \frac{\partial^2 f(x,y)}{\partial y^2} \tag{9}$$

Las aproximaciones discretas de la segunda derivada son:

$$\frac{\partial^2 f}{\partial x^2} = f(x+1,y) + f(x-1,y) - 2f(x,y) \tag{10}$$

$$\frac{\partial^2 f}{\partial y^2} = f(x,y+1) + f(x.y-1) - 2f(x,y) \tag{11}$$

De esta forma:

$$\nabla^2 f = [f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1)] - 4f(x,y) \tag{12}$$

Esta expresión se implementa a todos los puntos (x,y) de una imagen con la máscara de convolución de la *Figura 4.13*.

0	1	0
1	-4	1
0	1	0

Figura 4.13 Máscara Laplaciana

La unción **fspecial** de Matlab implementa de forma más genérica la máscara Laplaciana (*Figura 4.14*).

$$\begin{matrix} \frac{\alpha}{1+\alpha} & \frac{1-\alpha}{1+\alpha} & \frac{\alpha}{1+\alpha} \\ \frac{1-\alpha}{1+\alpha} & \frac{-4}{1+\alpha} & \frac{1-\alpha}{1+\alpha} \\ \frac{\alpha}{1+\alpha} & \frac{1-\alpha}{1+\alpha} & \frac{\alpha}{1+\alpha} \end{matrix}$$

Figura 4.14 Máscara Laplaciana más genérica

>> laplaciana = fspecial (‘laplacian’, alpha);

Resumiendo, las características principales de la Laplaciana son:

- Al ser un operador de segunda derivada es muy sensible al ruido y por tanto la detección de bordes es pobre. Para evitar esto en lugar de aplicar la Laplaciana directamente, se realiza primero un suavizado.
- Proporciona un borde de espesor unidad (un píxel).

Operador Canny

El detector de Canny es la herramienta de detección de bordes más completa implementada en la función edge. Este detector de bordes sigue los siguientes pasos:

- La imagen se suaviza con un filtro Gaussiano, con una determinada desviación estándar σ, de esta forma el ruido se reduce.
- En la dirección del gradiente se eliminan los puntos que no sean máximos locales del módulo. Desechando estos puntos se mejora la localización y se evitan detecciones falsas. Para esto se incluyen dos umbrales T1 y T2, siendo T1<T2. Si el valor de un borde es superior a T2, ese píxel del borde se etiqueta como “fuerte” y si el valor está entre T1 y T2, se etiqueta como “débil”.
- Finalmente, el algoritmo realiza una unión de los bordes, incorporando píxeles etiquetados como débiles a los píxeles etiquetados como fuertes, siempre que la conectividad sea de 8.

4.4 - Segmentación I. Transformada de Hough

La segmentación es el proceso que divide una imagen en regiones u objetos cuyos píxeles poseen atributos similares. Cada región segmentada suele tener un significado físico dentro de la imagen. Es uno de los procesos más importantes de un sistema automatizado de visión ya que permite extraer los objetos de la imagen para su posterior descripción y reconocimiento.

Las técnicas de segmentación pueden encuadrarse en tres grupos fundamentales: técnicas basadas en la detección de la frontera, técnicas de umbralización y técnicas basadas en el agrupamiento de píxeles.

En las técnicas basadas en la frontera, la segmentación de una imagen puede llevarse a cabo mediante la detección de los límites de cada región, es decir, detectando los bordes de la imagen. Las otras dos técnicas (umbralización y técnicas basadas en el agrupamiento de píxeles), enfocan la segmentación como un problema de clasificación de píxeles o grupos de píxeles, donde: píxeles de una región deben ser similares, píxeles de regiones distintas no deben ser no similares, las regiones resultantes deben tener cierto significado para el procesamiento posterior.

A continuación, veremos una técnica perteneciente al primer bloque de las técnicas de segmentación: “La transformada de Hough”.

La transformada de Hough

La transformada de Hough es una herramienta que permite detectar curvas en una imagen. Es una técnica muy robusta frente al ruido y a la existencia de huecos en la frontera del objeto. A la hora de aplicar la transformada de Hough a una imagen, es necesario obtener primero una imagen binaria de los píxeles que forman parte de la frontera del objeto.

El objetivo de la transformada de Hough para detectar rectas es encontrar puntos alineados que puedan existir en la imagen, es decir, puntos en la imagen que satisfagan la ecuación de la recta, para distintos valores de ρ y θ.

La ecuación de la recta en forma polar es la *Ecuación 13*. Por tanto hay que realizar una transformación entre el plano imagen (coordenadas x-y) y el plano o espacio de parámetros (ρ,θ).

$$\rho = x \cdot cos\theta + y \cdot sen\theta \tag{13}$$

Para aplicar la transformada de Hough es necesario discretizar el espacio de parámetros en una serie de celdas denominadas *celdas de acumulación*. Esta discretización se realiza sobre los intervalos (ρ_{min},ρ_{max}) y (θ_{min},θ_{max}).

El siguiente paso es evaluar la ecuación de la recta para cada punto de la imagen, si se cumple esta ecuación se incrementa en uno el número de votos de la celda. Un número de votos elevado indica que el punto pertenece a la recta.

Para detectar rectas con la transformada de Hough se utilizan los siguientes scripts de Matlab: hough.p, houghpeaks.p y houghlines.p.

- **Hough.p** realiza la transformada de Hough de una imagen. Sus parámetros de entrada son: una imagen de bordes binaria, el espaciado (en grados) de la transformada de Hough a lo largo del eje theta y el espaciado de la transformada de Hough en el eje rho. Los parámetros de salida son: la transformada de Hough, un vector que contiene el ángulo en grados correspondiente a cada columna de la transformada y un vector que contiene los valores de rho correspondiente a cada fila de la transformada.

  function [h, theta, rho] = hough (f, dtheta, drho)

- **Houghpeaks.p** detecta los picos que hay en la matriz de la transformada de Hough. Sus parámetros de entrada son: la matriz de la transformada de Hough, el número máximo de picos que debe buscar, el umbral a partir del cual una celda no será considerada como pico y un vector de dos elementos que especifica el tamaño para suprimir píxeles vecinos (debe ser positivo entero y par). Los parámetros de salida son: vectores con las filas y las columnas de las coordenadas de los picos identificados y la transformada de Hough con los píxeles vecinos a un pico suprimidos.

  function [r, c, hnew] = houghpeaks (h, numpeaks, threshold, nhood)

- **Houghlines.p** extrae segmentos de línea basándose en la transformada de Hough. Sus parámetros de entrada son: una imagen de bordes binaria a la que se le aplicará la transformada de Hoguh, vectores que devuelve el script hough.p, filas y columnas en los que buscar los segmentos de la transformada, la separación de píxeles a partir de la cual considera segmentos distintos y el número de píxeles mínimo que debe tener un segmento para ser considerado como tal. La salida es la estructura de longitud igual al número de segmentos de línea encontrados.

  function lines = houghlines (f, theta, rho, rr, cc, fillgap, minlength)

4.5 - Segmentación II. Umbralización

La umbralización es un proceso que permite convertir una imagen de niveles de gris en una imagen binaria, de tal forma que los objetos de interés se etiqueten con un valor distinto al de los píxeles del fondo. Es una técnica de segmentación rápida, que tiene un coste computacional bajo y que puede ser realizada en tiempo real, aunque al utilizar el histograma no se tiene en cuenta la información espacial sino solamente la distribución de grises en la imagen. Por ello, dos imágenes muy diferentes pueden tener el mismo histograma. Esto hace que, los métodos de segmentación basados en la umbralización, como único medio de segmentación, resulten limitados en muchos problemas reales. Aunque sí se usan con frecuencia como complemento de otros métodos.

Umbralización global

Este tipo de umbralización puede utilizarse cuando la iluminación es relativamente uniforme, es decir, hay una clara definición entre objetos y fondo. Una forma de elegir este umbral es mediante inspección visual de su histograma.

$$B(i,j) = \begin{cases} 1, si\ I(i,j) > U \\ 0, si\ I(i,j) < U \end{cases} \quad (14)$$

Otra forma de elegir el umbral de forma “automática”, será mediante el siguiente proceso iterativo:

1. Seleccionar un umbral inicial U, se recomienda que sea el punto medio entre los niveles máximo y mínimo de intensidad en la imagen.

2. Formar dos grupos de píxeles: G1, consistente en todos los píxeles cuyo valor de intensidad sea >= U, y G2, consistente en los píxeles con valores menor a U.
3. Calcular los valores de intensidad media m1 y m2 para los píxeles en las regiones G1 y G2.
4. Calcular el nuevo valor de umbral como:

$$U = \frac{1}{2}(m_1 + m_2) \tag{15}$$

5. Repetir desde los pasos 2 a 4 hasta que la diferencia en U en sucesivas iteraciones sea inferior a un umbral predefinido en U_o.

En Matlab la función **graythresh** calcula este umbral utilizando el método de Otsu, que trabaja con el histograma, tratándolo como una función discreta de probabilidad.

4.6 - Segmentación III. Crecimiento de regiones

Con la segmentación basada en regiones se determinan zonas dentro de una imagen basándose en criterios de similitud y proximidad entre los píxeles de la misma. El criterio utilizado para unir o dividir regiones de la imagen es la homogeneidad o falta de homogeneidad. Dicha homogeneidad se puede definir a partir de criterios como: el nivel de gris medio, el color, la forma, etc. El resultado de la segmentación es una partición de la imagen en regiones homogéneas.

Crecimiento de regiones mediante adición de píxeles

El crecimiento de regiones es un procedimiento que agrupa píxeles o subregiones en regiones más grandes. El método más simple es la adición de píxeles, donde se parte de un conjunto de puntos semillas a los que se le va añadiendo píxeles vecinos que poseen propiedades similares. Si se usan n semillas, al final se podrán obtener una segmentación con un máximo de n regiones, además del fondo.

La selección de las semillas normalmente se realiza con un conocimiento previo de la imagen. El criterio de parada es cuando no existan píxeles que

cumplen los criterios de inclusión a una región. En Matlab se utiliza el script **regiongrow.p**, para realizar el crecimiento de regiones.

Como parámetros de entrada tiene los siguientes:

- La imagen que va a ser segmentada.
- Una matriz del mismo tamaño que la imagen, o un escalar. En caso de tratarse de una matriz, debe tener un uno en las coordenadas donde se localiza una semilla y un cero en caso contrario. Si se trata de un escalar, hay que definir un valor de intensidad a partir del cual todos los puntos se convierten en semillas.
- Otra matriz del mismo tamaño que la imagen, o un escalar. En caso de tratarse de una matriz, contiene el valor de umbral para cada localización en la imagen. Si se trata de un escalar, se define un umbral global. Esta valor o valores de umbral, se utilizan para comprobar si un píxel en una imagen es suficientemente similar a la semilla de sus 8 vecinos conectados.

Como parámetros de salida tienes los siguientes:

- La imagen segmentada con los miembros de cada región etiquetados con un valor entero.
- El número de regiones diferentes.
- Una imagen que contiene los puntos semillas, del mismo tamaño que la imagen que va a ser segmentada.
- Una imagen que contiene los píxeles que pasan el umbral de test antes de ver su conectividad.

4.7 - Descripción de regiones basadas en el contorno

La representación original del contorno de un objeto segmentado en la imagen es el conjunto de píxeles que definen dicho contorno. Esta representación no es eficiente a nivel computacional y además es sensible a posibles defectos en el proceso de segmentación. Existen distintos métodos de representación del contorno: códigos de cadena, aproximación poligonal, representación polar, esqueletización y descriptores de Fourier. En Matlab nos centramos en los códigos de cadena utilizando distintos scripts.

Códigos de cadena

Son usados para representar la frontera (borde, contorno) en base a un conjunto de segmentos, de una longitud y dirección específica, conectados entre sí. El procedimiento básico de obtención del código de cadena de un contorno consiste en recorrer en sentido horario la frontera, mientras se va asignando el código que más se aproxima a ésta. Para que el código resultante no sea excesivamente largo y sensible al ruido existente, se suele tomar una rejilla asignando cada punto de la frontera del objeto al nodo de la rejilla que se encuentre más próximo a él. Además, una característica importante de este descriptor es que depende del punto de comienzo de la codificación. Para evitar esto se normaliza el código redefiniéndolo de tal forma que el número entero resultante sea el menor posible.

Finalmente, para conseguir una representación invariante frente a orientación se utiliza la diferencia de direcciones en lugar del propio código. Esta diferencia se calcula contando en sentido antihorario el número de direcciones que separan cada dos vectores adyacentes. Si esta diferencia es ordenada en función del entero menor, obtenemos el número de forma. Por lo tanto, el procedimiento seguido para obtener el código de cadena sería:

- Obtener el rectángulo mínimo que circunscribe al objeto.
- Determinar el tamaño de la rejilla, dependiendo de la precisión que se desee.
- Obtener el código de cadena de la figura, su primera diferencia y el número de forma.

Para obtener el código de cadena de un objeto utilizaremos las siguientes funciones y scripts de Matlab: boundaries, cellfun, bound2im, bsubsamp, connectpoly y fhcode.

- **Boundaries** traza los bordes exteriores de los objetos que hay en una imagen. Sus parámetros de entrada son: una imagen binaria, la conectividad de las fronteras de salida y la dirección con la que se trazan las fronteras, horaria o antihoraria. Devuelve la frontera de los objetos hallados en la imagen en un cell array, donde cada elemento son los píxeles de todos los bordes cerrados que se han detectado en la imagen.

```
B = boundaries (f, conn, dir)
```

- **Cellfun** permite obtener un cell array con el borde cerrado más grande.

```
d = cellfun ('length', B)
```

- **Bound2im** genera una imagen binaria, g, con el borde marcado de blanco y el resto de los píxeles en negro. Sus parámetros de entrada son el borde, el ancho y el alto de la imagen y, además, las coordenadas mínimas tanto del eje x como del eje y en la imagen.

```
g = bound2im (b, M, N, x0, y0)
```

- **Bsubsamp** devuelve un nuevo borde con menos muestras y puntos que constituyen el borde. Sus parámetros de entrada son el borde y el paso para tomar muestras del borde.

```
[s, su] = bsubsamp (b, gridsep)
```

- **Connectpoly** une los bordes obtenidos a través de un polígono.

```
cn = connectpoly (s(:,1), s(:,2))
```

- **Fchcode** genera el código de cadena.

```
c = fchcode (b, conn, dir)
```

Descriptores topológicos

Este tipo de descriptores elaboran una descripción global de regiones en una imagen, es decir, propiedades que no se ven afectadas por deformaciones. Algunos ejemplos pueden ser el área, centroide, bounding box y número de Euler de las regiones extraídas de la imagen que se está analizando.

Para obtener estos factores en Matlab se utiliza la función **regionprops**. Sus parámetros de entrada son una imagen binaria y el tipo de medida de forma (Area, BoundingBox, Centroid, Perimeter, etc). Un ejemplo para calcular el área sería el siguiente:

```
>> a = regionprops (b, 'Area');
```

Para calcular los momentos invariantes simplemente tenemos que utilizar **abs(log(invmoments(imagen)))**.

4.8 - Reconocimiento de formas geométricas y colores

Para el reconocimiento de formas geométricas y colores en Matlab se tiene que seguir el siguiente procedimiento:

- Lectura o adquisición de imagen.
- Preprocesamiento.
- Segmentación de la imagen.
- Reconocimiento de objetos.
- Reconocimiento de colores.

Los tres primeros pasos los hemos visto en los apartados anteriores, por lo que nos centraremos directamente en el reconocimiento de objetos y colores.

4.8.1 - Reconocimiento de objetos

El primer paso para el reconocimiento de objetos es usar **regionprops** para obtener las características de los objetos de la imagen: perímetro, área, centroide y boundingBox.

Después, utilizando BoundingBox, se recuadran las figuras detectadas y, por último, con el **método de circularidad** (*Figura 4.15*), se distinguen los tipos

de figuras que se han detectado en la imagen. En caso de trabajar en vivo, es necesario medir la circularidad de las piezas que se utilizan para que el error de reconocimiento sea mínimo.

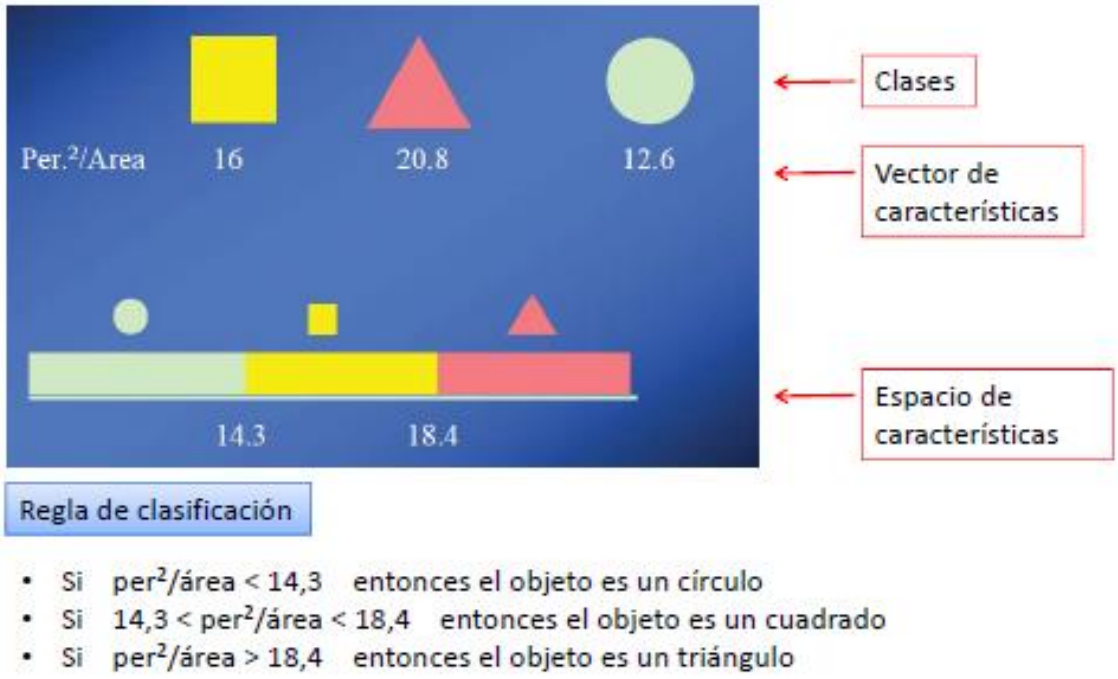


Figura 4.15 Método de circularidad

4.8.2 - Reconocimiento de colores

El reconocimiento de color se puede realizar con un toolbox de Matlab para el reconocimiento de imágenes en color, utilizando un método basado en la agrupación en clústeres (grupos), que separa grupos de objetos. Esto es, empleando **k-mean** se obtiene la ubicación de cada objeto en el espacio y el objetivo es encontrar particiones de manera que los objetos de cada clúster estén lo más cerca posible entre ellos.

En lugar de trabajar con datos de tipo RGB, se hace con LAB, puesto que la información se encuentra en a*b, es decir, lo objetos están compuestos por píxeles a y b. Los pasos a seguir son los siguientes:

- Leer la imagen y pasarla a LAB con **rgb2lab**.
- Convertir los datos de la imagen con **im2single**.
- Generar los grupos que se van a crear con **imsegkmean** con la imagen en single, el número de colores y describiendo 3 regiones.

- Crear imágenes que segmenten la imagen de H&E por color y agruparlas en clústeres.
- Segmentar los núcleos, asignando en el clúster 3 los objetos del color que indiquemos. De esta forma, se separan los colores oscuros de los claros ya que en el espacio de color LAB, los núcleos celulares son de color oscuro.
- Representación.

Para entenderlo mejor, en la *Figura 4.16* viene indicado el código empleado para el reconocimiento del color azul.

```
%PRIMERO LEO LA IMAGEN EN RGB
im=get(handles.hacer_foto,'UserData');

%DESPUES LA PASO A LAB
lab_he = rgb2lab(im);

ab = lab_he(:,:,:);
ab = im2single(ab);
nColors = 3;

pixel_labels = imsegkmeans(ab,nColors,'NumAttempts',3);

figure;imshow(pixel_labels,[]), title('Image Labeled by Cluster Index');
%1
mask1 = pixel_labels==1;
cluster1 = im .* uint8(mask1);

mask2 = pixel_labels==2;
cluster2 = im .* uint8(mask2);

mask3 = pixel_labels==3;
cluster3 = im .* uint8(mask3);
%2
L = lab_he(:,:,1);
L_blue = L .* double(mask3);
L_blue = rescale(L_blue);
idx_light_blue = imbinarize(nonzeros(L_blue));

blue_idx = find(mask3);
mask_dark_blue = mask3;
mask_dark_blue(blue_idx(idx_light_blue)) = 0;
blue_nuclei = im .* uint8(mask_dark_blue);

subplot(2,2,1),imshow(cluster1), title('REGIÓN DE COLORES 1');
subplot(2,2,2),imshow(cluster2), title('REGIÓN DE COLORES 2');
subplot(2,2,3),imshow(cluster3), title('REGIÓN DE COLORES 3');
subplot(2,2,4),imshow(blue_nuclei), title('ELIMINACIÓN DE TONOS AZULES CLAROS');
```

Figura 4.16 Código para el reconocimiento de color

5 - RESULTADOS

En el capítulo anterior se han explicado todas las prácticas de la asignatura optativa de Visión por computador y las funciones o scripts que se utilizan con Matlab. En este capítulo, veremos la implementación de las mismas con la herramienta de Jupyter Notebook junto con los resultados obtenidos.

Para ello trabajaremos con **OpenCV**, una biblioteca de enlaces de Python diseñada para resolver problemas de Visión por computador. Las librerías que utilizaremos son Matplotlib y Numpy.

Matplotlib es una biblioteca que se utiliza para la generación de gráficos a partir de datos contenidos en listas o arrays, en el lenguaje de programación Python y su extensión matemática Numpy. Dentro de Matplotlib podemos encontrar **pyplot**, un paquete que incluye todas las funciones de Matlab para utilizarlas en Python. Para hacerlo más simple, podemos hacer uso de la abreviatura *plt* para llamar a las funciones dentro del archivo.

Numpy significa “Numerical Python” y es una librería que da soporte para crear vectores y matrices grandes multidimensionales, junto con una gran colección de funciones matemáticas de alto nivel para operar con ellas. En este caso también podemos simplificarlo y utilizamos *np* para llamar a las funciones dentro del archivo.

5.1 - Manipulación de imágenes

El primer paso es importar las librerías de OpenCV, Matplotlib y Numpy. Para ello utilizamos las líneas de código mostradas en la *Figura 5.17*.

```
import cv2
from matplotlib import pyplot as plt
import numpy as np
```

Figura 5.17 Importación de librerías

Para cargar una imagen desde el directorio utilizamos la función **imread**. OpenCV y Matplotlib tienen diferentes órdenes de colores primarios. Mientras que OpenCV lee las imágenes en forma de BGR, Matplotlib sigue el orden RGB. Por tanto, cuando leemos un archivo a través de OpenCV, lo leemos como si contuviera canales en el orden azul, verde y rojo. Sin embargo, cuando mostramos la imagen usando Matplotlib, los canales rojo y azul se intercambian.

Para evitar este problema, usaremos la función **cvtColor**. Esta función permite cambiar el espacio de color de una imagen, siendo sus parámetros de entrada: la imagen cuyo espacio de color se va a cambiar y el código de conversión del espacio de color. Por tanto, utilizando **cv2.COLOR_BGR2RGB**, transformaremos el canal al orden de colores que sigue Matplotlib.

Es importante saber que la función cvtColor trabaja con arrays de 8 bits, así, para que la matriz numpy esté formada por el tipo de datos correcto, tenemos que agregar el parámetro *dtype*.

Después, utilizando las funciones **imshow** y **show,** mostramos la imagen que pasamos como argumento. Si queremos ponerle un título a la imagen simplemente tenemos que utilizar la función **title**, y si, además, no queremos que aparezcan los ejes en el figure que contiene la imagen, usamos **xticks([])** e **yticks([])**. Para aumentar o minimizar el tamaño del figure, utilizamos **figsize** en la función **figure**. Todo lo explicado hasta ahora podemos visualizarlo en la *Figura 5.18*.

```
img = cv2.imread('Pictures/big_Ben.jpg')

img1 = np.array(img, dtype=np.uint8)
imagen = cv2.cvtColor(img1, cv2.COLOR_BGR2RGB)

plt.figure(figsize = (5, 5))
plt.imshow(imagen)
plt.title('Imagen original')
plt.xticks([]),plt.yticks([])
plt.show()
```

Figura 5.18 Lectura y representación de una imagen

Una vez que tenemos la imagen original, vamos a pasarla a escala de grises. Para ello utilizaremos **cv2.COLOR_RGB2GRAY**, teniendo en cuenta que convierte a gris la imagen, pero elimina los campos RGB y la cambia a escala de grises. Por este motivo hay que utilizar **cv2.COLOR_GRAY2RGB**, para pasarla a 3 canales y convertirla nuevamente a RGB.

Si queremos mostrar tanto la imagen original como la imagen en escala de grises, una al lado de la otra, usamos un bucle *for* como el mostrado en la *Figura 5.19*.

```
img2 = cv2.cvtColor(imagen, cv2.COLOR_RGB2GRAY)
gray = cv2.cvtColor(img2, cv2.COLOR_GRAY2RGB)

titles = ['Imagen original', 'Imagen en escala de grises']
images = [imagen, gray]

plt.figure(figsize = (10, 10))
for i in range(2):
    plt.subplot(1,2,i+1)
    plt.imshow(images[i])
    plt.title(titles[i])
    plt.xticks([]),plt.yticks([])
plt.show()
```

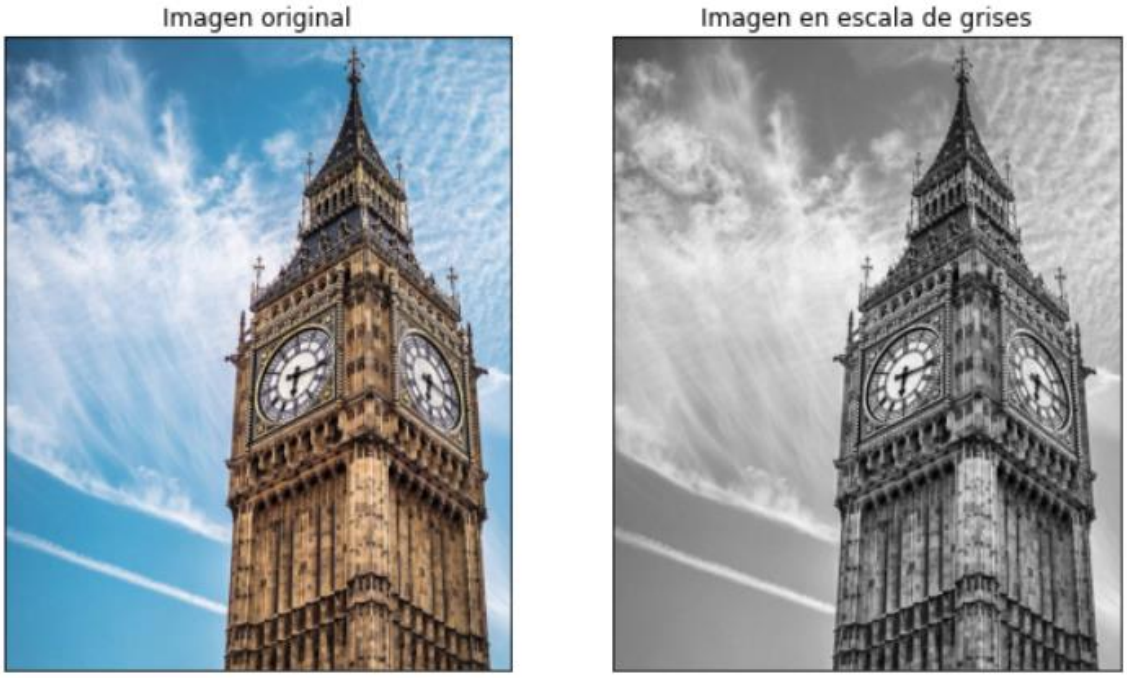


Figura 5.19 Conversión a escala de grises

Para añadir un borde alrededor de una imagen se puede utilizar la función **cv2.copyMakeBorder()**, aunque tiene más aplicaciones para operaciones de convolución, relleno de ceros, etc. Los argumentos de la función son los siguientes:

- Imagen de entrada.
- Ancho del borde en número de píxeles en las direcciones correspondientes (superior, inferior, izquierda y derecha).
- Marca que define qué tipo de borde se agregará.
    - Si el borde es de color constante: cv2.BORDER_CONSTANT.

  - Si el borde es un reflejo de espejo de los elementos del borde: cv2.BORDER_REFLECT.
  - Si el borde es lo mismo que en el caso anterior, pero con un ligero cambio: cv2.BORDER_REFLECT_101 o cv2.BORDER_DEFAULT.
  - Si el último elemento se replica en todo: cv2.BORDER_REPLICATE.
  - Si no se puede explicar: cv2.BORDER_WRAP.
- Color del borde si el tipo de borde es cv2.BORDER_CONSTANT.

En la *Figura 5.20* mostramos la imagen original y la imagen con bordes de color negro en todas las direcciones.

```
constante = cv2.copyMakeBorder(imagen,50,50,50,50, cv2.BORDER_CONSTANT, (0,0,0))

titles = ['Imagen original', 'Imagen con bordes']
images = [imagen, constante]

plt.figure(figsize = (10, 10))
for i in range(2):
    plt.subplot(1,2,i+1)
    plt.imshow(images[i])
    plt.title(titles[i])
    plt.xticks([]),plt.yticks([])
plt.show()
```

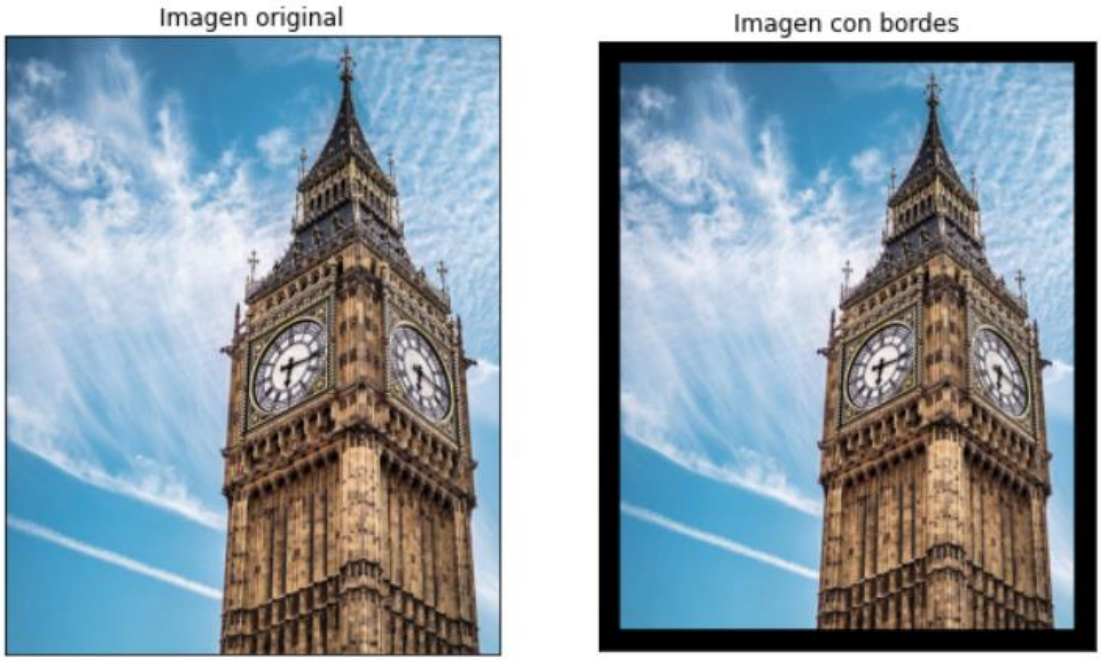


Figura 5.20 Añadir bordes a una imagen

Para rotar una imagen creamos la función *rotar* mostrada en la *Figura 5.21*, la cual no recorta los bordes de la imagen. Lo primero es tomar las dimensiones de la imagen y determinar el centro. Después, utilizando la función **cv2.getRotationMatrix2D()** para la matriz de rotación, rotamos la imagen aplicando un ángulo negativo para rotar en el sentido de las agujas del reloj y añadimos 1.0 de factor de escala. Calculamos los componentes de rotación y las nuevas dimensiones delimitadoras de la imagen. A continuación, ajustamos la matriz para tener en cuenta la traslación y transformamos la imagen original utilizando dicha matriz con la función **cv2.warpAffine()**. Por último, realizamos la rotación. La solución obtenida podemos visualizarla en la *Figura 5.22*.

```
def rotar (image, angle):
    (h,w) = imagen.shape[:2] # Dimensiones de la imagen
    (cx,cy) = (w//2, h//2) # Centro

    # Matriz de rotación
    M = cv2.getRotationMatrix2D((cx,cy), -angle, 1.0)
    # Componentes de rotación
    cos =np.abs(M[0,0])
    sin = np.abs(M[0,1])

    # Dimensiones delimitadoras
    nw = int((h*sin)+(w*cos))
    nh = int((h*cos)+(w*sin))

    # Ajuste de la matriz de rotación
    M[0,2]+=(nw/2)-cx
    M[1,2]+=(nh/2)-cy

    return cv2.warpAffine(imagen,M,(nw,nh))

imagen_rotada45 = rotar(imagen, 45)
imagen_rotada90 = rotar(imagen, 90)
imagen_rotada15 = rotar(imagen, 15)

titles = ['Imagen rotada 45º', 'Imagen rotada 90º', 'Imagen rotada 15º']
images = [imagen_rotada45, imagen_rotada90, imagen_rotada15]
plt.figure(figsize = (10, 10))
for i in range(3):
    plt.subplot(1,3,i+1)
    plt.imshow(images[i])
    plt.title(titles[i])
    plt.xticks([]),plt.yticks([])
plt.show()
```

Figura 5.21 Función para rotar una imagen

Figura 5.22 Imagen con diferentes rotaciones

La función **cv2.flip()** nos permite invertir una imagen horizontalmente, verticalmente y ambas. Esto depende de si el segundo argumento de entrada de la función vale 0, 1 o -1 respectivamente. Podemos visualizarlo en la *Figura 5.23*.

```
reflejo_horizontal = cv2.flip(imagen,0)
reflejo_vertical = cv2.flip(imagen,1)
reflejo_ambas = cv2.flip(imagen,-1)

titles = ['Reflejo horizontal', 'Reflejo vertical', 'Reflejo ambas']
images = [reflejo_horizontal, reflejo_vertical, reflejo_ambas]

plt.figure(figsize = (10, 10))
for i in range(3):
    plt.subplot(1,3,i+1)
    plt.imshow(images[i])
    plt.title(titles[i])
    plt.xticks([]),plt.yticks([])
plt.show()
```

Figura 5.23 Imagen invertida

OpenCV tiene una función que realiza la ecualización de histogramas, esta es **cv2.equalizeHist()**. Su entrada es una imagen en escala de grises y la salida la imagen ecualizada. En las *Figuras 24* y *25* tenemos el código y la representación de la imagen en escala de grises y la ecualizada junto a sus histogramas correspondientes.

```
ecualizar = cv2.equalizeHist(gray[:,:,0])

titles = ['Escala de grises','Histograma', 'Ecualizada', 'Histograma ecualizado']
images = [gray, 0, ecualizar, 0]

plt.figure(figsize = (20, 15))
for i in range(2):
    plt.subplot(2,2,i*2+1),plt.imshow(images[i*2],'gray')
    plt.title(titles[i*2]), plt.xticks([]), plt.yticks([])
    plt.subplot(2,2,i*2+2),plt.hist(images[i*2].ravel(),256)
    plt.title(titles[i*2+1]), plt.xticks([]), plt.yticks([])
```

Figura 5.24 Función para la ecualización

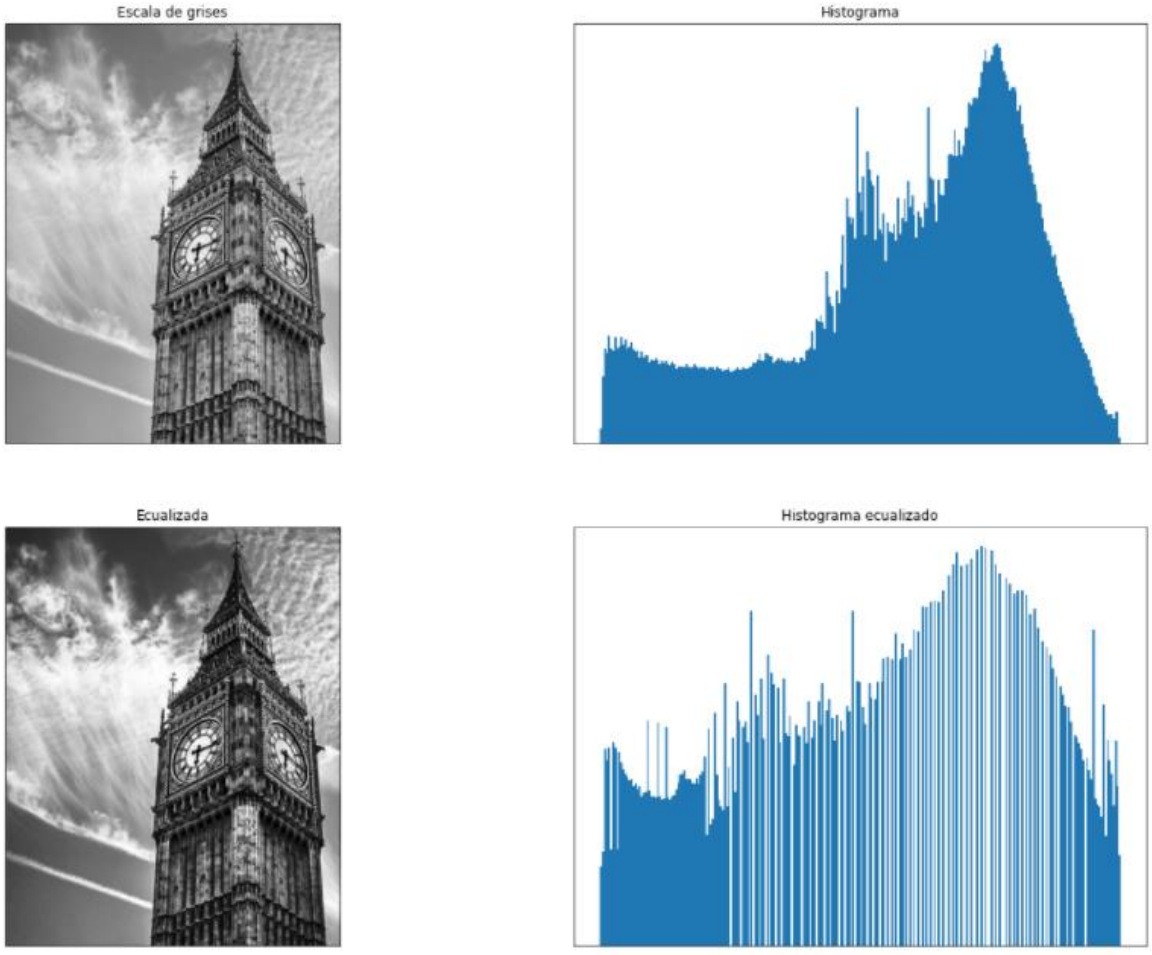


Figura 5.25 Resultados de la imagen ecualizada

5.2 - Reducción de ruido en una imagen digital

Para añadir ruido a una imagen lo primero que tenemos que hacer es importar la biblioteca **random**, que contiene una serie de funciones relacionadas con los valores aleatorios.

El *ruido de sal y pimienta* es un punto blanco aleatorio (ruido de sal) o un punto negro (ruido de pimienta). Pueden ser píxeles negros en áreas brillantes o píxeles blancos en áreas oscuras, o ambos.

En Python, para añadir este tipo de ruido a una imagen se emplea la función **sp_noise()** (*Figura 5.26*). Con esta función obtendremos una salida del mismo tamaño de la imagen con datos enteros de 8 bits, todo relleno de ceros, y también un umbral que será 1 menos la probabilidad que se le esté proporcionando.

Se recorrerá toda la imagen y dependiendo de si el umbral es mayor o menor a esa probabilidad, entonces se colocará un 0 (punto negro) o un 255 (punto blanco). En el caso contrario, se mantendrá la imagen tal cual.

La función **np.random.random ()** devuelve valores aleatorios de una distribución uniforme.

```
def sp_noise(image,prob):

    output = np.zeros(image.shape,np.uint8)
    thres = 1 - prob
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            rdn = random.random()
            if rdn < prob:
                output[i][j] = 0
            elif rdn > thres:
                output[i][j] = 255
            else:
                output[i][j] = image[i][j]

    return output
```

Figura 5.26 Ruido de sal y pimienta

Si la distribución de un ruido obedece a la distribución gaussiana (distribución normal) y su densidad espectral de potencia se distribuye de forma uniforme, hablamos de *ruido gaussiano*. Este ruido incluye los términos de ruido térmico y de disparo.

Se aplica mediante la función **gauss_noise()** (*Figura 5.27*). Con esta función estandarizamos la imagen en escala de grises y generamos el ruido gaussiano para después añadírselo. La función **np.random.normal ()** realiza la distribución normal.

Se buscará el valor más pequeño y si es menor que 0, se devolverá -1. En el caso de ser igual a 0, establecemos que los valores mayores que 1 se convierten en 1 y los menores de 0 se convierten en 0. Esto se hace con la función **np.clip ()**. Para terminar, le asignamos al ruido de salida un rango de 0-255.

```
def gauss_noise(image, mean=0, var=0.01):

    image = np.array(image/255, dtype=float)
    noise = np.random.normal(mean, var * 0.5, image.shape)
    out = image + noise

    if out.min() < 0:
        low_clip = -1.
    else:
        low_clip = 0.
    out = np.clip(out, low_clip, 1.0)
    out = np.uint8(out*255)

    return out
```

Figura 5.27 Ruido gaussiano

El *filtro de la media* se realiza convolucionando la imagen con un filtro de caja normalizado. Este filtro toma el promedio de todos los píxeles bajo el área de un kernel y reemplaza al elemento central por este promedio. Un ejemplo de un kernel de filtro pasa baja (FPB) de 5x5, para ayudar a eliminar el ruido, consiste en que sobre cada píxel de la imagen se centra una ventana de 5x5. Los píxeles contenidos en la ventana se suman y se dividen por 25, el valor resultante es asignado al píxel. Esto equivale a calcular el promedio del valor de

los píxeles que caen en la ventana de 5x5. Esta operación se repite para todos los píxeles, dando lugar a la imagen filtrada.

Podemos utilizar las funciones **cv2.blur()** o **cv2.boxFilter()**, mostradas en la *Figura 5.28*, especificando el ancho y la altura del kernel.

```
blur = cv2.blur(imagen,(7,7))
boxFilter = cv2.boxFilter(imagen, -1, (7,7))
```

Figura 5.28 Filtro de la media

Por otro lado, con la función **cv2.filter2D()** indicada en la *Figura 5.29*, aplicamos una convolución entre un kernel dado y una imagen.

```
#Creamos el kernel
kernel = np.ones((7,7),np.float32)/49
filtrada = cv2.filter2D(imagen,-1,kernel)
```

Figura 5.29 Creación de un kernel

El *filtro de la mediana* calcula la mediana de todos los píxeles mediante el kernel y el píxel central se reemplaza con este valor mediano. Es muy utilizado para eliminar el ruido de sal y pimienta y se hace con la función **cv2.medianBlur()** mostrada en la *Figura 5.30*. El tamaño del kernel debe ser positivo e impar.

```
mediana = cv2.medianBlur(imagen, 7)
```

Figura 5.30 Filtro de la mediana

El *filtro gaussiano* se hace con la función **cv2.GaussianBlur()** indicada en la *Figura 5.31*, en la que los parámetros de entrada son el ancho y la altura del kernel, que debe ser positivo e impar, al igual que en el filtro de la mediana, además de la desviación estándar en las direcciones X e Y. Si ambas se pasan como ceros, se calculan a partir del tamaño del núcleo gaussiano.

```
gaussiano = cv2.GaussianBlur(imagen, (7,7), 0)
```

Figura 5.31 Filtro gaussiano

A continuación, veremos la aplicación tanto de los ruidos como de los distintos filtros para una imagen. En la *Figura 5.32* tenemos la misma imagen en escala de grises, con el ruido de sal y pimienta y con el ruido gaussiano. En la *Figura 5.33* tenemos la imagen con ruido gaussiano y con los filtros gaussiano y con kernel. En las *Figuras 5.34* y *5.35* mostramos la imagen con el ruido de sal y pimienta y gaussiano, junto con los filtros de la media y la mediana.

```
#Ruido de sal y pimienta, con relación de ruido 0.05
sal_pimienta = sp_noise(gray, prob=0.05)

#Ruido gaussiano con un valor medio de 0 y una varianza de 0.01
gaussiano = gauss_noise(gray, mean=0, var=0.01)

titles = ['Escala de grises', 'Ruido de sal y pimienta','Ruido gaussiano']
images = [gray, sal_pimienta, gaussiano]

plt.figure(figsize = (20, 15))
for i in range(3):
    plt.subplot(1,3,i+1)
    plt.imshow(images[i])
    plt.title(titles[i])
    plt.xticks([]),plt.yticks([])
plt.show()
```

Figura 5.32 Ruido de sal y pimienta y ruido gaussiano

```
f_gaussiano = cv2.GaussianBlur(gaussiano, (7,7), 0)

kernel = np.ones((7,7),np.float32)/49
filtrada = cv2.filter2D(gaussiano,-1,kernel)

titles = ['Ruido gaussiano', 'Filtro gaussiano','Filtro con kernel']
images = [gaussiano, f_gaussiano, filtrada]

plt.figure(figsize = (20, 15))
for i in range(3):
    plt.subplot(1,3,i+1)
    plt.imshow(images[i])
    plt.title(titles[i])
    plt.xticks([]),plt.yticks([])
plt.show()
```

Figura 5.33 Ruido gaussiano con distintos filtros

```
blur = cv2.blur(gaussiano,(7,7))

mediana = cv2.medianBlur(gaussiano, 7)

titles = ['Ruido gaussiano', 'Filtro media','Filtro mediana']
images = [gaussiano, blur, mediana]

plt.figure(figsize = (20, 15))
for i in range(3):
    plt.subplot(1,3,i+1)
    plt.imshow(images[i])
    plt.title(titles[i])
    plt.xticks([]),plt.yticks([])
plt.show()
```

Figura 5.34 Ruido gaussiano con filtro de la media y mediana

```
blur = cv2.blur(sal_pimienta,(7,7))

mediana = cv2.medianBlur(sal_pimienta, 7)

titles = ['Ruido de sal y pimienta', 'Filtro media','Filtro mediana']
images = [sal_pimienta, blur, mediana]

plt.figure(figsize = (20, 15))
for i in range(3):
    plt.subplot(1,3,i+1)
    plt.imshow(images[i])
    plt.title(titles[i])
    plt.xticks([]),plt.yticks([])
plt.show()
```

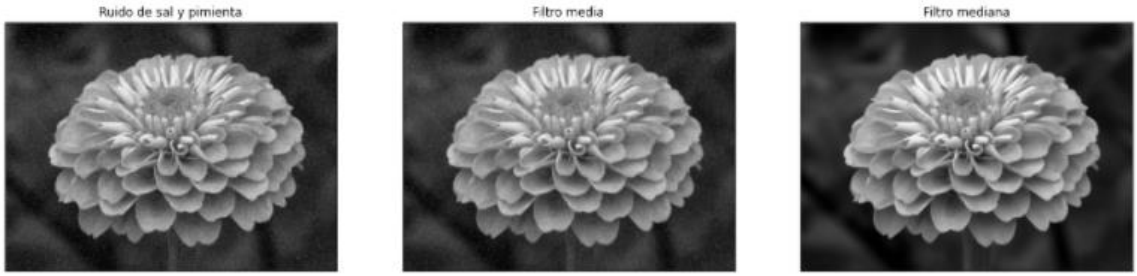


Figura 5.35 Ruido de sal y pimienta con filtro de la media y mediana

5.3 - Detección de bordes en una imagen

Como se ha explicado en el capítulo anterior, los *operadores de Roberts, Prewitt y Sobel*, utilizan máscaras de convolución. Para crear las máscaras horizontales y verticales de cada operador utilizamos **np.array()**. Después, aplicamos la función **cv2.filter2D()** para aplicar las máscaras a la imagen binarizada. En las *Figuras 5.36* y *5.37* se muestran los resultados del operador Roberts en horizontal, en vertical y en ambos. En las *Figuras 5.38* y *5.39*, tenemos los resultados del operador Prewitt y en las *Figuras 5.40* y *5.41* los del operador Sobel.

```
masc_Roberts_x = np.array([[0,0,0],[0,0,1],[0,-1,0]])
masc_Roberts_y = np.array([[-1,0,0],[0,1,0],[0,0,0]])

out1 = cv2.filter2D(binarizada, -1, masc_Roberts_x)
out2 = cv2.filter2D(binarizada, -1, masc_Roberts_y)

titles = ['Roberts horizontal', 'Roberts vertical']
images = [out1, out2]

plt.figure(figsize = (20, 15))
for i in range(2):
    plt.subplot(1,2,i+1)
    plt.imshow(images[i],cmap='gray')
    plt.title(titles[i])
    plt.xticks([]),plt.yticks([])
plt.show()
```

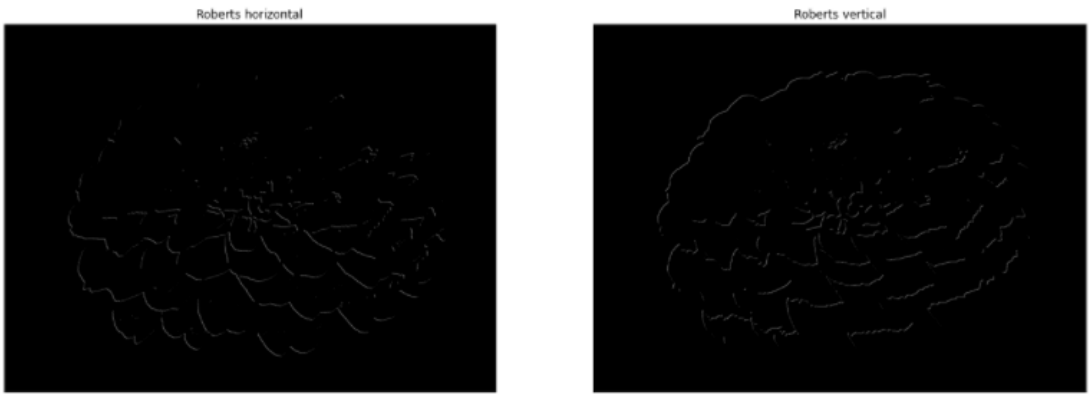


Figura 5.36 Operador Roberts horizontal y vertical

```
masc_Roberts = masc_Roberts_x + masc_Roberts_y
Roberts = cv2.filter2D(binarizada, -1, masc_Roberts)

plt.figure(figsize = (7, 7))
plt.imshow(Roberts, cmap='gray')
plt.title('Operador Roberts')
plt.xticks([]),plt.yticks([])
plt.show()
```

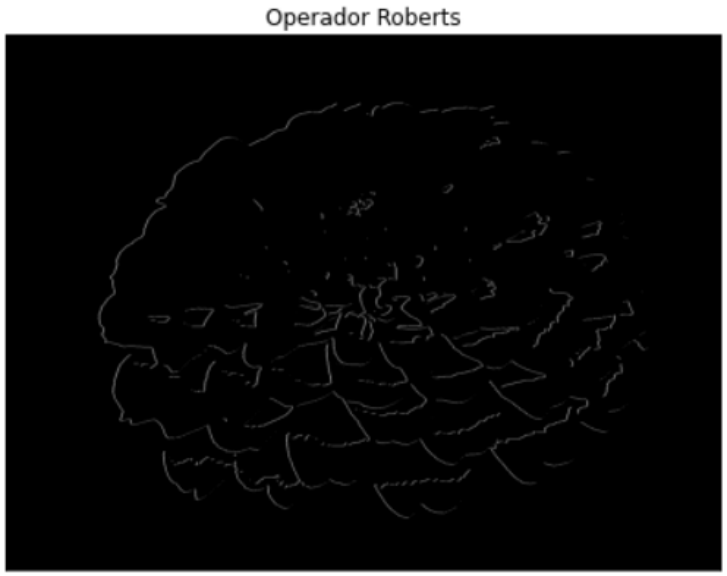


Figura 5.37 Operador Roberts

```
masc_Prewitt_x = np.array([[1,0,-1],[1,0,-1],[1,0,-1]])
masc_Prewitt_y = np.array([[-1,-1,-1],[0,0,0],[1,1,1]])

out5 = cv2.filter2D(binarizada, -1, masc_Prewitt_x)
out6 = cv2.filter2D(binarizada, -1, masc_Prewitt_y)

titles = ['Prewitt horizontal', 'Prewitt vertical']
images = [out5, out6]

plt.figure(figsize = (20, 15))
for i in range(2):
    plt.subplot(1,2,i+1)
    plt.imshow(images[i],cmap='gray')
    plt.title(titles[i])
    plt.xticks([]),plt.yticks([])
plt.show()
```

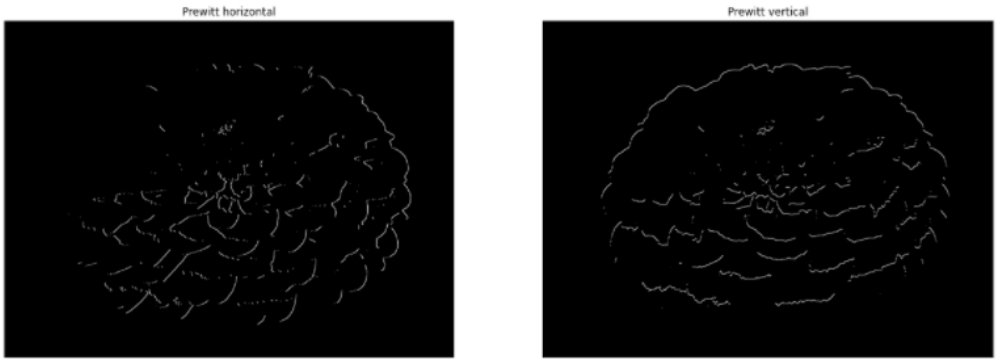


Figura 5.38 Operador Prewitt horizontal y vertical

```
masc_Prewitt = masc_Prewitt_x + masc_Prewitt_y
Prewitt = cv2.filter2D(binarizada, -1, masc_Prewitt)
plt.figure(figsize = (7, 7))
plt.imshow(Prewitt,cmap = 'gray')
plt.title('Operador Prewitt')
plt.xticks([]),plt.yticks([])
plt.show()
```

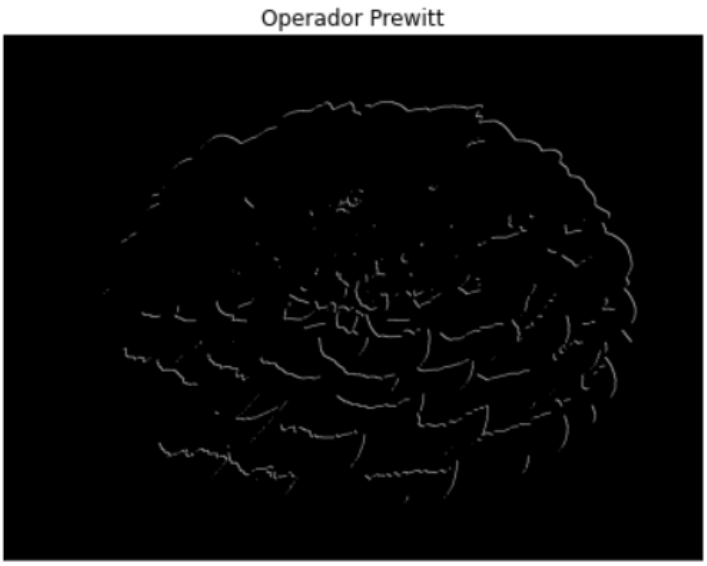


Figura 5.39 Operador Prewitt

```
masc_Sobel_x = np.array([[1,0,-1],[2,0,-2],[1,0,-1]])
masc_Sobel_y = np.array([[-1,-2,-1],[0,0,0],[1,2,1]])

out3 = cv2.filter2D(binarizada, -1, masc_Sobel_x)
out4 = cv2.filter2D(binarizada, -1, masc_Sobel_y)

titles = ['Sobel horizontal', 'Sobel vertical']
images = [out3, out4]

plt.figure(figsize = (20, 15))
for i in range(2):
    plt.subplot(1,2,i+1)
    plt.imshow(images[i],cmap='gray')
    plt.title(titles[i])
    plt.xticks([]),plt.yticks([])
plt.show()
```

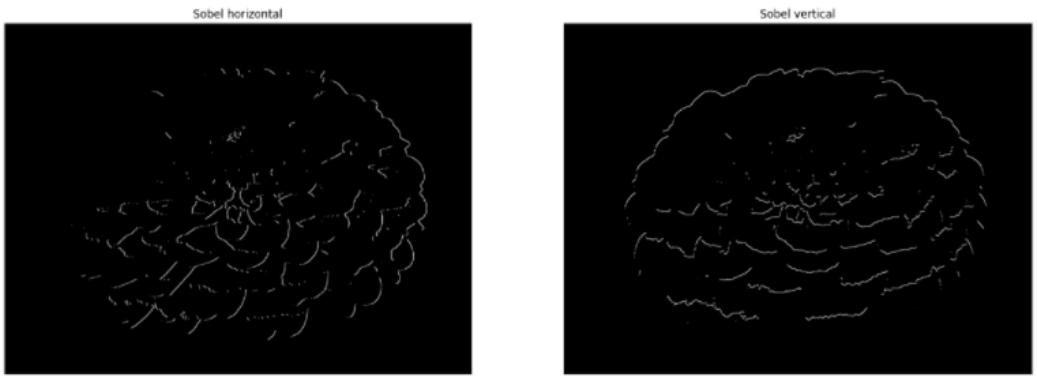


Figura 5.40 Operador Sobel horizontal y vertical

```
masc_Sobel = masc_Sobel_x + masc_Sobel_y
Sobel = cv2.filter2D(binarizada, -1, masc_Sobel)
plt.figure(figsize = (7, 7))
plt.imshow(Sobel,cmap='gray')
plt.title('Operador Sobel')
plt.xticks([]),plt.yticks([])
plt.show()
```

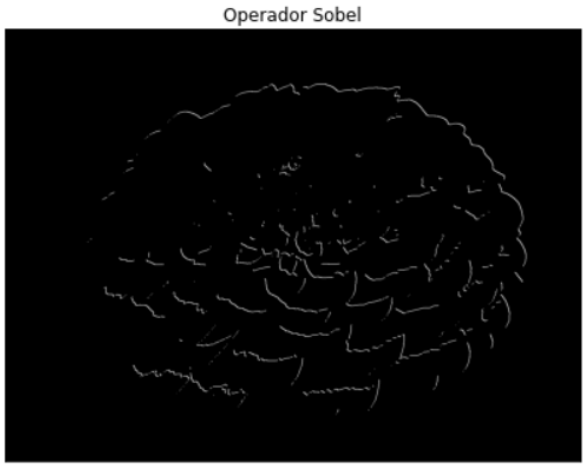


Figura 5.41 Operador Sobel

En el caso de la *Laplaciana*, el procedimiento que se sigue es el mismo que el realizado con los operadores de la primera derivada. En la *Figura 5.42* podemos visualizar el resultado.

```
masc_Laplaciana = np.array([[0,1,0],[1,-4,1],[0,1,0]])
Laplaciana = cv2.filter2D(binarizada, -1, masc_Laplaciana)

plt.figure(figsize = (7, 7))
plt.imshow(Laplaciana,cmap='gray')
plt.title('Laplaciana')
plt.xticks([]),plt.yticks([])
plt.show()
```

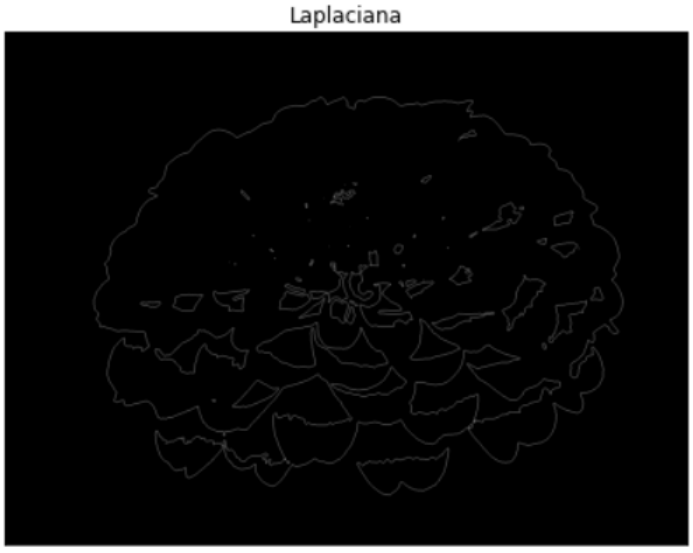


Figura 5.42 Laplaciana

Respecto al *operador Canny*, como podemos ver en la *Figura 5.43*, primero tenemos que añadir un filtro gaussiano a la imagen binarizada y después con la función **cv2.Canny()** aplicamos el operador. Sus parámetros de entrada son la imagen binarizada, el umbral mínimo y el umbral máximo. Estos dos valores se pueden ir variando hasta obtener la solución requerida.

```
gaussiana = cv2.GaussianBlur(binarizada, (5,5), 0)
canny = cv2.Canny(gaussiana, 50, 100)

plt.figure(figsize = (7, 7))
plt.imshow(canny, cmap = 'Greys_r')
plt.title('Canny')
plt.xticks([]),plt.yticks([])
plt.show()
```

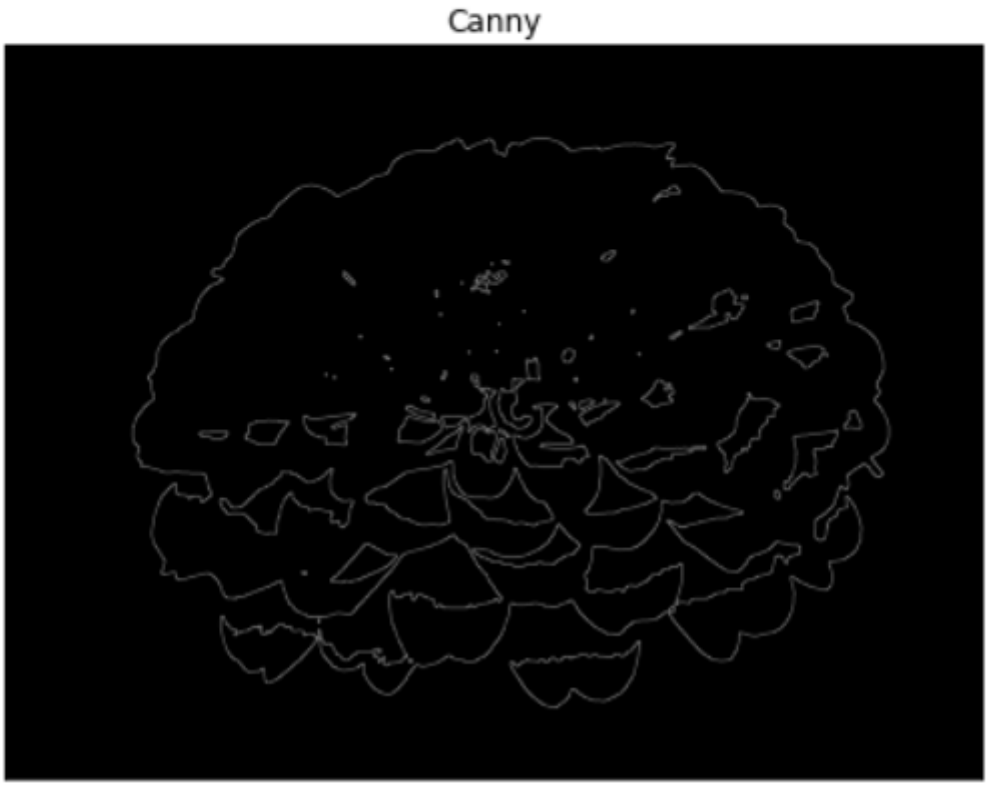


Figura 5.43 Operador Canny

5.4 - Segmentación I. Transformada de Hough

Al encontrar el número de intersecciones entre curvas, es posible detectar una línea. Si hay más curvas intersectando significa entonces que la línea representada tiene más puntos. Es por ello que se puede definir un umbral del número mínimo de intersecciones necesarias para poder detectar una línea.

La Transformada de Hough realiza un seguimiento de la intersección entre las curvas de cada punto de la imagen. En el caso de que el número de intersecciones esté por encima del umbral, se declara como una línea con parámetros (ρ,θ). Esta herramienta de detección de líneas es posible aplicarla mediante las funciones cv2.HoughLines() y cv2.HoughLinesP().

La función **cv2.HoughLines()** es la transformada de Hough estándar y devuelve una matriz de (ρ, θ) valores, donde ρ se mide en píxeles y θ en radianes. La imagen empleada en este apartado se puede visualizar en la *Figura 5.44* y el código utilizado para la función en la *Figura 5.45*. Sus parámetros de entrada son:

- Una imagen de fuente binaria de 8 bits.
- La resolución del parámetro ρ en píxeles. Se usa 1 píxel.
- La resolución del parámetro θ en radianes. Se usa 1 grado.
- El número mínimo de intersecciones, umbral, para detectar una línea. Solo se devuelven aquellas líneas que son mayores que el umbral. Este parámetro se puede ir variando para obtener diferentes soluciones.

```
img = cv2.imread('Pictures/casa1.jpg')

gaussiana = cv2.GaussianBlur(img, (5,5), 0)
canny = cv2.Canny(gaussiana, 40, 90)

titles = ['Imagen original', 'Canny']
images = [img, canny]

plt.figure(figsize = (10, 10))
for i in range(2):
    plt.subplot(1,2,i+1)
    plt.imshow(images[i], cmap = 'Greys_r')
    plt.title(titles[i])
    plt.xticks([]),plt.yticks([])
plt.show()
```

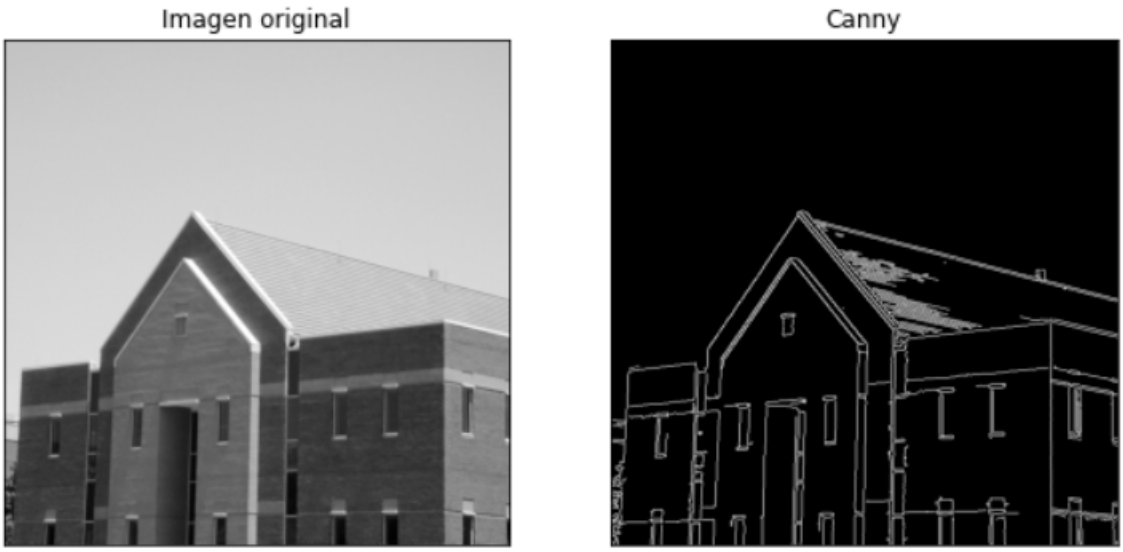


Figura 5.44 Imagen con bordes detectados (I)

```
lines = cv2.HoughLines(canny, 1, np.pi/180, 150)
lines1 = lines[:,0,:]
for rho, theta in lines1[:]:
    a = np.cos(theta)
    b = np.sin(theta)
    x0 = a*rho
    y0 = b*rho
    x1 = int(x0 + 1000*(-b))
    y1 = int(y0 + 1000*(a))
    x2 = int(x0 - 1000*(-b))
    y2 = int(y0 - 1000*(a))
    cv2.line(img,(x1,y1),(x2,y2),(255,0,200),2)
plt.imshow(img)
plt.xticks([]),plt.yticks([])
plt.show()
```

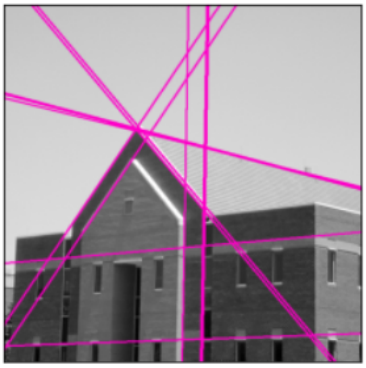


Figura 5.45 Transformada de Hough estándar

Por otro lado, la función **cv2.HoughLinesP()** es la transformada probabilística de Hough y con ella se obtiene directamente los extremos de las líneas detectadas (x_0, y_0, x_1, y_1). Como argumentos de entrada tiene:

- Una imagen de fuente binaria de 8 bits.
- La resolución del parámetro ρ en píxeles.
- La resolución del parámetro θ en radianes.
- El valor del umbral. Solo se devuelven aquellas líneas que son mayores que el umbral.
- La longitud mínima de la línea. Los segmentos de línea más cortos que ese se rechazan.
- El espacio máximo permitido entre puntos de la misma línea para vincularlos.

En la *Figura 5.46* aparece la imagen con la que se trabaja en este apartado y, además, la imagen obtenida después de aplicar Canny. La solución la podemos visualizar en la *Figura 5.47*. Variando los parámetros de *minLineLenght* y *maxLineCap*, obtenemos diferentes resultados.

```
img = cv2.imread('Pictures/matriz.png')

gaussiana = cv2.GaussianBlur(img, (5,5), 0)
canny = cv2.Canny(gaussiana, 45, 90)

titles = ['Imagen original', 'Canny']
images = [img, canny]

plt.figure(figsize = (10, 10))
for i in range(2):
    plt.subplot(1,2,i+1)
    plt.imshow(images[i], cmap = 'Greys_r')
    plt.title(titles[i])
    plt.xticks([]),plt.yticks([])
plt.show()
```

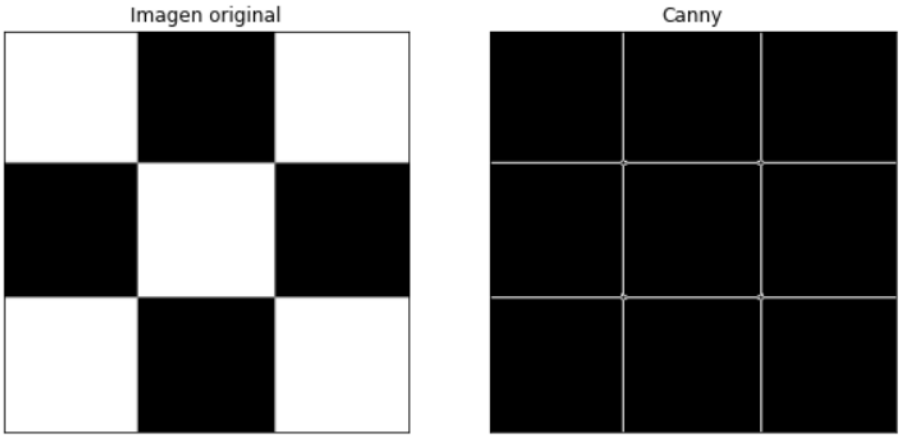


Figura 5.46 Imagen con bordes detectados (II)

```
minLineLenght = 20
maxLineGap = 10
lines = cv2.HoughLinesP(canny, 1, np.pi/180, 10, minLineLenght, maxLineGap)
lines1 = lines[:,0,:]
for x1,y1,x2,y2 in lines1[:]:
    cv2.line(img,(x1,y1),(x2,y2),(0,200,20),2)
plt.figure(figsize = (6,6))
plt.imshow(img)
plt.xticks([]),plt.yticks([])
plt.show()
```

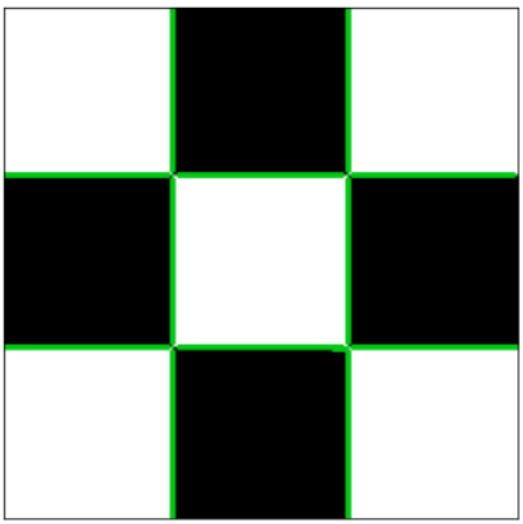


Figura 5.47 Transformada probabilística de Hough

5.5 - Segmentación II. Umbralización

Para separar la región deseada se establece un valor que define el umbral, de forma que los píxeles cuya intensidad superen el umbral serán rechazados o aceptados, según sea el caso. La función utilizada es **threshold**, siendo diferentes los tipos disponibles de umbralización global.

- *Umbral binario* (THRESH_BINARY). Si la intensidad del pixel es mayor al umbral establecido, el pixel de destino se establece al máximo valor definido, en caso contrario se establece a cero.
- *Umbral binario invertido* (THRESH_BINARY_INV). Similar al anterior, solo que aplicado al inverso. Si la intensidad supera el umbral definido el valor resultante será establecido a cero, en caso contrario se establecerá al máximo valor definido.
- *Umbralización de Otsu* (THRESH_OTSU). Se calcula de forma automática un valor de umbral desde el histograma de la imagen bimodial (imagen cuyo histograma posee dos picos). Para el valor umbral, utilizamos 0. El algoritmo encuentra el valor de umbral más óptimo y lo regresa en la salida t.

Como podemos observar en la *Figura 5.48*, los argumentos de entrada de la función **cv2.threshold()** son la imagen en escala de grises, el umbral con el que se va a comparar cada valor de los píxeles de la imagen, un nuevo valor que será asignado cuando el valor del píxel sea mayor al del umbral y el método de umbralización que se va a emplear.

```
img = cv2.imread('Pictures/flor1.png')
img1 = np.array(img, dtype=np.uint8)
imagen = cv2.cvtColor(img1, cv2.COLOR_BGR2RGB)
imagen1 = cv2.cvtColor(imagen, cv2.COLOR_RGB2GRAY)
gray = cv2.cvtColor(imagen1, cv2.COLOR_GRAY2RGB)

t1,a = cv2.threshold(gray,120,255,cv2.THRESH_BINARY)
t2,b = cv2.threshold(gray,120,255,cv2.THRESH_BINARY_INV)

titles = ['Escala de grises','Binary','Binary_Inv']
images = [gray,a,b]
plt.figure(figsize = (10, 8))
for i in range(3):
    plt.subplot(1,3,i+1)
    plt.imshow(images[i])
    plt.title(titles[i])
    plt.xticks([]),plt.yticks([])
plt.show()
```

Figura 5.48 Umbralización binaria e invertida

A continuación, en las *Figuras 5.49* y *5.50* mostramos el código y la solución obtenida al hacer una comparación entre una umbralización global manual y una umbralización Otsu.

```
# Umbralización global manual
t1,img1 = cv2.threshold(img,127,255,cv2.THRESH_BINARY)
# Umbralización Otsu
t2,img2 = cv2.threshold(img,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)
# Representación
images = [img, 0, img1,
          img, 0, img2]
titles = ['Imagen original','Histograma','Umbralización global manual(127)',
          'Imagen original','Histograma','Umbralización Otsu']
plt.figure(figsize = (20, 15))
for i in range(2):
    plt.subplot(2,3,i*3+1),plt.imshow(images[i*3],'gray')
    plt.title(titles[i*3]), plt.xticks([]), plt.yticks([])
    plt.subplot(2,3,i*3+2),plt.hist(images[i*3].ravel(),256)
    plt.title(titles[i*3+1]), plt.xticks([]), plt.yticks([])
    plt.subplot(2,3,i*3+3),plt.imshow(images[i*3+2],'gray')
    plt.title(titles[i*3+2]), plt.xticks([]), plt.yticks([])
plt.show()
```

Figura 5.49 Código para la umbralización de Otsu

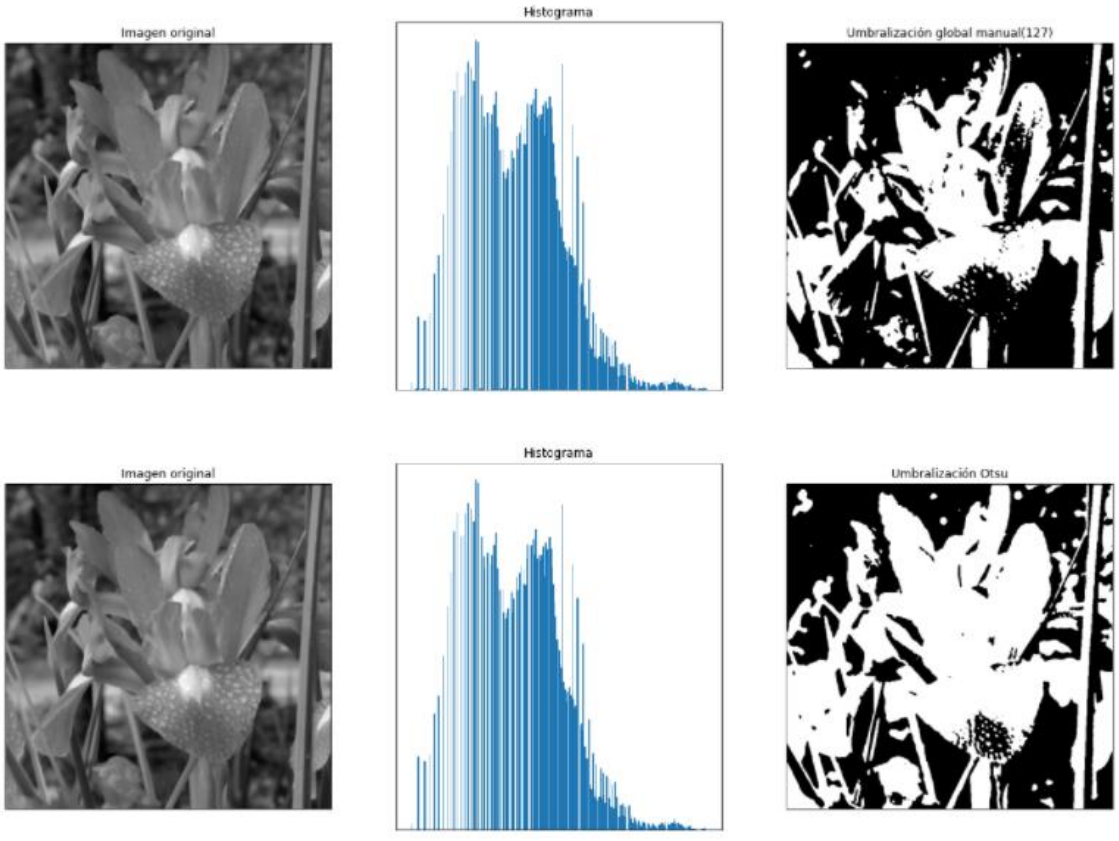


Figura 5.50 Resultados umbralización global manual y Otsu

5.6 - Segmentación III. Crecimiento de regiones

En primer lugar, para el crecimiento de regiones, se encuentra un píxel semilla para cada región que se segmentará como el punto de inicio del crecimiento, y luego el píxel semilla y los píxeles en el vecindario circundante que tienen las mismas propiedades o similares que el píxel semilla se fusionan en la misma región. Estos píxeles se tratan como nuevas semillas para continuar con el proceso anterior hasta que se puedan incluir los píxeles que no cumplan con las condiciones.

Las *Figuras 5.51* y *5.52* contienen el código implementado para este procedimiento y la *Figura 5.53* muestra el resultado de una semilla en el píxel (220,216).

```
class Point(object):
    def __init__(self,x,y):
        self.x = x
        self.y = y

    def getX(self):
        return self.x
    def getY(self):
        return self.y
def getGrayDiff(img,currentPoint,tmpPoint):
    return abs(int(img[currentPoint.x,currentPoint.y]) - int(img[tmpPoint.x,tmpPoint.y]))

def selectConnects(p):
    if p != 0:
        connects = [Point(-1, -1), Point(0, -1), Point(1, -1), Point(1, 0), Point(1, 1), \
                    Point(0, 1), Point(-1, 1), Point(-1, 0)]
    else:
        connects = [ Point(0, -1),  Point(1, 0),Point(0, 1), Point(-1, 0)]
    return connects
```

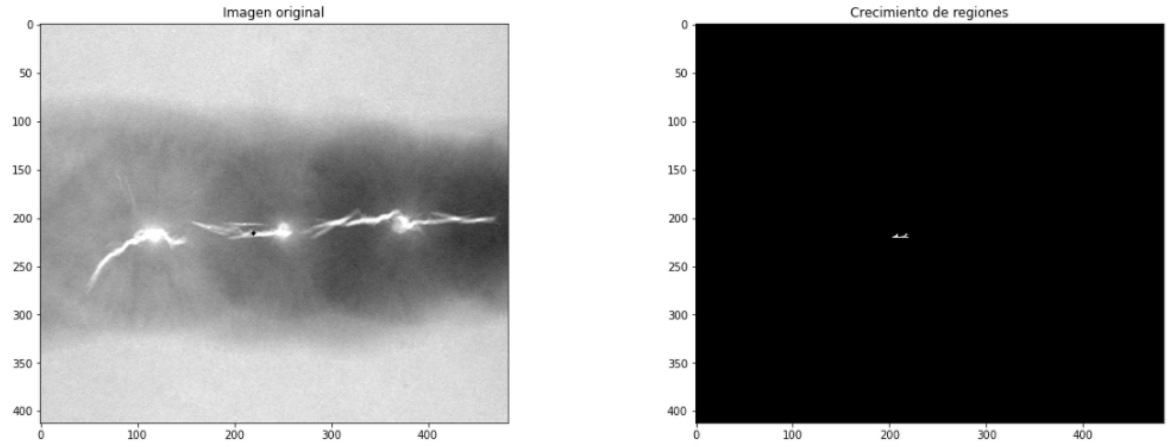
Figura 5.51 Código para el crecimiento de regiones (I)

```python
def regionGrow(img,seeds,thresh,p = 1):
    height, weight = img.shape
    seedMark = np.zeros(img.shape)
    seedList = []
    for seed in seeds:
        seedList.append(seed)
    label = 1
    connects = selectConnects(p)
    while(len(seedList)>0):
        currentPoint = seedList.pop(0)

        seedMark[currentPoint.x,currentPoint.y] = label
        for i in range(8):
            tmpX = currentPoint.x + connects[i].x
            tmpY = currentPoint.y + connects[i].y
            if tmpX < 0 or tmpY < 0 or tmpX >= height or tmpY >= weight:
                continue
            grayDiff = getGrayDiff(img,currentPoint,Point(tmpX,tmpY))
            if grayDiff < thresh and seedMark[tmpX,tmpY] == 0:
                seedMark[tmpX,tmpY] = label
                seedList.append(Point(tmpX,tmpY))
    return seedMark
```

Figura 5.52 Código para el crecimiento de regiones (II)

```python
img = cv2.imread('Pictures/xray.jpg',0)
seeds = [Point(220,216)]
regiones = regionGrow(img,seeds,10)
# Para indicar dónde está la semilla
cv2.circle(img,(220,216),2,(0,0,0),-1)
titles = ['Imagen original', 'Crecimiento de regiones']
images = [img, regiones]
plt.figure(figsize = (20, 15))
for i in range(2):
    plt.subplot(2,2,i+1)
    plt.imshow(images[i],'gray')
    plt.title(titles[i])
plt.show()
altura, anchura = img.shape
print ('Alto: ', altura, 'Ancho: ', anchura)
```




```
Alto:  412 Ancho:  483
```

Figura 5.53 Crecimiento de regiones

5.7 - Descripción de regiones basadas en el contorno

Para obtener una mayor precisión se utilizan imágenes binarias, por ello antes de encontrar el contorno, es necesario establecer un umbral o una detección de bordes Canny. Este paso lo visualizamos en la *Figura 5.54*.

```
img = cv2.imread('Pictures/Figuras.png')

img1 = np.array(img, dtype=np.uint8)
imagen = cv2.cvtColor(img1, cv2.COLOR_BGR2RGB)

imagen1 = cv2.cvtColor(imagen, cv2.COLOR_RGB2GRAY)
gray = cv2.cvtColor(imagen1, cv2.COLOR_GRAY2RGB)

canny = cv2.Canny(gray, 50, 120)

titles = ['Imagen original', 'Escala de grises', 'Canny']
images = [imagen, gray, canny]

plt.figure(figsize = (10, 10))
for i in range(3):
    plt.subplot(1,3,i+1)
    plt.imshow(images[i], cmap = 'gray')
    plt.title(titles[i])
    plt.xticks([]),plt.yticks([])
plt.show()
```

Figura 5.54 Descripción de regiones basadas en el contorno

La función que se utiliza para encontrar contornos en una imagen binaria es **cv2.findContours()**. Tiene 3 parámetros:

- Imagen de entrada.
- Modo de recuperación de contorno.
- Método de aproximación de contorno.

Sus tres valores de retorno son la imagen, el contorno (lista de Python) y la estructura tomográfica (contorneada).

Modo de recuperación de contorno

- CV_RETR_EXTERNAL: solo recupera el contorno más externo.
- CV_RETR_LIST: Recupera todos los contornos y los guarda en una lista vinculada.
- CV_RETR_CCOPM: Recupera todos los contornos y los organiza en dos capas: la capa superior es el límite exterior de cada parte y la segunda capa es el límite del vacío.
- CV_RETR_TREE: Recupera todos los contornos y reconstruye la jerarquía completa de contornos anidados.

Método de aproximación de contorno

- CV_CHAIN_CODE: Contorno de salida de código de cadena Freeman (secuencia de punto fijo).
- CV_CHAIN_APPROX_NONE: Traduce todos los puntos de la forma de código de cadena a la forma de secuencia de puntos.
- CV_CHAIN_APPROX_SIMPLE: Comprime las divisiones horizontales, verticales y diagonales, es decir, la función solo retiene los píxeles al final.
- CV_CHAIN_APPROX_TC89_L1, CV_CHAIN_APPROX_TC89_KCOS: Aplica algoritmo de aproximación de cadena Teh-Chin.
- CV_LINK_RUNS: Utiliza un algoritmo de extracción de contorno completamente diferente con fragmentos horizontales conectados como 1. En este método solo se puede utilizar el modo de extracción CV_RETR_LIST.

Utilizaremos como modo de recuperación el CV_RETR_EXTERNAL y como método de aproximación de contorno el CV_CHAIN_APPROX_SIMPLE.

Para dibujar el contorno, como podemos observar en la *Figura 5.55*, la función que se puede utilizar es **cv2.drawContours()**. Su primer parámetro es la imagen original, el segundo es el contorno, una lista de Python, y el tercero es el índice del contorno (estableciendo -1 para dibujar todos los contornos).

```
# Buscar contornos
contornos, jerarquia = cv2.findContours(canny.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

cv2.drawContours(imagen,contornos,-1,(0,0,0), 3)

plt.figure(figsize = (8, 8))
plt.imshow(imagen)
plt.title('Contornos')
plt.xticks([]),plt.yticks([])
plt.show()
```

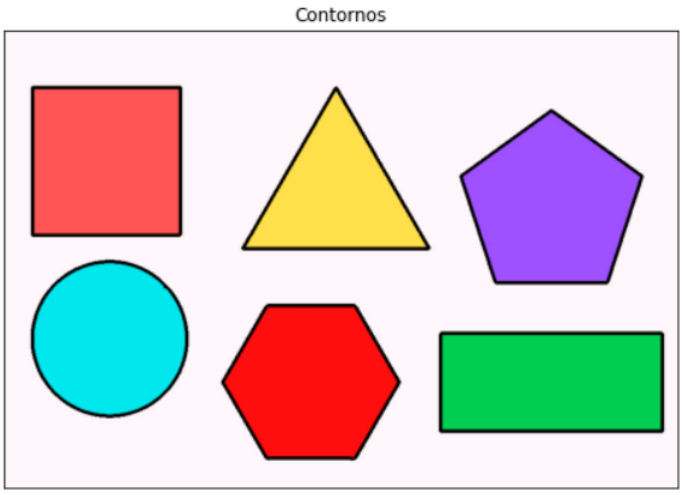


Figura 5.55 Búsqueda y representación de contornos

Descriptores topológicos

A continuación, en la *Figura 5.56* calculamos los momentos invariantes de las figuras geométricas contenidas en una imagen con la función **cv2.moments()** y, también, los descriptores topológicos: el área (**cv2.contourArea()**), el centroide, el perímetro (**cv2.arcLength()**) y el bounding box (**cv2.boundingRect()** y **cv2.rectangle()**). Con el comando *print*, sacamos los valores obtenidos de cada descriptor, empezando de derecha a izquierda y de abajo a arriba. En la *Figura 5.57* tenemos cada una de las figuras geométricas que aparecen en la imagen, encerradas en un cuadrado con sus respectivos centroides.

```
for i in range(len(contornos)):
    #Cálculo de los momentos invariantes
    cnt = contornos[i]
    M = cv2.moments(cnt)
    #print(M)

    # Cálculo del centroide
    cx = int(M['m10']/M['m00'])
    cy = int(M['m01']/M['m00'])
    print('El centroide es:')
    print(cx , cy)

    #Cálculo del área
    area = cv2.contourArea(cnt)
    print('El área es:')
    print(area)

    #Cálculo del perímetro
    perimetro = cv2.arcLength(cnt,True) #True nos indica que el contorno es cerrado.
    print('El perímetro es:')
    print(perimetro)

    cv2.circle(imagen,(cx,cy),10,(250,250,250),-1)
    # El 10 nos indica el radio del círculo y el -1 nos lo dibuja con relleno.

    # Texto con las coordenadas del centroide
    cv2.putText(imagen,'x:'+str(cx)+'y:'+str(cy),(cx,cy),1,1,(0,0,0),1)

    # Para encerrarlos en un cuadrado
    x,y,w,h=cv2.boundingRect(cnt)
    cv2.rectangle(imagen,(x,y),(x+w,y+h),(0,0,0),2)
```

Figura 5.56 Momentos invariantes y descriptores topológicos

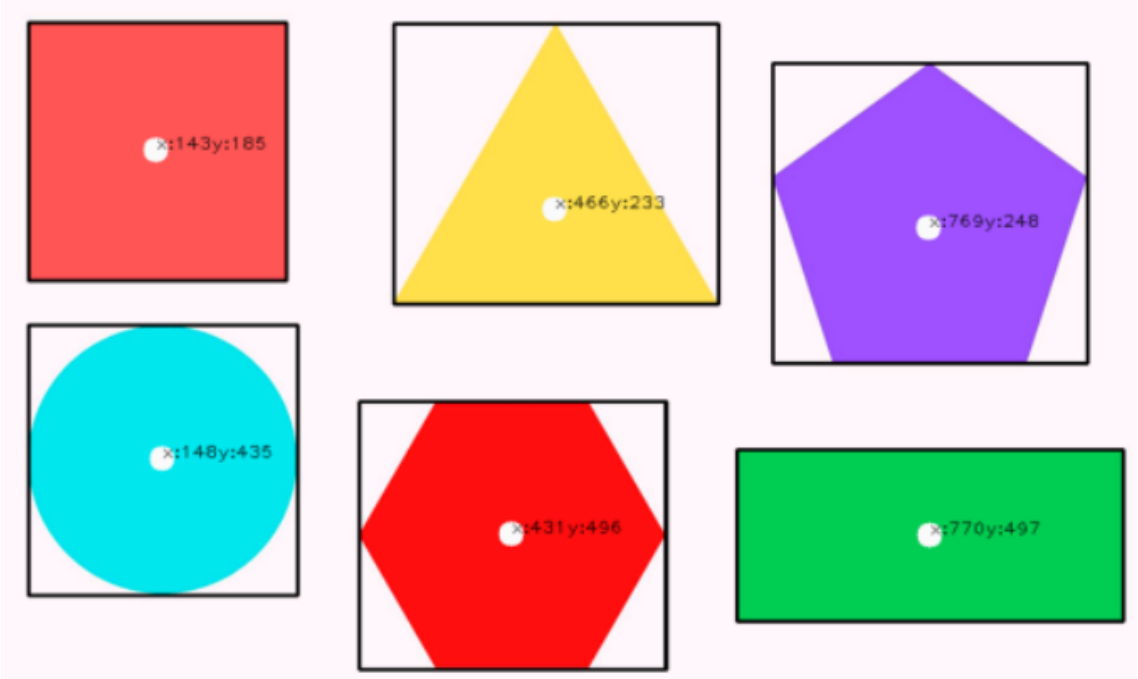


Figura 5.57 Figuras con centroides

5.8 - Blob detection

Un blob es un conjunto de píxeles que están conectados y comparten algunos atributos como los valores grises en una imagen. El objetivo de la detección de blobs es identificar y marcar las regiones oscuras conectadas, o llamado de otra forma, las manchas que aparecen en una imagen.

Con el módulo **SimpleBlobDetector** de OpenCV, es muy sencillo detectar y filtrar los blobs según sus características controlando los siguientes parámetros:

- Umbral. Convertir la imagen original en binaria aplicando umbrales desde *minThreshold* (valor incluido) hasta *maxTreshold* (valor no incluido), con distancia *umbralStep* entre umbrales vecinos.
- Agrupación. En cada imagen binaria, los píxeles blancos conectados se agrupan, llamando a estos blobs binarios.
- Fusión. Los centros de las imágenes binarias se agrupan por sus coordenadas. Los centros que se encuentran más cerca que el parámetro *minDistBetweenBlobs*, se combinan.
- Centro y radio. Se calculan los centros y radios finales de los blobs para devolverlos como tamaños y ubicaciones de puntos clave.

Estos parámetros se pueden configurar para filtrar el tipo de blob que queremos. Para ello se debe utilizar **filterBy** seguido de la filtración que se desea aplicar, y ponerlo en verdadero o falso para activar o desactivar la filtración correspondiente. Existen 5 tipos de filtraciones:

- Por color. Se debe configurar **filterByColor** para comparar la intensidad en una imagen binaria utilizando *blobColor*. Si blobColor = 0, se extraen manchas oscuras y si blobColor = 255, se extraen manchas claras.
- Por tamaño. Configurando **filterByArea** se filtran los blobs en función de su tamaño. Para ello se establece un área entre *minArea* (valor incluido) y *maxArea* (valor no incluido).
- Por circularidad. Los blobs tienen una circularidad entre *minCircularity* (valor incluido) y *maxCircularity* (valor no incluido). Se configura **filterByCircularity**.

- Por relación de inercia. Debe configurarse **filterByInertia**. Los blobs tienen la relación entre *minInertiaRatio* (valor incluido) y *maxInertiaRatio* (valor no incluido).
- Por convexidad. Los blobs tienen una convexidad entre *minConvexity* (valor incluido) y *maxConvexity* (valor no incluido). Se configura **filterByConvexity**.

Para dibujar los blobs se utiliza la función **cv2.drawKeypoints()**. Como argumentos de entrada tiene la imagen original, los puntos clave de la imagen, la imagen exterior, el color de los puntos clave y las banderas que configuran las características del dibujo. Esta función devuelve una imagen cuyo contenido dependerá del valor de las banderas utilizadas que definen lo que se dibuja en la imagen de salida. En este caso se utiliza **cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS**.

Todo lo descrito hasta ahora viene representado en la *Figura 5.58* y la solución obtenida en la *Figura 5.59*.

```
img = cv2.imread('Pictures/girasoles.jpg',0)

parametros = cv2.SimpleBlobDetector_Params()
#Umbral
parametros.minThreshold = 10
parametros.maxThreshold = 250
#Distancia mínima
parametros.minDistBetweenBlobs = 10
#Filtrar por color
parametros.filterByColor = True
parametros.blobColor = 0
#Filtrar por área
parametros.filterByArea = True
parametros.minArea = 30
#Filtrar por circularidad
parametros.filterByCircularity = True
parametros.minCircularity = 0.5
#Filtar por convexidad
parametros.filterByConvexity = True
parametros.minConvexity = 0.87
#Filtrar por inercia
parametros.filterByInertia = True
parametros.minInertiaRatio = 0.01

detector = cv2.SimpleBlobDetector_create(parametros)
puntos_clave = detector.detect(img)
print("Número de blobs: ", len(puntos_clave))

#Dibujar los blobs
img2 = cv2.drawKeypoints(img, puntos_clave, np.array([]), (0,0,255),
                         cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
```

Figura 5.58 Blob detection

```
titles = ['Imagen original', 'Puntos clave']
images = [img, img2]

plt.figure(figsize = (12,12))
for i in range(2):
    plt.subplot(1,2,i+1)
    plt.imshow(images[i], cmap='gray')
    plt.title(titles[i])
    plt.xticks([]),plt.yticks([])
plt.show()
```

```
Número de blobs:  184
```

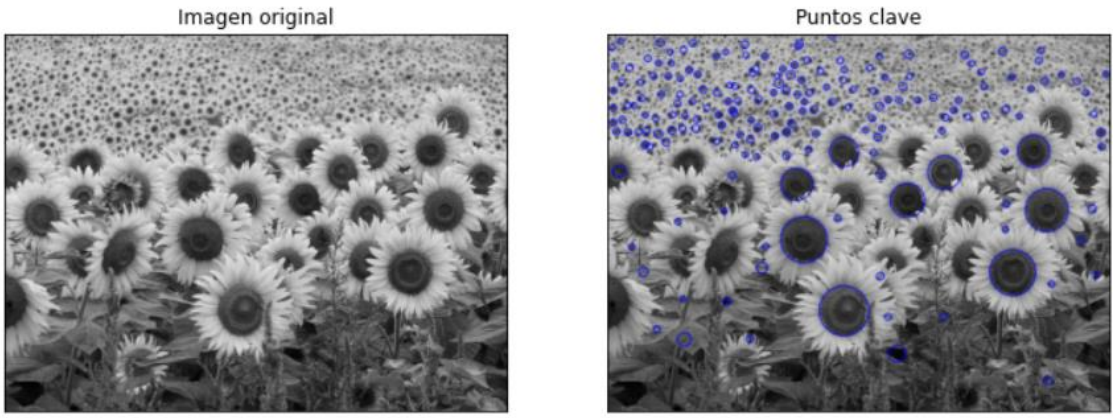


Figura 5.59 Resultado de Blob detection

5.9 - Reconocimiento de formas geométricas y colores

El proceso para el reconocimiento de formas sería el siguiente:

- Lectura de imagen y conversión a escala de grises.
- Detección de bordes.
- Búsqueda de contornos en la imagen binaria.
- Empleo de la función cv2.approxPolyDP().
- Representación.

Con **cv2.approxPolytDP()**, aproximamos una forma de contorno en otra con menor número de vértices, dependiendo de la precisión que le especifiquemos (*epsilon*). Para obtener el parámetro de precisión hacemos uso de la función **cv2.arcLength ()** y aplicamos un porcentaje del 1%.

Después calculamos la longitud de la variable *aprox (len(aprox))*, que nos dará el número de vértices de la figura geométrica que se está analizando. Si la longitud es 3, estaremos hablando de un triángulo, si es 4, de un cuadrado o un rectángulo, si es 5, de un pentágono, y si es 6, de un hexágono. En el caso de un círculo, se usará la condición de que la longitud sea mayor de 10 puesto que el número de vértices encontrados son 14.

Para diferenciar un cuadrado de un rectángulo, sacaremos la relación "*aspect ratio*" que hay entre el ancho y el alto de la figura. De esta forma, si la relación es muy próxima a 1, la figura sería un cuadrado, ya que sus lados tienen la misma longitud. En el caso contrario sería un rectángulo.

Todo lo comentado en este apartado podemos visualizarlo en las *Figuras 5.60* y *5.61*.

```
for i in contornos:
    # Parámetro de precisión
    epsilon = 0.01*cv2.arcLength(i,True)
    # Analizamos la cantidad de vértices de cada contorno
    aprox = cv2.approxPolyDP(i,epsilon,True)
    print(len(aprox))
    # Obtenemos los puntos, el ancho y el alto de cada contorno
    x,y,w,h = cv2.boundingRect(aprox)
  # Reconocimiento
    if len(aprox)>10:
        cv2.putText(imagen,'Circulo',(x+5,y-10),1,1,(0,0,0),1)

    if len(aprox)==3:
        cv2.putText(imagen,'Triangulo',(x+5,y-10),1,1,(0,0,0),1)

    if len(aprox)==4:
        aspect_ratio = float(w)/h
        print('aspect_ratio =', aspect_ratio)
        if 0.98 < aspect_ratio < 1.05 :
            cv2.putText(imagen,'Cuadrado',(x+5,y-10),1,1,(0,0,0),1)
        else:
            cv2.putText(imagen,'Rectangulo',(x+5,y-10),1,1,(0,0,0),1)

    if len(aprox)==5:
        cv2.putText(imagen,'Pentagono',(x+5,y-10),1,1,(0,0,0),1)

    if len(aprox)==6:
        cv2.putText(imagen,'Hexagono',(x+5,y-10),1,1,(0,0,0),1)

    cv2.drawContours(imagen, contornos, -1, (0,0,0), 2)
```

Figura 5.60 Código para el reconocimiento de figuras

```
4
aspect_ratio = 2.2517985611510793
6
14
5
3
4
aspect_ratio = 1.0
```

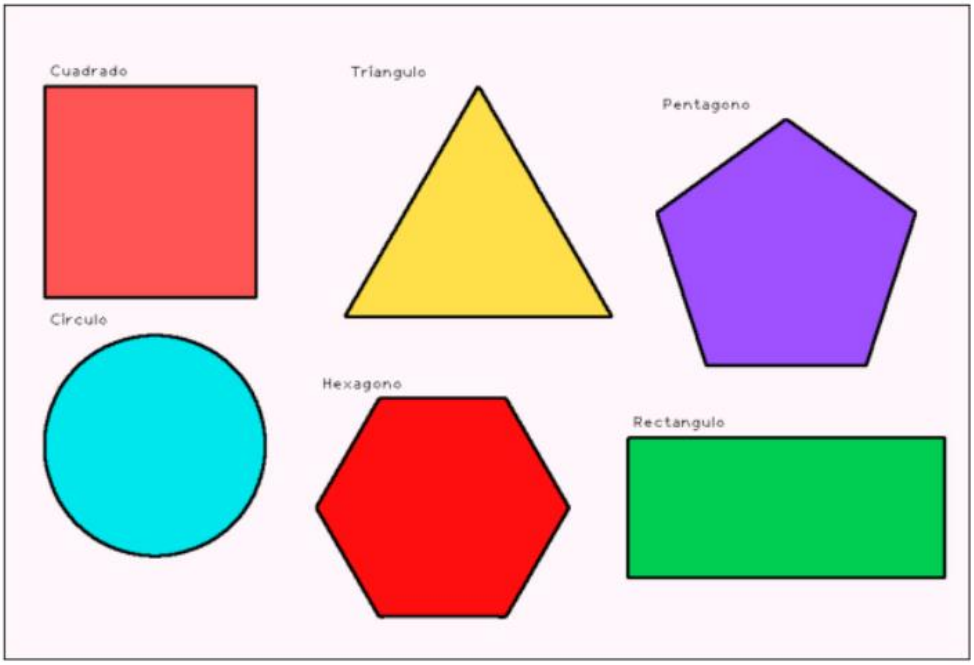


Figura 5.61 Reconocimiento de figuras

Para destacar los colores en una imagen, lo primero que hay que hacer es determinar los rangos del color a detectar en el espacio de color HSV (*Figura 5.62*).

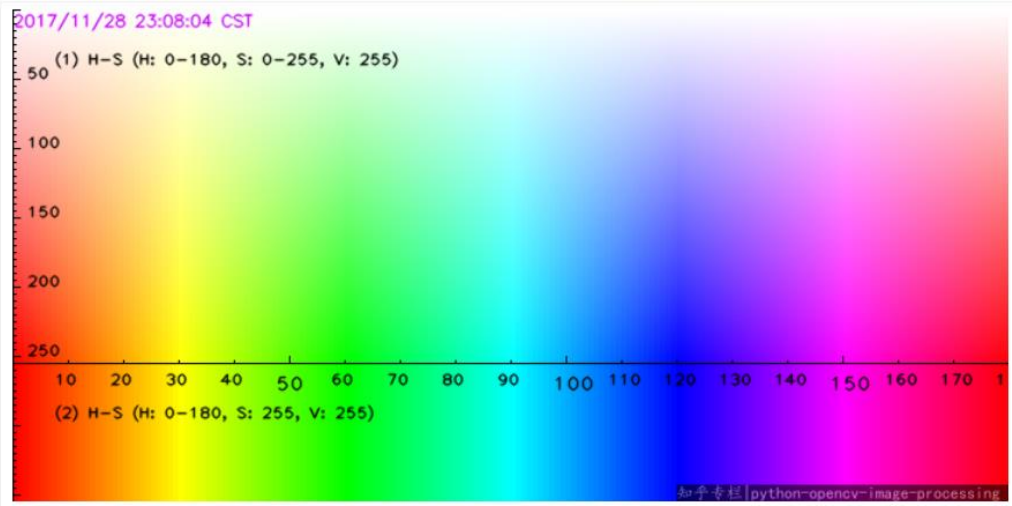


Figura 5.62 Espacio de color HSV

Después, buscamos los rangos del color contenidos en la imagen con la función **cv2.inRange()**. Para terminar, utilizamos la función **cv2.bitwise_and()** para visualizar el color detectado en la imagen, manteniendo el resto de la imagen en negro. En las *Figuras 5.63* y *5.64* encontramos el código empleado y las soluciones obtenidas.

```
HSV = cv2.cvtColor(imagen, cv2.COLOR_RGB2HSV)

# Rangos del color rojo en el espacio de color HSV
rojo1 = np.array([0, 20, 20], np.uint8)
rojo2 = np.array([20, 255, 255], np.uint8)
rojo3 = np.array([170, 20, 20], np.uint8)
rojo4 = np.array([179, 255, 255], np.uint8)
# De la imagen HSV encontramos los rangos de color rojo contenidos en la imagen
mask1 = cv2.inRange(HSV, rojo1, rojo2)
mask2 = cv2.inRange(HSV, rojo3, rojo4)
mask = mask1+mask2
# Visualizamos el color rojo detectado de la imagen
rojoDetectado = cv2.bitwise_and(imagen, imagen, mask = mask)

# Rangos del color azul en el espacio de color HSV
azul1 = np.array([85, 20, 20], np.uint8)
azul2 = np.array([100, 255, 255], np.uint8)
azul = cv2.inRange(HSV, azul1, azul2)
azulDetectado = cv2.bitwise_and(imagen, imagen, mask = azul)

# Rangos del color verde en el espacio de color HSV
verde1 = np.array([45, 20, 20], np.uint8)
verde2 = np.array([75, 255, 255], np.uint8)
verde = cv2.inRange(HSV, verde1, verde2)
verdeDetectado = cv2.bitwise_and(imagen, imagen, mask = verde)

# Rangos del color amarillo en el espacio de color HSV
amarillo1 = np.array([20, 20, 20], np.uint8)
amarillo2 = np.array([40, 255, 255], np.uint8)
amarillo = cv2.inRange(HSV, amarillo1, amarillo2)
amarilloDetectado = cv2.bitwise_and(imagen, imagen, mask = amarillo)

# Rangos del color morado en el espacio de color HSV
morado1 = np.array([130, 20, 20], np.uint8)
morado2 = np.array([140, 255, 255], np.uint8)
morado = cv2.inRange(HSV, morado1, morado2)
moradoDetectado = cv2.bitwise_and(imagen, imagen, mask = morado)
```

Figura 5.63 Código para el reconocimiento de colores

RESULTADOS - RECONOCIMIENTO DE FORMAS GEOMÉTRICAS COLORES

```
titles = ['Original','rojo', 'azul', 'verde', 'amarillo', 'morado']
images = [imagen, rojoDetectado, azulDetectado, verdeDetectado, amarilloDetectado, moradoDetectado]

plt.figure(figsize = (15,6))
for i in range(6):
    plt.subplot(2,3,i+1)
    plt.imshow(images[i])
    plt.title(titles[i])
    plt.xticks([]),plt.yticks([])
plt.show()
```

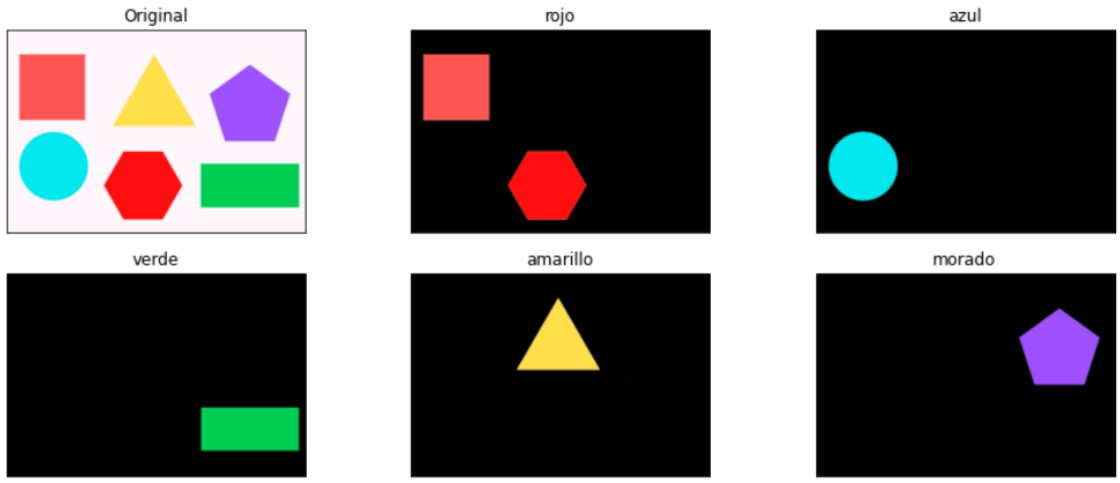


Figura 5.64 Reconocimiento de colores

6 - CONCLUSIONES

Como podemos ver, Jupyter Notebook es una herramienta bastante completa, que nos permite crear documentos que combinen tanto código como texto y representaciones gráficas.

Aunque Jupyter es muy sencilla de utilizar, aprender desde cero cómo funciona ha resultado ser un pequeño inconveniente. Por otro lado, primero hay que saber programar en Python y este ha sido otro de los inconvenientes que personalmente he tenido, puesto que anteriormente no había manejado este lenguaje de programación.

Después, manejándolo poco a poco y probando a hacer algunas cosas en los cuadernos, resultaba entretenido. Cuando conseguí aprender con mayor soltura algunas de sus bibliotecas, en concreto OpenCV, me llenaba de satisfacción ver los resultados que iba obteniendo.

Haciendo una pequeña comparación con Matlab, creo que es un buen cambio emplear Jupyter Notebook para aprender Visión por computador. Es muy útil dado que se pueden ver los resultados al instante y, además, se puede ir cambiando el código para visualizar mejor los cambios que conlleva. Casi todos los conceptos de la asignatura de Visión vienen comprendidos en funciones, lo que hace más sencillo su uso y comprensión.

Personalmente, he disfrutado mucho trabajando con esta herramienta y animo al resto de estudiantes a que también lo hagan. Tiene mucha variedad de opciones y esto ayuda a incrementar la motivación para seguir investigando nuevas posibilidades en el campo de visión.

Mirando hacia el futuro, pienso que debemos aprovechar todas las tecnologías y métodos innovadores que nos pueden beneficiar tanto en el ámbito personal como en el académico. Por ello considero que esta es una buena forma de empezar a ponerlo en práctica.

7 - BIBLIOGRAFÍA

- *Escalera Hueso, Arturo de la D. L. 2001. "Visión por computador: fundamentos y métodos"*
- *Forsyth, David A.; Ponce, Jean, coautor; Mukherjee, Soumen, col.; Bhattacharjee, Arup Kumar, col. 2012. "Computer vision: a modern approach"*
- *González, Rafael C.; Woods, Richard E., coautor; Eddins, Steven L., coautor. 2018. "Digital Image processing using Matlab"*
- *Joseph Howse, Joe Minichino. 2020. "Learning OpenCV 4 Computer Vision with Python 3 Third Edition."*