

Curso práctico multiplataforma de Tratamiento Digital de la Imagen con Jupyter

Ana Cuevas Bravo

6 de abril de 2021

Resumen

Your abstract goes here... ...

Índice general

1. Introducción	1
2. Objetivos	2
2.1. Objetivos	2
2.2. Requisitos	2
2.3. Metodología	2
3. Infraestructura	3
3.1. Python y tratamiento de imagen	3
3.1.1. OpenCV	4
3.1.2. Numpy	4
3.1.3. Matplotlib	5
3.1.4. Scikit-learn	5
3.2. Jupyter	5
3.3. Estructura de las Prácticas de TDI en Matlab	6
4. Prácticas de TDI	10
4.1. Práctica 1: Introducción	10
4.1.1. Teoría tratada en la práctica	10
4.1.2. Comparativa Matlab vs Python	11
4.1.3. Desarrollo de funciones	12
4.2. Práctica 2: Filtrado espacial	13
4.2.1. Teoría tratada en la práctica	13
4.2.2. Comparativa Matlab vs Python	15
4.2.3. Desarrollo de funciones	16
4.3. Práctica 3: Filtrado en el dominio de la frecuencia	17
4.3.1. Teoría tratada en la práctica	17
4.3.2. Comparativa Matlab vs Python	18
4.3.3. Desarrollo de funciones	19
4.4. Práctica 4: Segmentación de imagen I	19
4.4.1. Teoría tratada en la práctica	19

4.4.2.	Comparativa Matlab vs Python	20
4.5.	Práctica 5: Segmentación de imagen II	22
4.5.1.	Teoría tratada en la práctica	22
4.5.2.	Comparativa Matlab vs Python	25
4.5.3.	Desarrollo de funciones	26
4.6.	Práctica 6: Morfología binaria	27
4.6.1.	Teoría tratada en la práctica	27
4.6.2.	Comparativa Matlab vs Python	31
4.6.3.	Desarrollo de funciones	31
4.7.	Práctica 7l	33
4.7.1.	Teoría tratada en la práctica	33
4.7.2.	Comparativa Matlab vs Python	33
4.7.3.	Desarrollo de funciones	33
4.8.	Práctica 8	33
4.8.1.	Teoría tratada en la práctica	33
4.8.2.	Comparativa Matlab vs Python	33
4.8.3.	Desarrollo de funciones	33
4.9.	Práctica 9	33
4.9.1.	Teoría tratada en la práctica	33
4.9.2.	Comparativa Matlab vs Python	33
4.9.3.	Desarrollo de funciones	33
5.	Conclusiones	34

Capítulo 1

Introducción

Capítulo 2

Objetivos

2.1. Objetivos

2.2. Requisitos

2.3. Metodología

Capítulo 3

Infraestructura

En esta sección se detallan tanto el software empleado para la realización del trabajo, como la estructura previa que existía en la asignatura y que ha servido como base para las nuevas prácticas.

El lenguaje de programación utilizado ha sido Python. Profundizaremos en las bibliotecas que tiene este lenguaje para el tratamiento de imagen que han sido necesarias para el desarrollo del trabajo. Esta sección también explicará en que consiste la plataforma Jupyter Notebook y las ventajas que tiene en su uso para la docencia. Por último, se hablará de Matlab, la plataforma original que se usa en la asignatura de Tratamiento Digital de la Imagen y la estructura general que tienen las prácticas de esta asignatura.

3.1. Python y tratamiento de imagen

Python es un lenguaje de programación interpretado de alto nivel orientado a objetos, creado en los años 80 por Guido van Rossum. Se caracteriza por tener una sintaxis sencilla, fácil de leer, lo que facilita el mantenimiento de programas y lo hace accesible para principiantes. Tiene una amplia biblioteca base y además permite el uso de módulos y paquetes. Al ser interpretado en lugar de compilado el proceso de debugging es más rápido.

Python se está usando cada día más para el tratamiento de imagen, esto se debe a que es un lenguaje muy accesible, siendo gratuito y con una sintaxis sencilla. Con el tiempo han ido surgiendo bibliotecas específicas para el tratamiento de la imagen en Python como Pil (Python Imaging Library) también llamada Pillow, Scikit-image y Open-CV, esta última será la que más se use en este proyecto.

Otras bibliotecas que facilitan el tratamiento de imagen en Python son Numpy, una biblioteca para facilitar el uso de matrices y arrays en Python, así como las funciones matemáticas relacionadas y Matplotlib que permite la representación de datos en forma de gráficos.

3.1.1. OpenCV

OpenCV(Open Source Computer Vision Library) es una biblioteca centrada en el tratamiento de imagen y video (Computer Vision) y en aprendizaje de máquina, con interfaces para varios lenguajes de programación como C++, Python o Java. OpenCV es la biblioteca de procesamiento de imagen más usada en el mundo con más de 18 millones de descargas. Originalmente programada en C; el resto de lenguajes usa diferentes interfaces para acceder al código. Esto permite mejorar el rendimiento siendo C el lenguaje más rápido en ejecución.

OpenCV contiene, desde funciones de bajo nivel para el procesado de imagen, hasta algoritmos complejos para detección de caras. En este trabajo, ya que trata de dar una base de tratamiento de imagen, usaremos funciones de bajo nivel como filtros o funciones para cambiar de espacio de color.

Se ha usado la versión 4.2.0 de OpenCV.

3.1.2. Numpy

Numpy es el paquete principal para cálculos complejos con matrices en Python, esto lo convierte en un paquete esencial en el desarrollo de código para usos científicos y de robótica. La base de esta biblioteca es el objeto **ndarray**, que consiste en arrays de n-dimensiones con datos de un mismo tipo y un tamaño establecido al crear la variable, esto último lo diferencia de las listas de python que son dinámicas. Numpy contiene funciones que permiten realizar operaciones con grandes cantidades de datos de una forma más eficiente en memoria y rápida que usando las funciones propias de Python. Numpy está programado y pre-compilado en C. El uso de las funciones de Numpy también permite que el código se parezca más a la notación matemática estándar, lo que lo hace más fácil de leer y entender.

3.1.3. Matplotlib

Matplotlib es una biblioteca para la creación de gráficos en Python. También permite la representación de imágenes. Matplotlib está construido usando Numpy para funcionar con el paquete general de Scipy. Permite interactuar con los gráficos de una forma muy similar a la representación de Matlab. Matplotlib funciona sobre cualquier sistema operativo.

Una de las razones por las que se ha elegido esta biblioteca para representar las diferentes imágenes y gráficos de las prácticas de TDI es por cómo interactúa con los cuadernillos de Jupyter, ya que permite la visualización de los gráficos incrustada en el propio cuadernillo sin crear otra ventana.

Otro punto a favor de Matplotlib es la capacidad para interactuar con los gráficos, cómo ya se ha mencionado antes, esto es algo que permite Matlab e interesaba mantener en las prácticas ya que puede ayudar mucho a la hora de entender algunos de los ejercicios. Esta interactividad consiste en la posibilidad de hacer zoom en los gráficos y un cursor que indica el valor del pixel sobre el que está.

3.1.4. Scikit-learn

Esta es una biblioteca con un amplio rango de algoritmos de aprendizaje de máquina para problemas de tamaño medio, tanto supervisados como no supervisados.

Scikit-learn está programado mayoritariamente en Python aunque incorpora algunas bibliotecas de C++. Sólo depende de Scipy y Numpy e incorpora código compilado para mejorar su eficiencia.

3.2. Jupyter

El cuadernillo Jupyter es una interfaz web de código libre, que permite la creación de documentos que contienen código ejecutable, ecuaciones, visualización (gráficos, imágenes) y texto.

El cuadernillo se guarda como un JSON la extensión `.ipnyb`. La aplicación es un modelo cliente-servidor que se ejecuta a través de un navegador. Se trata de un servidor local que se crea al ejecutar la aplicación lo que implica que no se necesita conexión a internet para leer o modificar un cuadernillo. El servidor lee el documento `.ipnyb` y manda mensajes usando ZeroMQ (una

librería que manda mensajes usando un modelo asíncrono) a un kernel que ejecuta el código en el cuadernillo. El cliente funciona en un navegador y es lo que permite interactuar con el cuadernillo. 3.1

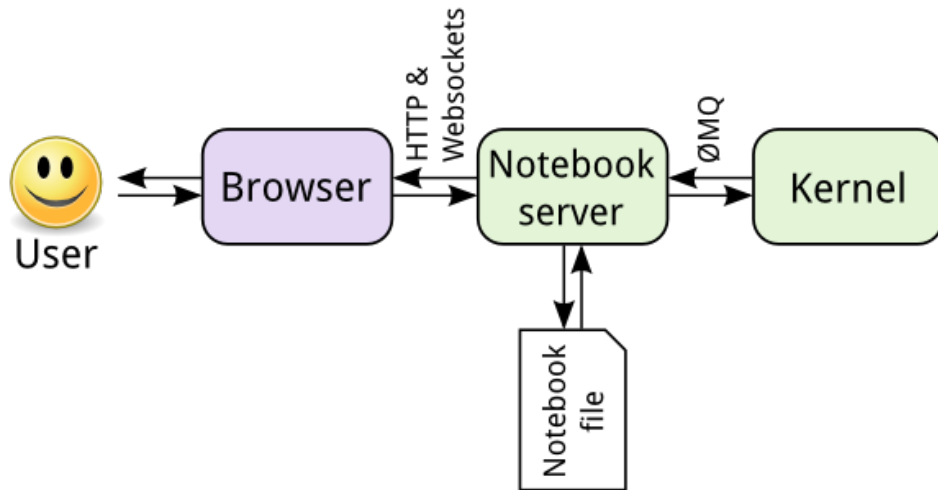


Figura 3.1: estructura de jupyter notebook

Los cuadernillos de Jupyter consisten principalmente en dos tipos de celdas: *code* y *markdown*. *Code* son las celdas ejecutables en las que se escribe código de Python, mientras que *markdown* son las celdas de texto. Para estilizar el texto se usa la sintaxis de markdown, que permite desde poner texto en negrita hasta insertar imágenes o fórmulas matemáticas complejas.

3.3. Estructura de las Prácticas de TDI en Matlab

Las prácticas originales para la asignatura de Tratamiento Digital de la Imagen consisten en una carpeta zip que se pone a disposición de los alumnos en el aula virtual de la asignatura. Dentro de esta carpeta están las imágenes que se van a utilizar en la práctica (cuando no eran imágenes internas de MatLab), funciones adicionales necesarias en caso de que no existieran previamente y un enunciado en pdf. En la imagen 3.2 se puede ver un ejemplo de una de estas carpetas.

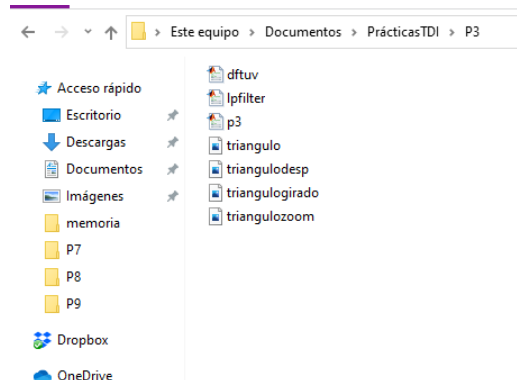


Figura 3.2: ejemplo de carpeta de prácticas

Las prácticas se realizaban en hora de clase en uno de los laboratorios de Windows del campus. Se podían realizar enteras en la duración de la clase.

La profesora daba una breve explicación inicial sobre el tema de la práctica, siempre relacionado con lo que se hubiera dado en las clases de teoría recientes, el resto de la práctica se podía seguir de forma libre con el enunciado, con la profesora respondiendo las dudas que surgieran en el desarrollo del ejercicio.

Los enunciados son instrucciones detalladas sobre los pasos a seguir para realizar la práctica, con poco o nada de teoría ya que, se asume, que el alumno ha recibido las clases de teoría previas. El alumno debe crear un nuevo documento en Matlab e ir ejecutando paso a paso lo que pide el enunciado.

Los enunciados suelen comenzar con un breve párrafo explicando el objetivo de la práctica, por ejemplo, en la práctica 4 que trata el tema de segmentación de Imagen la introducción dice: “El objetivo de esta práctica es comenzar a familiarizar al alumno con las herramientas básicas de segmentación de imagen en entorno MATLAB. Para ello se trabajará con la imagen en escala de grises ‘calculadora.tif’, que acompaña al material de esta práctica. ”

A partir de esta breve introducción la práctica suele estar dividida en secciones normalmente relacionadas con diferentes puntos tratados en la teoría y como llevarlos a cabo.

Los pasos a seguir en MatLab, por lo general, vienen dados en el enunciado que indica qué función usar y cómo llamar a las diferentes variables para mantener una consistencia en los nombres a lo largo de la práctica. En

algunos casos se indican las variables necesarias para llamar a la función que se va a usar en esa sección del ejercicio, mientras que en otros se pide al alumno que use el comando `+help` para informarse sobre el funcionamiento de la función. Del mismo modo si una función requiere de un valor a decidir, dependiendo de lo que se quiera conseguir con su uso, unas veces se da con el enunciado y otras se deja a criterio del alumno para que pruebe como cambian los resultados con diferentes variables y cuál sería la mejor solución para el problema planteado.

Otra característica común de los enunciados de TDI son las preguntas para responder, aunque no se pide una memoria en sí de las prácticas en los enunciados hay preguntas con el objetivo de que el alumno se plantee las respuestas y las razones de los pasos que se han ido realizando durante el ejercicio. La siguiente imagen 3.3 muestra un ejemplo de estas preguntas del principio de la práctica 4.

- ¿son todas las teclas del mismo tamaño?, ¿cuál es la de mayor tamaño?
- ¿tienen todas las teclas alguna letra/número en su interior?, ¿existe conectividad entre ellas?
- ¿aparecen letras/números en el exterior de las teclas?
- ¿qué letras/números presentan mayor intensidad luminosa, los del interior de las teclas o los del exterior?

Figura 3.3: ejemplo de preguntas realizadas en los enunciados de TDI

Algunos de los enunciados tienen imágenes del aspecto que tendría que tener la imagen sobre la que se está trabajando después de ciertos pasos para que el alumno pueda ver si el ejercicio progresa adecuadamente o si debería modificar alguna variable o revisar el código.

Si se pide un algoritmo un poco más complejo, en los enunciados aparece dado un trozo de ese código como ejemplo o directamente para copiar en Matlab ya que el objetivo de estas prácticas no era aprender a programar en MatLab sino ilustrar de forma práctica lo dado en teoría de la asignatura.

Las prácticas se suelen centrar en el tratamiento de una imagen en específico, elegida para ilustrar el tema del que trata la práctica, por ejemplo en la práctica 1, en el apartado que trata sobre color se escoge una imagen con diferentes verduras que muestran una variedad de colores para ilustrar la mezcla aditiva de color al representarse individualmente las matrices de cada color del sistema de representación RGB. Debido a que algunas de las

imágenes usadas en las prácticas originales pertenecen a Matlab o se desconoce su origen se ha tenido que buscar imágenes nuevas, siempre intentando mantener las características de la imagen original.

En total la asignatura consiste de 9 prácticas, 7 de imagen y 2 de video.

Capítulo 4

Prácticas de TDI

Este apartado tratará el desarrollo del proyecto, se irá analizando cada práctica de la asignatura y comparando la versión original con el resultado final en Python, se tratarán los diferentes problemas que han ido surgiendo a la hora de adaptar cada práctica y cómo se han solucionado.

4.1. Práctica 1: Introducción

La práctica 1 es una introducción al tratamiento de imagen usando la programación. Trata cosas como funciones para representar imágenes, formas de escalar una imagen y manipulación de matrices.

4.1.1. Teoría tratada en la práctica

En esta práctica se opera sobre la matriz de la imagen, también se explica de forma práctica la diferencia entre una imagen true color y una imagen indexada. Se pone un énfasis especial en los diferentes tipos de datos en matrices (double, uint64, float...) ya que a la hora de operar puede dar lugar a error.

El siguiente punto de la práctica se centra en la conversión de imágenes de color a gris y binario. A continuación se usan diferentes métodos para modificar la resolución de una imagen actuando tanto sobre la resolución espacial como la intensidad.

La práctica también explica cómo representar el histograma de una imagen y el efecto que tiene la ecualización del histograma sobre el contraste de la imagen.

Por último se verá la interpretación del color y cómo realizar transformaciones puntuales.

4.1.2. Comparativa Matlab vs Python

En este apartado vamos a mencionar las principales diferencias entre la práctica realizada en Matlab y el nuevo formato usando un cuadernillo de Jupyter.

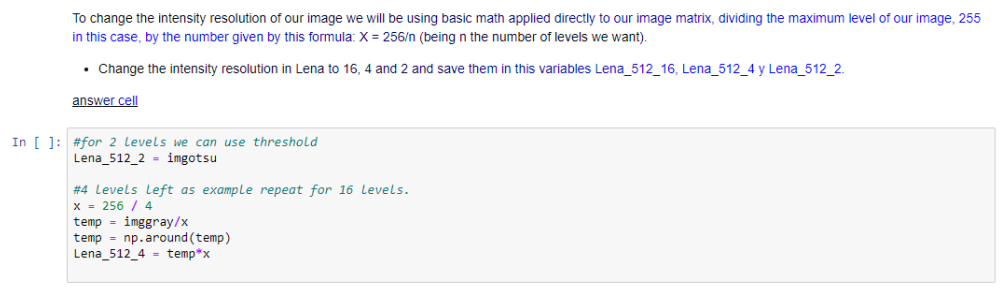
En el primer apartado de la práctica que se centra en la lectura y representación de imágenes podemos encontrar dos diferencias:

La primera es el espacio de color sobre el que se trabaja. Matlab y Matplotlib leen las imágenes de color como RGB mientras que OpenCV usa BGR, esto genera un conflicto entre la librería que se va a usar para leer imágenes y la que se va a usar para representar. Se resuelve en el siguiente apartado que trata sobre espacios de color.

La segunda son los métodos de representación. En Matlab teníamos una única función (**imshow**) que crea una ventana nueva con la imagen representada y herramientas como un cursor y la posibilidad de hacer zoom. En Python tenemos dos opciones de representación, una es **imshow** en la biblioteca OpenCv que al igual que Matlab genera una ventana nueva fuera del cuadernillo pero no tiene herramientas y la otra es **imshow** en Matplotlib que permite visualizar la imagen en el propio cuadernillo y tiene las mismas herramientas que Matlab.

La siguiente gran diferencia entre los dos ejercicios es en el apartado de conversión de tipos de imágenes. Como ya se mencionó, se ha añadido una transformación de espacios de color para solucionar el problema de lectura, pero la mayor diferencia tiene que ver con la falta de una función que convierta una imagen RGB a indexada que si existe en Matlab. Se probó a programar una versión equivalente en Python pero por cuestión de velocidad en la ejecución no se pudo implementar. El problema se solventó exportando las dos matrices que forman una imagen indexada desde Matlab e implementarlas en Python aprovechando para explicar como se crean los mapas de color.

La falta de la función RGB2ind vuelve a ser relevante en el punto sobre modificación de la resolución de intensidad de una imagen, ya que en la práctica original se usaba esta función para cambiar el número de niveles de intensidad en el mapa de color. Se resuelve explicando una resolución matemática al problema sin cambiar la imagen a indexada. Al ser una imagen en escala de grises con menos información que una imagen RGB también se puede usar la versión de Python de RGB2ind ya que el tiempo de ejecución es menor para imágenes en escala de grises. En la siguiente imagen 4.1 se puede ver la parte del enunciado en la que se explica la resolución matemática.



To change the intensity resolution of our image we will be using basic math applied directly to our image matrix, dividing the maximum level of our image, 255 in this case, by the number given by this formula: $X = 256/n$ (being n the number of levels we want).

- Change the intensity resolution in Lena to 16, 4 and 2 and save them in this variables `Lena_512_16`, `Lena_512_4` y `Lena_512_2`.

answer cell

```
In [ ]: #for 2 levels we can use threshold
Lena_512_2 = imgotsu

#4 levels left as example repeat for 16 levels.
x = 256 / 4
temp = imggray/x
temp = np.around(temp)
Lena_512_4 = temp*x
```

Figura 4.1: enunciado del apartado sobre resolución de intensidad

En el apartado sobre el histograma, el único cambio, es la imagen utilizada para ejemplificar la ecualización del histograma ya que la imagen original pertenece a Matlab. Sucede lo mismo en el último apartado con la imagen de los pimientos.

Por último, en los cuadernillos se ha añadido un nuevo apartado donde se muestra el aspecto que tiene que tener la imagen resultante de algunos apartados de la práctica.

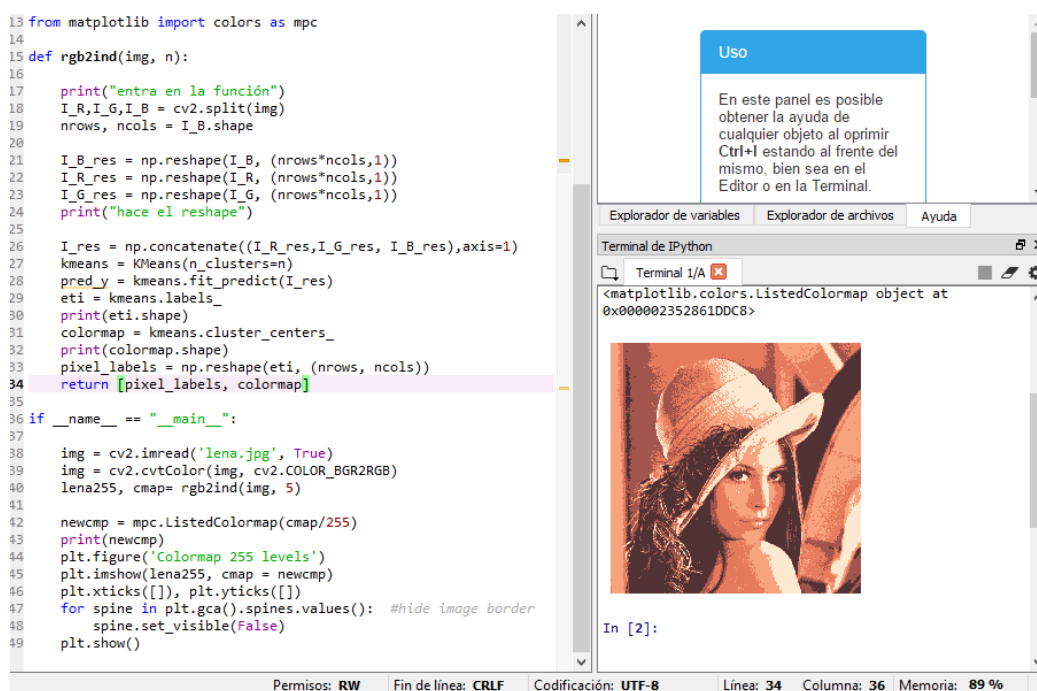
4.1.3. Desarrollo de funciones

En esta práctica solamente fue necesario desarrollar una función nueva: RGB2ind. Esta función se usa para convertir una imagen true color RGB en una imagen indexada con el número de niveles como parámetro.

A la hora del desarrollo se intentó buscar el código de la función original de Matlab pero, es de las funciones precompiladas en C y su código no es accesible. Se resolvió usando el algoritmo K-medias con un número de núcleos igual al número de niveles requeridos.

La función final tiene un problema de tiempo de ejecución. Para un número pequeño de núcleos funciona bien pero, por ejemplo para los 256 requeridos en la práctica el tiempo de ejecución interrumpiría el desarrollo cómodo del ejercicio. Para mejorarla se podría probar a usar una versión pre-compilada de Python o programar la función en C e implementarla.

Esta imagen 4.2 muestra como queda la solución para una imagen indexada de 5 niveles.



```

13 from matplotlib import colors as mpc
14
15 def rgb2ind(img, n):
16
17     print("entra en la función")
18     I_R,I_G,I_B = cv2.split(img)
19     nrows, ncols = I_B.shape
20
21     I_B_res = np.reshape(I_B, (nrows*ncols,1))
22     I_R_res = np.reshape(I_R, (nrows*ncols,1))
23     I_G_res = np.reshape(I_G, (nrows*ncols,1))
24     print("hace el reshape")
25
26     I_res = np.concatenate((I_R_res,I_G_res, I_B_res),axis=1)
27     kmeans = KMeans(n_clusters=n)
28     pred_y = kmeans.fit_predict(I_res)
29     eti = kmeans.labels_
30     print(eti.shape)
31     colormap = kmeans.cluster_centers_
32     print(colormap.shape)
33     pixel_labels = np.reshape(eti, (nrows, ncols))
34     return [pixel_labels, colormap]
35
36 if __name__ == "__main__":
37
38     img = cv2.imread('lena.jpg', True)
39     img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
40     lena255, cmap= rgb2ind(img, 5)
41
42     newcmap = mpc.ListedColormap(cmap/255)
43     print(newcmap)
44     plt.figure('Colormap 255 levels')
45     plt.imshow(lena255, cmap = newcmap)
46     plt.xticks([], plt.yticks([]))
47     for spine in plt.gca().spines.values(): #hide image border
48         spine.set_visible(False)
49     plt.show()

```

Uso

En este panel es posible obtener la ayuda de cualquier objeto al oprimir Ctrl+I estando al frente del mismo, bien sea en el Editor o en la Terminal.

Explorador de variables Explorador de archivos Ayuda

Terminal de IPython

Terminal 1/A

<matplotlib.colors.ListedColormap object at 0x000002352861DDC8>

In [2]:

Permisos: RW Fin de línea: CRLF Codificación: UTF-8 Línea: 34 Columna: 36 Memoria: 89 %

Figura 4.2: muestra de código y solución de RGB2ind

4.2. Práctica 2: Filtrado espacial

4.2.1. Teoría tratada en la práctica

Este segundo ejercicio se centra en el filtrado de imágenes en el espacio, se irán demostrando diferentes tipos de filtro y sus usos más comunes en la práctica.

El primer apartado enseña diferentes tipos de ruido y cómo contaminar una imagen, los tipos de ruido tratados son: ruido gaussiano, ruido de sal y

pimienta y ruido granular.

Los filtros se dividen en dos categorías: filtros paso bajo y filtros paso alto. En esta práctica se explican dos filtros paso bajos diferentes. El primero es el filtro de media que es un filtro lineal. Consiste en ir colocando una máscara sobre la imagen y calculando la media de los píxeles bajo la máscara, este filtro genera un suavizado de la imagen [4.3].

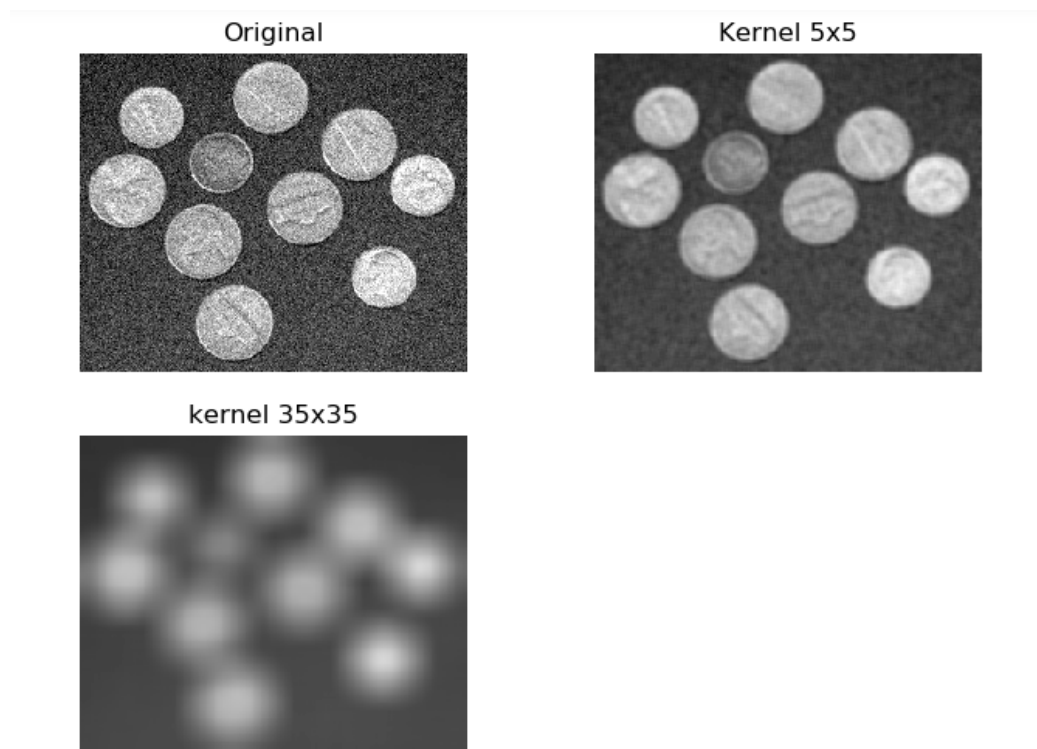
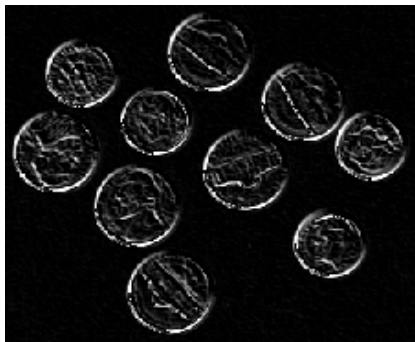


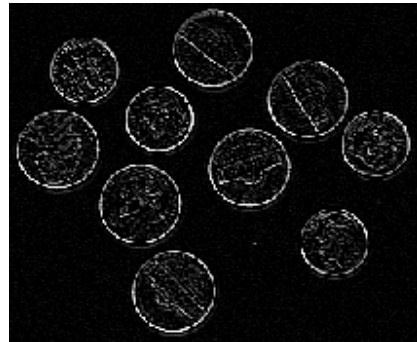
Figura 4.3: imagen del cuadernillo donde se demuestra el filtro de media con diferentes tamaños de máscara

El otro filtro paso bajo es el filtro de mediana que es un filtro no lineal. Este filtro coge los valores de debajo de la máscara y hace la mediana con esos valores, esto permite que se mantengan mejor los valores originales de la imagen, se usa para quitar ruido de sal y pimienta en la práctica.

Los filtros paso alto son filtros de realce de contornos. Se usan sobre todo para detectar bordes de objetos en la imagen. En esta práctica se dan dos tipos de filtro paso alto dependiendo de la máscara que usan: Prewitt e Isotrópico 4.8



(a) Prewitt.



(b) Isotrópico.

Figura 4.4: Filtros paso alto.

4.2.2. Comparativa Matlab vs Python

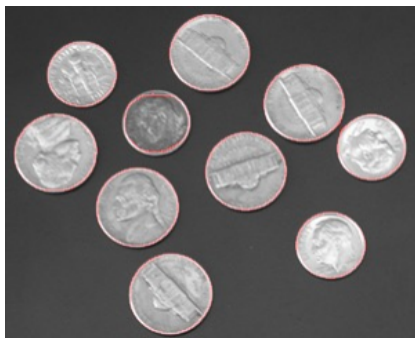
En el primer apartado de esta práctica se usan una serie de funciones que no pertenecen a Matlab y vienen entre los materiales de la práctica, estas funciones se recrean en Python con la misma funcionalidad.

En el apartado de filtros lineales la función más importante es **imfilter** que tiene de parámetros de entrada la imagen a filtrar, la máscara y el tipo de *padding*. El tipo de *padding* indica como se va a rellenar la máscara cuando el valor del centro es un borde de la imagen, los dos tipos de *padding* que se demuestran en esta práctica son *zero padding* que rellena con 0 y *mirror padding* que copia los píxeles cercanos al borde. La función **filter2D** de OpenCV funciona de la misma manera y con los mismos parámetros.

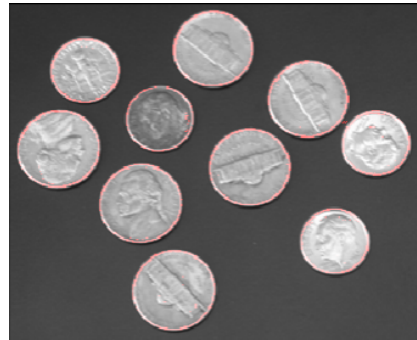
Es importante mencionar que para apreciar bien las diferencias de tipo de *padding* se tienen que ver bien los bordes de la imagen. Matplotlib, por defecto, añade un borde negro a sus figuras, esto impide que se aprecie bien la diferencia. Se encuentra un comando adicional para quitar estos bordes por defecto.

Para el filtro de mediana también existe una función en OpenCV (**medianblur**) que funciona exactamente igual que el equivalente de Matlab.

La mayor diferencia en el apartado de realce de contornos la encontramos en la función **fspecial** de Matlab que genera máscaras específicas para los diferentes filtros paso alto. Esta función no existe en OpenCV, se usa Numpy para generar las matrices y se van metiendo los valores correctos de acuerdo



(a) Matlab



(b) Python

Figura 4.5: Imágenes resultado del último apartado de la práctica 2

con el tipo de filtro que se quiere usar. Para hacer el filtrado en sí, se usa la misma función que para los filtros lineales paso bajo.

El último apartado de composición de imágenes requiere funciones que se usaron ya en la práctica uno y una función nueva de Numpy para realizar sumas punto a punto de matrices de más de dos dimensiones. La solución final queda muy parecida a la original de Matlab. 4.5

4.2.3. Desarrollo de funciones

Para este ejercicio se han tenido que desarrollar varias funciones, todas relacionadas con la función de añadir ruido en una imagen. La función principal **imnoise** tiene 3 parámetros de entrada: la imagen a contaminar, el tipo de ruido y un parámetro genérico que indica los parámetros de cálculo del ruido (por ejemplo si el ruido es gaussiano este parámetro contiene la media y la varianza). La imagen puede ser tanto en tonos de gris como en color. Para añadir el ruido se llama a **addnoise** que es la función que realiza las operaciones matemáticas necesarias.

Otras funciones que se han ido creando por necesidad de **imnoise** son **checkcolor** que, como su nombre indica, se encarga de comprobar si la imagen es de color y un par de funciones para cambiar el tipo de datos de la matriz de image,n por ejemplo, de unit8 a double y viceversa.

4.3. Práctica 3: Filtrado en el dominio de la frecuencia

4.3.1. Teoría tratada en la práctica

En esta práctica se trata la imagen en el dominio de la frecuencia. Se intenta mostrar de forma práctica cómo se ve una imagen en frecuencia y en qué consiste la transformada de Fourier bidimensional. Se explica porqué se necesitan dos representaciones diferentes para una misma imagen (módulo y fase) y qué representa cada una.

El segundo apartado demuestra las propiedades de la transformada de Fourier, qué transformaciones en el espacio afectan a la fase y cuáles al módulo y de qué manera.

El resto de la práctica se centra en el filtrado de imagen en el dominio frecuencial. Primero se explican dos filtros paso bajo: Gaussiano e ideal [4.6]. Se trata de filtrar la imagen en frecuencia y ver como afecta en el espacio tras hacer una tranformada de fourier inversa. Para concluir la práctica se

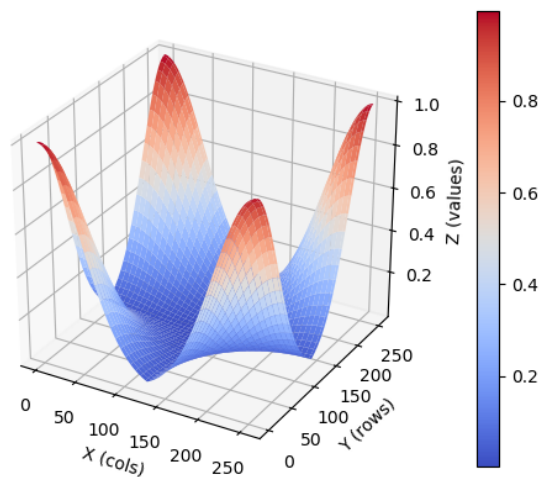


Figura 4.6: representación de un filtro paso bajo Guassiano

tratan los filtros paso alto que, en el dominio de la frecuencia, son iguales que los filtros paso bajo pero invertidos.

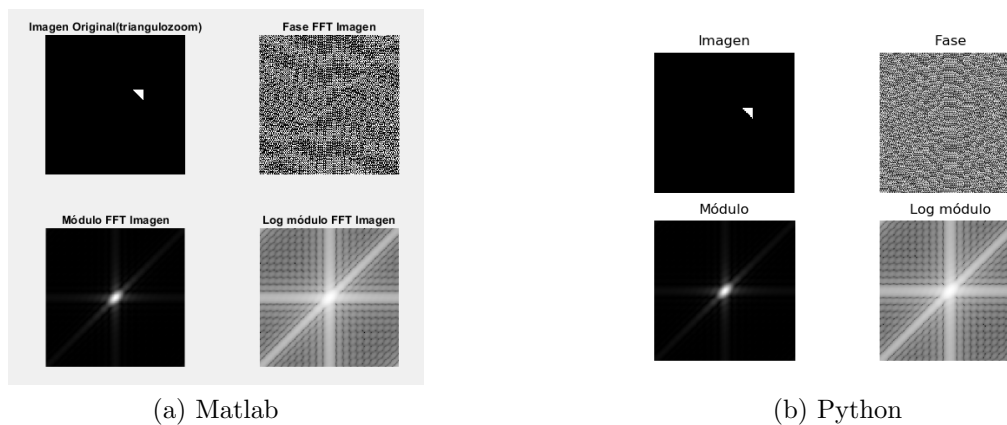


Figura 4.7: Comparativa de resultado en el apartado de propiedades de la transformada

4.3.2. Comparativa Matlab vs Python

En este ejercicio se usa una imagen dada con la práctica original de un triángulo blanco sobre un fondo negro.

La función más importante de esta práctica es **fft2**, la transformada de Fourier bidimensional. Existe un equivalente exacto en Numpy, **numpy.fft.fft2**. También existe equivalente para la función **fftshift** que modifica la representación de la imagen en frecuencia para que las frecuencias bajas se concentren en el centro de la imagen y las altas en los extremos.

Para las representaciones en 3D se ha usado una función sacada de Github de código abierto, que simplifica el código de forma que, para representar un gráfico, solo tienes que llamar a una función en lugar de meter todas las especificaciones de Matplotlib cada vez que se quiera representar algo. En el cuadernillo se deja el código demostrando como usar esta función.

El apartado de propiedades de la transformada de Fourier se mantiene igual de Matlab a Python.4.7

Al igual que en la práctica anterior hay una función que viene dada con los materiales de la práctica, en este caso es **lpfilter**, una función para generar filtros de tipo Guassiano o ideal. Se recrea con el mismo funcionamiento en Python. A la hora de filtrar, lo que en el espacio era una convolución (ir moviendo una máscara sobre cada pixel) en la frecuencia es una multiplicación punto a punto que se puede realizar con **numpy.multiply**.

El último apartado de filtros paso alto vuelve a usar la función **lpfilter** para después invertir el filtro de modo que actúe como un filtro paso alto. Para filtrar se repite el mismo proceso[4.8].

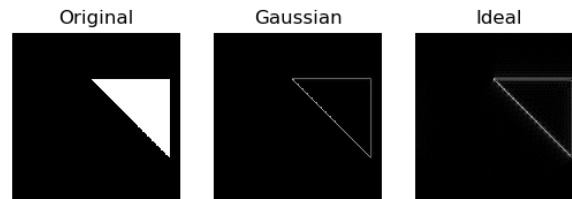


Figura 4.8: resultados del apartado sobre filtros paso alto

4.3.3. Desarrollo de funciones

En esta práctica se ha desarrollado una función **lpfilter** que como se ha mencionado en el apartado anterior genera filtros en frecuencia de dos tipos: Gaussiano e ideal. Como parámetros de entrada recibe el tipo de filtro que se quiere, las medidas del filtro que tienen que coincidir con las de la imagen que se quiere filtrar y D0 que es la apertura del filtro.

4.4. Práctica 4: Segmentación de imagen I

4.4.1. Teoría tratada en la práctica

Esta práctica se centra en entender algunos métodos de segmentación de imágenes usando el ejemplo de una calculadora. En el proceso de segmentar una de las teclas de esta calculadora se van usando varios métodos dados en la teoría de la asignatura.

El primero de estos métodos es umbralización usando el histograma, que consiste en elegir un valor como umbral a la hora de convertir una imagen en binaria de forma que queden en primer plano las partes de la imagen que nos interesan. En este caso, las letras de la calculadora.[4.9]

A continuación se explican la segmentación y caracterización de regiones. Segmentación consiste en dividir la imagen en diferentes conjuntos de píxeles



Figura 4.9: Imagen tras realizar la umbralización

llamados regiones, esas regiones forman la capa de segmentación. A cada una de estas regiones se le asigna una etiqueta, la etiqueta 0 se le suele asignar al fondo.

Hay varios tipos de segmentación, en este caso se va a usar segmentación binaria que busca grupos conexos de píxeles de primer plano y asigna etiquetas. No todas las regiones van a ser de interés ya que, en muchos casos la umbralización inicial no puede ser perfecta, por ejemplo, por ruido. Para decidir qué regiones son de interés se pueden mirar diferentes características como pueden ser la forma o el tamaño. En este caso se usa el área de la región para decidir cuales son de interés. Tras un proceso de filtrado la imagen queda como se puede ver en la figura 4.10

En este punto, cada letra es una región propia pero, lo que nos interesa segmentar son teclas. Se observa que la distancia entre letras de la misma tecla es muy inferior a la distancia entre letras en diferentes teclas. Para fundir las letras se usa un filtro paso bajo de media. Con esta nueva imagen se repiten todos los procedimientos anteriores hasta que se consigue segmentar solamente la tecla ENTER.[4.11]

4.4.2. Comparativa Matlab vs Python

En esta práctica hay muy pocas diferencias de Matlab a Python. El apartado de umbralización se mantiene igual, usando las funciones para el histo-

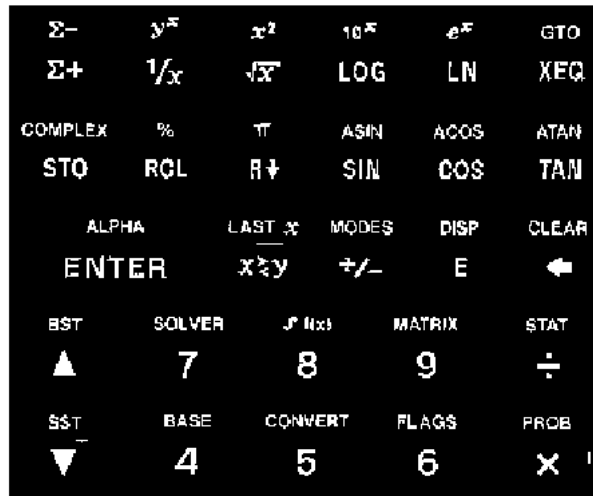


Figura 4.10: Resultado de filtrar regiones de area pequeña

grama y la umbralización que se utilizaron en la práctica 1.

En el apartado de segmentación encontramos la mayor diferencia entre los dos ejercicios que se debe al diferente funcionamiento de la función de segmentación.

En Matlab se usan dos funciones para la segmentación y la caracterización de regiones. La primera es **bwlabel** que crea las regiones con píxeles vecinos y devuelve una matriz del tamaño de la imagen original en la que a cada pixel se le ha asignado un número de acuerdo a su región, esta matriz es la capa de etiquetas. La segunda función es **regionprops** que recibe de entrada la capa de etiquetas y devuelve medidas de algunas características de la capa de etiquetas, por ejemplo, el área de cada región, la excentricidad o la posición de los píxeles extremos de cada región.

En OpenCV tenemos también dos funciones, aunque en realidad son la misma pero una tiene un argumento de salida más, son **connectedcomponents** y **connectedcomponentswithstats**. **Connectedcomponents** funciona igual que **bwlabel**. Para tener la funcionalidad de **regionprops** hay que usar **connectedcomponentswithstats** que, además de la capa de etiquetas, devuelve un array con algunas propiedades de las regiones. Aunque no afecta en esta práctica, el número de propiedades de las regiones que devuelve cada función es bastante diferente, OpenCV devuelve 4 propiedades: área, píxeles extremos, alto y ancho, mientras que Matlab devuelve más de



Figura 4.11: Resultado final de la práctica 4 en Python

20. En esta práctica vamos a usar la versión con stats, ya que nos interesa una característica específica de las regiones.

Para representar la capa de etiquetas en color Matlab tiene una función dedicada, mientras que en Python se puede usar cualquiera de los mapas de color disponibles de Matplotlib que generan el mismo efecto. En este caso se usa 'nipy-special' porque mantiene el fondo de color negro. [4.12]

El resto de la práctica usa la función de filtrado de media de la práctica 2 y repite los mismos pasos de los apartados anteriores.

4.5. Práctica 5: Segmentación de imagen II

4.5.1. Teoría tratada en la práctica

Esta práctica continúa el tema de la práctica anterior. Intenta segmentar una imagen, en este caso es la figura de un pájaro. Para realizar este proceso de segmentación se va a usar aprendizaje de máquina Kmedias.

Usa un sistema de ensayo/error con el objetivo de que el alumno entienda que no todos los métodos funcionan igual para todas las imágenes porque cada imagen tiene unas características diferentes.



Figura 4.12: Capa de etiquetas representada usando 'nipy-special'

El primer método que se intenta para segmentar la imagen es el utilizado en la práctica anterior: umbralización. Al ver el histograma de la imagen del Cormorán queda claro que la umbralización no va a ser posible ya que, no hay una distinción tan clara de tonos como en la imagen de la calculadora.

El siguiente método que se usa es el algoritmo K-medias aplicado sobre la imagen en RGB. Cogemos una descripción sencilla del funcionamiento de K-medias de la universidad de Oviedo: "K-means es un algoritmo de clasificación no supervisada (clusterización) que agrupa objetos en k grupos basándose en sus características. El agrupamiento se realiza minimizando la suma de distancias entre cada objeto y el centroide de su grupo o cluster. Se suele usar la distancia cuadrática." [Bibliografía]

La característica sobre la que vamos a agrupar estos datos es el color, en este caso se ordenan en una gráfica 3D donde cada eje es una componente de color en RGB[4.13].

Se decide usar 3 centroides ya que son los colores principales que se aprecian en la imagen (azul para el mar y cielo, amarillo para el tronco y marrón para el pájaro).

En la figura 4.14 se ve el resultado de la segmentación, se puede apreciar que tiene varios fallos, por ejemplo, gran parte de las alas del pájaro se han segmentado en el grupo asignado al tronco. Esto se considera sobresegmen-

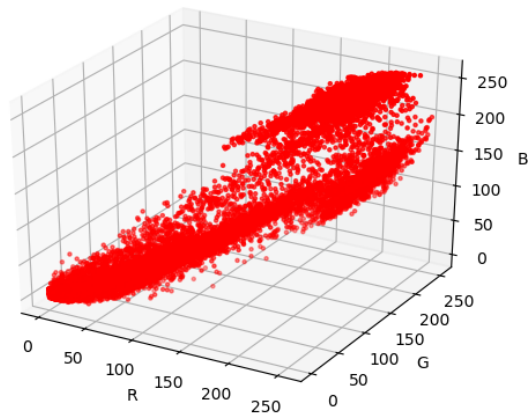


Figura 4.13: Representación de los píxeles de la imagen en gráfico 3D

tación.

Para solucionar el problema anterior se prueba un nuevo método, vuelve a usar K-medias, pero en lugar de organizar los datos en el espacio de color RGB se va a usar Lab. Lab es un espacio de color con tres componentes, la componente L indica la luminancia y las componentes a y b el tono de color.

En este ejercicio vamos a usar solamente las componentes ab ya que estamos tratando con tonos muy distintos y la luminancia no es tan relevante. Esto además simplifica los cálculos porque pasamos de usar 3 características a 2 [4.15].

El resultado vuelve a tener problemas, esto se debe a que los valores de a y b no están estandarizados lo que resulta en que una de las características domina sobre la otra. Una vez se estandarizan el resultado es el que se puede ver en la figura 4.16

En el cuarto apartado de la práctica se trata la caracterización por textura como alternativa al color. Para realizar esta caracterización se usan diferentes filtros con diferentes resultados en la segmentación también realizada con K-medias.

Por último, se prueba a usar varias características de distinta naturaleza, en este caso color y textura, esto resulta en la sobresegmentación de la imagen.[4.17].



Figura 4.14: Resultado de realizar la segmentación K-medias sobre RGB

4.5.2. Comparativa Matlab vs Python

El primer apartado de la práctica se ha mantenido idéntico ya que no requiere nada más que la función de representación del histograma.

En el siguiente apartado lo primero que se pide es recolocar los datos de la imagen de modo que queden en una matriz con 3 columnas y tantas filas como píxeles tiene la imagen. Cada columna contiene los datos de una de las componentes de color RGB, esto nos sirve para la representación en forma de scatter plot. Para convertir una matriz de dos dimensiones en un vector Numpy tiene la función de **reshape**.

Para aplicar kmedias vamos a usar la clase **Kmeans** de la biblioteca Scikit-learn. Primero se inicializa la clase indicando el número de centroides. Después se llama a la función **fit_predict** con nuestra matriz de datos como parámetro, esta función es la que ejecuta el algoritmo, por defecto se realizan un máximo de 10 iteraciones. En el atributo **cluster_centers** se guarda la posición final de los centroides, y en **labels** la capa de etiquetas que es una matriz con la misma forma que la matriz de datos. EL resultado de esta apartado es idéntico al de Matlab.

Para representar la capa de etiquetas hay que realizar las transformacio-

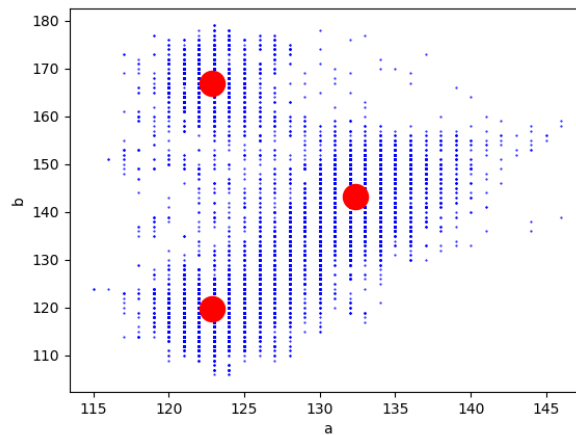


Figura 4.15: Representación de los píxeles de la imagen en gráfico con ejes a y b

nes inversas a las del inicio del apartado. En la figura 4.18 se puede ver las capas de etiquetas obtenidas en Matlab y Python.

Una diferencia en el apartado sobre el espacio de color LAB es la función usada para cambiar de espacio de color. En la práctica original la función viene dada con el enunciado de la práctica ya que no está implementada en Matlab. En OpenCV la función **CVTcolor** si que permite pasar de RGB a LAB.

En este apartado se pide la normalización de las variables, la secuencia de código necesaria viene dada en el enunciado, esto se ha replicado en el cuadernillo de Jupyter modificando el código para funcionar en Python.

El apartado sobre texturas es el que más cambios ha requerido ya que, las funciones de filtro de textura disponibles en Matlab no existen en OpenCV y han tenido que ser recreadas, los resultados son idénticos. En figura 4.19 se puede ver la comparativa de aplicar el filtro de entropía en Matlab y Python.

4.5.3. Desarrollo de funciones

En este ejercicio se desarrollan 3 funciones nuevas:

- **stdfilt**: esta función devuelve la imagen filtrada acorde a su variación estándar. Funciona de una manera similar a un filtro de media, con una máscara por defecto de tamaño 3X3 pero, en lugar de hacer la media con los píxeles debajo de la máscara calcula su desviación estándar.

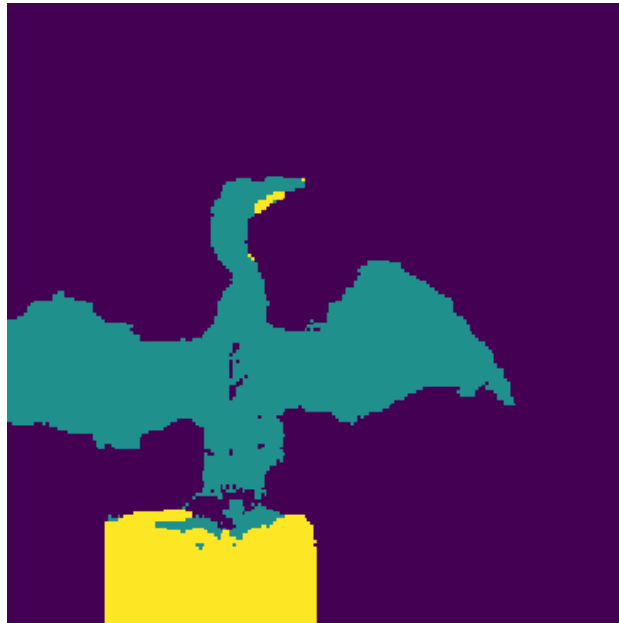


Figura 4.16: Resultado de realizar la segmentación K-medias sobre Lab

- `entropyfilt:`
- `rangefilt:`

4.6. Práctica 6: Morfología binaria

4.6.1. Teoría tratada en la práctica

Esta práctica se centra en el uso de la morfología matemática para la segmentación de una imagen. Los operadores morfológicos son una serie de operadores no lineales relacionados con la forma de los objetos en la imagen. Como en las dos prácticas anteriores se va a usar la segmentación de una imagen en concreto para demostrar el funcionamiento de estos operadores. En este caso la imagen es de un circuito impreso, el objetivo es segmentar los 7 chips.

Al igual que en la práctica anterior, lo primero que se va a hacer es decidir en qué espacio de color es más cómodo trabajar para esta segmentación. Tras probar en RGB se decide usar HSI, que es un espacio de color que define tono, saturación e intensidad. Mientras que en RGB las tres componentes se veían igual debido a la prominencia de tonos grises en la imagen, en HSI



Figura 4.17: Resultado de realizar la segmentación K-medias sobre componentes de color y textura



(a) Matlab



(b) Python

Figura 4.18: Comparativa de resultado en la capa de etiquetas

podemos apreciar una gran diferencia entre componentes. En la componente S se ven claramente diferenciados los chips, así que es la que se escoge para continuar el proceso de segmentación.

A continuación, se usa la umbralización para trabajar sobre una imagen

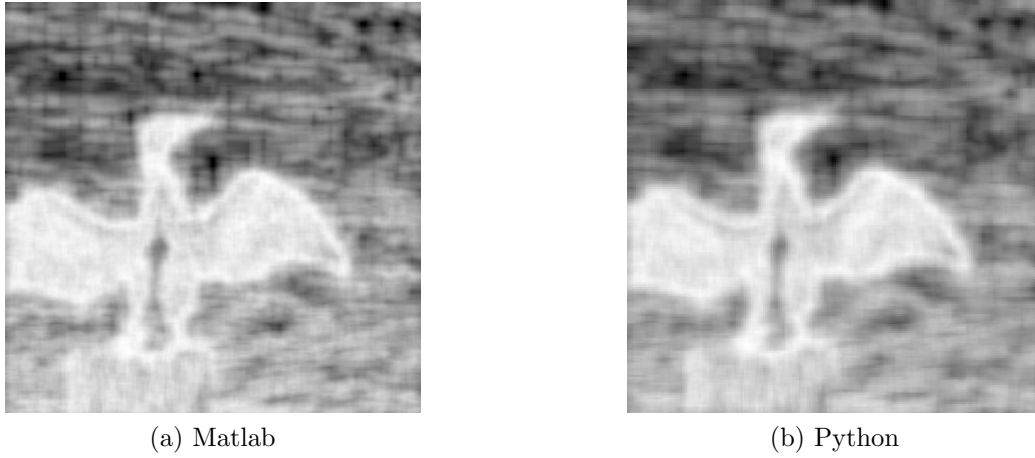


Figura 4.19: Comparativa de resultados de aplicar el filtro de entropía

binaria. Tras este proceso queda algún pixel blanco dentro de los chips que nos interesa eliminar. Se usa un filtro de mediana para homogeneizar la superficie de los chips.

En el siguiente apartado, se empiezan a usar operadores morfológicos. Contamos con 4 operadores morfológicos diferentes: erosión, dilatación, apertura y cierre. Para aplicar un operador morfológico se debe usar un elemento estructural. Los operadores morfológicos modifican los píxeles debajo del elemento estructural, la dilatación, por ejemplo, se queda con el máximo valor bajo el elemento estructural, la erosión el mínimo. La apertura y el cierre son elementos compuestos. La apertura es realizar una dilatación tras una erosión y el cierre lo opuesto.[4.20]

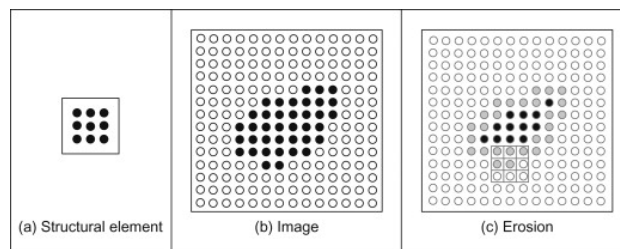


Figura 4.20: Ejemplo del funcionamiento del operador morfológico erosión

Para demostrar el efecto de cada uno de estos operadores morfológicos en la práctica se pide que se usen todos sobre la imagen binaria a segmentar

probando a cambiar el tamaño del elemento estructural. El objetivo es decidir qué operador morfológico es más apropiado para esta situación y terminar el apartado con una imagen en la que solamente están diferenciados los chips como la de la figura 4.21 que es el resultado de aplicar cierre sobre la imagen.

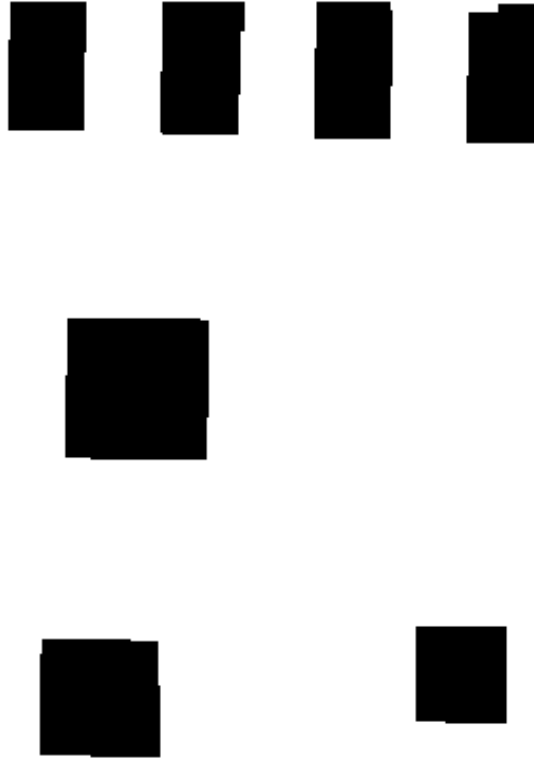


Figura 4.21: Resultado de aplicar cierre sobre la imagen

Para realizar la segmentación se invierte la imagen de forma que queden los chips en primer plano. Se segmenta la imagen usando segmentación binaria.

La práctica tiene dos apartados más: el primero pide que se use la excentricidad de las etiquetas para separar los chips cuadrados de los rectangulares, esto se hace de la misma manera que en la práctica 4 cuando se discriminaron etiquetas por su área. El segundo y último apartado pide que se usen operadores morfológicos para conseguir los bordes de los chips. Esto se consigue restando el resultado de la dilatación (que expande objetos) menos la erosión

(que reduce el tamaño).

4.6.2. Comparativa Matlab vs Python

A la hora de desarrollar el cuadernillo, la primera complicación se encuentra al pasar la imagen de RGB a HSI. En la práctica original, esta función viene dada con el enunciado. En OpenCV tampoco existe esta función por lo que se tiene que desarrollar de cero. El resultado es idéntico al de Matlab.

OpenCV tiene una función con un amplio rango de opciones para operadores morfológicos. Para la erosión y la apertura se usan funciones dedicadas (**erode** y **dilate**), mientras que para la apertura, cierre y otros operadores se usa la misma función (**morphologyEx**) indicando la operación a realizar con un parámetro. Para el elemento estructural se usa una matriz de Numpy.

Para diferenciar los chips cuadrados de los rectangulares nos encontramos con un problema: el enunciado pide que se use la característica excentricidad que es una de las características que devuelve la función **regionprops** de Matlab; como ya mencionamos hace un par de prácticas, el equivalente en OpenCV no devuelve esta característica, pero en este caso se puede solucionar de una manera relativamente fácil. La excentricidad se puede aproximar usando las características de largo y ancho del objeto que sí devuelve **connectedcomponentswithstats**. Con este sistema el resultado queda idéntico al de Matlab.

En el último apartado, que el objetivo es la detección de bordes, se usa un elemento estructural circular. Matlab tiene una función para crear elementos estructurantes de diferentes tamaños. OpenCV tiene una equivalente. El operador morfológico para detección de bordes se llama gradiente y la función **morphologyEx** permite aplicarlo directamente pero, ya que la práctica original pide que se razone cómo se obtendrían los bordes se ha decidido mantener el proceso tal y como estaba. Al final del cuadernillo se menciona la existencia de esta función como información adicional.

4.6.3. Desarrollo de funciones

Este ejercicio ha requerido el desarrollo de una función para cambiar del espacio de color RGB a HSI. Solo tiene un parámetro de entrada y uno de salida, la propia imagen en uint8. No es una función demasiado compleja ya

que solamente aplica las fórmulas matemáticas necesarias para esta transformación.[4.22]

RGB \rightarrow HIS

$$I = \frac{1}{3}(R + G + B)$$

$$S = 1 - \frac{3}{(R + G + B)}[\min(R, G, B)]$$

if $B \leq G$

$$H = \cos^{-1} \left[\frac{\frac{1}{2}[(R - G) + (R - B)]}{\sqrt{(R - G)^2 + (R - B)(G - B)}} \right]$$

else, $H = 360 - H$

Figura 4.22: Fórmulas para pasar de RGB a HSI

4.7. Práctica 7l

4.7.1. Teoría tratada en la práctica

4.7.2. Comparativa Matlab vs Python

4.7.3. Desarrollo de funciones

4.8. Práctica 8

4.8.1. Teoría tratada en la práctica

4.8.2. Comparativa Matlab vs Python

4.8.3. Desarrollo de funciones

4.9. Práctica 9

4.9.1. Teoría tratada en la práctica

4.9.2. Comparativa Matlab vs Python

4.9.3. Desarrollo de funciones

Capítulo 5

Conclusiones