

[en cada práctica pondría un puntero URL a tu repositorio donde tienes la última versión operativa de esa práctica]

Capítulo 4

[Para darle forma software, metería todas las funciones que has tenido que realizar en una biblioteca Python, que se usa en las prácticas nuevas junto con OpenCV... Y lo comentaría así en la memoria]

Prácticas de TDI

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

~~también~~

4.1. Estructura de las prácticas en Jupyter Notebook

A la hora de plantear las prácticas en los cuadernillos de Jupyter lo primero que se hizo fue decidir qué partes se iban a mantener igual y en qué se iban a diferenciar de las de Matlab. La estructura en sí, el temario tratado, el orden de los ejercicios y en los casos que se podía las imágenes originales se intentan mantener intactas. Lo más importante era mantener el cometido de la práctica, lo que se pretendía mostrar y enseñar con ella.

El cambio más obvio entre las dos versiones de los ejercicios tiene que ver con los enunciados, en los cuadernillos el enunciado está en inglés ya que se ~~había planteado~~ ^{planteó} el TFG para subir las prácticas a Robotics Academy, esto las haría más accesible ^Sa un mayor número de personas. Otro cambio en los enunciados es que han sido ampliados, esto se debe a que las prácticas originales son un material que se apoya sobre la teoría dada en clase y asumen que esa teoría ha sido impartida, mientras que en Robotics Academy las prácticas serían el único contacto que tendría el alumno con el tema por lo que los nuevos enunciados ~~ampliaban~~ ^{ampliaron} un poco las explicaciones teóricas añadiendo enlaces a más recursos para ampliar conocimientos.

estudiantes internacionales, no sólo de la URJC

Una cosa común en los enunciados originales son los trozos de código facilitados para copiar y pegar, en Jupyter estas líneas de código aparecen en celdas tipo *code* con indicaciones para cambiar los nombres de las variables si es necesario.

En cuanto al desarrollo de los ejercicios el cambio principal es el lenguaje de programación, de Matlab a Python.

Un cambio significativo en la presentación de la práctica es el uso de ventanas emergentes. Cuando ejecutas las prácticas en Matlab con la cantidad de imágenes que se pide representar pueden aparecer hasta 20 pestañas nuevas (Figura 4.1) lo que puede ser bastante abrumador. en la versión desarrollada en este trabajo se han ~~intentado~~ ^{evitado} evitar las ventanas emergentes, representando siempre que se ha podido en el propio cuadernillo, esto también hace las prácticas más fáciles de consultar una vez se ha terminado el código ya que se puede guardar el cuadernillo una vez ejecutado y las imágenes se mantienen mientras no se vacíe el output.



Figura 4.1: Ventanas emergentes en el apartado 2 de la práctica 8 en Matlab.

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. En una clase práctica de TDI siempre estaba la profesora, esto permitía que en caso de duda se le pudiese preguntar si iba bien la práctica enseñando la imagen resultado de un apartado en el que se

tuviera duda. Las imágenes respuesta pretenden sustituir esta opción para los alumnos que accedan a las prácticas fuera del contexto de la asignatura de TDI. **en la URJC.**

4.2. Guía de instalación

Para facilitar el acceso a las prácticas se ha creado una guía de instalación en *markdown* directamente accesible en Github desde el repositorio en el que están **la** prácticas.

La guía requiere que primero se compruebe si el ordenador tiene instalado Python3, en caso negativo, se pone un link a la página oficial de Python con la guía de instalación para los diferentes sistemas operativos.

A continuación, se indica **como** instalar Pip, el instalador de paquetes en Python y se indica el comando **para** instalar Pipenv.

La guía continúa explicando cómo acceder a la prácticas y como iniciar el entorno de Pipenv necesario para ejecutarlas.

Por último, se explica **como** abrir un cuadernillo de Jupyter importando las librerías necesarias en la primera práctica para comprobar que el entorno se ha instalado correctamente.

Esta guía está en inglés, al igual que las prácticas y pensada para ser usada y funcionar por igual en cualquiera de los principales sistemas operativos. Ha sido probada en Windows **y** Linux Ubuntu.

de Microsoft

4.3. Práctica 1: Introducción

4.3.1. Teoría tratada ~~en la práctica~~

en itálica

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~~^{segundo} 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 ^{sobre} 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~~^X la interpretación del color y cómo realizar transformaciones puntuales.

[para nombres de funciones no usaría negrita, sino texttt]

4.3.2. Comparativa Matlab vs Python

En este apartado ~~vamos a~~^{mos} mencionar^X 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~~^{que} se centra en la lectura y representación de imágenes, ~~podemos~~^{podemos} encontrar dos diferencias:

itemize

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~~^{que} al igual que Matlab genera una ventana nueva fuera del cuadernillo pero no tiene herramientas ~~y~~^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 convier-

ta 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 figura 4.2 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: $X = 255/n$ (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 4.2: 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.

4.3.3. 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.

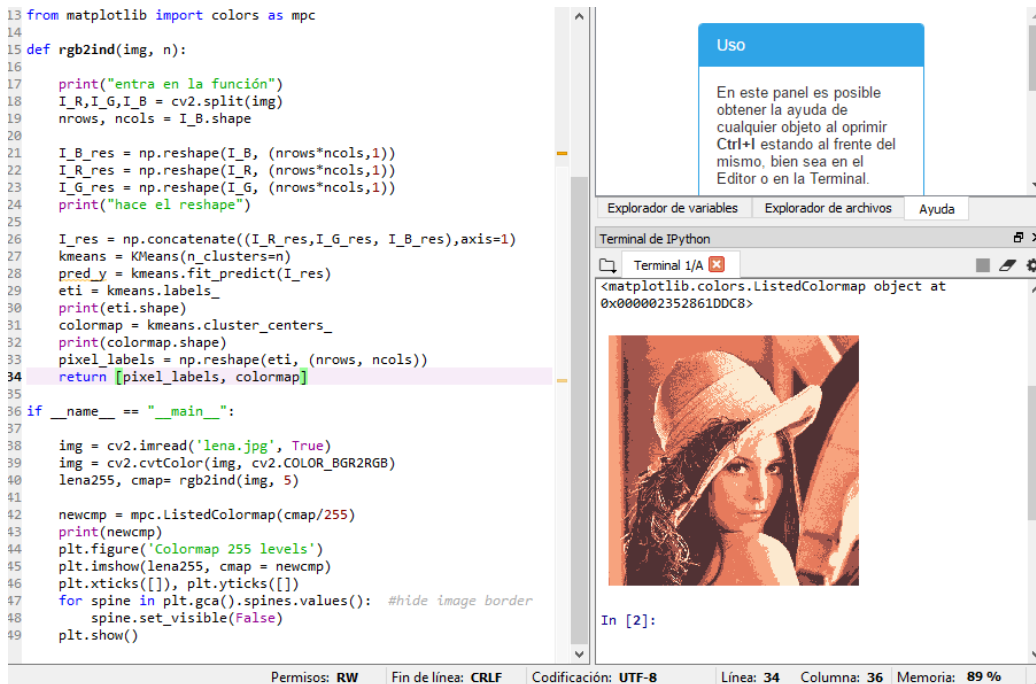
\texttt

itálica

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.

La figura 4.3 muestra cómo 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 4.3: Muestra de código y solución de RGB2ind.

4.4. Práctica 2: Filtrado espacial

4.4.1. 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 [4.4].

Según se aprecia en la Figura 4.4

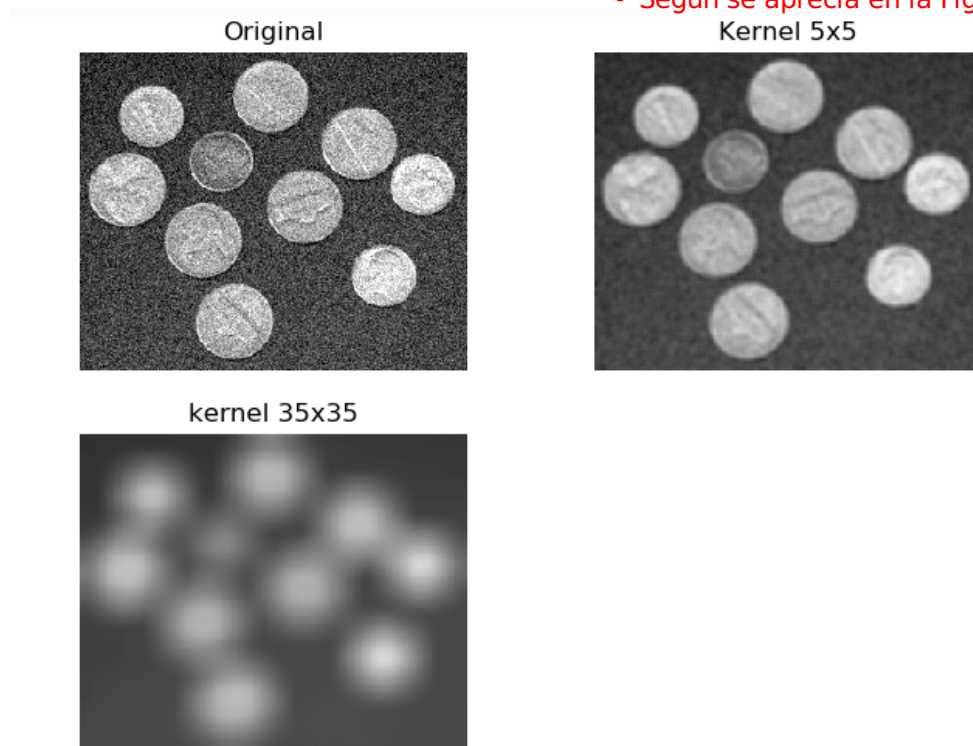
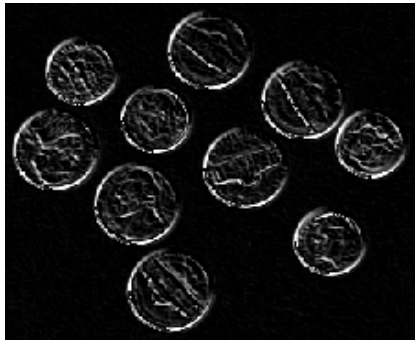
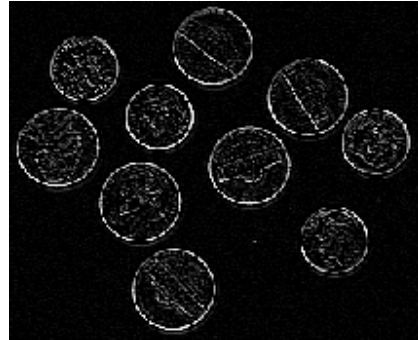


Figura 4.4: Imagen del cuadernillo donde se demuestra el filtro de media con diferentes tamaños de máscara.



(a) Prewitt.



(b) Isotrópico.

Figura 4.5: Filtros paso alto.

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 4.9

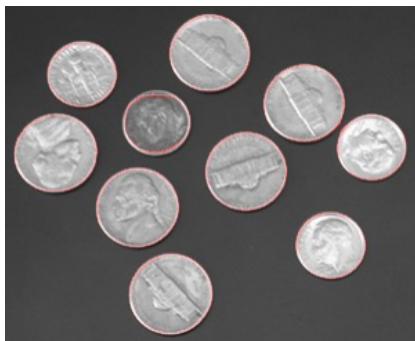
según se aprecia en la Figura 4.9

4.4.2. Comparativa Matlab vs Python

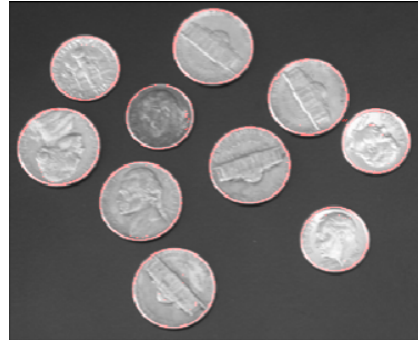
En el primer apartado de esta práctica se usan una serie de funciones que no pertenecen a Matlab y vienen entre los materiales de la práctica, estas funciones se recrean en Python con la misma funcionalidad.

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 cómo 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



(a) Matlab



(b) Python

Figura 4.6: Imágenes resultado del último apartado de la práctica 2.

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 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. 4.6

según se aprecia en la Figura 4.6

4.4.3. 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 imagen, por ejemplo, de **uint8** a **double** y viceversa.

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

4.5.1. 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 [Figura 4.7]. Se trata de filtrar la imagen en frecuencia y ver cómo afecta en el espacio tras hacer una transformada de Fourier inversa.

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

4.5.2. 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

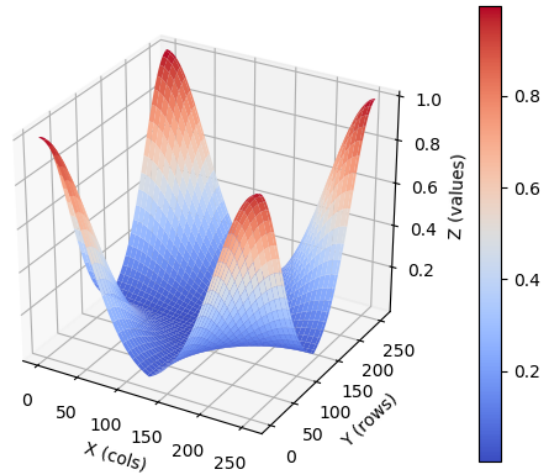


Figura 4.7: Representación de un filtro paso bajo Guassiano.

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. Como se ve en la figura 4.8.

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 [Figura 4.9].

, como muestra la

[imágenes un poco más grandes??]

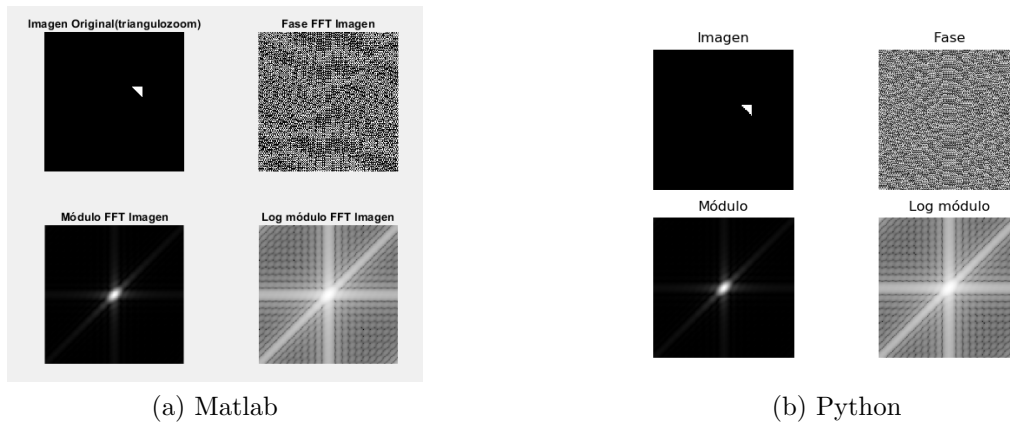


Figura 4.8: Comparativa de resultado en el apartado de propiedades de la transformada.

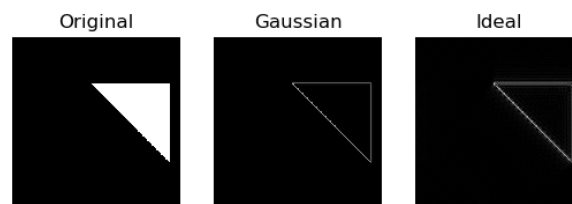


Figura 4.9: Resultados del apartado sobre filtros paso alto.

4.5.3. 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 D_0 que es la apertura del filtro.

4.6. Práctica 4: Segmentación de imagen I

4.6.1. 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 [Figura 4.10].



Figura 4.10: Imagen tras realizar la umbralización.

A continuación se explican la segmentación y caracterización de regiones. Segmentación consiste en dividir la imagen en diferentes conjuntos de píxeles 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 cuáles son de interés. Tras un proceso de filtrado la imagen queda como se puede ver en la figura 4.11.

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

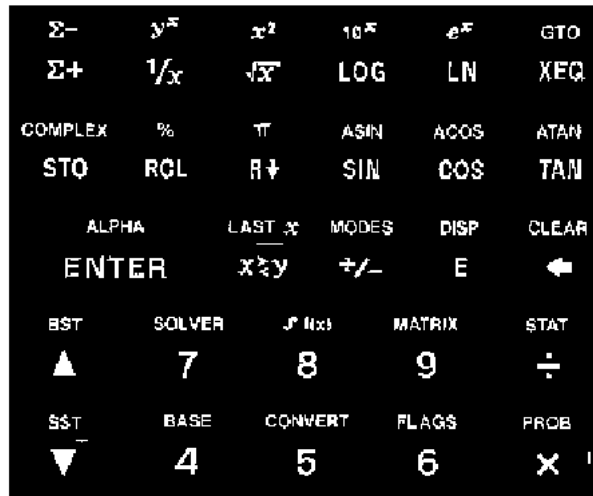


Figura 4.11: Resultado de filtrar regiones de área pequeña.

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 [Figura 4.12]

, tal y como se muestra en la

4.6.2. 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 histograma y la umbralización que se utilizaron en la práctica 1.

En el apartado de segmentación en Python nos encontramos con dos opciones: la función de segmentación de OpenCV y la de la librería Scikit-image. Se usa la de scikit image ya que funciona de manera mucho más similar a la de Matlab, sobre todo a la hora de conseguir las propiedades de las etiquetas. En OpenCV la función que devuelve propiedades de etiquetas es muy limitada con muy poca información, mientras que la de Scikit devuelve casi exactas las mismas propiedades que el equivalente en MatLab.

Se usan dos funciones para la segmentación y la caracterización de regiones. La primera es **bwlabel** en Matlab, **bwlabel** en scikit, que crea las regiones con píxeles vecinos y devuelve una matriz del tamaño de la imagen original en la que a cada píxel se le ha asignado un número de acuerdo a su región, esta matriz es la capa de etiquetas. La segunda función es **re-**

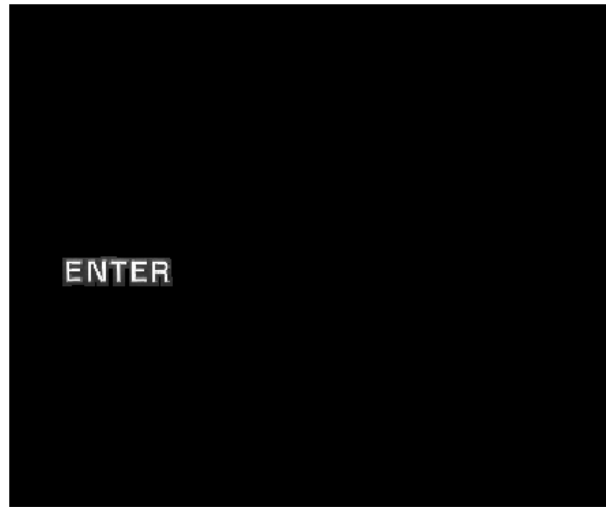


Figura 4.12: Resultado final de la práctica 4 en Python.

gionprops, se mantiene el mismo nombre en ambas versiones, 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.

La mayor diferencia entre **label** en Matlab y Scikit es el tratamiento de los píxeles de fondo. Matlab trata todo píxel de intensidad 0 como una única etiqueta de valor 0, mientras que Scikit, aunque en la imagen etiquetada pone los píxeles negros con la etiqueta 0, no lo cuenta como una etiqueta, por lo que, por ejemplo, en **regionprops** no tenemos la opción de recibir datos sobre los píxeles de fondo. A la función se le puede indicar que etiquete los píxeles negros pero no funciona igual que Matlab, ya que los trata como una etiqueta más, es decir, cuenta como etiquetas diferentes los píxeles negros no conexos. Esto no supone ningún problema en la realización de esta práctica excepto una ligera incomodidad ya que en el array devuelto por **regionprops** no coinciden los índices con el número de etiqueta en la imagen etiquetada, hay que sumarle uno al índice.

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.

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.

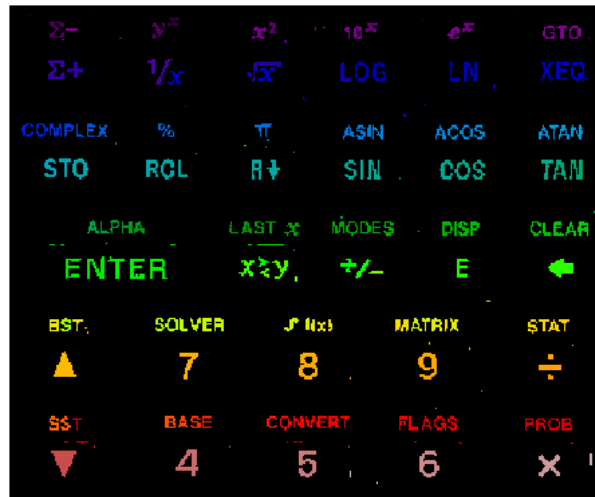


Figura 4.13: Capa de etiquetas representada usando *'nipy-special'*
italica

4.7. Práctica 5: Segmentación de imagen II

4.7.1. Teoría tratada ~~en la práctica~~ *dónde está la imagen original??*

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 K-medias.

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.

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
itálica

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 [Figura 4.14].

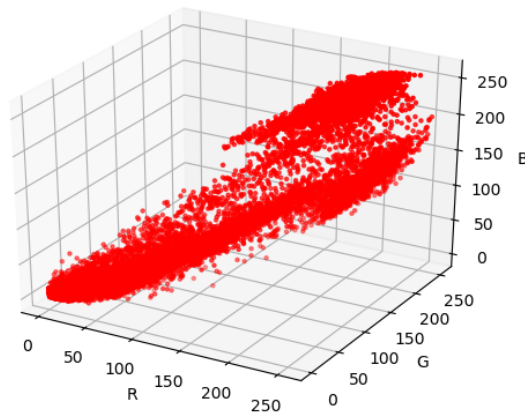


Figura 4.14: Representación de los píxeles de la imagen en gráfico 3D.

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~~ 4.15 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 sobresegmentación.

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



Figura 4.15: Resultado de realizar la segmentación K-medias sobre RGB.

a 2.

El resultado vuelve a tener problemas, esto se debe a que los valores de a y b no están estandarizados lo que provoca que una de las características domine 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. Los filtros de textura se relacionan todos con características estadísticas como la entropía, la desviación estándar o el rango. Se escogen dos de estas características para realizar la segmentación con k -medias, en este caso, la entropía y la desviación estándar. En la figura 4.17 se puede ver que los resultados no son buenos ya que segmenta los bordes del cormorán diferente del resto del cuerpo.

Por último, se prueba a usar varias características de distinta naturaleza, en este caso color y textura. Se cogen las características a y b del espacio de color Lab y la entropía, se estandarizan las 3 características y se aplica k -medias. La imagen resultante queda sobresegmentada con pequeñas regiones dentro del cuerpo del ave mal etiquetadas. Para solucionar esto se sugiere aplicar un filtro de media sobre las características de color. El resultado de

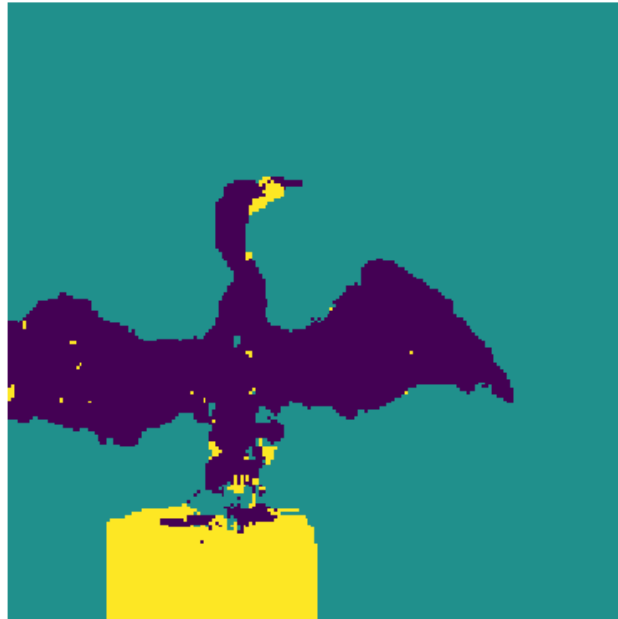


Figura 4.16: Resultado de realizar la segmentación K-medias sobre Lab.

la segmentación mejora como se puede apreciar en la **F**igura 4.18.

El pico del pájaro nunca quedará segmentado como parte del ave en este ejercicio, ya que en cuestión de color el pico se parece más al tronco sobre el que se posa el ave que al cuerpo.

4.7.2. 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.

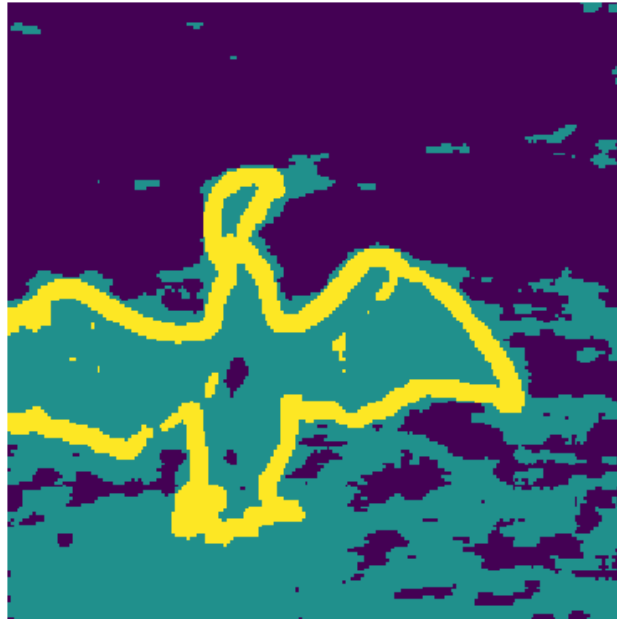


Figura 4.17: Resultado de realizar la segmentación K-medias sobre componentes de textura caracterizados por la desviación típica y la entropía.

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 este apartado es idéntico al de Matlab.

Para representar la capa de etiquetas hay que realizar las transformaciones inversas a las del inicio del apartado. En la figura 4.19 se puede ver las capas de etiquetas obtenidas en Matlab y Python.

En el apartado sobre segmentación en el espacio de color Lab tanto en Matlab como en Python se usa una función específicamente programada para este ejercicio para pasar de RGB a Lab. En ambos lenguajes existe una función que hace esta transformación, la razón por la que no se usa es el tipo de transformación Lab que utilizan. Las normas que definen un espacio de color respecto a otro se han ido revisando y cambiando con el tiempo. La versión de Lab con la que se planteó esta práctica es la descrita en la norma de 1976, mientras que la función que convierte de RGB a Lab en Scikit-image usa la norma de 1999.



Figura 4.18: Resultado de realizar la segmentación K-medias sobre componentes de color y textura.



(a) Matlab

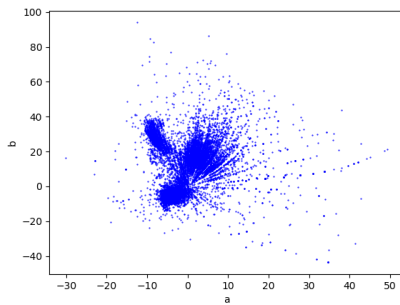


(b) Python

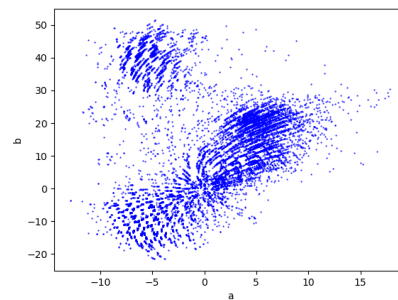
Figura 4.19: Comparativa de resultado en la capa de etiquetas

Para mantener la práctica tal y como se planteó se ha creado la función **RGB2LAB76** aunque también se ha hecho una versión de esta práctica con la función de Scikit-image. En la figura 4.20 se puede ver la representación de las componentes a y b en las diferentes normas y en la figura 4.21 se muestra

texttt

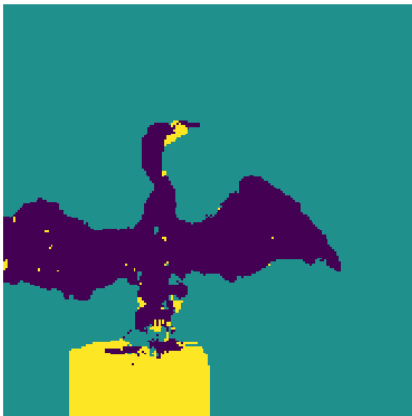


(a) Norma de 1976.



(b) Norma de 1999

Figura 4.20: Diferencias entre las componentes a y b en las diferentes normas.



(a) Norma de 1976.



(b) Norma de 1999

Figura 4.21: Resultados de la segmentación con las diferentes normas de Lab.

la diferencia en la segmentación con kmédias. ponlo siempre igual K-medias

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

En el apartado de texturas, Matlab tiene funciones para realizar los diferentes filtros necesarios. En Python existe un equivalente del filtro de entropía con resultados idénticos. Los otros dos filtros no tienen equivalente en Python por lo que se han tenido que programar. Los resultados son muy similares, ✓ como se puede comprobar en la Figura 4.22.

El último apartado utiliza herramientas ya empleadas en el resto de la

[imágenes un poco más grandes]

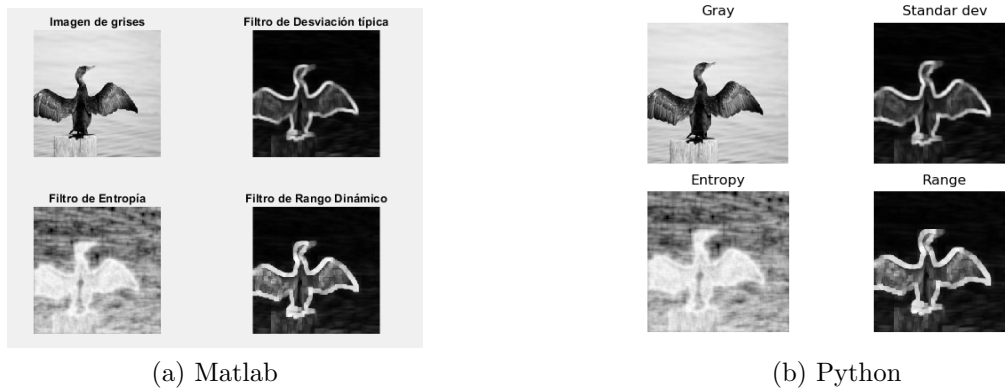


Figura 4.22: Comparativa filtros de textura en Matlab y Python.



Figura 4.23: Espacios de color RGB y CIE Lab.

práctica y un filtro de media explicado en la práctica 2.

4.7.3. Desarrollo de funciones

En este ejercicio se desarrollan 3 funciones nuevas:

- **RGB2Lab76:** Esta función realiza el cambio de espacio de color acorde a la norma de 1976. Primero se pasa de RGB a XYZ multiplicando cada componente por una serie de índices. Para pasar de XYZ a Lab se crean una serie de índices acotando los valores de cada componente de XYZ para sacar las coordenadas en el nuevo espacio de color. En la **F**igura 4.23 se pueden ver los espacios de color RGB y Lab.

texttt

texttt

- **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, ^{es} calcula su desviación estándar.
- **rangefilt**: Un filtro de rango devuelve el rango de valores bajo una máscara, ^{es} que ^{es} la diferencia entre el máximo valor y el mínimo. Para programar este filtro se han empleado herramientas morfológicas. Se usa la operación de erosión como un filtro para conseguir el mínimo bajo el elemento estructural y la dilatación como un filtro de máximo. La resta de los resultados de la dilatación y erosión devuelve el rango.

4.8. Práctica 6: Morfología binaria

4.8.1. 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.

[Imagen original??]

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 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 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, aper-

tura 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. [4.24]

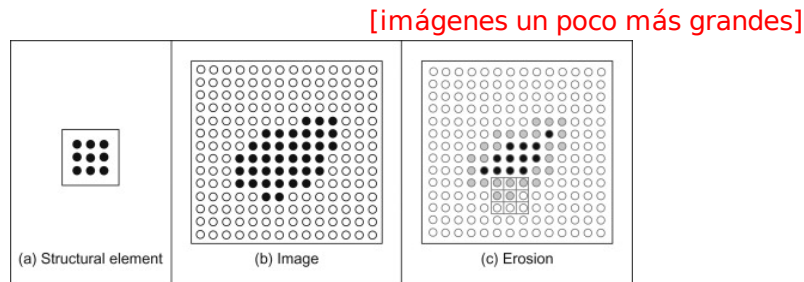


Figura 4.24: Ejemplo del funcionamiento del operador morfológico erosión.

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 **F**igura 4.25 que es el 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).

4.8.2. 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

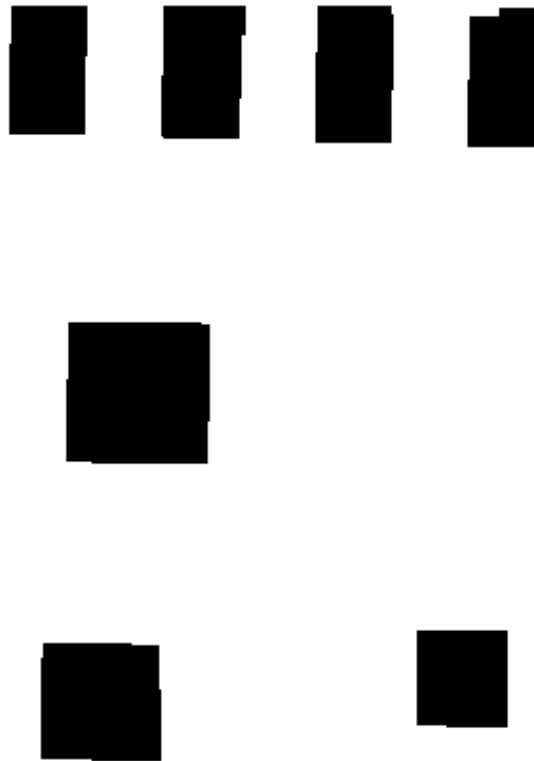


Figura 4.25: Resultado de aplicar cierre sobre la imagen.

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 se mantiene igual, ya que la función **regionprops** de scikit ~~image~~ también tiene la característica de excentricidad.

En el último apartado, cuyo objetivo es la detección de bordes, se usa un elemento estructural circular. Matlab tiene una función para crear elemen-

texttt —
 tos 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.

4.8.3. Desarrollo de funciones

texttt
 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. [4.26]

, que se muestran en la Figura

RGB → HIS

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

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

$$\text{if } B \leq G$$

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

$$\text{else, } H = 360 - H$$

Figura 4.26: Fórmulas para pasar de RGB a HSI.

4.9. Práctica 7

[un poco más descriptivo el título]

4.9.1. Teoría tratada en la práctica

Esta práctica trata el método de segmentación por *Watershed* con marcadores para contar el número de células enteras que hay en una imagen.

Watershed es un algoritmo de segmentación basado en morfología matemática que trata la imagen como una serie de desniveles topográficos que forman picos y valles (Figura 4.27) que se van inundando. Se cogen las depresiones de intensidad como marcadores de los que crece el agua (la etiqueta). El proceso de inundación continúa hasta que toda la imagen está inundada, donde se toca el agua de diferentes regiones se forman las líneas de *Watershed* que limitan las regiones.

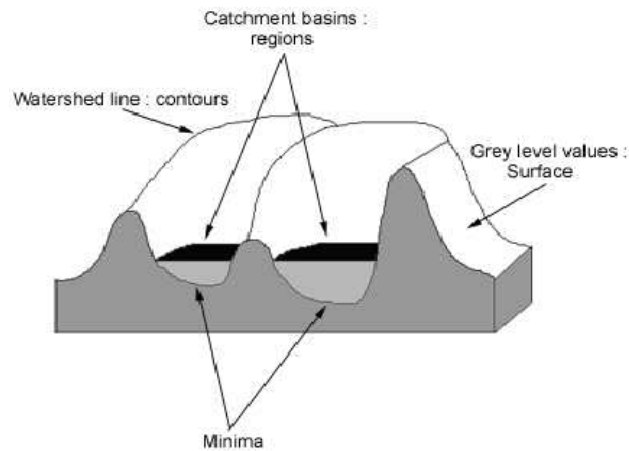


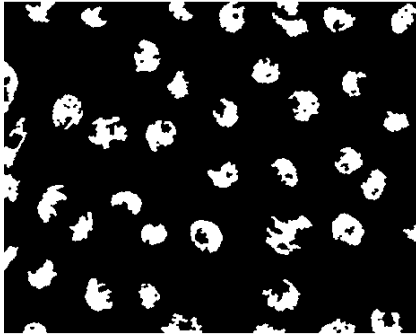
Figura 4.27: Ejemplo de *Watershed*.

Para que *Watershed* funcione de la manera deseada es necesario extraer los marcadores que indicarán los valles. Primero se extraen los marcadores internos, para ello se trata la imagen en escala de grises con un filtro secuencial alternado que se utiliza para suavizar la imagen. Los filtros secuenciales alternados consisten en realizar de forma secuencial una serie de operaciones morfológicas con diferente elemento estructural. En este caso, se alterna entre apertura y cierre con elementos estructurales con forma de disco de radios 1, 2 y 3.

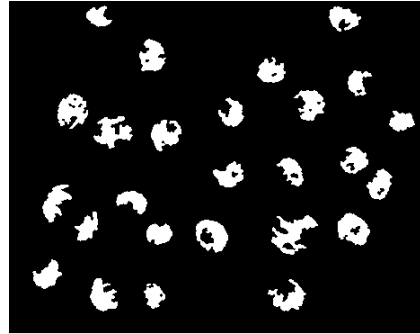
italica por anglicismo

A continuación, se va a usar reconstrucción, que es una operación morfológica que toma una imagen (marker) y la dilata repetidamente hasta que su forma coincide con otra imagen (máscara). Para crear el marcador aplicamos erosión sobre el negativo y usamos el propio negativo como máscara.

El siguiente paso es extraer los mínimos regionales. Un mínimo regional es una región (conjunto de píxeles conexos del mismo nivel) cuyo nivel es inferior al de todos los píxeles que la rodean. Esto devuelve una imagen bi-



(a) Máximos regionales.



(b) Máximos regionales procesados.

Figura 4.28: Proceso para conseguir los marcadores internos.

naria con más regiones que células en la imagen. Para reducir el número de regiones se usan 2 procesos diferentes:

Primero se limpian los bordes, no nos interesan células que no estén enteras, ya que, podrían aparecer en la siguiente imagen de microscopio y ser contadas dos veces.

El segundo proceso, consiste en comparar la posición de nuestras regiones con la posición de los núcleos de las celdas en la imagen original basándonos en que los núcleos aparecen en un color más oscuro. Se eliminan todas las regiones cuya posición coincida con un pixel demasiado claro en la imagen original para ser un núcleo.

El resultado de estos dos procesos se puede ver en la figura 4.28

Con eso ya tendríamos los marcadores internos. Para conseguir los marcadores externos (o de fondo) dilatamos la imagen de los marcadores internos para salir de la célula y hacemos una transformada de distancia que es una operación que devuelve una imagen en escala de grises donde el valor de cada pixel corresponde a la distancia a la que se encuentra de un pixel de primer plano. Los píxeles de primer plano en la transformada de distancia tienen valor 0.

Sobre la imagen de la transformada de distancia aplicamos *Watershed*. Los bordes que se forman entre regiones marcan los marcadores externos, siendo los puntos más alejados del nucleo de cada célula.

A continuación se crea una imagen que combina los marcadores internos

y externos (Figura 4.29).

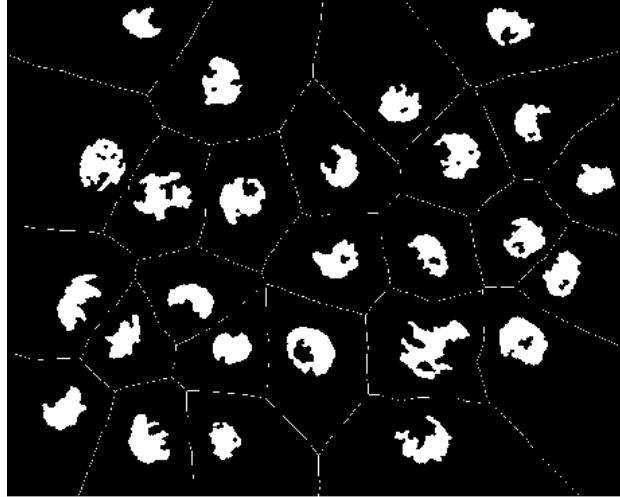


Figura 4.29: Marcadores internos y externos.

Para conseguir los bordes reales de las células se vuelve a aplicar *Watershed*, pero esta vez usando marcadores. Como imagen de referencia, en lugar de la transformada de distancia, se usa el gradiente de la imagen reconstruido de forma que nuestros marcadores queden como los únicos mínimos regionales y que los bordes de las células sean los puntos de mayor intensidad de la imagen mientras que el fondo y los núcleos, al tener menos variación, tienen menor intensidad. En la Figura 4.30 se puede ver el resultado final de esta práctica.

4.9.2. Comparativa Matlab vs Python

La primera diferencia en esta práctica entre Matlab y Python es la imagen empleada. La imagen original fue sacada de un artículo que no se ha podido localizar y se desconoce su licencia de copyright. Se ha encontrado otra imagen también de células muy cercanas las unas a las otras, en este caso de cebolla.

El primer apartado se mantiene igual en cuanto a funciones empleadas, pero los resultados no son los mismos. Esto se debe al tipo de padding que usan las funciones de apertura y cierre en Matlab y Python y que no se permiten modificar. En el caso de Matlab cuando hace erosión usa padding

itálica

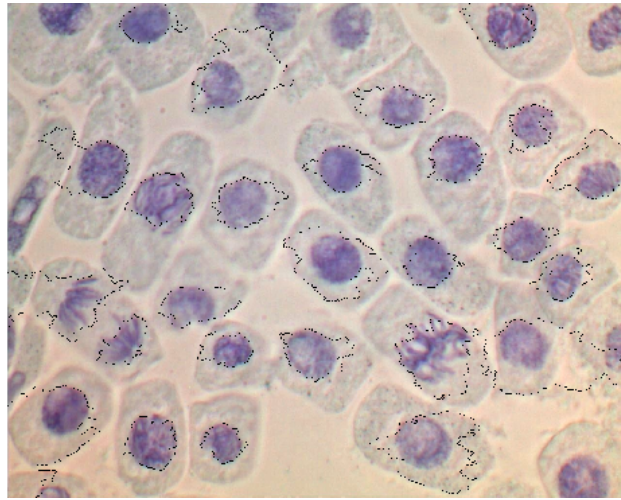


Figura 4.30: Resultado de la práctica de **Watershed**.

1, mientras que para dilatar usa ceros. En la función de OpenCV no indica el padding que usa pero, por los resultados, parece que usan *mirror padding*. Esto crea una diferencia en los valores cercanos al borde que en este caso no genera problemas pero que se tiene que tener en cuenta para otros casos.

Debido al cambio de imagen algunos de los valores empleados como elemento estructural se han tenido que cambiar, es el caso del elemento estructural usado para la erosión antes de la reconstrucción. Para la reconstrucción se encuentra una función en Scikit-image que funciona exactamente igual al equivalente en Matlab. **imreconstruct** en los dos casos recibe una imagen de máscara y una de marcadores.

ImregionalMax es una función de Matlab que devuelve los máximos regionales de una imagen. En Python no existe ninguna función equivalente por lo que se ha tenido que programar. La nueva función devuelve exactamente el mismo resultado que la de Matlab.

A continuación, se eliminan los marcadores que corresponden a las células del borde de la imagen que no están completas, en Matlab se usa la función **imclearborder**. En Python no hay versión en ninguna librería oficial, pero se ha encontrado una versión en Github subida por el usuario hatamiarash7 que cumple la misma función. *[referencia como bibliografía]*

Lo siguiente es eliminar los marcadores que no coincidan con un núcleo en la imagen original. En Matlab se usa la función **regionprops** con el atributo

mean intensity, **regionprops** en **scipy-image** tiene el mismo atributo con el mismo funcionamiento.

Para obtener los marcadores externos se usa la función de dilatar ya mencionada en la práctica anterior y la transformada de distancia para realizarla Matlab tiene la función **bwdist** mientras que OpenCV tiene **distanceTransform**. La principal diferencia entre estas dos funciones es que trabajan al revés: la de Matlab devuelve la distancia al punto de primer plano más cercano, mientras que la versión OpenCv devuelve la distancia a 0. Esto se resuelve pasando el negativo de los marcadores a la función en el cuadernillo de Jupyter.

A continuación, se aplica *Watershed* sobre esta imagen. En Matlab la función de **Watershed** recibe como argumento la imagen a segmentar. En Python tenemos dos opciones aunque ninguna funciona exactamente igual que la de Matlab.

La primera que se prueba es la versión de OpenCV que se descarta inmediatamente, ya que requiere una imagen RGB como primer argumento y la imagen que queremos pasar nosotros es en escala de grises.

La segunda opción es la de Scikit-image, también llamada **watershed**. Esta sí que permite imágenes en escala de grises. La principal diferencia con Matlab es que como segundo argumento pide una imagen binaria de marcadores y si no viene especificada como marcadores se eligen los mínimos locales de la imagen. En la versión de Matlab se cogen los mínimos regionales. En este primer uso de *Watershed* la diferencia no es relevante ya que coinciden los mínimos.

Otra diferencia, ya menor, es que la versión de Matlab automáticamente devuelve las fronteras señaladas en la imagen con un 0 de etiqueta, mientras que en la versión de Scikit-image se tiene que especificar que se requiere etiqueta para las líneas de *Watershed*.

Con esto ya tenemos los marcadores, tanto internos como externos. A continuación se usa un filtro paso alto para sacar el gradiente del mismo modo que se hizo en la práctica 2.

Y se vuelve a aplicar *Watershed*, esta vez la diferencia entre los mínimos sí que es relevante así que, en la versión de la práctica en Python, a la función de **watershed** además de pasarle el gradiente con marcadores como segundo

argumento se pasa la imagen binaria de marcadores.

4.9.3. Desarrollo de funciones

Para esta práctica se ha tenido que desarrollar la función **imregionalmax**. Esta función toma la imagen de la que se quiere conseguir los máximos regionales y se le resta 1. A continuación, se hace reconstrucción con la imagen original como máscara y la imagen a la que hemos restado 1 de marcador sobre el que se va realizando la dilatación repetida de la reconstrucción. Esto genera una imagen cuya única diferencia con la original son las regiones de máximos. Si restamos el resultado de la reconstrucción a la imagen original obtenemos una imagen en la que los únicos valores diferentes de cero son los máximos regionales. Esta imagen se pasa a binario y es lo que devuelve la función `imregionalmax`.

4.10. Práctica 8 [epigrafe más informativo]

4.10.1. Teoría tratada ~~en la práctica~~

[usar fotograma en vez de frame en toda esta sección??]

Esta práctica es la primera práctica de video. Tiene dos partes muy marcadas.

La primera es una introducción a las herramientas de video. Se ~~demue~~stra cómo leer un video, visualizar los *frames*, en qué consiste el *frame rate* y cómo afecta a la reproducción de video. Se pide que se modifiquen un par de *frames* con transformaciones como filtros y pasar a negativo para comprobar qué transformación es más perceptible.

La segunda se centra en demostrar la estimación de movimiento basada en bloques, específicamente 3 algoritmos: EBMA, HBMA y correlación de fase.

La estimación de movimiento basada en bloques se utiliza para codificación de vídeo. Estos algoritmos se dividen en dos grupos: algoritmos en el dominio del tiempo (EBMA y HBMA) y algoritmos en el dominio de la frecuencia (correlación de fase).

EBMA (*exhaustive block-matching algorithm*) consiste en una búsqueda exhaustiva por bloques comparando en el ~~frame~~ **fotograma** actual el bloque más similar

al del bloque de referencia. Es el menos eficiente de los algoritmos, pero el más preciso.

HBMA(*hierarchical block-matching algorithm*) funciona de una manera similar a EBMA, pero primero busca cambios en bloques más grandes y una vez localizadas las áreas de cambio reduce el tamaño de los bloques a comparar en esta nueva área. Esto hace que sea un algoritmo más eficiente.

El último algoritmo que se demuestra en esta práctica es el de correlación de fase, que trabaja en frecuencia comparando las fases de los dos ~~frames~~ **fotogramas**

Para comparar el funcionamiento de los algoritmos se calcula el MSE (*mean stimation error*) de cada uno.

4.10.2. Comparativa Matlab vs Python

En la primera mitad de esta práctica lo que genera la mayor diferencia es la forma de tratar el video entre los dos lenguajes de programación. Cuando lees un video en Matlab, usando la función **videoreader**, esta devuelve una variable que contiene todos los ~~frames~~ **fotograma** y otra información como el número de frames, el ~~framerate~~ **itálica** y el tamaño del video. Al llamar a la función **VideoCapture** de OpenCV lo que se crea es un objeto de la clase VideoCapture con una serie de métodos que en algunos casos devuelven la misma información que contiene la variable de video de Matlab.

En Matlab, lo primero que se pide es leer el video e inspeccionar la variable que se crea en el *workspace*. Para simular esto en Python se crea una instancia de la clase **VideoCapture** y se van guardando los ~~frames~~ **fotogramas** usando la función **read()**. **VideoCapture** en OpenCV permite capturar directamente video de la cámara que esté conectada al ordenador pero, en este caso, se usa el video 'shopping_center.mpg' que es el mismo que se usa en la versión de la práctica en Matlab.

A continuación, en la práctica de Matlab se piden una serie de características del video que se pueden obtener de la variable. En Python, se puede averiguar llamando al método **get()** de la clase VideoReader. Esto nos permite saber el ancho y alto de cada ~~frame~~ **fotograma** y el framerate. Conseguir el número de frames es algo más complejo, ya que el contador de frames del método *get()* es bastante inexacto y tiende a fallar. La mejor forma de saber el número de frames es mirando la longitud del array guardado en el

apartado anterior.

Posteriormente, se inspecciona el primer ~~frame~~ ^{fotograma}. En esto no hay diferencia con como se representan imágenes en prácticas anteriores.

A la hora de reproducir el video se puede usar **imshow** de OpenCV en un bucle pero esto abre una nueva ventana. Se intenta el mismo método con Matplotlib pero la reproducción no es fluida. Para solucionar esto, se usa una nueva librería: Bokeh. Es una librería de representación de gráficos que permite embeberlos en el cuadernillo. La reproducción de video queda fluida pero esta librería lee las imágenes con el eje girado y en el espacio de color RGBA por lo que hay que transformar cada ~~frame~~ ^{fotograma} antes de reproducirlo. Se usa un bucle while y la función **time.sleep()** de Python para que se pinte un nuevo ~~frame~~ ^{fotograma} cada 1/25 segundo, lo indicado por el ~~framerate~~ ^(itálica) del video.

El siguiente punto del ejercicio pide reproducir el video con un framerate distinto.

En la parte de estimación de movimiento, en Matlab, se utilizan 3 funciones programadas para la asignatura. En Python se hace lo mismo consiguiendo los mismos resultados.

En este apartado se pide usar el algoritmo EBMA con 3 tamaños diferentes de bloque (8, 16 y 32), representar el frame estimado, el mapa vectorial que devuelve la función y calcular el MSE para cada una de las 3 instancias. La figura 4.31 muestra el frame estimado en Matlab y en Python con un tamaño de bloque de 32.

A continuación se pide ejecutar HBMA con un bloque de tamaño 16 y 3 niveles.

Por último, ejecutar la función 'PhaseCorrelation' que, automáticamente, representa el frame estimado, el mapa vectorial y devuelve el MSE.

Las diferencias en este apartado vienen a la hora de representar los vectores de movimiento. La función **quiver** de Matplotlib representa mapas vectoriales pero, a no ser que se le indique de otro modo, el tamaño de las flechas varía para que la representación quede más visualmente interesante. En la ~~figura~~ ^F 4.32 se puede ver una comparación de la representación de los vectores de movimiento del algoritmo EBMA con bloques de tamaño 8 en Matlab, en Matplotlib con los valores por defecto y en Matplotlib con los



(a) Matlab

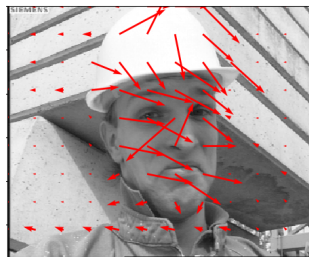


(b) Python

Figura 4.31: Comparativa de resultados EBMA con bloque 32.



(a) Matlab



(b) Python sin modificar



(c) Python mejorado

Figura 4.32: Diferencias en el mapa de vectores de movimiento.

valores corregidos. Los vectores utilizados son exactamente iguales.

4.10.3. Desarrollo de funciones

En esta práctica se han desarrollado 3 funciones, los algoritmos de estimación de movimiento.

- **EBMA:** Esta función recibe 3 atributos: el frame de referencia (anchor frame), el frame actual (target frame) y el tamaño de bloque que se va a usar. La función empieza declarando variables como el ancho y alto de cada frame y los arrays que van a formar el mapa de vectores de movimiento.

itálica

fotograma

A continuación, se itera por la imagen de referencia en saltos del ta-

maño del bloque, se define el bloque dentro del frame de referencia. Para buscar movimiento se va desplazando el bloque de referencia en diferentes posiciones sobre el frame actual buscando el mayor parecido en un número limitado de posiciones. Una vez encontrado el bloque más parecido se guarda en una matriz del mismo tamaño que los frames. Esta matriz será el frame estimado. Se guarda en el array de origen la posición del bloque de referencia y en el array de destino la posición en el nuevo frame. Esto se repite hasta que se ha iterado por toda la imagen.

- **HBMA:** Este algoritmo es muy parecido al anterior pero para mejorar el rendimiento se añaden unos pasos previos. La idea es primero usar bloques más grandes para descartar las áreas en las que no hay cambios. Para estos bloques grandes se crea una nueva imagen de la mitad de tamaño que la original y manteniendo el tamaño de bloque original. Se recorre esta imagen más pequeña y se guarda en un array la posición de las áreas que han cambiado.

A continuación, se hace EBMA sobre la imagen original con el tamaño de bloque pedido (16 en este caso) pero sólo en las áreas marcadas en el paso anterior. Esto quiere decir que en lugar de recorrer la imagen original entera como se hacía en la función anterior, sólo se recorren secciones específicas.

El tercer nivel funciona igual que el anterior: en las áreas de cambio detectadas en el paso anterior se hace EBMA sobre la imagen ampliada. Esto crea la sensación de que se ha usado un bloque de la mitad de tamaño esto hace que no se cambien más pixels de los estrictamente necesarios.

- **PhaseCorrelation:** Esta función también calcula un frame estimado a partir de dos frames dividiéndolos en bloques, la diferencia con los dos anteriores es que, en este caso, se trabaja en frecuencia. Primero se define una ventana con forma de campana que se va a usar para suavizar los bordes del bloque. A continuación se va dividiendo la imagen en bloques y haciendo la transformada de Fourier de esos bloques. Se calcula la coherencia entre el bloque actual y el bloque de referencia multiplicando el bloque actual por el conjugado del bloque de referencia y normalizando el resultado. Obtenemos la correlación de los dos

bloques haciendo la transformada inversa del resultado de la operación anterior. A continuación se localizán los máximos que indican dónde se encuentra la mayor diferencia entre los bloques. Para terminar se sustituyen los valores creando una nueva matriz que será el frame estimado.

4.11. Práctica 9

epígrafe más informativo

4.11.1. Teoría tratada en la práctica

La última práctica de la asignatura trata el uso de filtros en video.

La práctica se estructura de una forma muy similar a la práctica 2. Se empieza contaminando el video con ruido Gaussiano y sal y pimienta para, a continuación, filtrar el video y comprobar los efectos de los filtros temporales. En la figura 4.33 se puede ver el frame original, el frame contaminado con ruido gaussiano y con ruido sal y pimienta.



Figura 4.33: Un frame de video contaminado con dos tipos de ruido.

Se van a usar dos filtros temporales diferentes. Un filtro temporal es un filtro que en lugar de usar una máscara que coge píxeles del mismo frame cercanos al pixel que se está filtrando coge los píxeles en la misma posición de diferentes frames.

El primero de los filtros que se usa es un filtro temporal de media que, como indica su nombre, hace la media entre el número de frames indicado para filtrar. Al analizar el video resultante de estos filtros se comprueba que los primeros frames no quedan filtrados; esto se debe a que si, por ejemplo, estamos filtrando con 5 frames, dos anteriores al frame actual y dos poste-

rios no se puede empezar a filtrar hasta el tercer frame.

fotograma

El segundo filtro que usa es un filtro temporal de mediana que calcula la mediana de los frames que se estén usando. La peculiaridad en esta práctica es que no se da una función para realizar este filtro si no que se pide que se modifique la función usada para el filtro de media.

4.11.2. Comparativa Matlab vs Python

La primera diferencia entre la versión de Matlab y Python de esta práctica es el video usado. Se ha escogido un video sin copyright de una librería que almacena videos para uso didáctico.

Al igual que pasaba en la práctica 2 no existe una función para añadir ruido en Python, de hecho en Matlab, tampoco existe una función para añadir ruido a videos. Se crea la función **addnoise** para contaminar una secuencia de video RGB. Los resultados son los mismos que en Matlab.

Ni en Python ni en Matlab existen funciones para aplicar filtros temporales por lo que se tiene que programar **tempNoiseFilter** para realizar un filtro de media temporal.

Tanto la función **addnoise** como **tempNoiseFilter** crean nuevos videos que quedan guardados en la misma carpeta de la práctica como se puede ver en la Figura 4.34

| Nombre de archivo | Tamaño | Fecha |
|-----------------------|-----------|------------------|
| .ipynb_checkpoints | | 30/11/2020 11:04 |
| __pycache__ | | 30/11/2020 12:15 |
| filteredgrandma | 1,854 KB | 17/02/2021 10:37 |
| gaussgrandma | 5,435 KB | 03/12/2020 13:07 |
| noisegrandma | 5,438 KB | 17/02/2021 10:36 |
| 9_Videorestitution... | 26 KB | 17/02/2021 10:49 |
| P9.ipynb | 898 KB | 30/11/2020 11:04 |
| imfunctions | 4 KB | 30/11/2020 11:07 |
| videofunctions | 4 KB | 02/12/2020 10:43 |
| grandma_gcif.y4m | 32,304 KB | 30/11/2020 12:11 |
| videocutdit | 1 KB | 03/12/2020 13:00 |
| Practica9 | 125 KB | 23/11/2020 12:48 |

[imagen un poco más grande]

Figura 4.34: Contenido de la carpeta de la práctica 9 tras ejecutar addnoise y TempNoiseFilter.

4.11.3. Desarrollo de funciones

Esta práctica usa dos funciones principales: **addnoise** y **tempNoiseFilter**.

La función **addnoise** recibe como argumentos: el nombre del video a contaminar, el nombre que se le va a dar al video contaminado, el tipo de ruido y un array con los frames a contaminar. En el caso de que el array esté vacío se contaminan todos los frames. Esta función se sirve de varias funciones más pequeñas. Primero se usa **getframes** que recibe el nombre del video y devuelve un array con todos los frames. Tras una serie de comprobaciones (como que el array de frames no esté vacío y el número de frames a contaminar) se itera sobre cada frame llamando a la función **plusnoise** con el frame y el tipo de ruido y esta llama a la función **imnoise** usada en la práctica 2. La función **plusnoise** devuelve una imagen contaminada que se guarda en un nuevo array que formará parte de la nueva estructura de video contaminado. En la Figura 4.35 se puede ver el código de esta función.

La otra función **tempNoiseFilter**, al igual que en **addnoise**, sus primeros argumentos son los nombres del video de entrada y de salida. El tercer argumento de la función es el número de frames que se va a usar para calcular la media. Primero se lee el video usando **getframes**. A continuación se itera por el array de frames calculando la media entre el número de frames establecido. Al igual que **addnoise** guarda un nuevo video con el nombre indicado en el segundo argumento.


```
def addnoise(movie, out, noiseType, noisyframes):

    Frames = getframes(movie)
    fps = getfps(movie)
    newframes = Frames
    if len(Frames) < 1:
        print('error al leer el video')

    if not isinstance(noisyframes, list):
        print('noisyframes must be a list')

    if len(noisyframes) == 0:
        noisyframes = [i for i in range(len(Frames))]
    else:
        for item in noisyframes:
            if type(item) != int:
                print('noisyframes must contain integers')
            if item not in range(len(Frames)):
                print('frame number out of bounds')

    height = Frames[0].shape[0]
    width = Frames[0].shape[1]
    size = (width,height)

    for frame_number in noisyframes:

        newframes[frame_number] = plusnoise(Frames[frame_number], noiseType)

    out = cv2.VideoWriter(out,cv2.VideoWriter_fourcc(*'DIVX'), fps, size)
```

Figura 4.35: Imagen del código de la función addnoise.