Visión por Computador

Grado en Informática Industrial y Robótica (GIIROB)

Práctica 5:

Segmentación de imágenes en color

Contenido:

Autor: José Miguel Valiente González

Bibliografía:

- J. Howse & J. Minichino. “Learning OpenCV 4 Computer Vision with Python 3” (third Ed.) Packt Publishing Ltd. 2020.
- Sandigan Dei, “Python Image Processing. Cookbook”. Packt Publñishing Ltd. 2020.
- B.R. Hunt; R.L. Lipsman, J.M. Rosenberg. “A guide to MATLAB”. Cambridge University Press. 2001.
- Rafael Gonzalez, Richard Woods, Steven Eddins. ”Digital Image Processing Using Matlab”. Second Edition, McGraw Hill, 2010
- Official documentation for Python 3.10 https://docs.python.org/es/3/tutorial/index.html

En esta práctica nos vamos a centrar en las operaciones de **procesamiento de imágenes en color**, para lo cual vamos a revisar un conjunto de programas en **PYTHON** para este propósito.

Para ello vamos a utilizar varias técnicas de segmentación: Componentes de color y k_means.. De la técnica k-Means, a proponer la modificación denominada k-Means++. Otras técnicas de segmentación, como algunas de las que se muestra en el tema de teoría, se pueden encontrar en Apps específicas de Matlab o en funciones de OpenCV.

Objetivos

- Describir algunas de las funciones de segmentación de imágenes en color disponibles en las bibliotecas: **OpenCV**, **Matlpotlib** o **Pillow**.
- Desarrollar algunas de esas técnicas en Python.

Material

- Programas: IDE Thonny, Python 3.10 o similar, Matplotlib, OpenCV y Pillow.
- Archivos de imagen y *.py disponibles en PoliformaT.
- https://www.geeksforgeeks.org/machine-learning/types-of-machine-learning/
- https://www.geeksforgeeks.org/machine-learning/ml-mean-shift-clustering/
- https://www.geeksforgeeks.org/machine-learning/k-means-clustering-introduction/

1. Introducción

En la práctica anterior hemos visto que las bibliotecas **Pillow**, **Matplotlib, scikit-Image** y **OpenCV** tienen múltiples funciones para trabajar con imágenes. Ahora se trata de ver qué funciones pueden disponer para el proceso de segmentación de imágenes en color. Pero en esta práctica vamos a intentar implementar directamente alguna de estas funciones, en vez de sólo utilizar las disponibles. Excepto la segmentación basada en la media deslizante (Mean Shift), para la cual utilizaremos la clase y métodos disponibles en la biblioteca **scikit-learn.** OpenCV también dispone de múltiples herramientas, pero de momento nos quedaremos con las anteriores.

También recordar lo importante que es el uso de las ayudas**.** Dada la gran cantidad de funciones que incorporan todas estas herramientas, así como la diversidad de parámetros, es conveniente tener siempre abierta una ventana de ayuda para consultar durante el trabajo con el entorno.

2. Segmentación basada en componentes de color

Como ya se estudió en la práctica anterior, la umbralización por color es una potente técnica de segmentación de este tipo de imágenes. La idea principal consiste en utilizar alguna/as de las componentes de la imagen color RGB, o bien transformar a otros espacios como HSV, L*a*b, L*u*v, etc.., para discriminar los objetos o regiones de interés.

La técnica consiste en definir un rango de valores para las coordenadas de color en el cual se discriminan los objetos o partes de la imagen que puedan tener algún significado desde el punto de vista de la aplicación. Se pueden emplear coordenadas RGB, HSV, L*a*b, etc. El script '**color_thresholder.py**', disponible en PoliformaT nos facilita experimentar con esta idea. Al ejecutarlo nos permitirá seleccionar el directorio donde se localizan las imágenes de trabajo y a continuación aparecerá una ventana que nos muestra la primera imagen y a su derecha el resultado de su umbralización por color. El resto del conjunto de imágenes puede recorrerse con los botones "Anterior" y "Siguiente" (figura 1). Bajo la imagen se nos muestran sus histogramas para cada canal de color del espacio seleccionado (RGB, HSV o L*a*b*) y bajo cada histograma un deslizador nos permite seleccionar un rango de valores. Serán objetos o regiones de interés todos aquellos píxeles de la imagen que tengan unas componentes de color dentro del rango seleccionado en cada canal. Dichos objetos se mostrarán en color blanco, dejando el resto de la imagen resultante en color negro.

Se puede también hacer clic sobre la imagen original (o sobre la umbralizada) para seleccionar muestras de color que se marcan sobre los histogramas. De esta forma podemos seleccionar un conjunto de muestras (incluso de varias imágenes) con diferentes variaciones de color correspondientes a un determinado objeto que pretende segmentarse y ajustar los intervalos de selección de manera que incluyan a todo este conjunto de variaciones. Por último, se nos ofrece un control que permite simular alteraciones en la luminosidad de la imagen original (similares al efecto que provocaría un cambio en la intensidad de la fuente que iluminara la escena) para experimentar la sensibilidad a este factor de los parámetros de segmentación actualmente establecidos.

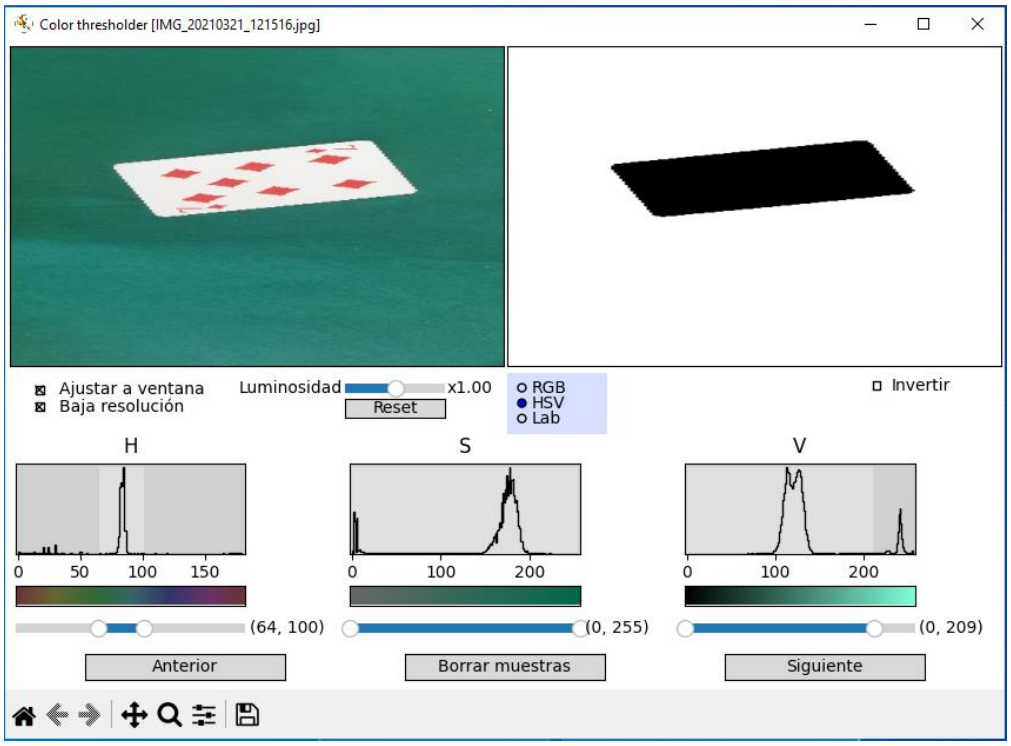

Figura 1 – Vista del programa 'color_thresholder.py' y resultado de la umbralización por rango de color en HSV.

Comenzaremos tomando las imágenes de los peces payaso (nemoXX.jpg) que pondremos en una carpeta "./images/" dentro de la carpeta de la práctica. Emplearemos el siguiente script para leer y visualizar cada imagen, junto su visualización en un espacio tridimensional de color (scatter)

El siguiente script muestra la forma de actuar en Python.

```
import matplotlib.pyplot as plt
from matplotlib import colors
import numpy as np
from PIL import Image
import cv2

nemo = cv2.imread("./images/nemo0.jpg")
fig = plt.figure()
fig.add_subplot(1, 2, 1, title="Original")
plt.imshow(nemo)

    # Plotting the image on 3D plot
r, g, b = cv2.split(nemo)
axis = fig.add_subplot(1, 2, 2, projection="3d")
pixel_colors = nemo.reshape((np.shape(nemo)[0] * np.shape(nemo)[1], 3))
norm = colors.Normalize(vmin=-1.0, vmax=1.0)
norm.autoscale(pixel_colors)
pixel_colors = norm(pixel_colors).tolist()

axis.scatter( r.flatten(), g.flatten(), b.flatten(), facecolors=pixel_colors, marker=".")
axis.set_xlabel("Red")
axis.set_ylabel("Green")
axis.set_zlabel("Blue")
plt.show()
```

Ejercicio 1 – Lectura de imagen y espacio de color 3D

1. Escriba el código anterior y pruebe el resultado.

Supuestamente deberá obtener una imagen de nemo con tonos rojos, sin embargo, el pez se muestra en tonos azules.

¿Podría indicar a qué es debido esto e introducir la solución en el script?

2. Repita lo anterior, pero trabajando con la imagen en HSV. Deberá convertir a HSV y utilizar la función:
   h, s, v = cv2.split(nemo_hsv)

   Puede también cambiar las etiquetas por “Hue”, “Saturation” y “Value”

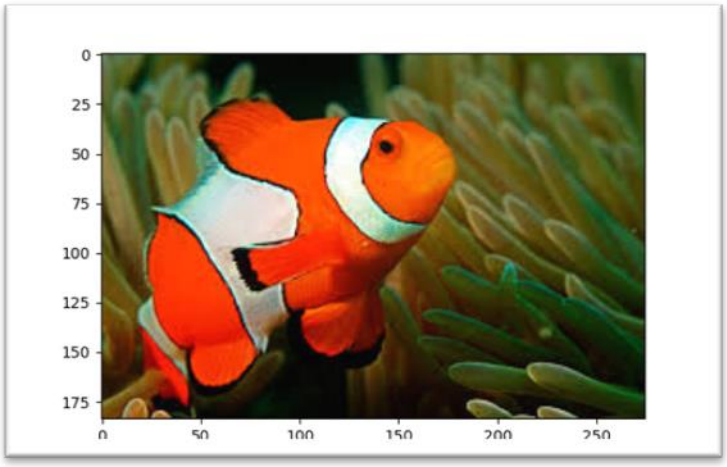

Figura 2 – Vista original de nemo.

A continuación, podemos obtener la segmentación por intervalos de color. Para esto definiremos el intervalo de los colores naranjas y emplearemos la función cv2.inRange() como se indica en el script adjunto. Se puede observar que lo que se obtiene es una imagen binaria (0..255). Después se aplica una función cv2.bitwise_and(nemo, nemo, mask=mask_orange) que obtiene un resultado que es igual a la imagen original en aquellos píxeles donde las dos imágenes indicadas son iguales y la máscara es distinta de cero. Al representarla obtenemos la región de la imagen original segmentada (Figura 3).

```
light_orange = (1, 190, 200)
dark_orange = (18, 255, 255)
     # Segment Nemo using inRange() function
Mask_orange = cv2.inRange(nemo_hsv, light_orange, dark_orange)

    # Bitwise-AND mask and original image
result = cv2.bitwise_and (nemo, nemo, mask=mask_orange)

plt.subplot(1, 2, 1)
plt.imshow(mask_orange, cmap="gray")
plt.subplot(1, 2, 2)
plt.imshow(result)
plt.show()
```

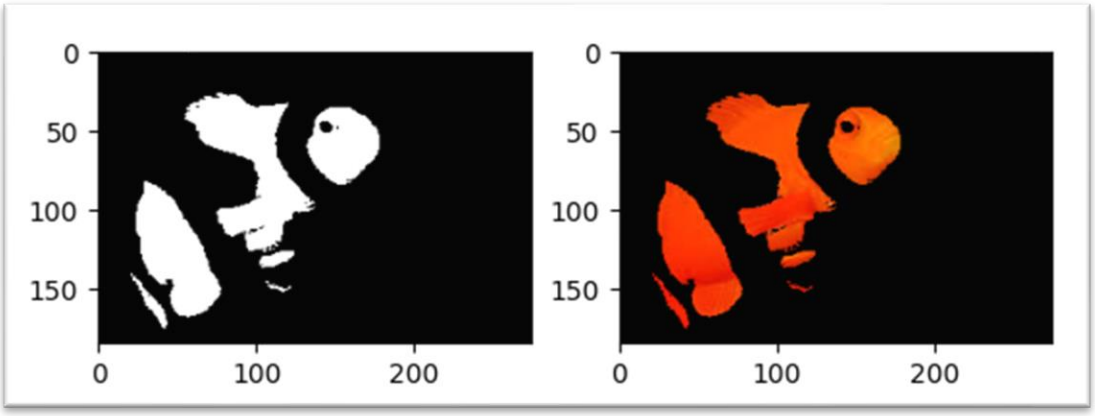

Figura 3 – Vista de la máscara de la segmentación y el original segmentado por dicha máscara.

Ejercicio 2 – Segmentación de los tonos naranjas

1. Modifique el programa del ejercicio 1 para incluir la segmentación indicada en el código anterior.
2. Compruebe que funciona bien con las 6 imágenes de nemo que hay en la carpeta "./images/".

Sin embargo, el proceso anterior tiene algunos inconvenientes:

- El proceso sólo puede seleccionar regiones de la imagen con el mismo rango de colores, pero puede haber varias regiones que pertenezcan al mismo objeto, con rangos de colores distintos.
- Hay que seleccionar unos rangos de colores que sean apropiados para un conjunto de imágenes similares de la aplicación, las cuales pueden variar de unas a otras.
- Hay que seleccionar los rangos manualmente.

Algunos de estos inconvenientes se pueden soslayar, aunque la selección manual parece inevitable. Por ejemplo, si hay 2 o más regiones con rangos distintos se pueden obtener las máscaras de discriminación para cada región y luego combinarles con operaciones lógicas (and). Por ejemplo, la imagen de nemo tiene dos regiones que forman el pez. Una es la región naranja antes obtenida y la otra en una región de tonos blancos.

Ejercicio 3 – Segmentación de los tonos naranjas y blancos

Supongamos que los tonos blancos están en el rango:
light_white = (0, 0, 200)
dark_white = (145, 60, 255)

1. Modifique el programa del ejercicio 2 para añadir una segunda segmentación de los tonos blancos y obtenga una máscara mask_white

2. Combine ambas máscaras y muestra el resultado incluir la segmentación indicada en el código anterior.
   final_mask = mask_orange + mask_white
   final_result = cv2.bitwise_and(nemo, nemo, mask=final_mask)

3. Compruebe que funciona bien con las 6 imágenes de nemo que hay en la carpeta "./images/".

El resultado previsible se muestra en la figura 4.

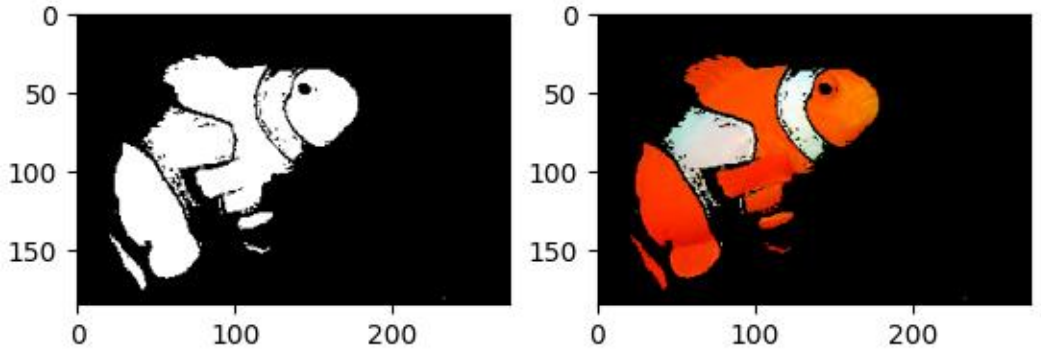

Figura 4 – Vista de la máscara combinada (naranja + blanco) y el original segmentado por dicha máscara.

Al final, la selección de los rangos de colores en dependiente de la aplicación por lo que en cada caso habrá que seguir un proceso similar al anterior.

Para segmentar de forma más automática habrá que seguir alguna técnica alternativa, como la indicadas en la teoría. Por ejemplo, veremos una técnica supervisada de agrupamiento de píxeles, denominada k-Means.

3. Segmentación basada en agrupamiento k-Means

La técnica k-Means es, en realidad, un método de clasificación supervisada, pero se puede también emplear como técnica de segmentación de los píxeles de una imagen en 'k' agrupaciones o clases.

El objetivo del algoritmo es organizar los datos en grupos (clusters), de tal manera que los miembros de cada grupo o cluster sean similares entre sí en cuanto a que sus características estén más próximas y diferentes a los de otro cluster. El procedimiento consiste en dividir N observaciones en k clusters, de manera que cada observación pertenece al cluster más cercano.

No es necesario conoce las clases reales a las que pertenece cada pixel, sólo el número total de clases 'k'.

El algoritmo contiene las siguientes etapas:

Entrada: k y conjunto de N datos $x_1, x_2, x_3, \ldots x_N$.

Paso 1 .- Inicializar aleatoriamente los k centros en el espacio de representación de los objetos.
Paso 2 .- Asignar cada dato al centro más cercano.
Paso 3 .- Recalcular las posiciones de los centros teniendo en cuenta las asignaciones anteriores.
Paso 4 .- Repetir los pasos 2 y 3 hasta que los centros ya no se muevas.

La figura 5 muestra un ejemplo sencillo de lo anterior.

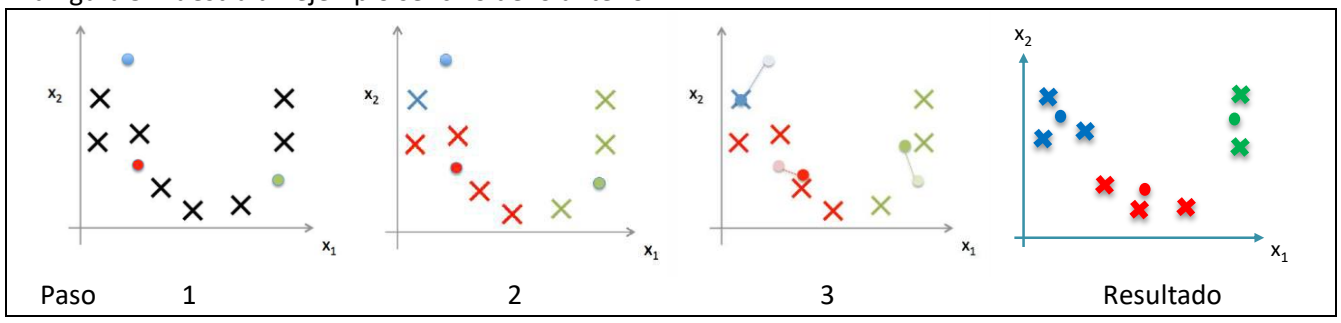

Figura 5 – Ejemplo del método k-Means.

Ejercicio 4 – Segmentación basada en k-Means

1. Tome el script 'k-means_alt.py' disponible en PoliformaT .
2. Ejecute el código con k=3, 4 y 5 clases y compruebe el resultado

Puede suceder que al aumentar el número de clusters se necesiten más colores en la función scater() Modifique la función si es necesario.

Observe que este script realiza bien el método k-Means arriba indicado, pero no se trata de datos de tipo imagen. Utiliza datos bidimensionales 'X' de tamaño (600,2).

► *Ejercicio 5 – Segmentación de imagen basada en k-Means*

1. Modifique el código anterior y cree el script "k-means-image.py" que permita hacer el algoritmo k_Means con datos de tipo imagen. Ahora los datos serán de tamaño (n_pixeles, 3) correspondientes a las 3 coordenadas de color (RGB, HSV, etc..).
2. En caso necesario, modifique la visualización en la función "plt.scatter()".
3. Utilice este algoritmo con las imágenes de nemo anteriormente indicadas.

   Es posible que sólo hagan falta 2 componentes de color (h,s) o similar. Pruebe esto para ver qué sucede.

El método k-means ha sido ampliamente utilizado durante los últimos 25 años. Sin embargo, tiene serios inconvenientes:

- Es lento para datos de tipo imagen en color debido al tamaño de estas.
- Es muy dependiente del número de clusters 'k'.
- Si la asignación inicial de los centroides no es acertada puede dar lugar a resultados no convenientes. Incluso pueden ser necesarios más o menos clusters.
- Una mala asignación inicial de centroides puede dar lugar a un número de iteraciones muy grande.

Mas recientemente, se han propuesto distintas alternativas para reducir los inconvenientes del k-Means. Básicamente se actúa sobre la selección inicial de los centroides, para comenzar con unos centros más apropiados. El algoritmo mejorado más difundido se denomina 'k-Means++'

Ejercicio 6 – Segmentación basada en k-Means++ [Opcional]

1. Busque en internet el algoritmo k-Means++ y documente como sería
2. Busque un script Python de ese algoritmo y pruébelo.

4. Segmentación basada en la media deslizante (Mean Shift)

El algoritmo de media deslizante es una aproximación no supervisada para segmentar imágenes en color que no requiere conocer el número de regiones u objetos a los que subdividir la imagen ni el valor de color de estas regiones o clases.

Básicamente, el algoritmo Mean Shift tiene los siguientes pasos:

- Paso 1.- Hacer un vector S con todos los datos.
- Paso 2.- Tomar aleatoriamente un dato de S como centroide inicial.
- Paso 3.- Obtener el resto de puntos de S dentro de un radio de vecindad 'h' y calcular el valor medio, que será un vector.

- Paso 4.- Modificar el centroide deslizándolo por ese vector media anterior. Todos los puntos utilizados se marcan con la misma etiqueta para que ya no vuelvan a ser utilizados.
- Paso 5.- Repetir los pasos 4 y 5 hasta que la media ya no se mueve.
- Paso 6.- Todos los puntos que se han atravesado tendrán la misma etiqueta y su valor será la del ultimo centroide hallado.
- Paso 7.- Todos los pasos desde 2 al 6 se vuelven a repetir hasta que ya no queden puntos sin etiquetar o se haya alcanzado un criterio de terminación

El algoritmo se puede utilizar con datos n-dimensionales. El único parámetro del algoritmo es el valor del radio h, también llamado 'bandwidth'. Edl problema de este algoritmo es la búsqueda de los puntos que están dentro de la vecindad de uno dado, lo cual requiere un proceso de 'fuerza-bruta'. Por eso se han implementado versiones que reducen ese proceso y utilizan un mejor esquema para buscar los centroides iniciales en el paso 2.

Dada complejidad de este algoritmo no vamos a proponer eralizarlo directamente, sino que emplearemos las funciones de scikit.leaarn o de OpenCV para esto.

En Scikit.learn la función es:

- class sklearn.cluster.MeanShift(*, bandwidth=None, seeds=None, bin_seeding=False, min_bin_freq=1, cluster_all=True, n_jobs=None, max_iter=300) -> ms

retorna un elemento de esa clase MeanShift. Este elemento dispone de los siguientes métodos:

- fit(X, y=None) # realiza clustering de los datos X
- fit_predict(X, y=None, **kwargs) # realiza clustering y devuelve las etiquetas de los clusters
- predict(X) # predice el cluster más próxima de cada dato de X
- set_params(**params) # defina los parámetros del algoritmo
- get_params(Deep = True) # devuelve los parámetros del algoritmo

Los atributos del elemento de la clase son:

- cluster_centers_: ndarray(n_clusters, n_features)
- labels_ : ndarray(n_samples)
- n_iter_: int
- n_features_in_: int
- feature_names_in_ : ndarray(n_features_in_,)

A modo de ejemplo tenemos el siguiente script en Python:

```
import numpy as np
import cv2
from sklearn.cluster import MeanShift, estimate_bandwidth

img = cv2.imread('./images/nemo0.jpg')

# filter to reduce noise
img = cv.medianBlur(img, 3)

# flatten the image
flat_image = img.reshape((-1,3))
flat_image = np.float32(flat_image)

# meanshift
bandwidth = estimate_bandwidth(flat_image, quantile=.06, n_samples=3000)
```

```
ms = MeanShift(bandwidth=bandwidth, max_iter=800, bin_seeding=True)
ms.fit(flat_image)
labeled=ms.labels_

# get number of segments
segments = np.unique(labeled)
print('Number of segments: ', segments.shape[0])

# get the average color of each segment
total = np.zeros((segments.shape[0], 3), dtype=float)
count = np.zeros(total.shape, dtype=float)
for i, label in enumerate(labeled):
    total[label] = total[label] + flat_image[i]
    count[label] += 1
avg = total/count
avg = np.uint8(avg)

# cast the labeled image into the corresponding average color
res = avg[labeled]
result = res.reshape((img.shape))

# show the result
Cv2.imshow('result',result)
Cv2.waitKey(0)
Cv2.destroyAllWindows()
```

Ejercicio 7 – Segmentación basada en la media deslizante (Mean Shift)

1. Copie el script anterior en un archivo 'MeanShift.py' y ejecútelo.
2. Compruebe el resultado.
3. Pruebe distintos valores del parámetro 'bandwidth' (20, 30, 40, ..).
4. Compruebe el resultado con otras imágenes de nemo.

La función en OpenCV se llama cv2.pyrMeanShiftFiltering(…) y se puede encontrar en la ayuda de esa biblioteca. No vamos a hacer ejemplos de esa clase.

5. Entrega de resultados

Para entregar los resultados de la práctica, haga un documento (Word o similar) con los distintos ejercicios incluyendo, en cada uno, lo siguiente:

- Enunciado del ejercicio
- Script de PYTHON (copy – paste)
- Imágenes de los resultados (menú *Edit->Copy figure* en la ventana y paste en el Word)
- Comentarios personales sobre el ejercicio y los resultados (si procede)

Incluya, en la página inicial, el título de la práctica y el nombre del/los alumnos implicados. Convierta el archivo a pdf y súbalo a PoliformaT al espacio compartido de cada alumno implicado.