Segmentación de imágenes usando espacios de color

Importamos librerías:

In [49]:
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import colorsys
import skimage
import plotly.graph_objects as go
from pathlib import Path
```

In [48]:
```
ASSETS_FOLDER_PATH = "./assets"
OUTPUT_FOLDER_PATH = "./results"
Path(OUTPUT_FOLDER_PATH).mkdir(parents=True, exist_ok=True)
```

Definimos las funciones a utilizar:

In [4]:
```
def plot_image(img, title):
    plt.imshow(img, cmap='gray')
    plt.axis('off')
    plt.title(title)
    plt.show()
```

In [5]:
```
def plot_images(img1, title1, img2, title2):
    fig = plt.figure(figsize=(6, 7))

    fig.add_subplot(1, 2, 1)
    plt.imshow(img1, cmap='gray')
    plt.axis('off')
    plt.title(title1)

    fig.add_subplot(1, 2, 2)
    plt.imshow(img2, cmap='gray')
    plt.axis('off')
    plt.title(title2)

    plt.subplots_adjust(wspace=0.05, hspace=0)
    plt.show()
```

Ej 1.a - Finding Nemo

Espacios de color al leer imágenes

Guardamos en `flags` las posibles conversiones de espacios de colores que nos ofrece OpenCV:

In [6]:
```
flags = [i for i in dir(cv2) if i.startswith('COLOR_')]
print(f"len: {len(flags)}")
```

```
print(f"first 3: {flags[:3]}")
```

```
len: 374
first 3: ['COLOR_BAYER_BG2BGR', 'COLOR_BAYER_BG2BGRA', 'COLOR_BAYER_BG2BGR_EA']
```

donde el `2` separa los espacios de colores origen y destino.

Al abrir la imagen del pez payaso, vemos que tiene los canales rojo y azul invertidos:

In [7]:
```
nemo_bgr = cv2.imread('./assets/nemo_images/nemo0.jpg')
plot_image(nemo_bgr, 'Nemo BGR')
```

Esto es porque OpenCV lee por default las imágenes en formato BGR. Podemos arreglarlo con `cvtColor(image, flag)` especificando el flag que convierte de BGR a RGB:

In [8]:
```
nemo_rgb = cv2.cvtColor(nemo_bgr, cv2.COLOR_BGR2RGB)
plot_image(nemo_rgb, 'Nemo RGB')
```

Nemo RGB

RGB vs HSV

Leemos la imagen en formato HSV:

In [9]:
```
nemo_hsv = cv2.cvtColor(nemo_bgr, cv2.COLOR_BGR2HSV)
```

Comparamos la distribución de colores de los pixeles de los espacios de color de RGB y HSV:

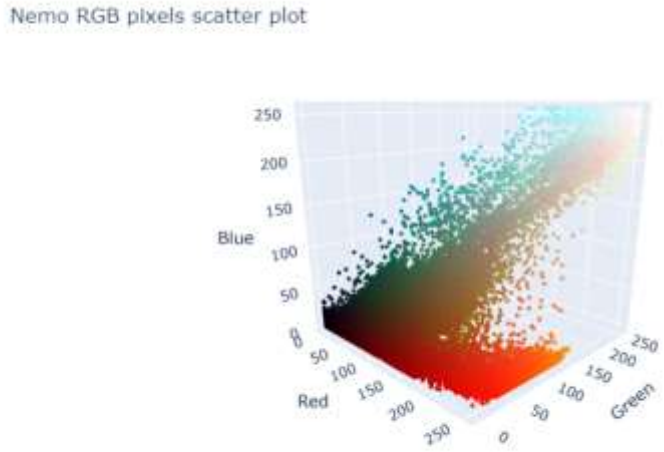
In [10]:
```
def rgb_3d_scatter(img, title):
    r, g, b = cv2.split(img)
    pixel_colors = img.reshape((np.shape(img)[0]*np.shape(img)[1], 3)) / 255.0
    fig = go.Figure(data=[go.Scatter3d(x=r.flatten(), y=g.flatten(), z=b.flatten
                                       mode='markers',
                                       marker=dict(color=pixel_colors, size=2))])
    fig.update_layout(title=title,
                      scene=dict(xaxis_title='Red', yaxis_title='Green', zaxis_t
                      scene_camera = dict(eye=dict(x=1.5, y=-1.5, z=0.5), center
                      margin=dict(t=60, b=35, l=25, r=0))
    fig.show()
```

In [45]:
```
def hsv_3d_scatter(img, title):
    h, s, v = cv2.split(img)
    rgb_values = [
        colorsys.hsv_to_rgb(h_val/179, s_val/255, v_val/255)
        for h_val, s_val, v_val in zip(h.flatten(), s.flatten(), v.flatten())
    ]

    fig = go.Figure(data=[go.Scatter3d(x=h.flatten(), y=s.flatten(), z=v.flatten
                                       mode='markers',
                                       marker=dict(color=rgb_values, size=2))])
    fig.update_layout(title=title,
                      scene=dict(xaxis_title='Hue', yaxis_title='Saturation', za
                      scene_camera = dict(eye=dict(x=1.5, y=-1.5, z=1), center=d
```

```
                      margin=dict(t=60, b=35, l=25, r=0))
    fig.show()
```

In [44]:
```
rgb_3d_scatter(nemo_rgb, 'Nemo RGB pixels scatter plot')
```

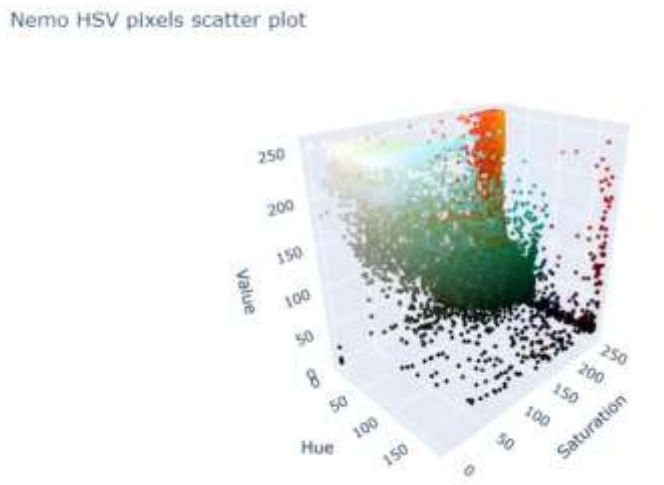
In [51]:
```
my_image = skimage.io.imread(fname=f"{OUTPUT_FOLDER_PATH}/scatter_rgb_nemo.jpg")
plot_image(my_image, '')
```



De esta forma, podemos notar por ejemplo que los pixeles naranjas con distintos niveles de brillo o saturación se encuentran repartidos en el espacio RGB, complicando el poder hallar un área definida que los englobe para separarlos del resto.

In [13]:
```
hsv_3d_scatter(nemo_hsv, 'Nemo HSV pixels scatter plot')
```

In [52]:
```
my_image = skimage.io.imread(fname=f"{OUTPUT_FOLDER_PATH}/scatter_hsv_nemo.jpg")
plot_image(my_image, '')
```



Con HSV podemos separar mejor los naranjas, ya que se encuentran localizados en un área determinada. Esta ventaja se debe a que el espacio RGB no tiene una separación clara entre la información de color y la de brillo, lo que puede complicar la segmentación cuando hay variaciones de iluminación como vimos con los naranjas. El canal Hue en

HSV permite identificar colores con mayor precisión sin importar su nivel de brillo o saturación, permitiendo segmentar más facilmente.

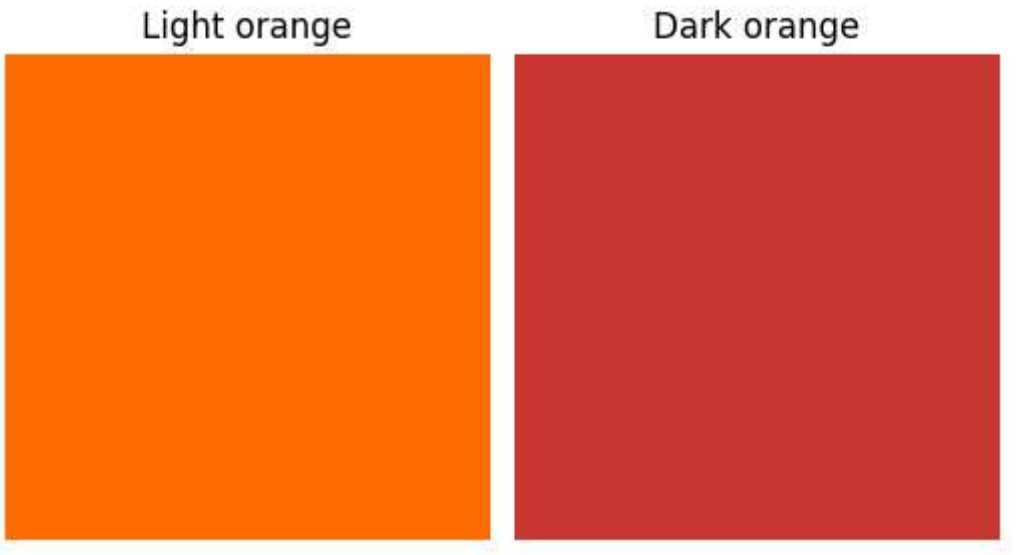
Creamos las máscaras

In [14]:
```
from matplotlib.colors import hsv_to_rgb
```

In [15]:
```
def plot_colors(color1, title1, color2, title2):
    lo_square = np.full((10, 10, 3), color1, dtype=np.uint8) / 255.0
    do_square = np.full((10, 10, 3), color2, dtype=np.uint8) / 255.0
    plt.subplot(1, 2, 1)
    plt.imshow(hsv_to_rgb(do_square))
    plt.axis('off')
    plt.title(title1)
    plt.subplot(1, 2, 2)
    plt.imshow(hsv_to_rgb(lo_square))
    plt.axis('off')
    plt.title(title2)
    plt.subplots_adjust(wspace=0.05, hspace=0)
    plt.show()
```

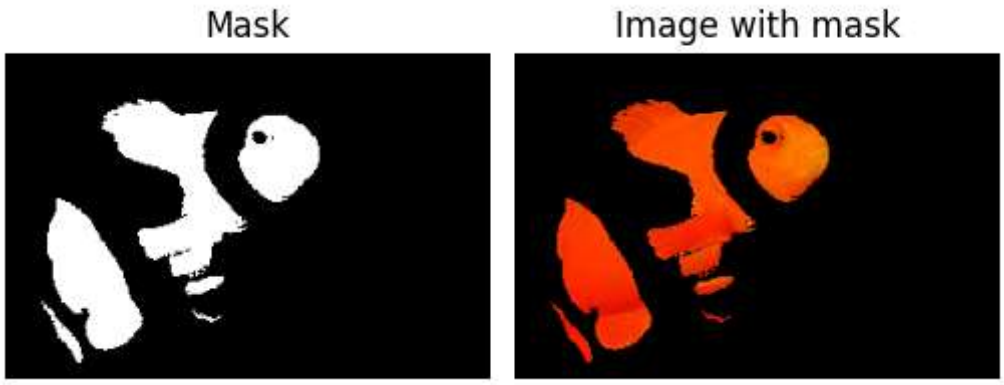
Tomamos dos colores como límites inferior y superior:

In [16]:
```
light_orange = (1, 190, 200)
dark_orange = (18, 255, 255)
plot_colors(light_orange, "Light orange", dark_orange, "Dark orange")
```



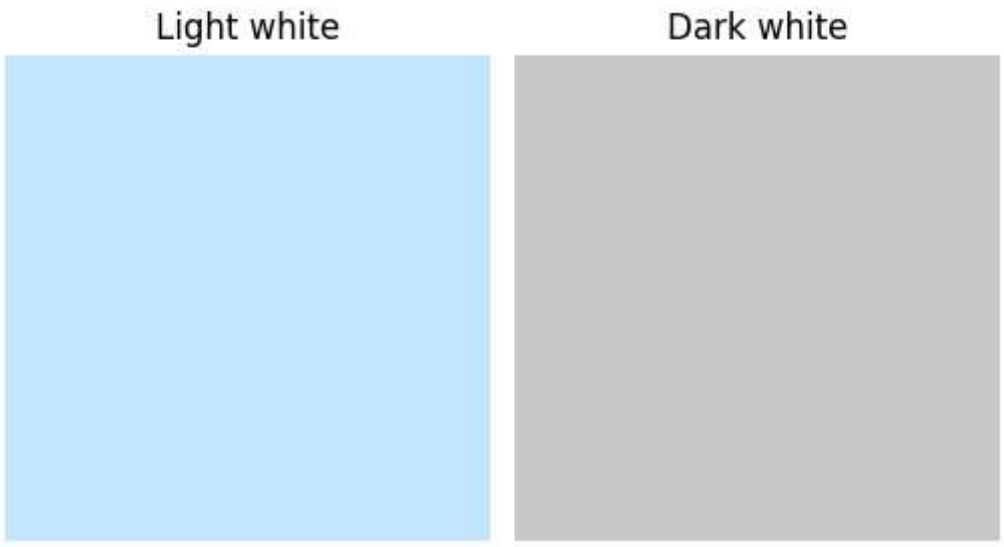
In [17]:
```
def plot_mask_result(mask, result):
    plt.subplot(1, 2, 1)
    plt.imshow(mask, cmap="gray")
    plt.title("Mask")
    plt.axis('off')
    plt.subplot(1, 2, 2)
    plt.imshow(result)
    plt.axis('off')
    plt.title("Image with mask")
    plt.subplots_adjust(wspace=0.05, hspace=0)
    plt.show()
```

In [18]:
```
mask = cv2.inRange(nemo_hsv, light_orange, dark_orange)
result = cv2.bitwise_and(nemo_rgb, nemo_rgb, mask=mask)
plot_mask_result(mask, result)
```

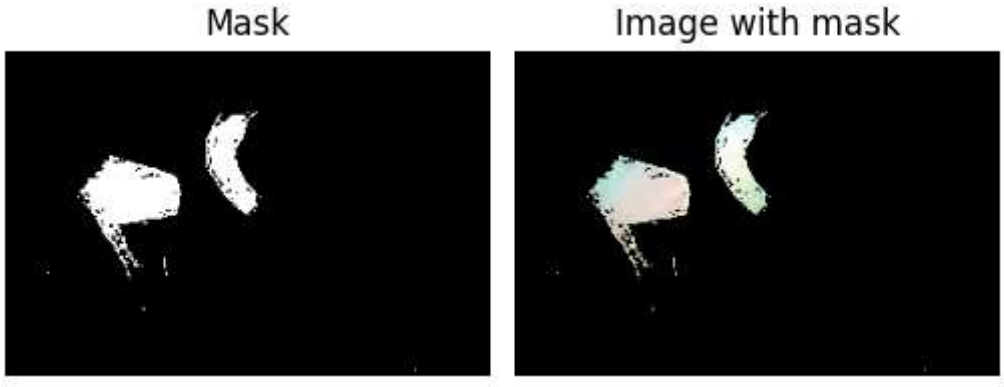


Aislamos también las rayas de Nemo creando una máscara para colores blancos:

In [19]:
```
light_white = (0, 0, 200)
dark_white = (145, 60, 255)
plot_colors(light_white, "Light white", dark_white, "Dark white")
```

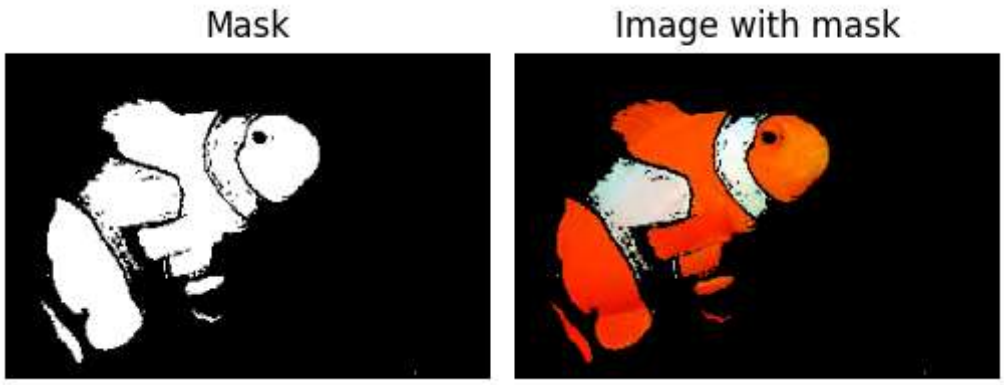


In [20]:
```
mask_white = cv2.inRange(nemo_hsv, light_white, dark_white)
result = cv2.bitwise_and(nemo_rgb, nemo_rgb, mask=mask_white)
plot_mask_result(mask_white, result)
```



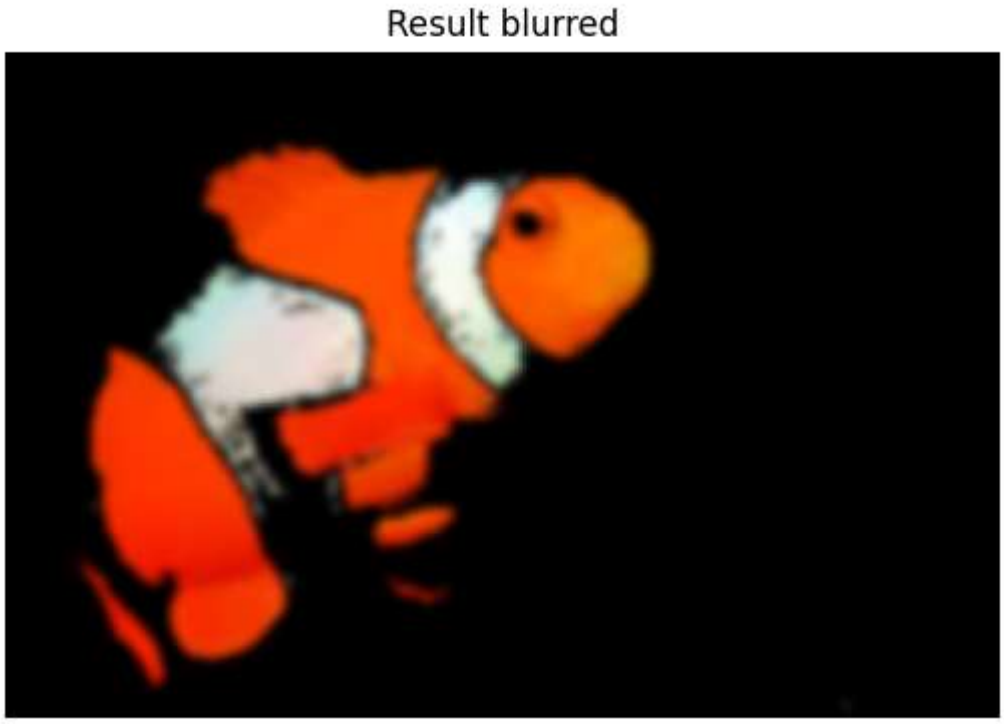
Combinamos las máscaras:

In [21]:
```
final_mask = mask + mask_white
result = cv2.bitwise_and(nemo_rgb, nemo_rgb, mask=final_mask)
plot_mask_result(final_mask, result)
```



Para mejorar los bordes, aplicamos Gaussian Blur:

In [22]:
```
result_blur = cv2.GaussianBlur(result, (7, 7), 0)
plot_image(result_blur, "Result blurred")
```



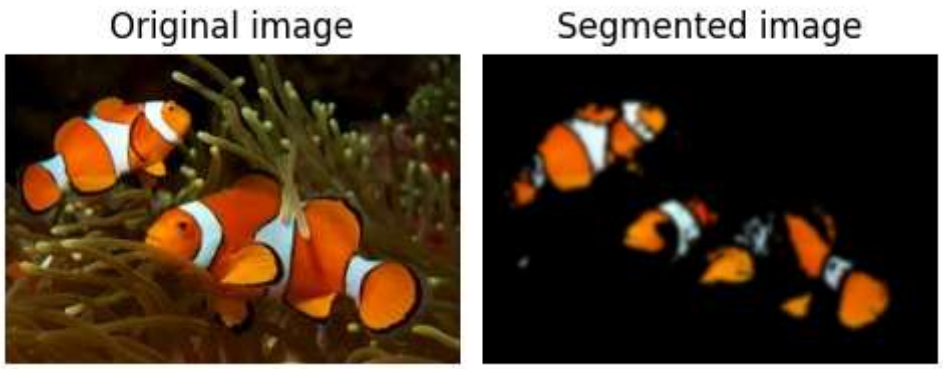
Aplicación de la segmentación a otras imágenes

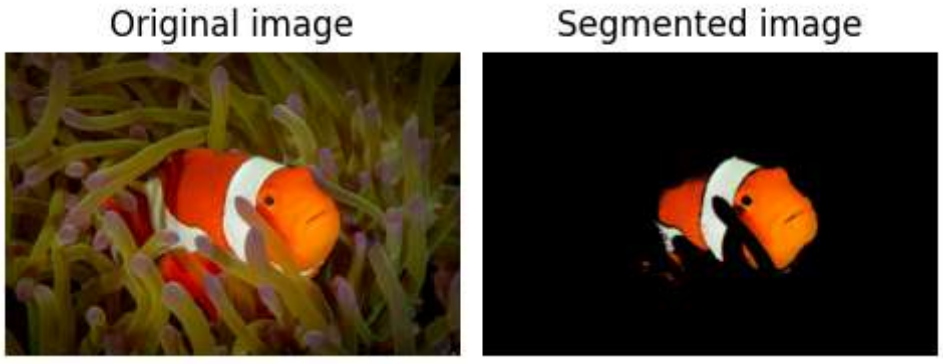
In [23]:
```
def segment_clownfish(image):
    hsv_image = cv2.cvtColor(image, cv2.COLOR_RGB2HSV)
    mask = cv2.inRange(hsv_image, light_orange, dark_orange)
    mask_white = cv2.inRange(hsv_image, light_white, dark_white)
    final_mask = mask + mask_white
    result = cv2.bitwise_and(image, image, mask=final_mask)
    blur = cv2.GaussianBlur(result, (7, 7), 0)
    return blur
```

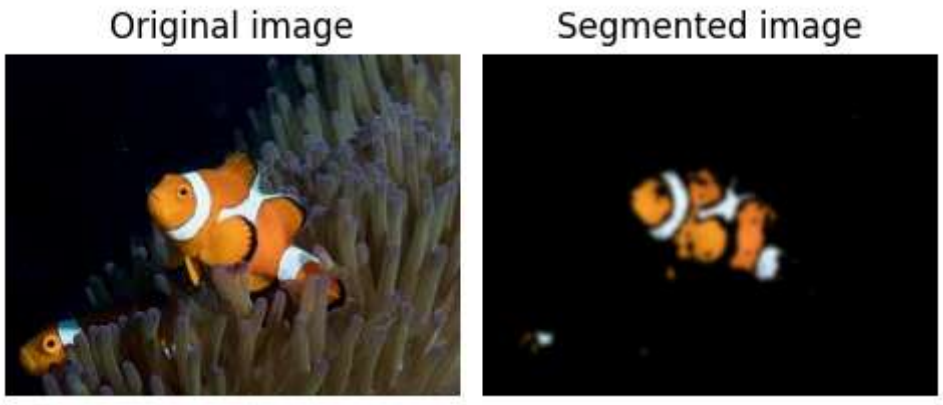
In [24]:
```
path = "./assets/nemo_images/nemo"
```

```
nemos_friends = []
for i in range(6):
   friend = cv2.cvtColor(cv2.imread(path + str(i) + ".jpg"), cv2.COLOR_BGR2RGB)
   nemos_friends.append(friend)

results = [segment_clownfish(friend) for friend in nemos_friends]
```

In [25]:
```
for i in range(1,6):
    plot_images(nemos_friends[i], 'Original image', results[i], 'Segmented image
```







En general los resultados fueron buenos, pero al generalizar el rango de colores sobre determinada imagen con cierta iluminación y fondo, algunas generalizaciones fallan.

Ej 1.b: Segmentación de un pájaro

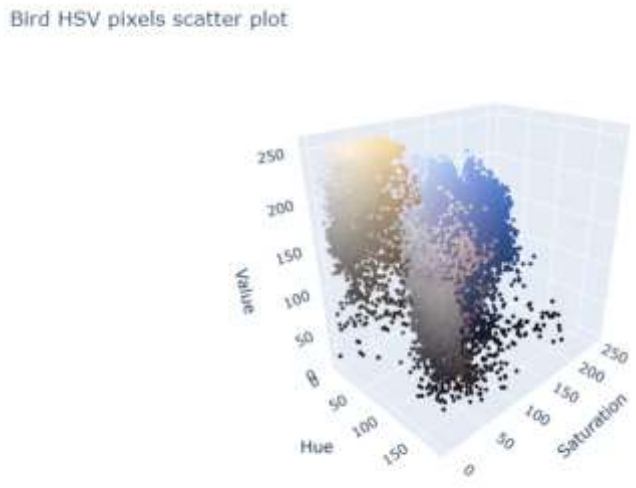
Aplicamos lo visto a otra imagen:

In [26]:
```
bird_bgr = cv2.imread('./assets/bird.jpg')
bird_bgr = cv2.resize(bird_bgr, (0,0), fx=0.25, fy=0.25)
bird_rgb = cv2.cvtColor(bird_bgr, cv2.COLOR_BGR2RGB)
plot_image(bird_rgb, 'Bird')
```

Visualizamos los pixeles en el espacio HSV:

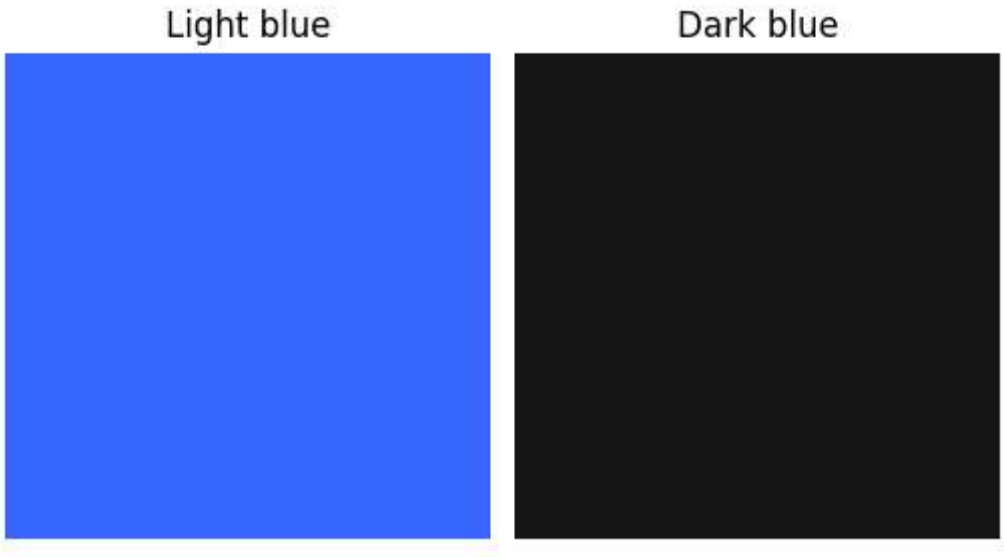
In [27]:
```
bird_hsv = cv2.cvtColor(bird_bgr, cv2.COLOR_BGR2HSV)
hsv_3d_scatter(bird_hsv, 'Bird HSV pixels scatter plot')
```

In [53]:
```
my_image = skimage.io.imread(fname=f"{OUTPUT_FOLDER_PATH}/scatter_hsv_bird.jpg")
plot_image(my_image, '')
```

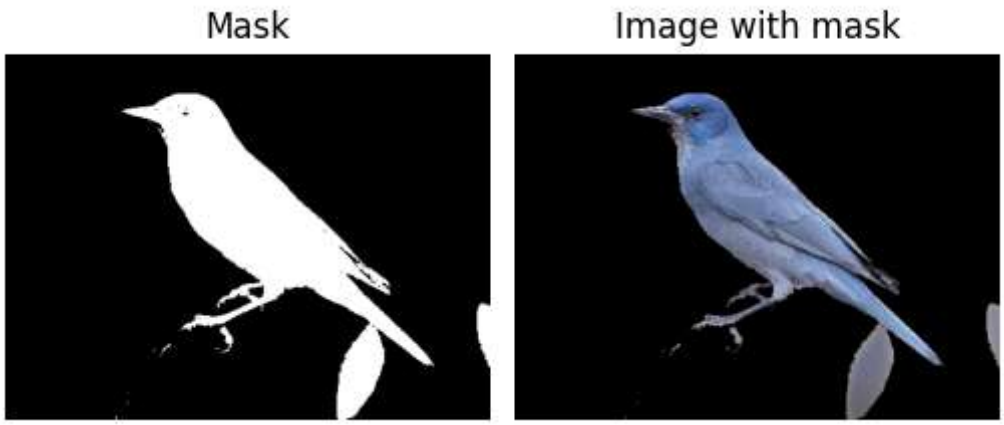


Se eligió esta librería de gráficos en particular para poder tomar más fácilmente el valor de los colores para el rango, la versión interactuable solo es visible desde Jupyter.

In [28]:
```
light_blue = (90, 0, 20)
dark_blue = (160, 200, 255)
plot_colors(light_blue, "Light blue", dark_blue, "Dark blue")
```

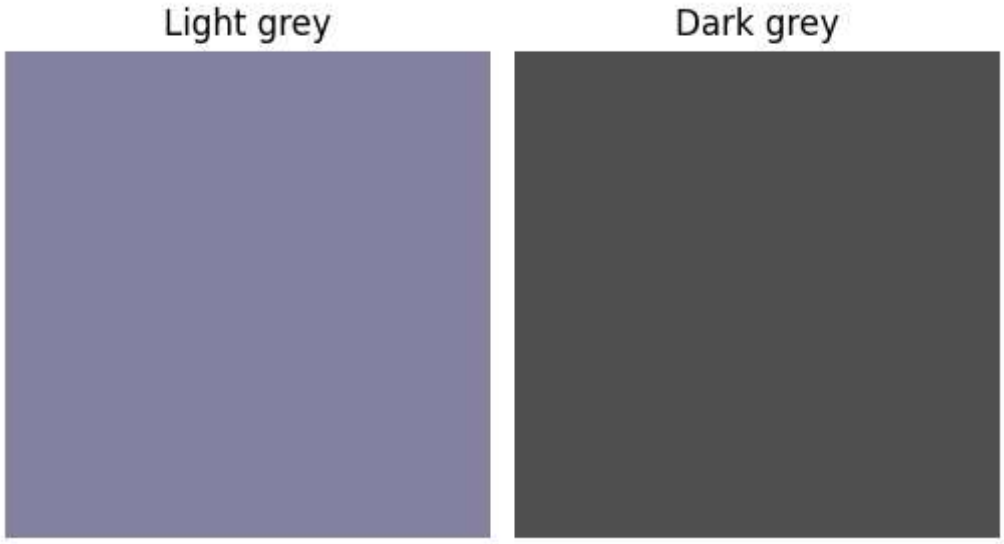


In [29]:
```
mask_blue = cv2.inRange(bird_hsv, light_blue, dark_blue)
result_bird = cv2.bitwise_and(bird_rgb, bird_rgb, mask=mask_blue)
plot_mask_result(mask_blue, result_bird)
```



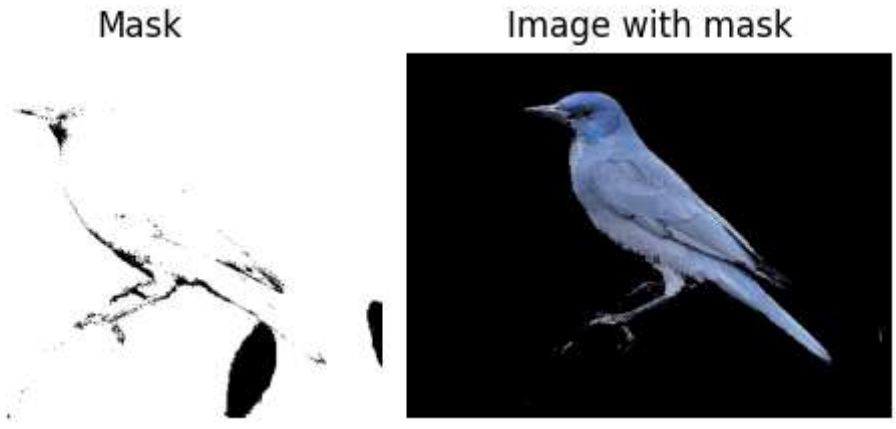
Nos quedan ciertos tonos grises del fondo, por lo que hacemos otra máscara:

In [30]:
```
light_grey = (75, 0, 80)
dark_grey = (175, 50, 160)
plot_colors(light_grey, "Light grey", dark_grey, "Dark grey")
```



Creamos una máscara inversa con dichos tonos de gris, y la aplicamos sobre el resultado anterior:

In [31]:
```
mask_grey = (255 - cv2.inRange(bird_hsv, light_grey, dark_grey))
result = cv2.bitwise_and(result_bird, result_bird, mask=mask_grey)
plot_mask_result(mask_grey, result)
```



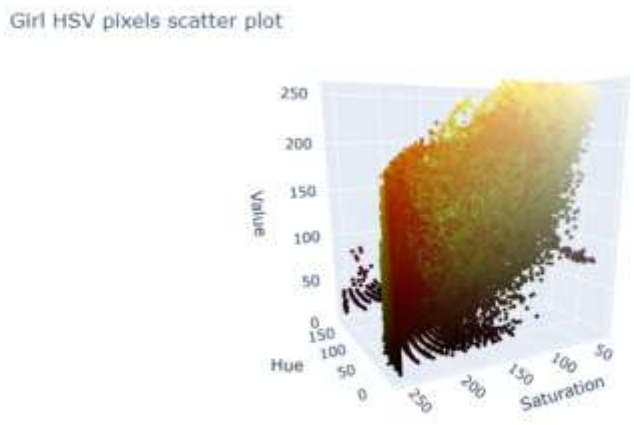
Ej 1.c: Segmentación de una rosa

In [32]:
```
girl_bgr = cv2.imread('./assets/Girl_and_rose.jpg')
girl_rgb = cv2.cvtColor(girl_bgr, cv2.COLOR_BGR2RGB)
plot_image(girl_rgb, 'Girl')
```

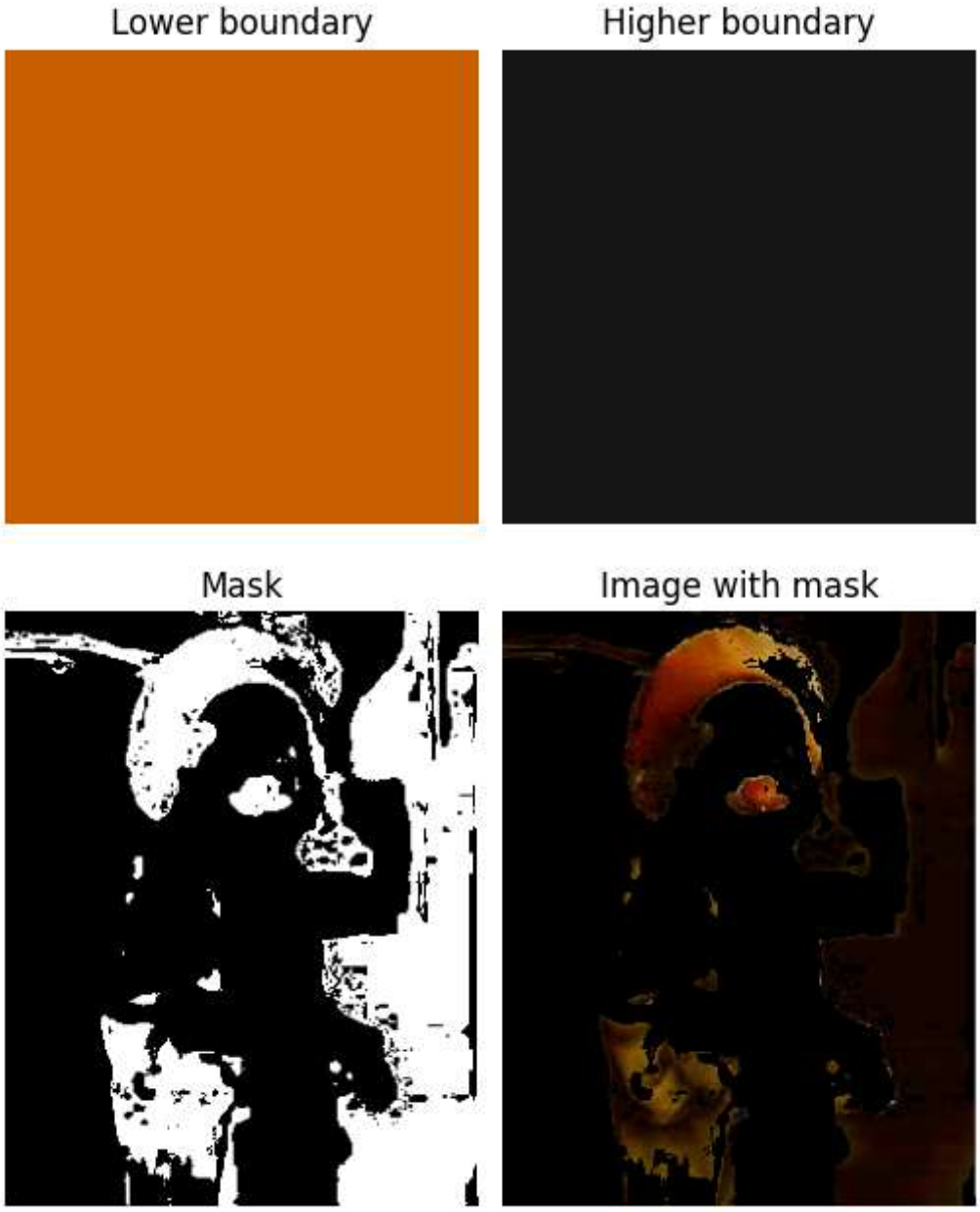
Analizamos los pixeles en el espacio HSV:

In [33]:
```
girl_hsv = cv2.cvtColor(girl_bgr, cv2.COLOR_BGR2HSV)
hsv_3d_scatter(girl_hsv, 'Girl HSV pixels scatter plot')
```

In [54]:
```
my_image = skimage.io.imread(fname=f"{OUTPUT_FOLDER_PATH}/scatter_hsv_girl.jpg")
plot_image(my_image, '')
```

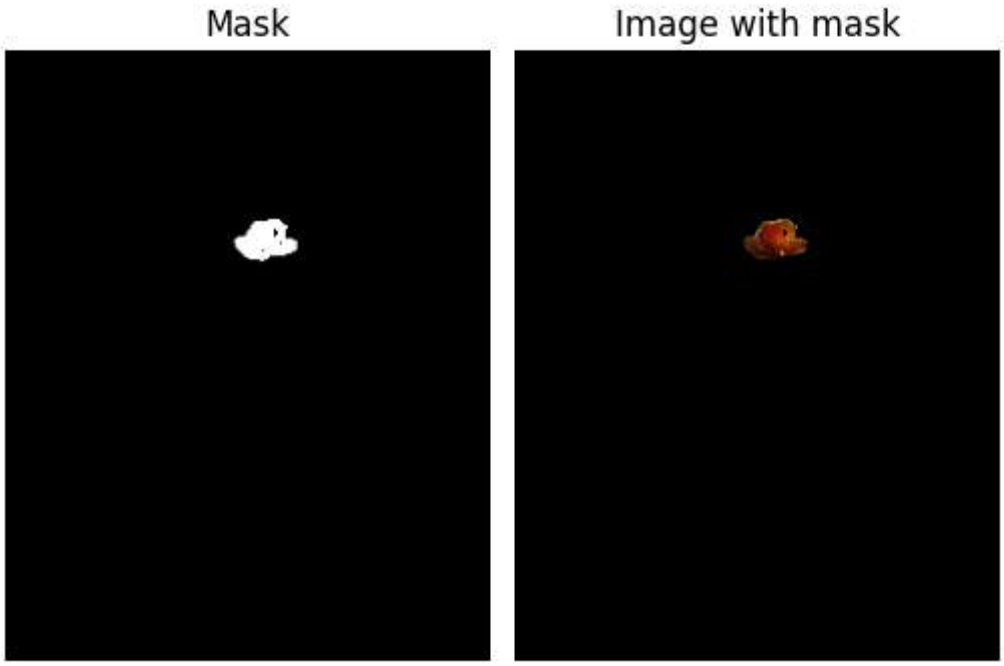


In [34]:
```
light_orange = (1, 0, 20)
dark_orange = (20, 255, 200)
plot_colors(light_orange, "Lower boundary", dark_orange, "Higher boundary")
mask_rose = cv2.inRange(girl_hsv, light_orange, dark_orange)
result_girl = cv2.bitwise_and(girl_rgb, girl_rgb, mask=mask_rose)
plot_mask_result(mask_rose, result_girl)
```



Como queremos solo la rosa:

In [35]:
```
mask_rose[:80, :] = 0
mask_rose[116:315, :] = 0
mask_rose[80:116, :115] = 0
mask_rose[80:116, 151:] = 0
result_rose = cv2.bitwise_and(girl_rgb, girl_rgb, mask=mask_rose)
plot_mask_result(mask_rose, result_rose)
```



Quitamos la rosa de la imagen original:

In [36]:
```
mask_without_rose = cv2.bitwise_not(mask_rose)
girl_without_rose = cv2.bitwise_and(girl_rgb, girl_rgb, mask=mask_without_rose)
```

Desaturamos la imagen para volverla blanco y negro:

In [37]:
```
girl_desaturated = cv2.cvtColor(girl_without_rose, cv2.COLOR_RGB2HSV)
girl_desaturated[:,:,1] = 0
girl_desaturated = cv2.cvtColor(girl_desaturated, cv2.COLOR_HSV2RGB)
plot_image(girl_desaturated, 'Girl desaturated')
```

Combinando ambas máscaras:

In [38]:
```
result_girl_rose = girl_desaturated + result_rose
plot_image(result_girl_rose, 'Girl bw + rose')
```

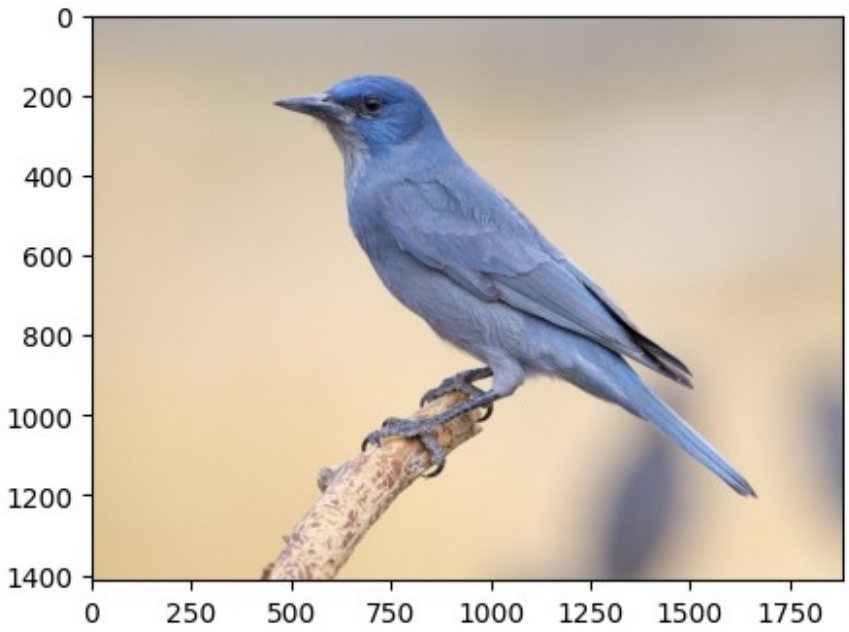
Segmentación de color de imagenes con algoritmo K-Means Clustering.

K-Means es un algoritmo de agrupamiento no supervisado, lo que significa que no requiere que los datos estén etiquetados previamente. Su objetivo es agrupar datos en base a su similitud, de forma que los elementos de un grupo (o cluster) sean más parecidos entre sí que a los de otros grupos

En este caso, K representa el número de clusters deseados. El proceso del algoritmo es el siguiente:

1. Se selecciona el número de clusters (K).
2. Los píxeles de la imagen se asignan aleatoriamente a uno de los K clusters.
3. Se calcula el centro de cada cluster.
4. Se mide la distancia de cada píxel al centro de cada cluster.
5. Se reasigna cada píxel al cluster cuyo centro esté más cercano.
6. Se actualizan los centros de los clusters.
7. Los pasos 4, 5 y 6 se repiten hasta que las asignaciones de los píxeles no cambien, o se alcance el número máximo de iteraciones

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from scipy.cluster.vq import kmeans, vq

bird = Image.open("./assets/bird.jpg")
bird = bird.convert("RGB")

fig = plt.figure()
fig.set_size_inches(5, 5)
plt.imshow(bird)
plt.show()
```




```
# Es necesario un arreglo 2D para k-means
bird_array = np.array(bird)
pixel_vals = bird_array.reshape((-1, 3)).astype(np.float32)
pixel_vals.shape
```

```
(2665390, 3)
```

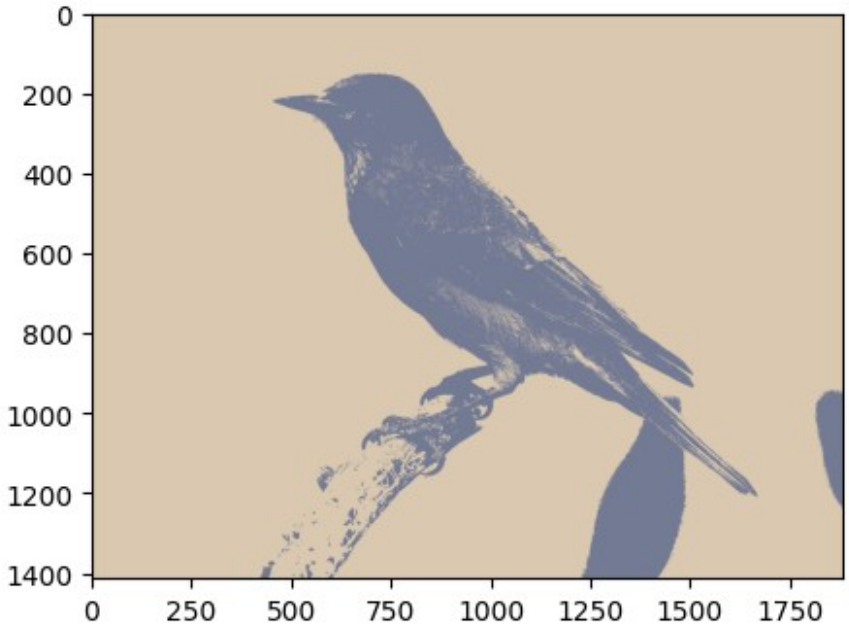
```
# Segmentamos la imagen en dos clusters (K=2).
k = 2
# Cuando se llegue a las 100 iteraciones, o que la precisión lograda
(el e) sea del 95% se da la finalizacion
centers, _ = kmeans(pixel_vals, k, iter=100, thresh=0.95)

# Asignamos cada pixel al cluster mas cercano
labels, _ = vq(pixel_vals, centers)

centers = centers.astype(np.float32)
segmented_data = centers[labels.flatten()]
segmented_image =
segmented_data.reshape((bird_array.shape)).astype(int)
print(centers)
print(segmented_data)
print(segmented_image)
```

```
[[115.06263 122.42949 148.11615]
 [218.4734  201.59184 176.82185]]
[[218.4734  201.59184 176.82185]
 [218.4734  201.59184 176.82185]
 [218.4734  201.59184 176.82185]
 ...
```

```
 [218.4734  201.59184 176.82185]
 [218.4734  201.59184 176.82185]
 [218.4734  201.59184 176.82185]]
[[[218 201 176]
  [218 201 176]
  [218 201 176]
  ...
  [218 201 176]
  [218 201 176]
  [218 201 176]]

 [[218 201 176]
  [218 201 176]
  [218 201 176]
  ...
  [218 201 176]
  [218 201 176]
  [218 201 176]]

 [[218 201 176]
  [218 201 176]
  [218 201 176]
  ...
  [218 201 176]
  [218 201 176]
  [218 201 176]]

 ...

 [[218 201 176]
  [218 201 176]
  [218 201 176]
  ...
  [218 201 176]
  [218 201 176]
  [218 201 176]]

 [[218 201 176]
  [218 201 176]
  [218 201 176]
  ...
  [218 201 176]
  [218 201 176]
  [218 201 176]]

 [[218 201 176]
  [218 201 176]
  [218 201 176]
  ...
  [218 201 176]
```

```
  [218 201 176]
  [218 201 176]]]
```

```
fig = plt.figure()
fig.set_size_inches(5, 5)
plt.imshow(segmented_image)
plt.show()
```



En la máscara obtenida se logra una segmentación adecuada del ave, aunque presenta algunos errores. El principal problema es que una parte del fondo, cuyo color original era un gris similar al de ciertas zonas del ave, fue agrupada en el mismo cluster. Es importante destacar que este inconveniente también se observó al segmentar utilizando espacios de color, como HSV. En ese caso, el problema se solucionó aplicando una segunda máscara.

Filtro de Bayer

El filtro de Bayer es un tipo de matriz de filtros de color (CFA, Color Filter Array) utilizado en cámaras digitales para capturar imágenes en color. Fue inventado por Bryce Bayer en 1974 y se emplea para permitir que los fotosensores de las cámaras, que solo pueden medir la intensidad de la luz, capten información sobre el color. El filtro de Bayer es fundamental para las cámaras digitales modernas, ya que permite que los sensores capturen imágenes en color de manera eficiente.

Las cámaras digitales tienen una matriz de fotosensores que capturan la luz, pero no pueden distinguir la longitud de onda, es decir, el color de la luz. Entonces, el filtro de Bayer se coloca sobre los fotosensores para que cada uno solo reciba un determinado color (rojo, verde o azul). El patrón RGGB define la disposición de los filtros: en un arreglo de 2x2 píxeles, hay 2 píxeles verdes, 1 rojo y 1 azul (se utiliza el doble de píxeles verdes porque el ojo humano es más sensible a la luz verde, lo que mejora la percepción visual).

La imagen capturada inicialmente es incompleta, ya que cada píxel solo registra un color. Para obtener una imagen completa, es necesario reconstruir las componentes rojo, verde y azul en cada píxel, un proceso conocido como demosaicing

Demosaicing

Demosaicing es el proceso que permite reconstruir una imagen completa a partir del patrón de Bayer, utilizando técnicas de interpolación para estimar los colores que no fueron medidos en cada píxel. Este proceso utiliza la información de los píxeles vecinos para estimar los valores faltantes de color.

El objetivo es recuperar la imagen completa a partir del patrón de Bayer, donde cada píxel solo tiene información de un color (rojo, verde o azul). Hace uso de un metodo que se basa en técnicas de interpolación para estimar los valores de color no medidos. Cómo se combine esta información en el resultado dependerá de los algoritmos. La elección del algoritmo está sujeta a la capacidad de procesamiento, el costo computacional y energético, y la calidad de la imagen deseada.

Tipos de Algoritmos de Demosaicing

Interpolación por vecino más cercano (Nearest-neighbor interpolation):

Reemplaza las componentes de color faltantes de un píxel con el valor del píxel más cercano que tenga esa información.

Interpolación bilineal (Bilinear interpolation):

Sustituye la componente de color faltante por la media aritmética de los valores de los píxeles vecinos que tienen información de ese color.

Algoritmos adaptativos:

Utilizan la correlación espacial o espectral de los píxeles para estimar de manera más precisa las componentes de color faltantes.

El demosaicing es fundamental para obtener una imagen en color de alta calidad a partir de los datos capturados por el filtro de Bayer.

```
from scipy.signal import convolve2d
import matplotlib.pyplot as plt
from PIL import Image
import numpy as np
import matplotlib
import math

def wbscalematrix(m, n, wb_scales, align):
    # Makes a white-balance scaling matrix for an image of size m-by-n
    # from the individual RGB white balance scalar multipliers
[wb_scales] = [R_scale G_scale B_scale].
    #
    # [align] is string indicating the 2x2 Bayer arrangement:
    # 'rggb':
    #    R G
    #    G B
    # 'gbrg':
    #    G B
    #    R G
    # 'grbg','bggr' follow as before.

    # First, for convenience only, we're going to fill the scale
matrix
    # with all green pixels. Then, we're going to replace the red and
blue
    # pixels later on with the correct scalars.
    #
    scalematrix = wb_scales[1] * np.ones((m,n)) # Initialize to all
green values

    # Fill in the scales for the red and blue pixels across the matrix
    if (align == 'rggb'):
        scalematrix[0::2, 0::2] = wb_scales[0] # r
        scalematrix[1::2, 1::2] = wb_scales[2] # b
    elif (align == 'bggr'):
        scalematrix[1::2, 1::2] = wb_scales[0] # r
        scalematrix[0::2, 0::2] = wb_scales[2] # b
```

```
    elif (align == 'grbg'):
        scalematrix[0::2, 1::2] = wb_scales[0] # r
        scalematrix[0::2, 1::2] = wb_scales[2] # b
    elif (align == 'gbrg'):
        scalematrix[1::2, 0::2] = wb_scales[0] # r
        scalematrix[0::2, 1::2] = wb_scales[2] # b
    return scalematrix

def apply_cmatrix(img, cmatrix):
    # Applies color transformation CMATRIX to RGB input IM.
    # Finds the appropriate weighting of the old color planes to form
the new color planes,
    # equivalent to but much more efficient than applying a matrix
transformation to each pixel.
    if (img.shape[2] != 3):
        raise ValueError('Apply cmatrix to RGB image only.')

    r = cmatrix[0,0] * img[:,:,0] + cmatrix[0,1] * img[:,:,1] +
cmatrix[0,2] * img[:,:,2]
    g = cmatrix[1,0] * img[:,:,0] + cmatrix[1,1] * img[:,:,1] +
cmatrix[1,2] * img[:,:,2]
    b = cmatrix[2,0] * img[:,:,0] + cmatrix[2,1] * img[:,:,1] +
cmatrix[2,2] * img[:,:,2]
    corrected = np.stack((r,g,b), axis=2)
    return corrected

def debayering(input):
    # Bilinear Interpolation of the missing pixels
    #
    # Assumes a Bayer CFA in the 'rggb' layout
    #   R G R G
    #   G B G B
    #   R G R G
    #   G B G B
    #
    # Input: Single-channel rggb Bayered image
    # Returns: A debayered 3-channels RGB image
    #
    img = input.astype(np.double)

    m = img.shape[0]
    n = img.shape[1]

    # First, we're going to create indicator masks that tell us
    # where each of the color pixels are in the bayered input image
    # 1 indicates presence of that color, 0 otherwise
    red_mask = np.tile([[1,0],[0,0]], (int(m/2), int(n/2)))
    green_mask = np.tile([[0,1],[1,0]], (int(m/2), int(n/2)))
    blue_mask = np.tile([[0,0],[0,1]], (int(m/2), int(n/2)))
```

```
    r = np.multiply(img, red_mask)
    g = np.multiply(img, green_mask)
    b = np.multiply(img, blue_mask)

    # Confirm for yourself:
    # - What are the patterns of values in the r,g,b images?
    # Sketch them out to help yourself.

    # Next, we're going to fill in the missing values in r,g,b
    # For this, we're going to use filtering - convolution - to
implement bilinear interpolation.
    # - We know that convolution allows us to perform a weighted sum
    # - We know _where_ our pixels lie within a grid, and where the
missing pixels are
    # - We know filters come in odd sizes

    # Interpolating green:
    filter_g = 0.25 * np.array([[0,1,0],[1,0,1],[0,1,0]])
    missing_g = convolve2d(g, filter_g, 'same')
    g = g + missing_g

    # To conceptualize how this works, let's continue to draw it out
on paper.
    # - Sketch the first 5 rows and columns of the g image
    # - Sketch the 3x3 filter and add the numeric weights (they sum to
1)
    # - Sketch the output image

    # Move the filter through the valid region of the image.
    # - What is the output at pixel 1,1 ?  [0-index, remember]
    # - What is the output at pixel 2,1 ?
    # - What is the output at pixel 3,1 ?
    # - What is the output at pixel 1,2 ?
    # - What is the output at pixel 2,2 ?
    # - What is the output at pixel 3,2 ?

    # See how it works?
    # The filter only produces output if the surrounding pixels match
its pattern.
    # When they do, it produces their mean value.

    # Note that we're going to have some incorrect values at the image
boundaries,
    # but let's ignore that for this exercise.

    # Now, let's try it for blue. This one is a two-step process.
    # - Step 1: We fill in the 'central' blue pixel in the location of
the red pixel
    # - Step 2: We fill in the blue pixels at the locations of the
green pixels,
```

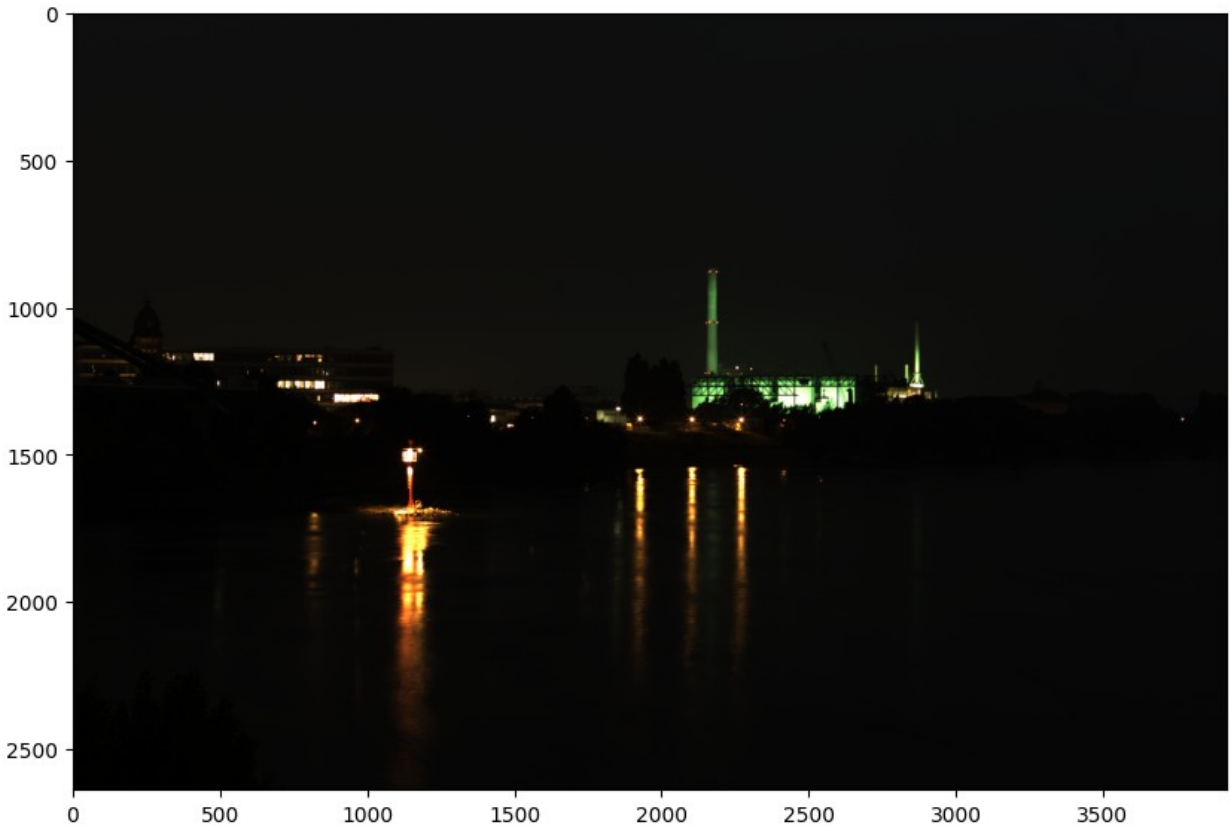
```
    #           similar to how the green interpolation worked, but
offset by a row/column
    #
    # Sketch out the matrices to help you follow.
    # Remember, we'll still have some incorrect value at the image
boundaries.

    # Interpolating blue:
    # Step 1:
    filter1 = 0.25 * np.array([[1,0,1],[0,0,0],[1,0,1]])
    missing_b1 = convolve2d(b, filter1, 'same')
    # Step 2:
    filter2 = 0.25 * np.array([[0,1,0],[1,0,1],[0,1,0]])
    missing_b2 = convolve2d(b + missing_b1, filter2, 'same')
    b = b + missing_b1 + missing_b2

    # OK! Only red left.

    # Interpolation for the red at the missing points
    # TODO: Complete the following two lines. Follow a similar
strategy to the blue channel.
    missing_r1 = convolve2d(r, filter1, 'same')
    missing_r2 = convolve2d(r + missing_r1, filter2, 'same')
    r = r + missing_r1 + missing_r2

    output = np.stack((r,g,b), axis=2)
    return output

# Primero, utilizamos el programa dcraw para ejecutar una rutina de
reconocimiento en la imagen de prueba sample.DNG.
# Como resultado, obtenemos los siguientes valores

# Step 0: Convert RAW file to TIFF
black = 0
saturation = 16383
wb_multipliers = [2.217041, 1.000000, 1.192484]

# Open the CFA image
raw_data = Image.open('./assets/sample.tiff')
raw = np.array(raw_data).astype(np.double)

# En otro paso, se linealiza y normaliza el contenido de los píxeles
para utilizar el rango dinámico completo.

# Step 1: Normalization
linear_bayer = (raw - black) / (saturation - black)

# Plot the CFA image
plt.figure(figsize=(10, 10))
```

```
plt.imshow(linear_bayer, cmap='gray')
plt.show()
```


```
# Se escalan las componentes de color rojo, verde y azul con el
objetivo de llevar los colores presentes en la imágen
# a una escala apropiada tomando como referencia el color blanco.
# Esto es lo que se conoce como balance de blancos.

# Step 2: White balancing
mask = wbscalematrix(linear_bayer.shape[0], linear_bayer.shape[1],
wb_multipliers, 'rggb')
balanced_bayer = np.multiply(linear_bayer, mask)

# Plot balanced result
plt.figure(figsize=(10, 10))
plt.imshow(balanced_bayer, cmap='gray')
plt.show()
```

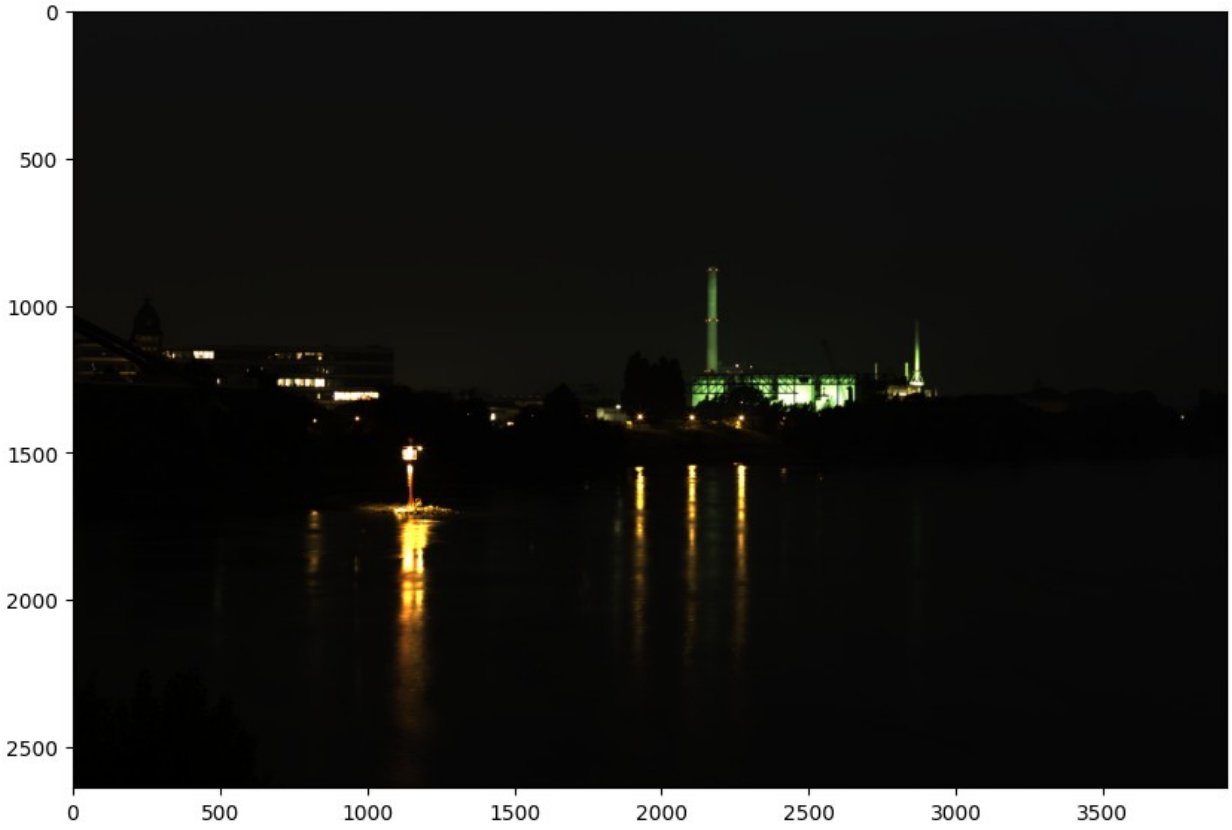

```
# Se aplica el debayering o demosaicing, esto es, una interpolación
para reconstruir las componentes de color rojo, verde y azul
# que no se encuentran presentes por el filtrado Bayer que aplican las
cámaras fotográficas.
# En este caso, se realizan promediados para reconstruir pixel a pixel
estas componentes.

# Step 3: Debayering (also called demosaicing)
lin_rgb = debayering(balanced_bayer)

# Plot the result
plt.figure(figsize=(10, 10))
plt.imshow(lin_rgb)
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data
([0..1] for floats or [0..255] for integers).
```

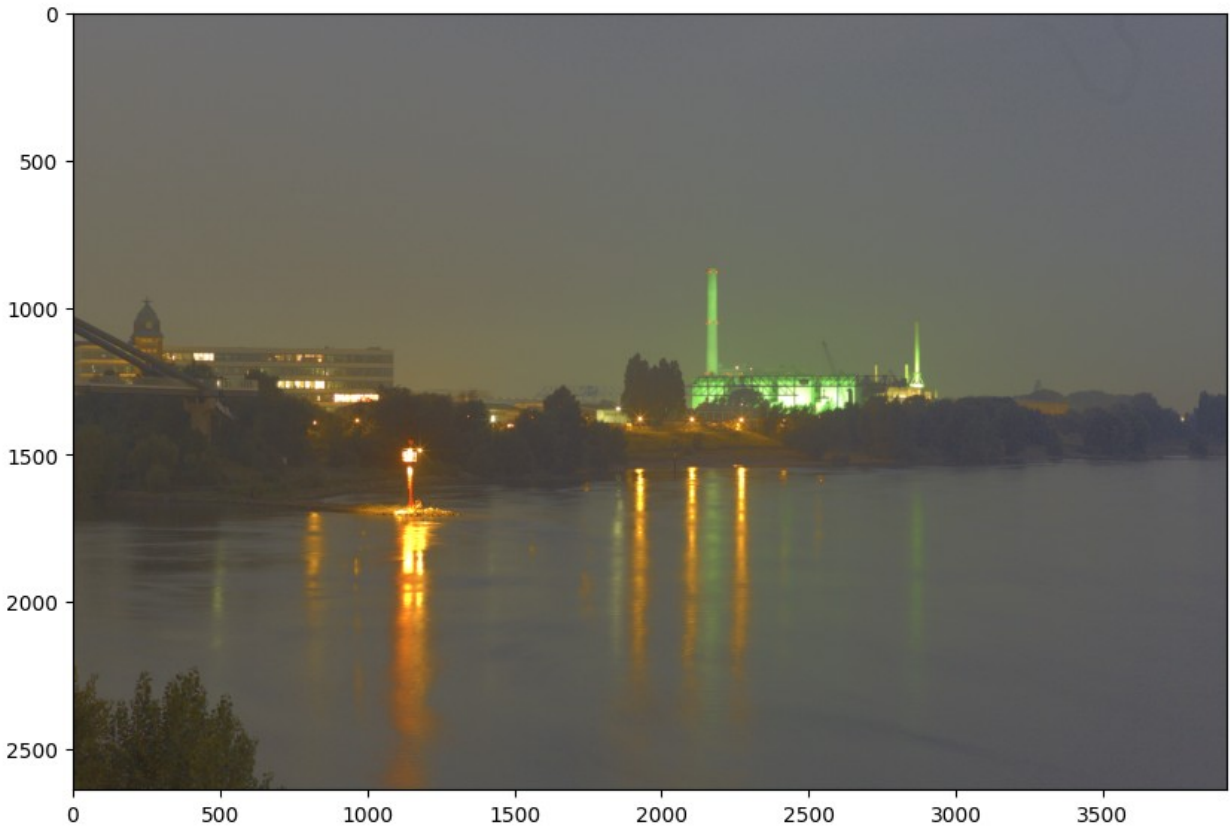



```
# Se aplica una transformación para llevar los colores a un espacio de color sRGB que puede ser interpretado
# y representado correctamente por Windows y las computadores modernas

# Step 4: Color space conversion --- we do this one for you.
# Convert to sRGB. xyz2cam is found in dcraw's source file adobe_coeff.
rgb2xyz = np.array([[0.4124564, 0.3575761, 0.1804375],
            [0.2126729, 0.7151522, 0.0721750],
            [0.0193339, 0.1191920, 0.9503041]])
xyz2cam = np.array([[0.6653, -0.1486, -0.0611],
            [-0.4221, 1.3303, 0.0929],
            [-0.0881, 0.2416, 0.7226]])
rgb2cam = xyz2cam * rgb2xyz # Assuming previously defined matrices
denom = np.tile(np.reshape(np.sum(rgb2cam,axis=1),(3,-1)), (1,3))
rgb2cam = np.divide(rgb2cam, denom) # Normalize rows to 1
cam2rgb = np.linalg.inv(rgb2cam)
lin_srgb = apply_cmatrix(lin_rgb, cam2rgb)
lin_srgb[lin_srgb > 1.0] = 1.0 # Always keep image clipped b/w 0-1
lin_srgb[lin_srgb < 0.0] = 0.0

# Plot the results
plt.figure(figsize=(10, 10))
```

```
plt.imshow(lin_srgb)
plt.show()
```




```
# Finalmente, se debe aplicar una corrección de brillo y de gamma.

# Step 5: Brightness and gamma correction
gamma_hsv = matplotlib.colors.rgb_to_hsv(lin_srgb)
adjust_brightness = 0.00
magic_touch = 0.05
gamma_hsv[:,:,2] += adjust_brightness
gamma = np.log10(0.5) / np.log10(gamma_hsv[:,:,2].mean()) +
magic_touch
gamma_hsv[:,:,2] = np.power(gamma_hsv[:,:,2], gamma)
gamma_srgb = matplotlib.colors.hsv_to_rgb(gamma_hsv)
gamma_srgb[gamma_srgb > 1.0] = 1.0
gamma_srgb[gamma_srgb < 0.0] = 0.0


plt.figure(figsize=(10, 10))
plt.imshow(gamma_srgb)
plt.show()
```


0
500
1000
1500
2000
2500
0
500
1000
1500
2000
2500
3000
3500