Percepción Computacional

Caso grupal: Segmentación de Imágenes

Autores:

- Silvana Cisneros
- Victor Jaramillo
- Fabian Quijosaca

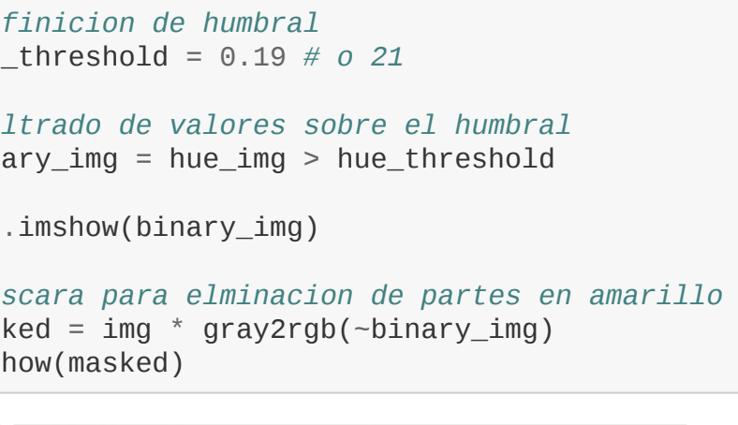
In [1]:
```
# Cargar las librerías necesarias
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from skimage.color import rgb2gray, label2rgb, rgb2hsv, gray2rgb, hsv2rgb
from skimage import data
from skimage.filters import gaussian, sobel, threshold_otsu
from skimage.segmentation import (active_contour, felzenszwalb, quickshift,  watershed,
                                  mark_boundaries, slic, clear_border, relabel_sequential)

from skimage.future.graph import rag_mean_color, cut_normalized, cut_threshold
import io
from itertools import product
from skimage.util import img_as_float
from skimage import morphology
```

In [2]:
```
# Defino una función para mostrar una imagen por pantalla con el criterio que considero más
 acertado
def imshow(img):
    fig, ax = plt.subplots(figsize=(7, 7))
    # El comando que realmente muestra la imagen
    ax.imshow(img,cmap=plt.cm.gray)
    # Para evitar que aparezcan los números en los ejes
    ax.set_xticks([]), ax.set_yticks([])
    plt.show()
```

In [3]:
```
# Defino una función para mostrar una cuadricula de 4 imagene por pantalla con sus titulos
def imshow_grid(arr_img, arr_title):
    fig, ax = plt.subplots(2, 2, figsize=(12, 12), sharex=True, sharey=True)

    ax[0, 0].imshow(arr_img[0])
    ax[0, 0].set_title(arr_title[0])

    ax[0, 1].imshow(arr_img[1])
    ax[0, 1].set_title(arr_title[1])

    ax[1, 0].imshow(arr_img[2])
    ax[1, 0].set_title(arr_title[2])

    ax[1, 1].imshow(arr_img[3])
    ax[1, 1].set_title(arr_title[3])

    for a in ax.ravel():
        a.set_axis_off()

    plt.tight_layout()
    plt.show()
```

In [4]:
```
# Función para comparar cómo de buena es una determinada segmentación en comparación con el
 ground truth
# Ambas imagenes deben contener valores True/False
def factor_f_evaluation(binary_image, ground_truth):
    TP = np.sum(np.logical_and(binary_image, ground_truth))
    TN = np.sum(np.logical_and(np.logical_not(binary_image), np.logical_not(ground_truth)))
    FP = np.sum(np.logical_and(np.logical_not(binary_image), ground_truth))
    FN = np.sum(np.logical_and(binary_image, np.logical_not(ground_truth)))
    P = TP/np.float(TP+FP)
    R = TP/np.float(TP+FN)
    if P+R == 0:
        F = 0
    else:
        F = 2*P*R/(P+R)
    return F
```

In [5]:
```
# https://lmb.informatik.uni-freiburg.de/resources/datasets/moseg.en.html
img = mpimg.imread('ducks01_0100.jpg')
gt_img = rgb2gray(mpimg.imread('ducks01_0100_gt.ppm'))<1
imshow(img)
```

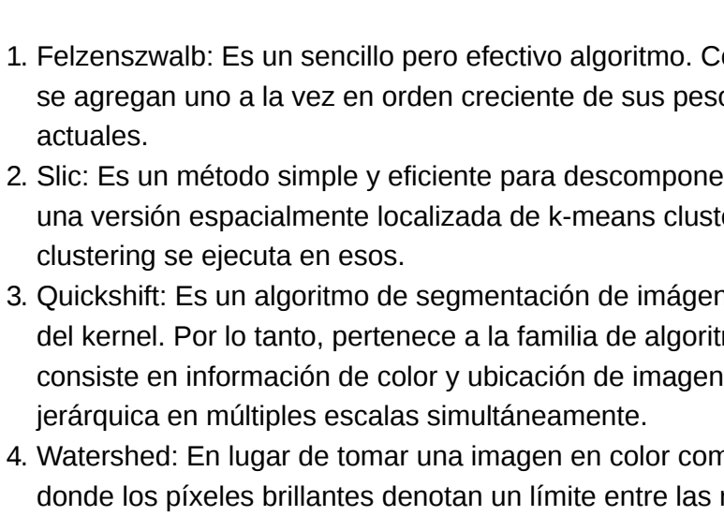


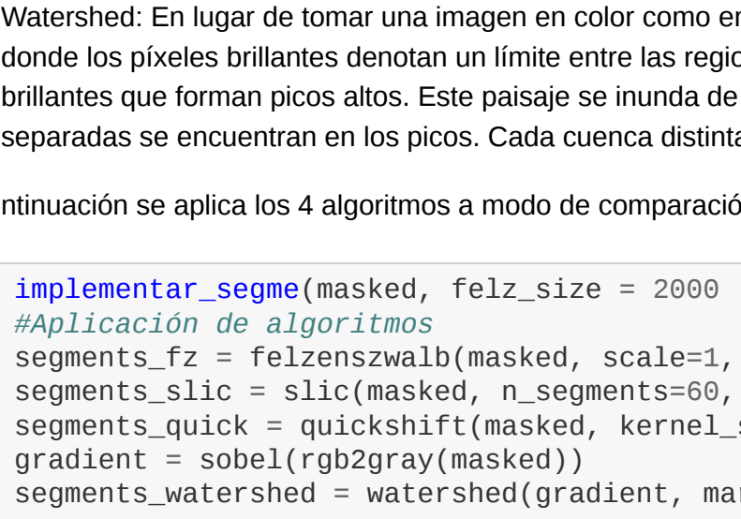
Preprocesamiento de la imagen

Lo primero a realizarse es un pre procesamiento de la imagen para eliminar los elementos incensarios, esto permitirá obtener mejores resultados y facilita la ejecución de los algoritmos de segmentación

In [6]:
```
#Método que permite convertir una imagen de formato RGB a HVS (Matiz, Saturación, Valor)
#Y retornar unicamente los caloes de color (matiz)
#Se usa este formato para poder distinguir de mejor forma y realizar un
#procesamiento mas eficiente en la imagen
def convertir_hsv(img):
    hsv_img = rgb2hsv(img)
    return hsv_img[:, :, 0]
```

In [7]:
```
#Eliminacin de forn dever
# Input data
#orig_img = img
#Obtener matiz de la imagen
hue_img = convertir_hsv(img)

#Definicion de humbral
hue_threshold = 0.19 # o 21

#Filtrado de valores sobre el humbral
binary_img = hue_img > hue_threshold

plt.imshow(binary_img)

#Mascara para elminacion de partes en amarillo
masked = img * gray2rgb(~binary_img)
imshow(masked)
```





Empleo de algoritmos de segmentación basada en color

A continuación se muestra la comparación de 4 algoritmos de segmentación. Los algoritmos utilizados son NO SUPERVISADOS.

1. Felzenszwalb: Es un sencillo pero efectivo algoritmo. Comienza con el gráfico completamente desconectado, los bordes se agregan uno a la vez en orden creciente de sus pesos y mantiene un bosque de MST para sus componentes actuales.
2. Slic: Es un método simple y eficiente para descomponer una imagen en regiones visualmente homogéneas. Se basa en una versión espacialmente localizada de k-means clustering. Cada píxel está asociado a un vector y luego k-means clustering se ejecuta en esos.
3. Quickshift: Es un algoritmo de segmentación de imágenes 2D. Se basa en una aproximación del desplazamiento medio del kernel. Por lo tanto, pertenece a la familia de algoritmos de búsqueda de modo local y se aplica al espacio 5D que consiste en información de color y ubicación de imagen. Uno de los beneficios es que calcula una segmentación jerárquica en múltiples escalas simultáneamente.
4. Watershed: En lugar de tomar una imagen en color como entrada, requiere una imagen de gradiente en escala de grises, donde los píxeles brillantes denotan un límite entre las regiones. El algoritmo ve la imagen como un paisaje, con píxeles brillantes que forman picos altos. Este paisaje se inunda de los marcadores dados, hasta que cuencas de inundación separadas se encuentran en los picos. Cada cuenca distinta forma un segmento de imagen diferente.

A continuación se aplica los 4 algoritmos a modo de comparación y se presenta los límites entre las regiones segmentadas:

In [8]:
```
def implementar_segme(masked, felz_size = 2000 ):
    #Aplicación de algoritmos
    segments_fz = felzenszwalb(masked, scale=1, sigma=0.2, min_size=felz_size)
    segments_slic = slic(masked, n_segments=60, compactness=30, sigma=1.2,max_iter=40)
    segments_quick = quickshift(masked, kernel_size=5, max_dist=1000, ratio=1)
    gradient = sobel(rgb2gray(masked))
    segments_watershed = watershed(gradient, markers=400, compactness=0.01)

    #Conteo del numero de segmentos obtenidos en cada algoritmo
    print(f"Felzenszwalb cantidad de segmentos: {len(np.unique(segments_fz))}")
    print(f"SLIC cantidad de segmentos: {len(np.unique(segments_slic))}")
    print(f"Quickshift cantidad de segmentos: {len(np.unique(segments_quick))}")
    print(f"Compact watershed cantidad de segmentos: {len(np.unique(segments_watershed))}")

    arr_segm = []
    arr_segm.append(segments_fz)
    arr_segm.append(segments_slic)
    arr_segm.append(segments_quick)
    arr_segm.append(segments_watershed)

    return arr_segm
```

In [9]:
```
#Implementacion de segmentacion
arr_segm = implementar_segme(masked)
#Arreglos a ser graficados
arr_img = []
arr_title = []
#Creacion de imagenes y segmentos generados
arr_img.append(mark_boundaries(masked, arr_segm[0]))
arr_img.append(mark_boundaries(masked, arr_segm[1]))
arr_img.append(mark_boundaries(masked, arr_segm[2]))
arr_img.append(mark_boundaries(masked, arr_segm[3]))
#Titulos de las imagenes
arr_title.append('Método Felzenszwalbs')
arr_title.append('Método SLIC')
arr_title.append('Método Quickshift')
arr_title.append('Método Compact watershed')

imshow_grid(arr_img, arr_title)
```

```
Felzenszwalb cantidad de segmentos: 18
SLIC cantidad de segmentos: 57
Quickshift cantidad de segmentos: 307
Compact watershed cantidad de segmentos: 405
```

In [10]:
```
#Dibujado de las imagenes con las secciones segmentadas u convertidas a RGB
#Arreglos a ser graficados
arr_img_grey = []

arr_img_grey.append(label2rgb(arr_segm[0], masked, kind = 'avg'))
arr_img_grey.append(label2rgb(arr_segm[1], masked, kind = 'avg'))
arr_img_grey.append(label2rgb(arr_segm[2], masked, kind = 'avg'))
arr_img_grey.append(label2rgb(arr_segm[3], masked, kind = 'avg'))

imshow_grid(arr_img_grey, arr_title)
```

La segmentación más apropiada que genero la comparativa superior es:

FELZENSZWALBS - Segmentación de imagen basada árbol de expansión

Este algoritmo segmenta utilizando un agrupamiento rápido, mínimo, basado en un árbol de expansión en la cuadrícula de la imagen.

In [11]:
```
# Se utiliza una segmentación felzenszwalb
labels = felzenszwalb(masked, scale=1, sigma=0.2, min_size=2000)
#Para cada segmento se asocia su valor promedio
out  = label2rgb(labels, masked, kind = 'avg')
imshow(out)
```

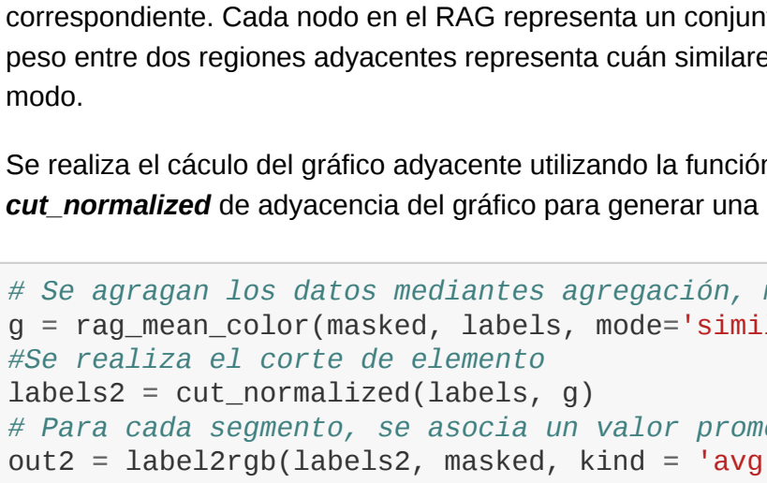
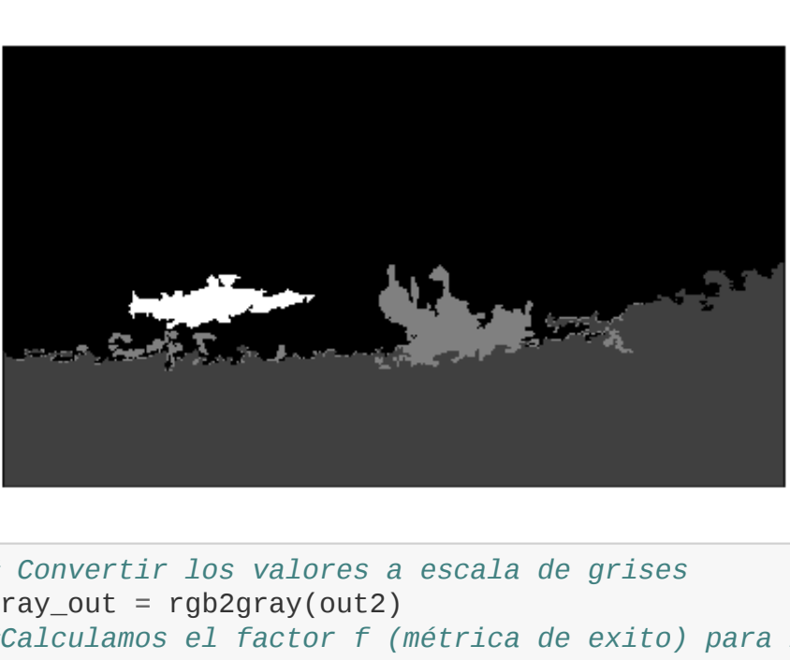


Gráfico de adyacencia de región - (RAG: Region Adjacency Graph)

Dada una imagen y su segmentación inicial, este método construye el Gráfico de adyacencia de región (RAG) correspondiente. Cada nodo en el RAG representa un conjunto de píxeles dentro de la imagen con la misma etiqueta. El peso entre dos regiones adyacentes representa cuán similares o diferentes son dos regiones dependiendo del parámetro de modo.

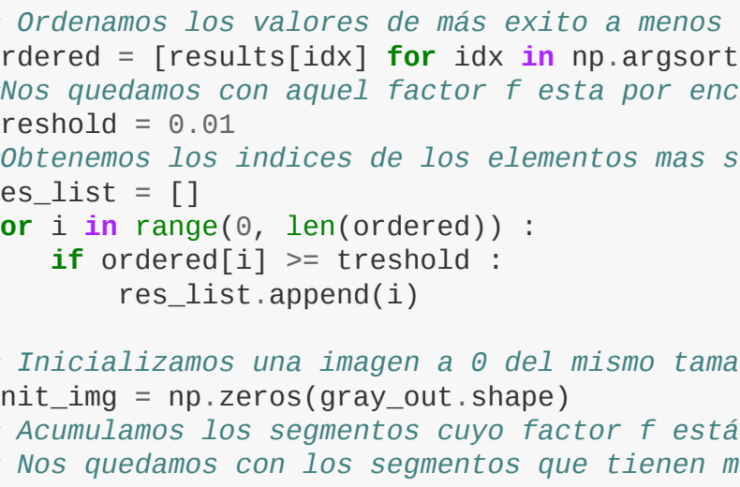
Se realiza el cáculo del gráfico adyacente utilizando la función ***rag_min_color***, luego se realiza el corte normalizado ***cut_normalized*** de adyacencia del gráfico para generar una primera versión segementada y procesada de la imagen.

In [12]:
```
# Se agragan los datos mediantes agregación, multiescala basada en grafos (RAG)
g = rag_mean_color(masked, labels, mode='similarity')
#Se realiza el corte de elemento
labels2 = cut_normalized(labels, g)
# Para cada segmento, se asocia un valor promedio
out2 = label2rgb(labels2, masked, kind = 'avg')
#Se muestra la imagen procesada
imshow(out2)
#Se muestra el resultado de los cortes normalizados
imshow(labels2)
```



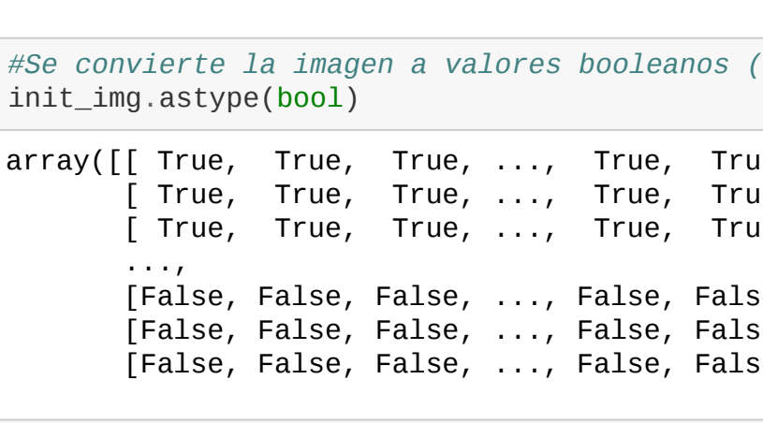
In [13]:
```
# Convertir los valores a escala de grises
gray_out = rgb2gray(out2)
#Calculamos el factor f (métrica de exito) para los diferentes segmentos
results = [factor_f_evaluation(gray_out == a, gt_img) for a in np.unique(gray_out)]
```

In [14]:
```
# La imagen en escala de gris
imshow(gray_out)
```



In [15]:
```
# Ordenamos los valores de más exito a menos
ordered = [results[idx] for idx in np.argsort(results)[::-1]]
#Nos quedamos con aquel factor f esta por encima del 1%
treshold = 0.01
#Obtenemos los indices de los elementos mas significantes
res_list = []
for i in range(0, len(ordered)) :
    if ordered[i] >= treshold :
        res_list.append(i)

# Inicializamos una imagen a 0 del mismo tamano que la imagen original
init_img = np.zeros(gray_out.shape)
# Acumulamos los segmentos cuyo factor f está por encima del valor threshold
# Nos quedamos con los segmentos que tienen más coincidencia
for iter_most_significants_values in res_list:
    init_img = init_img + (gray_out == np.unique(gray_out)[iter_most_significants_values])
```

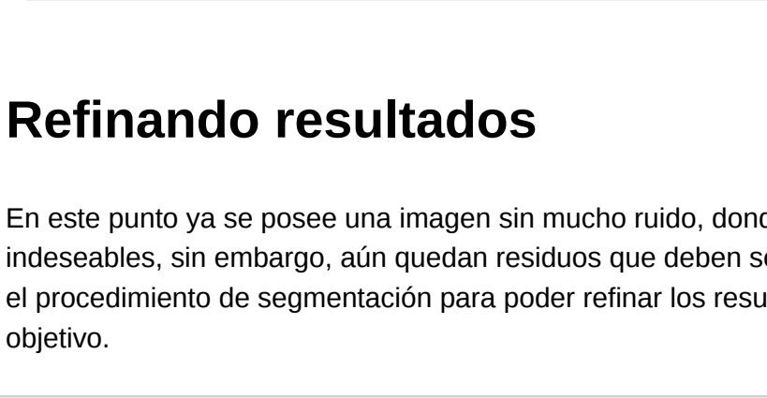
In [16]:
```
#Obtenemos una imagen donde lo que nos queda son elemento netamente indeseables
imshow(init_img)
```



In [17]:
```
#Se convierte la imagen a valores booleanos (0-1)
init_img.astype(bool)
```

Out[17]:
```
array([[ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True],
       ...,
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False]])
```

In [18]:
```
# Se realiza un filtrado entre la imagen segmentada (con elemento inecesarios) y
# la imagen obtenida en el preprocesado (con todos los elementos)
# quedando netamente los elementos de interés
final = masked * gray2rgb(init_img)
imshow(final.astype(np.uint8))
```



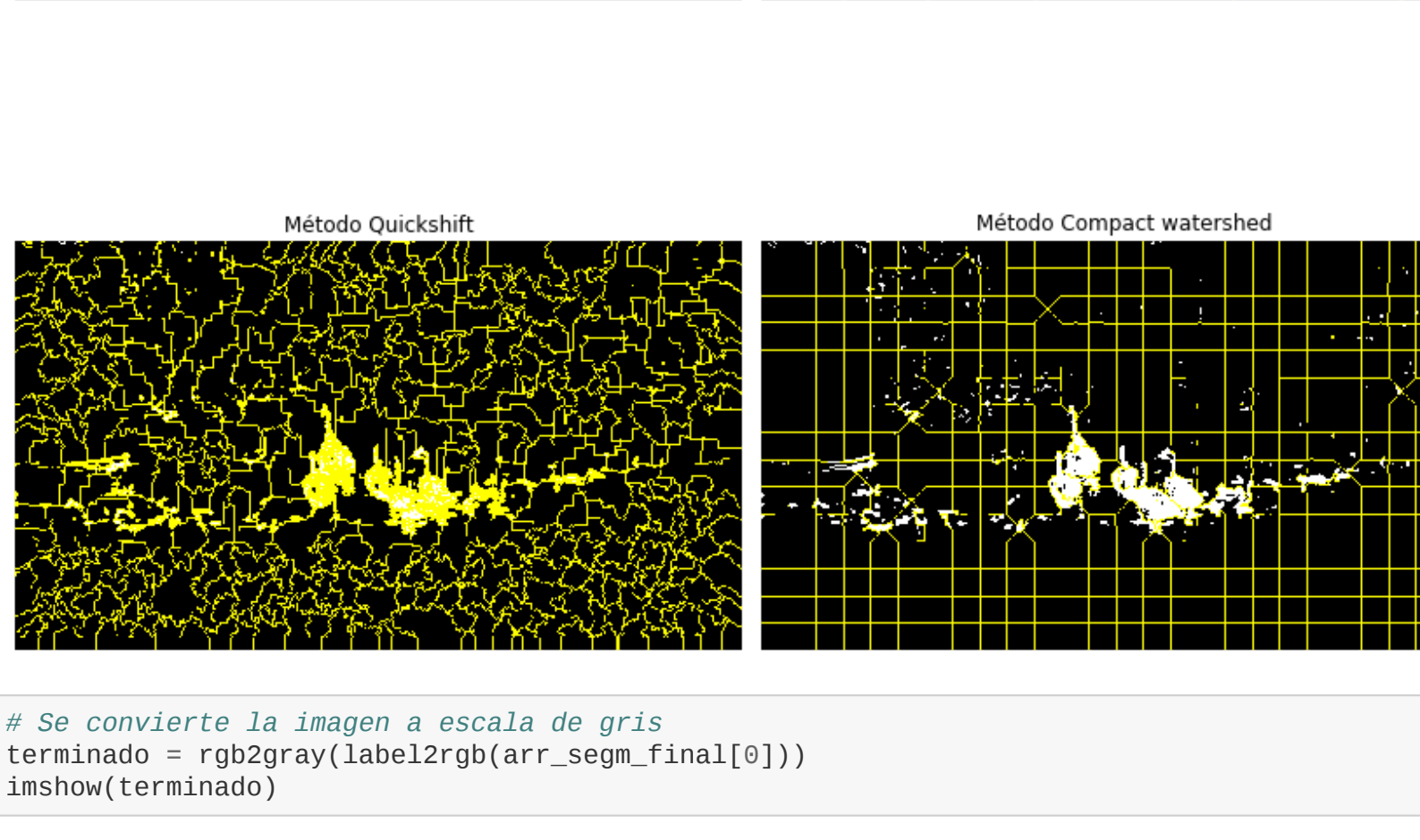
Refinando resultados

En este punto ya se posee una imagen sin mucho ruido, donde se ha logrado eliminar la mayor cantidad de elementos indeseables, sin embargo, aún quedan residuos que deben ser eliminados, por tal motivo se procede a reaplicar nuevamente el procedimiento de segmentación para poder refinar los resultados y mejorar la aproximación a la respuesta planteada como objetivo.

In [19]:
```
#Implementacion de segmentacion
arr_segm_final = implementar_segme(final, 1500)

#Arreglos a ser graficados
arr_img_final = []

#Creacion de imagenes y segmentos generados
arr_img_final.append(mark_boundaries(final, arr_segm_final[0]))
arr_img_final.append(mark_boundaries(final, arr_segm_final[1]))
arr_img_final.append(mark_boundaries(final, arr_segm_final[2]))
arr_img_final.append(mark_boundaries(final, arr_segm_final[3]))

imshow_grid(arr_img_final, arr_title)
```

```
Felzenszwalb cantidad de segmentos: 5
SLIC cantidad de segmentos: 29
Quickshift cantidad de segmentos: 815
Compact watershed cantidad de segmentos: 405
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..25
5] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..25
5] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..25
5] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..25
5] for integers).
```



In [20]:
```
# Se convierte la imagen a escala de gris
terminado = rgb2gray(label2rgb(arr_segm_final[0]))
imshow(terminado)
```

In [21]:
```
#Se procede a discretizar valores para seleccionar solo lo mas relevantes
terminado[terminado <= 0.1] = 1
imshow(terminado)
```

In [22]:
```
# Se aplica el filtro Otsu para igualar los valores
image = terminado
thresh = threshold_otsu(image)
#Se convierte valores a binario
binary = image > thresh
imshow(binary)
```

Resultado de la segmentación

Evaluación de la segmentación

- Si es cercano 1, es el resultado es mejor.
- Si es cercano a 0 indica que la segmentación no es buena.

Al depurar la segmentación se consiguió valores de factor **f** por encima del 0.5, exactamente **0.78**.

In [23]:
```
factor_f_evaluation(binary > 0, gt_img)
```

Out[23]:
```
0.7866062988208369
```

Evaluación del Grupo

- Todos los miembros se han integrado al trabajo del grupo : Si
- Todos los miembros participan activamente : Si
- Todos los miembros respetan otras ideas aportadas : Si
- Todos los miembros participan en la elaboración del informe : Si
- Me he preocupado por realizar un trabajo cooperativo con mis compañeros : Si
- Señala si consideras que algún aspecto del trabajo en grupo no ha sido adecuado: No

.

In [ ]:

In [ ]:

In [ ]: