



Tecnológico  
de Monterrey

## Computer Vision (Gpo 10)

### Professors:

- Dr. Gilberto Ochoa Ruiz
- Mtra. Yetnalezi Quintas Ruiz

### Students:

- Armando Bringas Corpus - A01200230
- Guillermo Alfonso Muñiz Hermosillo - A01793101
- Jorge Luis Arroyo Chavelas - A01793023
- Samantha R Mancias Carrillo - A01196762
- Sofia E Mancias Carrillo - A01196563

## Image Segmentation - Watershed

### Table of Contents

1. [Libraries](#)
2. [Implementation with OpenCV](#)
  - a. [Marker-based image segmentation](#)
3. [Implementation with skimage](#)
  - a. [Compact Watershed](#)
  - b. [Morphological Watershed Algorithm](#)
4. [Applications](#)
  - a. [Magnetic Resonance Imaging \(MRI\) - Brain Segmentation](#)
  - b. [Cell Nuclei Analysis - Osteosarcoma](#)

# Libraries

In [1]:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
from skimage.color import rgb2gray, label2rgb
from skimage.feature import canny
from skimage.filters import sobel
from skimage.segmentation import mark_boundaries, watershed, clear_border
from skimage.io import imread
from scipy import ndimage
from skimage import measure, color, io
from scipy import ndimage as ndi
from skimage import data

/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/site-packages/skimage/i
o/manage_plugins.py:23: UserWarning: Your installed pillow version is < 8.1.2. Several s
ecurity issues (CVE-2021-27921, CVE-2021-25290, CVE-2021-25291, CVE-2021-25293, and mor
e) have been fixed in pillow 8.1.2 or higher. We recommend to upgrade this library.
    from .collection import imread_collection_wrapper
```

## Implementation with Open CV

### Marker-based image segmentation

In [2]:

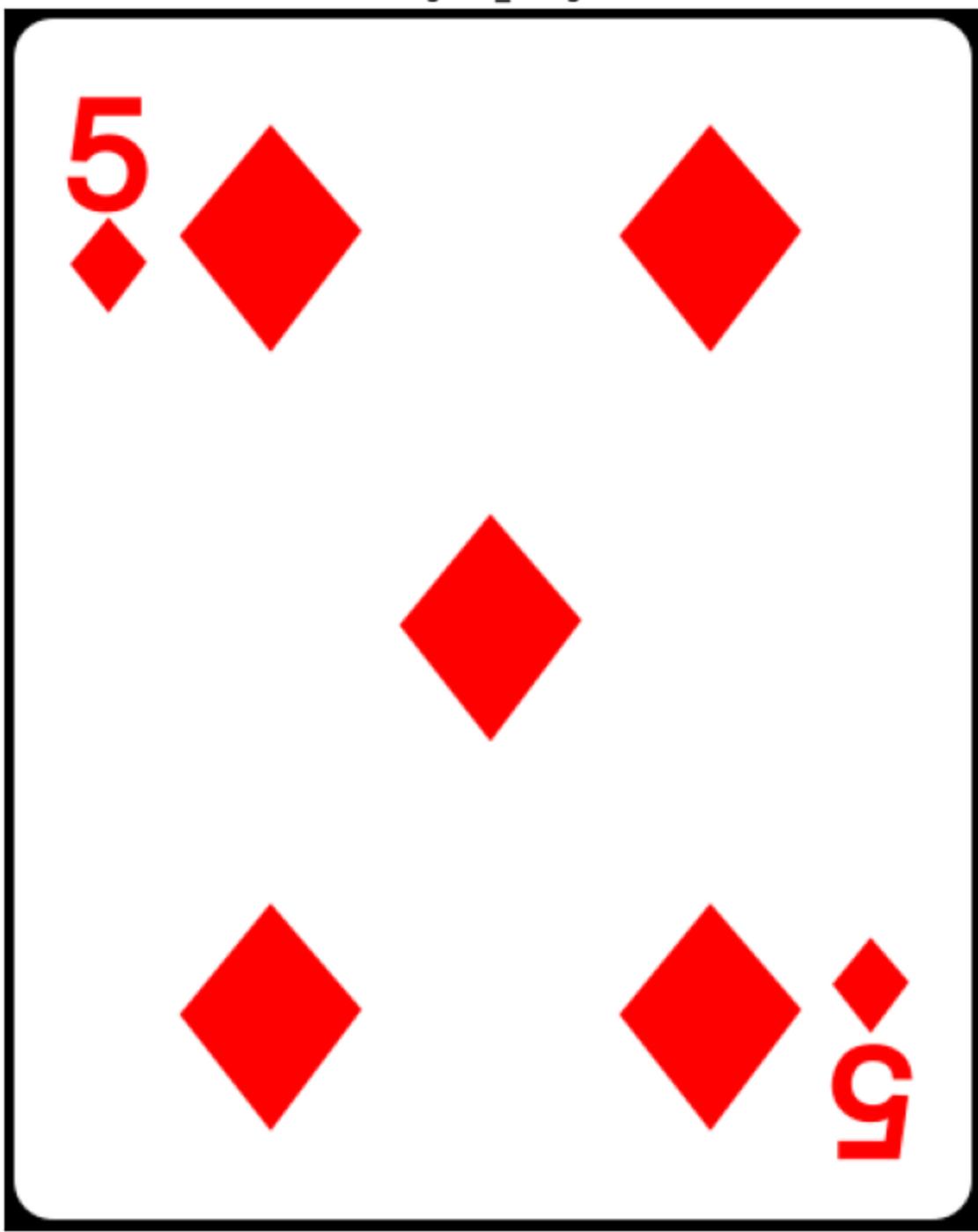
```
def plot_images(img1, img2, img1_title, img2_title):
    plt.figure(figsize=[20, 20])
    plt.subplot(121), plt.imshow(img1, cmap='gray')
    plt.title(img1_title, size=15), plt.xticks([]), plt.yticks([])
    plt.subplot(122), plt.imshow(img2, cmap='gray')
    plt.title(img2_title, size=15), plt.xticks([]), plt.yticks([])
    plt.show()
```

In [3]:

```
# Load image
img = cv2.imread('img/card.png')
img_RGB = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

plt.figure(figsize=[9, 9])
plt.imshow(img_RGB), plt.title('Original_image'), plt.xticks([]), plt.yticks([])
plt.show()
```

Original\_image



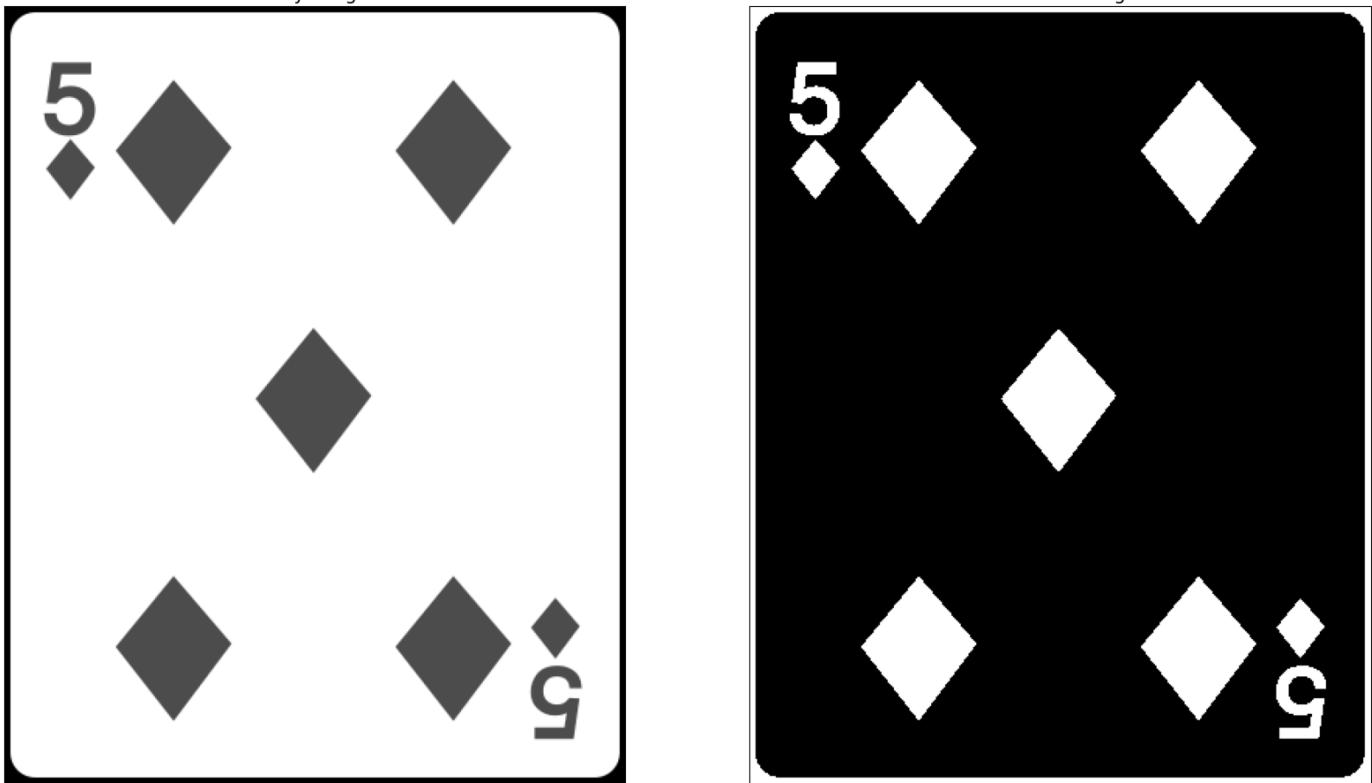
Convertimos la imagen importada a escala de grises y realizamos un operación de umbralizado por medio del algoritmo de binarización de Otsu el cual nos permite dividir la imagen en dos regiones, en negros y blancos.

In [4]:

```
# convert from color to grayscale
gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Divide the image into two regions, blacks and whites
ret, thresh = cv2.threshold(gray_img, 0, 255, cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)

plot_images(gray_img, thresh, 'Gray Image', 'Threshold image')
```



A continuación, es normal que nuestra imagen pueda contener ruido, lo cual puede dificultar el procesamiento posterior por lo que se puede aplicar una transformación morfológica que implica la dilatación o erosión de las regiones blancas de la imagen en varios pasos para eliminar el ruido. Posteriormente, se puede utilizar la operación abierta morfológica, que consiste en erosionar después de dilatar, para que de esta manera las regiones blancas grandes se expandan y se "tragen" las regiones negras pequeñas, lo cual contribuye a eliminar el ruido. Las regiones negras grandes, que representan objetos reales, permanecen relativamente sin cambios. Se puede utilizar la función cv2.morphologyEx con el argumento cv2.MORPH\_OPEN para realizar esta operación.

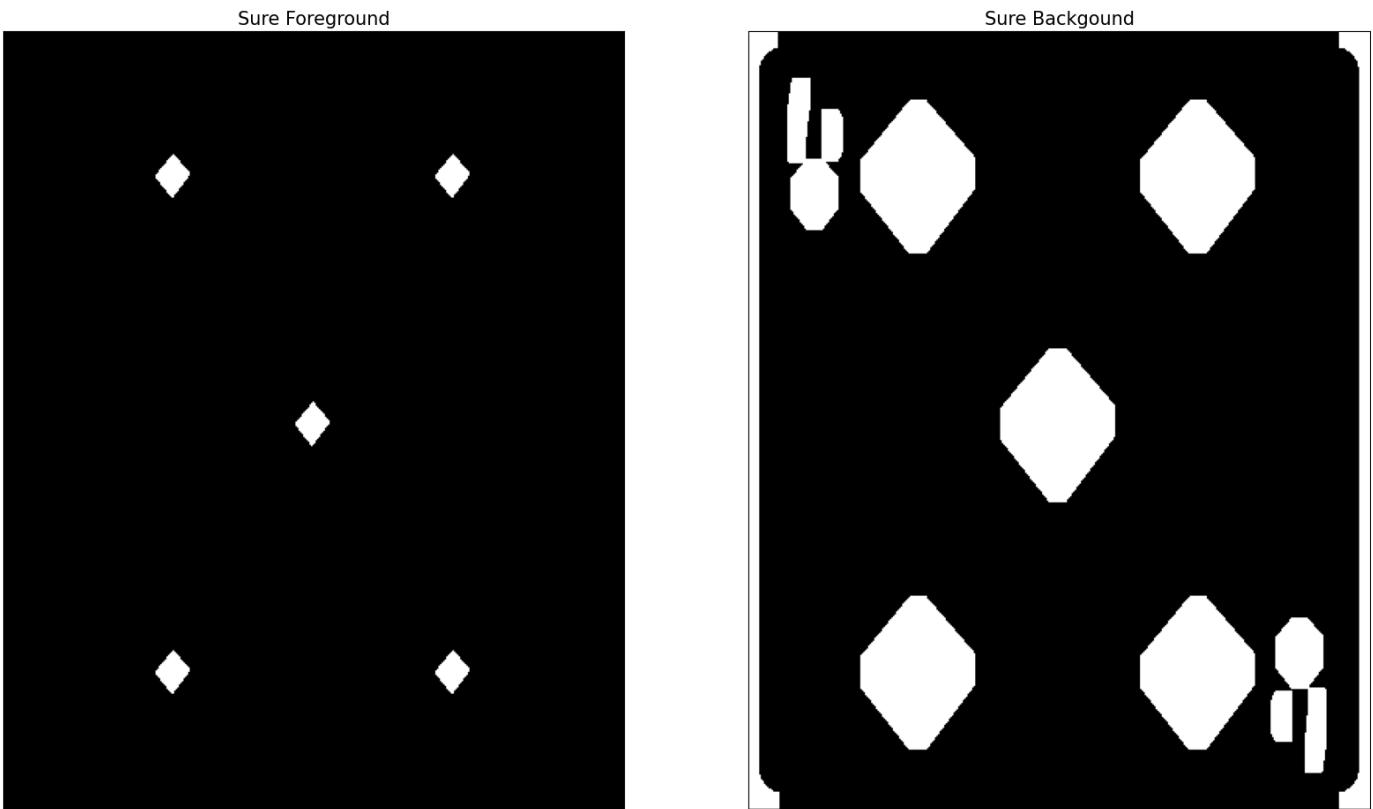
Posteriormente, se puede dilatar el resultado de la transformación abierta para obtener regiones que probablemente sean el fondo de la imagen. Para la obtención de regiones seguras del primer plano, se puede aplicar una transformación de distancia la cual en términos prácticos, podemos estar más seguros de que un punto es realmente parte del primer plano si está lejos del borde del primer plano-fondo más cercano. Después de aplicar la transformación de distancia, se puede aplicar un umbral para seleccionar las regiones que son seguras para el primer plano.

Las regiones intermedias, que no son seguras ni para el primer plano ni para el fondo, se pueden encontrar restando el primer plano seguro del fondo seguro. Estas regiones son denominadas como regiones inseguras o desconocidas.

In [5]:

```
# noise removal
kernel = np.ones((5,3),np.uint8)
opening = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel, iterations = 2)
# sure background area
sure_bg = cv2.dilate(opening,kernel,iterations=3)
# Finding sure foreground area
dist_transform = cv2.distanceTransform(opening, cv2.DIST_L2, 5)
ret, sure_fg = cv2.threshold(dist_transform, 0.7*dist_transform.max(), 255, 0)
# Finding unknown region
sure_fg = np.uint8(sure_fg)
unknown = cv2.subtract(sure_bg,sure_fg)
```

```
# Plot the results
plot_images(sure_fg, sure_bg, 'Sure Foreground', 'Sure Background')
```



Al encontrar las regiones anteriormente descritas, podemos construir nuestras barreras para evitar que el agua se mezcle. Esto se hace con la función `connectedComponents` que proviene de la teoría de grafos cuando analizamos el algoritmo GrabCut y conceptualizamos una imagen como un conjunto de nodos que están conectados por bordes. Teniendo las áreas seguras de primer plano, algunos de estos nodos se conectarán entre sí, pero otros no. Los nodos desconectados pertenecen a diferentes valles de agua, por lo que debe haber una barrera entre ellos.

Para los marcadores, agregamos 1 a las etiquetas para todas las regiones porque solo queremos que las incógnitas permanezcan en 0.

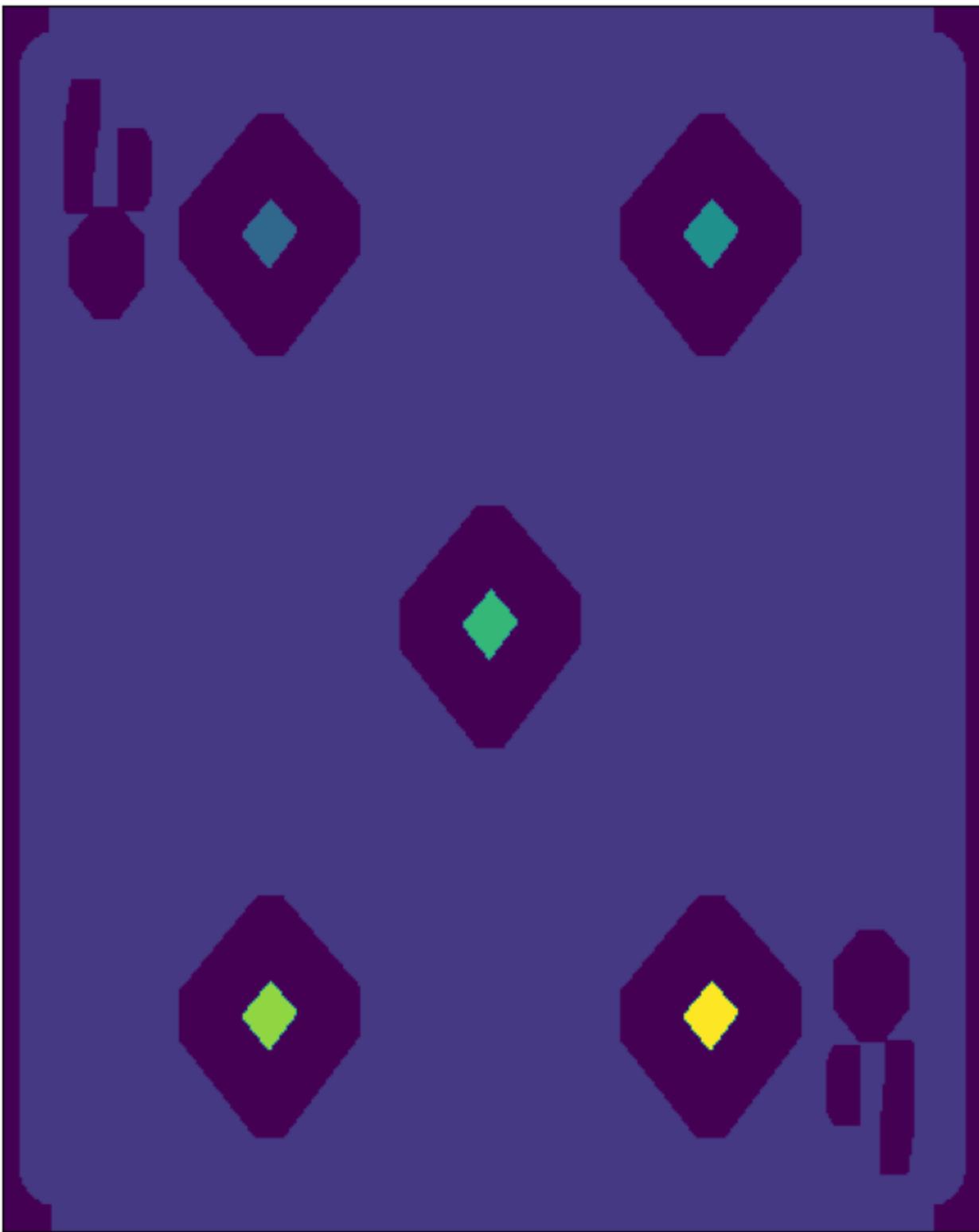
```
In [6]: # Marker labelling
ret, markers = cv2.connectedComponents(sure_fg)

# Add one to all labels so that sure background is not 0, but 1
markers = markers+1

# Now, mark the region of unknown with zero
markers[unknown==255] = 0

plt.figure(figsize=[10, 10])
plt.imshow(markers), plt.title('Marker Labelling'), plt.xticks([]), plt.yticks([])
plt.show()
```

## Marker Labelling

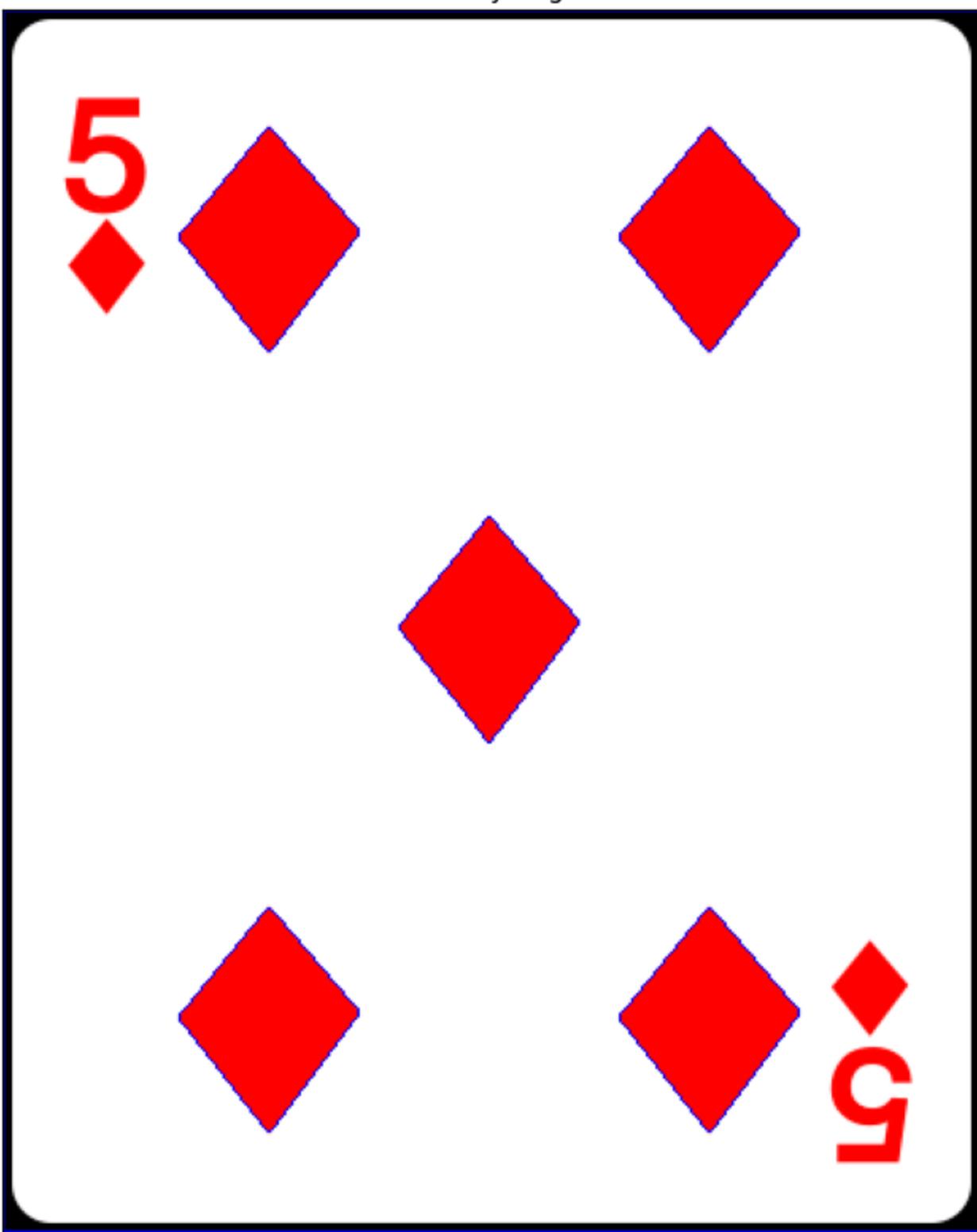


Finalmente ejecutamos el algoritmo de watershed, en cual dejamos que fluya el agua, la función cv2.watershed asigna la etiqueta -1 a los píxeles que son bordes entre componentes. Coloreamos estos bordes de azul en la imagen original y utilizamos la librería matplotlib para visualizar los resultados.

In [7]:

```
# Apply watershed, boundary region will be marked with -1
markers = cv2.watershed(img,markers)
img[markers == -1] = [255,0,0]

plt.figure(figsize=[10, 10])
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB)), plt.title('Boundary Region'), plt.xticks(),
plt.show()
```



En este caso la implementación funciona de forma adecuada, pero no para todos los elementos de la imagen. Al momento de probar con otras imágenes pudimos darnos cuenta de que no siempre se obtienen resultados satisfactorios, se requiere de hacer un ajuste de los parámetros en las operaciones morfológicas que se realizan. Sin embargo la implementación anteriormente expuesta de las más conocidas e implementada con la librería Open CV, existen diferentes variaciones para implementar Watershed, a continuación veremos algunas variaciones utilizando diferentes técnicas y las librerías skimage y simpleITK.

## Implementation with skimage

# Compact Watershed

El algoritmo de Watershed Compacto, es una variante del algoritmo de Watershed en el que se utiliza un parámetro de compactacion, el cual controla la suavidad de la línea divisoria entre los objetos segmentados.

- Un valor más alto dará como resultado una línea de segmentación más suave y se ajustará mejor a los contornos de los objetos.
- Un valor más bajo producirá una línea de segmentación más fragmentada y seguirá las características de la imagen de gradiente con mayor precisión

A diferencia del algoritmo de Watershed clásico, en el cual la línea divisoria se genera entre los mínimos locales en la imagen segmentada, lo que puede dar como resultado una línea muy fragmentada y desigual. En cambio la compactación, a veces también conocida como rigidez o resistencia, controla la forma en que la línea divisoria se dibuja entre los objetos segmentados.

```
In [8]: # Cargar la imagen y seleccionar cada pixel en cada dimension
img = imread('img/Peony-Rose-Flowers.jpg')[:, :, :3]

# Calcular el gradiente de la imagen, es decir como varia la imagen en diferentes direcciones
gradient = sobel(rgb2gray(img))

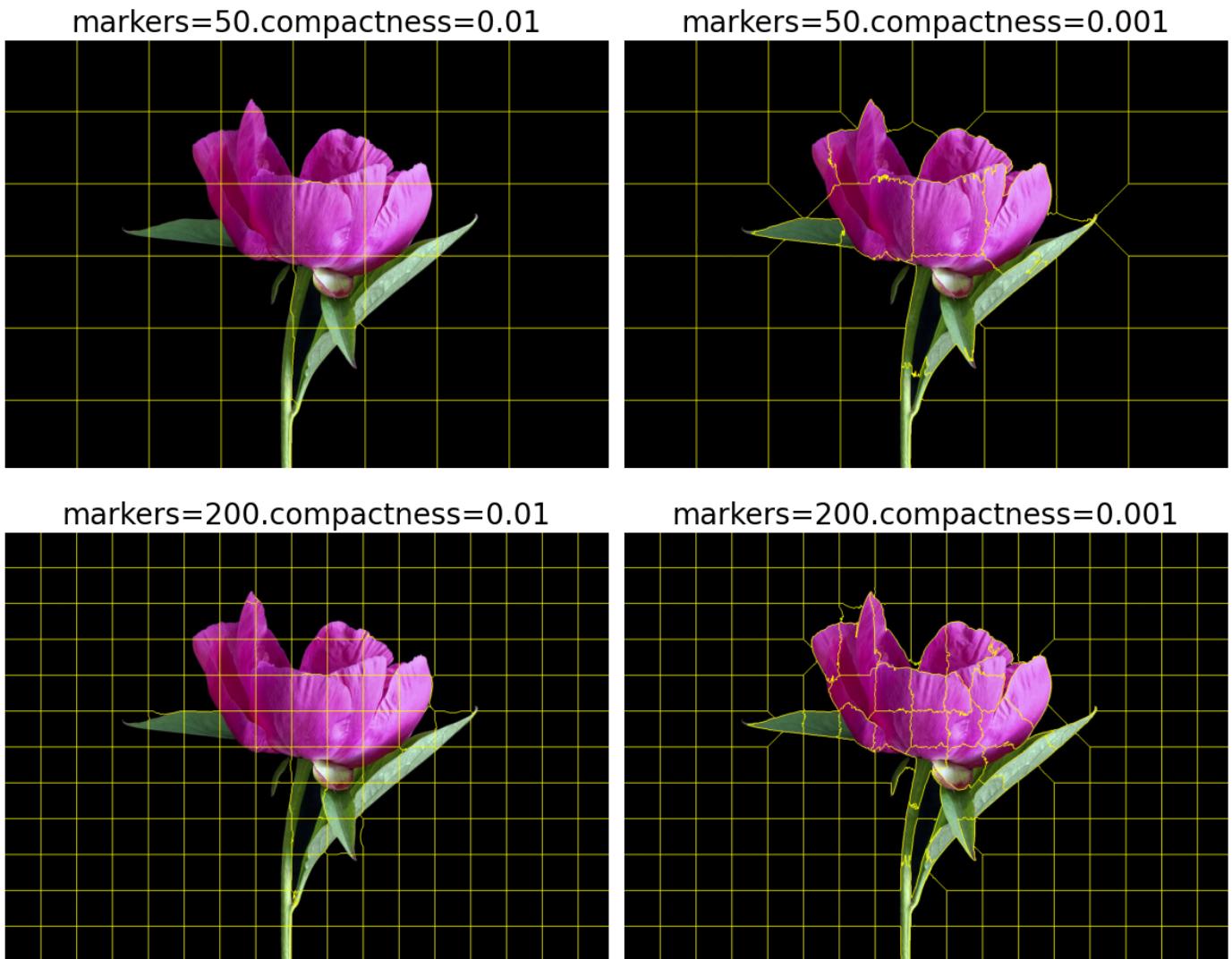
fig, axs = plt.subplots(nrows=2, ncols=2, figsize=(12, 12))
axs = axs.ravel()

# Configuracion de parametros para watershed.
# Lista de marcadores y lista de parametro de compactacion.
markers_list = [50, 50, 200, 200]
compactness_list = [0.01, 0.001, 0.01, 0.001]

# Aplicar el algoritmo watershed empaquetando los valores de las listas y alimentandolos
for i, (markers, compactness) in enumerate(zip(markers_list, compactness_list)):
    segments_watershed = watershed(gradient, markers=markers, compactness=compactness)
    axs[i].imshow(mark_boundaries(img, segments_watershed))
    axs[i].set_title(f"markers={markers}.compactness={compactness}", size=20)
    axs[i].axis('off')

plt.suptitle('Compact watershed', size=30)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

# Compact watershed



## Morphological Watershed Algorithm

El algoritmo watershed morfológico es una variación del algoritmo Watershed en la cual se utiliza la transformación morfológica, la distancia euclídea, el gradiente morfológico y el algoritmo watershed para separar los objetos en una imagen. Este método es útil en aplicaciones que requieren la segmentación precisa de objetos en una imagen, como en el análisis de imágenes médicas o en la inspección de calidad industrial.

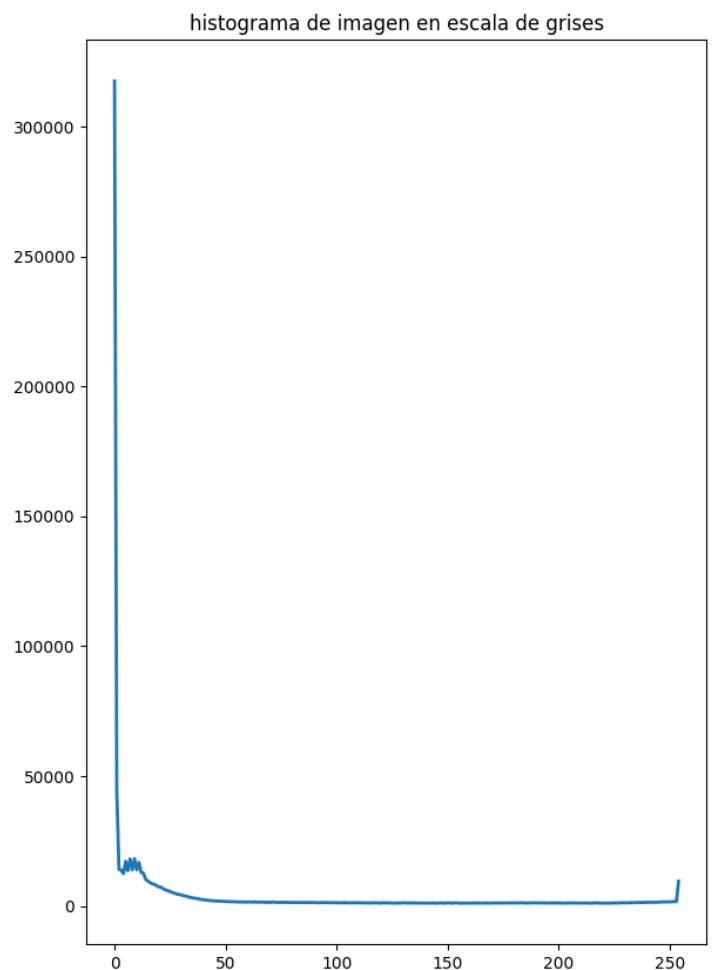
- Una *transformación morfológica* es un tipo de operación que se utiliza en el procesamiento de imágenes para modificar la forma, la estructura o el contenido de los objetos en una imagen binaria.
- El *gradiente morfológico* es una operación que se utiliza para resaltar los bordes en una imagen. Es una operación que se basa en la diferencia entre la dilatación y la erosión de una imagen.

Implementacion inspirada en Hands-On Image Processing with Python: Expert techniques for advanced image analysis and effective interpretation of image data. [2]

```
In [9]: coins = cv2.imread("img/coins2.jpg", 0)
# Invertimos los colores debido a que facilita la segmentacion contar con fondos negros
coins = 255 - coins
hist = np.histogram(coins, bins=np.arange(0, 256))
```

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 10))
ax1.imshow(coins, cmap=plt.cm.gray, interpolation='nearest')
ax1.axis('off')
ax2.plot(hist[1][:-1], hist[0], lw=2)
ax2.set_title('histograma de imagen en escala de grises')
```

Out[9]: Text(0.5, 1.0, 'histograma de imagen en escala de grises')



In [27]: # Aplicamos el umbralado a la imagen en escala de grises

```
# Aplicamos el umbralado a la imagen en escala de grises
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 12), sharex=True, sharey=True)
ax1.imshow(coins > 15, cmap=plt.cm.gray, interpolation='nearest')
ax1.set_title('coins > 15')
ax1.axis('off')
ax1.set_adjustable('box')
ax2.imshow(coins > 100, cmap=plt.cm.gray, interpolation='nearest')
ax2.set_title('coins > 100')
ax2.axis('off')
margins = dict(hspace=0.01, wspace=0.01, top=1, bottom=0, left=0, right=1)
fig.subplots_adjust(**margins)
```



```
In [82]: # Calculamos el gradiente de la imagen para calcular los bordes de la imagen
elevation_map = sobel(coins)

fig, ax = plt.subplots(figsize=(15, 12))
ax.imshow(elevation_map, cmap=plt.cm.gray, interpolation='nearest')
ax.axis('off')
ax.set_title('Puntos de Elevacion')
```

Out[82]: Text(0.5, 1.0, 'Puntos de Elevacion')



```
In [84]: markers = np.zeros_like(coins)
# Creamos los marcadores etiquetando los diferentes valores de nuestra imagen de acuerdo
markers[coins < 15] = 1
markers[coins > 100] = 2

fig, ax = plt.subplots(figsize=(15, 12))
ax.imshow(markers, cmap=plt.cm.Spectral, interpolation='nearest')
```

```
ax.axis('off')
ax.set_title('Marcadores')
```

Out[84]: Text(0.5, 1.0, 'Marcadores')



```
In [85]: # Una vez obtenidos los marcadores aplicamos el algoritmo watershed usando estos marcadores
segmentation = watershed(elevation_map, markers)
fig, ax = plt.subplots(figsize=(15, 12))
ax.imshow(segmentation, cmap=plt.cm.gray, interpolation='nearest')
ax.axis('off')
ax.set_title('Imagen Segmentada')
```

Out[85]: Text(0.5, 1.0, 'Imagen Segmentada')

Imagen Segmentada



Finalmente podemos sobreponer nuestra imagen segmentada a la imagen original y resaltar los bordes para obtener las líneas de puntos mas altos en la imagen original y asi observar el trabajo realizado.

```
In [86]: segmentation = ndi.binary_fill_holes(segmentation - 1)
labeled_coins, _ = ndi.label(segmentation)

image_label_overlay = label2rgb(labeled_coins, image=coins)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(30, 24), sharex=True, sharey=True)
ax1.imshow(coins, cmap=plt.cm.gray, interpolation='nearest')
ax1.contour(segmentation, [0.2], linewidths=3, colors='r')
ax1.axis('off')
ax1.set_title('Imagen segmentada con bordes')
ax1.set_adjustable('box')
ax2.imshow(image_label_overlay, interpolation='nearest')
ax2.axis('off')
ax2.set_title('Imagen Etiquetada sobrepuesta ')
ax2.set_adjustable('box')
```



# Applications

## Magnetic Resonance Imaging (MRI) - Brain Segmentation

In [168...]

```
%matplotlib inline
import matplotlib.pyplot as plt
import SimpleITK as sitk
from myshow import myshow, myshow3d

# Download data to work on]
%run update_path_to_download_script
from downloaddata import fetch_data as fdata

from ipywidgets import interact, interactive, FloatSlider
```

<Figure size 640x480 with 0 Axes>

En los siguientes ejemplos se emplea la librería de SimpleITK y una implementación de watershed basada en etiquetado y Otsu. Se emplean dos parámetros sigma y un nivel de watershed que permiten ajustar el resultado de la segmentación. El primer sigma controla qué tan cerca pueden estar las regiones detectadas (spot\_sigma) y el segundo controla qué tan precisamente se contornan los objetos segmentados (outline\_sigma). De forma general, este filtro aplica dos desenfoques gaussianos, detección de manchas, umbralización de Otsu y etiquetado por medio de Voronoi. La imagen binaria umbralizada se inunda utilizando Voronoi a partir de los máximos locales encontrados. Cabe destacar que la eliminación de ruido sigma para la detección de manchas y la umbralización se pueden configurar por separado.

Esta implementación está inspirado en el método de Voronoi-Otsu-Labeling [1].

In [169...]

```
def watershed_otsu_labeling(img, spot_sigma=2, outline_sigma=2, watershed_level=10, show_=_
    blurred_spots = sitk.GradientMagnitudeRecursiveGaussian(img, sigma=spot_sigma)
    blurred_outline = sitk.DiscreteGaussian(img, variance=[outline_sigma, outline_sigma,
    binary_otsu = sitk.OtsuThreshold(blurred_outline, 0, 1)
    ws = sitk.MorphologicalWatershed(blurred_spots, markWatershedLine=show_watershed_edg_
    labels = sitk.Mask(ws, sitk.Cast(binary_otsu, ws.GetPixelID()))
    if show_RGB:
        myshow(sitk.LabelToRGB(labels), title="Watershed Segmentation")
    else:
        myshow(labels, title='Watershed Segmentation')
```

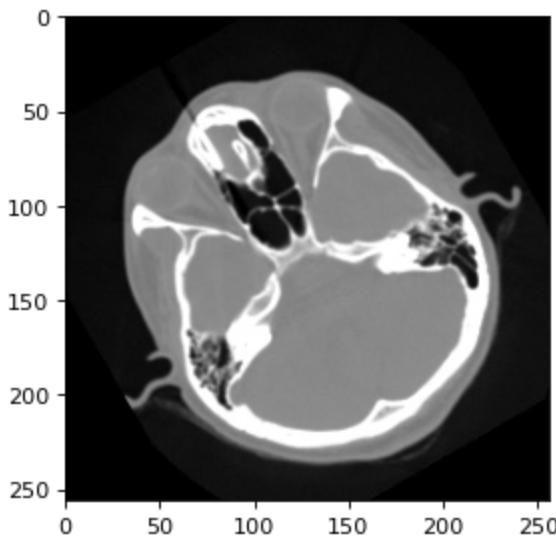
El siguiente ejemplo corresponde a una imagen obtenida por medio de MRI del plano transversal del cerebro humano, el algoritmo de Watershed nos puede ayudar a identificar estructuras cerebrales y a identificar regiones que sean de interés, como por ejemplo separar la materia blanca de la gris, o para la identificación de los giros y regiones del cerebro como los lóbulos, núcleos y corteza.

De igual manera, el algoritmo de Watershed puede mejorar la precisión y la reproducibilidad de las mediciones en la imagen de resonancia magnética del cerebro, que puede ser de gran utilidad en la investigación, el diagnóstico de enfermedades cerebrales y el análisis de trastornos neurológicos.

In [47]:

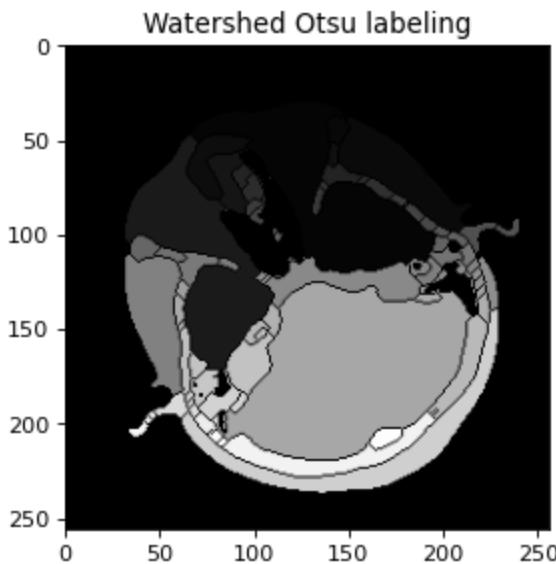
```
img = sitk.ReadImage(fdata("cthead1.png"))
myshow(img)
```

Fetching cthead1.png

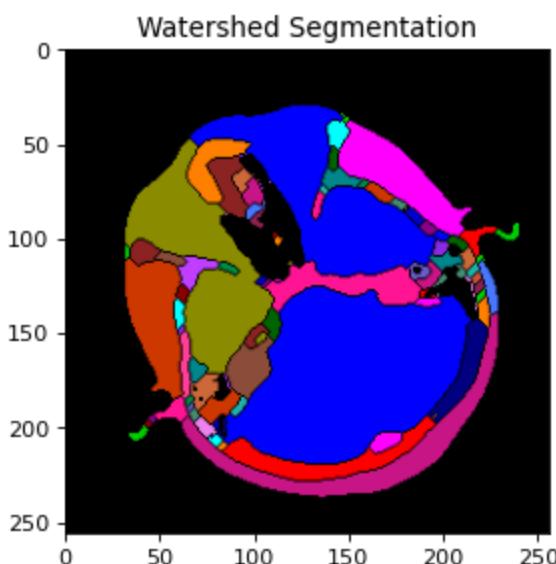


Al ajustar los parámetros sigma y nivel de watershed podemos obtener diferentes segmentaciones de acuerdo al área de interés, de igual manera podemos cambiar el color de las regiones detectadas para facilitar su identificación.

```
In [48]: watershed_otsu_labeling(img, spot_sigma=2, outline_sigma=2, watershed_level=5, show_water
```



```
In [49]: watershed_otsu_labeling(img, spot_sigma=2, outline_sigma=2, watershed_level=5, show_water
```



```
In [50]: def callback(img, *args, **kwargs):
    watershed_otsu_labeling(img, *args, **kwargs)

    interact(
        lambda **kwargs: callback(img, **kwargs),
        spot_sigma = FloatSlider(min=.1, max=5, step=.1, value=2),
        outline_sigma = FloatSlider(min=.1, max=5, step=.1, value=2),
        watershed_level = FloatSlider(min=1, max=10, step=.1, value=5),
        show_watershed_edge = True,
        show_RGB = True
    )

    interactive(children=(FloatSlider(value=2.0, description='spot_sigma', max=5.0, min=0.1), FloatSlider(value=2...
Out[50]: <function __main__.<lambda>(**kwargs)>
```

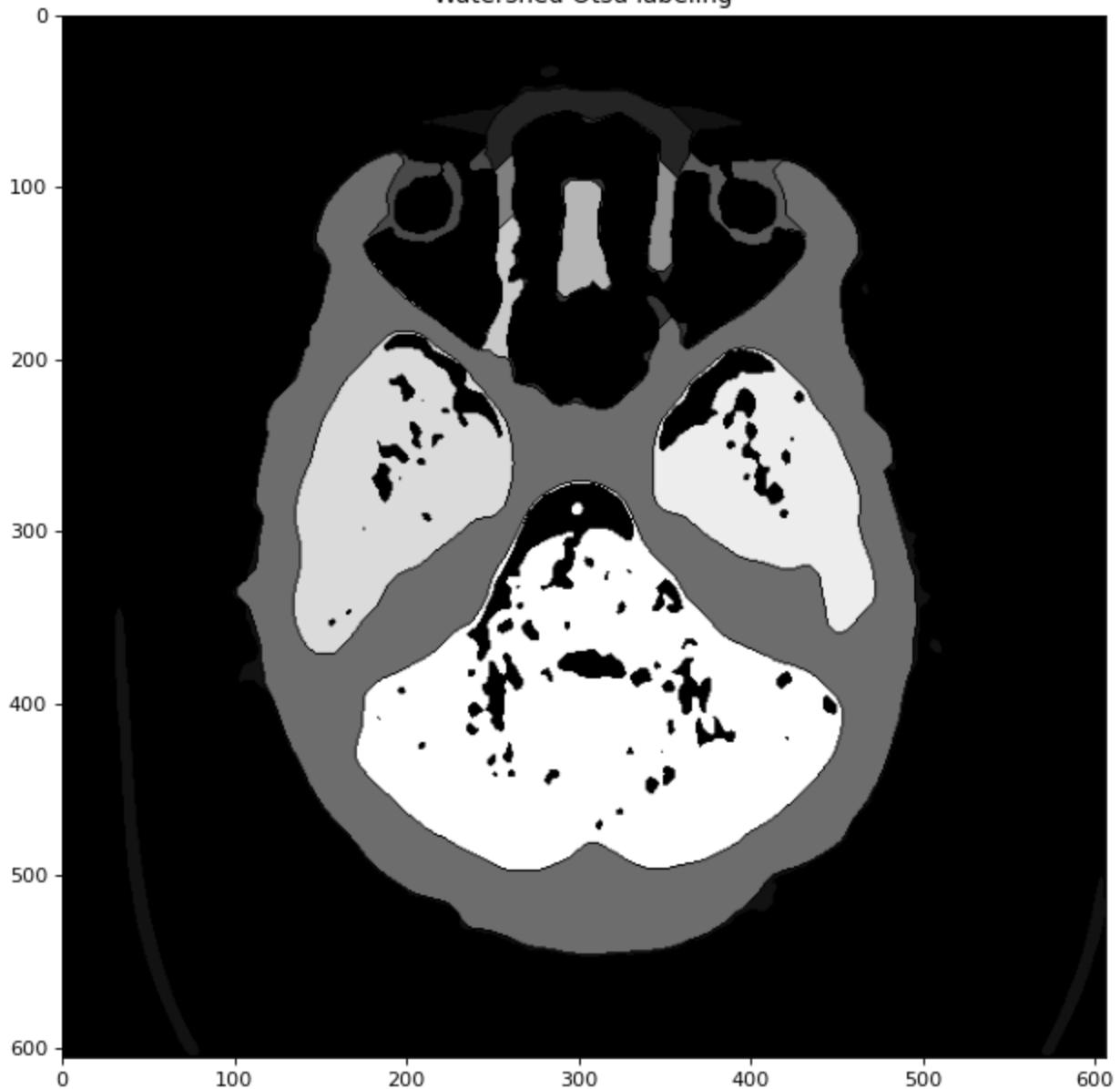
Este es otro ejemplo de otro corte realizado en el plano sagital. Pudimos percibirnos que cada caso es específico y que los parámetros del algoritmo se tienen que ajustar con base al caso que se tenga.

```
In [51]: img = sitk.ReadImage ('img/normal-ct-brain.jpg', sitk.sitkFloat32)
myshow(img)
```



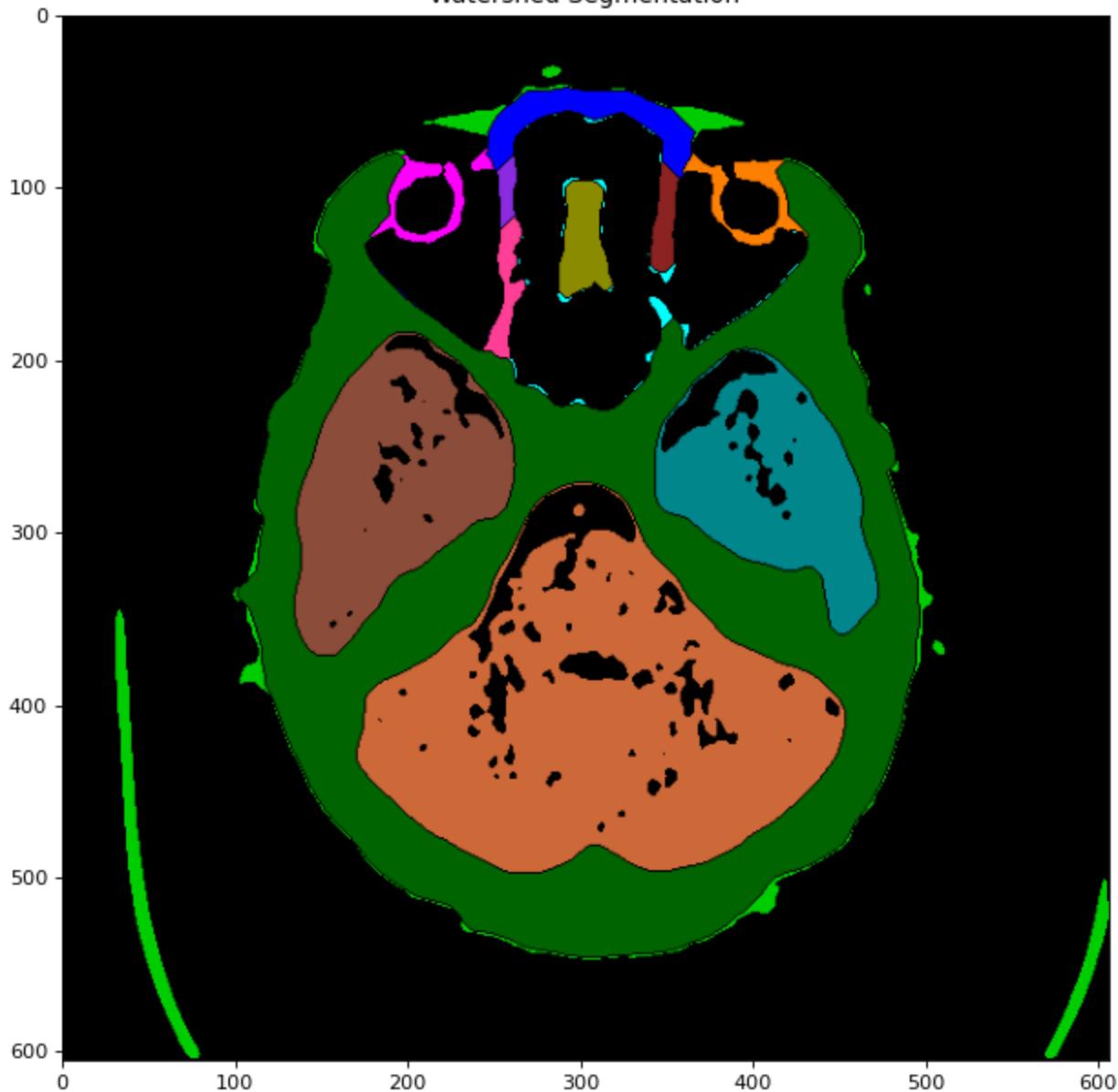
```
In [52]: watershed_otsu_labeling(img, spot_sigma=5, outline_sigma=5, watershed_level=5, show_water
```

Watershed Otsu labeling



```
In [53]: watershed_otsu_labeling(img, spot_sigma=5, outline_sigma=5, watershed_level=5, show_water
```

## Watershed Segmentation



```
In [54]: def callback(img, *args, **kwargs):
    watershed_otsu_labeling(img, *args, **kwargs)

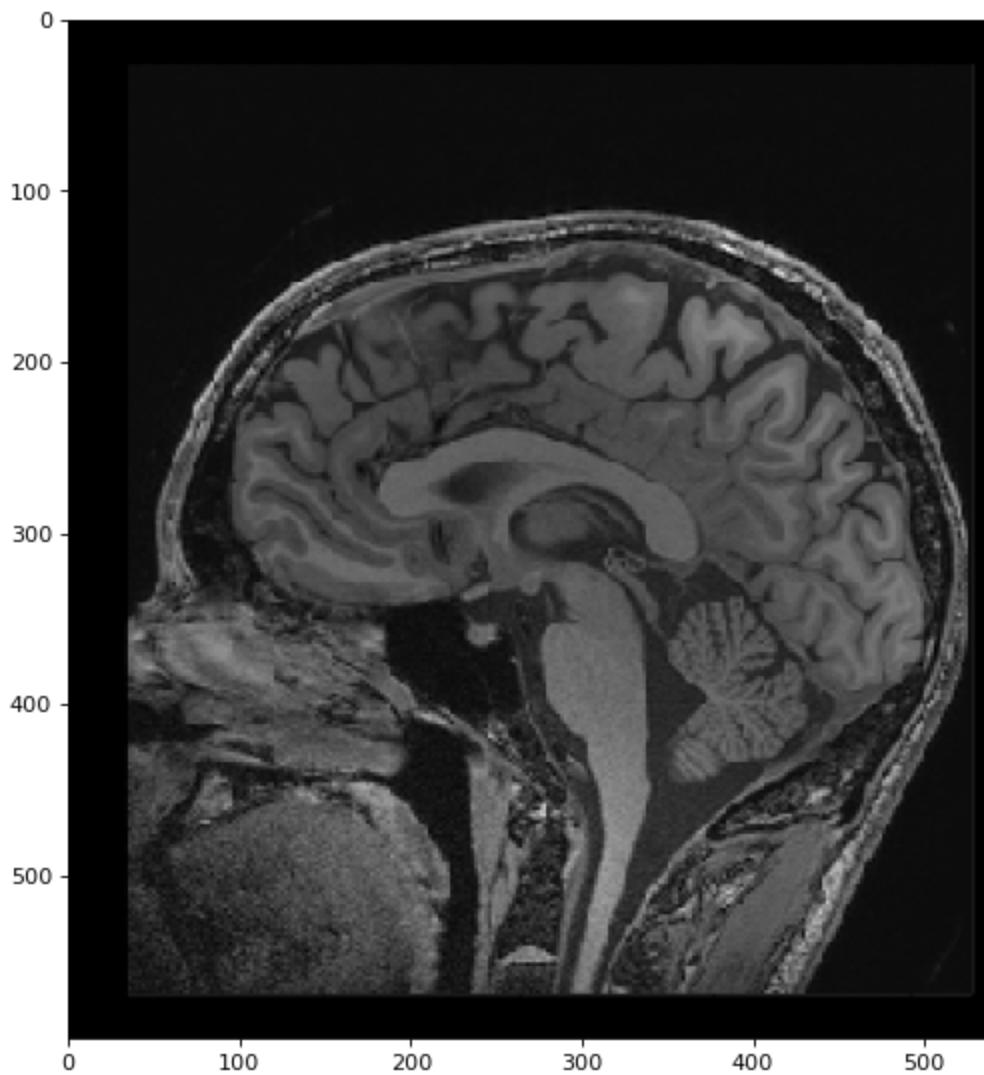
    interact(
        lambda **kwargs: callback(img, **kwargs),
        spot_sigma = FloatSlider(min=.1, max=5, step=.1, value=5),
        outline_sigma = FloatSlider(min=.1, max=5, step=.1, value=5),
        watershed_level = FloatSlider(min=1, max=10, step=.1, value=5),
        show_watershed_edge = True,
        show_RGB = True
    )

interactive(children=(FloatSlider(value=5.0, description='spot_sigma', max=5.0, min=0.1), FloatSlider(value=5.0, description='outline_sigma', max=5.0, min=0.1), FloatSlider(value=5.0, description='watershed_level', max=10.0, min=1.0), Output()), layout=Layout(display='flex', align_items='center', justify_content='space-around'))
```

Out[54]: <function \_\_main\_\_.lambda(\*\*kwargs)>

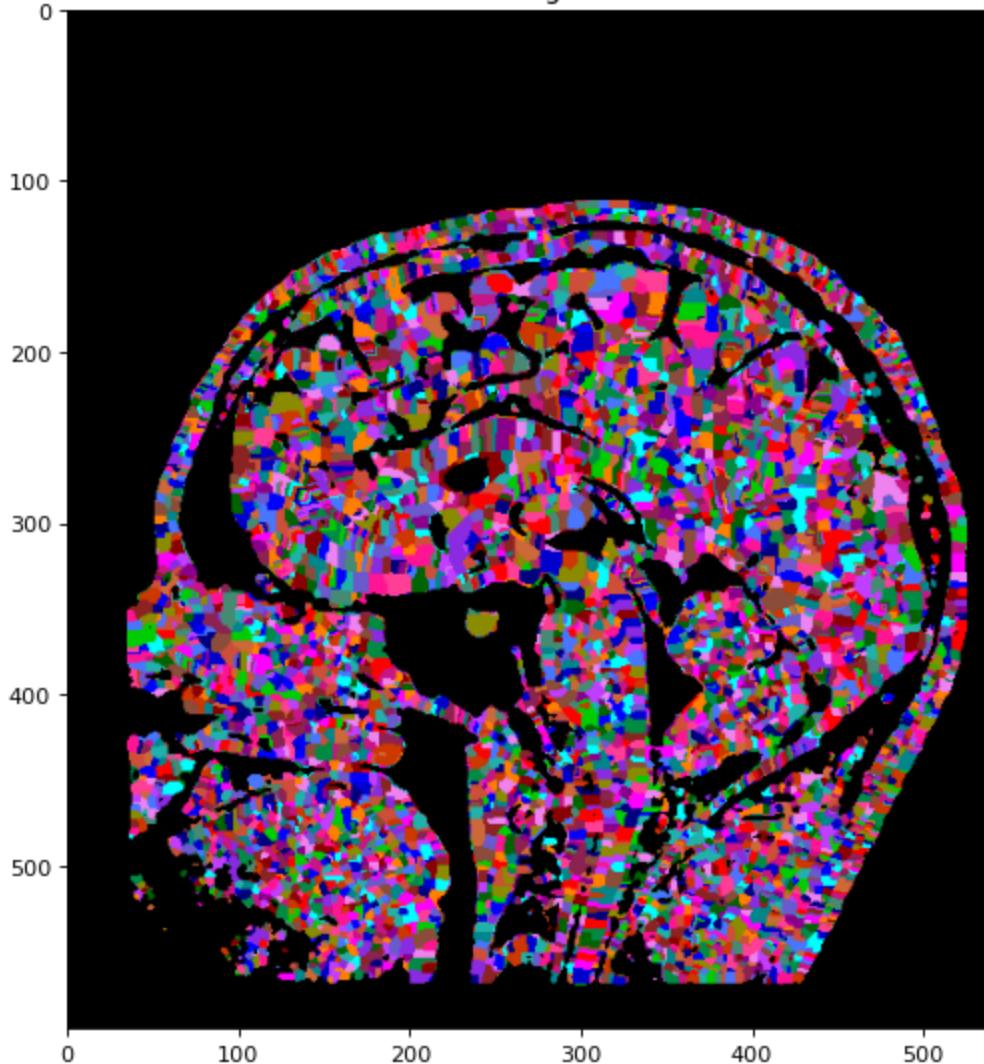
A continuación vemos un caso de un corte en plano sagital en donde por más que se intento ajustar los parámetros los resultados fueron de sobre-segmentación, inferimos que se puede deber por la resolución de la imagen y que se necesita trabajar más la parte de preprocesamiento o incluso probar con algún otro algoritmo de segmentación. Sin embargo, decidimos dejar el ejemplo ya que nos pareció visualmente llamativo y también como ejemplo de los problemas y desventajas que podemos enfrentar al utilizar este algoritmo.

```
In [25]: img = sitk.ReadImage('img/MRI_2.png', sitk.sitkFloat32)
myshow(img)
```



```
In [26]: watershed_otsu_labeling(img, spot_sigma=2, outline_sigma=2, watershed_level=3.7, show_wat
```

## Watershed Segmentation



```
In [27]: def callback(img, *args, **kwargs):
    watershed_otsu_labeling(img, *args, **kwargs)

    interact(
        lambda **kwargs: callback(img, **kwargs),
        spot_sigma = FloatSlider(min=.1, max=5, step=.1, value=2),
        outline_sigma = FloatSlider(min=.1, max=5, step=.1, value=2),
        watershed_level = FloatSlider(min=1, max=10, step=.1, value=3.7),
        show_watershed_edge = False,
        show_RGB = True
    )

interactive(children=(FloatSlider(value=2.0, description='spot_sigma', max=5.0, min=0.1), FloatSlider(value=2.0, description='outline_sigma', max=5.0, min=0.1), FloatSlider(value=3.7, description='watershed_level', max=10.0, min=1.0), Checkbox(value=False, description='show_watershed_edge'), Checkbox(value=True, description='show_RGB')))

Out[27]: <function __main__.<lambda>(**kwargs)>
```

## Analisis de Nucleo de Celulas - Osteosarcoma

En las siguientes celdas explicamos como llevar a cabo el analisis de imagenes de muestras de laboratorio para llevar a cabo la segmentacion de los nucleos de las celula.

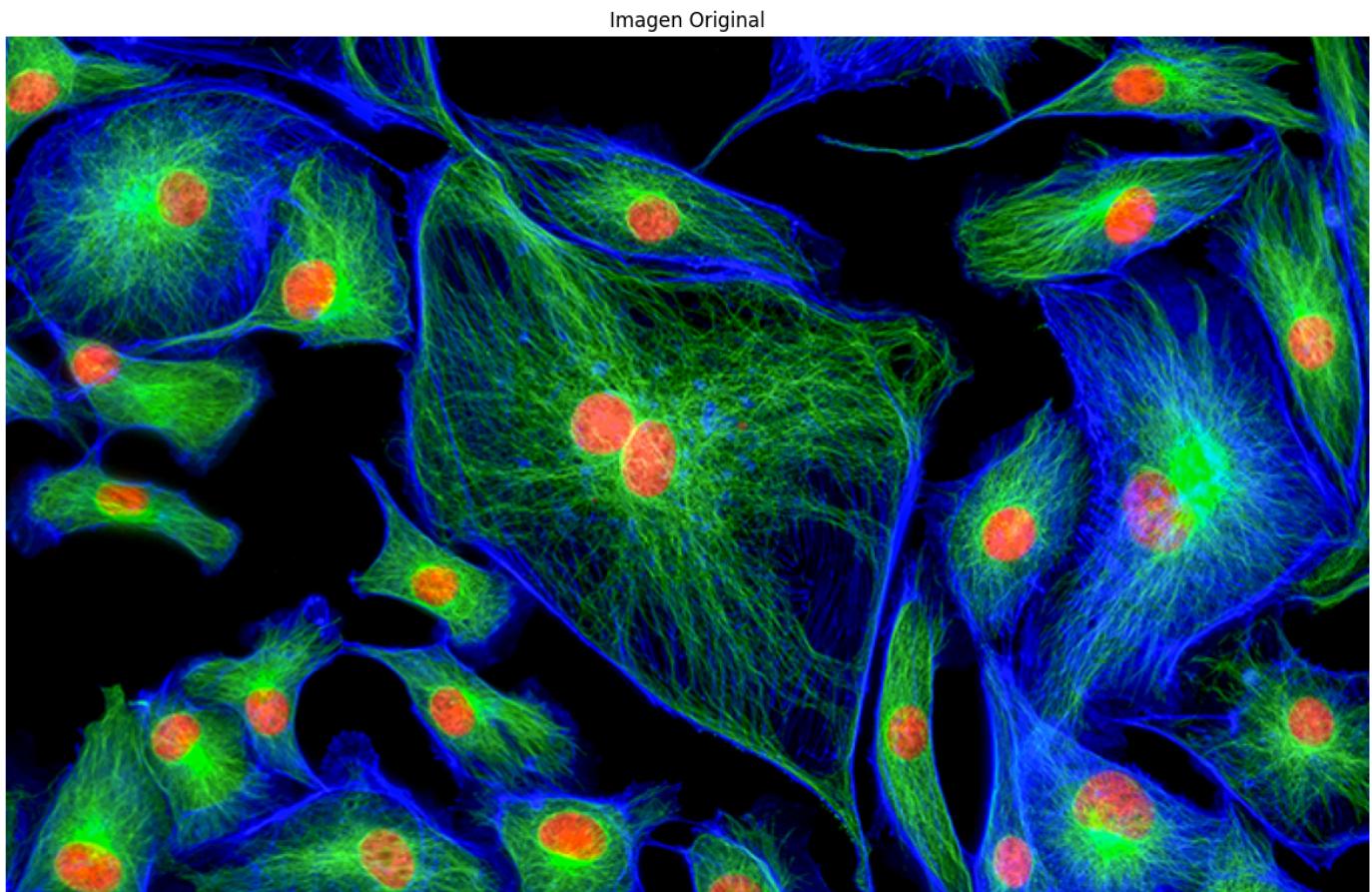
Esta tecnica funciona con imagenes de fluorescencia microscopica, en las cuales las muestras han sido coloreadas con tinta de color AZUL llamada DAPI. La cual es una buena manera que tienen los cientificos para separar los nucleos de las celulas, es decir, es un proceso que facilita la segmentacion de la imagen,

haciendo solamente necesario aplicar tecnicas de umbralado y separacion para tener una mejor segmentacion.

Esta implementacion esta inspirada en Cell Nuclei analysis in Python using watershed segmentation [3]

```
In [22]: def plot_img(img, operation, cmap="gray"):  
    fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(15, 10))  
    ax.imshow(img, cmap=cmap)  
    ax.set_title(operation)  
    ax.axis('off')  
  
    plt.show()
```

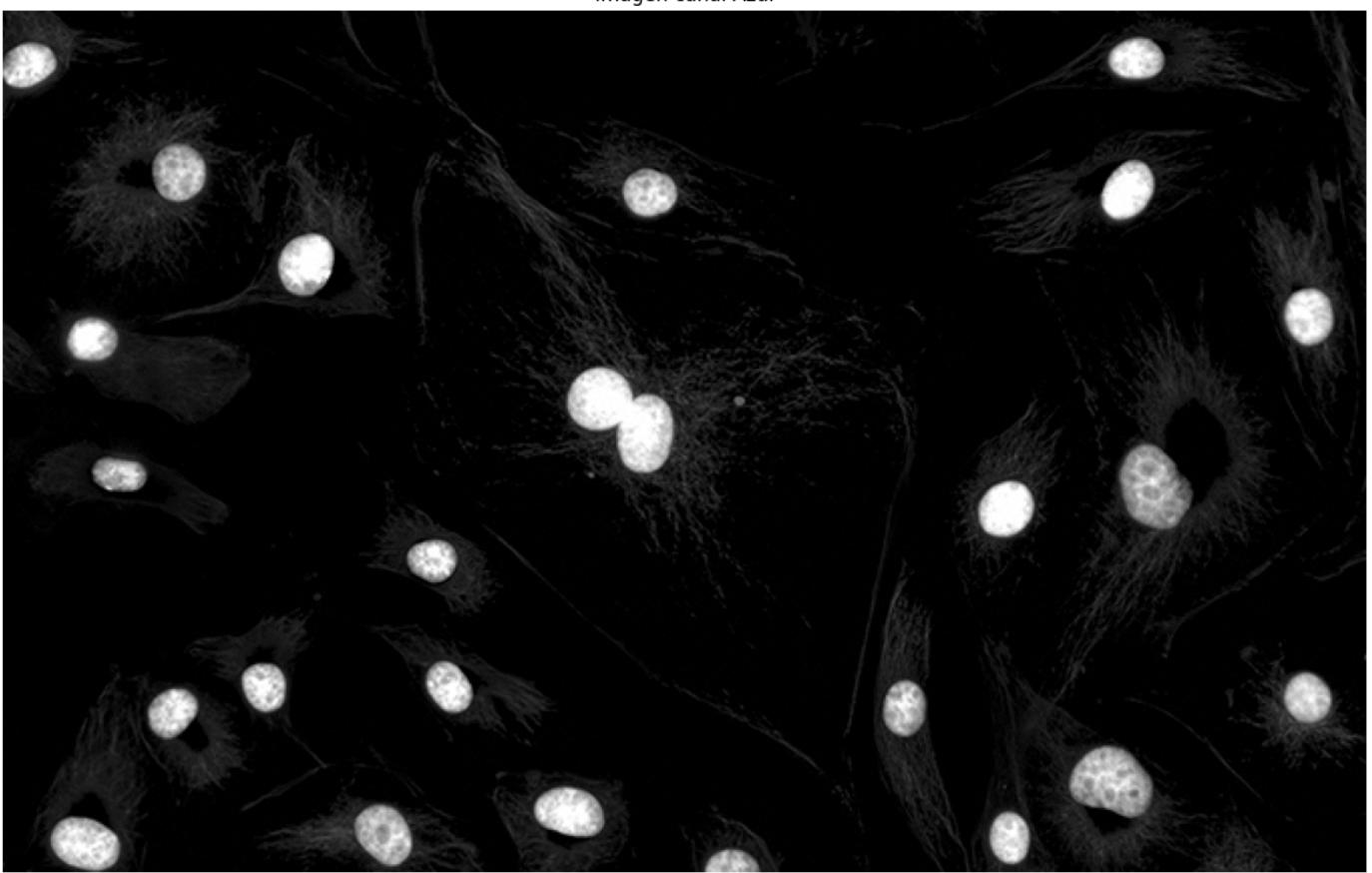
```
In [23]: # Empezamos leyendo nuestra imagen de microscopio en formato tif  
orig_img = cv2.imread("img/cell1.jpg")  
plot_img(orig_img, "Imagen Original")
```



Para poder llevar a cabo la segmentacion de estas imagenes, necesitamos extraer el azul ya que estas imagenes fueron obtenidas con una tinta azul especial para detectar las celulas cancerigenas.

```
In [31]: blue_img = orig_img[:, :, 0]  
plot_img(blue_img, "Imagen canal Azul")
```

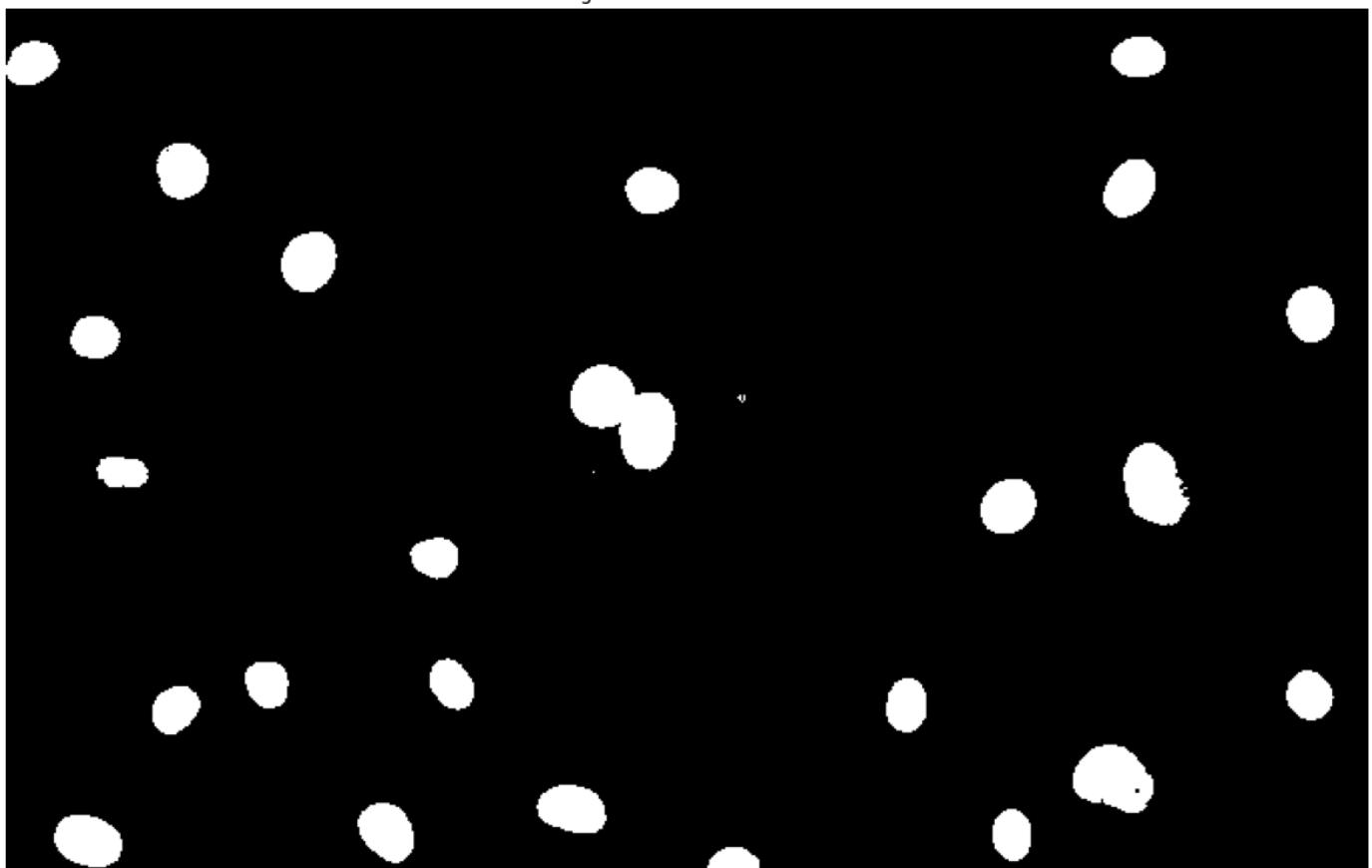
Imagen canal Azul



Una vez extraido el color azul es necesario establecer un umbral de intensidad.

Para este ejemplo, usaremos el metodo de OTSU para extraer una imagen binaria, es decir todos los pixeles umbralados seran puestos en 255 o 0

```
In [143]: ret1, thresh = cv2.threshold(blue_img, 0, 255, cv2.THRESH_BINARY+cv2.THRESH_OTSU)  
plot_img(thresh, "Imagen con umbralado OTSU")
```



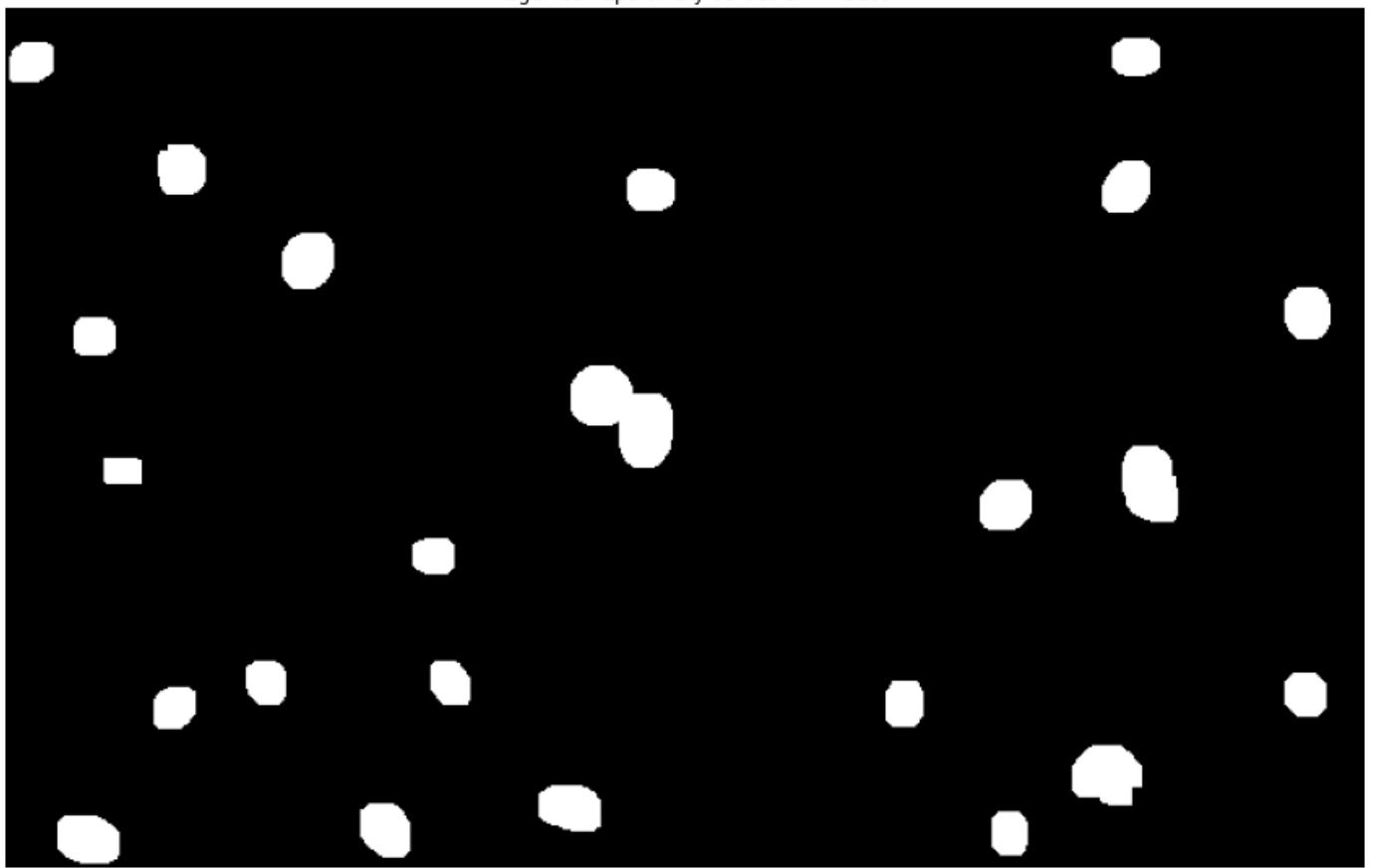
Un paso improtante despues de aplicar el umbralado y contar con una imagen binaria es el realizar la operacion morfologica de apertura.

Si recordamos de lo aprendido en el curso, la operacion de apertura es aplicar a una imagen erocionada (eliminar pixeles en los limites de objetos) una dilatacion (agregar pixeles a los limites de un objeto) usando el mismo tamano de kernel para ambas operaciones. Usando la funcion clear border en la imagen de apertura, eliminamos aquellos objetos que esten tocando el borde de la imagen.

```
In [32]: kernel = np.ones((3,3), np.uint8)
opening_img = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel, iterations = 5)

opening_img = clear_border(opening_img)

plot_img(opening_img, "Imagen con apertura y bordes eliminados")
```

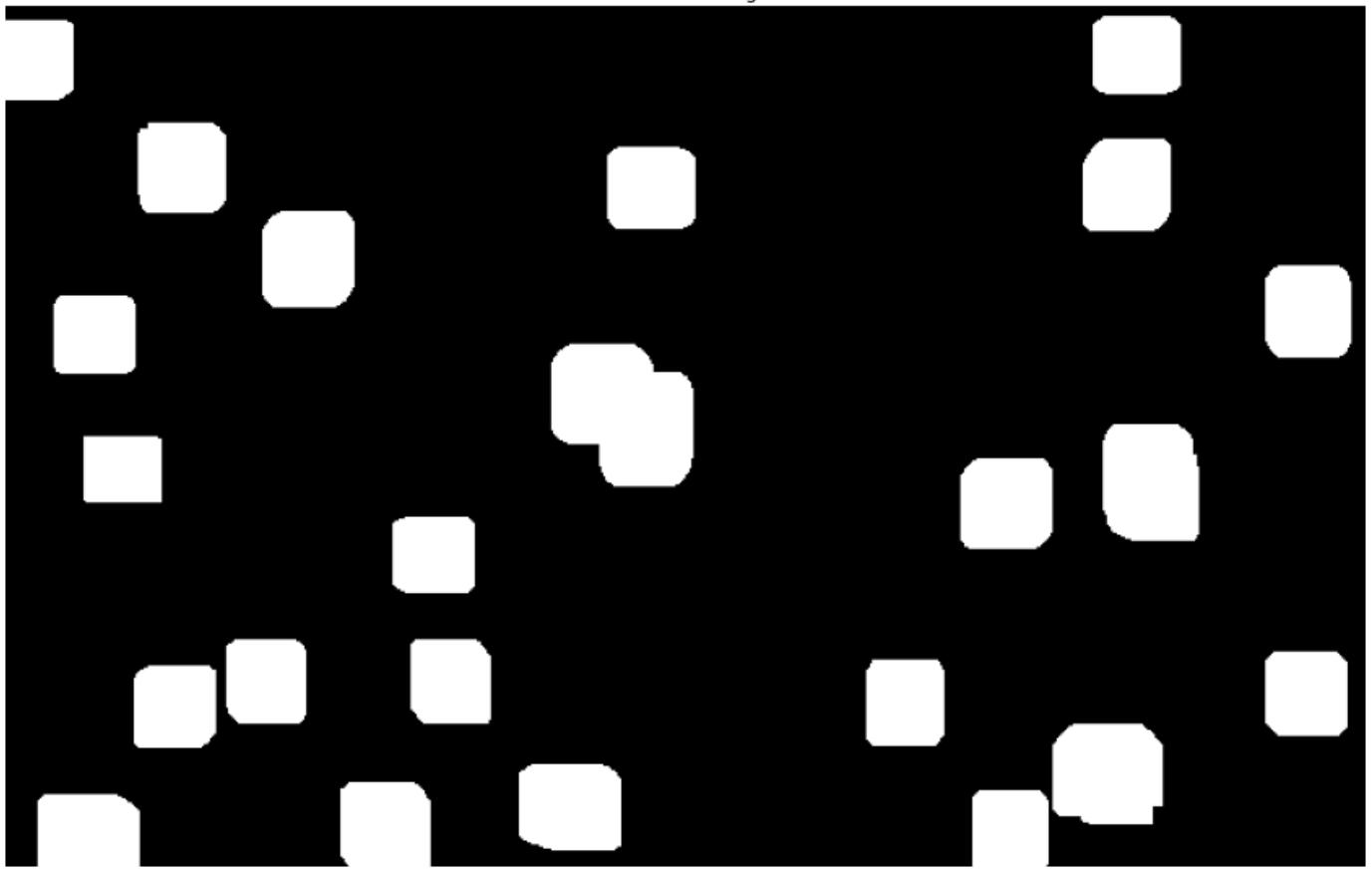


Comenzamos pues identificando el fondo de la imagen (background). Al dilatar los pixeles varias veces se incrementa los límites de la imagen con respecto al fondo. De esta manera lo que quede después de la dicha operación seguramente pertenece al fondo(background)

El área entre el fondo y el frente se conoce como el área ambigua o desconocida. El algoritmo de Watershed debe de ser capaz de encontrar esta área por nosotros.

Lo que se muestre en fondo negro es seguramente el fondo de la imagen, lo blanco no estamos seguros aun a que clase pertenece.

```
In [33]: sure_bg = cv2.dilate(opening_img,kernel,iterations=10)
plot_img(sure_bg, "Fondo de la imagen")
```

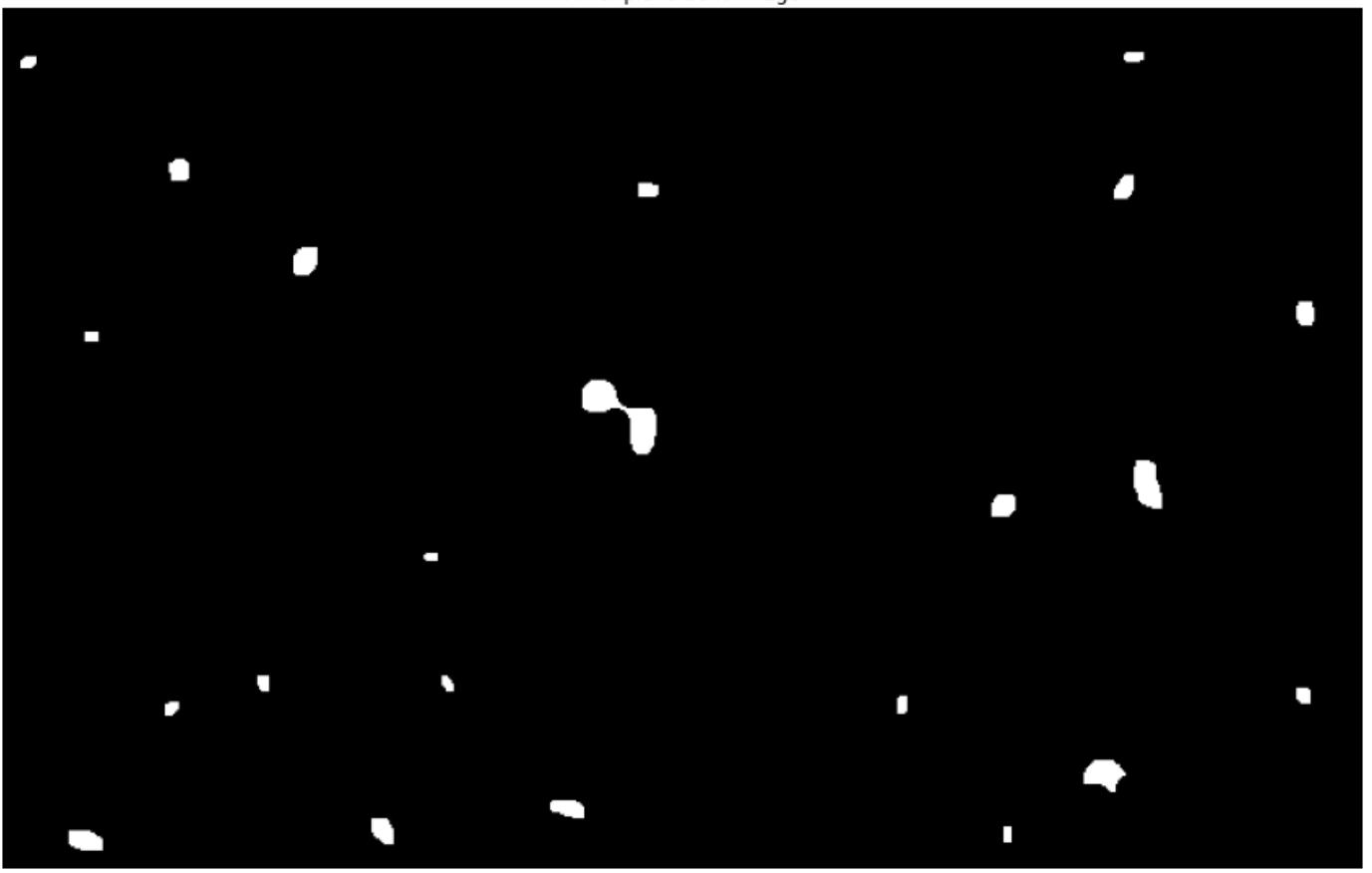


El siguiente paso es encontrar el primer plano de nuestra imagen, para hacer esto es necesario encontrar para un pixel dado cual es la distancia mas corta para el mas cercano a 0 (pixel negro o el fondo de la imagen). A esto se le conoce como calcular la distancia euclidianas de una imagen.

Las areas cercanas al fondo de la imagen tendran un valor de distancia bajo y las mas lejanas un valor alto. Es por esto que necesitamos adaptar el resultado de la distancia para poder tener un umbral de intensidad que permita la correcta separacion de los componentes de la imagen, es decir, es necesario encontrar el umbral ideal. Para este ejemplo decidimos comenzar con la mitad del valor maximo obtenido.

```
In [34]: dist_transform = cv2.distanceTransform(opening_img, cv2.DIST_L2, 5)
ret2, sure_fg = cv2.threshold(dist_transform, 0.5*dist_transform.max(), 255, 0)

plot_img(sure_fg, "Primer plano de la Imagen")
```

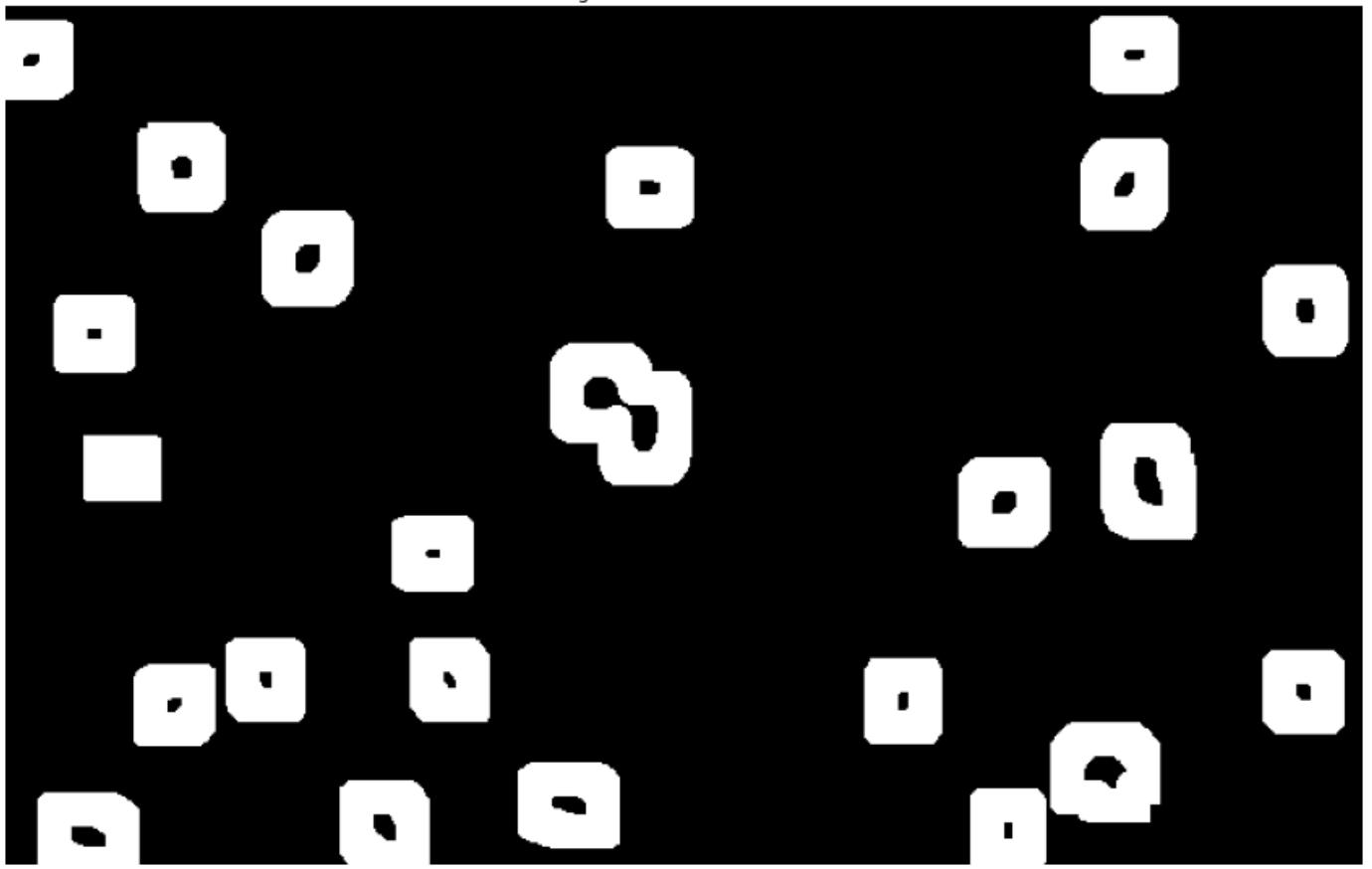


Ahora nos queda solamente definir nuestra area desconocida, la cual el algoritmo de Watershed se encargara de llenar. Es decir, tenemos que encontrar el area que no estamos seguros si es parte del fondo o del primer plano de la imagen.

Esta area la calculamos con una operacion de extraccion del primer plano y el fondo.

```
In [35]: sure_fg = np.uint8(sure_fg)
unknown_img = cv2.subtract(sure_bg,sure_fg)

plot_img(unknown_img, "Imagen del area desconocida")
```



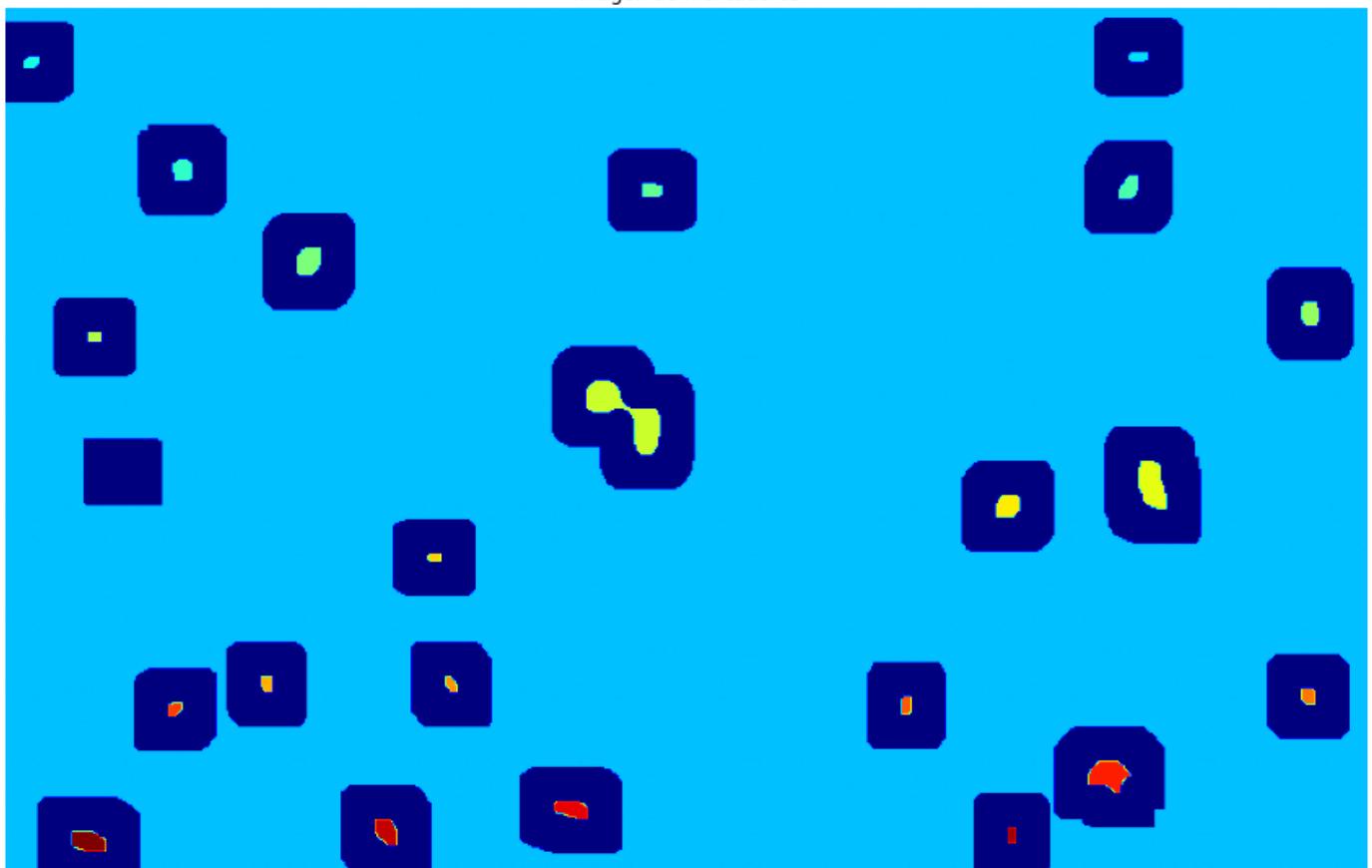
Es necesario crear marcadores y etiquetas para las regiones dentro de la imagen. Las áreas que tenemos certeza de a dónde pertenecen deben ser etiquetadas con valores numéricos positivos. Las regiones desconocidas con 0.

Un problema al usar la librería *connectedComponents* es que a todos los pixeles de fondo se les asigna el valor de 0. Eso podría causar que nuestro algoritmo de watershed considere estas regiones como desconocidas. Para evitarlo, agregamos un valor positivo a todas las etiquetas para asegurarnos que las regiones que son seguramente de fondo no son 0.

```
In [36]: ret3, markers = cv2.connectedComponents(sure_fg)

markers = markers+10

# Finalmente marcamos las regiones desconocidas con 0.
markers[unknown_img==255] = 0
plot_img(markers, "Imagen de marcadores", "jet")
```



Una vez que contamos con marcadores correctos, podemos aplicar el algoritmo de watershed usando la imagen original y nuestros marcadores obtenidos.

OpenCV asigna los límites que rodean a los segmentos de la imagen con valor de -1 después de aplicado el algoritmo, por lo que podemos colorear estos para identificarlos al mostrar la imagen de nuevo.

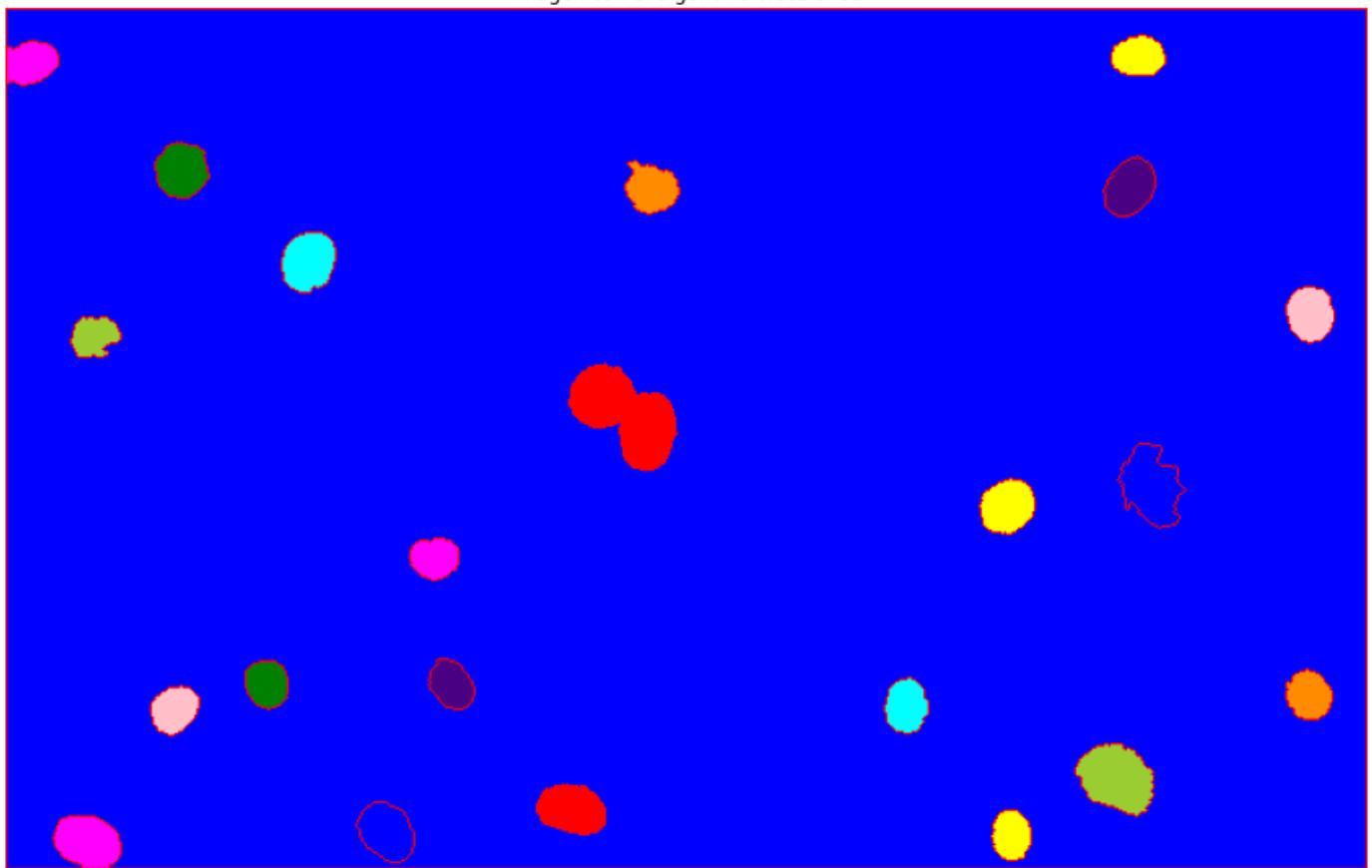
```
In [37]: markers = cv2.watershed(orig_img,markers)

orig_img[markers == -1] = [255,255,0]

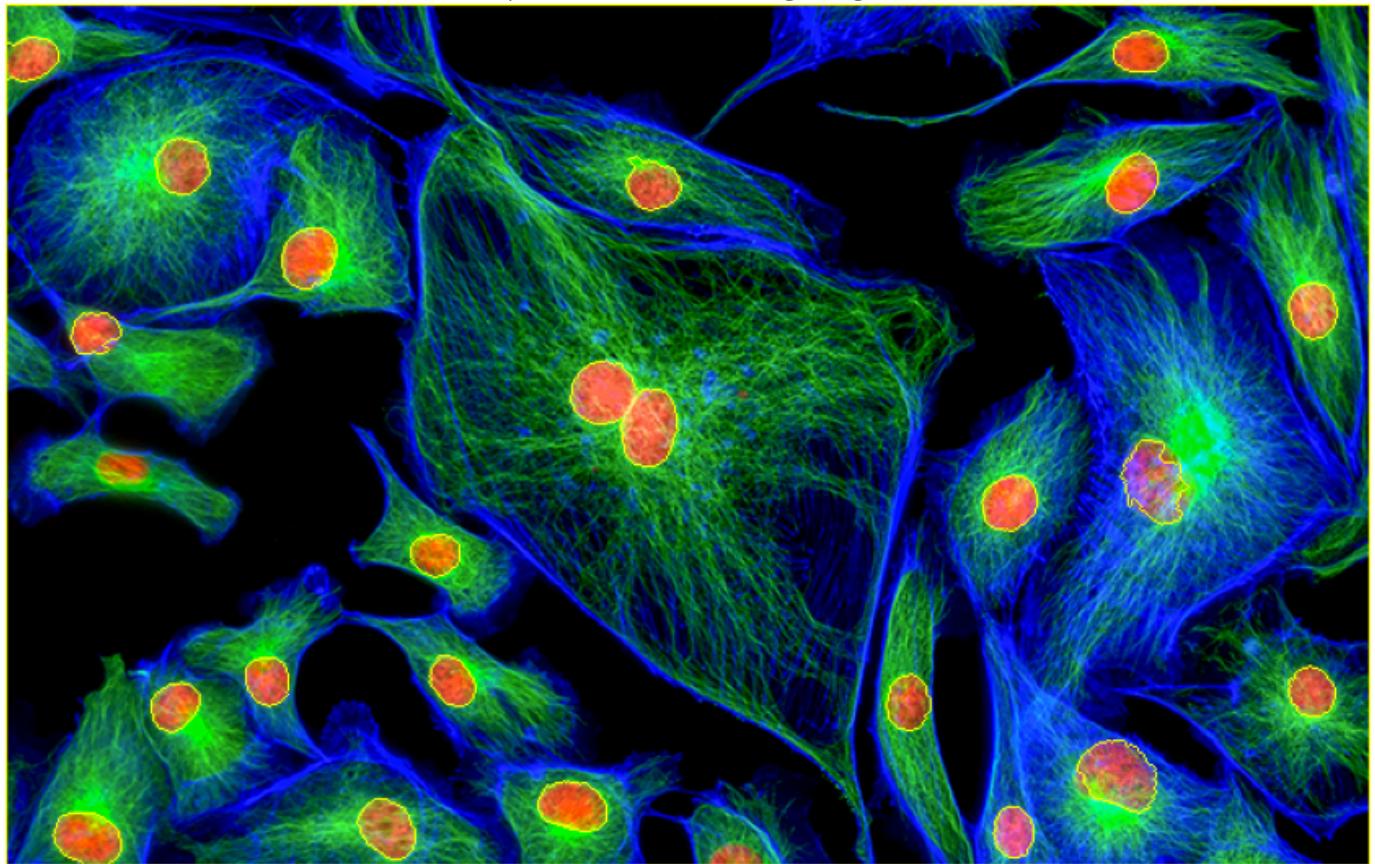
watershed_img = color.label2rgb(markers, bg_label=0)

plot_img(watershed_img, "Imagen con el algoritmo Watershed")
plot_img(orig_img, "Etiquetado de bordes en la imagen original")
```

Imagen con el algoritmo Watershed



Etiquetado de bordes en la imagen original



## REFERENCIAS

[0] 32\_Watersheds\_Segmentation. (s/f). Github.io. de [http://insightsoftwareconsortium.github.io/SimpleITK-Notebooks/Python\\_html/32\\_Watersheds\\_Segmentation.html](http://insightsoftwareconsortium.github.io/SimpleITK-Notebooks/Python_html/32_Watersheds_Segmentation.html)

[1]

[https://github.com/c1Esperanto/pyclesperanto\\_prototype/blob/master/demo/segmentation/voronoi\\_otsu\\_labeli](https://github.com/c1Esperanto/pyclesperanto_prototype/blob/master/demo/segmentation/voronoi_otsu_labeli)

[2] Dey, S. (2018). Hands-On Image Processing with Python: Expert techniques for advanced image analysis and effective interpretation of image data. Packt Publishing.

[3] DigitalSreeni [@DigitalSreeni]. (2019, junio 6). 35 - Cell Nuclei analysis in Python using watershed segmentation. Youtube. <https://www.youtube.com/watch?v=AsTvGxuiqKs>

In [ ]: