

Pregunta 6

November 20, 2019

1 Pregunta 6

(2 puntos) La imagen Ex3Preg6(a).tif muestra una imagen tomada con un microscopio de cultivo de bacterias identificadas por los círculos intensos:

- (0.5 puntos) Usando una técnica de umbralización global, segmente la imagen y muestre el resultado de la segmentación.
- (0.5 puntos) A la imagen original se le aplicó una umbralización con valores locales y al resultado se le realizó una apertura morfológica obteniendo la imagen Ex3Preg6(b).tif. Usando esta imagen, cuente y etiquete cuantos objetos de la segmentación pueden considerarse células independientes.
- (1 punto) Continuando con la imagen anterior. Cuente y etiquete cuantos objetos de la segmentación pueden considerarse 2 células agrupadas, y cuantos y cuales más de 2 células.

```
[1]: # Functional programming tools :
from functools import partial, reduce

# Visualisation :
import matplotlib.pyplot as plt
import matplotlib.image as pim
import matplotlib.patches as mpatches
import seaborn as sns

# Data tools :
import numpy as np
import pandas as pd

# Image processing :
import cv2 as cv
from skimage import data
from skimage.filters import threshold_otsu
from skimage.segmentation import clear_border
from skimage.measure import label, regionprops
from skimage.morphology import closing, square
from skimage.color import label2rgb

# Machine Learning :
```

```
from sklearn.cluster import KMeans

# Jupyter reimport utils :
import importlib
```

```
[2]: # Custom :
import mfilt_funcs as mfs
importlib.reload(mfs)
import mfilt_funcs as mfs

import utils
importlib.reload(utils)
import utils
```

```
[3]: #plt.style.available
```

```
[4]: plt.style.use('seaborn-deep')
plt.rcParams['figure.figsize'] = (10, 5)
```

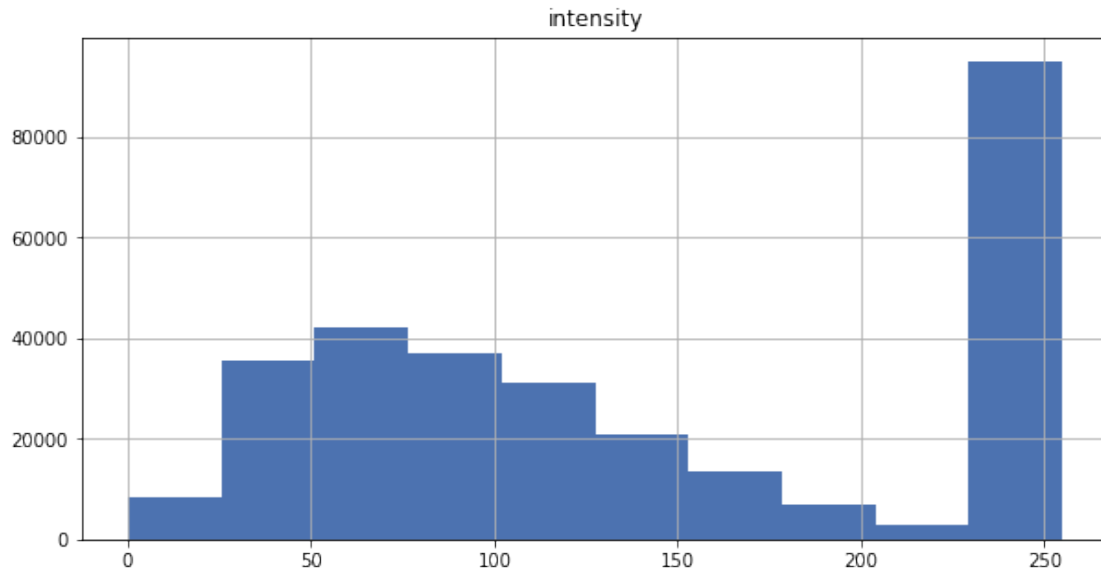
```
[6]: img = cv.imread('imagenes/Ex3Preg6(a).tif', cv.IMREAD_GRAYSCALE)
color = cv.cvtColor(img, cv.COLOR_GRAY2RGB) # Color copy, to draw colored circles
```

1.1 a. (0.5 puntos) Usando una técnica de umbralización global, segmente la imagen y muestre el resultado de la segmentación.

```
[7]: intensities = pd.core.frame.DataFrame(dict(intensity=img.flatten()))
```

```
[8]: intensities.hist()
```

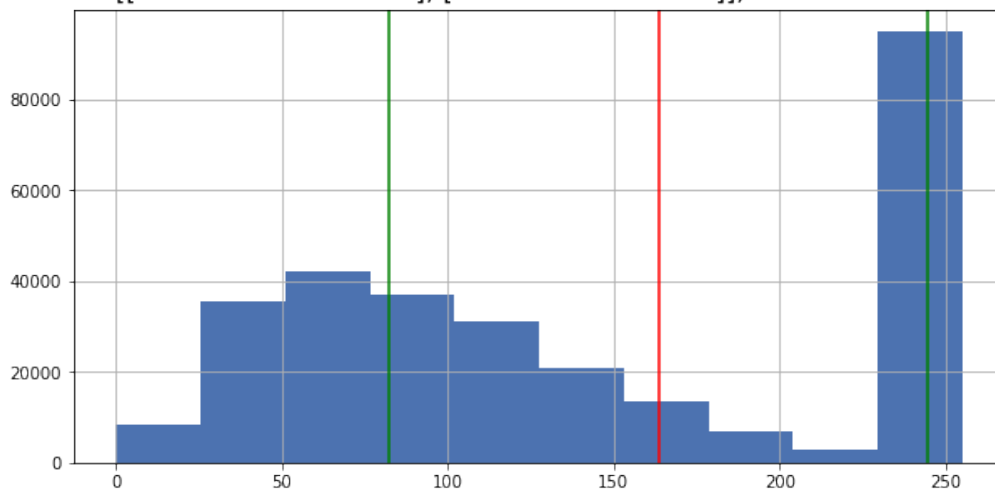
```
[8]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x1c252e0c90>]],
          dtype=object)
```



```
[9]: kmeans = KMeans(n_clusters=2, random_state=0, verbose=False).fit(intensities)
      K = kmeans.cluster_centers_.mean()
```

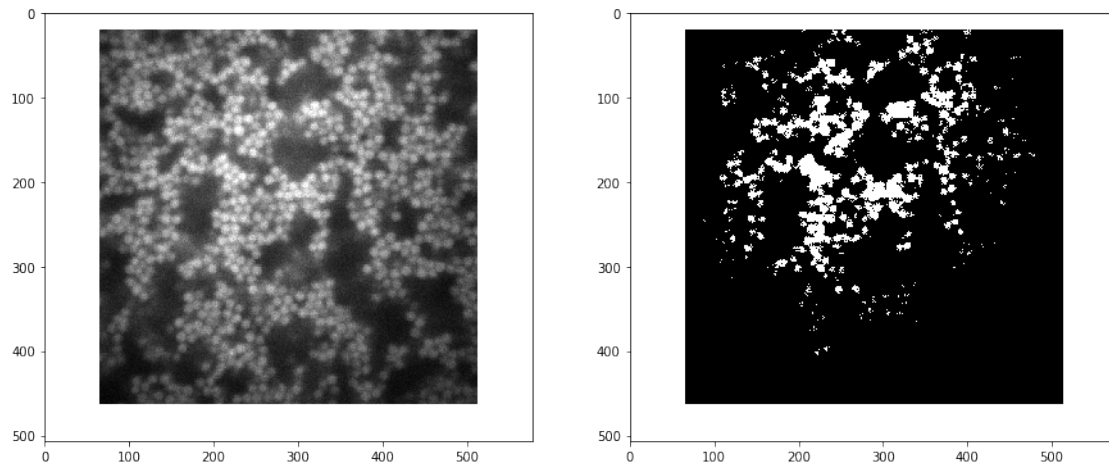
```
[10]: intensities.hist()
      plt.axvline(K, color='r')
      list(map(lambda x: plt.axvline(x, color='g'), kmeans.cluster_centers_))
      _ = plt.title(f"Means = {kmeans.cluster_centers_.tolist()}, K = {K}", size=16)
```

Means = [[244.6166321877507], [82.3559870548567]], K = 163.4863096213037



```
[11]: thresh1 = cv.threshold(img, K, 255, cv.THRESH_BINARY)[1]
```

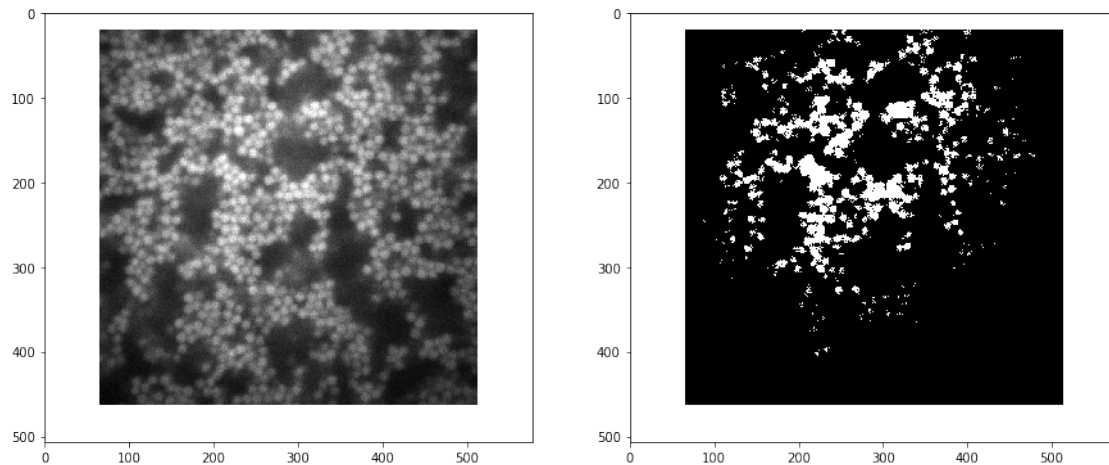
```
[12]: utils.side_by_side(img, thresh1)
```



Como podemos ver, una técnica de umbralización estándar como k-medias móviles, con dos medias, da resultados muy pobres.

```
[13]: otsu1 = cv.threshold(img,0,255,cv.THRESH_BINARY+cv.THRESH_OTSU)[1]
```

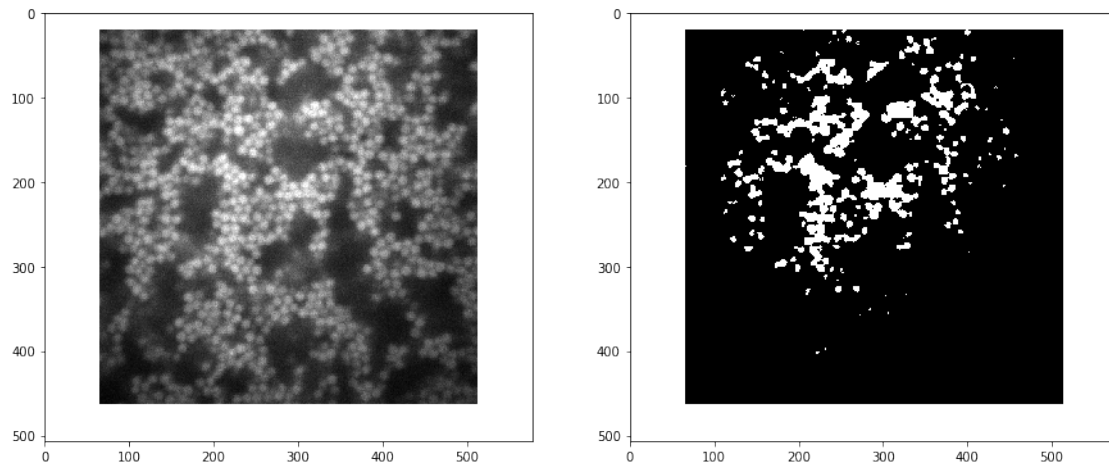
```
[14]: utils.side_by_side(img, otsu1)
```



El algoritmo de Otsu no logra mejorar mucho la segmentación (esto era de esperarse dado que el histograma original era claramente bimodal).

```
[15]: gblur = cv.GaussianBlur(img,(3,3),0)
otsu2 = cv.threshold(gblur,0,255,cv.THRESH_BINARY+cv.THRESH_OTSU)[1]
```

```
[16]: utils.side_by_side(img, otsu2)
```



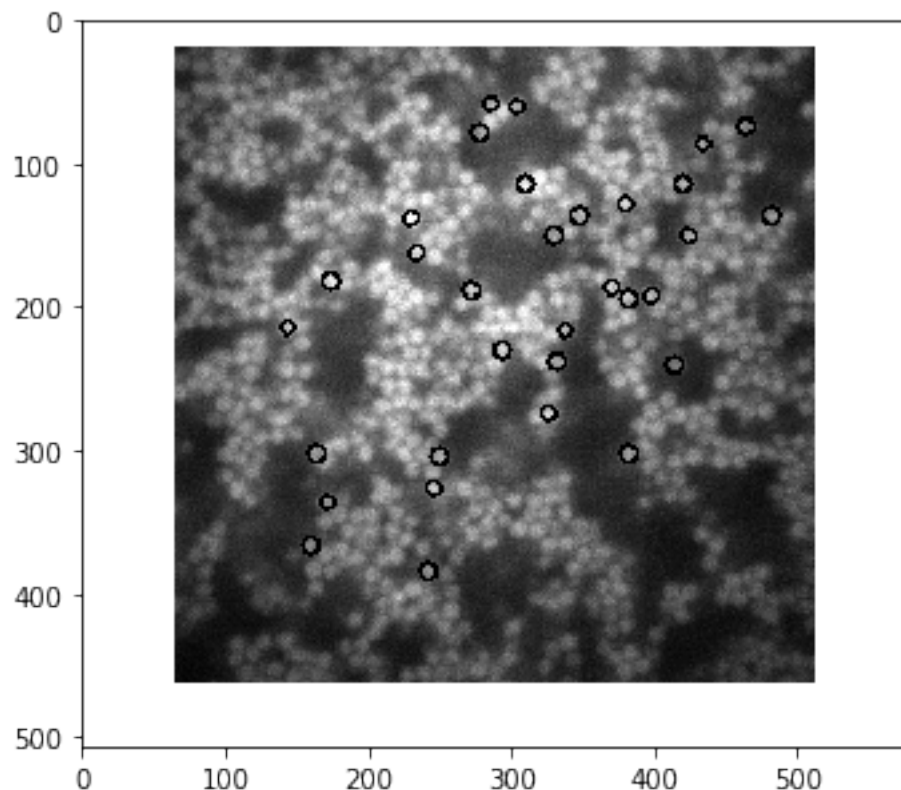
El algoritmo de Otsu no logra mejorar mucho la segmentación aún en combinación con un suavizado Gaussiano.

```
[17]: img_blur = cv.medianBlur(img, 5)
      circles = cv.HoughCircles(img_blur, cv.HOUGH_GRADIENT, 1, img.shape[0]/64,
      ↪ param1=200, param2=10, minRadius=5, maxRadius=7)
```

```
[18]: if circles is not None:
      circles = np.uint16(np.around(circles))
      for i in circles[0, :]:
          cv.circle(img, (i[0], i[1]), i[2], (0, 0, 255), 2)
```

```
[19]: plt.imshow(img, cmap='gray')
```

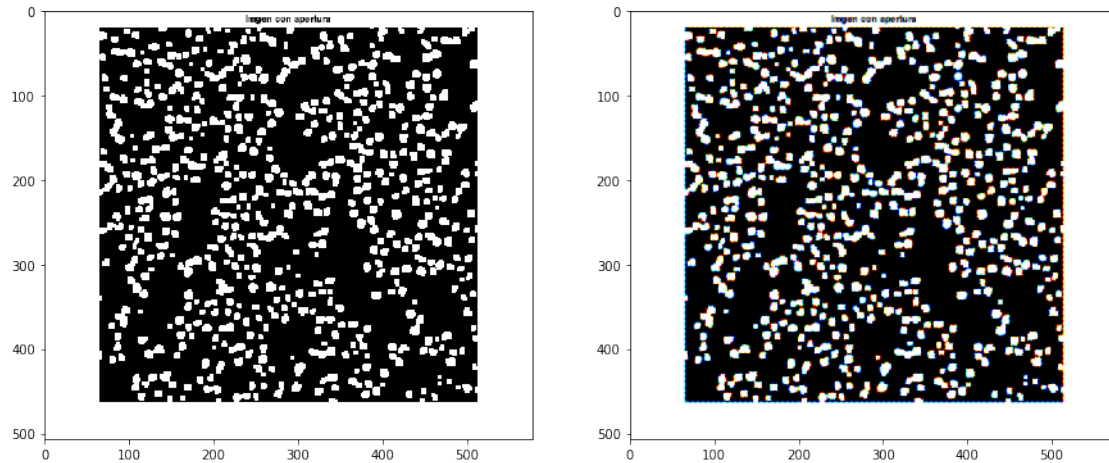
```
[19]: <matplotlib.image.AxesImage at 0x1c26f285d0>
```



Al ver que las células tenían una apariencia más o menos circular, parecía una buena idea usar una transformada de Hough para buscar los círculos de la imagen, pero esto entregó resultados muy pobres. Tal vez esto podría funcionar con la imagen binaria.

1.2 b. (0.5 puntos) A la imagen original se le aplicó una umbralización con valores locales y al resultado se le realizó una apertura morfológica obteniendo la imagen `Ex3Preg6(b).tif`. Usando esta imagen, cuente y etiquete cuantos objetos de la segmentación pueden considerarse células independientes.

```
[20]: imgb = cv.imread('imagenes/Ex3Preg6(b).tif', cv.IMREAD_GRAYSCALE)
      imgbc = cv.cvtColor(imgb, cv.COLOR_BAYER_GB2RGB)
      utils.side_by_side(imgb, imgbc)
```



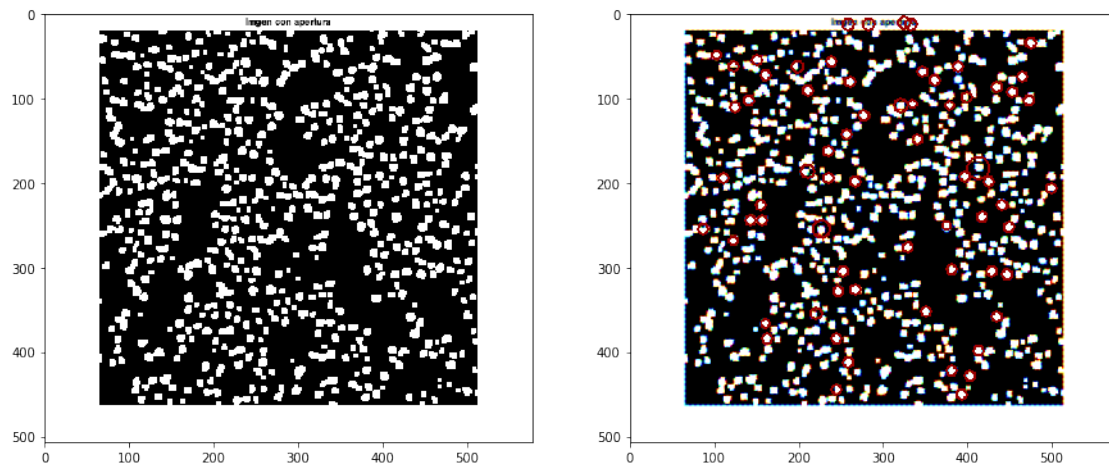
1.2.1 Primera aproximación :

El uso de la transformada de Hough para círculos, no para líneas. Al ver la imagen, uno podría pensar que un círculo es una buena aproximación de la forma de una célula, por lo tanto los círculos encontrados por una transformada de Hough serían las células que buscamos identificar, caracterizar y contabilizar

```
[21]: circles = cv.HoughCircles(imgb, cv.HOUGH_GRADIENT, 1, img.shape[0]/64,
    ↪ param1=200, param2=10, minRadius=5, maxRadius=15)
```

```
[22]: if circles is not None:
    circles = np.uint16(np.around(circles))
    for i in circles[0, :]:
        cv.circle(imgbc, (i[0], i[1]), i[2], (155, 0, 0), 2)
```

```
[23]: utils.side_by_side(imgb, imgbc)
```

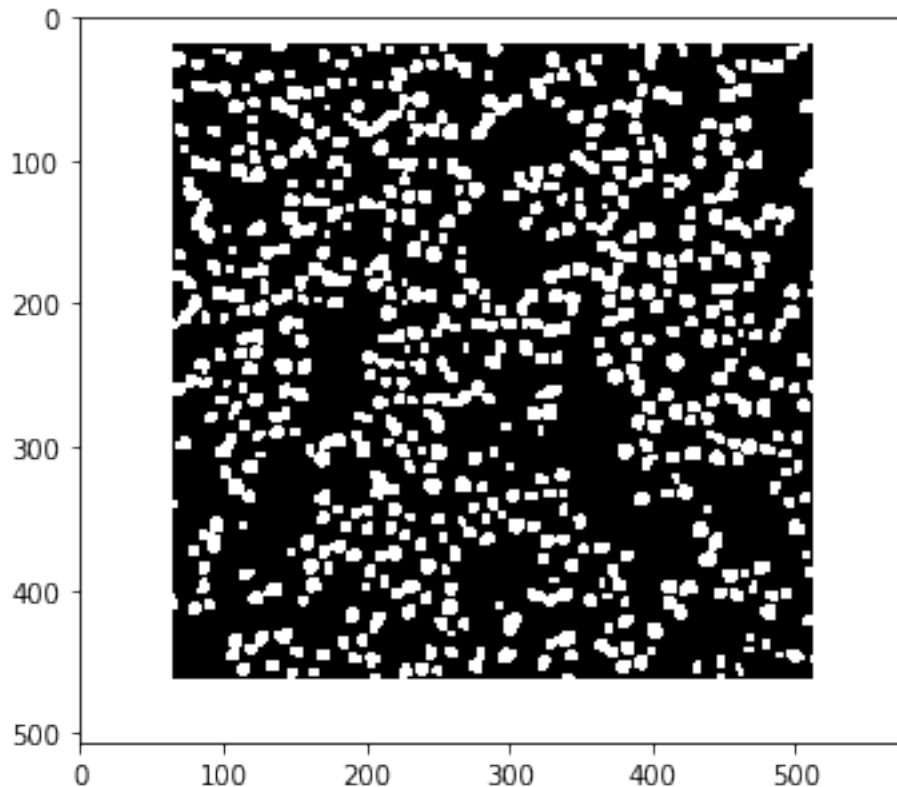


Aquí podemos ver que aunque la imagen principal carece de falsos positivos (es decir todos los círculos dibujados dentro de la región útil de la imagen contienen una célula) el **número de falsos negativos es altísimo** : sólo una pequeña parte de las células observadas fueron identificadas por `cv.HoughCircles()`.

Esto nos indica que tal vez las células no se asemejan tanto a un círculo. Por esta razón, no se explorará más a fondo esta vía de acción. Cabe mencionar que la transformada encuentra círculos en el texto de encabezado : **Imagen con apertura**. Por esta razón, en adelante se trabajará con otra imagen recortada a mano para excluir este texto que podría causar problemas en la segmentación más adelante.

```
[28]: imgb2 = cv.imread('imagenes/Ex3Preg6(b)3.tif', cv.IMREAD_GRAYSCALE)
plt.imshow(imgb2, cmap='gray')
```

```
[28]: <matplotlib.image.AxesImage at 0x1c274d78d0>
```



```
[34]: sns.distplot(imgb2.flatten())
```

```
[34]: <matplotlib.axes._subplots.AxesSubplot at 0x1c28096bd0>
```

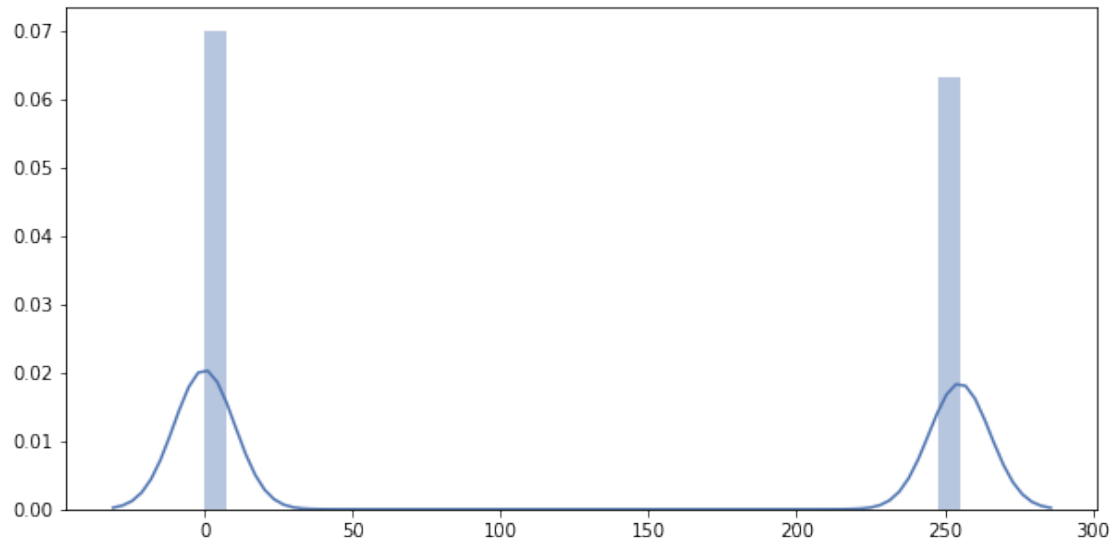
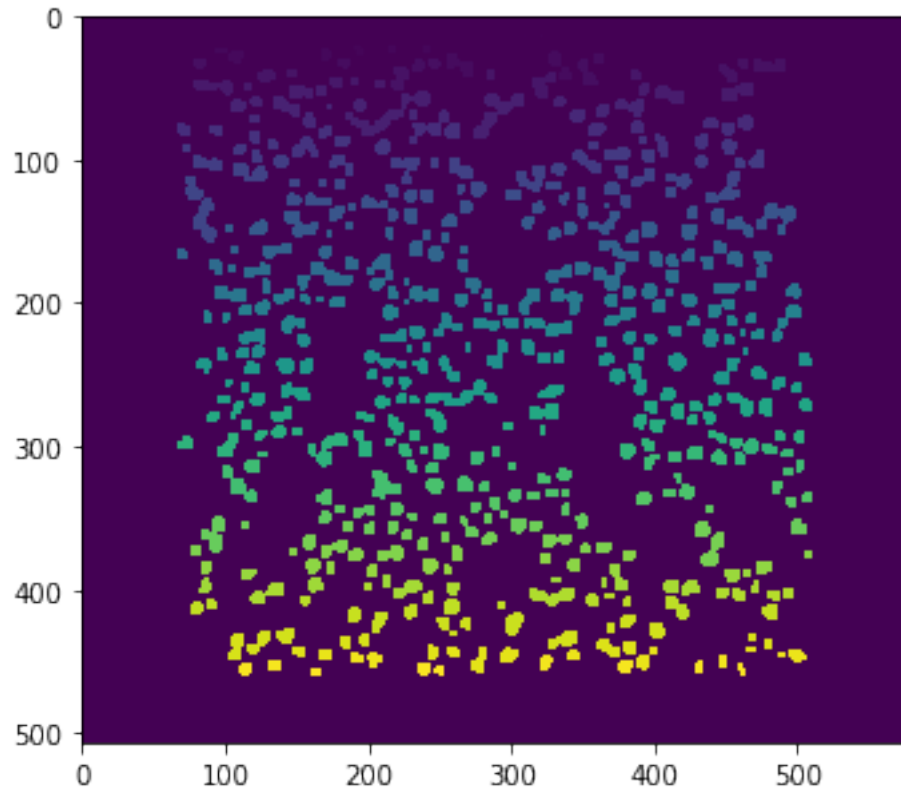



Imagen claramente binaria.

```
[29]: label_image, n_objs = label(imgb2, return_num=True)
      plt.imshow(label_image)
      print(n_objs)
```

475

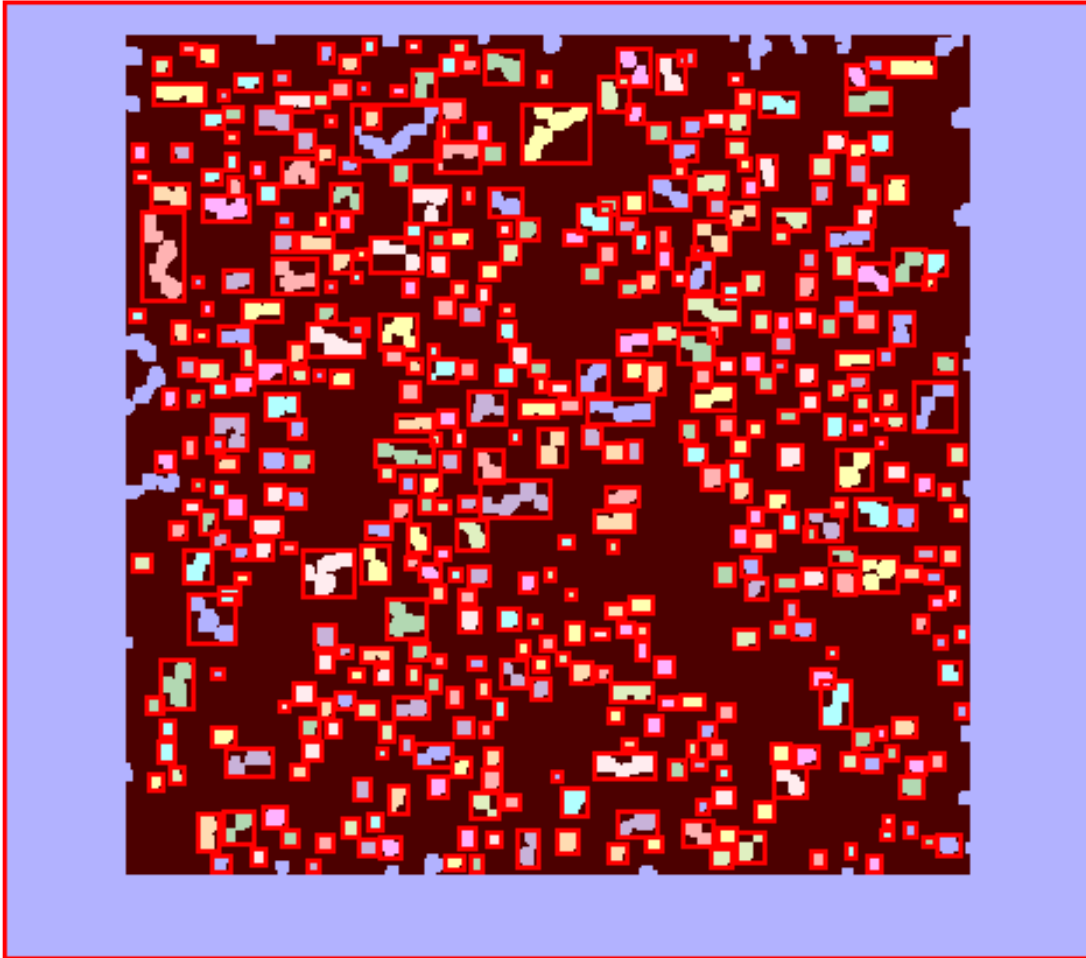


```
[35]: image_label_overlay = label2rgb(label_image, image=imgb2)

fig, ax = plt.subplots(figsize=(10, 6))
ax.imshow(image_label_overlay)

for region in regionprops(label_image):
    # take regions with large enough areas
    if region.area >= 10:
        # draw rectangle around segmented coins
        minr, minc, maxr, maxc = region.bbox
        rect = mpatches.Rectangle((minc, minr), maxc - minc, maxr - minr,
                                  fill=False, edgecolor='red', linewidth=2)
        ax.add_patch(rect)

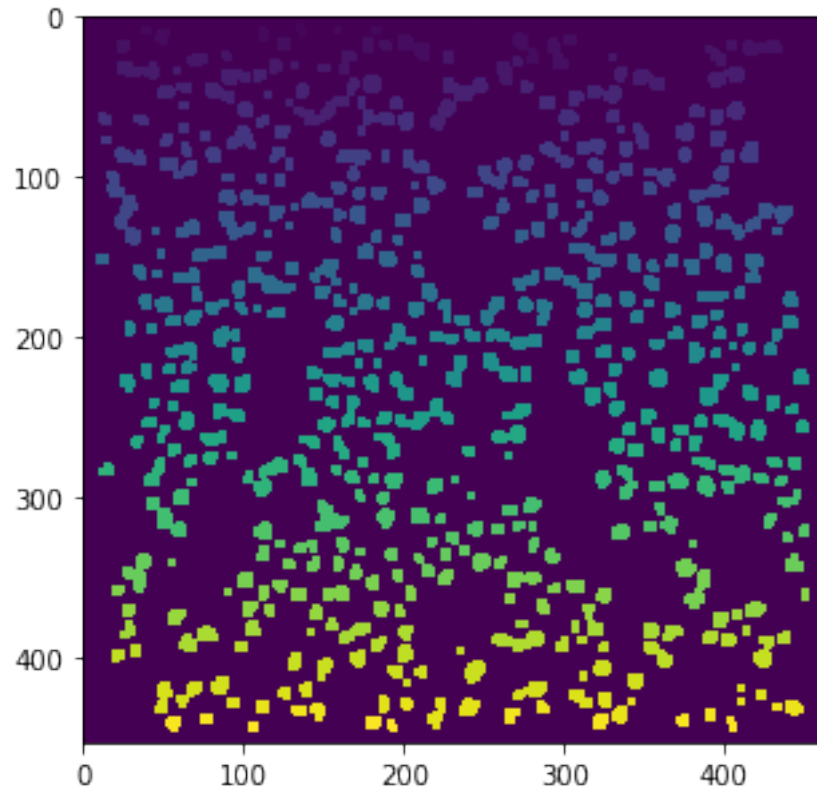
ax.set_axis_off()
plt.tight_layout()
plt.show()
```



2 Extra

```
[26]: label_image, n_objs = label(imgb2, return_num=True)
      plt.imshow(label_image)
      print(n_objs)
```

477

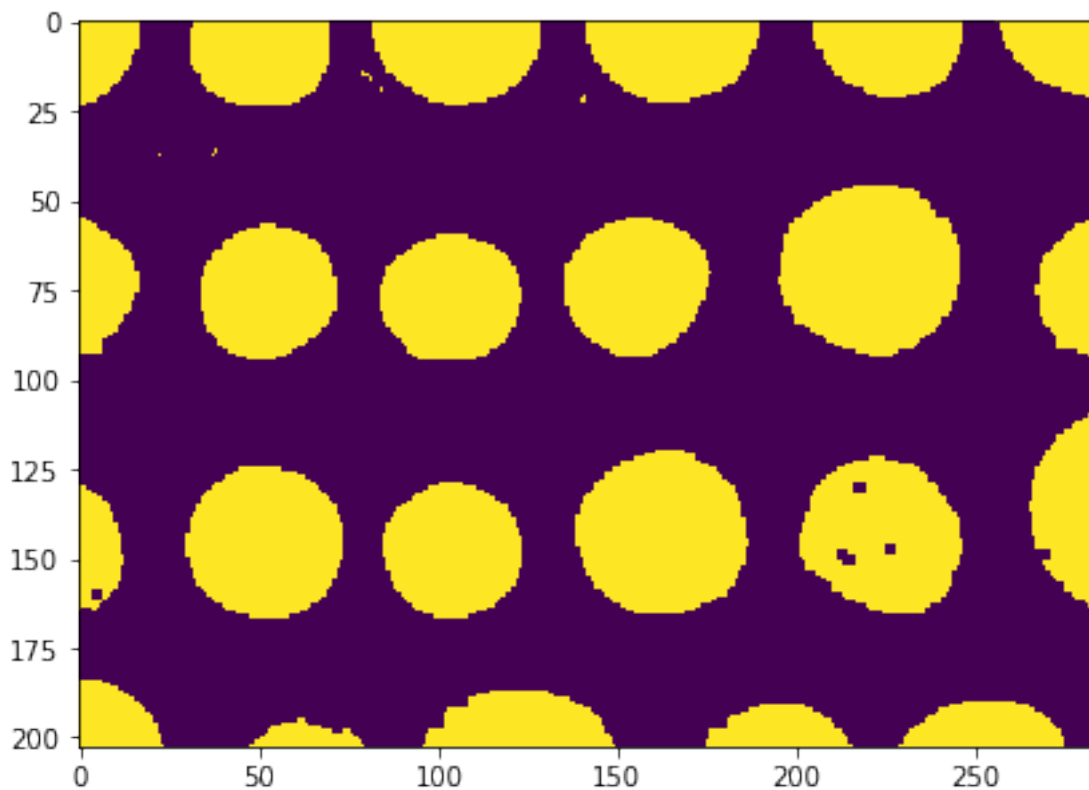


```
[ ]: cleared = clear_border()  
plt.imshow(cleared)
```

3 Scikit-Image example :

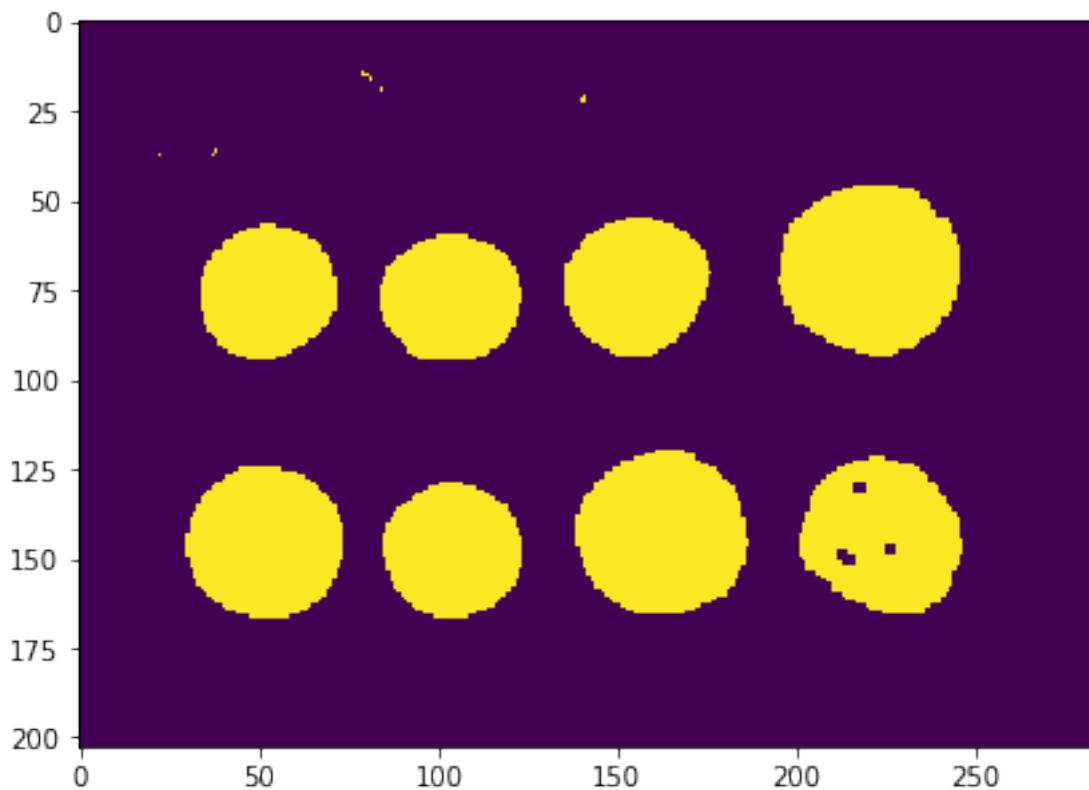
```
[151]: image = data.coins()[50:-50, 50:-50]  
  
# apply threshold  
thresh = threshold_otsu(image)  
bw = closing(image > thresh, square(3))  
plt.imshow(bw)
```

```
[151]: <matplotlib.image.AxesImage at 0x1c2ff27b90>
```



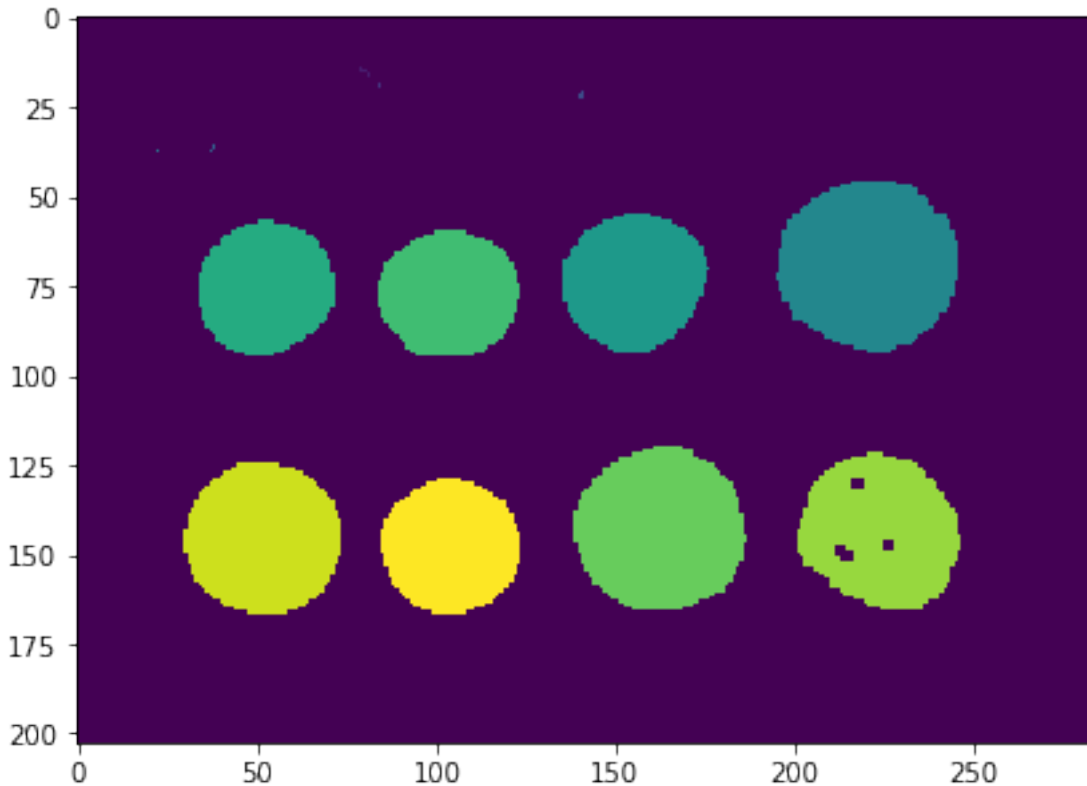
```
[152]: # remove artifacts connected to image border  
cleared = clear_border(bw)  
plt.imshow(cleared)
```

```
[152]: <matplotlib.image.AxesImage at 0x1c32f9a690>
```



```
[154]: # label image regions
label_image = label(cleared)
plt.imshow(label_image)
```

```
[154]: <matplotlib.image.AxesImage at 0x1c31283190>
```

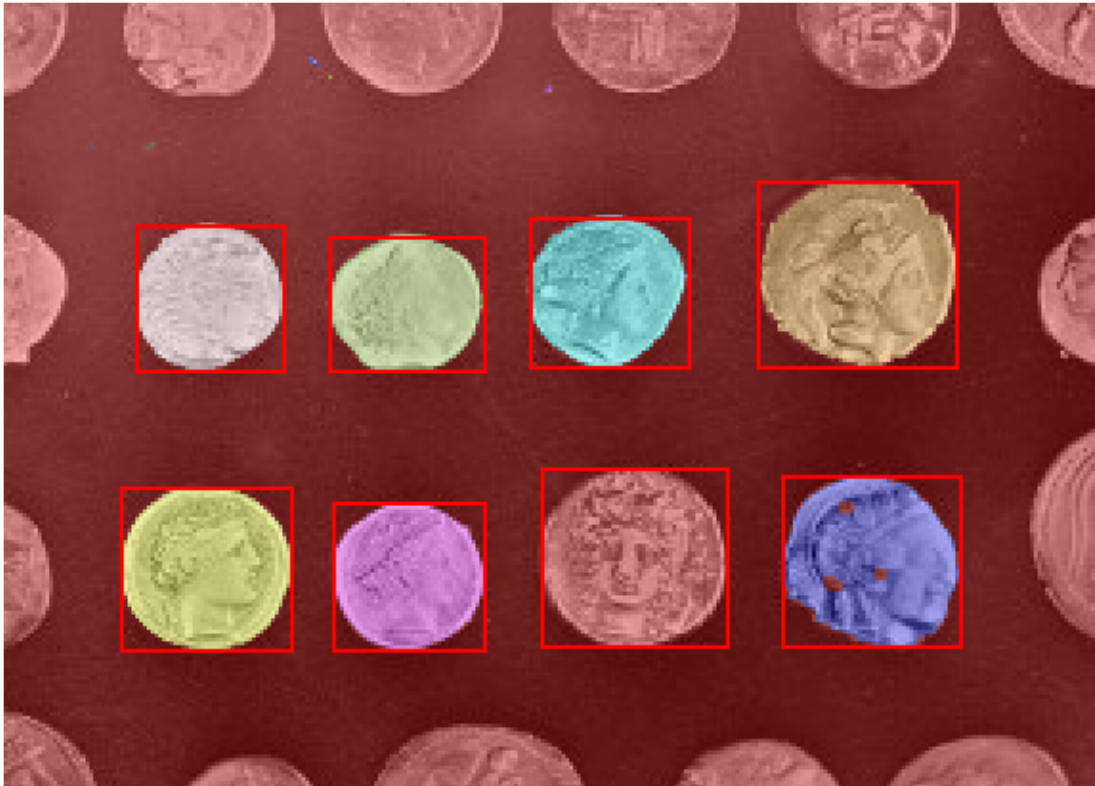


```
[155]: image_label_overlay = label2rgb(label_image, image=image)

fig, ax = plt.subplots(figsize=(10, 6))
ax.imshow(image_label_overlay)

for region in regionprops(label_image):
    # take regions with large enough areas
    if region.area >= 100:
        # draw rectangle around segmented coins
        minr, minc, maxr, maxc = region.bbox
        rect = mpatches.Rectangle((minc, minr), maxc - minc, maxr - minr,
                                  fill=False, edgecolor='red', linewidth=2)
        ax.add_patch(rect)

ax.set_axis_off()
plt.tight_layout()
plt.show()
```



```
[147]: #help(cv2.HoughCircles)
```

```
[148]: #help(cv2.cvtColor)
```

```
[158]: help(label)
```

Help on function label in module skimage.measure._label:

```
label(input, neighbors=None, background=None, return_num=False,
connectivity=None)
```

Label connected regions of an integer array.

Two pixels are connected when they are neighbors and have the same value. In 2D, they can be neighbors either in a 1- or 2-connected sense. The value refers to the maximum number of orthogonal hops to consider a pixel/voxel a neighbor::

1-connectivity	2-connectivity	diagonal connection close-up
<pre> [] []--[x]--[] </pre>	<pre> [] [] [] \ / []--[x]--[] </pre>	<pre> [] <- hop 2 [x]--[] </pre>

	/		\	hop 1
[]	[]	[]	[]	

Parameters

`input` : ndarray of dtype int

Image to label.

`neighbors` : {4, 8}, int, optional

Whether to use 4- or 8-"connectivity".

In 3D, 4-"connectivity" means connected pixels have to share face, whereas with 8-"connectivity", they have to share only edge or vertex.

****Deprecated, use** ``connectivity`` **instead.****

`background` : int, optional

Consider all pixels with this value as background pixels, and label them as 0. By default, 0-valued pixels are considered as background pixels.

`return_num` : bool, optional

Whether to return the number of assigned labels.

`connectivity` : int, optional

Maximum number of orthogonal hops to consider a pixel/voxel as a neighbor.

Accepted values are ranging from 1 to `input.ndim`. If ``None``, a full connectivity of ``input.ndim`` is used.

Returns

`labels` : ndarray of dtype int

Labeled array, where all connected regions are assigned the same integer value.

`num` : int, optional

Number of labels, which equals the maximum label index and is only returned if `return_num` is `True`.

See Also

`regionprops`

References

- .. [1] Christophe Fiorio and Jens Gustedt, "Two linear time Union-Find strategies for image processing", Theoretical Computer Science 154 (1996), pp. 165-181.
- .. [2] Kensheng Wu, Ekow Otoo and Arie Shoshani, "Optimizing connected component labeling algorithms", Paper LBNL-56864, 2005, Lawrence Berkeley National Laboratory (University of California), <http://repositories.cdlib.org/lbnl/LBNL-56864>

Examples

```

-----
>>> import numpy as np
>>> x = np.eye(3).astype(int)
>>> print(x)
[[1 0 0]
 [0 1 0]
 [0 0 1]]
>>> print(label(x, connectivity=1))
[[1 0 0]
 [0 2 0]
 [0 0 3]]
>>> print(label(x, connectivity=2))
[[1 0 0]
 [0 1 0]
 [0 0 1]]
>>> print(label(x, background=-1))
[[1 2 2]
 [2 1 2]
 [2 2 1]]
>>> x = np.array([[1, 0, 0],
...               [1, 1, 5],
...               [0, 0, 0]])
>>> print(label(x))
[[1 0 0]
 [1 1 2]
 [0 0 0]]

```

[]: