Universidad Nacional de San Antonio Abad del Cusco
Departamento Académico de Ing. Informática
VISION COMPUTACIONAL
Práctica Nº 08

DETECCIÓN DE CONTORNOS

Iván C. Medrano Valencia

1. OBJETIVO.

- Detectar contornos en una imagen
- Utilizar las funciones de OpenCV para detectar contornos.

2. INTRODUCCIÓN.

Un contorno es el límite de un objeto: es una forma cerrada a lo largo de la parte de una imagen que tiene el mismo color o intensidad.

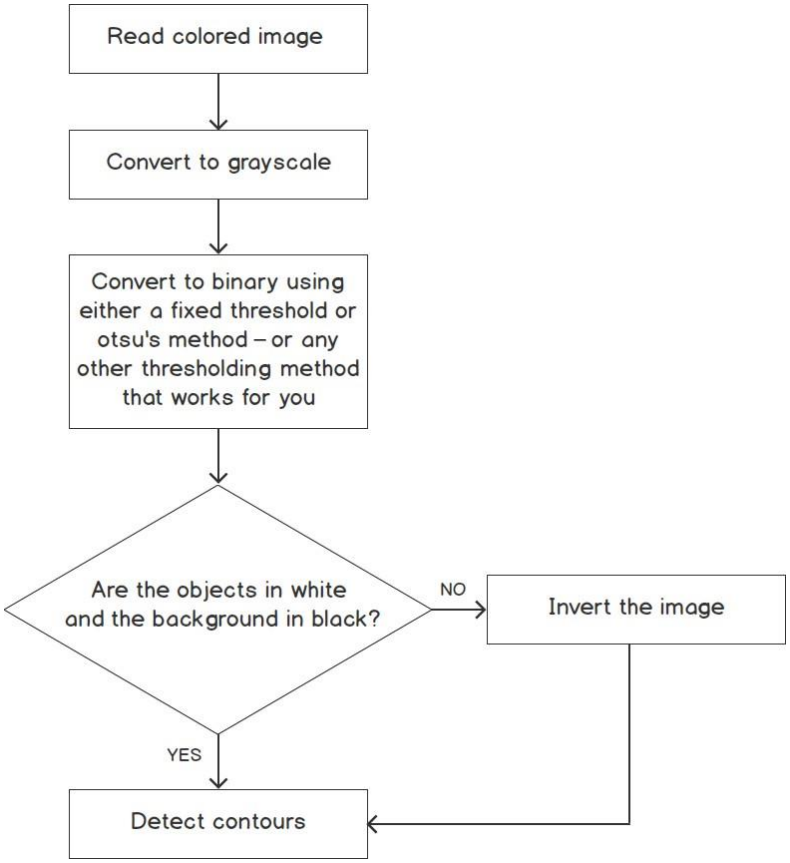
Hablando en términos de OpenCV, un contorno es el límite de un grupo de píxeles blancos sobre un fondo negro. Sí, en OpenCV, los contornos solo se pueden extraer de imágenes binarias.

En términos prácticos, los contornos pueden ayudar a contar la cantidad de objetos en una imagen. También se puede usar contornos para identificar su(s) objeto(s) de interés en una imagen dada, por ejemplo, para detectar una canasta de baloncesto en una imagen. Además, la detección de contornos se puede usar para identificar objetos con una relación de altura o ancho particular.

3. CONTORNOS: DETECCIÓN Y TRAZADO BÁSICOS

Un contorno sólo se puede detectar en una imagen binaria usando OpenCV. Para detectar contornos en imágenes en color (BGR) o en escala de grises, primero deberá convertirlas a binarias.

Para encontrar los contornos en una imagen en color, primero deberá convertirla a escala de grises. Después de eso, lo segmentará para convertirlo a binario. Esta segmentación se puede realizar con un umbral basado en un valor de escala de grises fijo de su elección o mediante el método de Otsu (o cualquier otro método que se adapte a sus datos). A continuación, se muestra un diagrama de flujo de detección de contornos:



El comando para detectar contornos en una imagen es el siguiente: `contours, hierarchy = cv2.findContours(source_image, retrieval_mode, approx_method)`

Hay dos salidas de esta función:

- **contours** es una lista que contiene todos los contornos detectados en la imagen de origen.
- **hierarchy** es una variable que le indica la relación entre los contornos. Por ejemplo, ¿un contorno está encerrado dentro de otro? En caso afirmativo, ¿ese contorno más grande se encuentra dentro de un contorno aún más grande?

Aquí, una breve explicación de las entradas de esta función:

- **source_image** es su imagen binaria de entrada. Para detectar contornos, el fondo de esta imagen binaria debe ser negro. Los contornos se extraen de manchas blancas.
- **retrieval_mode** es una bandera que instruye a la función cv2.findContours sobre cómo buscar todos los contornos. (¿Deberían obtenerse todos los contornos de forma independiente?). Si un contorno se encuentra dentro de otro contorno más grande, ¿debería devolverse esa información?

Si muchos contornos se encuentran dentro de un contorno más grande, ¿deberían devolverse los contornos internos o solo el contorno externo será suficiente?.

Si solo desea obtener todos los contornos exteriores extremos, debe mantenerlo en **cv2.RETR_EXTERNAL.** Esto ignorará todos los contornos que se encuentran dentro de otros contornos.

Si simplemente desea recuperar todos los contornos de forma independiente y no le importa cuál se encuentra dentro de cuál, entonces **cv2.RETR_LIST** sería su opción preferida. Esto no crea relaciones entre padres e hijos, por lo que todos los contornos están en el mismo nivel jerárquico.

Si está interesado en encontrar los límites externos e internos de los objetos (solo dos niveles: el borde de la forma externa y el límite del agujero interno), entonces buscaría **cv2.RETR_CCOMP**.

Si desea crear un árbol genealógico detallado que cubra todas las generaciones (padre, hijo, nieto y bisnieto), debe establecer esta entrada como **cv2.RETR_TREE**.

- **approx_method** es una bandera que le dice a esta función cómo almacenar los puntos límite de un contorno detectado. ¿Debería guardarse cada punto de coordenadas? Si es así, mantenga el valor de esta bandera como **cv2.CHAIN_APPROX_NONE**. ¿O deberían guardarse solo los puntos que son estrictamente necesarios para dibujar el contorno? En ese caso, establezca la bandera en **v2.CHAIN_APPROX_SIMPLE**. le ahorrará mucha memoria.

Suponga que tiene una imagen de siluetas y necesita detectar contornos en ella:

Podemos almacenar la imagen anterior en una variable img. Luego, para detectar contornos, podemos aplicar el comando **cv2.findContours**. En este ejemplo introductorio, usaremos la detección de contorno externa y el método **CHAIN_APPROX_NONE:**

```
contours, hierarchy = cv2.findContours(img, cv2.RETR_EXTERNAL,
                                       cv2.CHAIN_APPROX_NONE)
```

Para trazar los contornos detectados en una imagen BGR, el comando es el siguiente:

```
marked_img = cv2.drawContours(img, contours, contourIdx, color, \
       thickness, lineType = cv.LINE_8, \
```

```
hierarchy = new cv.Mat(), \
maxLevel = INT_MAX, \
offset = new cv.Point(0, 0)))
```

El resultado de esta función es la imagen, img, con los contornos dibujados en ella.

Aquí, una breve explicación de las entradas de esta función:

- **img** es la versión BGR de la imagen en la que desea que se marquen los contornos. Debería ser BGR porque necesitará dibujar los contornos con algún color que no sea negro o blanco y el color tendrá tres valores en su código BGR. Entonces, necesitará tres canales en la imagen.
- **contours** es la lista de Python de contornos detectados.
- **contourIdx** es el contorno que desea dibujar de la lista de contornos. Si desea dibujarlos todos, entonces su valor de entrada será -1.
- **color** es el código de color BGR del color que desea utilizar para trazar. Por ejemplo, para el rojo será (0, 0, 255. Para convertir estos códigos de color RGB a BGR, simplemente invierta el orden de los tres valores.
- **thickness** es el ancho de la línea utilizada para trazar los contornos. Esta es una entrada opcional. Si especifica que sea negativo, los contornos dibujados se rellenarán con color.
- **lineType** es un argumento opcional que especifica la conectividad de línea.
- **hierarchy** es una entrada opcional que contiene información sobre los niveles de jerarquía presentes, de modo que, si desea establecer un nivel de jerarquía en particular, puede especificarlo en la siguiente entrada.
- **maxLevel** corresponde al nivel de profundidad de la jerarquía que desea dibujar. Si maxLevel es 0, solo se dibujarán los contornos exteriores. Si es 1, se dibujarán todos los contornos y sus contornos anidados (hasta el nivel 1). Si es 2, se dibujarán todos los contornos, sus contornos anidados y los contornos anidados a anidados (hasta el nivel 2).
- **offset** es un parámetro de desplazamiento de contorno opcional.

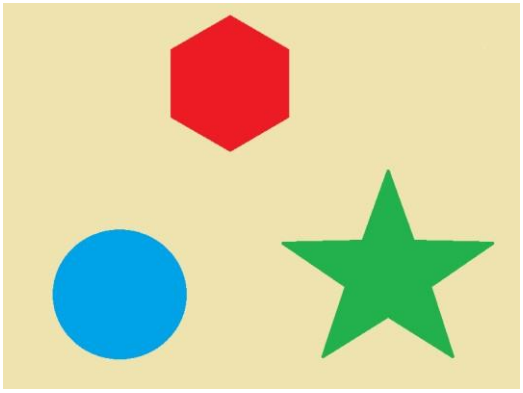
Para aplicar este comando al ejemplo anterior, usemos un color rojo (código BGR: 0, 0, 255) y un grosor de 2:

```
with_contours = cv2.drawContours(im_3chan, contours,-1,(0,0,255),2)
```

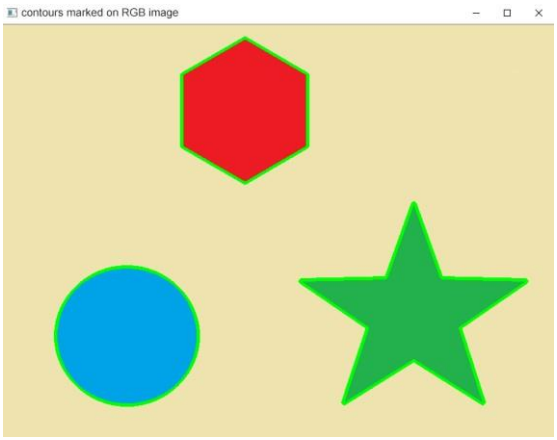
Aquí, **im_3chan** es la versión BGR de la imagen en la que queremos dibujar los contornos. Ahora, si visualizamos la imagen de salida, veremos lo siguiente:

4. ACTIVIDAD I. DETECTAR FORMAS Y MOSTRARLAS EN IMÁGENES BGR.

Se tiene la siguiente imagen coloreada con diferentes formas:



La tarea es contar todas las formas y detectar sus límites exteriores de la siguiente manera:

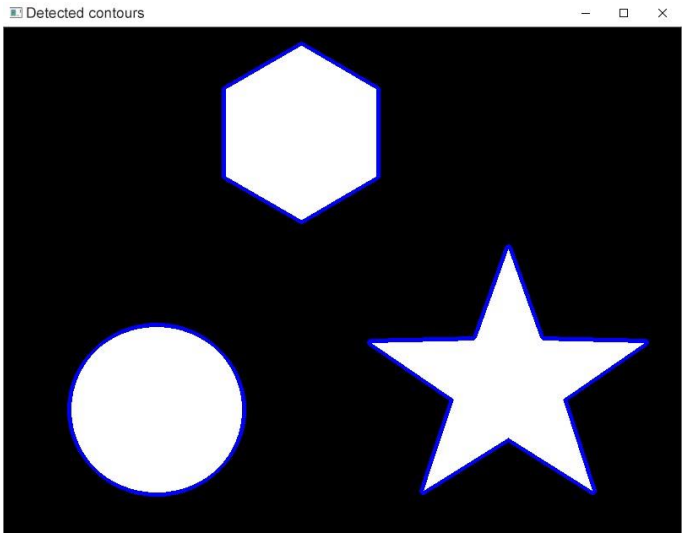


Ejercicio 8.1

```
import cv2
#--Lea la imagen como una imagen BGR:
image = cv2.imread('images/sample shapes.png')
#--Conviértelo a escala de grises:
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
cv2.imshow('gray' , gray_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
#--Convertir a imafgen binaria con Otsu
ret,binary_im = cv2.threshold(gray_image,0,255,cv2.THRESH_OTSU)
cv2.imshow('imagen binaria', binary_im)
cv2.waitKey(0)
cv2.destroyAllWindows()
#--Invierta la imagen y muéstrela
inverted_binary_im= ~binary_im
cv2.imshow('inverso de la imagen binaria', inverted_binary_im)
cv2.waitKey(0)
cv2.destroyAllWindows()
#--Encuentra los contornos en la imagen binaria
contours,hierarchy = cv2.findContours(inverted_binary_im,
cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
#--Marque todos los contornos detectados en la imagen BGR original en  cualquier
color (digamos, verde).
#--Estableceremos el grosor en 3:
with_contours = cv2.drawContours(image, contours, -1,(0,255,0),3)
cv2.imshow('Contornos detectados en la imagen RGB', with_contours)
cv2.waitKey(0)
cv2.destroyAllWindows()
#--Finalmente, muestre el recuento total de los contornos detectados:
print('El número total de contornos detectados es:')
print(len(contours))
```

5. ACTIVIDAD II. DETECTAR FORMAS Y MOSTRARLAS EN IMÁGENES EN BLANCO Y NEGRO

En el ejercicio anterior, trazó los contornos en una imagen en color. Ahora la tarea es trazar los contornos detectados en un color azul (código BGR: 255, 0, 0), pero en la siguiente imagen en blanco y negro:



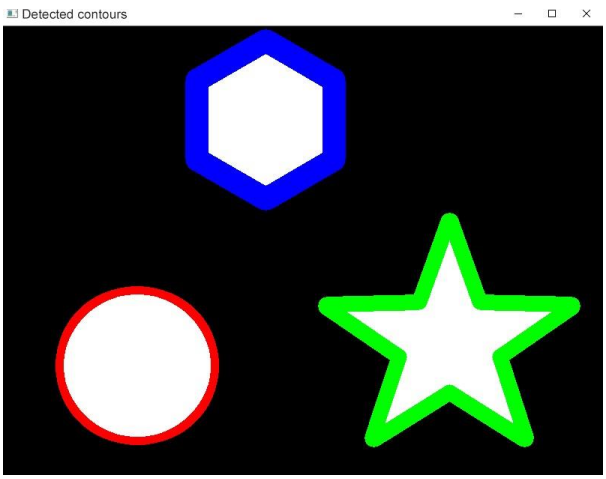
Ejercicio 8.2

```
import cv2
#--Lea la imagen como una imagen BGR:
image = cv2.imread('images/sample shapes.png')
#--Conviértelo a escala de grises: gray_image =
cv2.cvtColor(image, cv2.COLOR_BGR2GRAY) cv2.imshow('gray' ,
gray_image) cv2.waitKey(0) cv2.destroyAllWindows()
#--Convertir a imagen binaria con Otsu ret,binary_im =
cv2.threshold(gray_image,0,255,cv2.THRESH_OTSU) cv2.imshow('imagen
binaria', binary_im) cv2.waitKey(0) cv2.destroyAllWindows()

#--Invierta la imagen y muéstrela inverted_binary_im= ~binary_im
cv2.imshow('inverso de la imagen binaria', inverted_binary_im)
cv2.waitKey(0) cv2.destroyAllWindows()
#-la imagen debe tener tres canales, replicaremos el plano único de la
# imagen binaria tres veces y luego fusionaremos los tres planos p ara
# extenderlo al espacio de color BGR. bgr = cv2.merge([inverted_binary_im,
inverted_binary_im, inverted_ binary_im]);
#--Encuentra los contornos en la imagen binaria
contours,hierarchy = cv2.findContours(inverted_binary_im, cv2.RETR
_TREE, cv2.CHAIN_APPROX_SIMPLE)
#--
Marque todos los contornos detectados en la imagen BGR original en  cualquier
color (digamos, verde).
#--Estableceremos el grosor en 3: with_contours = cv2.drawContours(bgr,
contours, -1, (255,0, 0),3) cv2.imshow('Contornos detectados', with_contours)
cv2.waitKey(0) cv2.destroyAllWindows()
#--
Finalmente, muestre el recuento total de los contornos detectados: print('El
número total de contornos detectados es:') print(len(contours))
```

6. ACTIVIDAD III. VISUALIZACIÓN DE DIFERENTES CONTORNOS CON DIFERENTES COLORES Y ESPESORES

Esta es una extensión del ejercicio anterior. Haremos lo siguiente:

1. Marque el contorno número 1 en color rojo con un grosor de 10.
2. Marque el contorno número 2 en color verde con un grosor de 20.
3. Marque el contorno número 3 en color azul con un grosor de 30. Y se obtendrá los siguiente:



Ejercicio 8.3

```
import cv2
#--Lea la imagen como una imagen BGR:
image = cv2.imread('images/sample shapes.png')
#--Conviértelo a escala de grises:
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
cv2.imshow('gray' , gray_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
#--Convertir a imagen binaria con Otsu
ret,binary_im = cv2.threshold(gray_image,0,255,cv2.THRESH_OTSU)
cv2.imshow('imagen binaria', binary_im)
cv2.waitKey(0)
cv2.destroyAllWindows()

#--Invierta la imagen y muéstrela
inverted_binary_im= ~binary_im
cv2.imshow('inverso de la imagen binaria', inverted_binary_im)
cv2.waitKey(0)
cv2.destroyAllWindows()
#--la imagen debe tener tres canales, replicaremos el plano único de la
# imagen binaria tres veces y luego fusionaremos los tres planos para
```

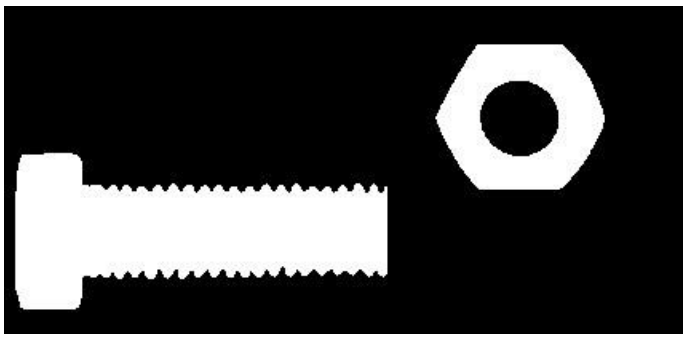
```
# extenderlo al espacio de color BGR.
bgr = cv2.merge([inverted_binary_im, inverted_binary_im, inverted_binary_im]);
#--Encuentra los contornos en la imagen binaria
contours,hierarchy = cv2.findContours(inverted_binary_im, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)
#--Marque todos los contornos detectados en la imagen BGR original en cualquier
color (digamos, verde).
#--Estableceremos el grosor en 3:

with_contours = cv2.drawContours(bgr, contours, 0,(0,0,255),10)
with_contours = cv2.drawContours(with_contours, contours, 1,(0, 255, 0),20)
with_contours = cv2.drawContours(with_contours, contours, 2, (255,0, 0), 30)

cv2.imshow('Contornos detectados', with_contours)
cv2.waitKey(0)
cv2.destroyAllWindows()
#--Finalmente, muestre el recuento total de los contornos detectados:
print('El número total de contornos detectados es:')
print(len(contours))
```

7. ACTIVIDAD IV: DETECTAR UN PERNO Y UNA TUERCA

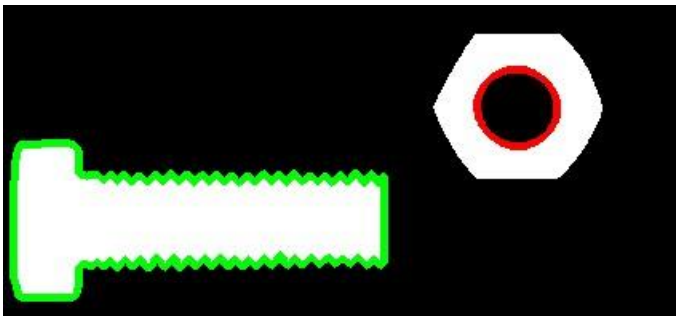
Suponga que está diseñando un sistema en el que se va a insertar un perno en una tuerca. Con la imagen proporcionada, debe averiguar la ubicación exacta de lo siguiente:

- El cerrojo, para que el robot pueda levantarlo.
- El orificio interior de la tuerca, para que el robot sepa la ubicación exacta donde se insertará el perno.

Dada la siguiente imagen binaria del perno y la tuerca, detecte el perno y el orificio interior de la tuerca:



El perno y el orificio interior de la tuerca deben marcarse en colores separados de la siguiente manera:



Ejercicio 8.4

```
import cv2
image_3chan = cv2.imread('images/nut_bolt.png')
image_3chan_copy= image_3chan.copy()
cv2.imshow('Imagen Original', image_3chan)
cv2.waitKey(0)
cv2.destroyAllWindows()
gray_image = cv2.cvtColor(image_3chan, cv2.COLOR_BGR2GRAY)
ret,binary_im = cv2.threshold(gray_image,250,255,cv2.THRESH_BINARY)
cv2.imshow('Imagen Binaria', binary_im)
cv2.waitKey(0)
cv2.destroyAllWindows()

contours_list,hierarchy =
cv2.findContours(binary_im,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
print('Información de jerarquía de todos los contornos:')
print (hierarchy)
for i in range(0, len(contours_list)):
    contour_info= hierarchy[0][i, :]
    print('Información de jerarquía del contorno actual:')
    print(contour_info)
    # no parent, no child
    if contour_info[2]==-1 and contour_info[3]==-1:
        with_contours = cv2.drawContours(image_3chan_copy,
contours_list,i,[0,255,0],thickness=3)
        print('Se detecta el contorno del perno')
```

```
    if contour_info[2]==-1 and contour_info[3]!=-1:
        with_contours =
cv2.drawContours(with_contours,contours_list,i,[0,0,255],thickness=3)
        print('Se detecta un agujero de tuerca')

cv2.imshow('Contornos marcados en la imagen RGB', with_contours)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

8. ACTIVIDAD V: DETECTAR UNA RED DE BALONCESTO EN UNA IMAGEN

La tarea es dibujar un cuadro delimitador alrededor de la canasta después de detectarla. El enfoque que seguiremos aquí es que convertiremos esta imagen a binaria usando un umbral tal que toda la región blanca de la canasta se detecte como un solo objeto. Este será, sin duda, el contorno con el área más grande aquí. Luego, marcaremos este contorno más grande con un cuadro delimitador. Aquí está la imagen basketball.jpg:

La salida debería ser algo como esto:

Ejercicio 8.5

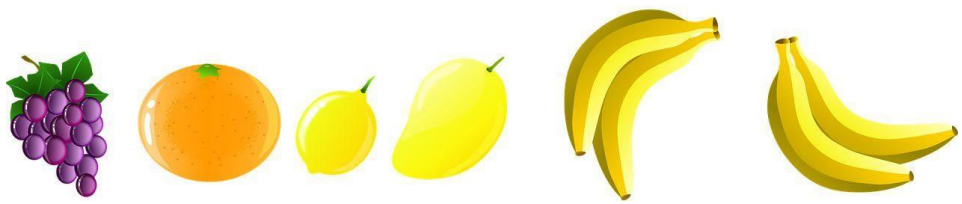
```
import cv2
image = cv2.imread('images/basketball.jpg')
```

```
#--Haga una copia de esta imagen y guárdela en otra variable
imageCopy= image.copy()
cv2.imshow('imagen BGR', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
#--Convierta la imagen a escala de grises y visualice
gray_image = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
cv2.imshow('gray', gray_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
#--Convierta esta imagen en escala de grises en una imagen binaria utilizando
un umbral
# tal que toda la región del límite blanco de la red de baloncesto se detecte
como una sola mancha:
ret,binary_im = cv2.threshold(gray_image,100, 255, cv2.THRESH_BINARY)
cv2.imshow('Binario', binary_im)
cv2.waitKey(0)
cv2.destroyAllWindows()
#--Detecta todos los contornos
contours,hierarchy = cv2.findContours(binary_im,
cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
#--Dibuje todos los contornos detectados en la imagen y luego muestre la
imagen.
contours_to_plot= -1
plotting_color= (0,255,0) #--verde
# si queremos rellenar con color los contornos dibujados
thickness= -1
with_contours = cv2.drawContours(image,contours, contours_to_plot,
plotting_color,thickness)
cv2.imshow('Contornos', with_contours)
cv2.waitKey(0)
cv2.destroyAllWindows()
#--A continuación, debemos trazar cuadros delimitadores alrededor de todos
los contornos.
for cnt in contours:
    x,y,w,h = cv2.boundingRect(cnt)
    image = cv2.rectangle(image,(x,y),(x+w,y+h), (0,255,255),2)
#--mostrar la imagen
```

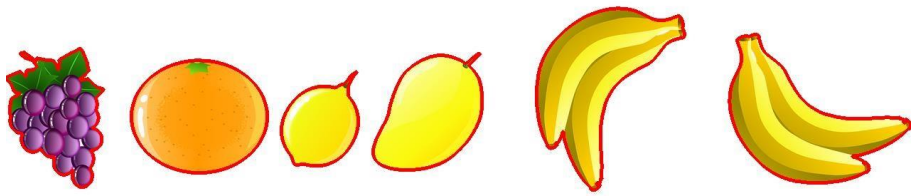
```
cv2.imshow('Contornos', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
#--Encuentra el contorno con el área más grande:
required_contour = max(contours, key = cv2.contourArea)
#--Encuentre las coordenadas x e y iniciales y el ancho y alto
# de un cuadro delimitador rectangular que debe encerrar este contorno más
grande
x,y,w,h = cv2.boundingRect(required_contour)
#--Dibuje este cuadro delimitador en una copia de la imagen en color original
# que había guardado anteriormente:
img_copy2 = cv2.rectangle(imageCopy, (x,y),(x+w, y+h), (0,255,255),2)
#--mostrar el resultado
cv2.imshow('Contorno más grande', img_copy2)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

9. ACTIVIDAD VI: DETECTANDO FRUTAS EN UNA IMAGEN

Se tiene la siguiente imagen de una selección de frutas:



La tarea es detectar todas las frutas presentes en esta imagen. Haga esto a través de la detección de contornos y dibuje las formas de los contornos en la imagen, de modo que los contornos de la fruta se dibujen en rojo:



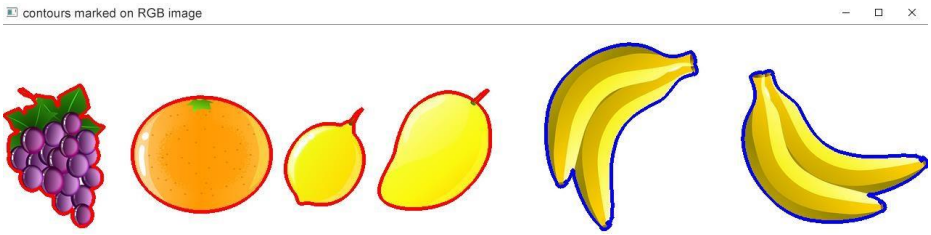
Ejercicio 8.6

```
import cv2
image = cv2.imread('images/many fruits.png')
```

```
cv2.imshow('Imagen original', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
#--convertir a escala de grises
gray_image = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
cv2.imshow('gray', gray_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
#--Convertir a binario con un umbral adecuado
ret,binary_im = cv2.threshold(gray_image,245, 255,cv2.THRESH_BINARY)
cv2.imshow('binario', binary_im)
cv2.waitKey(0)
cv2.destroyAllWindows()
#--invertir la imagen
binary_im= ~binary_im
cv2.imshow('binario invertido', binary_im)
cv2.waitKey(0)
cv2.destroyAllWindows()
#--encontrar los contornos externos
contours,hierarchy = cv2.findContours(binary_im,
cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIMPLE)
#--dibujar los contornos
with_contours = cv2.drawContours(image,contours, -1,(0,0,255),3)
cv2.imshow('contornos marcados en la imagen RGB', with_contours)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

10. TAREA

Identificar todos los plátanos presentes en la imagen del ejercicio anterior y marcarlos en azul, de la siguiente manera:



11. REFERENCIAS BIBLIOGRÁFICAS

- Dawson-Howe, K. (2014). A Practical Introduction to Computer Vision With OpenCV, Wiley & Sons Ltd.
- Hafsa Asad, W. R., Nikhil Singh (2020). The Computer Vision Workshop. Birmingham U.K., Pack Publishing.
- Szeliski, R. (2011). Computer Vision Algorithms and Applications. London, Springer.