

<b>Laboratorio N° 4- Detectores de Primitivas</b>			
<b>Asignatura</b>	Tópicos Especiales - Visión Artificial	<b>Código</b>	0756
<b>Profesor</b>	José Carlos Rangel Ortiz		

## Introducción

En este laboratorio se mostraran los diferentes algoritmos disponibles en OpenCV para la detección de puntos clave dentro de las imágenes. Se abordaran los detectores de esquinas y regiones en diferentes imágenes. De igual manera se presenta al estudiante la manera de generar descriptores para puntos clave de las imágenes.

Dentro del contenido se emplearan de igual manera las librerías Numpy y Matplotlib, las cuales seguirán siendo utilizadas a lo largo del semestre.

## Fundamentos Teóricos

El ser humano tiene una capacidad sorprendente para unir las piezas o trozos de un objeto o elemento que está roto o desarmado. Si nos centramos en el ejemplo de un rompecabezas, los humanos desde tempranas edades demuestran la aptitud de armarlos basándose en la forma y contenido visual de cada pieza.

Este proceso que se puede decir está programado en nuestro cerebro, se enfoca en comparar las piezas y encontrar en ellas fragmentos de información que permitan determinar la continuidad de la imagen cuando dichas piezas se unan.

Instintivamente los seres humanos detectan estas “características” en cada pieza y usando este método son capaces de completar zonas de mayor tamaño cada vez.

La Visión Artificial al tratar de emular este comportamiento ha definido en sus procesos la manera de identificar estas características y utilizarlas en sus procesos. Para ello se basa en diversas funciones matemáticas que analizan la imagen de entrada y generan una lista de puntos que son considerados claves en la imagen y los cuales representan esta mediante un número reducido de parámetros.

Al trabajar con imágenes en el computador se hace difícil para muchos procesos emular el comportamiento humano, por lo cual, se han desarrollado una diversidad de algoritmos que detectan estos puntos clave (*keypoints*), según el tipo o contenido de las imágenes. De igual manera han surgido los algoritmos que buscan no solo un punto, sino un conjunto o región de estos ya sea una esquina, o alguna forma geométrica primitiva.

Este laboratorio presenta de manera práctica los diferentes algoritmos para detectar primitivas geométricas. Estos algoritmos se fundamentan mayoritariamente en ecuaciones algebraicas que permiten interpretar las intensidades de color en la imagen e identificar la presencia de alguna

forma en concreto. Estos algoritmos a menudo son utilizados como base para otros algoritmos de procesamiento y detección a mayor escala.

## **Objetivo**

Aplicar detectores de formas geométricas básicas presentes en imágenes.

## **Entregables**

Al final de cada sección del laboratorio se indicará cuales son acciones que deberá cumplir el estudiante y los cuales serán evaluados en la actividad. El informe final debe contener todo lo solicitado y se debe redactar utilizando la Guía de Estilos para Memorias y Laboratorios Disponible en Moodle. El informe debe ser entregado en formato PDF y siguiendo las indicaciones para nombrar el archivo.

## 1. Detectores

### 1.1. Detector de Esquinas Harris

En una imagen las esquinas son regiones que presentan grandes cambios en la intensidad en todas las direcciones. El detector de Harris fue un intento temprano para encontrar estas esquinas dentro de las imágenes. Fue presentado en el artículo *A Combined Corner and Edge Detector*[1] en 1988.

Los autores tomaron la características de variación de la intensidad y la llevaron a un forma matemática. Ellos miden básicamente la diferencia de intensidad para un desplazamiento  $(u, v)$  en todas las direcciones.

$$E(u, v) = \sum_{x,y} \underbrace{w(x, y)}_{\text{window function}} \left[ \underbrace{I(x + u, y + v)}_{\text{shiftedIntensity}} - \underbrace{I(x, y)}_{\text{intensity}} \right]^2 \quad (1)$$

En esta ecuación *window function* es una ventana rectangular o Gaussiana que le da una ponderación a los píxeles bajo ella. Luego de aplicar una expansión usando la Serie de Taylor a esta ecuación se obtiene la siguiente aproximación:

$$E(u, v) \approx \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix} \quad (2)$$

Donde:

$$M = \sum_{x,y} w(x, y) \begin{bmatrix} I_x I_x & I_x I_y \\ I_x I_y & I_y I_y \end{bmatrix} \quad (3)$$

En esta ecuación  $I_x$  e  $I_y$  son las derivadas de la imagen en  $x$  e  $y$  respectivamente. Como siguiente paso se tiene la creación de una puntuación, básicamente una ecuación que determina si una ventana puede contener una esquina o no.

$$R = \det(M) - k(\text{trace}(M))^2 = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2 \quad (4)$$

Donde:

- $\det(M) = \lambda_1 \lambda_2$
- $\text{trace}(M) = \lambda_1 + \lambda_2$
- $\lambda_1$  y  $\lambda_2$  son los *eigenvalues* o vetores propios de  $M$  estos se pueden calcular utilizando un filtro de *Sobel*.

Son precisamente las magnitudes de estos vectores los que definen si una región es una esquina, un borde o es plano.

- Cuando  $|R|$  es pequeño lo cual, ocurre cuando  $\lambda_1$  y  $\lambda_2$  son pequeños, la región es plana.
- Cuando  $R < 0$ , lo cual ocurre cuando  $\lambda_1 \gg \lambda_2$  o vice versa, la región es un borde.
- Cuando  $R$  es grande, lo cual ocurre cuando  $\lambda_1$  y  $\lambda_2$  son grandes y  $\lambda_1 \sim \lambda_2$ , la región es una esquina.

En el caso de OpenCV se cuenta con la función `cv.cornerHarris()`, la cual cuenta con los siguientes parámetros:

**img** La imagen de entrada debe estar en escala de grises

**blockSize** Tamaño del vecindario considerado para la detección de esquinas

**ksize** Parámetro de apertura para el filtro Sobel utilizado (debe ser impar)

**k** Parámetro libre en la ecuación de Harris

El siguiente ejemplo muestra como utilizar la función `cv.cornerHarris()` de OpenCV para encontrar las esquinas dentro de una imagen.

Para este y los siguientes ejemplos se puede utilizar Jupyter o cualquier IDE para desarrollo con Python, se supondrá que dentro de la carpeta donde guarda su script existe una carpeta llamada *img* donde están todas las imágenes que se utilizarán como entrada y otra carpeta llamada *out* donde se almacenarán todas las imágenes que produzca el *script*.

1. Primer paso crear su *script/notebook* y añadir las importaciones a OpenCV, Numpy(np) y Pyplot(plt) de Matplotlib.
2. En el siguiente código cargamos nuestra imagen y se convierte a escala de grises, tal cual lo pide la función.

---

```
imgHarris = cv2.imread('img/crucil1.jpg')
#Convertir a escala de grises
gray = cv2.cvtColor(imgHarris, cv2.COLOR_BGR2GRAY)
gray = np.float32(gray)
```

---

3. El siguiente paso es la definición de los parámetros y la aplicación del método de Harris. El algoritmo produce como salida una imagen de igual tamaño a la imagen de entrada y en la cual se almacena las respuestas de los detectores de Harris. En esta porción de código esta imagen recibe un proceso de dilatado<sup>1</sup> para facilitar la visualización de los resultados.

---

<sup>1</sup>Dilatar en OpenCV

---

```
blockSize = 2
ksize = 3
k = 0.08

dst = cv2.cornerHarris(gray,blockSize,ksize,k)
#print(dst)
# Se dilata el resultado para marcar las esquinas
# se hacen más visibles al visualizar la imagen
dst = cv2.dilate(dst,None)
```

---

4. El resto del código se encarga de pintar los puntos detectados en la imagen original, mostrar y guardar la imagen con los puntos identificados.

```
# Umbral para un valor óptimo, puede variar según la imagen
puntos = imgHarris[dst>0.01*dst.max()].sum()

print( "Puntos Totales Usando Harris : {}".format(puntos) )

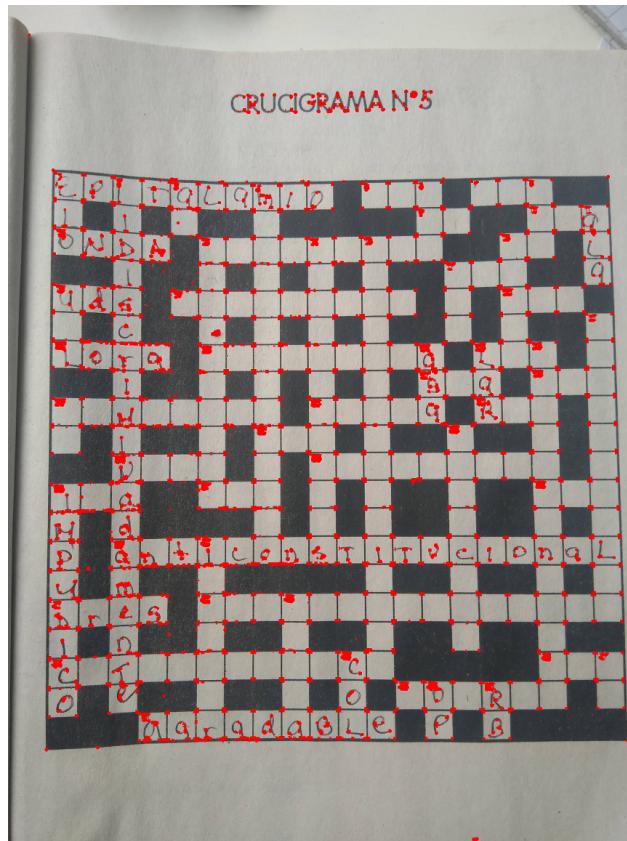
imgHarris[dst>0.01*dst.max()]=[0,0,255]
cv2.imshow('dst',imgHarris)
cv2.waitKey(0)

# Para destruir todas las ventanas creadas
cv2.destroyAllWindows()
cv2.imwrite("out/Harris.png", imgHarris)

plt.imshow(imgHarris[:, :, ::-1] )
plt.axis('off')
plt.show()
```

---

5. Lo cual producirá el siguiente resultado:



## 1.2. Detector de Esquinas Shi-Tomasi

Este detector consiste en una mejora producida en la formulación del detector de Harris. Dicha mejora fue propuesta por Shi y C. Tomasi en 1994 en su artículo *Good Features to Track* [2]. La modificación se aplica en la ecuación que evalúa los puntos, en Harris se usa la siguiente formula:

$$R = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2 \quad (5)$$

Mientras que Shi-Tomasi lo plantea de la siguiente manera:

$$R = \min(\lambda_1, \lambda_2) \quad (6)$$

De tal manera que si el valor es mayor a un *threshold* definido, se considera el punto como una esquina. Para este detector OpenCV cuenta con la función `cv.goodFeaturesToTrack()`<sup>2</sup>. Esta función al igual que en Harris debe recibir una imagen en escala de grises. Los parámetros que recibe son los siguientes:

**img** Imagen en escala de grises para procesar

---

<sup>2</sup>`goodFeaturesToTrack()`

**cantidad** cantidad de puntos deseados

**calidad** un valor entre 0 – 1. Indica la calidad mínima aceptada para las esquinas detectadas.

**dist** Distancia Euclídea mínima entre puntos/esquinas detectadas.

1. Como primer paso crea un *script/notebook* y añade las importaciones a OpenCV, Numpy(np) y Pyplot(plt) de Matplotlib.
2. En el siguiente código cargamos nuestra imagen y se convierte a escala de grises, tal cual lo pide la función, se definen los parámetros y se aplica el método a la imagen de entrada.

---

```
imgShi = cv2.imread('img/tower2.png')
gray = cv2.cvtColor(imgShi, cv2.COLOR_BGR2GRAY)

# Cantidad Deseada
cantidad = 65

# Calidad [0-1]
qlt = 0.08

# Distancia Euclidea mínima entre puntos
euDist = 10
corners = cv2.goodFeaturesToTrack(gray, cantidad, qlt, euDist)
corners = np.intp(corners)

print("Keypoints Totales Usando Shi-imgTomasi : {}".format(len(corners)))
```

---

3. La siguiente fracción de código establece la configuración para vizualizar las esquinas detectadas en la imagen de entrada y guardar la imagen en disco.

---

```
# Radio del circulo
radio = 8

# color en BGR
color = (0, 255, 0)

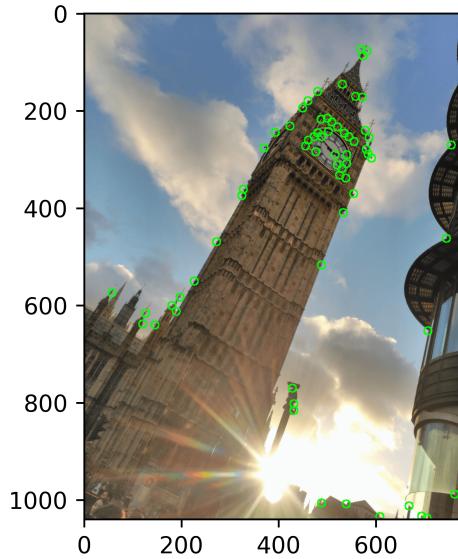
# grosor de la línea
thickness = 2

for i in corners:
    x,y = i.ravel()
    cv2.circle(imgShi,(x,y),radio, color, thickness)

plt.imshow(imgShi[:, :, ::-1])
plt.savefig("out/Tomasi.png", dpi=600, orientation='portrait', transparent=True)
plt.axis('off')
plt.show()
```

---

4. Lo cual producirá el siguiente resultado:



### 1.3. Detector Canny

1. En esta sección aplicaremos el detector de Canny para analizar una imagen y detectar los diferentes bordes que se encuentran en la imagen.
2. Este método recibe como argumentos los siguientes parámetros:
  - Imagen en escala de grises ( $8 - bits$ )
  - threshold1 usado en el proceso de histéresis
  - threshold2 usado en el proceso de histéresis
  - Tamaño de apertura del operador de sobel (3, 5, 7), usado para el calculo del gradiente
3. El valor menor entre los threshold es utilizado como límite inferior y el mayor como límite superior para la histéresis.

---

```
# importaciones
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('monedasParcial.jpg')

# Verificar la existencia de la imagen
```

```
if img2 is None:  
    sys.exit('Fallo al cargar la imagen')  
  
img_gris = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)  
img.blur = cv2.blur(img_gris, (3,3))  
  
edges = cv2.Canny(img_gris, 50, 150, apertureSize = 3)  
cv2.imwrite('edgesCanny.jpg', edges)  
  
  
plt.subplot(121)  
plt.axis("off")  
plt.imshow(img[:, :, ::-1])  
plt.title('Imagen Original')  
  
plt.subplot(122)  
plt.axis("off")  
plt.imshow(edges, cmap='gray')  
plt.title('Bordes Detectados')  
  
plt.savefig("BordesDetectados.png", dpi=600)  
plt.show()
```

- 
4. Luego de la ejecución se obtendrá el siguiente resultado (figura 1).

#### Entregables para la Sección

1. Seleccione una imagen y aplique los detectores de Harris y Shi-Tomasi. Aplique a cada imagen, cada detector.
2. Aplique un filtro bilateral a cada imagen original. Posteriormente, aplique los detectores a la imagen filtrada, grafique los resultados.
3. Presente los resultados a modo comparativo de la detección antes y después de aplicar el filtro bilateral.

## 2. Transformada de Hough

La Transformada de Hough es una técnica popular para detectar cualquier forma, si se puede representar esa forma de manera matemática. Puede detectar la forma incluso si está rota o un poco distorsionada.

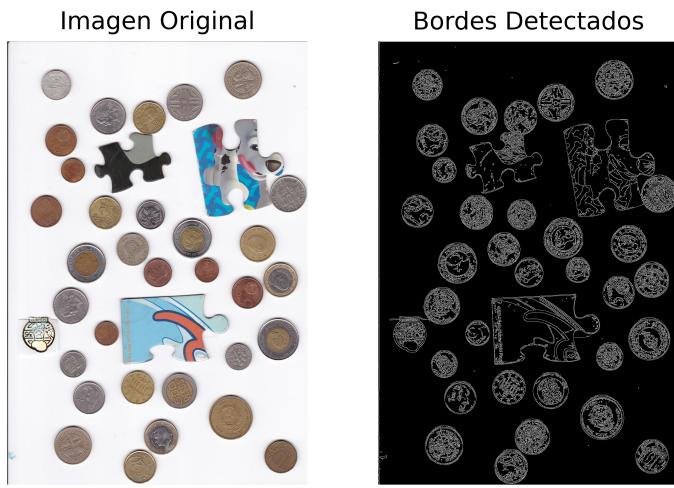
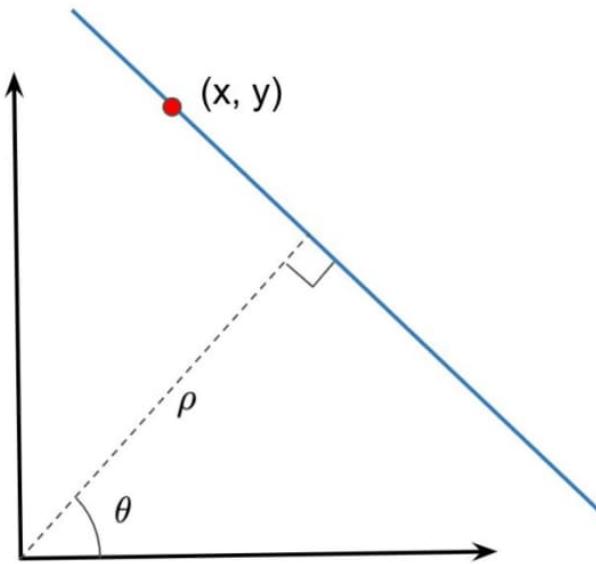


Figura 1: Salida producida por el operador de Canny.

Para una línea el funcionamiento se puede expresar de la siguiente manera. Una línea se puede representar como  $y = mx + b$  o en forma paramétrica, como  $\rho = x\cos\theta + y\sin\theta$  donde  $\rho$  es la distancia perpendicular desde el origen a la línea, y  $\theta$  es el ángulo formado por esta línea perpendicular y el eje horizontal medido en sentido antihorario (esa dirección varía según la forma en que se representa el sistema de coordenada). Esta representación se utiliza en OpenCV). Verifique la siguiente imagen:



Entonces, si la línea pasa por debajo del origen, tendrá un  $\rho$  positivo y un ángulo( $\theta$ ) menor de 180. Si va por encima del origen, en lugar de tomar un ángulo mayor que 180, el ángulo se toma menos de 180 y  $\rho$  es negativo. Cualquier línea vertical tendrá 0 grados y las líneas horizontales 90 grados.

Ahora veamos cómo funciona la Transformada de Hough para líneas. Cualquier línea se puede representar en estos dos términos,  $(\rho, \theta)$ . Entonces, primero se crea una matriz o acumulador 2D (para mantener los valores de dos parámetros) y se establece en 0 inicialmente. Se define que las filas denotan el  $\rho$  y las columnas denotan  $\theta$ . El tamaño de la matriz depende de la precisión que necesite. Supongamos que se desea una precisión de los ángulos sea de 1 grado, se necesitarían 180 columnas. Por otra parte, para  $\rho$ , la distancia máxima posible es la longitud diagonal de la imagen. Entonces, tomando la precisión de un píxel, el número de filas puede ser la longitud de la diagonal de la imagen.

Consideremos una imagen de  $100 \times 100$  con una línea horizontal en el medio. Tomando el primer punto de la línea, se conocen sus valores  $(x, y)$ . Ahora en la ecuación de la línea, se colocan los valores  $\theta = 0, 1, 2, \dots, 180$  y se verifica el valor de  $\rho$  que se obtenga. Para cada par  $(\rho, \theta)$ , se incrementa el valor en uno(1) en las celdas correspondientes de nuestro acumulador  $(\rho, \theta)$ . Entonces ahora en el acumulador, la celda  $(50, 90) = 1$ ,  $\rho = 50$  ya que es la recta esta en la mitad de la imagen y  $\theta = 90$ , el angulo que se forma con el origen.

Ahora tomando el segundo punto de la línea. Se repite lo mismo que arriba. Se incrementan los valores en las celdas correspondientes  $(\rho, \theta)$  a lo que se obtuvo. Esta vez, la celda  $(50, 90) = 2$ . Lo que realmente hace el procedimiento es votar los  $(\rho, \theta)$  valores. Se continúa este proceso para cada punto de la línea. En cada punto, la celda  $(50, 90)$  se incrementará o votará a favor, mientras que otras celdas pueden o no votar a favor. De esta forma, al final, la celda  $(50, 90)$  tendrá el

máximo de votos. Entonces, si se busca en el acumulador el máximo de votos, se obtiene el valor (50, 90) que dice, hay una línea en esta imagen a una distancia de 50 del origen y en un ángulo de 90 grados. Este proceso se muestra bien en animación de Amos Storkey, disponible en este enlace<sup>3</sup>.

## 2.1. Detectar Líneas

Todo lo explicado anteriormente está encapsulado en la función de OpenCV, `cv2.HoughLines()`. Simplemente devuelve una matriz de valores  $(\rho, \theta)$ .  $\rho$  se mide en píxeles y  $\theta$  se mide en radianes. El primer parámetro, es la imagen de entrada, debe ser una imagen binaria, así que se aplica umbralización o el Algoritmo de Canny<sup>4</sup> antes de aplicar la transformada. El segundo y tercer parámetro son las precisiones en  $\rho$  y  $\theta$  respectivamente. El cuarto argumento es el umbral, lo que significa, el voto mínimo que se debe obtener para que se considere una línea. Recuerde, el número de votos depende del número de puntos de la línea. Por lo tanto, representa la longitud mínima de línea que debe detectarse.

El método `cv2.HoughLines()` produce una lista de pares  $(\rho, \theta)$ , por lo cual para graficar esto en nuestra imagen original debemos realizar el procedimiento indicado en el código y así marcar la línea identificada en la imagen. Para mayor información se puede consultar en el siguiente enlace<sup>5</sup>.

1. Como primer paso crea un *script/notebook* y añade las importaciones a OpenCV, `Numpy(np)` y `Pyplot(plt)` de Matplotlib.
2. En este caso se aplicará la transformada de Hough para detectar líneas en una imagen.

---

```



```

---

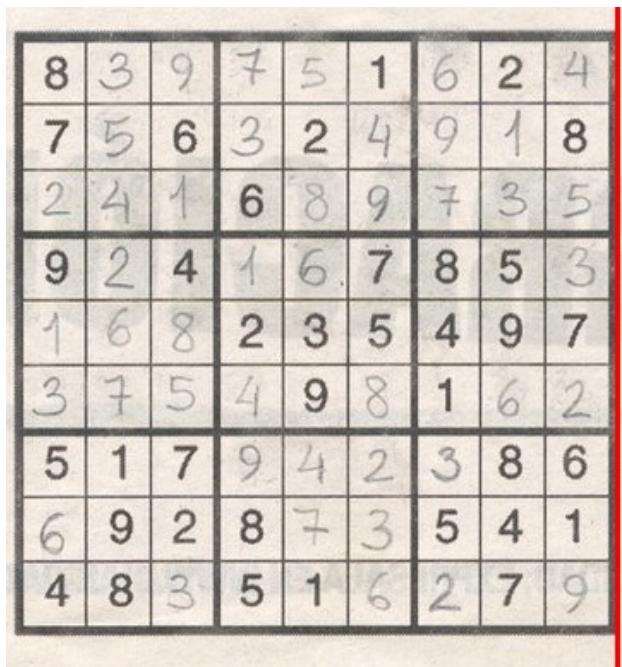
<sup>3</sup>Hough

<sup>4</sup>Canny

<sup>5</sup>Hough Línea

```
cv2.imwrite('out/houghlines1.jpg', img1)
plt.imshow(img1[:, :, ::-1])
```

3. Lo cual producirá el siguiente resultado:



## 2.2. Transformada Probabilística de Hough

En la transformada de Hough, se puede ver que incluso para una línea con dos argumentos se necesita mucho cálculo. La Transformada probabilística de Hough es una optimización de la transformada de Hough que vimos. No toma en consideración todos los puntos, sino que solo toma un subconjunto aleatorio de puntos y eso es suficiente para la detección de líneas. Solo tenemos que disminuir el umbral.

Con el método anterior cada elemento de la lista que producía Hough estaba compuesto por un par de valores  $(\rho, \theta)$ , en este caso se utilizará la Transformada Probabilística de Hough `cv2.HoughLinesP()`. Con esta función se creará un conjunto de 4 que denotan las coordenadas de una línea en la imagen de entrada  $(x_1, y_1, x_2, y_2)$ . Por lo cual ya no se deben calcular en base a las coordenadas polares devueltas por el método visto en el apartado anterior.

En este caso se manejan 2 nuevos parámetros:

- `minLineLength`: longitud mínima de la línea. Los segmentos de línea más cortos que este se rechazan.

- maxLineGap: espacio máximo permitido entre segmentos de línea para tratarlos como una sola línea.
1. Como primer paso crea un *script/notebook* y añade las importaciones a OpenCV, Numpy(np) y Pyplot(plt) de Matplotlib.
  2. Agregue el siguiente código

---

```
img3 = cv2.imread('img/cruc1.jpg')
h,w,c = img3.shape
# Nuevo Tamaño
dsize = (int(w*0.250), int(h*0.250))
# resize image
img4 = cv2.resize(img3, dsize)

gray4 = cv2.cvtColor(img4, cv2.COLOR_BGR2GRAY)
edges4 = cv2.Canny(gray4, 50, 150, apertureSize = 3)

lines4 = cv2.HoughLinesP(edges4, 1, np.pi/180, 100, minLineLength=100, maxLineGap=10)
print(lines4[0])

for line in lines4:
    x1, y1, x2, y2 = line[0]
    cv2.line(img4, (x1,y1), (x2,y2), (0,255,0), 1, cv2.LINE_AA)

cv2.imwrite("out/lineasHough4.png", img4)

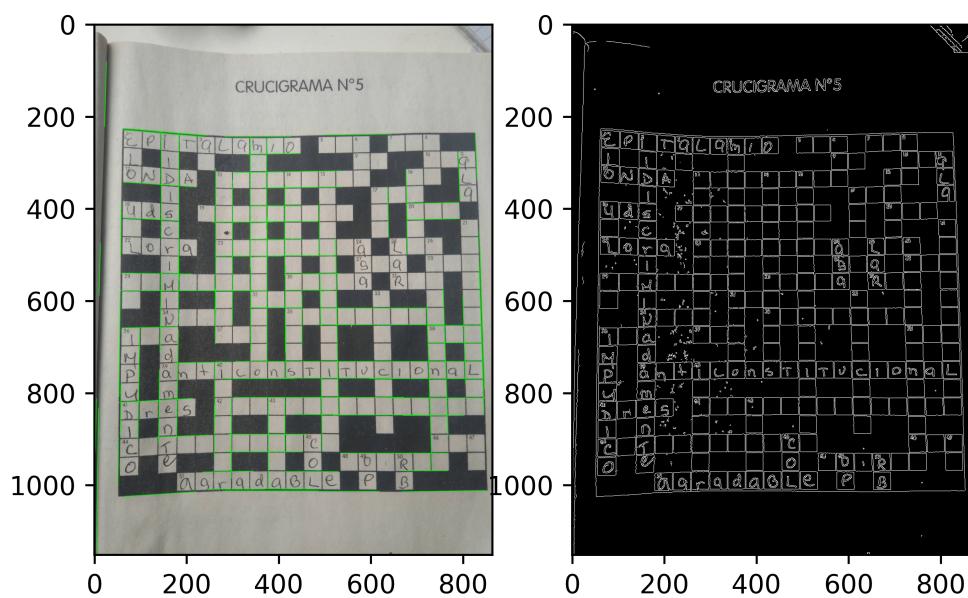
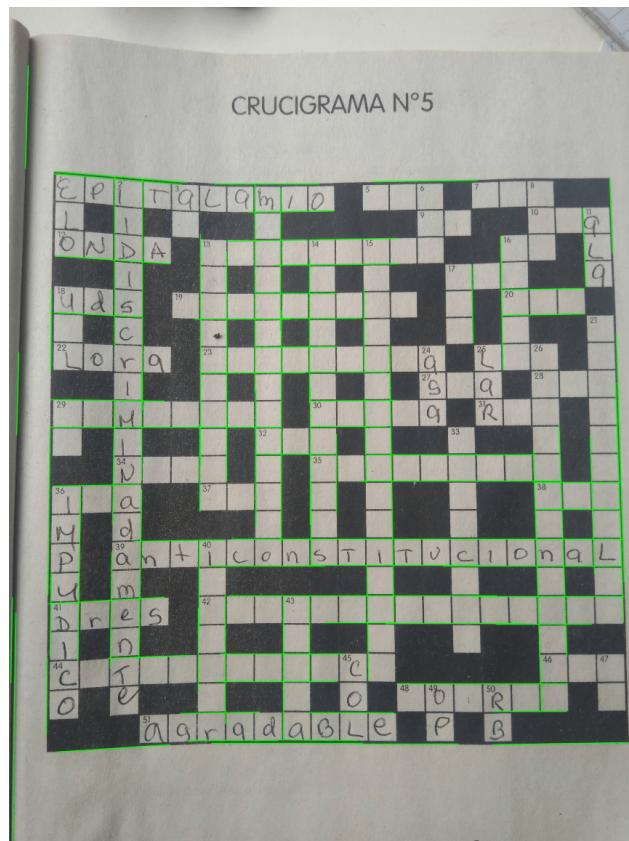
plt.subplot(121)
plt.imshow(img4[:, :, ::-1])

plt.subplot(122)
plt.imshow(edges4,cmap = 'gray')

plt.savefig("out/LineasHough_Comparativa.png", dpi=600)
plt.show()
```

---

3. Lo cual producirá el siguiente resultado:



## 2.3. Detectar Círculos

Otro de los usos para el Algoritmo de Hough es la detección círculos. En este caso, como para la línea, se hace necesario expresar el círculo mediante su ecuación:

$$(x - a)^2 + (y - b)^2 = r^2 \quad (7)$$

Donde  $(a, b)$  son el centro del círculo y  $r$  el radio del mismo. En este caso se utilizan acumuladores  $3D$  y cada punto en la imagen vota por el círculo en los que pudiera estar y al final el algoritmo busca los picos en el acumulador para determinar el radio y centro. En caso de que se conozca el radio solo sería necesario utilizar un acumulador de  $2D$ .

En este caso OpenCv cuenta con la función `cv2.HoughCircles()` para la detección de círculos en imágenes. Para más información puede consultar el siguiente enlace<sup>[6](#)</sup> <sup>[7](#)</sup>

Los parámetros de la función `cv2.HoughCircles()` son los siguientes:

- **image:** Imagen en escala de grises de un solo canal.
- **method:** Define el método para detectar círculos en las imágenes. Hasta ahora el único método implementado es el `cv2.HOUGH_GRADIENT` que esta expresado en el articulo [3].
- **dp:** Este parámetro es inversamente proporcional al acumulador de resolución de la imagen. Entre más grande el valor de `dp`, menos arreglos obtiene el acumulador.
- **minDist:** Distancia mínima entre las coordenadas  $(a, b)$  del centro del círculo detectado. Si `minDist` es muy pequeña, se detectaran múltiples círculos en el mismo vecindario y el original podría ser falsamente detectado. Si es muy grande, algunos círculos no serán detectados.
- **param1:** Valor del gradiente usado para manejar el detector de bordes, como se explica en [3]
- **param2:** Umbral del acumulador para el método de `cv2.HOUGH_GRADIENT`. Un umbral pequeño detecta más círculos (incluidos falsos positivos). Un Umbral grande una mayor cantidad de potenciales círculos serán regresados.
- **minRadius:** Mínimo valor del radio (en píxeles).
- **maxRadius:** Máximo valor del radio (en píxeles).

1. Como primer paso crea un *script/notebook* y añade las importaciones a OpenCV, `Numpy(np)` y `Pyplot(plt)` de Matplotlib.

---

<sup>6</sup>Hough Círculos

<sup>7</sup>Hough Círculos 2

2. Seguidamente copiamos el siguiente código que implementa Hough, muestra y almacena el resultado en disco.
- 

```
imgIn = cv2.imread('img/coin.jpg')

h,w,c = imgIn.shape
# Nuevo Tamaño
dsizes = (int(w*0.250), int(h*0.250))
# resize image
imgCir = cv2.resize(imgIn, dsizes)

src = cv2.medianBlur(imgCir, 5)
src = cv2.cvtColor(src, cv2.COLOR_BGR2GRAY)

circles = cv2.HoughCircles(src, cv2.HOUGH_GRADIENT, 1, 20,
                           param1=50, param2=80, minRadius=0, maxRadius=0)

circles = np.uint16(np.around(circles))
for i in circles[0,:]:
    # dibujar circulo
    cv2.circle(imgCir, (i[0], i[1]), i[2], (0,255,0), 2)
    # dibujar centro
    cv2.circle(imgCir, (i[0], i[1]), 2, (0,0,255), 3)

cv2.imwrite("out/circulos.png", imgCir)

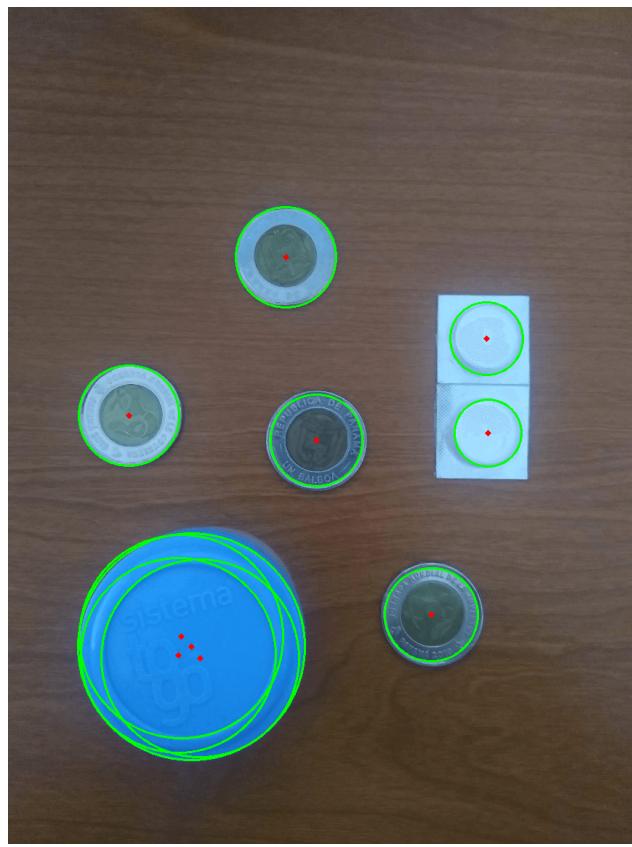
plt.imshow(imgCir[:, :, ::-1])
```

---

3. Lo cual producirá el siguiente resultado:

### 2.3 Detectar Círculos

### Visión Artificial / Laboratorio 4



### Entregables para la Sección

Para esta sección en su informe debe incluir la modificación de una imagen de su propiedad. En dicha imagen se deben realizar las siguientes operaciones:

1. El código mostrado en la Sección [2.1](#) muestra solo una línea detectada en la imagen, realice la modificación necesaria en este para que se muestren todas líneas detectadas. Adicional pinte cada una de las líneas de un color elegido aleatoriamente entre 10 posibles colores. Para esto debe utilizar la imagen `sudo.PNG`.
2. Utilizando el archivo `monedas.jpg` aplique el algoritmo de Hough para detectar la mayor cantidad de círculos en la imagen. Pinte cada círculo detectado con un contorno azul e indique su centro en color amarillo. Recorte cada círculo detectado en la imagen y muestre el resultado en una matriz, utilizando `subplots` de `matplotlib`, se recomienda utilizar un número de columnas predefinido y un número de filas dinámico.

## Referencias

- [1] C. Harris and M. Stephens, "A combined corner and edge detector," in *In Proc. of Fourth Alvey Vision Conference*, 1988, pp. 147–151.
- [2] Jianbo Shi and Tomasi, "Good features to track," in *1994 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 1994, pp. 593–600.
- [3] H. Yuen, J. Princen, J. Illingworth, and J. Kittler, "Comparative study of hough transform methods for circle finding," *Image and Vision Computing*, vol. 8, no. 1, pp. 71 – 77, 1990. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/026288569090059E>