Trabajo Práctico N° 2

Procesamiento de Imágenes I

Tecnicatura Universitaria en Inteligencia Artificial

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Universidad Nacional de Rosario

2023

Integrantes:

Augusto Farias,

Guido Lorenzetti,

Micaela Pozzo,

Patricio Vercesi

PROBLEMA 1 – Detección y clasificación de monedas y dados

a) Procesar la imagen de manera de segmentar las monedas y los dados de manera automática.

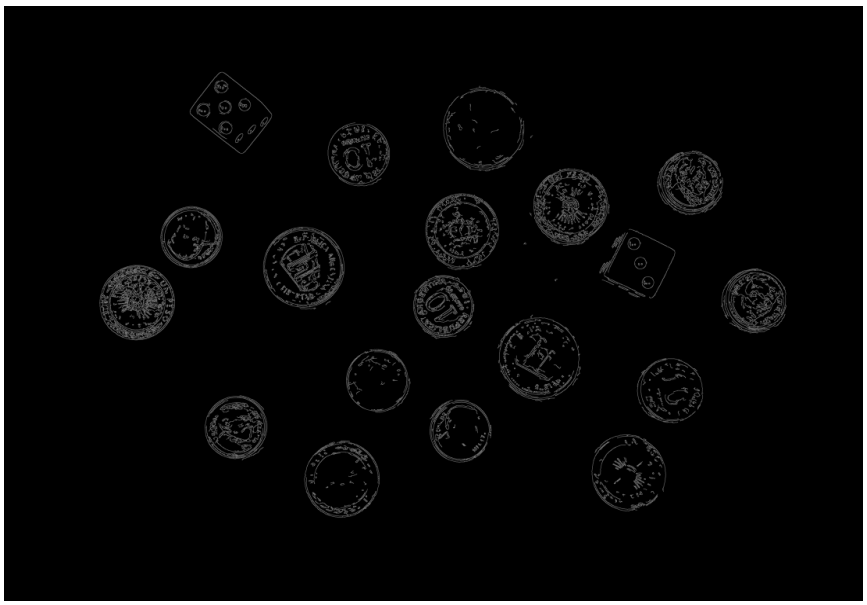
- Leemos la imagen de dados y monedas

```
img= cv2.imread("monedas.jpg")

img_color = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

plt.imshow(img_color), plt.show(block = False)
```

- Pasamos imagen a escala de grises, pasamos un filtro pasabajos y aplicamos canny para la detección de bordes.

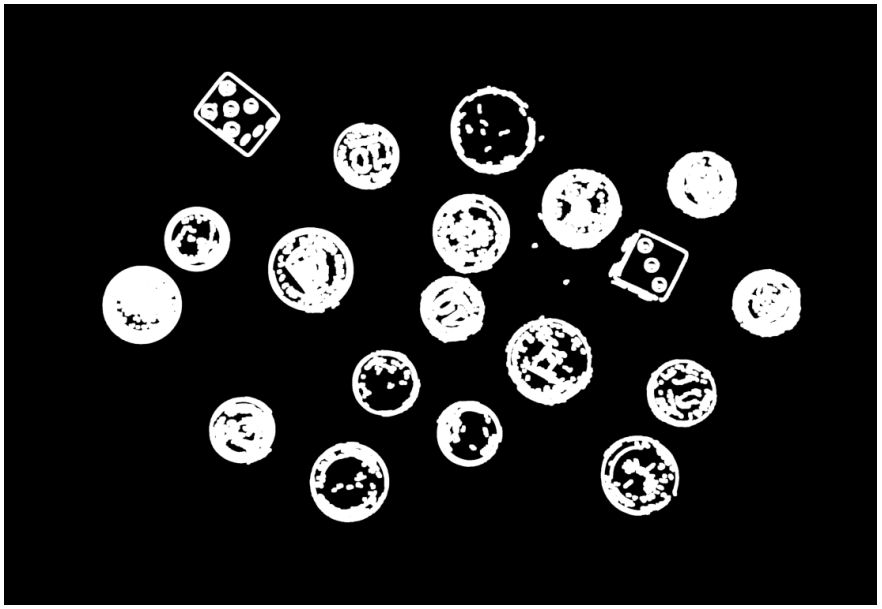
- Los parámetros fueron elegidos mediante testeos hasta lograr un resultado óptimo

```
img_gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

img_filtered = cv2.medianBlur(img_gray, 7)

img_canny_CV2 = cv2.Canny(img_filtered, 35, 110)

plt.imshow(img_canny_CV2, cmap="gray"), plt.show(block = False)
```



- Aplicamos dilatación para engrosar bordes

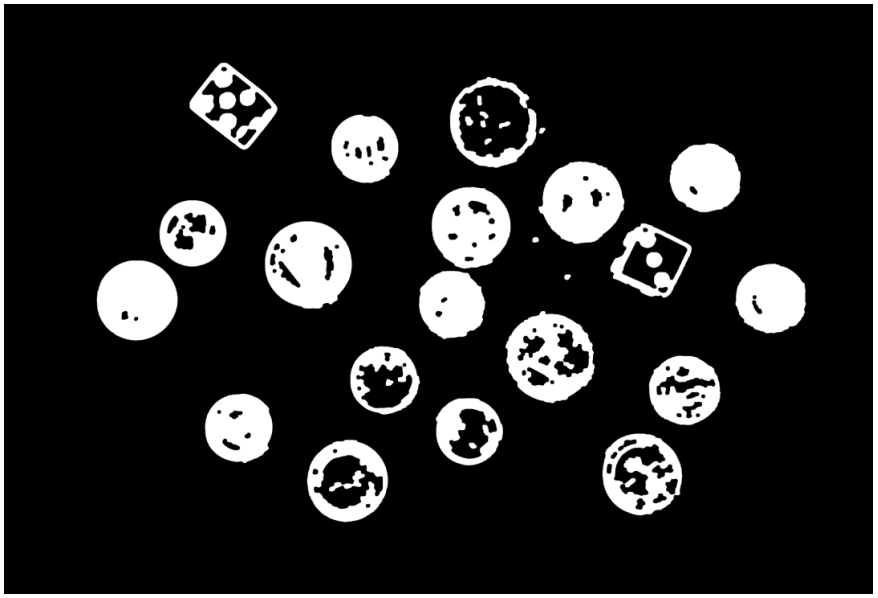
```
img_dilated = cv2.dilate(img_canny_CV2, cv2.getStructuringElement(cv2.MORPH_ELLIPSE,
(12, 12)), iterations=1)

img_dilated = cv2.dilate(img_dilated, cv2.getStructuringElement(cv2.MORPH_RECT, (5,
5)), iterations=1)

plt.imshow(img_dilated, cmap = "gray"), plt.show(block=False)
```



- Observamos que hay algunas monedas que no terminan de completar la figura con sus bordes, por lo cual aplicamos una apertura y clausura

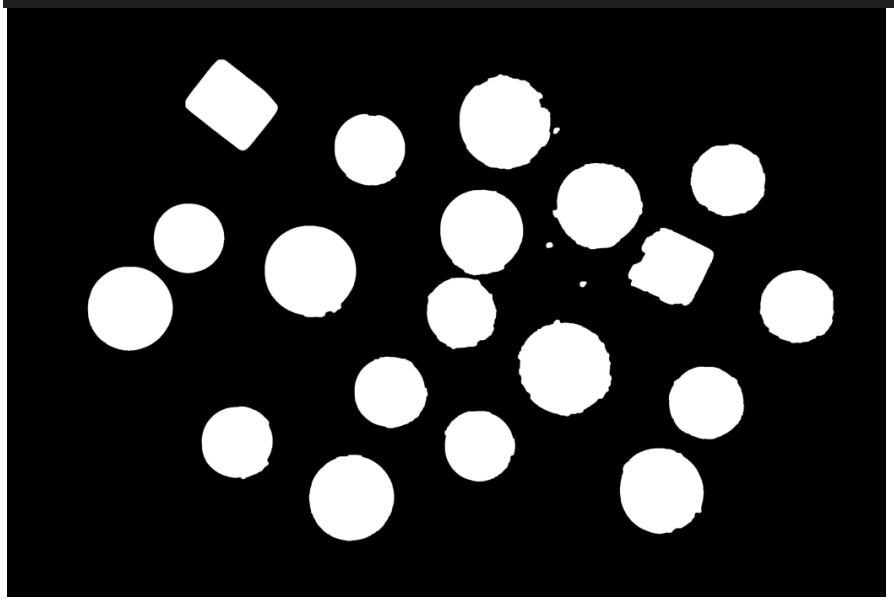
```
img_open = cv2.morphologyEx(img_dilated, cv2.MORPH_OPEN,
cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (11, 11)))
```

```
img_close = cv2.morphologyEx(img_open, cv2.MORPH_CLOSE,
cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (16, 16)))

plt.imshow(img_close, cmap="gray"), plt.show(block= False)
```



- Podemos observar, ahora si, que las figuras están completas y cerradas, ahora podemos aplicar la función imfillhole() para rellenar las figuras.

```
img_fh = imfillhole_v2(img_close)

plt.imshow(img_fh, cmap = "gray"), plt.show(block=False)
```



- Utilizamos componentes conectadas para obtener las stats de las figuras y poder analizar factor de forma y área de las figuras.

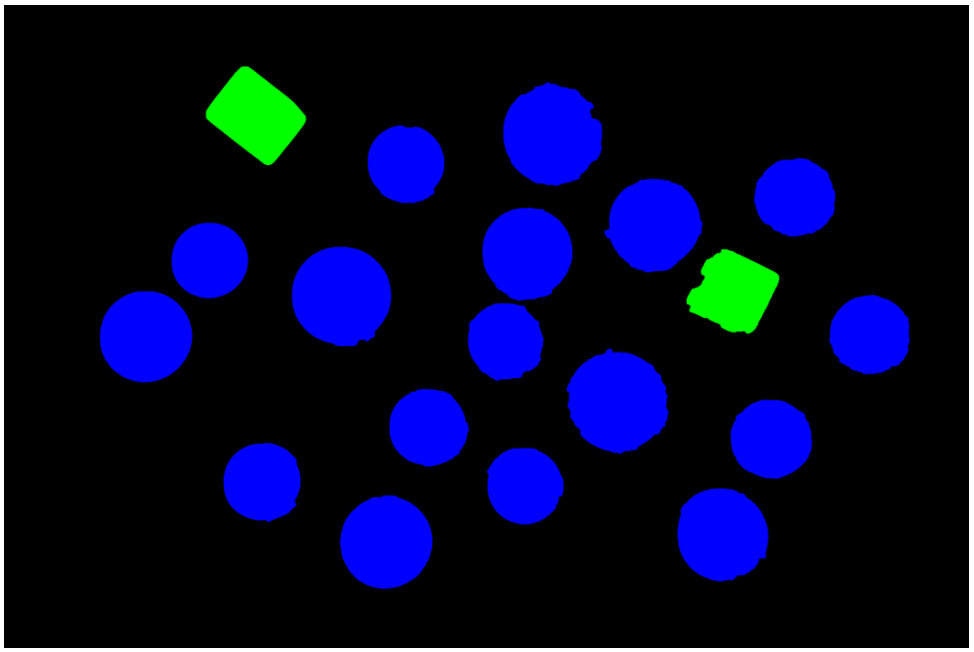
```
num_labels, labels, stats, centroids = cv2.connectedComponentsWithStats(img_fh, 4,
cv2.CV_32S)
```

```
# --- Defino parametros para la clasificación
----------------------------------------

RHO_TH = 0.8    # Factor de forma (rho), si es circulo el valor es mayor a 0.8

AREA_TH = 5000   # Umbral de area para descartar los labels que no sean figuras

aux = np.zeros_like(labels)

labeled_image = cv2.merge([aux, aux, aux])
```

```
# --- Clasificación
-----------------------------------------------------------------

# Clasifico en base al factor de forma

for i in range(1, num_labels):

    # --- Remuevo celulas con area chica -------------------------------------

    if (stats[i, cv2.CC_STAT_AREA] < AREA_TH):

        continue

    # --- Selecciono el objeto actual ----------------------------------------

    obj = (labels == i).astype(np.uint8)

    # --- Calculo Rho --------------------------------------------------------

    ext_contours, _ = cv2.findContours(obj, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

    area = cv2.contourArea(ext_contours[0])

    perimeter = cv2.arcLength(ext_contours[0], True)

    rho = 4 * np.pi * area/(perimeter**2)

    flag_circular = rho > RHO_TH

    # --- Calculo cantidad de huecos -----------------------------------------

    all_contours, _ = cv2.findContours(obj, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)

    holes = len(all_contours) - 1

    # --- Clasifico ----------------------------------------------------------

    if rho > RHO_TH:

        labeled_image[obj == 1, 2] = 255
```

```
    else:
        labeled_image[obj == 1, 1] = 255
plt.figure(); plt.imshow(labeled_image); plt.show(block=False)
```

- Podemos observar la segmentacion de los dados y monedas



b) Clasificar los distintos tipos de monedas y realizar un conteo, de manera automática.

```
# Convertir a profundidad de 8 bits por canal
monedas_8u = labeled_image[:, :, 2].astype(np.uint8)
# Aplicar umbral
_, monedas_binary = cv2.threshold(monedas_8u , 15, 255, cv2.THRESH_BINARY)
plt.imshow(monedas_binary, cmap= "gray"), plt.show(block=False)
num_labels, labels, stats, centroids =
cv2.connectedComponentsWithStats(monedas_binary, 4, cv2.CV_32S)
# Definir los rangos de áreas para cada tipo de moneda
area_rangos = {
    '10_cent': (0, 85000),
    '1_peso': (85000, 105000),
    '50_cent': (105000, 300000)
}
```

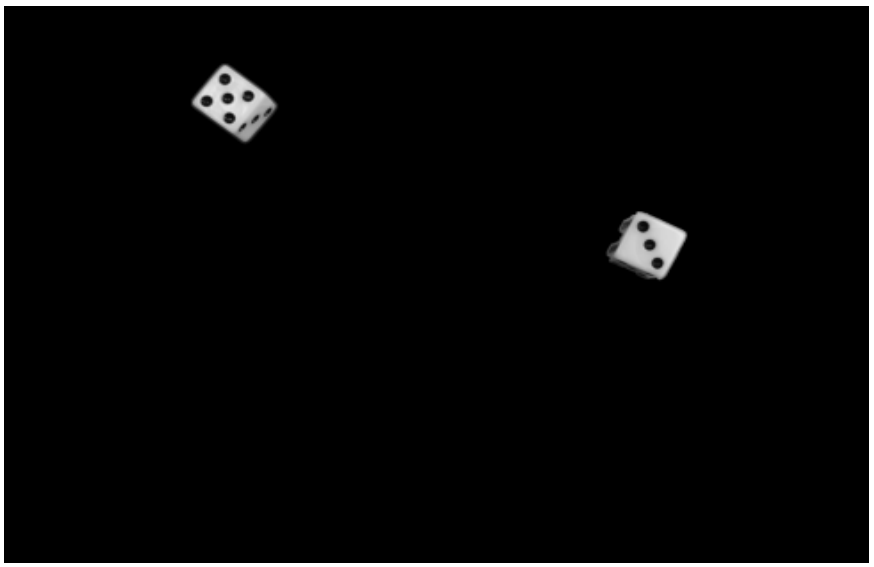
```
# Crear una lista para almacenar el tipo de cada moneda
tipos_monedas = []
cant_monedas = {"10 centavos": 0,"50 centavos": 0, "1 peso": 0}
# Clasificar cada moneda según su área
for i in range(1, stats.shape[0]):
    area = stats[i, cv2.CC_STAT_AREA]
    # Comparar el área con los rangos definidos
    if area_rangos['10_cent'][0] <= area <= area_rangos['10_cent'][1]:
        tipo = '10 CENT'
        cant_monedas["10 centavos"] +=1
    elif area_rangos['1_peso'][0] <= area <= area_rangos['1_peso'][1]:
        tipo = '1 PESO'
        cant_monedas["1 peso"] +=1
    elif area_rangos['50_cent'][0] <= area <= area_rangos['50_cent'][1]:
        tipo = '50 CENT'
        cant_monedas["50 centavos"] +=1
    else:
        tipo = 'No Clasificado'
    tipos_monedas.append([tipo, i])
# Crear una copia de la imagen original para agregar etiquetas
etiquetas_image = img_color.copy()
# Configurar la fuente y otros parámetros del texto
font = cv2.FONT_HERSHEY_SIMPLEX
escala = 2.5
color = (0, 0, 0)  # Color del texto en blanco
# Agregar texto para cada tipo de moneda en la posición central
for tipo, i in tipos_monedas:
    centro = (int(centroids[i, 0]), int(centroids[i, 1]))
    cv2.putText(etiquetas_image, tipo, centro, font, escala, color, 2, cv2.LINE_AA)
# Agregar texto con la cantidad de monedas por tipo
```

```
posicion_y = 2300

for tipo, cantidad in cant_monedas.items():

    texto_cantidad = f'{tipo}: {cantidad} monedas'

    cv2.putText(etiquetas_image, texto_cantidad, (30, posicion_y), font, 3, color,
2, cv2.LINE_AA)

    posicion_y += 90  # Ajusta el espacio vertical entre líneas según tus
    preferencias

plt.imshow(etiquetas_image), plt.show(block=False)
```

c) Determinar el número que presenta cada dado mediante procesamiento automático

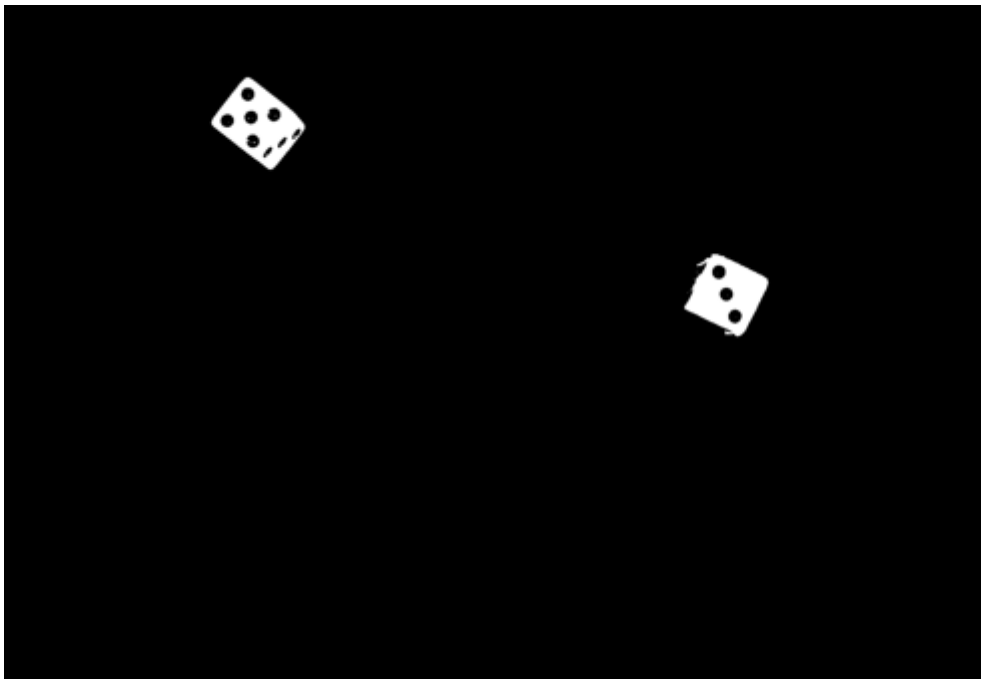
-Leemos la imagen filtrada donde quedan solo los dados y trabajamos en esta imagen.

```
image = cv2.imread("dadostp2.jpg", cv2.IMREAD_COLOR)      # Cargo imagen
plt.figure(), plt.imshow(image), plt.show(block=False)
```



-Pasamos la imagen original a escala de grises, binarizamos, definimos el kernel y visualizamos lo obtenido.

```
#Imagen filtrada gray scale
img_gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
plt.imshow(img_gray, cmap="gray"), plt.show(block = False)

# --- Binarizo ---------------------------------------------------------------------------------
th, binary_img = cv2.threshold(img_gray, 125, 1, cv2.THRESH_OTSU)
plt.figure(), plt.imshow(binary_img, cmap='gray'), plt.show(block=False)

kernel_size = 7  # Tamaño del kernel de mediana
img_filtered = cv2.medianBlur(img_gray, kernel_size)
```



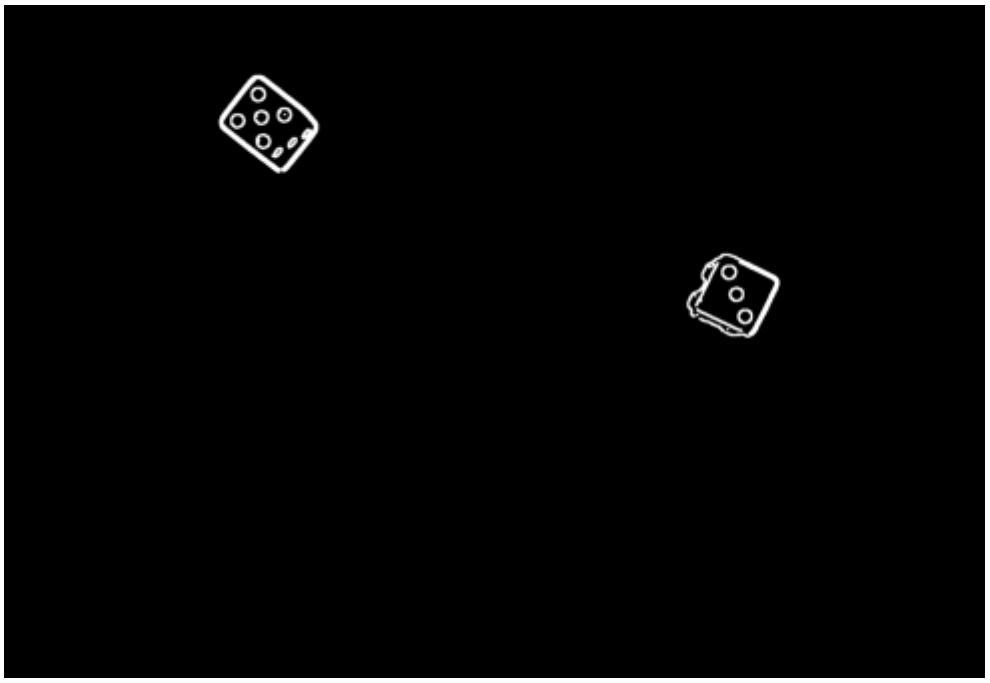
-Aplicamos operaciones morfológicas como Canny y fuimos variando el umbral hasta conseguir uno óptimo. Visualizamos lo obtenido.

```
#Canny
img_canny_CV2 = cv2.Canny(img_filtered, 150, 150)#, apertureSize=3, L2gradient=True)
plt.imshow(img_canny_CV2, cmap="gray"), plt.show(block = False)
```

-Aplicamos dilatación en el resultado obtenido probando con varios umbrales y visualizamos.

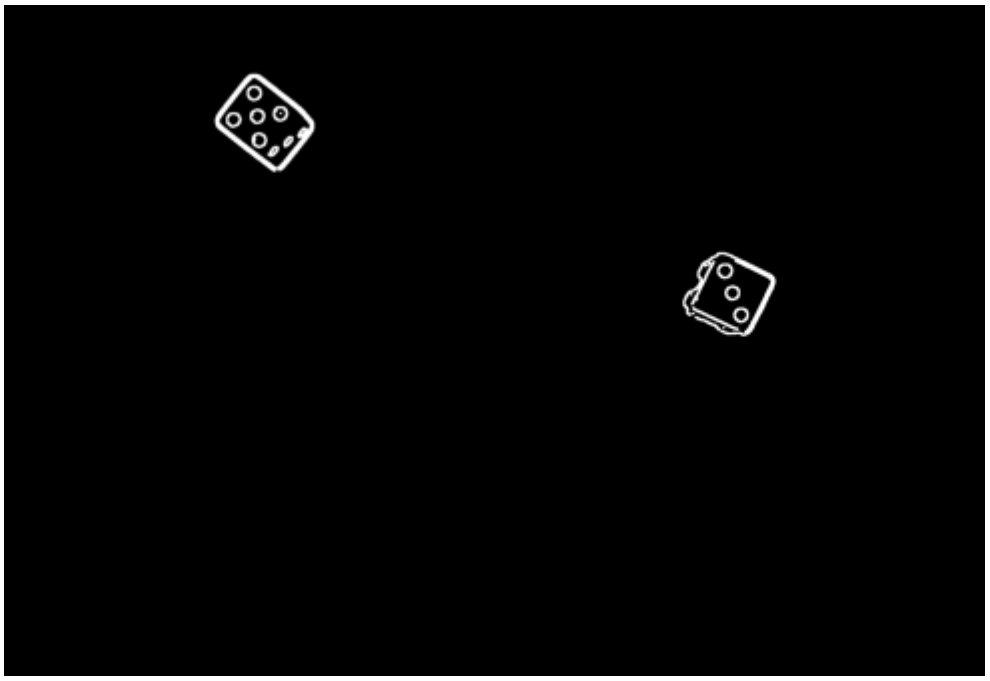
```
#Dilato
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (4, 4))
Fd = cv2.dilate(img_canny_CV2, kernel, iterations=1)
plt.imshow(Fd, cmap = "gray"), plt.show(block=False)
```

✓ 1.8s



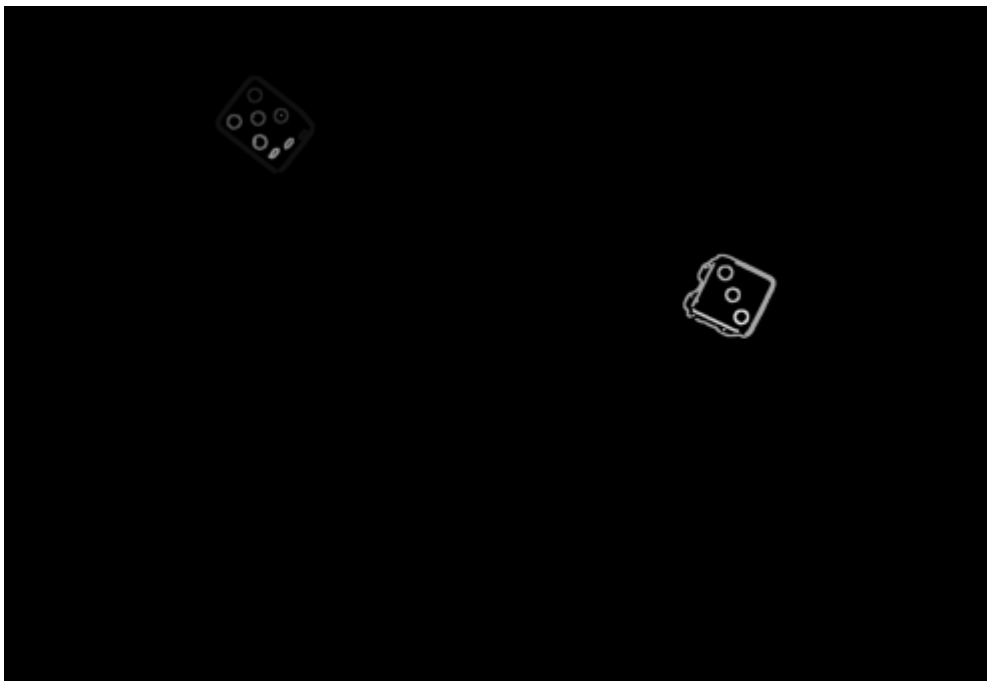
- Aplicaciones operaciones morfológicas como la apertura y clausura para mejorar la imagen obtenida anteriormente.

```
# --- Operaciones morfológicas para mejorar la segmentación obtenida ------------------------------
se = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
binary_img = cv2.morphologyEx(binary_img, cv2.MORPH_OPEN, se)   # Apertura para remover elementos pequeños
binary_img = cv2.morphologyEx(binary_img, cv2.MORPH_CLOSE, se)  # Clausura para rellenar huecos.
plt.figure(), plt.imshow(Fd, cmap='gray'), plt.show(block=False)
```



-Obtenemos las componentes conectadas con la función `connected Components With Stats`

```
# --- Componentes conectadas ----------------------------------------------------------------
num_labels, labels, stats, centroids = cv2.connectedComponentsWithStats(Fd)
plt.figure(), plt.imshow(labels, cmap='gray'), plt.show(block=False)              # Visualizo los objetos con sus labels en ESCALA

labels_color = np.uint8(255/(num_labels-1)*labels)                # Llevo el rango de valores a [0 255] para diferenciar mejor los
# np.unique(labels_color)                                         # Por si quieren verificar los valores asignados....
im_color = cv2.applyColorMap(labels_color, cv2.COLORMAP_JET)
im_color = cv2.cvtColor(im_color, cv2.COLOR_BGR2RGB)              # El mapa de color que se aplica está en BGR --> convierto a RGB
plt.figure(), plt.imshow(im_color), plt.show(block=False)
```



- Defino el umbral del factor de forma del círculo y el área

```
# --- Defino parametros para la clasificación ----------------------------------------
RHO_TH = 0.8    # Factor de forma (rho)
AREA_TH = 500   # Umbral de area
aux = np.zeros_like(labels)
labeled_image = cv2.merge([aux, aux, aux])
```

- Iteramos en las labels, removemos área más chica que el umbral definido anteriormente.
- Obtenemos los contornos de la imagen
- Calculamos Rho del círculo , área y perímetro. Luego volvemos a clasificar el objeto y contamos la cantidad de puntos que tiene cada dado a partir del contorno.

```
labeled_shapes = np.zeros_like(im_color)
# Clasifico en base al factor de forma
for i in range(1, num_labels):

    # --- Remuevo celulas con area chica ---------------------------------------
    if (stats[i, cv2.CC_STAT_AREA] < AREA_TH):
        continue

    # --- Selecciono el objeto actual ------------------------------------------
    obj = (labels == i).astype(np.uint8)

    # --- Calculo Rho ----------------------------------------------------------
    ext_contours, _ = cv2.findContours(obj, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    area = cv2.contourArea(ext_contours[0])
    perimeter = cv2.arcLength(ext_contours[0], True)
    rho = 4 * np.pi * area/(perimeter**2)
    flag_circular = rho > RHO_TH
```

```python
    # --- Selecciono el objeto actual ---------------------------------------
    obj = (labels == i).astype(np.uint8)

    # --- Calculo Rho -------------------------------------------------------
    ext_contours, _ = cv2.findContours(obj, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    area = cv2.contourArea(ext_contours[0])
    perimeter = cv2.arcLength(ext_contours[0], True)
    rho = 4 * np.pi * area/(perimeter**2)
    flag_circular = rho > RHO_TH

    # --- Clasifico ---------------------------------------------------------

    if flag_circular:
        print("es una moneda")
    else:
        # --- Calculo cantidad de puntos ------------------------------------

        all_contours, _ = cv2.findContours(obj, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
        holes = len(all_contours) - 1
        print(f" Es un dado y tiene {holes} puntos")
```

Finalmente, visualizamos el resultado obtenido.

```
Es un dado y tiene 5 puntos
Es un dado y tiene 3 puntos
```

PROBLEMA 2 – Detección de patentes

Ambos apartados están resueltos en el archivo ej2.py, que está compuesto de tres funciones: detect_patent, detect_characters, y show_full_patents_detected. El código al ser ejecutado llamará a show_full_patents_detected que llamará a las otras 2 una vez por cada archivo. Primero llamará a detect_patent que realizará la detección de patentes pedida en el punto a, y luego sobre esa respuesta llamará a detect_characters para encontrar los caracteres dentro de la supuesta patente, realizando el punto b.

Aclaramos que son las supuestas patentes porque la función de detectar patentes detecta otros patrones en la imagen que no son patentes, pero como la función de detectar caracteres no encontrará ninguno no la mostrará. Lo único que queda importante a resaltar de la función “raíz” es que sobre cada posible patente, si no consigue encontrar 6 caracteres al llamar por primera vez a detect_characters, lo llama una segunda con un argumento de un threshold distinto, que en varios casos hace la diferencia:

```
        for patent in possible_patents:
            characters = detect_characters(patent, filename, 140)
            if len(characters) != 6:
                characters = detect_characters(patent, filename, 120)
```

a) Detección de patentes

Volviendo a la función de detectar patentes que se corre antes, primero se pasa de BGR a HSV para encontrar los blancos de las patentes conservando la información de la profundidad de color, ignorando las Matices de color y clasificando en blancos si tienen muy poca Saturación y Valor medio-alto.

```
    # Convert the image from BGR to HSV
    hsv_image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

    # Define lower and upper bounds for white color in HSV format
    lower_white = np.array([0, 0, 120], dtype=np.uint8)
    upper_white = np.array([180, 75, 255], dtype=np.uint8)
    # Create a mask to identify white pixels within the specified HSV range
    white_mask = cv2.inRange(hsv_image, lower_white, upper_white)
```

Después usamos el canal del valor que indica cuan blanco vs negro es un pixel para pasar la imagen a escala de grises, detectamos bordes con un filtro Laplaciano sobre el blur Gaussiano, y hacemos una apertura y cerradura morfológica, porque en la práctica testeando esos métodos en ese orden nos dieron los mejores resultados para separar los componentes conectados que estábamos buscando de los demás píxeles algo blancos sin que se nos erosionen demasiado.

```
    # Convert the Value channel to grayscale
    gray = whitish_pixels[:, :, 2]

    # Laplacian of Gaussian
    blur = cv2.GaussianBlur(gray, (3,3), 0)
    LoG = cv2.Laplacian(blur, cv2.CV_64F, ksize=3)
    LoG_abs = cv2.convertScaleAbs(LoG)    # Pasamos a 8 bit
    LoG_abs_th = LoG_abs > LoG_abs.max()*0.45
    filtered = LoG_abs_th.astype(np.uint8) * 255
```

```
    # Morphological Opening & Closing
    B = cv2.getStructuringElement(cv2.MORPH_RECT, (2, 2))
    filtered = cv2.morphologyEx(filtered, cv2.MORPH_OPEN, B)
    filtered = cv2.morphologyEx(filtered, cv2.MORPH_CLOSE, B)
```

Y por último encontramos las componentes conectadas, filtramos las que no estén cerca de los bordes, y que tengan cierto tamaño y ratio que deberían tener como patentes y las retornamos.

```
num_labels, labels, stats, centroids = cv2.connectedComponentsWithStats…
for st in stats:
    x, y, w, h, area = st
    ratio = w / h
    if (1 < ratio < 3) and (65 < w < 110) and (20 < x < 620) and (20 < y…:
          possible_patents.append(image[y-5:y+h+5, x-5:x+w+5])
    return possible_patents
```

Los resultados fueron relativamente buenos y seguro pueden ser mejorados. Todas las patentes detectadas por las componentes conectadas pasaron el filtro pero eso excluyó a dos imágenes que no pudimos separar de sus autos correctamente, la 7 y la 12.

b) Detección de caracteres

El proceso es casi el mismo que con las patentes hasta la escala de grises. Donde se binariza la imagen en vez de filtrarla con Laplace, Gauss y morfología.

```
binary = cv2.threshold(gray, 200, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
```

Luego las componentes conectadas contenían a la mayoría de caracteres (gracias a cambiar el V_threshold) y fue relativamente sencillo separarlo del resto de componentes conectadas. Lo cual en este caso era 100% necesario por tratarse de la segmentación final.

```
    for st in stats:
        x, y, w, h, area = st
        ratio = w / h
        if (0.3 < ratio < 1) and (10 < h):
            characters.append((patent[y-2:y+h+2, x-2:x+w+2], x))
```

Por último, antes de devolver los caracteres, los ordenamos según su posición en el eje x para ser que estén ordenados en el plot.

```
        characters = sorted(characters, key=lambda x: x[1])
        characters = [x[0] for x in characters]
        return characters
```

Al final hubo dos imágenes de patentes de las 10 que teníamos las cuales la función no pudo encontrar los 6 caracteres, quedándonos entonces con un total de 8 patentes completas identificadas.

Mostramos un ejemplo:

Imagen original:

Patente segmentada:

Caracteres de la patente: