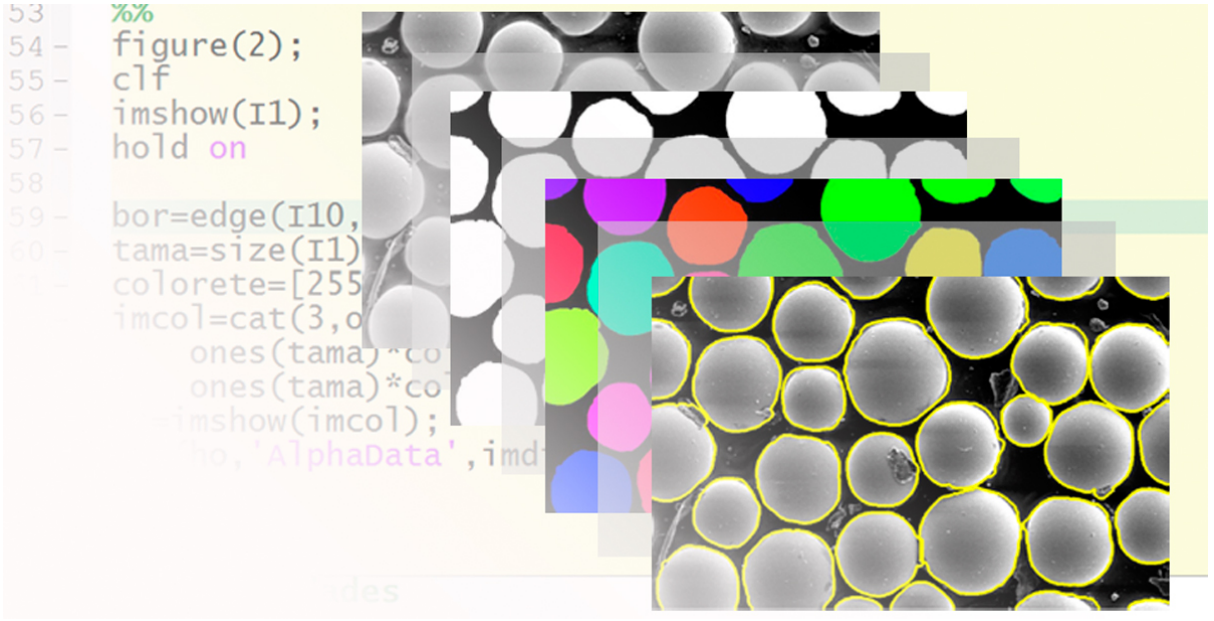
Trabajo Práctico Nº2
Procesamiento de Imágenes 1
TUIA



GRUPO 7

Integrantes:

- **Ferrari, Enzo**
- **Rodriguez, Abril Nazarena**
- **Ferrero, Santiago**
- **Loza, Santiago**

ÍNDICE

INTRODUCCIÓN

En el presente informe se detallan los pasos y los conceptos aplicados para la resolución del **Trabajo Práctico N°2** de la asignatura, el cual consta de dos ejercicios.

En el primero de ellos, el propósito es llevar a cabo un procesamiento en la imagen titulada **'monedas.jpg'** mediante diversas técnicas con el objetivo de segmentar dados y monedas. Posteriormente, se procederá a clasificar cada objeto según su tamaño (en el caso de las monedas) y su valor (en el caso de los dados).

En el segundo ejercicio, nos enfocaremos en identificar patentes en 12 imágenes distintas y extraer los caracteres correspondientes.

Al final del documento, se encontrarán las conclusiones generales que nos dejó la resolución del trabajo.

Dependencias:

Los que utilizamos para resolver el trabajo son Visual Studio Code y Google Colab. En cuanto a las librerías, son las siguientes.

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

Se cuenta con:

- **cv2**: OpenCV, para manipular las imágenes.
- **numpy:** ofrece funciones y métodos para manipular arrays.
- **matplotlib**: para realizar gráficas y visualizar imágenes.

RESOLUCIÓN EJERCICIO 1

Como hemos mencionado, el ejercicio consiste en analizar la imagen **‘monedas.jpg’**, que incluye monedas de diferentes valores y dados. El objetivo principal es llevar a cabo la segmentación de ambos objetos. En el caso de las monedas, se busca identificarlas y clasificarlas según su tamaño, mientras que en el caso de los dados, se pretende determinar el número que muestran.

Comenzamos definiendo la función **‘imshow’** la cual utilizamos en ambos ejercicios. Esta función recibe una imagen y la muestra por pantalla. Nos resulta muy útil para visualizar qué está pasando en cada etapa de procesamiento de la imagen y facilita la toma de próximas decisiones (Imagen 1).

```
def imshow(img, new_fig=True, title=None, color_img=False, blocking=False, colorbar=True, ticks=False):
    if new_fig:
        plt.figure()
    if color_img:
        plt.imshow(img)
    else:
        plt.imshow(img, cmap='gray')
    plt.title(title)
    if not ticks:
        plt.xticks([]), plt.yticks([])
    if colorbar:
        plt.colorbar()
    if new_fig:
        plt.show(block=blocking)
```

Imagen 1. Función para mostrar imágenes

Cargamos la imagen en la variable **‘f’** en escala de grises y la visualizamos con ayuda de la función anteriormente definida. (Imagen 2)

```
f = cv2.imread('monedas.jpg', cv2.IMREAD_GRAYSCALE)
imshow(f)
```

Imagen 2. Carga y visualización de la imagen en escala de grises.

Definimos un filtro pasa bajo, con un kernel de tamaño 2x2. Luego, aplicamos este filtro a nuestra imagen **‘f’** mediante **‘cv2.filter2D’**. Almacenamos el resultado en la variable **‘img1’**. (Imagen 3)

```
w1 = np.ones((2,2))/(2*2)
img1 = cv2.filter2D(f,-1,w1)
imshow(img1)
```

Imagen 3. Aplicamos filtro pasa bajo.

Realizamos un suavizado utilizando un filtro Gaussiano mediante **‘cv2.GaussianBlur’** con un kernel de tamaño 7x7 y un sigma (desviación estándar) de 1. Luego,aplicamos **‘cv2.Canny’** con umbrales **‘threshold1’** y **‘threshold2’**. Almacenamos el resultado en la variable **‘gcan’**. (Imagen 4)

```
f_blur = cv2.GaussianBlur(img1, ksize=(7, 7), sigmaX=1)
gcan = cv2.Canny(f_blur, threshold1=0.12*255, threshold2=0.62*255)
imshow(gcan)
```

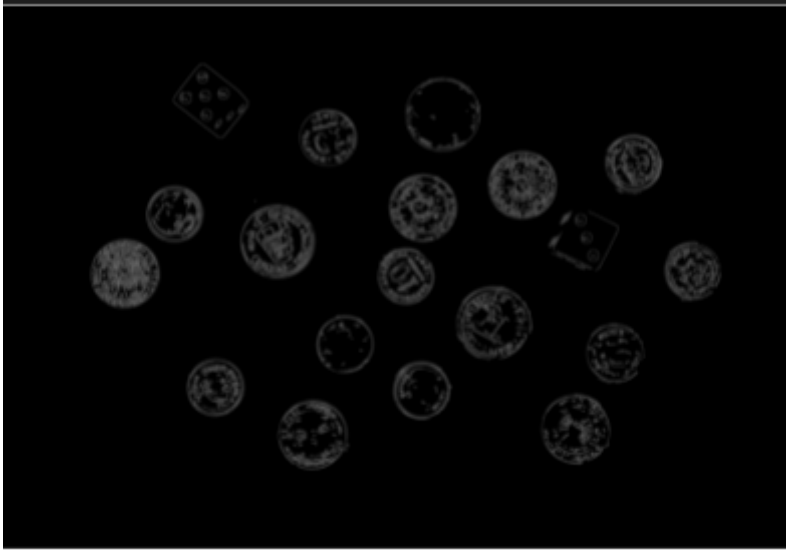


Imagen 4. Detectamos bordes con Canny.

A continuación, aplicaremos dilatación a nuestra imagen **‘gcan’,** ya que aplicando la misma resaltaremos y expandimos las regiones de píxeles blancos de nuestra imagen binaria. Esto nos favorece para cerrar los contornos externos de las monedas y dados.

Creamos un kernel estructurante eliptico utilizando **‘cv2.getStructuringElement’**. Luego, aplicamos dilatación mediante ‘cv2.dilate’ con el kernel definido y una sola iteración. Almacenamos el resultado en la variable **‘Fd’**. (Imagen 5)

```
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (36,36))
Fd = cv2.dilate(gcan, kernel, iterations=1)
imshow(Fd)
```

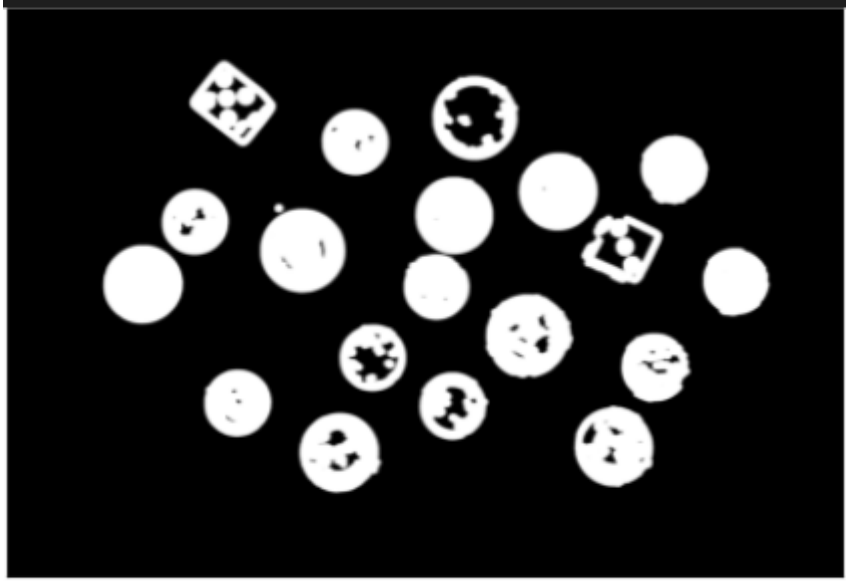


Imagen 5. Dilatamos con un kernel de 36x36

Definimos dos funciones:

- **‘imreconstruct’** : Esta función realiza la operación de reconstrucción morfológica, que es útil para ampliar regiones en una imagen. Toma un marcador y una máscara como entrada y dilata iterativamente el marcador hasta que no hay cambios en la intersección con la máscara. El resultado es la región reconstruida. (Imagen 6)
- **‘imfillhole’**: Esta función se utiliza para rellenar huecos en una imagen binaria. Toma una imagen binaria como entrada, identifica los bordes, y utiliza la operación de reconstrucción **‘imreconstruct’** para llenar los huecos en la imagen original. El resultado es una imagen binaria con los huecos rellenados. (Imagen 6)

```python
def imreconstruct(marker, mask, kernel=None):
    if kernel==None:
        kernel = np.ones((3,3), np.uint8)
    while True:
        expanded = cv2.dilate(marker, kernel)                          # Dilatacion
        expanded_intersection = cv2.bitwise_and(src1=expanded, src2=mask)   # Interseccion
        if (marker == expanded_intersection).all():                    # Finalizacion?
            break                                                      #
        marker = expanded_intersection
    return expanded_intersection


#rellenamos
def imfillhole(img):
    # img: Imagen binaria de entrada. Valores permitidos: 0 (False), 255 (True).
    mask = np.zeros_like(img)                                                  # Genero mascara
    mask = cv2.copyMakeBorder(mask[1:-1,1:-1], 1, 1, 1, 1, cv2.BORDER_CONSTANT, value=int(255)) #
    marker = cv2.bitwise_not(img, mask=mask)                # El marcador lo defino como el compl
    img_c = cv2.bitwise_not(img)                            # La mascara la defino como el comple
    img_r = imreconstruct(marker=marker, mask=img_c)        # Calculo la reconstrucción R_{f^c}(f
    img_fh = cv2.bitwise_not(img_r)                         # La imagen con sus huecos rellenos e
    return img_fh
```

Imagen 6. Definimos dos funciones de reconstrucción morfológica

Aplicamos **‘imfillhole’** a nuestra imagen **‘Fd’**. (Imagen 7)

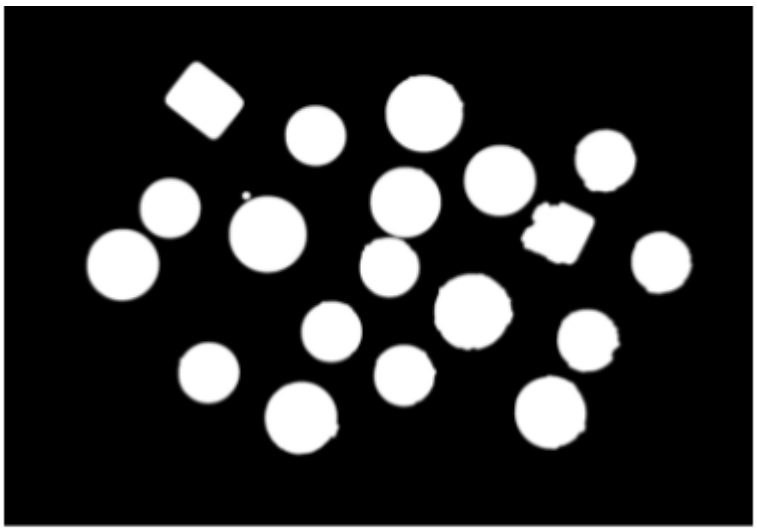


Imagen 7. Resultado luego de la reconstrucción morfológica

Aplicaremos apertura morfológica en la imagen binaria dilatada **‘Fd’,** para eliminar pequeños detalles.

Primero, creamos un elemento estructurante elíptico **‘B’** mediante **‘cv2.getStructuringElement’**. Luego, se aplica la operación de apertura **‘cv2.morphologyEx’** a la imagen dilatada utilizando el elemento estructurante **‘B’** . Almacenamos el resultado en la variable **‘Aop’**(Imagen 8)

```
B = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (50, 50))
Aop = cv2.morphologyEx(Fd, cv2.MORPH_OPEN, B)
imshow(Aop)
```

Imagen 8. Resultado luego de aplicar apertura

Aplicamos **‘cv2.connectedComponentsWithStats’** a nuestra imagen para poder identificar las componentes conectadas de la misma. Luego, graficamos los resultados y podemos observar los objetos detectados.(Imagen 9)

```
num_labels, labels, stats, centroids = cv2.connectedComponentsWithStats(Aop)

labels_color = np.uint8(255/(num_labels-1)*labels)
# np.unique(labels_color)
im_color = cv2.applyColorMap(labels_color, cv2.COLORMAP_JET)
im_color = cv2.cvtColor(im_color, cv2.COLOR_BGR2RGB)
imshow(im_color)
```

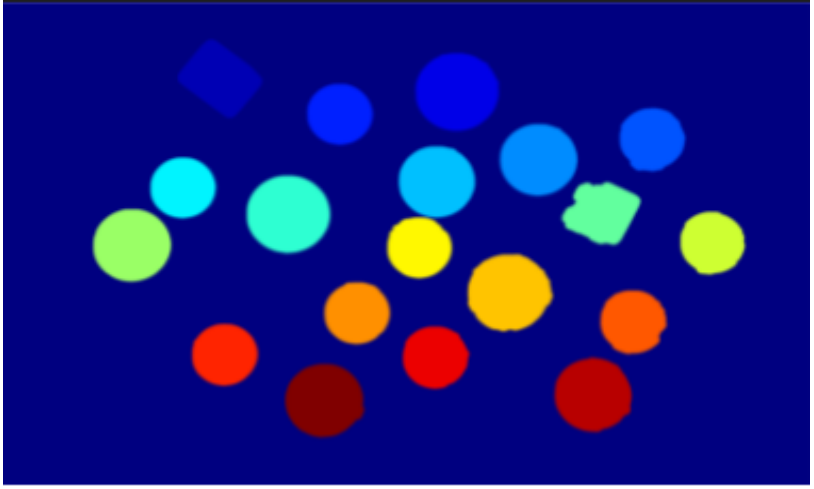


Imagen 9. Hallamos componentes conectadas y las visualizamos

Realizamos un bucle que itera sobre cada componente conectado en la imagen, y calculamos algunas características del mismo para luego poder clasificar. Luego, clasificamos cada componente como moneda o dado según una fórmula planteada , asignándole un color específico en una imagen etiquetada. La información de cada objeto se almacena en un diccionario y se agrega a una lista llamada **'lista_objetos'**, que contiene detalles sobre todos los objetos detectados en la imagen. La clasificación de las monedas se realiza según su área, determinando los valores de 0.5 (Moneda 50 centavos), 1(Moneda 1 peso), o 0.1 (Moneda 10 centavos).(Imagen 10)

```
for i in range(1, num_labels):

    #Creamos diccionario para almacenar el objeto y sus datos
    objeto_dicc = {}

    # --- 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)

    fp = area / perimeter**2

    area = stats[i, cv2.CC_STAT_AREA] // 1000

    # Guardamos el objeto y su centroide
    objeto_dicc['labels'] = obj
    objeto_dicc['centroid'] = centroids[i]
    if 1/fp > 10 and 1/fp < 15:
      objeto_dicc['clase'] = 'moneda'
      if (area > 110):
        labeled_image[obj == 1, 0] = 255    # moneda de 50 (Rojo)
        moneda50 += 1
        objeto_dicc['valor'] = '0.5'
      elif (area < 110 and area > 100 ):
        labeled_image[obj == 1, 1] = 255    # moneda de 1 (Verde)
        moneda1 += 1
        objeto_dicc['valor'] = '1'
      else:
        labeled_image[obj == 1, 2] = 255    # moneda de 10 (azul)
        moneda10 += 1
        objeto_dicc['valor'] = '0.1'
    else:
        labeled_image[obj == 1, 1] = 50    # dados (verde oscuro)
        objeto_dicc['clase'] = 'dado'
    lista_objetos.append(objeto_dicc)
```

Imagen 10. Clasificación de ambos objetos

Una vez detectados los distintos tipos de moneda, identificaremos y contaremos los puntos en cada dado. Primero, creamos una lista llamada **‘dados’** que contiene sólo los objetos clasificados como dados. Luego, para cada dado, se utiliza el operador **‘Canny’** para resaltar los puntos y aplicamos componentes conectados para iterar sobre cada objeto. La clasificación se basa en el factor de forma, filtrando los componentes que no sean círculos y tengan un área grande. La imagen etiquetada ‘**labeled_image**’ se actualiza con los puntos identificados, y el número de cada dado se asigna al diccionario dado como **dado[‘valor’]**.(Imagen 11)

```
# Hacemos una máscara con solo los dados
dados = [dicc for dicc in lista_objetos if dicc['clase'] == 'dado']

#placeholder
labeled_image = np.zeros_like(labels)

for dado in dados:
  numero_dado = 0
  # Decidimos usar Canny para identificar cada punto. Umbralizar también es una opción
  dado_canny = dado['labels'] * gcan
  # Hacemos componentes conectadas para separar cada punto del dado
  num_labels, labels, stats, centroids = cv2.connectedComponentsWithStats(dado_canny)
  # Clasifico en base al factor de forma
  for i in range(1, num_labels):
      # --- 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)

      fp = area / perimeter**2
      # Removemos el caso donde fp = 0, es decir, una componente conectada de un solo punto
      if fp == 0:
        continue

      # Identificamos círculos y nos quedamos con aquellos más grandes (filtramos según área)
      if 1/fp > 10 and 1/fp < 15 and stats[i][4] > 80:
        labeled_image[obj == 1] = 255
        # Contamos los puntos
        numero_dado += 1
  dado['valor'] = str(numero_dado)
```

Imagen 11. Clasificación de dados

Una vez realizado esto, en el diccionario **’lista_objetos’** tenemos toda la información de cada objeto.

Por último, graficamos una imagen en la que podremos visualizar los distintos tipos de monedas clasificadas por color y los dados. Arriba de cada objeto se podrá ver una etiqueta, que en caso de ser una moneda marcará de que valor es (0.5 (Moneda 50 centavos), 1(Moneda 1 peso), o 0.1 (Moneda 10 centavos)) y en caso de ser un dado marcará de que valor es el mismo. (Imagen 12)

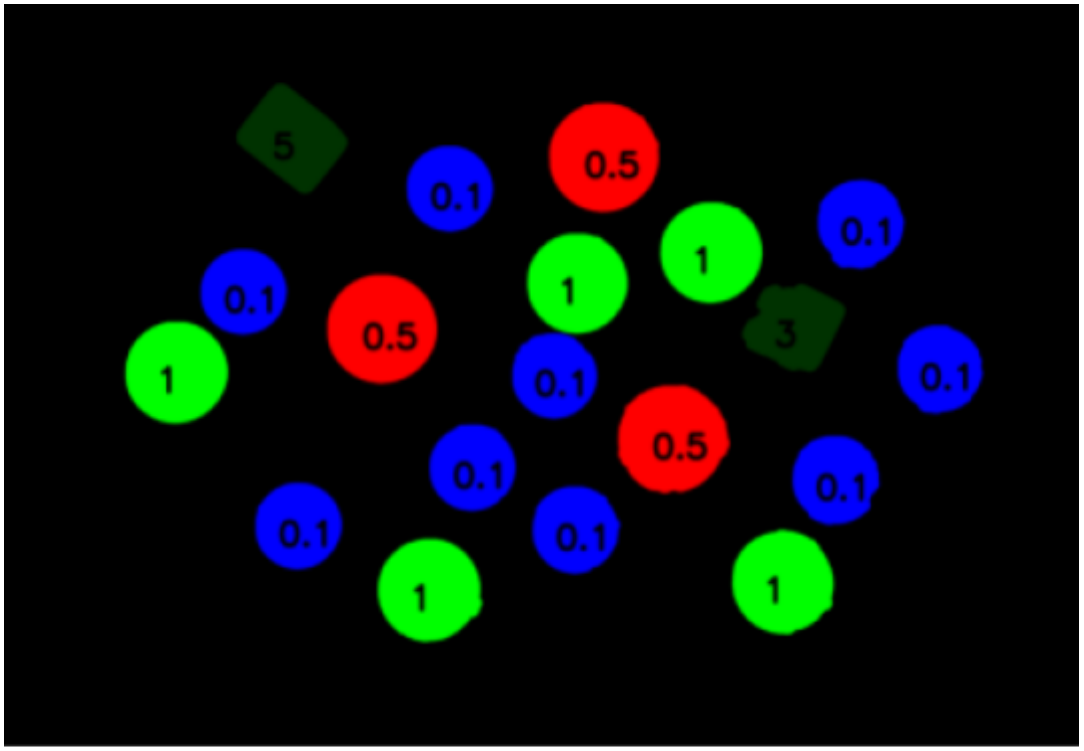


Imagen 12. Resultado final

RESOLUCIÓN EJERCICIO 2

En este segundo ejercicio, nuestro objetivo consistió en localizar patentes en 12 imágenes diferentes y, posteriormente, llevar a cabo la segmentación de los caracteres en dichas patentes. La estructura de nuestro código se dividió en dos secciones: la primera destinada a la localización y segmentación de las patentes, y la segunda focalizada en la segmentación de los caracteres dentro de dichas patentes.

Es importante señalar que hemos explorado diversos enfoques para abordar esta problemática, y la solución que presentamos aquí fue la más eficiente para las 12 imágenes. Aunque los resultados no fueron ideales y el código presenta algunas fallas en ciertos casos, consideramos que nos aproximamos significativamente a los resultados deseados.

La mayor complejidad que enfrentamos radicó en la búsqueda de un código único capaz de segmentar las patentes y los caracteres de manera óptima en el 100% de las imágenes. Hemos experimentado con diversas técnicas, utilizando distintos kernels y en diferentes secuencias, algunas de las cuales mostraron buen rendimiento en ciertas imágenes, mientras que otras no.

Para localizar las patentes en cada imagen, la resolución que planteamos es: iterar sobre cada una de ellas, aplicar técnicas de procesamiento de imágenes como borrosidad y análisis morfológico, hallar las componentes conectadas, filtrarlas para quedarnos con aquellas que cumplan con ciertos requisitos, y, finalmente, segmentar la patente según el bounding box.

Comenzamos creando una lista **‘image_files’** con los nombres de cada imagen para luego poder iterar sobre ellas. (Imagen 1)

```
#Lista con los nombres de nuestras imagenes
image_files = ['img01.png', 'img02.png', 'img03.png', 'img04.png',
               'img05.png', 'img06.png', 'img07.png', 'img08.png',
               'img09.png', 'img10.png', 'img11.png', 'img12.png']
```

Imagen 1. Lista de imágenes

Luego, para localizar las patentes estructuramos el código en un for, donde cada iteración corresponde con el procesamiento aplicado a cada una de las imágenes.

Dentro de este bucle, en primer lugar cargamos la imagen correspondiente en la variable **'img'.** (Imagen 2)

```
# Iteramos sobre cada imagen
for image_file in image_files:
    # Lee la imagen
    img = cv2.imread(image_file, cv2.IMREAD_GRAYSCALE)
```

Imagen 2. Guardamos la imagen en escala de grises

A continuación, aplicamos un filtro de borrosidad Gaussiana utilizando un kernel de 5x5. Seguidamente, implementamos el algoritmo de Canny con el objetivo de detectar los bordes de la imagen. (Imagen 3)

```
# CANNY
f_blur = cv2.GaussianBlur(img, ksize=(5, 5), sigmaX=1)
gcan = cv2.Canny(f_blur, threshold1=0.2*255, threshold2=1*255)
```

Imagen 3. Filtro y Canny

Con el fin de quitar líneas no deseadas que no aportan información relevante para segmentar las patentes, erosionamos con **‘cv2.erode’** la imagen con un kernel rectangular de 1x3. (Imagen 4)

```
# Erosionamos para quitar líneas no deseadas
kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (1, 3))
erosion = cv2.erode(gcan, kernel, iterations=1)
```

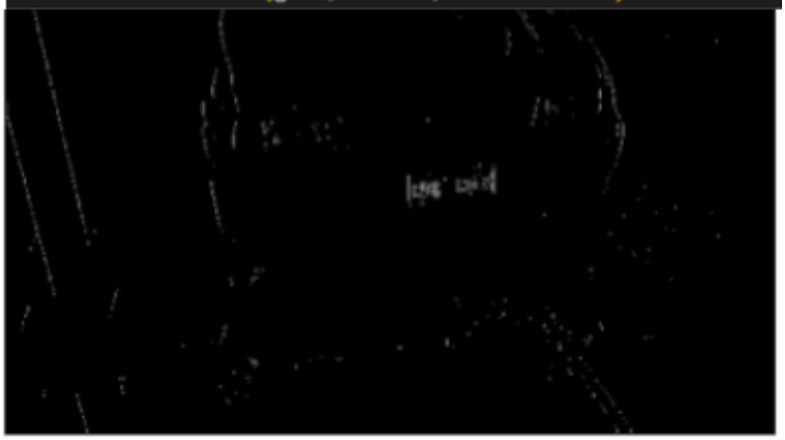


Imagen 4. Erosión

A continuación, dilatamos para resaltar los bordes que sí nos interesan. Definimos un kernel rectangular de 10x5 y aplicamos dilatación mediante **‘cv2.dilate’** (Imagen 5)

```
# Dilatamos
kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (10, 5))
Fd = cv2.dilate(erosion, kernel, iterations=1)
```

Imagen 5. Dilatación

Aplicamos clausura para mejorar la conectividad de los objetos en la imagen. Definimos un kernel rectangular de 7x2 y aplicamos clausura mediante **'cv2.morphology'** (Imagen 6)

Imagen 6. Clausura

Aplicamos **'cv2.connectedComponentsWithStats'** a nuestra imagen para poder identificar las componentes conectadas de la misma. Nuestro objetivo es identificar las patentes de los autos en base a criterios de área y relación de aspecto. Establecemos umbrales mínimos y máximos para el área y la relación de aspecto, y luego filtramos las componentes conectadas que cumplen con estos criterios.

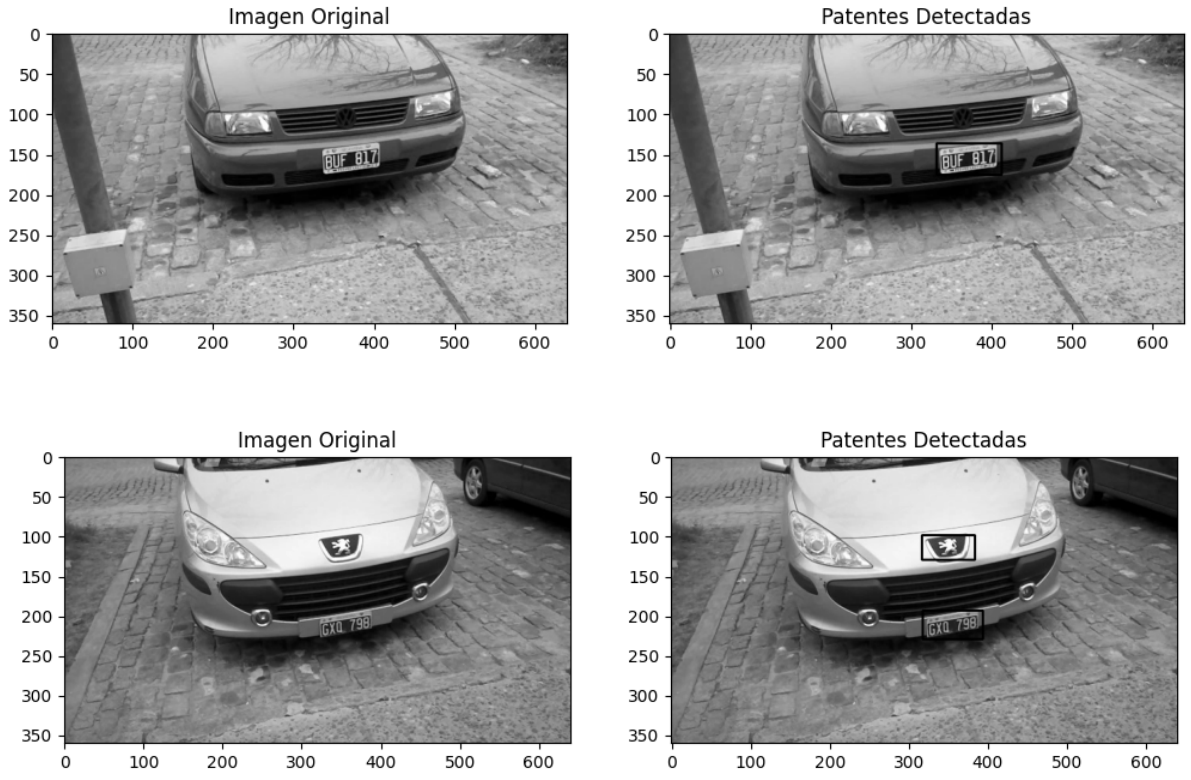
Dichos umbrales los seleccionamos a partir del análisis particular de las componentes conectadas de cada imagen, de manera de optimizar estas constantes y filtrar la mayor cantidad de componentes.

Las regiones seleccionadas se resaltan en la imagen original, y se almacenan en una lista llamada patentes. Finalmente, agregamos esta lista de regiones de interés a una lista general llamada **'todas_las_patentes'**. (Imagen 7)

```python
#Componentes conectadas
num_labels, labels, stats, centroids = cv2.connectedComponentsWithStats(closing, connectivity=8)

canvas = img.copy()

area_umbral_min = 800
area_umbral_max = 2630

aspect_ratio_min = 1.80
aspect_ratio_max = 3.10

patentes = []

# Filtramos
for label in range(1, num_labels):
    if (
        area_umbral_min < stats[label, cv2.CC_STAT_AREA] < area_umbral_max
        and
        aspect_ratio_min < stats[label, cv2.CC_STAT_WIDTH] / stats[label, cv2.CC_STAT_HEIGHT] < aspect_ratio_max
    ):
        x, y, w, h = stats[label, cv2.CC_STAT_LEFT], stats[label, cv2.CC_STAT_TOP],
        stats[label, cv2.CC_STAT_WIDTH], stats[label, cv2.CC_STAT_HEIGHT]

        cv2.rectangle(canvas, (x, y), (x + w, y + h), (0, 255, 0), 2)
        patente = img[y:y+h, x:x+w]
        patentes.append(patente)

todas_las_patentes.append(patentes)
```

Imagen 7. Hallazgo y filtrado de componentes conectadas

A continuación se muestran las 12 imágenes originales, y a la derecha de cada una, el bounding box de cada componente encontrado.



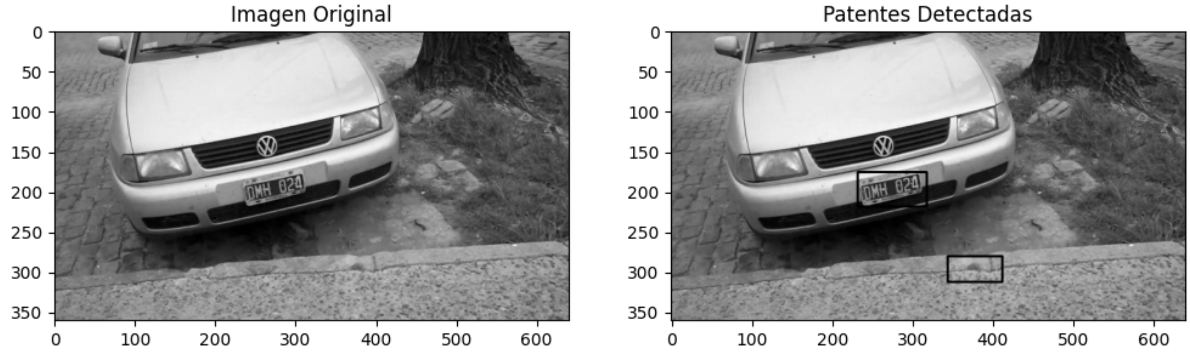


Imagen Original
Patentes Detectadas
DMH 024
DMH 024

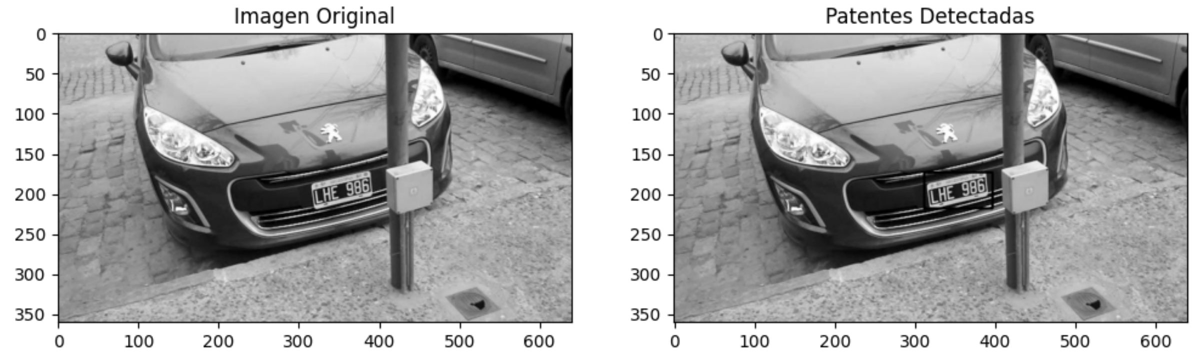


Imagen Original
Patentes Detectadas
LHE 986
LHE 986

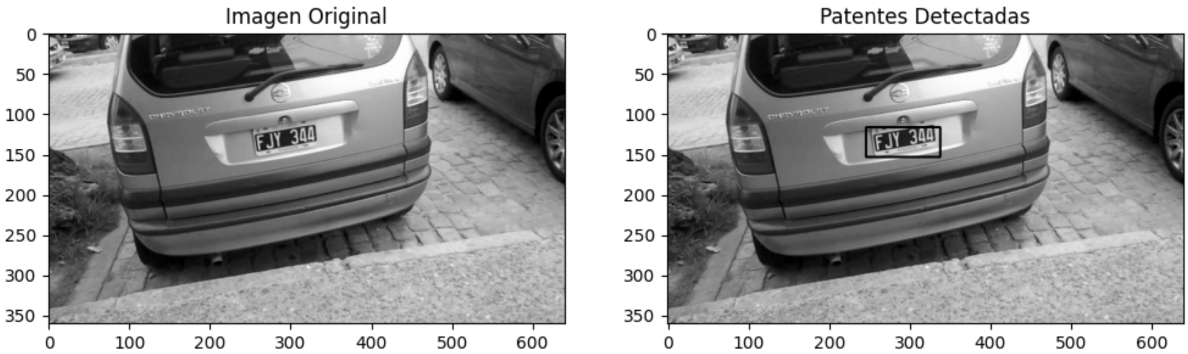


Imagen Original
Patentes Detectadas
FJY 344
FJY 344

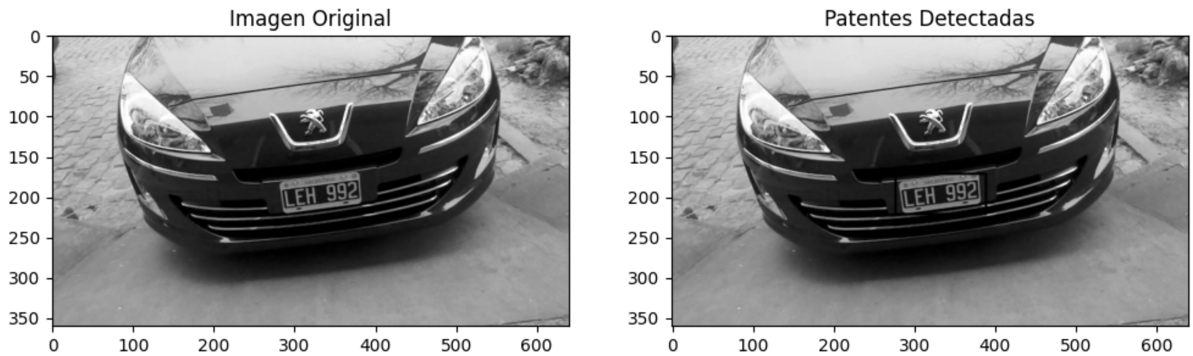


Imagen Original
Patentes Detectadas
LEH 992
LEH 992

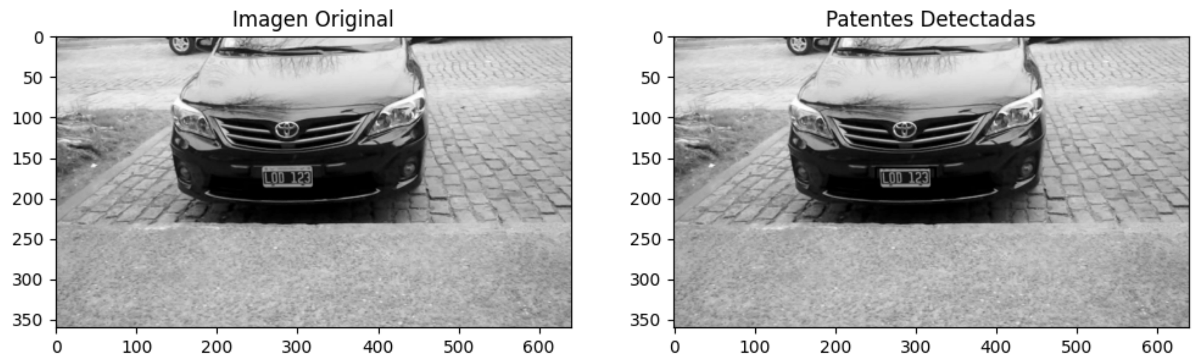


Imagen Original
Patentes Detectadas
LOD 123

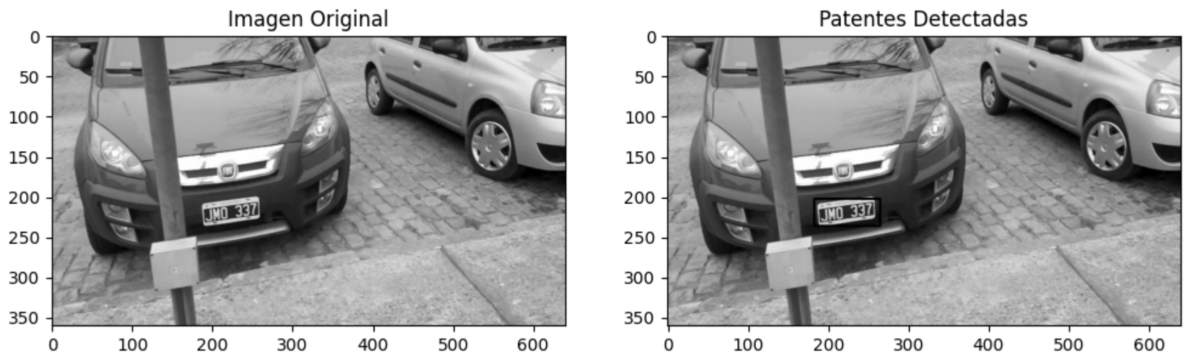


Imagen Original
Patentes Detectadas
JMO 337

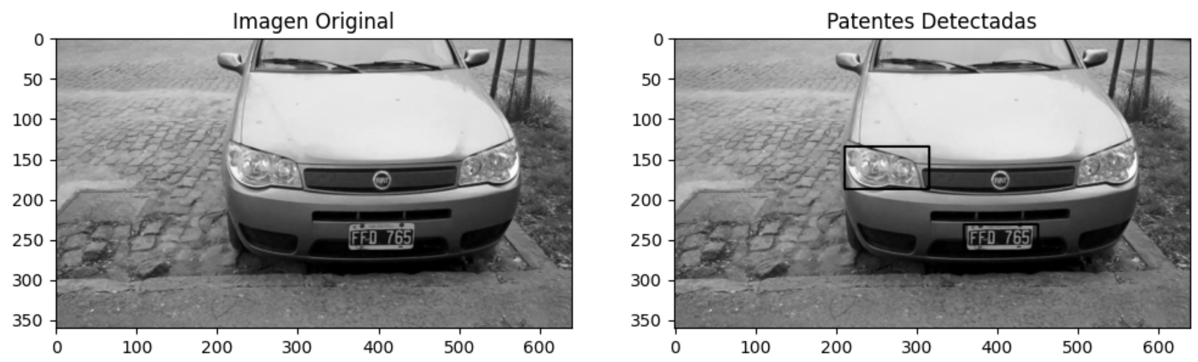


Imagen Original
Patentes Detectadas
FFD 765

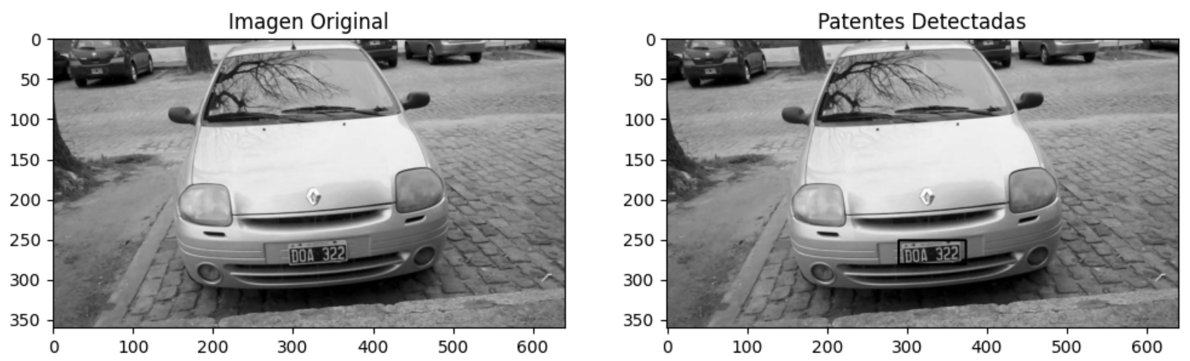
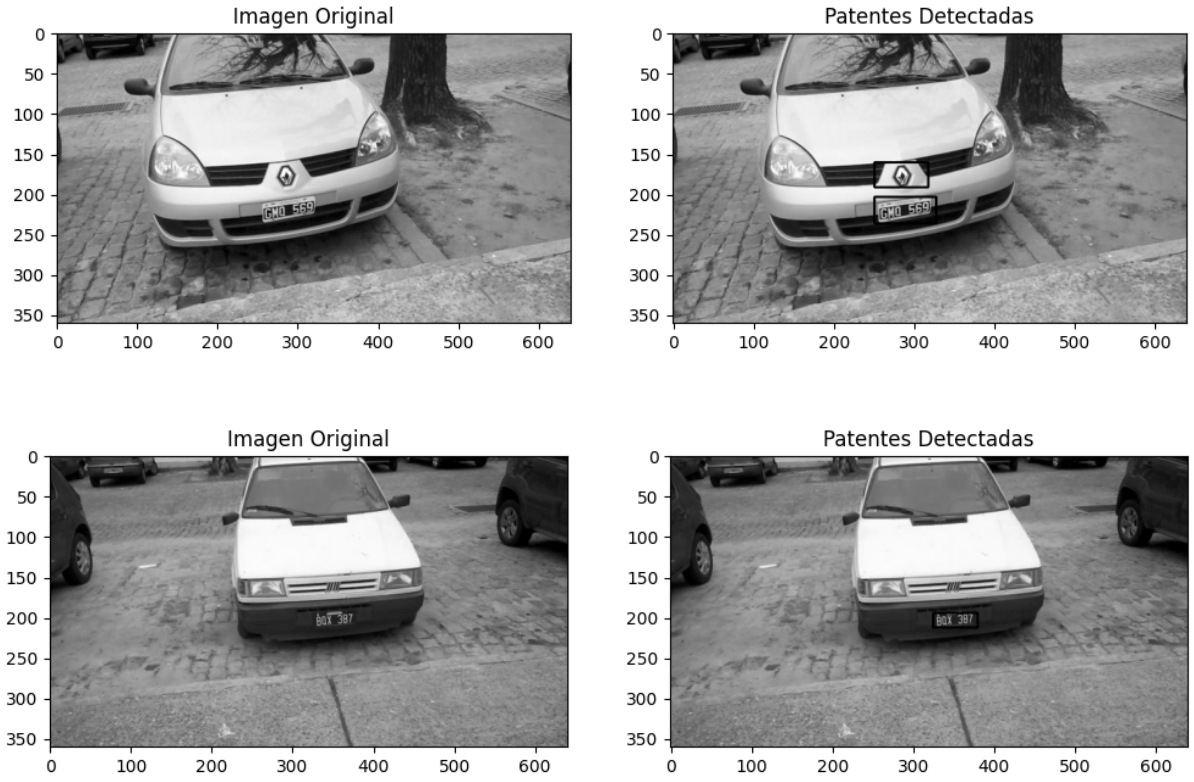


Imagen Original
Patentes Detectadas
DOA 322



Como las componentes conectadas han detectado algunos objetos que cumplen con los requisitos de filtrado además de las patentes, los eliminamos. Así, en la lista **'todas_las_patentes'** sólo nos quedan las imágenes de las patentes. (Imagen 8)

```
# Índices a eliminar: patentes[1][0], patentes[2][1], patentes[8][0], patentes[10][0]
indices_a_eliminar = [(1, 0), (2, 1), (8, 0), (10, 0)]

todas_las_patentes = [
    [patente for j, patente in enumerate(patentes) if (i, j) not in indices_a_eliminar]
    for i, patentes in enumerate(todas_las_patentes)]
```

Imagen 8. Eliminamos las componentes conectadas que no son patentes

A continuación, detectaremos los caracteres de cada patente. Para ello, iteramos sobre las imágenes de las patentes, y a cada una de ellas le aplicaremos lo siguiente:

En primer lugar, aplicamos **'Black Hat'** con un kernel 40x40 para resaltar regiones oscuras en la imagen. (Imagen 9)

Cabe aclarar que, si bien no hemos dado '**Black Hat**' en la materia (sino un método similar: '**Top Hat**'), nos hemos informado sobre su aplicación en la documentación de opencv y nos hemos asegurado la aprobación del docente de teoría antes de hacer uso del método para este ejercicio.

```
# Iteramos sobre todas las patentes
for i, patente in enumerate(todas_las_patentes):
    for j, patente in enumerate(patente):

        # Aplicamos 'Black Hat' para resaltar regiones oscuras en la imagen
        se = cv2.getStructuringElement(cv2.MORPH_RECT, (40, 40))
        g3 = cv2.morphologyEx(patente, kernel=se, op=cv2.MORPH_BLACKHAT)
```

Imagen 9. Black Hat

Se observa que Black Hat nos ayuda en gran medida resaltar los caracteres de las patentes, lo cual será más sencillo a la hora de detectar las componentes conectadas.

A continuación, binarizamos la imagen a través de **‘cv2.threshold’**, con un umbral de 70. (Imagen 10)

Dicho umbral fue elegido en base a pruebas realizadas. Se optó por tal número ya que maximiza el número de caracteres segmentados.

```
# Binarizamos la imagen
_, pat_binarizada = cv2.threshold(g3, 70, 255, cv2.THRESH_BINARY_INV)
```

Imagen 10. Binarizamos la imagen

A continuación, hacemos una copia de nuestra imagen y luego aplicamos **‘cv2.connectedComponentsWithStats’** a nuestra imagen para poder identificar las componentes conectadas de la misma. Nuestro objetivo es identificar los caracteres de los autos en base a criterios de área y relación de aspecto. Establecemos umbrales mínimos y máximos para el área y la relación de aspecto, y luego filtramos las componentes conectadas que cumplen con estos criterios.

Nuevamente, dichos umbrales fueron elegidos luego de analizar los componentes conectados de cada imagen, asegurándonos de elegir los más óptimos y filtrar la mayor cantidad de componentes elegidas. (Imagen 11)

```python
#Hacemos una copia de la imagen original
canvas = patente.copy()

#Hacemos componentes conectadas
num_labels, labels, stats, centroids = cv2.connectedComponentsWithStats(pat_binarizada, connectivity=8)

 # Definimos umbrales
area_umbral_min = 15
area_umbral_max = 300

aspect_ratio_min = 0.4
aspect_ratio_max = 1.2

# Iteramos sobre cada componente conectada y filtramos
for label in range(1, num_labels):
    if (
        area_umbral_min < stats[label, cv2.CC_STAT_AREA] < area_umbral_max
        and aspect_ratio_min < stats[label, cv2.CC_STAT_WIDTH] / stats[label, cv2.CC_STAT_HEIGHT] < aspect_ratio_max
    ):
        # Obtenemos las coordenadas del cuadro delimitador de la componente conectada
        x, y, w, h = stats[label, cv2.CC_STAT_LEFT], stats[label, cv2.CC_STAT_TOP], stats[label, cv2.CC_STAT_WIDTH], stats[label, cv2.CC_STAT_HEIGHT]

        # Dibujamos un cuadro alrededor de la componente conectada
        cv2.rectangle(canvas, (x, y), (x + w, y + h), (255, 0, 255), 2)

        #Imprimos informacion
        #print(stats[label, cv2.CC_STAT_WIDTH], stats[label, cv2.CC_STAT_HEIGHT], "area: ", stats[label, cv2.CC_STAT_AREA])
```

Imagen 11. Hallazgo y filtrado de componentes conectadas

A continuación, se presentan los resultados de la segmentación de los caracteres en cada patente. Es evidente que los resultados obtenidos no alcanzaron la perfección, a pesar de nuestros múltiples intentos por encontrar un código único que funcione de manera consistente para las 12 patentes.

En el caso de la tercera patente, que presenta mayor borrosidad que las demás, el primer carácter ('D') se fusiona con el borde. Aunque logramos detectar todos sus caracteres al aumentar el tamaño del kernel rectangular en la operación de 'Black Hat', este ajuste no resultaba efectivo para otras patentes.

Por otro lado, en la patente 5, a pesar de haber segmentado correctamente los caracteres, identificamos una componente conectada adicional que cumple con los criterios de filtrado, aunque no corresponde a un carácter.

Finalmente, en el caso de la patente 10, los dos últimos caracteres fueron reconocidos como uno solo. Este problema surgió debido a que, con las técnicas aplicadas, los píxeles se mezclaron, generando una única componente conectada. Si bien este problema podría resolverse aplicando una clausura mayor, como mencionamos anteriormente, dicha solución afectaría la detección de caracteres en otras patentes.

Patente 1

Detección OK

Patente 2

Detección OK

Patente 3

Detección Inc.

Patente 4

Detección OK

Patente 5

Detección -

Patente 6

Detección OK

Patente 7

Detección OK

Patente 8

Detección OK

Patente 9

Detección OK

Patente 10

Detección -

Patente 11

Detección OK

Patente 12

Detección OK

Aquí concluye el ejercicio 2. Si bien reconocemos que no hemos alcanzado el resultado esperado, consideramos que la solución presentada representa el enfoque más óptimo entre todos los intentos que realizamos.

CONCLUSIONES FINALES

A partir de los ejercicios realizados, hemos adquirido conocimientos y aplicado diversas técnicas de preprocesamiento de imágenes con el objetivo de segmentar áreas de interés específicas. Estas técnicas fueron empleadas tanto en la segmentación de monedas y dados, como se observó en el primer ejercicio, así como en la identificación y delimitación de regiones relevantes en el caso del ejercicio 2, centrado en el análisis de patentes y caracteres.

Si bien logramos obtener resultados exitosos en la mayoría de los casos, nos encontramos con ciertas dificultades y dudas durante el desarrollo del trabajo práctico. Específicamente, en el ejercicio 2 exploramos diversas aproximaciones antes de llegar a la solución final que detallamos en este informe. Inicialmente, probamos un enfoque diferente que no incluía el uso de bordes Canny, pero observamos que los resultados eran menos óptimos en comparación con la estrategia que finalmente adoptamos.

La principal complicación que enfrentamos allí fue la creación de un código único capaz de funcionar con las 12 imágenes, las cuales presentan variadas características, como el color del automóvil, reflejos de la luz solar en la patente y detalles adicionales en el entorno, como el pavimento empedrado. Estos elementos no relevantes complicaron la tarea de encontrar una configuración de hiperparámetros adecuada, especialmente para los kernels. Determinar el formato correcto del kernel (ej. elipse o rectángulo) y su tamaño se convirtió en nuestro mayor desafío, ya que buscábamos una configuración que fuera efectiva para todas las imágenes sin comprometer la calidad de los resultados.

En resumen, aunque experimentamos con diversas estrategias de procesamiento, los resultados del ejercicio 2 no alcanzaron la excelencia. Sin embargo, consideramos que sí mostraron una notable aproximación a nuestros objetivos.