

6.1 Shape descriptors

The main objective of region description is to obtain a mathematical representation of a segmented region from an image consisting of a vector of features $\mathbf{x} = [x_1, \dots, x_n]$.



In this notebook we will see a branch of region description called **shape analysis**. Shape analysis aims to construct this feature vector using only shape features (e.g., size, perimeter, circularity or compactness).

Depending on the application, it could be needed that the used descriptor be **invariant** to the position in the image in which the regions appears, its orientation, and/or its size (scale). Some examples:



This notebook **covers simple shape descriptors of regions** based on their area, perimeter, minimal bounding-box, etc (sections 6.1.1 and 6.1.2). We will also study **if these descriptors are invariant to position, orientation and size** (section 6.1.3). Let's go!

Problem context - Number-plate recognition

So here we are again! UMA called for us to join a team working on their parking access system. This time, they want to upgrade their obsolete number-plate detection algorithm by including better and more efficient methods.



Here is where our work starts, we are going to **apply shape analysis to each of the characters** that can appear on a license plate, that is, numbers from 0 to 9, and letters in the alphabet. The idea is to **produce a unique feature vector** for each character that could appear on a plate (e.g. $\mathbf{x}^0, \mathbf{x}^1, \dots, \mathbf{x}^A, \mathbf{x}^B$, etc.) so it could be later used to **train an automatic classification system** (we will see this in the next chapter!).

```
In [1]: import numpy as np
import cv2
import matplotlib.pyplot as plt
import matplotlib
matplotlib.rcParams['figure.figsize'] = (15.0, 8.0)

images_path = './images/'
```

Initial data

UMA's parking security team have sent us some segmented plate characters captured by their camera in the parking. They have binarized and cropped these images, providing us with regions representing such characters as white pixels. These cropped images are `region_0.png` (region with a zero), `region_6.png` (region with a six), `region_B.png` (region with a B), and `region_J.png` (region with a J).

Let's visualize them!

In [2]:

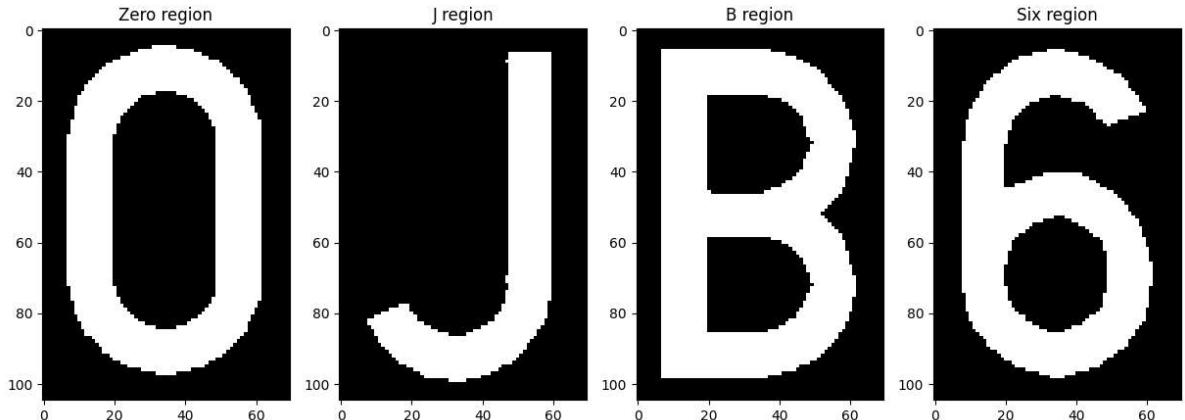
```
# Read the images
zero = cv2.imread(images_path + 'region_0.png',0)
J = cv2.imread(images_path + 'region_J.png',0)
B = cv2.imread(images_path + 'region_B.png',0)
six = cv2.imread(images_path + 'region_6.png',0)

# And show them!
plt.subplot(141)
plt.imshow(zero, cmap='gray')
plt.title('Zero region')

plt.subplot(142)
plt.imshow(J, cmap='gray')
plt.title('J region')

plt.subplot(143)
plt.imshow(B, cmap='gray')
plt.title('B region')

plt.subplot(144)
plt.imshow(six, cmap='gray')
plt.title('Six region');
```



6.1.1 Compactness

The first feature we are going to work with is **compactness**:

$$\text{compactness} = \frac{\text{area}}{\text{perimeter}^2}$$

\[5pt]

As you can see, this feature associates the area with the perimeter of a region. Informally, it tells how *rounded* and *closed* is a region. The most compact shape is the circle, with $\text{compactness} = 1/(4\pi)$.

No description has been provided for this image

OpenCV pill

OpenCV uses contours for analysing shapes. A contour is a list of points that defines a region. We can obtain the contours of a region using `cv2.findContours()`.

ASSIGNMENT 1: Computing compactness

What to do? Complete the function bellow, named `compactness()`, which computes the compactness of an input region.

For that, we are going to use the `cv2.findContours()` function, which takes as input:

- A binary image (containing the region as white pixels).
- Contour retrieval mode, it can be:
 - `RETR_EXTERNAL` : only returns the external contour
 - `RETR_LIST` : returns all contours (e.g. the character 0 would contain two contours: external and internal)
 - `RETR_CCOMP` : returns all contours and organize them in a two-level hierarchy. At the top level, there are external boundaries of the components. At the second level, there are boundaries of the holes.
- Method: controls how many points of the contours are being stored, this is for optimization purposes.
 - `CHAIN_APPROX_NONE` : stores absolutely all the contour points.
 - `CHAIN_APPROX_SIMPLE` : compresses horizontal, vertical, and diagonal segments and leaves only their end points.
 - `CHAIN_APPROX_TC89_L1` : applies an optimization algorithm.

And returns:

- a list containing the contours,
- and a list containing information about the image topology. It has as many elements as the number of contours.

For simplicity, we are going to take into account **only the external boundary** (as if the regions have not holes), so the second output is not relevant.

Having the contours, you can obtain the **area** and the **perimeter** of the region through `cv2.contourArea()` and `cv2.arcLength()`. Both functions take the contours of the region as input.

Note: Use `cv2.RETR_EXTERNAL` and `cv2.CHAIN_APPROX_NONE`.

```
In [3]: # Assignment 1
def compactness(region):
    """ Compute the compactness of a region.

        Args:
            region: Binary image

        Returns:
            compactness: Compactness of region (between 0 and 1/4pi)
    """
    plt.imshow(region,cmap='gray')
    plt.show()
    # Get external contour
    contours,_ = cv2.findContours(region, cv2.RETR_EXTERNAL ,cv2.CHAIN_APPROX_NONE)
    cnt = contours[0]
```

```

img_contours = np.zeros(region.shape)
# draw the contours on the empty image
cv2.drawContours(img_contours, contours, -1, (255,255,255), 1)
plt.imshow(img_contours,cmap='gray')
plt.show()

# Calculate area
area = cv2.contourArea(cnt)

# Calculate perimeter
perimeter = cv2.arcLength(cnt,True)

print("Area:",area)
print("Perimeter:", perimeter)

# Calculate compactness
compactness = area/(perimeter*perimeter)

return compactness

```

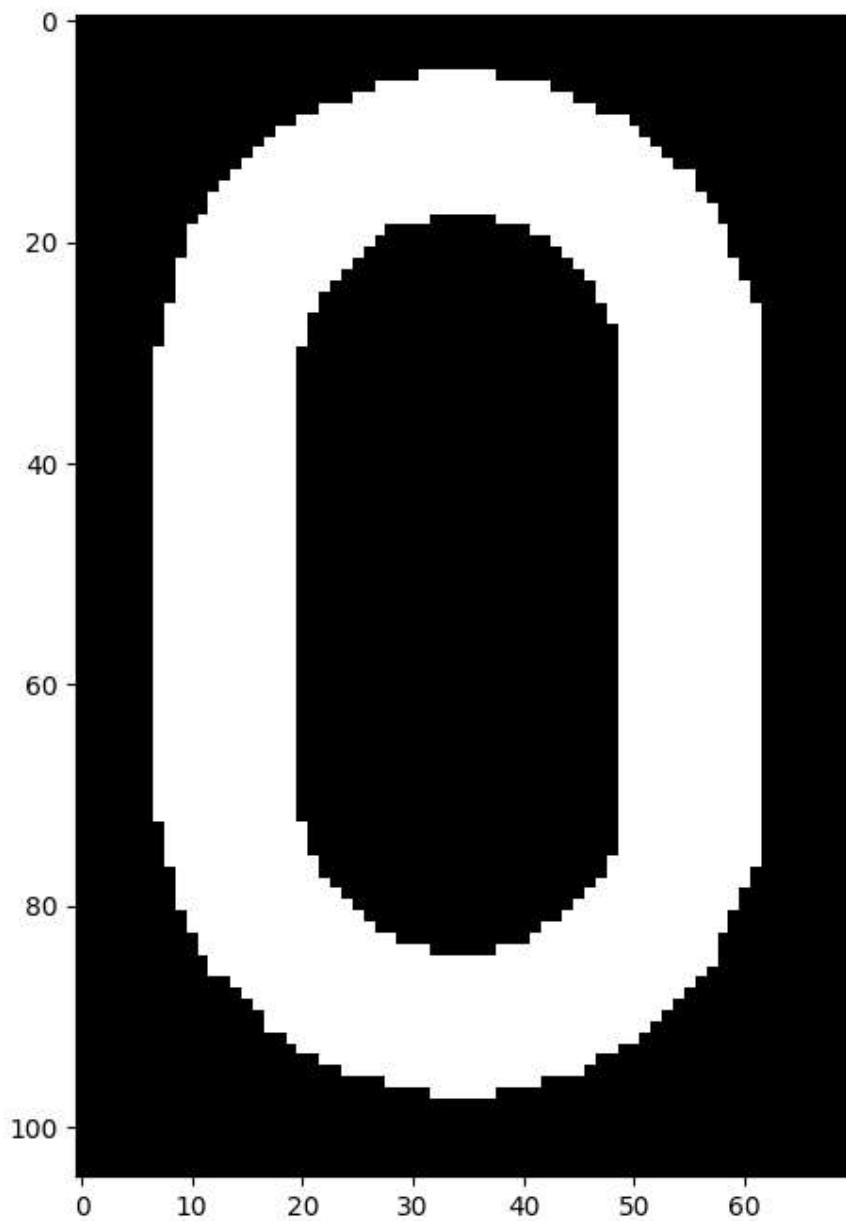
You can use next code to **test if the results are right**:

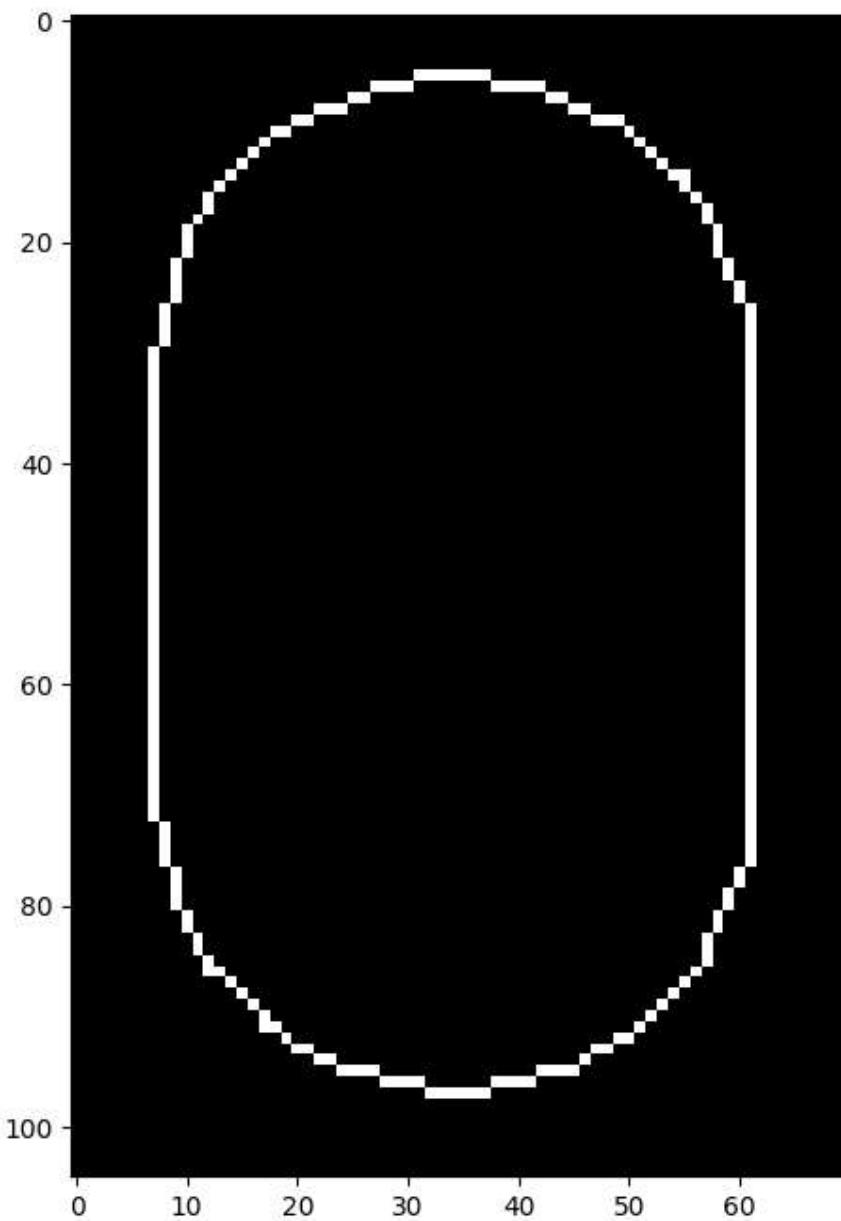
```

In [4]: # Read the images
zero = cv2.imread(images_path + 'region_0.png',0)
J = cv2.imread(images_path + 'region_J.png',0)
B = cv2.imread(images_path + 'region_B.png',0)
six = cv2.imread(images_path + 'region_6.png',0)

# And show their compactness!
print(" Compactness of 0: ", round(compactness(zero),5), "\n",
      "Compactness of J: ", round(compactness(J),5), "\n",
      "Compactness of B: ", round(compactness(B),5), "\n",
      "Compactness of 6: ", round(compactness(six),5))

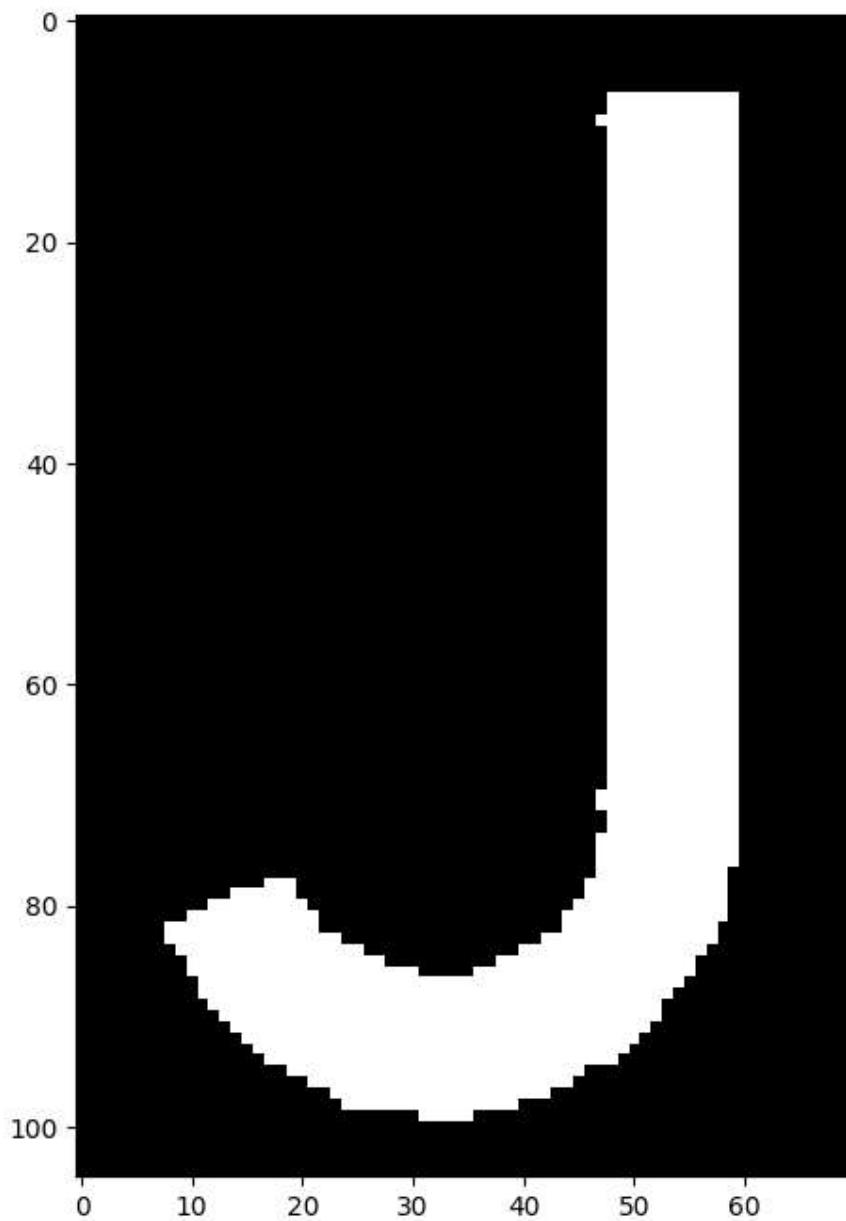
```

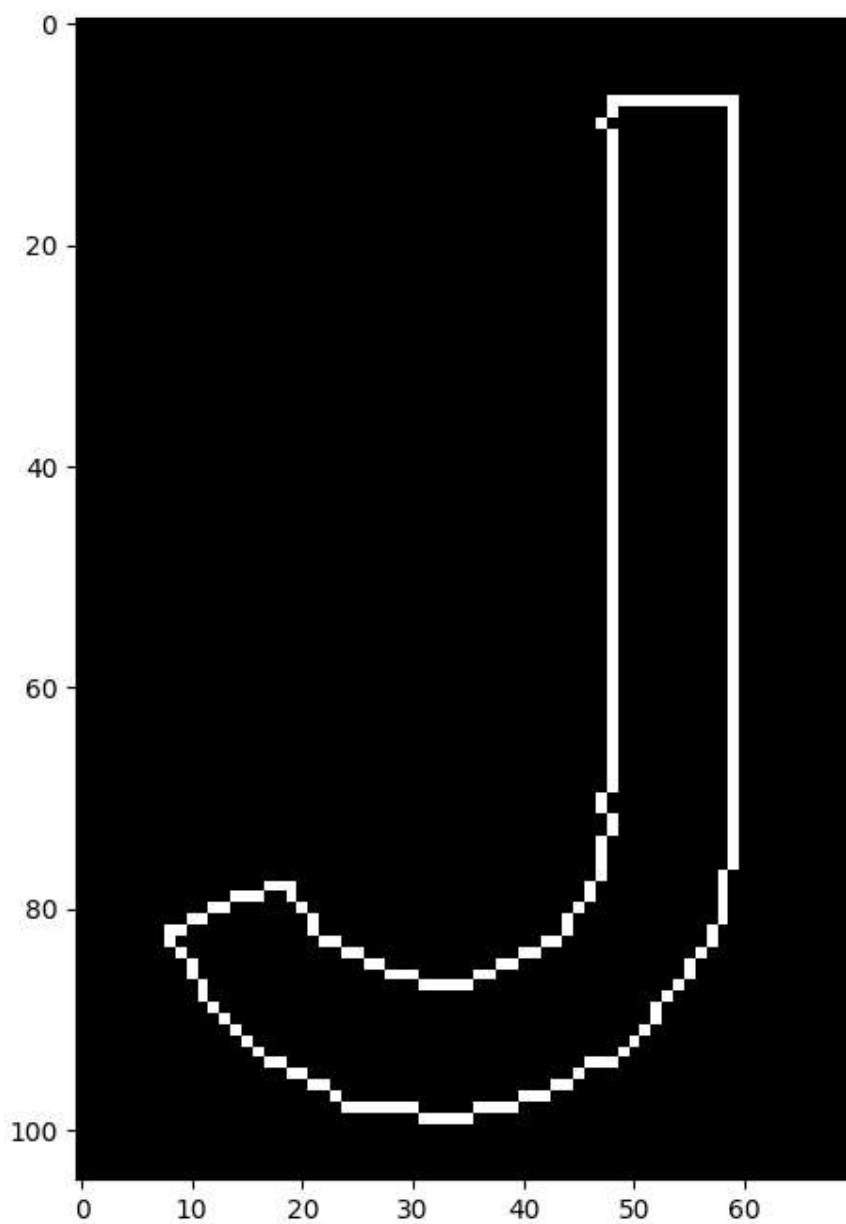




Area: 4307.0

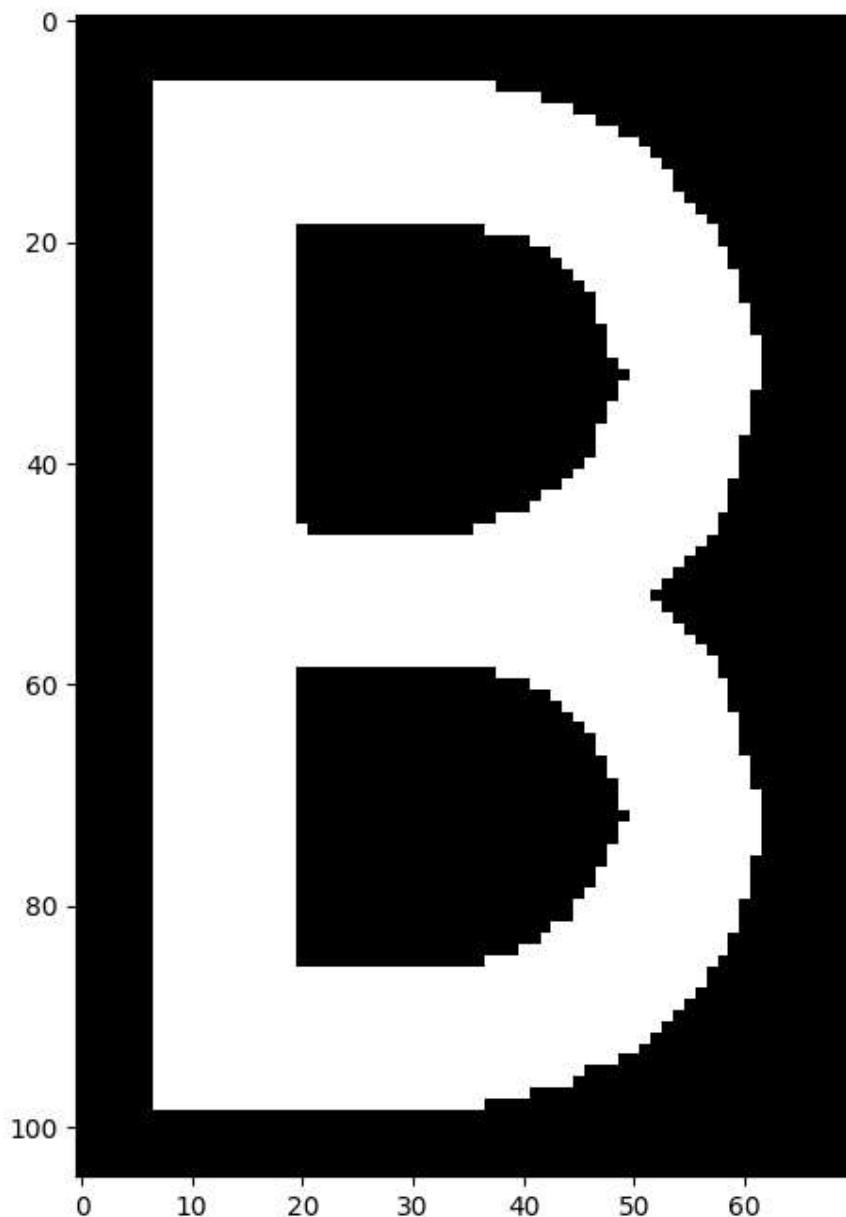
Perimeter: 255.68123936653137

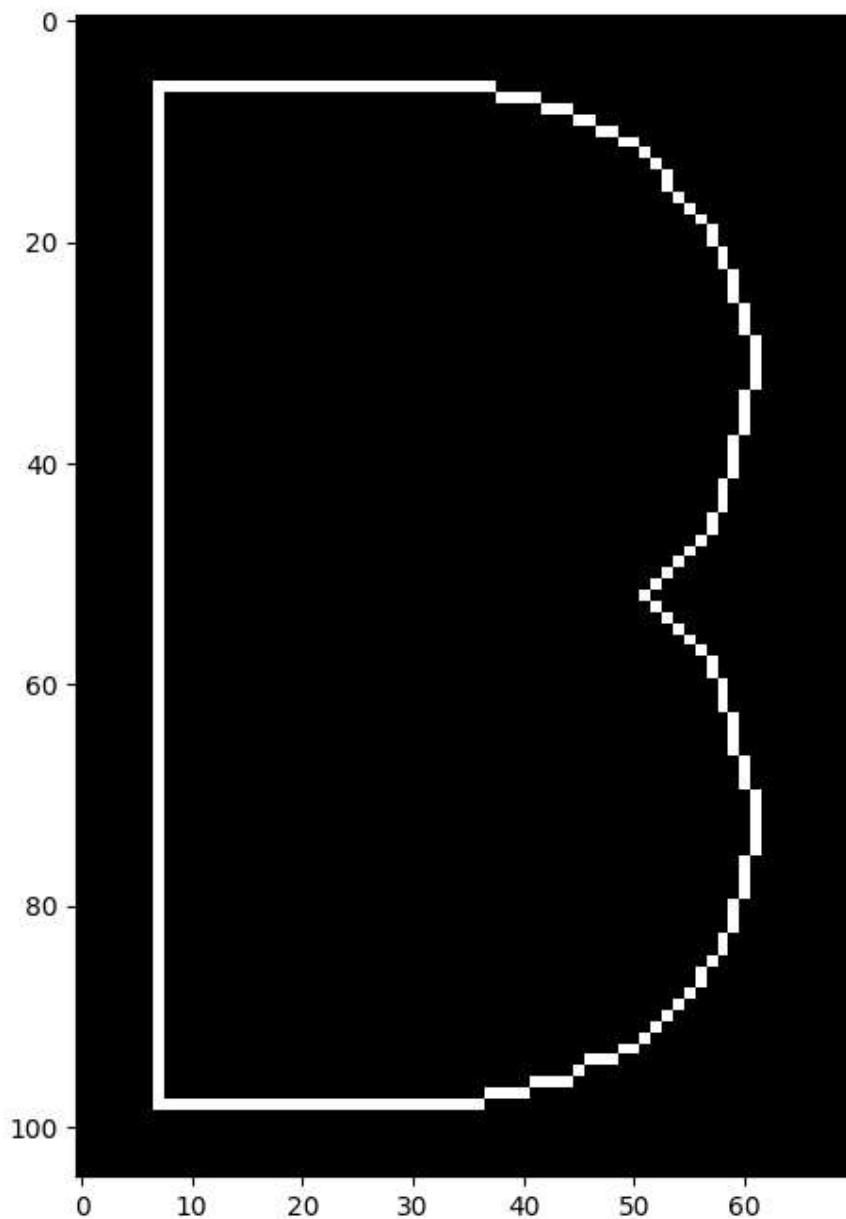




Area: 1386.0

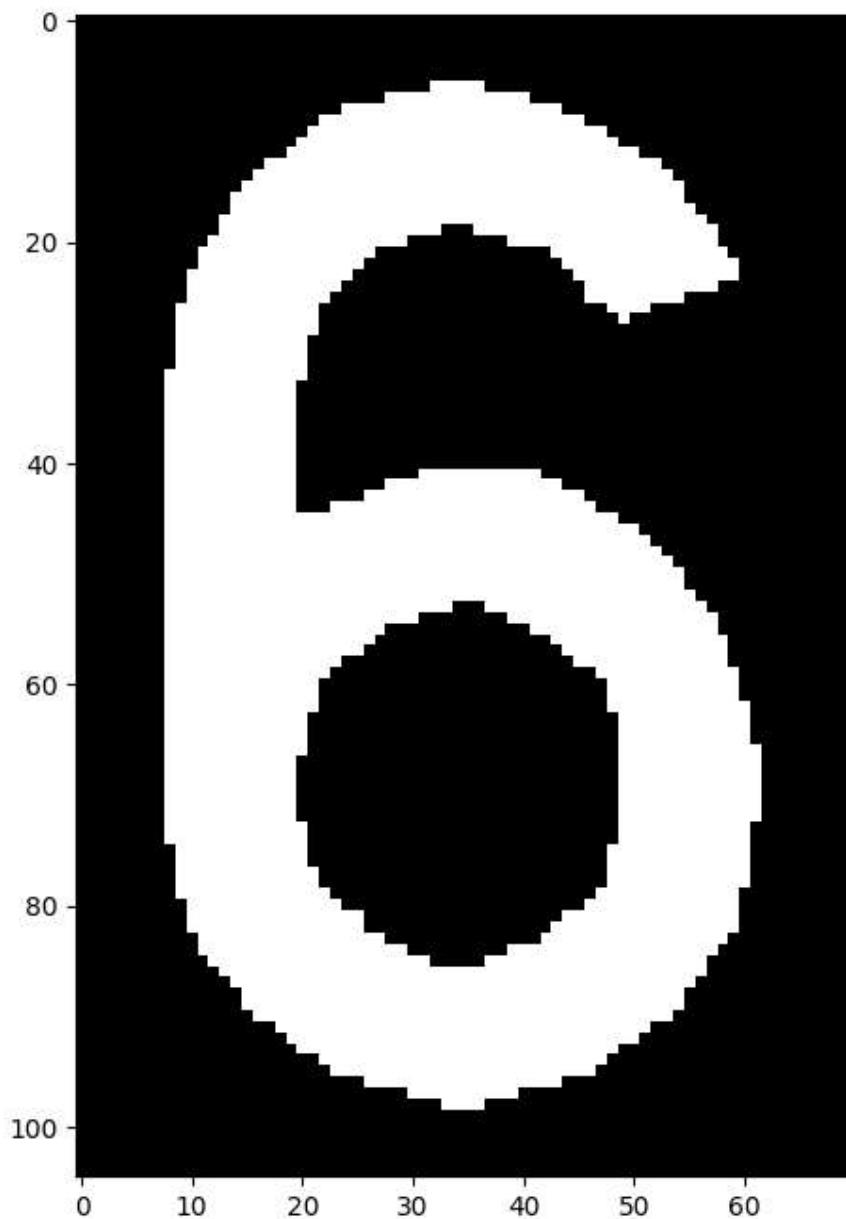
Perimeter: 276.3675310611725

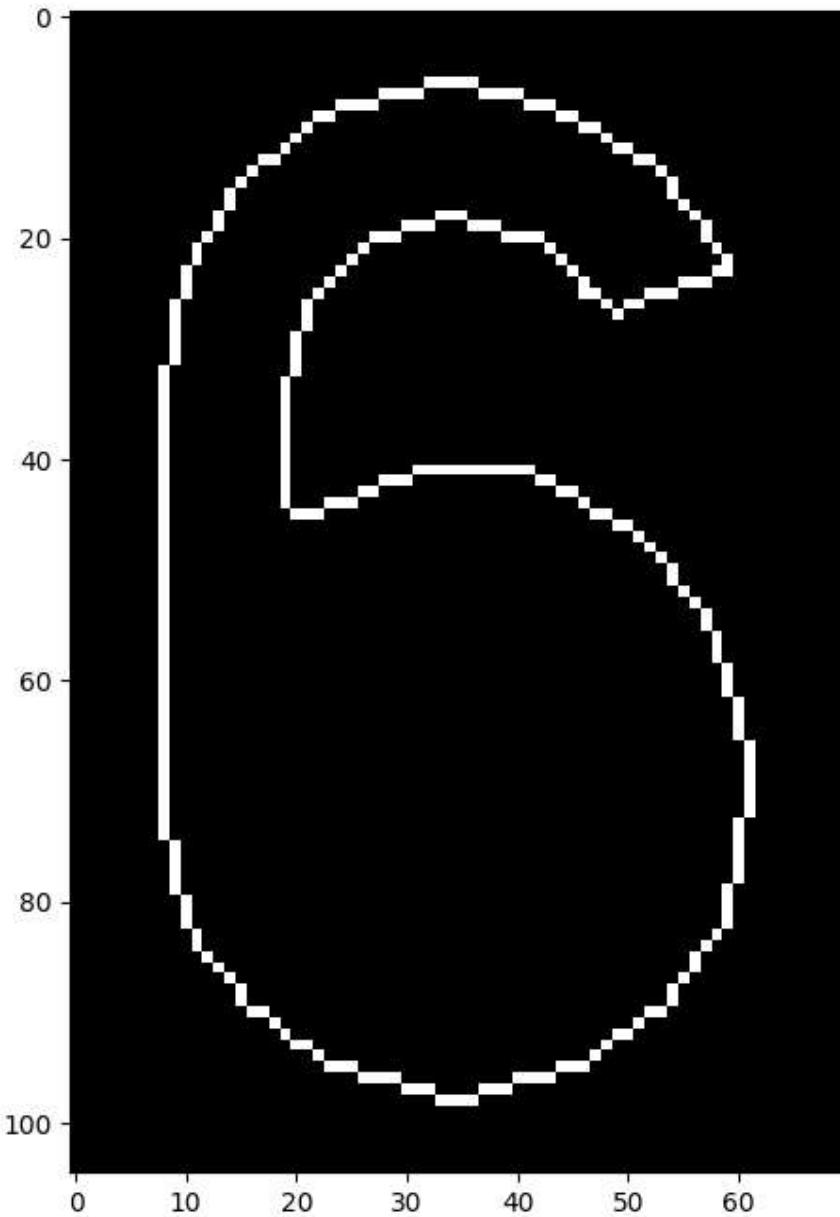




Area: 4498.0

Perimeter: 281.53910398483276





Area: 3217.5

Perimeter: 334.4924215078354

Compactness of 0: 0.06588

Compactness of J: 0.01815

Compactness of B: 0.05675

Compactness of 6: 0.02876

Expected output (using `CHAIN_APPROX_NONE`):

Compactness of 0: 0.06588

Compactness of J: 0.01815

Compactness of B: 0.05675

Compactness of 6: 0.02876

Thinking about it (1)

Excellent! Now, **answer the following questions:**

- Why `region_0.png` have the greatest compactness?

Porque el contorno que extraemos (el de un 0) es el que más se parece a un círculo de todos las imágenes propuestas, y hemos visto que los círculos son la forma más compacta, ya que

su área es bastante grande respecto a su perímetro.

- Could we differentiate all characters using only this feature as feature vector?

No, puesto que puede coincidir que diferentes caractéres tengan una compacidad igual o muy similar, sin ser el mismo carácter, ya que puede darse la casualidad de que la relación entre el área y el perímetro sea la misma. Por ejemplo, en el caso de M y W podríamos tener la misma compacidad pero los caracteres serían distintos.

- Is compactness invariant to position, orientation or scale?

Sí, ya que no tiene en cuenta la posición ni la orientación ni la escala, es decir, si variasemos cualquiera de estas tres cualidades la compacidad se mantendría igual, ya que la relación entre el área y el perímetro se mantiene constante.

6.1.2 Extent

Another shape descriptor is **extent** of a shape:

$$\text{extent} = \frac{\text{area}}{\text{bounding rectangle area}}$$

\[5pt]

This feature associates the area of the region with the area its bounding rectangle. A **bounding rectangle** can be defined as the minimum rectangle that contains all the pixels of a region whose bottom edge is horizontal and its left edge is vertical.



No description has been provided for this image

The shape with the highest extent value is the rectangle, with $\text{extent} = 1$, while the lowest one is an empty region so $\text{extent} = 0$.

ASSIGNMENT 2: Time to compute the extent

Complete the function `extent()`, which receives the `region` to be described as input and returns its `extent`.

Tip: compute the bounding rectangle using `cv2.boundingRect()`, which also takes the contours as input.

```
In [5]: def extent(region):
    """ Compute the extent of a region.

    Args:
        region: Binary image

    Returns:
        extent: Extent of region (between 0 and 1)
    """

    # Get external contour
    contours, _ = cv2.findContours(region, cv2.RETR_CCOMP, cv2.CHAIN_APPROX_SIMPLE)
    cnt = contours[0]

    # Calculate area
    area = cv2.contourArea(cnt)
```

```

# Get bounding rectangle
_, _, w, h = cv2.boundingRect(cnt)

# Calculate bounding rectangle area
rect_area = w*h

# Calculate extent
extent = float(area)/rect_area

return extent

```

You can use next code to **test if the obtained results are correct**:

```

In [6]: # Read the images
zero = cv2.imread(images_path + 'region_0.png', 0)
J = cv2.imread(images_path + 'region_J.png', 0)
B = cv2.imread(images_path + 'region_B.png', 0)
six = cv2.imread(images_path + 'region_6.png', 0)

# And show their extent!
print("Extent of 0: ", round(extent(zero), 5), "\n",
      "Extent of J: ", round(extent(J), 5), "\n",
      "Extent of B: ", round(extent(B), 5), "\n",
      "Extent of 6: ", round(extent(six), 5))

```

Extent of 0: 0.84203
Extent of J: 0.2866
Extent of B: 0.87937
Extent of 6: 0.64068

Expected output (using CHAIN_APPROX_NONE):

Extent of 0: 0.84203
Extent of J: 0.2866
Extent of B: 0.87937
Extent of 6: 0.64068

Thinking about it (2)

Now, **answer the following questions**:

- Why `region_B.png` have the greatest extent?

Esto se debe a que el carácter B, es el que más se asemeja a un rectángulo, y por ende, el rectángulo que lo encierra tiene un incremento en área menor que el resto de caracteres, lo que hace que el valor de la proporción entre área del contorno y área del rectángulo que lo encierra aumente.

- Is extent invariant to position, orientation or scale? If not, how could we turn it into a invariant feature?

Es invariante a la posición y a la escala, sin embargo, no es invariante a la orientación, como podemos ver en el ejemplo anterior, ya que el rectángulo tiene como restricción que ha de contener todos los píxeles cuyo borde inferior es horizontal y su borde izquierdo es vertical, es decir, el rectángulo no puede estar inclinado. Para hacer que fuese invariante, podríamos quitar esta restricción y permitir que el rectángulo también se pueda orientar de formas distintas.

6.1.3 Building a feature vector

Now that we can compute two different features, compactness (x_1) and extent (x_2), we can build a feature vector (\mathbf{x}) for characterizing each region by concatenating both features, that is, $\mathbf{x} = [x_1, x_2]$.

Before sending to UMA our solution for region description, let's see if these features are discriminative enough to differentiate between the considered characters.

ASSIGNMENT 3: Plotting feature vectors

You task is to plot the feature vectors, computed by the functions `compactness()` and `extent()`, in a 2D-space called the **feature space!**. In such a space, the **x-axis represents the compactness** of a region and the **y-axis its extent**.

In this way, if the descriptions of the considered characters in this space don't appear close to each other, that means that they can be differentiated by relying on those features. **The problem appears if two or more characters have similar features** (their respective points are near). This tell us that **those features are just not enough** for automatically detect the plate characters.

Tip: intro to pyplot.

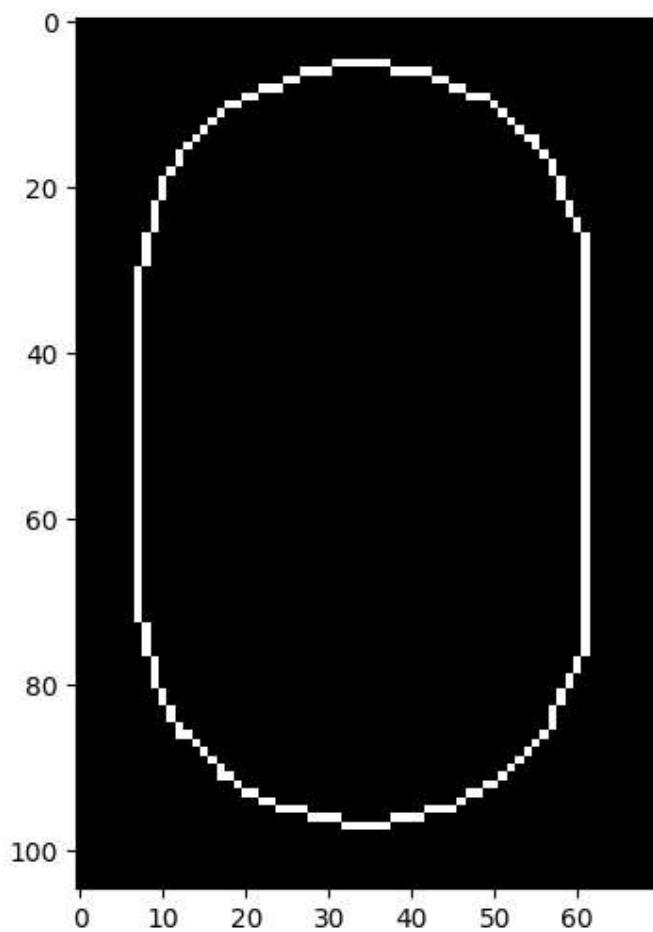
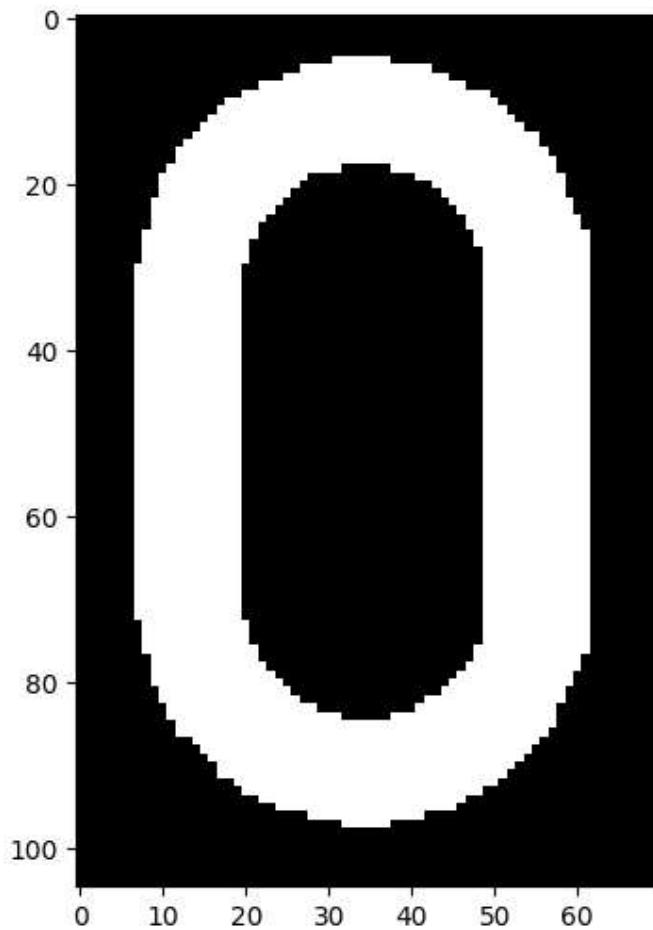
```
In [7]: # Assignment 3
matplotlib.rcParams['figure.figsize'] = (6.0, 6.0)

# Read the images
zero = cv2.imread(images_path + 'region_0.png', 0)
J = cv2.imread(images_path + 'region_J.png', 0)
B = cv2.imread(images_path + 'region_B.png', 0)
six = cv2.imread(images_path + 'region_6.png', 0)

# Build the feature vectors
x_zero = np.array([compactness(zero), extent(zero)])
x_J = np.array([compactness(J), extent(J)])
x_B = np.array([compactness(B), extent(B)])
x_six = np.array([compactness(six), extent(six)])

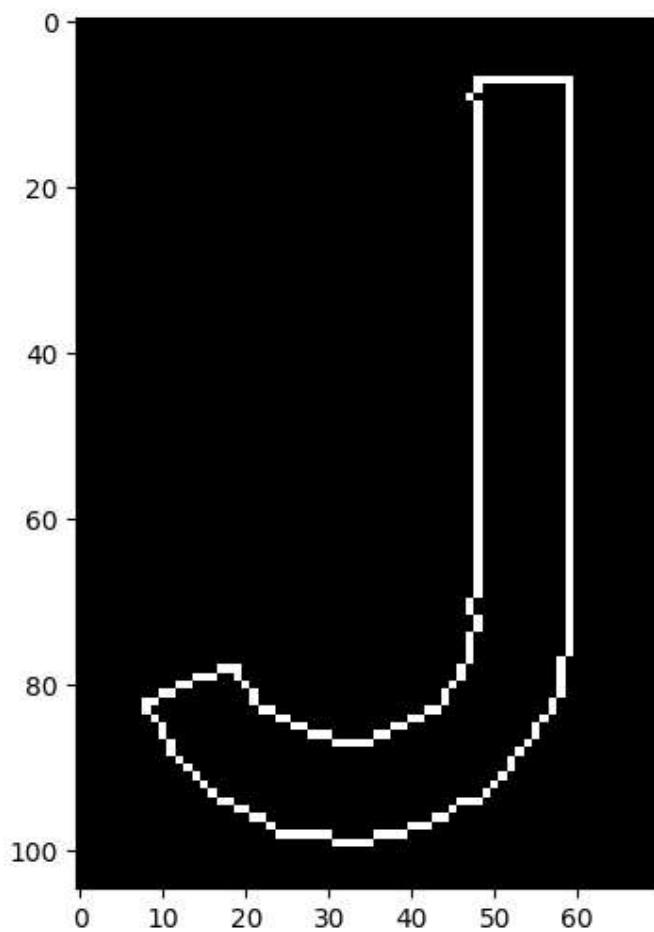
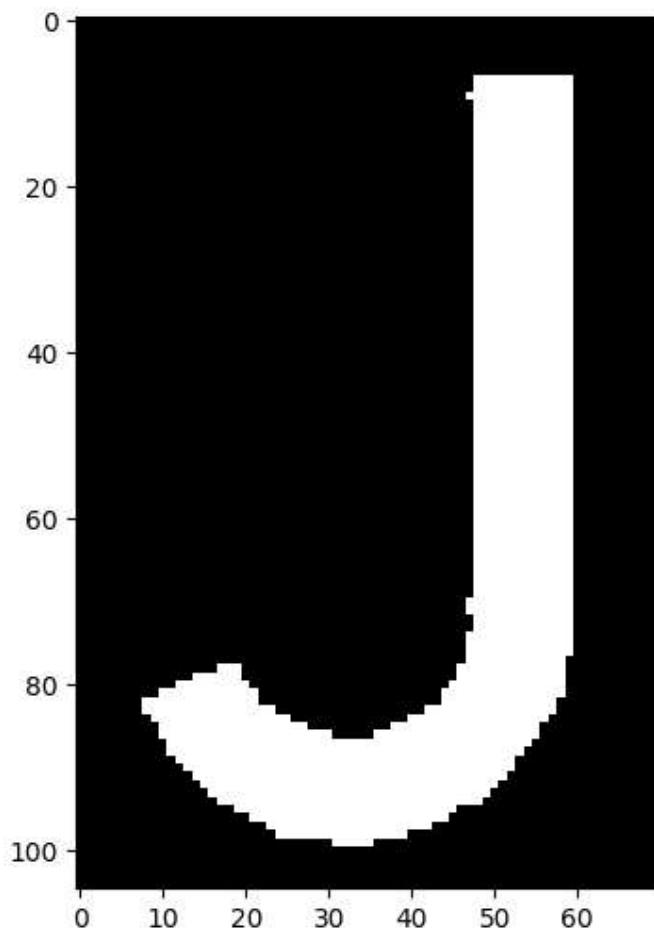
# Define the scatter plot
fig, ax = plt.subplots()
plt.axis([0, 1/(4*np.pi), 0, 1])
plt.xlabel("Compactness")
plt.ylabel("Extent")

# Plot the points
plt.plot(x_zero[0], x_zero[1], 'go')
plt.text(x_zero[0]+0.005, x_zero[1]+0.05, '0', bbox={'facecolor': 'green', 'alpha': 0.5})
plt.plot(x_J[0], x_J[1], 'ro')
plt.text(x_J[0]+0.005, x_J[1]+0.05, 'J', bbox={'facecolor': 'red', 'alpha': 0.5, 'pad': 5})
plt.plot(x_B[0], x_B[1], 'mo')
plt.text(x_B[0]+0.005, x_B[1]+0.05, 'B', bbox={'facecolor': 'magenta', 'alpha': 0.5, 'pad': 5})
plt.plot(x_six[0], x_six[1], 'bo')
plt.text(x_six[0]+0.005, x_six[1]+0.05, '6', bbox={'facecolor': 'blue', 'alpha': 0.5, 'pad': 5})
```



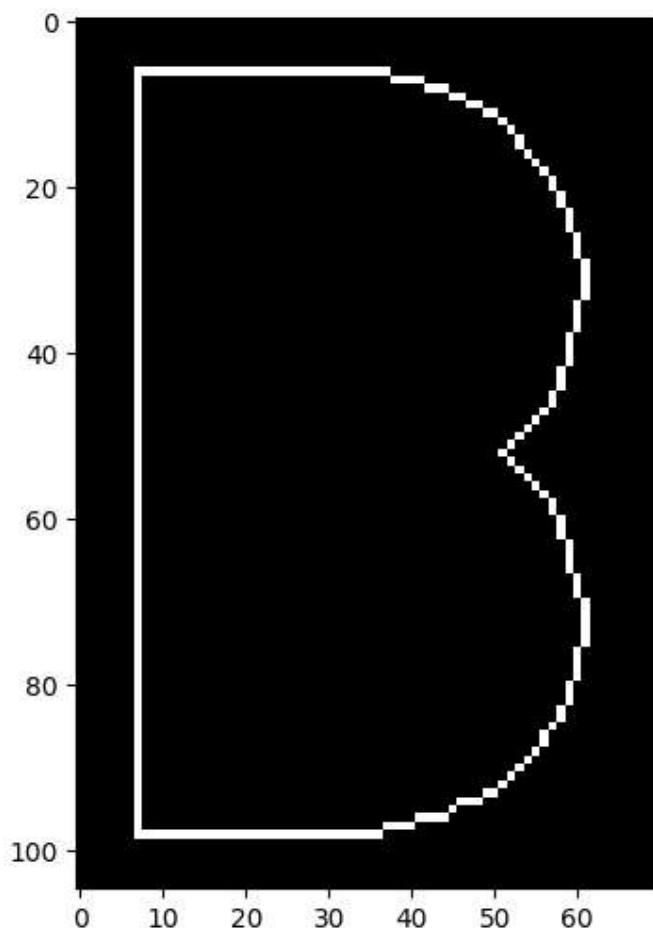
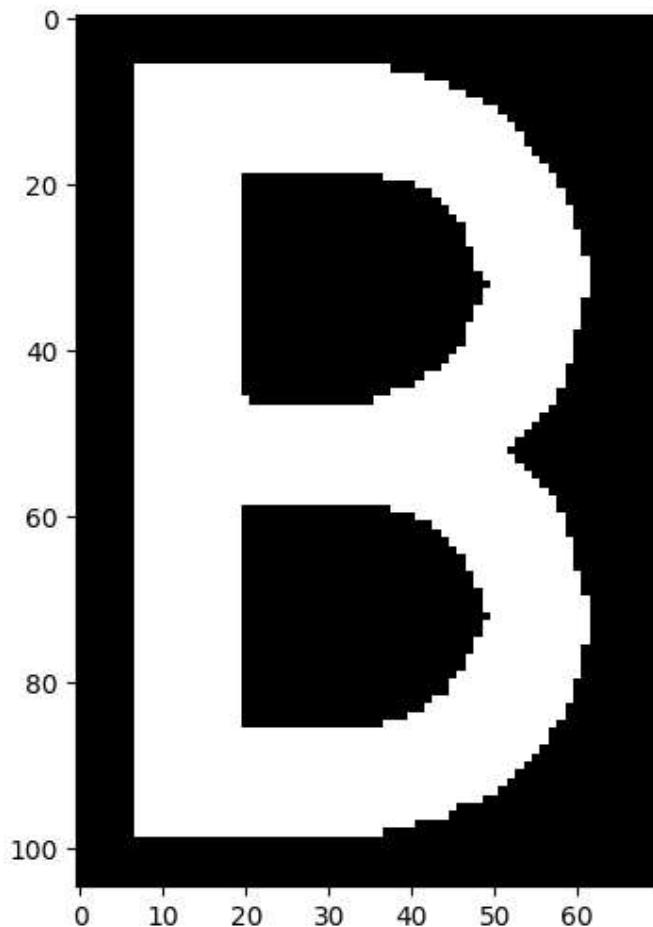
Area: 4307.0

Perimeter: 255.68123936653137



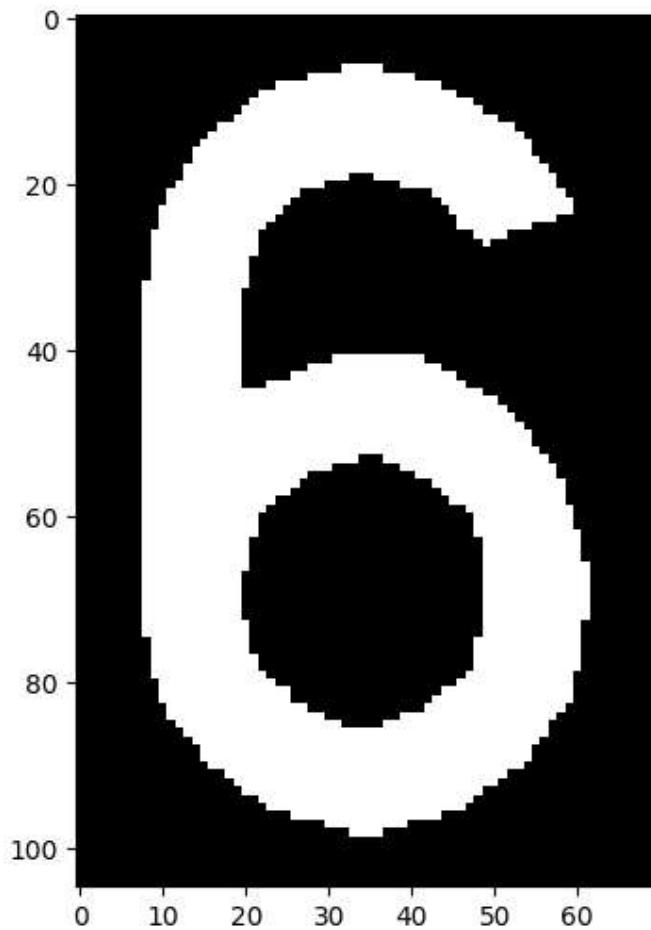
Area: 1386.0

Perimeter: 276.3675310611725



Area: 4498.0

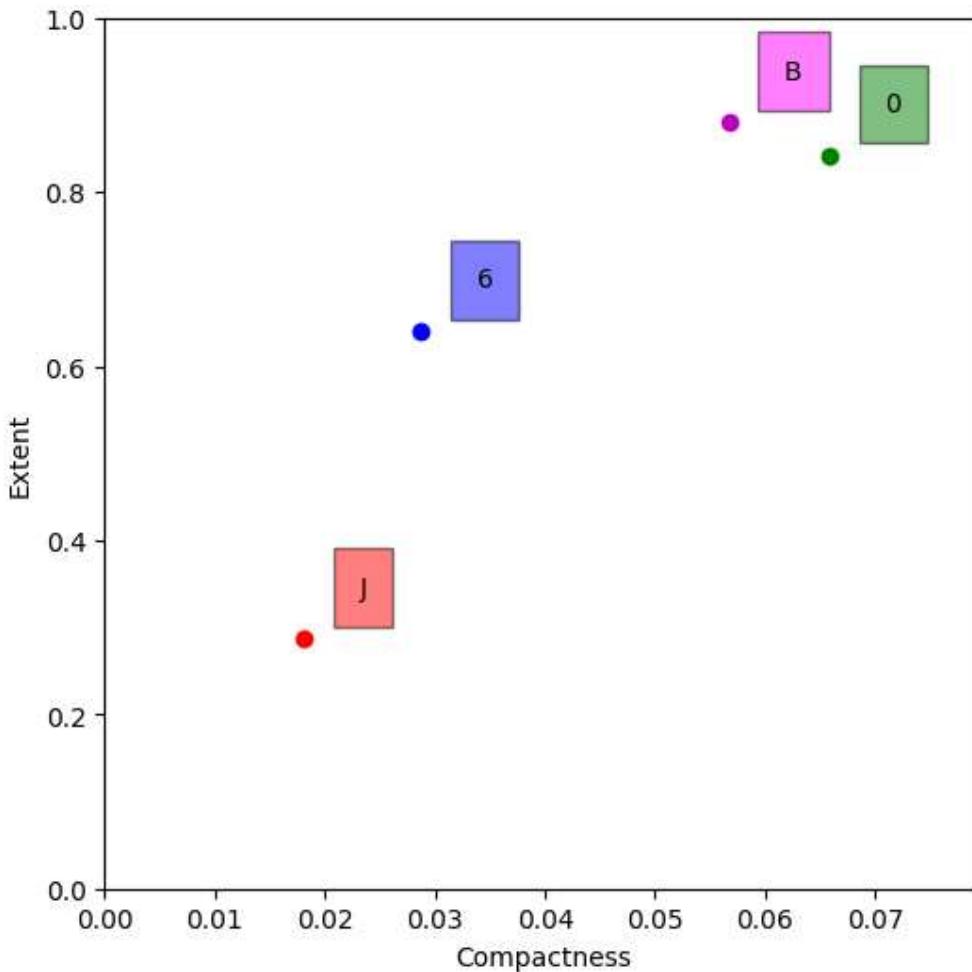
Perimeter: 281.53910398483276



Area: 3217.5

Perimeter: 334.4924215078354

Out[7]: Text(0.033757159783557804, 0.6906810035842295, '6')



Thinking about it (3)

What do you think?

- Are they discriminative enough?

No, ya que podemos ver que los puntos de B y O están muy cerca, ya que en ambos casos sus características son muy similares, entonces no podríamos diferenciarlos con certeza.

- If your answer is no, how could we handle this problem?

Para solucionar este problema, podríamos hacer uso de otras características las cuales nos permitan discernir correctamente estas dos figuras, como las que veremos en el apartado 6.2.

OPTIONAL

Surf the internet looking for **more shape features**, and try to find a pair of them working better than compactness and extent.

END OF OPTIONAL PART

OPTIONAL

Take an image of a car plate, apply the techniques already studied in the course to improve its quality, and binarize it. Then, extract some shape features and check where the numbers/letters are projected in the feature space.

END OF OPTIONAL PART

Conclusion

Great work! You have learned about:

- what is the aim of region descriptors,
- the ideas behind two simple shape descriptors: compactness and extent, and
- to build a vector of features and analyze its discriminative power.

Unfortunately, it seems that those two features are not enough to differentiate the plate characters, so let's try more complex descriptors in the next notebook!

Extra work

Surf the internet looking for **more shape features**, and try to find a pair of them working better than compactness and extent.

6.2 Region-content description

Unlike shape description techniques, which work with regions' contours, region-content description focuses on characterizing the content of segmented regions through their distribution in the image, their textures, etc. Regarding textures, it gives us information about the spatial arrangement of color or intensities in an image or selected region of an image. Textures can be used to help in segmentation or classification of images. Notice that these methods doesn't require binary images as input.

This notebook covers different region-content description techniques:

- 2D image moments ([section 6.2.1](#))
- Hu moments ([section 6.2.2](#))
- Image histogram moments ([section 6.2.3.1](#))
- Co-occurrence matrices ([section 6.2.3.2](#))

Problem context - Car plates

In this notebook, our task is twofold!

Number-plate detection for UMA



Basically, we have to continue with our number-plate detection work looking for a way to obtain a feature vector that distinguishes each character in a Spanish car plate. In this notebook we will try more advanced methods, like **image moments** or **Hu moments**.

Identification of the State of a license plate state

An American company contacted us for developing a **texture description method** that describes a **license plate according to its State of origin** instead of the characters appearing. As you may know, USA uses a different license plates for each state in the country:



You will use some region description methods applied to this problem like **co-occurrence matrices** or **image histogram moments**. Again, your task is to develop a method returning a feature vector that allows for the identification of the State of origin of such license plates.

In [1]:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import matplotlib
matplotlib.rcParams['figure.figsize'] = (15.0, 8.0)
from scipy import stats

images_path = './images/'
```

6.2.1 Image moments

An **image moment** (2D-moment) is the weighted average (or moment) of the intensity of the pixels in the image/region, or a function combining other moments. Moments usually have some attractive property or interpretation, and they can work in both grayscale and color images. For example, when working with 1 dimension (e.g. with a histogram), the moment of order 0 represents the number of pixels in the image, while when dealing with 2 dimensions (e.g. an image) it represents its area, that is, the number of white pixels (if the image is binary).

There are 3 main types of moments:

- **Non-central moments:**

$$m_{ij} = \sum_{y=1}^{\text{rows}} \sum_{x=1}^{\text{cols}} x^i y^j I(y, x)$$

where $I(y, x)$ represents the intensity of the pixel in the (y, x) coordinates of image I .

- **Central moments:**

$$\mu_{ij} = \sum_{y=1}^{\text{rows}} \sum_{x=1}^{\text{cols}} (x - \bar{x})^i (y - \bar{y})^j I(y, x)$$

being $(\bar{x}, \bar{y}) = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right)$ the centroid of the region.

When dealing with big images/regions, it is possible to save some computation time computing the central moments using the non-central ones:

$$\begin{aligned} \mu_{00} &= m_{00} \equiv \mu \\ \mu_{01} &= 0 \\ \mu_{10} &= 0 \\ \mu_{20} &= m_{20} - \mu \bar{x}^2 \\ \mu_{11} &= m_{11} - \mu \bar{x} \bar{y} \\ \mu_{02} &= m_{02} - \mu \bar{y}^2 \\ \mu_{30} &= m_{30} - 3m_{20}\bar{x} + 2\mu \bar{x}^3 \\ \mu_{21} &= m_{21} - m_{20}\bar{y} - 2m_{11}\bar{x} + 2\mu \bar{x}^2 \bar{y} \\ \mu_{12} &= m_{12} - m_{02}\bar{x} - 2m_{11}\bar{y} + 2\mu \bar{y}^2 \bar{x} \\ \mu_{03} &= m_{03} - 3m_{02}\bar{y} + 2\mu \bar{y}^3 \end{aligned}$$

In general, the following formula can be used to retrieve an arbitrary central moment:

$$\mu_{pq} = \sum_m^p \sum_n^q \binom{p}{m} \binom{q}{n} (-\bar{x})^{(p-m)} (-\bar{y})^{(q-n)} m_{mn} \backslash [5pt]$$

- **Scale invariant moments:**

Can be built from central moments by dividing through a properly scaled zero-th central moment:

$$\eta_{ij} = \mu_{ij} / \mu_{00}^{1+((i+j)/2)}$$

where $i + j \geq 2$.

OpenCV pill

OpenCV defines a method for computing some central, non-central and scale-invariant moments called `cv2.moments()`, which gets:

- working with intensity images: a contour (array of 2D points) delimiting the segmented regions.
- working with grayscale images: the image itself.

This function returns a dictionary containing the computed moments.

ASSIGNMENT 1: Computing image moments

What to do? Your first tasks is to complete the method `image_moments()`, which applies the previously mentioned `cv2.moments()` to a binary image, for example a thresholded image containing the numbers of a plate.

```
In [2]: # Assignment 1
def image_moments(region):
    """ Compute moments of the region in a binary image.

    Args:
        region: Binary image

    Returns:
        moments: dictionary containing all moments of the region
    """
    # Compute moments
    moments = cv2.moments(region)

    return moments
```

You can use the next code to **test if the results are correct**, rounding the output of your `image_moments()` function to have 2 decimals:

```
In [3]: region = np.array([[255,255,255,255,255],[255,0,0,0,255],[255,0,0,255,255],[255,0,255,
moments = image_moments(region)

# Round moments for visualization matters
for k, v in moments.items():
    moments[k] = round(v,2)

print(moments)

{'m00': 4335.0, 'm10': 9945.0, 'm01': 8160.0, 'm20': 32895.0, 'm11': 20655.0, 'm02': 24
990.0, 'm30': 115515.0, 'm21': 68595.0, 'm12': 65535.0, 'm03': 83130.0, 'mu20': 10080.
0, 'mu11': 1935.0, 'mu02': 9630.0, 'mu30': -6199.41, 'mu21': -2203.24, 'mu12': 920.29,
'mu03': -164.12, 'nu20': 0.0, 'nu11': 0.0, 'nu02': 0.0, 'nu30': -0.0, 'nu21': -0.0, 'nu
12': 0.0, 'nu03': -0.0}
```

Expected output

```
{'m00': 4335.0, 'm10': 9945.0, 'm01': 8160.0, 'm20': 32895.0, 'm11': 20655.0, 'm02': 24
990.0, 'm30': 115515.0, 'm21': 68595.0, 'm12': 65535.0, 'm03': 83130.0, 'mu20': 10080.
0, 'mu11': 1935.0, 'mu02': 9630.0, 'mu30': -6199.41, 'mu21': -2203.24, 'mu12': 920.29,
'mu03': -164.12, 'nu20': 0.0, 'nu11': 0.0, 'nu02': 0.0, 'nu30': -0.0, 'nu21': -0.0, 'nu
12': 0.0, 'nu03': -0.0}
```

The next code illustrates the moments retrieved from two toy images containing a square, a corner and a line. As you can see, they show differences, so these moments postulate as good descriptors for differentiating them.

Recall some interesting facts:

- Number of white pixels in the image (if binary): $m_{00} = \sum_{y=1}^{\text{rows}} \sum_{x=1}^{\text{cols}} I(y, x)$
- Centroid: $(\bar{x}, \bar{y}) = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right)$
- Eccentricity: $e = \frac{(\mu_{20}-\mu_{02})^2+4\mu_{11}^2}{(\mu_{20}+\mu_{02})^2}$ (*Ratio of the longest chord and longest perpendicular chord*)

```
In [4]: def print_features(moments):
    print('\n'+'Some features:')
    # Eccentricity
    e = ((moments['mu20']-moments['mu02'])**2+4*(moments['mu11']**2))/((moments['mu20']+moments['mu02'])**2)
    print('Number of white pixels =',moments['m00']/255)
    print('Centroid (x,y) = (' ,moments['m10']/moments['m00'],',',
          moments['m01']/moments['m00'],')')
    print('Eccentricity e = ', round(e,2), '\n')

im_square = np.array([[0,0,0,0],[0,255,255,255],[0,255,255,255],[0,255,255,255]], dtype=np.uint8)
im_corner = np.array([[0,255,255,255],[0,255,0,0],[0,255,0,0],[0,255,0,0]], dtype=np.uint8)
im_line = np.array([[0,255,0,0],[0,255,0,0],[0,255,0,0],[0,255,0,0]], dtype=np.uint8)

moments_square = image_moments(im_square)
moments_corner = image_moments(im_corner)
moments_line = image_moments(im_line)

for k, v in moments_square.items():
    moments_square[k] = round(v,2)

for k, v in moments_corner.items():
    moments_corner[k] = round(v,2)

for k, v in moments_line.items():
    moments_line[k] = round(v,2)

plt.subplot(131)
plt.imshow(im_square,cmap='gray')
plt.title('Square image')
print('Moments square image: ' + str(moments_square))
print_features(moments_square)

plt.subplot(132)
plt.imshow(im_corner,cmap='gray')
plt.title('Corner image')
print('Moments corner image: ' + str(moments_corner))
print_features(moments_corner)

plt.subplot(133)
plt.imshow(im_line,cmap='gray')
plt.title('Line image')
print('Moments line image: ' + str(moments_line))
print_features(moments_line)
```

Moments square image: {'m00': 2295.0, 'm10': 4590.0, 'm01': 4590.0, 'm20': 10710.0, 'm11': 9180.0, 'm02': 10710.0, 'm30': 27540.0, 'm21': 21420.0, 'm12': 21420.0, 'm03': 2754.0, 'mu20': 1530.0, 'mu11': 0.0, 'mu02': 1530.0, 'mu30': 0.0, 'mu21': 0.0, 'mu12': 0.0, 'mu03': 0.0, 'nu20': 0.0, 'nu11': 0.0, 'nu02': 0.0, 'nu30': 0.0, 'nu21': 0.0, 'nu12': 0.0, 'nu03': 0.0}

Some features:

Number of white pixels = 9.0

Centroid (x,y) = (2.0 , 2.0)

Excentricity e = 0.0

Moments corner image: {'m00': 1530.0, 'm10': 2295.0, 'm01': 1530.0, 'm20': 4335.0, 'm11': 1530.0, 'm02': 3570.0, 'm30': 9945.0, 'm21': 1530.0, 'm12': 3570.0, 'm03': 9180.0, 'mu20': 892.5, 'mu11': -765.0, 'mu02': 2040.0, 'mu30': 765.0, 'mu21': -510.0, 'mu12': -255.0, 'mu03': 1530.0, 'nu20': 0.0, 'nu11': -0.0, 'nu02': 0.0, 'nu30': 0.0, 'nu21': -0.0, 'nu12': -0.0, 'nu03': 0.0}

Some features:

Number of white pixels = 6.0

Centroid (x,y) = (1.5 , 1.0)

Excentricity e = 0.43

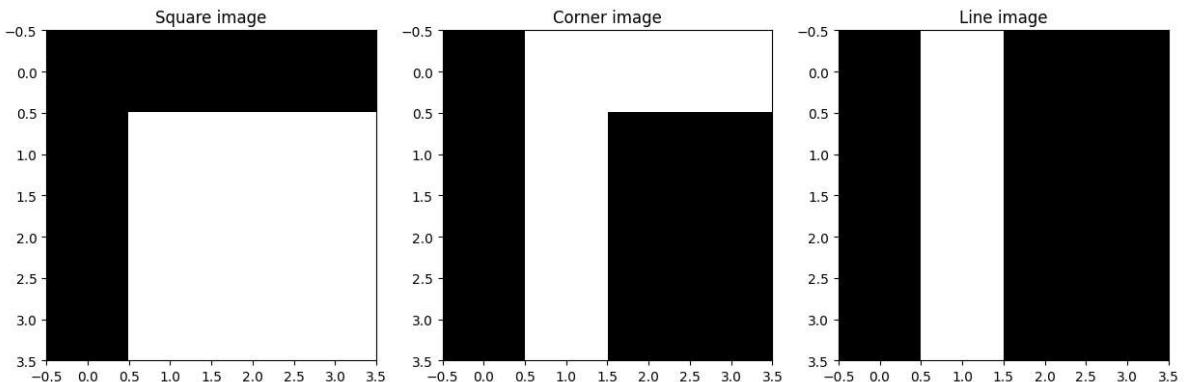
Moments line image: {'m00': 1020.0, 'm10': 1020.0, 'm01': 1530.0, 'm20': 1020.0, 'm11': 1530.0, 'm02': 3570.0, 'm30': 1020.0, 'm21': 1530.0, 'm12': 3570.0, 'm03': 9180.0, 'mu20': 0.0, 'mu11': 0.0, 'mu02': 1275.0, 'mu30': 0.0, 'mu21': 0.0, 'mu12': 0.0, 'mu03': 0.0, 'nu20': 0.0, 'nu11': 0.0, 'nu02': 0.0, 'nu30': 0.0, 'nu21': 0.0, 'nu12': 0.0, 'nu03': 0.0}

Some features:

Number of white pixels = 4.0

Centroid (x,y) = (1.0 , 1.5)

Excentricity e = 1.0



Invariance analysis

Image moments could be good descriptors for addressing the problem posed by UMA. We could compute the moments of a segment region and use them as feature vector

$\mathbf{x} = [x_1, \dots, x_n]^T$. Also, as i and j in the previous equations can take any integer, we could have a feature vector of any desired length. For example, if you design a region descriptor system that considers the first three non-central, central, and scale invariant moments, you will use: $\mathbf{x} = [m_{00}, m_{10}, m_{01}, \mu_{20}, \mu_{11}, \mu_{02}, \eta_{20}, \eta_{11}, \eta_{02}]^T$.

In the context of the number-plate detection problem, the results have to be (at least) position and scale invariants, because a car could stop closer or further away from the camera and in different positions (rotation is not that important).

To check if these moments have such invariances, you are going to compute the moments of a region, as well as of a scaled and rotated versions of it. To visually check the results, we are going to use bar charts, showing the moments for the original, rotated and scaled images, which should look like this:



ASSIGNMENT 2: Checking invariances

Complete the method `compare_moments()`, which takes:

- a list of labels for the bar chart, and
- three lists containing the moments of a region, and its rotated and scaled versions. These methods, using said arguments, plots the chart bar previously showed.

For the plot you can use `plt.bar(labels, values)`, where `labels` is a list of strings (e.g. `keys` of the dictionary of moments) and `values` a list of numbers (e.g. `values` of such dictionary).

```
In [5]: # Assignment 2
def compare_moments(labels, moments, moments_rotated, moments_scaled):
    """ Plot a bar chart comparing the three input moment arrays

    Args:
        labels: Labels of the bar chart
        moments: list containing moments of a original region
        moments_rotated: list containing moments of the original region, but previous
        moments_scaled: list containing moments of the original region, but previous

    """
    # Show original moments
    plt.subplot(131)
    plt.title("Original")
    plt.bar(labels, moments)

    # Show rotated moments
    plt.subplot(132)
    plt.title("Rotated")
    plt.bar(labels, moments_rotated)

    # Show scaled moments
    plt.subplot(133)
    plt.title("Scaled")
    plt.bar(labels, moments_scaled)
```

We are going to separately analyze the invariance of:

- the **non-central** (first 10 values of moment dictionary),
- **central** (following 7), and
- **scale-invariant** (last 7) moments.

Take a look at the result of the previous assignment to check this!

First, let's compute the moments from an initial image, a rotated version of it (90 degrees), and a scaled version (by a factor of 2 in both horizontal and vertical axes). Finally show those images.

Hint: You can rotate a numpy array using `np.rot90()` and scale an image using `cv2.resize()`, although there are many more options.

```
In [6]: # Read binary image and compute moments
region = cv2.imread(images_path + 'region_6.png', 0)
moments = image_moments(region)

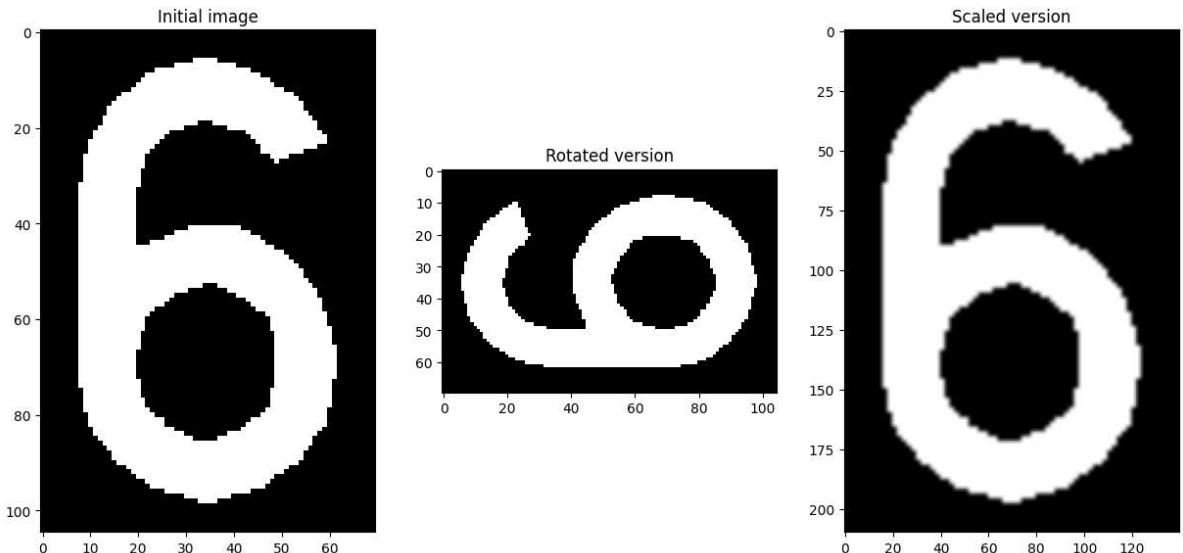
# Rotate image and compute moments
region_rotated = np.rot90(region)
moments_rotated = image_moments(region_rotated)

# Resize image and compute moments
region_scaled = cv2.resize(region, dsize=None, fx=2, fy=2) # keep the dsize=None
moments_scaled = image_moments(region_scaled)

# Show the initial image
plt.subplot(131)
plt.title('Initial image')
plt.imshow(region, cmap='gray')

# Show the rotated version
plt.subplot(132)
plt.title('Rotated version')
plt.imshow(region_rotated, cmap='gray')

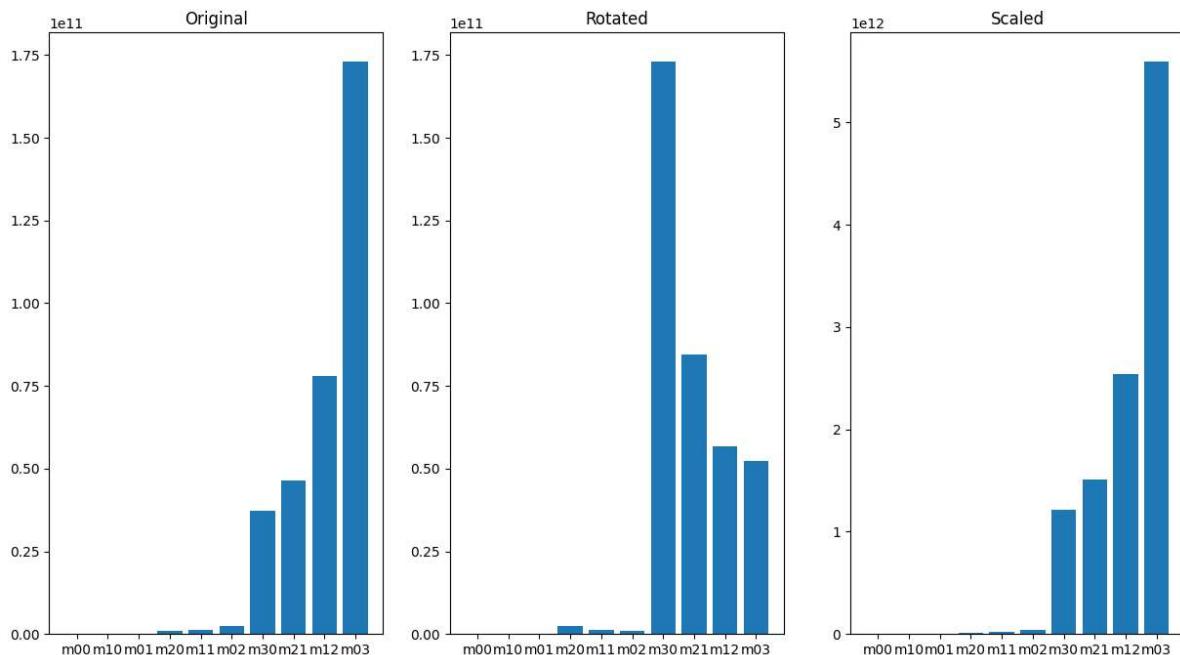
# Show the scaled version
plt.subplot(133)
plt.title('Scaled version')
plt.imshow(region_scaled, cmap='gray');
```



Now, let's start comparing the **Non-central moments** of the three images! *Hint: pay special attention to the scale of the axes in the plot!*

```
In [7]: # Compare results for non-central moments
labels = list(moments.keys())[:10]
non_central_moments = list(moments.values())[:10]
non_central_rotated = list(moments_rotated.values())[:10]
non_central_scaled = list(moments_scaled.values())[:10]

compare_moments(labels, non_central_moments, non_central_rotated, non_central_scaled)
```



Thinking about it (1)

Now, answer the following questions:

- Are these moments invariant to rotation?

Podemos ver claramente que el histograma original y el rotado son diferentes, así que no son invariantes a la rotación.

- Are these moments invariant to scale?

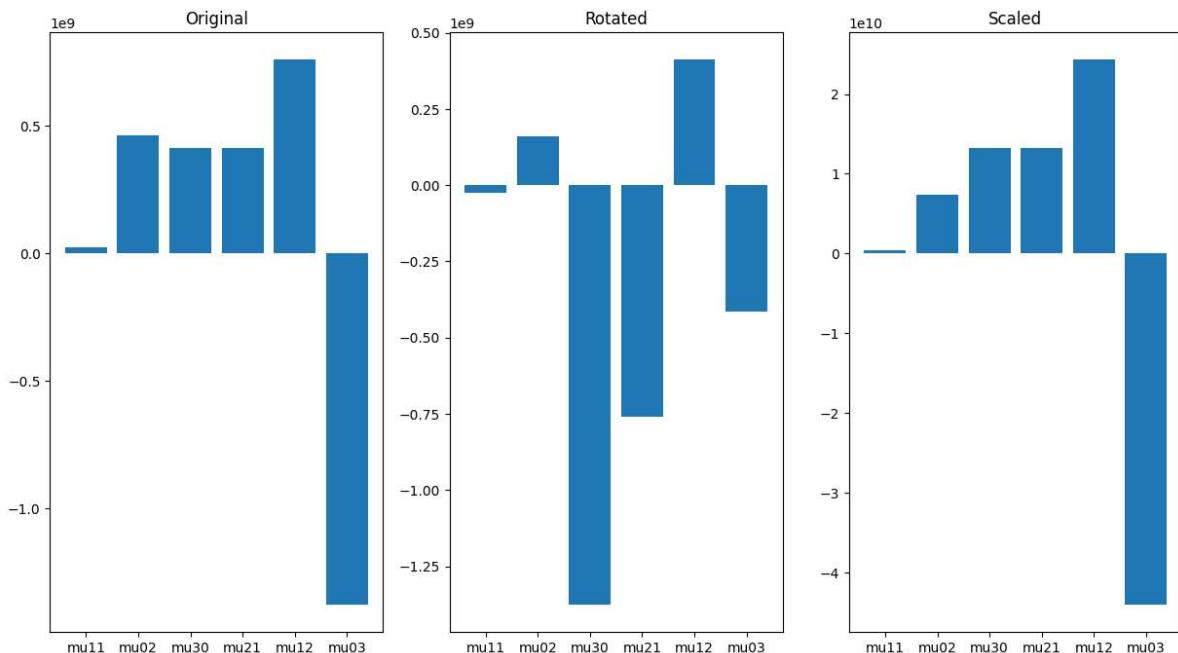
Los valores de la versión escalada son del orden de 1e12, mientras que los de la versión original son del orden 1e11, por tanto, tampoco son invariantes a la escala.

Let's continue with **central moments!**

```
In [8]: # Compare results for central moments

labels = list(moments.keys())[11:17]
central_moments = list(moments.values())[11:17]
central_rotated = list(moments_rotated.values())[11:17]
central_scaled = list(moments_scaled.values())[11:17]

compare_moments(labels,central_moments,central_rotated,central_scaled)
```



Thinking about it (2)

Now, answer the following questions:

- Are these moments invariant to rotation?

No, no son invariantes a la rotación, pasa lo mismo que en el apartado anterior, los histogramas son diferentes en la original y la rotada./i>

- Are these moments invariant to scale?

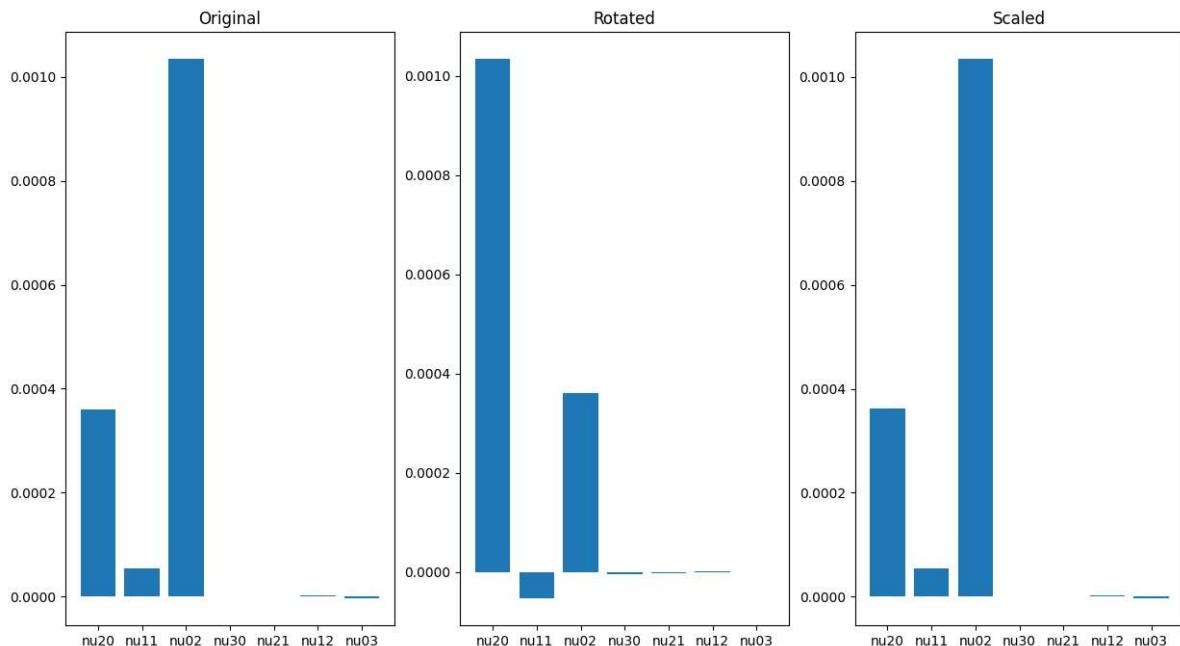
De forma similar, la versión escalada tiene valores del orden 1e12, mientras que la normal del orden 1e11, por tanto, tampoco es invariante a la escala.

And we conclude with **scale-invariant moments**.

```
In [9]: # Compare results for scale-invariant moments

labels = list(moments.keys())[len(list(moments.keys()))-7:]
invariant_moments = list(moments.values())[len(list(moments.keys()))-7:]
invariant_rotated = list(moments_rotated.values())[len(list(moments.keys()))-7:]
invariant_scaled = list(moments_scaled.values())[len(list(moments.keys()))-7:]

compare_moments(labels,invariant_moments,invariant_rotated,invariant_scaled)
```



Thinking about it (3)

Now, answer the following questions:

- Are these moments invariant to rotation?

No son invariantes a la rotación, pasa lo mismo que en los dos casos anteriores.

- Are these moments invariant to scale?

En este caso, podemos ver que sí son invariantes a la escala, ya que los histogramas de la imagen original y la escalada son iguales.

6.2.2 Hu moments

The **Hu moments** (published in 1962 by Ming-Kuei Hu) are a set of 7 particular moments showing **interesting invariance properties**. They are calculated using scale-invariant ones:

$$\begin{aligned}
 v_1 &= \eta_{20} + \eta_{02} \\
 v_2 &= (\eta_{20} - \eta_{02}) + 4\eta_{11}^2 \\
 v_3 &= (\eta_{20} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2 \\
 v_4 &= (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2 \\
 v_5 &= (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[(\eta_{21} + \eta_{03})^2 - 3(\eta_{30} + \eta_{12})^2] \\
 v_6 &= (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} - \eta_{03})^2 + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03})] \\
 v_7 &= (3\eta_{21} - \eta_{03})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{30} + \eta_{12})^2] + (\eta_{30} - 3\eta_{12})(\eta_{21} + \eta_{03})[(\eta_{21} + \eta_{03})^2 - 3(\eta_{21} + \eta_{03})^2]
 \end{aligned}$$

being: $\eta_{pq} = \frac{\mu_{pq}}{\mu_{00}^\gamma}$ $\gamma = \frac{p+q}{2} + 1$

OpenCV pill

OpenCV provides a method to retrieve the Hu moments, called (wait for it...)

`cv2.HuMoments()` !. This method takes as input the dictionary of moments returned by

`cv2.moments`. Recall that the scale-invariant moments used for their computation are the `nuij` moments in the dictionary.

ASSIGNMENT 3: Exploring Hu moments invariances

Previously, we tested the invariances of non-central, central and scale-invariant moments. Now, **we are interested in checking the invariances of the Hu moments**, so we can verify if they are more suitable for the UMA parking problem.

For that, use your brand-new `compare_moments()` function in the same way as in the previous exercises.

```
In [10]: # Assignment 3

# Read binary image and compute Hu moments
region = cv2.imread(images_path + 'region_J.png', 0)
moments = image_moments(region)
hu = cv2.HuMoments(moments)

# Rotate image and compute Hu moments
region_rotated = np.rot90(region)
moments_rotated = image_moments(region_rotated)
hu_rotated = cv2.HuMoments(moments_rotated)

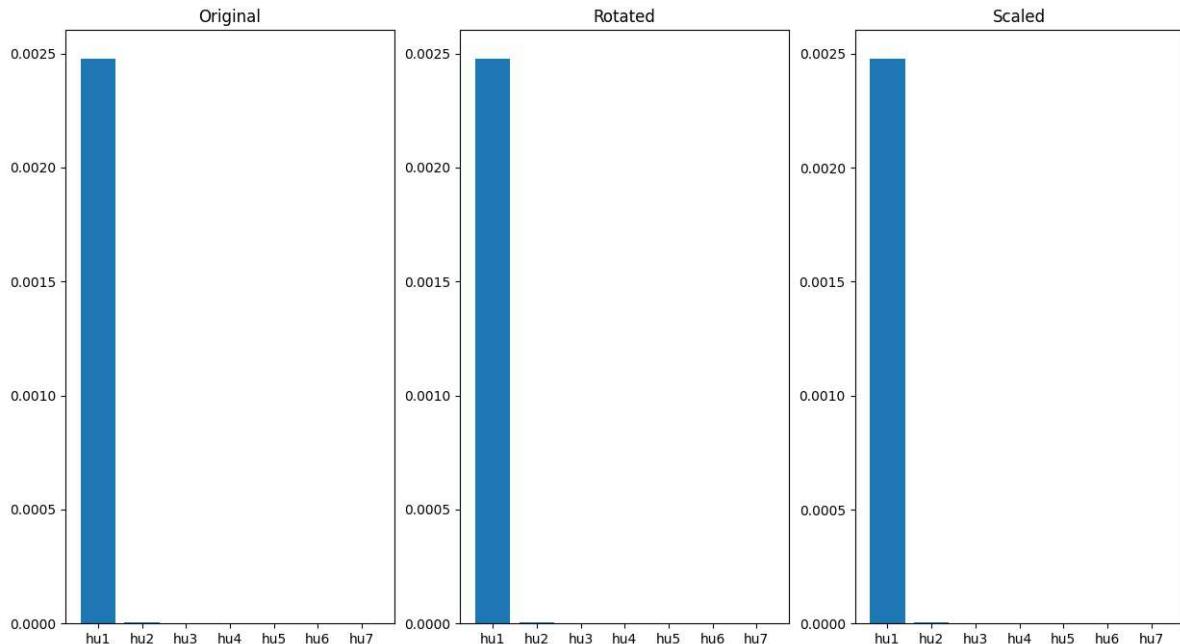
# Resize image and compute Hu moments
region_scaled = cv2.resize(region, dsize=None, fx=2, fy=2) # keep the dsize=None
moments_scaled = image_moments(region_scaled)
hu_scaled = cv2.HuMoments(moments_scaled)

# Compare results for Hu moments
labels = ["hu1", "hu2", "hu3", "hu4", "hu5", "hu6", "hu7"]
hu_moments = hu.flatten()
hu_rotated = hu_rotated.flatten()
hu_scaled = hu_scaled.flatten()

compare_moments(labels, hu_moments, hu_rotated, hu_scaled)

print('hu_moments: ' + str(hu_moments))
print('hu_rotated: ' + str(hu_rotated))
print('hu_scaled: ' + str(hu_scaled))

hu_moments:[2.47861399e-03 3.77838124e-06 5.43870558e-09 1.18195649e-09
 1.41101979e-18 7.49092356e-13 2.64376868e-18]
hu_rotated:[2.47861399e-03 3.77838124e-06 5.43870558e-09 1.18195649e-09
 1.41101979e-18 7.49092356e-13 2.64376868e-18]
hu_scaled: [2.47958786e-03 3.77838124e-06 5.43870558e-09 1.18195649e-09
 1.41101979e-18 7.49092356e-13 2.64376868e-18]
```



Thinking about it (4)

Now, answer the following questions:

- Are these moments invariant to rotation?

Sí, son invariantes a la rotación ya que el histograma original y el rotado son iguales.

- Are these moments invariant to scale?

De forma similar, son invariantes a la escala, ya que el histograma original y el escalado son iguales.

- Now that you can deal with different ways to describe a binary region, **what descriptor would you use** for the UMA parking problem? **Why?**

Usaría los momentos de Hu, ya que la imagen obtenida de la matrícula dependerá de la distancia a la que se encuentre el coche, por tanto puede variar la escala y la rotación de la imagen, entonces usando los momentos de Hu obtendremos una mayor exactitud.

6.2.3 Texture

The previous techniques are useful for describing the distribution of the regions over the image. There is another branch of algorithms that pursuit the description of regions by **characterizing the texture of the pixels they enclose**. Such methods measure the spatial arrangement of the colors/intensities in a region, providing information about their smoothness, coarseness, and regularity. In this way, if a region does not present changes in intensity, we say that it is a untextured region.



Usually, texture descriptors have spatial (position, orientation and scale) and radiometric (contrast and brightness) invariance. We are going to explore two of these descriptors:

- 1D moments of the histogram, and
- Gray Level Co-Occurrence Matrix (GLCM)

6.2.3.1 1D moments of histogram

The **central moments of the histogram** of the pixels within a region statistically describes the frequency of their intensities. They permit us to compactly describe the region through a feature vector containing a few features. They are computed using the equation:

$$\mu_n = \sum_{i=0}^{255} (z_i - \bar{z})^n h(z_i)$$

where $h(z_i)$ represents the value stored in the histogram $h(\cdot)$ for the intensity z_i . Keep in mind that:

- $\mu_0 = \mu_1 = 0$
- μ_2 : variance (contrast)
- μ_3 : histogram skew
- μ_4 : histogram uniformity

However, they have a serious drawback: they don't encode pattern structures, so different textures may have similar histograms:



Nevertheless, they can be a good option depending on the application, so do not underestimate them!

ASSIGNMENT 4: Analyzing histograms

In order to play a bit with these moments, we move to our second application: the state recognition in USA car plates. Two examples of such license plates:



Two examples of USA car plates

As we can see, the main difference between them is the texture in the plate background, as each state has a different one. Let's try 1D moments of the histogram for describing those textures!

Your first task is to plot the histogram of the previous images: `nevada.jpg` and `hawaii.jpg`, and check if the shape of the histograms is enough to differentiate them. *Hint: recall the `np.ravel()` function*

In [11]:

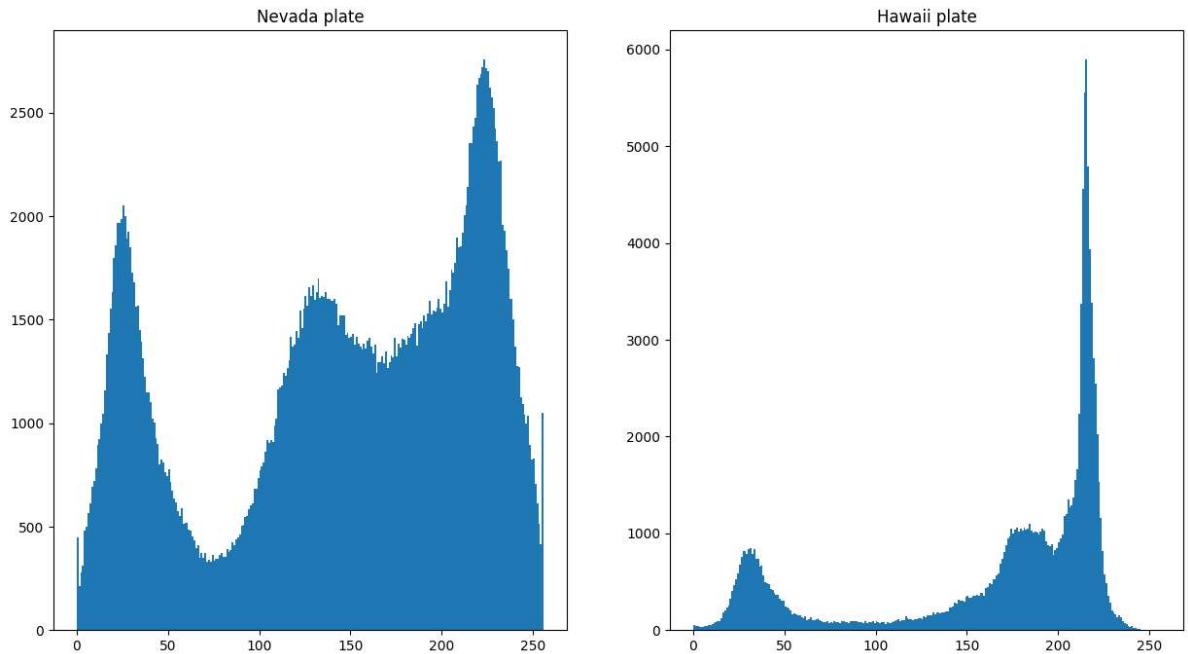
```
# Assignment 4

# Read images
nevada = cv2.imread(images_path + 'nevada.jpg', 0)
hawaii = cv2.imread(images_path + 'hawaii.jpg', 0)

# Show first one histogram
plt.subplot(121)
plt.title("Nevada plate")
plt.hist(nevada.ravel(), 256, [0, 256])

# And the second one!
plt.subplot(122)
plt.title("Hawaii plate")
```

```
plt.hist(hawaii.ravel(),256,[0,256])
plt.show()
```



Now, let's complete the method `histogram_moments()` that implements the retrieval of the central moments of the histogram shown above. This method takes as input:

- an image, and
- the number of moments to be calculated

and returns an array containing those moments of the image's histogram.

```
In [12]: def histogram_moments(image,k):
    """ Compute central moments of the histogram of an image.

    Args:
        image: input image
        k: number of moments to compute

    Returns:
        histogram_moments: array containing the histogram moments
    """

    # Compute histogram
    hist = cv2.calcHist([image],[0],None,[256],[0,256]) # Keep the None in this function

    # Compute mean average intensity/brightness of the image
    z_mean = np.dot(hist[hist.nonzero()[0]].flatten(),hist.nonzero()[0].flatten())/hist.sum()

    # Compute moments
    histogram_moments = np.zeros(k)
    for i in range(k):
        moment = 0.0
        for z in range(1,256):
            moment += float(z-z_mean)**i * float(hist[z])

        histogram_moments[i] = moment/hist.sum()

    # The previous code could be replaced by just one line!
    # histogram_moments[i] = np.average((np.arange(1,256) - z_mean)**i, weights =
```

```
    return(histogram_moments)
```

You can use the next code to **test if the results are correct**:

```
In [13]: image = np.array([[10,60,20],[60,22,74],[72,132,2]], dtype=np.uint8)

moments = histogram_moments(image,6)

print(moments)
```

```
[1.0000000e+00 0.0000000e+00 1.50795062e+03 3.83609108e+04
 6.08670794e+06 3.62329032e+08]
```

```
C:\Users\Usuario\AppData\Local\Temp\ipykernel_4208\2433238964.py:24: DeprecationWarning:
g: Conversion of an array with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you extract a single element from your array before performing this operation. (Deprecated NumPy 1.25.)
```

```
moment += float(z-z_mean)**i * float(hist[z])
```

Expected output:

```
[1.0000000e+00 0.0000000e+00 1.50795062e+03 3.83609108e+04
 6.08670794e+06 3.62329032e+08]
```

Invariance analysis

Now that we can obtain the first `k` moments of an image histogram, we are going to see if this method is invariant to scale and rotation. As in the UMA parking problem, our solution must be scale invariant, so let's check if it is.

ASSIGNMENT 5: Checking the invariance of 1D moments

What to do? Check if **the first six 1D moments** of the histogram of an image, a rotated version of it, and a scaled version, are the same. Use `np.array_equal()` for that.

```
In [14]: # Assignment 5

# Read image and compute histogram moments
image = cv2.imread(images_path + 'hawaii.jpg',0)
moments = histogram_moments(image,6)

# Rotate image and compute histogram moments
image_rotated = np.rot90(image)
moments_rotated = histogram_moments(image_rotated,6)

# Resize image and compute histogram moments
image_scaled = cv2.resize(image, dsize=None, fx=2, fy=2) # keep the dsize=None
moments_scaled = histogram_moments(image_scaled,6)

# Compare results
print("Rotation invariance: ", np.array_equal(moments,moments_rotated))
print("Scale invariance: ", np.array_equal(moments,moments_scaled))

# Show the initial image
plt.subplot(131)
plt.title('Initial image')
plt.imshow(image, cmap='gray')

# Show the rotated version
plt.subplot(132)
```

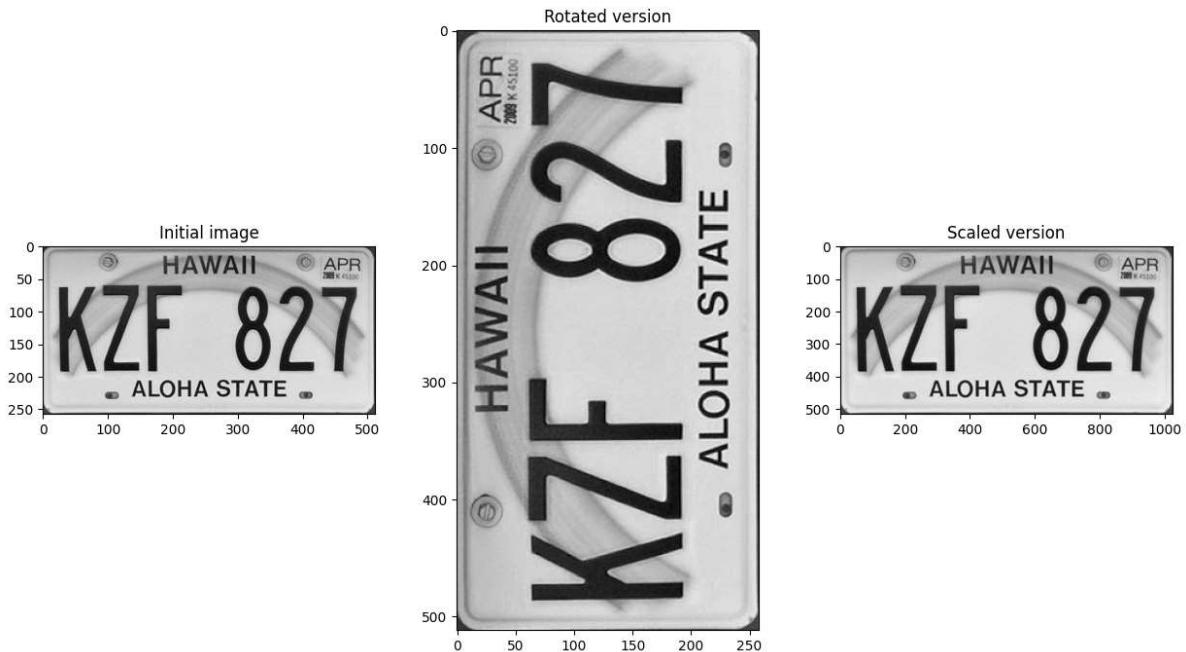
```

plt.title('Rotated version')
plt.imshow(image_rotated, cmap='gray')

# Show the scaled version
plt.subplot(133)
plt.title('Scaled version')
plt.imshow(image_scaled, cmap='gray');

```

C:\Users\Usuario\AppData\Local\Temp\ipykernel_4208\2433238964.py:24: DeprecationWarning:
g: Conversion of an array with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you extract a single element from your array before performing this operation. (Deprecated NumPy 1.25.)
moment += float(z-z_mean)**i * float(hist[z])
Rotation invariance: True
Scale invariance: False



Thinking about it (5)

Now, answer the following questions:

- Is it invariant to rotation? If not, how can we turn this method into it?

Es invariante a la rotación ya que la frecuencia de las intensidades de los píxeles se mantiene constante.

- Is it invariant to scale? If not, how can we turn this method into it?

No es invariante a la escala, ya que se duplica el número de píxeles, por tanto los valores de los momentos cambiarán. Para solucionar esto podríamos convertir los momentos en momentos de escala invariante, de forma similar a como lo hemos hecho con los momentos centrales 2D, es decir, dividiendo los momentos centrales por un momento central cero propiamente escalado.

6.2.3.2 Co-occurrence matrix

Another technique also obtaining a statistical representation of the texture within a region is the **co-occurrence matrix**, a square matrix $A(i, j)$ in which:

- i and j represent intensity values (e.g. 0 to 255).

- The entry a_{ij} indicates how many times the intensity i co-occurs with intensity j in some designated spatial relationships P (texture pattern).
- P is given by a displacement vector $d = [dr, dc]$, where dr and dc are the displacement in rows and columns, respectively.

 No description has been provided for this image

The issue with this approach is how to select the appropriate displacement d . Once the co-occurrence matrix of a region has been computed, a number of features can be extracted from it:

- **Maximum probability:** gives us the strongest response to the texture pattern $P \setminus [1pt]$

$$\max_{ij} c_{ij}$$

- **Energy:** minimum when all the entries c_{ij} are identical (maximum uniformity)

$$\sum_{i=0}^{255} \sum_{j=0}^{255} c_{ij}^2$$

- **Entropy:** measure randomness. Maximum value when all the entries c_{ij} are identical (maximum entropy → minimum energy)

$$-\sum_{i=0}^{255} \sum_{j=0}^{255} c_{ij} \log c_{ij}$$

- **Order k central moment**

$$\sum_{i=0}^{255} \sum_{j=0}^{255} (i - j)^k c_{ij}$$

ASSIGNMENT 6: Computing co-occurrence matrices

Let's implement the method `co_occurrence_matrix_features()`, which has to compute the normalized co-occurrence matrix of `image` using the displacement vector `[dr, dc]` and normalizes it, obtaining `C(i, j)`. Note that `dr` and `dc` may take positive or negative values. Thereby, it takes as inputs:

- an image,
- a 2-size displacement vector, and
- a number of central moments to compute.

and returns:

- a feature vector with size $3 + n_moments$ being: `[max_prob, energy, entropy, moments (optional)]`

In [15]: `# Assignment 6`

```
def co_occurrence_matrix_features(image, d, n_moments):
    """ Compute features from a image using a co-occurrence matrix.

    Args:
        image: Binary image
        d: displacement vector
        n_moments: number of moment to be computed

    Returns:
        features: feature vector
```

```

"""
(n_r, n_c) = image.shape
co = np.zeros((256,256))
features = np.zeros(3+n_moments)

# Compute image ranges to iterate from displacement vector

if d[0] >= 0:
    range_rows = range(0, n_r-d[0], 1)
else:
    range_rows = range(-d[0], n_r, 1)

if d[1] >= 0:
    range_columns = range(0, n_c-d[1], 1)
else:
    range_columns = range(-d[1], n_c, 1)

# Compute co-occurrence matrix
for r in range_rows:
    for c in range_columns:

        i = image[r,c]
        j = image[r+d[0],c+d[1]]

        co[i,j] += 1

# Normalize co-occurrence matrix
co = co/np.sum(co)

# Maximum probability
features[0] = co.max()

# Energy
energy = 0
for i in range(len(co)):
    for j in range(len(co[i])):
        energy += co[i][j]**2
features[1] = energy

# Entropy
mask = np.where(co!=0, True, False)
entropy = 0
for i in range(len(co)):
    for j in range(len(co[i])):
        if(mask[i][j]):
            entropy += co[i][j]*(np.log(co[i][j]))
features[2] = entropy

# Central moments
for k in range(n_moments):
    moment = 0
    for i in range(co.shape[0]):
        for j in range(co.shape[1]):
            moment += ((i-j)**k)*co[i,j]

    features[3+k] = moment

return np.round(features,5)

```

You can use the next code to **test if the results are right**:

```
In [16]: np.set_printoptions(suppress=True)

image = np.array([[10,60,20],[60,22,74],[72,132,2]], dtype=np.uint8)

features = co_ocurrence_matrix_features(image,d=[1,-2],n_moments=4)

print(features)
```

[0.5 0.5 -0.69315 1. -19.
 802. -31996.]

Expected output:

[0.5 0.5 0.693 1. -19. 802. -31996.]

ASSIGNMENT 7: Studying the invariance of co-occurrence matrices

Compare the results returned by `co_ocurrence_matrix_features()` when using the original image `hawaii.jpg`, with those returned by a rotated or scaled version of it.

```
In [17]: # Assignment 7

# Read image and compute co-occurrence matrix features
image = cv2.imread(images_path + 'hawaii.jpg',0)
features = co_ocurrence_matrix_features(image,d=[1,-2],n_moments=4)

# Rotate image and compute co-occurrence matrix features
image_rotated = np.rot90(image)
features_rotated = co_ocurrence_matrix_features(image_rotated,d=[1,-2],n_moments=4)

# Resize image and compute co-occurrence matrix features
image = cv2.resize(image, dsize=None, fx=2, fy=2) # keep the dsize=None
features_scaled = co_ocurrence_matrix_features(image,d=[1,-2],n_moments=4)

# Compare results
print("Features original ", features, "\n Features rotated ", features_rotated, "\n Fea
```

Features original [0.01232 0.00135 -8.25008 1. -0.15981 1351.0
7967 -3613.80618]
Features rotated [0.01192 0.00134 -8.24679 1. -0.52598 1226.84705 -12101.7668]
Features scaled [0.01745 0.00233 -7.80778 1. -0.06358 358.81004 -346.16178]

Thinking about it (6)

Now, answer the following question:

- Compare the invariance of each feature in the feature vector and comment why it is invariant or not to rotation and scale.

La probabilidad máxima no es invariante a la rotación ni a al escalado. La energía es invariante a la rotación pero no es invariante al escalado. La entropía es invariante a la rotación pero no al escalado. Los momentos no son invariantes salvo en el caso en el que k = 0 (orden 0). Las invarianzas en el caso del escalado se producen debido a que al tener un mayor número de píxeles, el número de repeticiones de los patrones aumenta, pero no de forma proporcional, ya que el escalado no es perfecto. Debido a esto, los resultados para la

Imagen escalada serán distintos. En el caso de la imagen rotada, pienso que las diferencias se deben a que en ciertos casos, no se reconocen correctamente los patrones al estar rotada la imagen, y esos casos hacen que los valores de las características varíen levemente. Donde se notan más los cambios es en los momentos, que como ya hemos visto antes, no son invariantes a la rotación ni escala, y es por eso que los valores son tan dispares.

Conclusion

Awesome! This was a laborious and dense notebook, but you carried it through to the end!

In this notebook you have learned:

- how to compute non-central, central, scale-invariant and Hu moments for describing a region, and apply them to the plate number recognition problem.
- how to describe textures using 1D moments of the histogram and co-occurrence matrices, using them in the context of the state identification problem.

Extra

Usually, the co-occurrence matrices **of the image rotated** 45, 90, and 135 degrees are also calculated. **What do you think this is due to?**

Implement this new procedure for co-occurrence matrices and then, check again the invariances. **What happened?**