5.1 Contour-based techniques for image segmentation

Image segmentation is the process of **assigning a label to every pixel in an image** such that pixels with the same label share certain characteristics. The techniques addressing those problems can be classified into to major groups:

- Those producing regions with similar properties (*e.g.*, intensity, color, texture, or location in the image), addressed by classical segmentation methods and some DNN models (like SAM).
- Those segmenting regions with **similar meaning** (*e.g.*, **semantic segmentation** where pixels belonging to the same object category are assigned to the same region, or **instance segmentation** where each region represents an object, see Fig.1). This is faced by Deep NN methods.

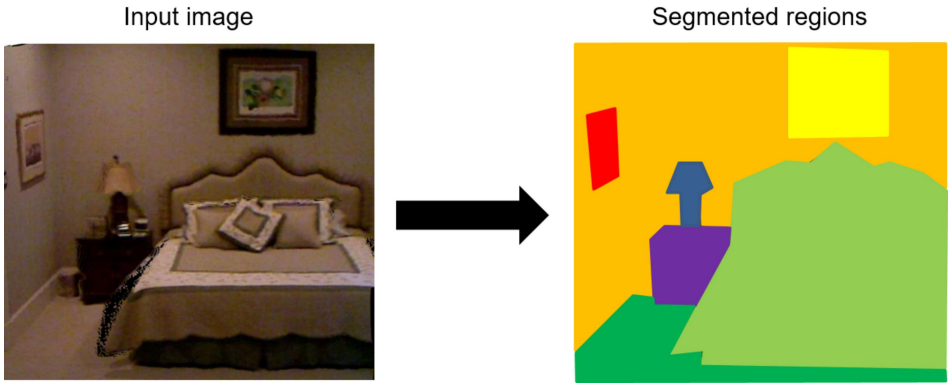


Fig. 1: Example of instance segmentation where each region corresponds to an object in the scene.

Classical methods can be further classifed into:

- **Contour-based techniques**, which attempt to identify the image regions by detecting their contours. Edge pixels enclose a region with similar intensities.
- **Clustering-based techniques** that group together pixels that are share some properties.

In this book we are going to experience both, starting with **contour-based techniques**, whose are based on detecting specific contours in the image (e.g. circles). In this context, image contours are defined as edge pixels that enclose a region.

Contour-based techniques could be roughly classified into:

- **Local tecniques.** Try to segment regions by detecting closed contours, which typically enclose pixels with similar intensities.
  - LoG + zero crossing.
  - Edge following (Canny operator).
- **Global techniques.** Detect particular shapes in the image (circles, lines, etc.).
  - Hough transform.

This notebook will cover the **Hough transform** (section 5.1.1), a contour-based technique that can be used for detecting regions with an arbitrary shape in images.

Problem context - Self-driving car

A prestigious company located at PTA (The Andalusia Technology Park) is organizing a hackathon for this year in order to motivate college students to make further progress in the autonomous cars field. Computer vision students at UMA decided to take part in it, but the organizers have posed an initial basic task to guarantee that participants have expertise in image processing techniques.

This way, the company sent to students a task for **implementing a basic detector of road lane lines using OpenCV in python**. We are lucky! These are two tools that we know well ;).

Detecting lines in a lane is a fundamental task for autonomous vehicles while driving on the road. It is the building block to other path planning and control actions like breaking and steering.

So here we are! We are going to detect road lane lines using Hough transform in OpenCV.

In [1]:
```
import numpy as np
import cv2
import math
import matplotlib.pyplot as plt
import matplotlib
import scipy.stats as stats

matplotlib.rcParams['figure.figsize'] = (10.0, 10.0)
images_path = './images/'
```

5.1.1 Hough transform

The **Hough Transform** is a technique for the detection of arbitrary shapes in an image. For that, such shapes must be expressed in:

- analytical form (**classic Hough**), e.g. using the mathematical representation of lines, circles, ellipses, etc., or
- in a numerical form (**generalized Hough**) where the shape is given by a table.

Since our goal is to detect lines, we will focus here on analytically expressed shapes. Specifically, a line can be represented analytically as:

$$y = ax + b$$

where a is the slope and b is the y-incertcept, what is called the **explicit representation** of a line. However, this representation is problematic when representing vertical lines ($m = \inf$) or when discretizing the possible values for m, as we will see). Other option is its **parametric form**:

$$\rho = x\cos\theta + y\sin\theta$$

where ρ is the perpendicular distance from the origin $(0,0)$ to the line, and θ is the angle formed by this perpendicular line and the horizontal axis measured in counter-clockwise (see Fig. 3). Thereby, we are going to represent lines using the pair of parameters (ρ, θ).

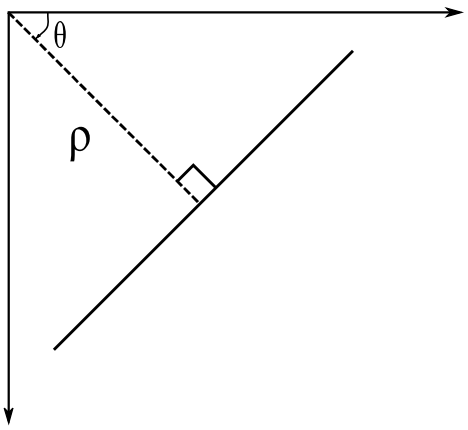


Fig. 3. Parametric representation of a line.

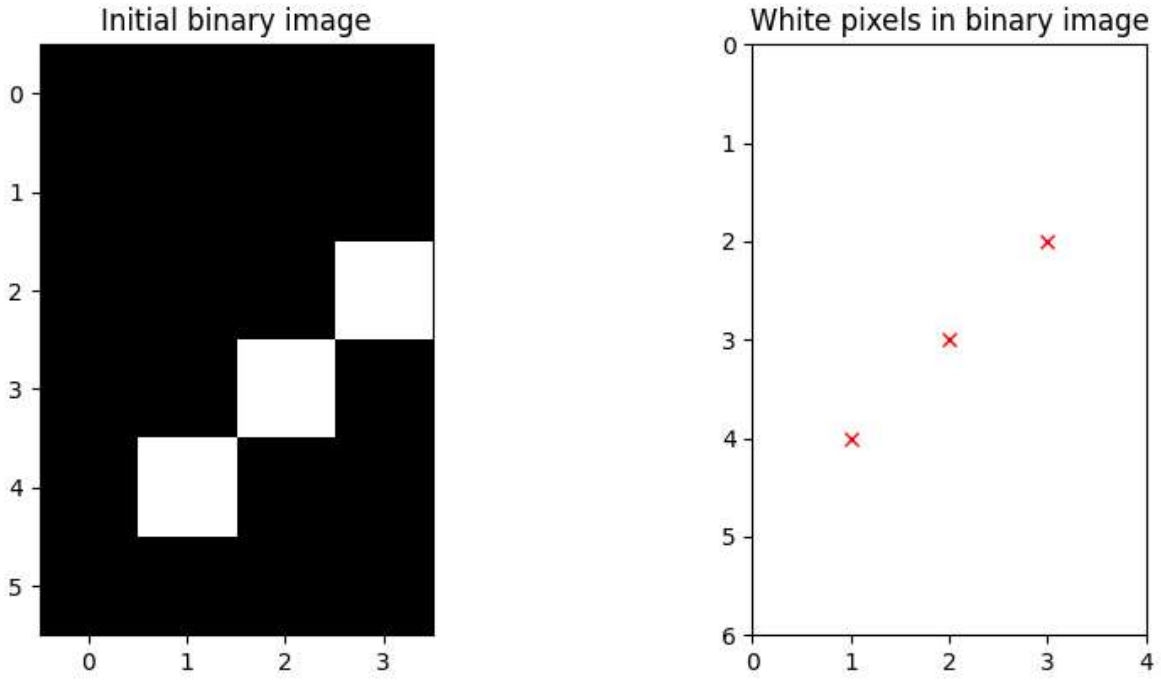
The Hough transform works by a voting procedure, which is carried out in a parameter space (ρ, θ in our case). This technique consists of the following steps:

1. **Build an accumulator matrix**, where rows index the possible values of ρ, and columns those for θ. For example, if the possible values for ρ are $0, 1, 2, \ldots d$ (where d is the max distance e.g. diagonal size of the image) and those for θ are $0, 1, 2 \ldots 179$, the matrix shape would be $(d, 180)$.

In [2]:
```
# Define possible rho and theta values
theta_values = [0,np.pi/4,np.pi/2,3*np.pi/4]
rho_values = [0,1,2,3,4,5]

# Create the accumulator
acc = np.zeros([len(rho_values),len(theta_values)])
```

2. **Binarize the input image** to obtain pixels that are candidates to belong to the shape contours (e.g. by applying an edge detector).

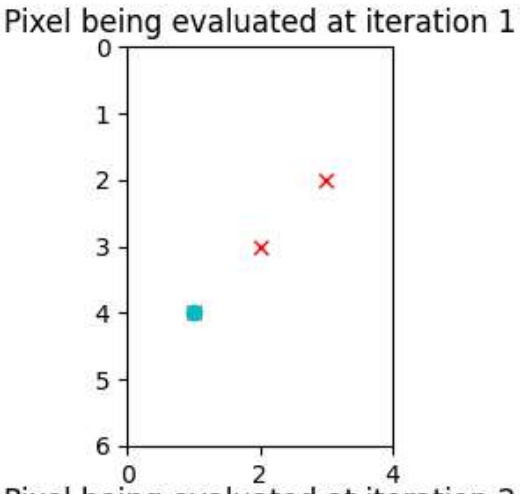
In [3]:
```
# Coordinates of white points in binary image
xs = np.array([1,2,3])
ys = np.array([4,3,2])

# Initial image
plt.subplot(221)
blank_image = np.zeros((6,4,1), np.uint8)
blank_image[ys,xs] = 255
plt.imshow(blank_image, cmap='gray');
plt.title('Initial binary image')

# Show them!
plt.subplot(222)
plt.plot(xs,ys,'rx')
plt.axis('scaled')
axes = plt.gca()
axes.set_xlim([0,4])
axes.set_ylim([0,6])
plt.gca().invert_yaxis()
plt.title('White pixels in binary image');
```

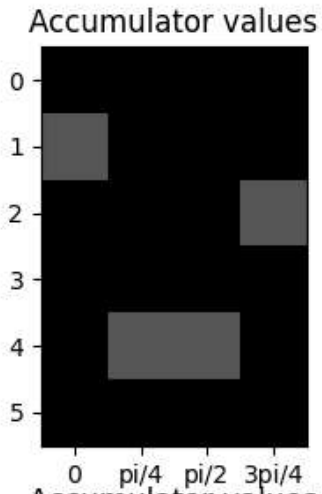


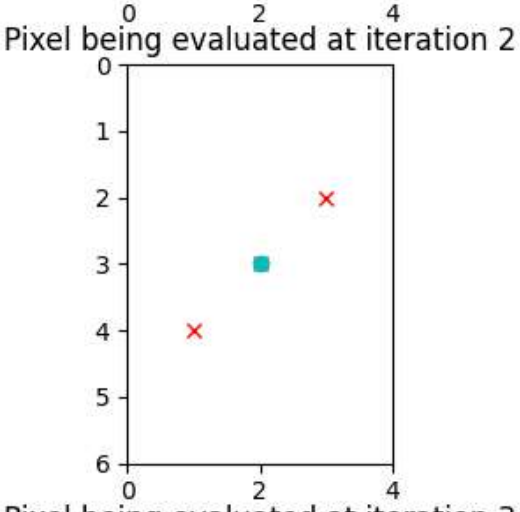
3. For each candidate (white pixel):
   A. **Evaluate**: Since the point coordinates (x, y) are known, place them in the line parametric form and iterate over the possible values of θ to obtain the values for ρ. In the previous example $\rho_i = x\cos\theta_i + y\sin\theta_i, \forall i \in [0, 180]$
   B. **Vote**: For every obtained pair (ρ_i, θ_i) increment by one the value of its associated cell in the accumulator.

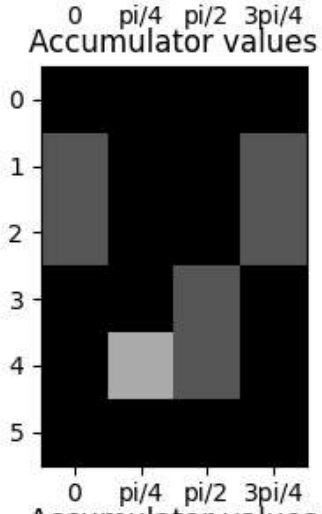
If we do this with an accumulator with an enough resolution, we would get a sinusoid.

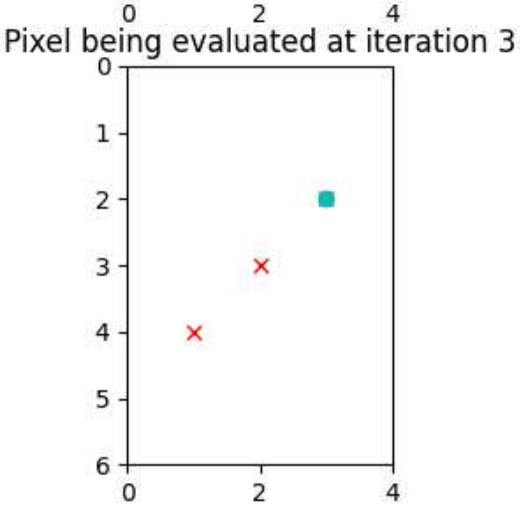
In [4]:
```
# For each white pixel

for i in range (0,len(xs)):
    x = xs[i]
    y = ys[i]

    # Show the point voting
    subplot_index = str(32) + str(i*2+1)
    plt.subplot(int(subplot_index))
    plt.axis('scaled')
    axes = plt.gca()
    axes.set_xlim([0,4])
    axes.set_ylim([0,6])

    plt.plot(xs,ys,'rx')
    plt.plot(x,y,'co')

    plt.title('Pixel being evaluated at iteration ' + str(i+1))
    plt.gca().invert_yaxis()


    # Ecaluate the (x,y) coordinates for different values of theta, and
    # retrieve rho
    for theta_index in range(0,len(theta_values)):
        theta = theta_values[theta_index]
        rho = x*np.cos(theta) + y*np.sin(theta)
        rho = int(np.round(rho))
        # Vote!
        acc[rho][theta_index] += 1.0

    # Show the acccumulator
    subplot_index = str(32) + str(i*2+2)
    plt.subplot(int(subplot_index))
    plt.imshow(acc,cmap='gray',vmax=3);
    plt.xticks([0, 1, 2, 3], ['0','pi/4','pi/2','3pi/4'])

    plt.title('Accumulator values')
```

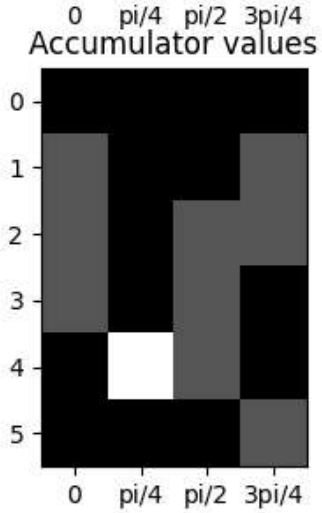












4. Finally, **obtain the shape candidates** by setting a threshold to control how many votes needs a pair (ρ, θ) to be considered a line, and by applying local maxima in the accumulator space.

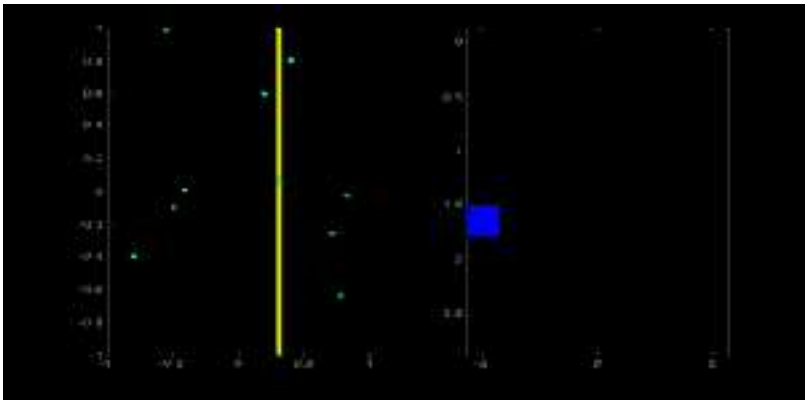


Fig. 3. Left, image space. Right, parameter space illustrating the evolution of the votes. Note that in this example θ have only 8 possible values.

The idea behind this algorithm is that when a pixel in the image space votes for all the lines that go through it in the parameter space, when a second pixel belonging to the same line votes, then the line connecting both pixels would have two votes.

OpenCV pill

OpenCV implements the method `cv2.HoughLines()` for detecting lines using the Hough transform. However, prior to its usage, and as commented in the `step 1.` of the algorithm, it is needed a binary image. For that we are going to resort to our old friend the Canny algorithm, so the detected edges will be the white pixes in the binary image.

As we now, noisy images seriously hamper the performance of computer vision techniques, and since cv2.Canny() does not include blurring, we provide here a method called `gaussian_smoothing()` to assist you in that task.

In [5]:
```
def gaussian_smoothing(image, sigma, w_kernel):
    """ Blur and normalize input image.

        Args:
            image: Input image to be binarized
            sigma: Standard deviation of the Gaussian distribution
            w_kernel: Kernel aperture size

        Returns:
            binarized: Blurred image
    """

    # Define 1D kernel
    s=sigma
    w=w_kernel
    kernel_1D = [np.exp(-z*z/(2*s*s))/np.sqrt(2*np.pi*s*s) for z in range(-w,w+1

    # Apply distributive property of convolution
    kernel_2D = np.outer(kernel_1D,kernel_1D)

    # Blur image
    smoothed_img = cv2.filter2D(image,cv2.CV_8U,kernel_2D)

    # Normalize to [0 254] values
    smoothed_norm = np.array(image.shape)
```

```
    smoothed_norm = cv2.normalize(smoothed_img,None, 0, 255, cv2.NORM_MINMAX)

    return smoothed_norm
```

ASSIGNMENT 1: Detecting lines with Hough

Your first task is to apply `cv2.HoughLines()` to the image `car.png`, a test image taken from the frontal camera of a car. Draw the resultant lines using `cv2.line()`.

The main inputs of `cv2.HoughLines()` are:

- *image*: binary input image
- *rho*: distance resolution of the accumulator in pixels (usually 1, it may be bigger for high resolution images)
- *theta*: angle resolution of the accumulator in radians. (usually $\frac{\pi}{180}$)
- *threshold*: only line candidates having a number of votes $>$ threshold are returned.

And it returns:

- a $(n\_lines x 1 x 2)$ array containing, in each row, the parameters of each detected line in the $[\rho, \theta]$ format.

Note that, for drawing the lines, you have to transform the resultant lines from the (ρ, θ) space to Cartesian coordinates.

Try different parameter values until you get something like this:

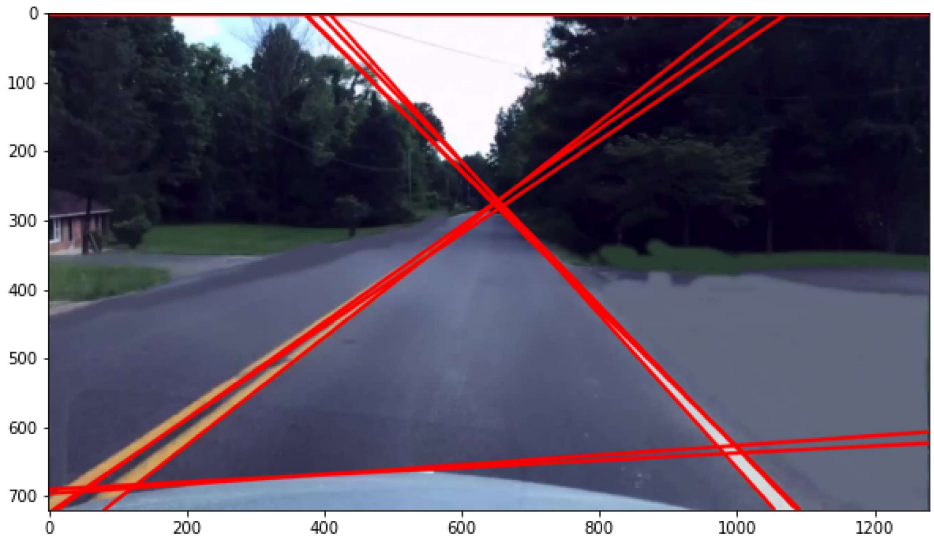


Fig. 4. Example of lines detection.

In [6]:
```
# Assignment 1
# Read the image
image = cv2.imread(images_path + "car.png",-1)

# Convert to RGB and get gray image
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

# Blur the gray image
```

```
gray = gaussian_smoothing(gray,2,5)

# Apply Canny algorithm
edges = cv2.Canny(gray,100,200,apertureSize = 3)

# Search for lines using Hough transform
lines = cv2.HoughLines(edges, rho=1, theta=np.pi/180, threshold=150)

# For each line
for i in range(0, len(lines)):

    # Transform from polar coordinates to cartesian coordinates
    rho = lines[i][0][0]
    theta = lines[i][0][1]

    a = math.cos(theta)
    b = math.sin(theta)

    x0 = rho * a
    y0 = rho * b

    # Get two points in that line
    x1 = int(x0 + 2000*(-b));
    y1 = int(y0 + 2000*(a))
    pt1 = (x1,y1)
    x2 = int(x0 - 2000*(-b))
    y2 = int(y0 - 2000*(a))
    pt2 = (x2,y2)

    # Draw the line in the RGB image
    cv2.line(image, pt1, pt2, (255,0,0), 3, cv2.LINE_AA)

# Show resultant image
plt.imshow(image);
```

5.1.2.1 Probabilistic Hough transform

For high-resolution images and large accumulator sizes the Hough transform may need long execution times. However, in applications like autonomous cars a fast execution is mandatory. For example, having a car moving at 100km/h covers ~ 28 meters in a second. Imagine how much lines can change in that time!

OpenCV pill

OpenCV provides with the method `cv2.HoughLinesP()` a more complex implementation of the Hough Line Transform, which is called **probabilistic Hough Transform**. This alternative does not take all the points in the binary image into account, but a random subset of them that are still enough for line detection. This also results in lower thresholds when deciding if a line exists or not.

ASSIGNMENT 2: Another option for detecting lines

Apply `cv2.HoughLinesP()` to the image `car.png` and draw the detected lines.

This function returns:

- line segments `[x1,y1,x2,y2]` instead of the line equation parameters.

For that, two additional arguments are needed:

- minLineLength: line segments shorter than that are rejected.
- maxLineGap: maximum allowed gap between points on the same line to link them.

Try different parameter values until you get something like this:

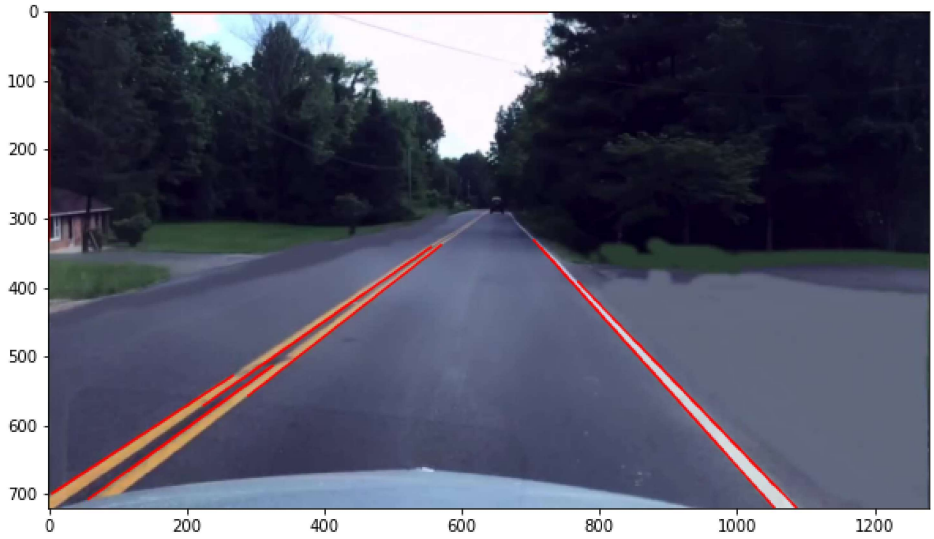
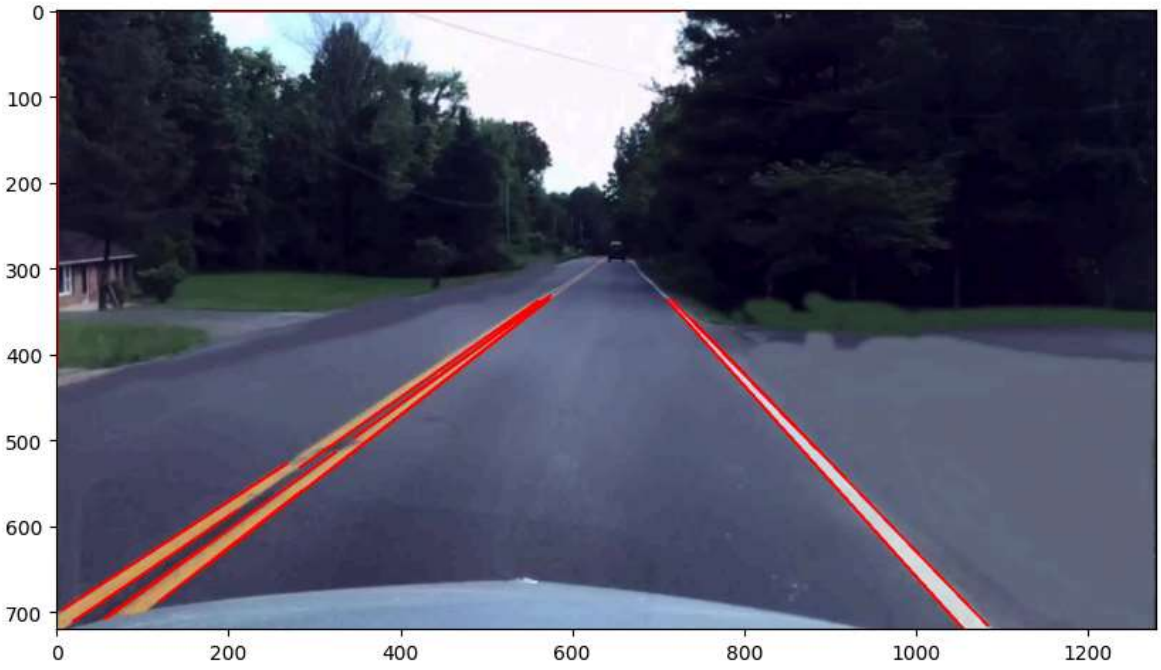


Fig. 5. Lines detection example with the probabilistic Hough Transform.

In [7]:
```
# Assigment 2
# Read the image
image = cv2.imread(images_path + "car.png",-1)

# Convert to RGB and get gray image
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
```

```
# Blur the gray image
gray = gaussian_smoothing(gray,1,2)

# Apply Canny algorithm
edges = cv2.Canny(gray,100,200,apertureSize = 3)

# Search for lines using probabilistic Hough transform
rho = 1
theta = np.pi/180
threshold = 155
lines = cv2.HoughLinesP(edges, rho, theta, threshold,
                        minLineLength=100,maxLineGap=19)
# For each line
for line in lines:

    # Draw the line in the RGB image
    x1,y1,x2,y2 = line[0]
    cv2.line(image,(x1,y1),(x2,y2),(255,0,0),2)

# Show resultant image
plt.imshow(image);
```



Thinking about it (1)

Now that you have played with the Hough Transform, **answer the following questions**:

- In the first assignment, we obtained an image with a number of red lines drawn on it. However, these lines goes over pixels in the image that do not contains lines! (*e.g.* belonging to the sky). What could we do to fix this, that is, obtain the pixels belonging to lines in the image?

  Se podría evitar que las rectas continuarán más allá de la verdadera recta de la imagen haciendo la intersección entra la línea detectada por Hough y los píxeles que votaron esta recta. De esa forma, la recta solo se extendería hasta donde se extienda la recta en el gradiente.

- Without restrictions regarding execution time, which method would you use, Hough Transform or its probabilistic counterpart?

  Ya que no contamos con restricciones de coste computaiconal, elegiría la versión estándar, ya que siempre será mejor el algoritmo original que su aproximación

- Could there be a cell in the accumulator with a value higher than the number of white pixels in the binary image? why?

  No, ya que solo los píxeles blancos de la imagen binarizada son los que pueden votar una recta, y solo se puede realizar un voto por píxel. En una imagen de N píxeles blancos habrá una celda de, a lo sumo, valor N.

- Which should be the maximum value for ρ in the accumulator? Why?

  La distancia diagonal de la imagen para que todos los píxeles de la imagen puedan ser contenidos en la matriz.

OPTIONAL

Think about any other application where the finding of straight lines could be useful. Get some images related to said application and detect lines with the Hough transform!

END OF OPTIONAL PART

Conclusion

Terrific work! In the road lane lines detection context you have learned:

- to detect shapes in images using Hough transform .

Also, you obtained some knowledge about:

- self-driving cars and computer vision, and
- lane line detection for autonomous cars.

See you in the next one! Keep learning!

5.2 Clustering-based techniques for image segmentation

In the previous notebook we had fun with contour based techniques for image segmentation. In this one we will play with region-based techniques, where the resulting segments cover the entire image. Concretely we will address two popular region-based methods:

- K-means (section 5.2.1)
- Expectation-Maximization (EM, section 5.2.2)

Problem context - Color quantization

Color quantization is the process of reducing the number of distinct colors in an image while preserving its color appearance as much as possible. It has many applications, like image compression (e.g. GIFs, which only support 256 colors!), content-based image retrieval, edge detection and segmentation preprocessing, printing, medical imaging, etc.

Image segmentation techniques can be used to achieve color quantization, let's see how it works!

In [1]:
```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import matplotlib
import scipy.stats as stats
from ipywidgets import interact, fixed, widgets
matplotlib.rcParams['figure.figsize'] = (10.0, 10.0)
images_path = './images/'

import sys
sys.path.append("..")
from utils.PlotEllipse import PlotEllipse
```

5.2.1 K-Means

As commented, region-based techniques try to group together pixels that are similar. Such issue is often called the *clustering problem*. Different attributes can be used to decide if two pixels are similar or not: intensity, texture, color, pixel location, etc.

The **k-means algorithm** is a region-based technique that, given a set of elements (image pixels in our case), makes K clusters out of them. Thereby, it is a perfect technique for addressing color quantization, since our goal is to reduce the color palette of an image to a fixed number of colors K. Concretely, k-means aims to minimize the sum of squared Euclidean distances between points x_i in a given space (*e.g.* grayscale or RGB color representations) and their nearest cluster centers m_k:

$$\arg\min_{M} D(X, M) = \Sigma_{\text{Cluster } k} \Sigma_{\text{point } i \text{ in cluster } k} (x_i - m_k)^2$$

In our case, the point x_i could be interpreted as a **feature vector** describing the i^{th} pixel that, as mentioned, could include information like the pixel color, intensity, texture, etc. Thus, m_k represents **the mean of the feature vector** of the pixels in cluster k.

Let's see how the k-means algorithm works in a color domain, where each pixel is represented in a feature n-dimensional space (*e.g.* grayscale images define a 1D feature space, while RGB images a 3D space):

1. Pick the number K, that is, the number of clusters in which the image will be segmented (e.g. number of colors).
2. Place K centroids m_k in the color space (e.g. randomly), these are the centers of the regions.
3. Each pixel is assigned to the cluster with the closest centroid, hence creating new clusters.

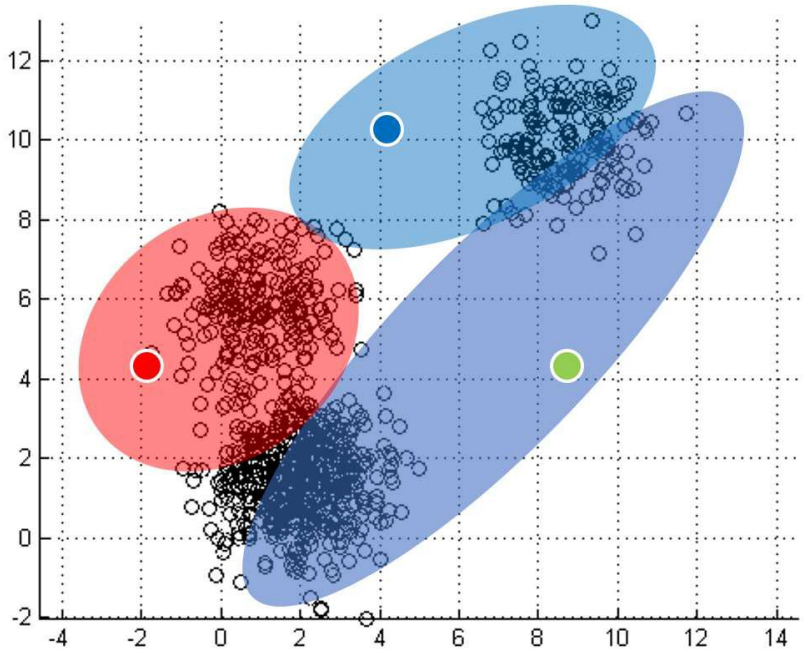


Fig 1. Example in a 2D space (e.g. YCbCr color space) with 3 clusters. Each point is assigned to its closest centroid

4. Compute the new means m_k of the K clusters.

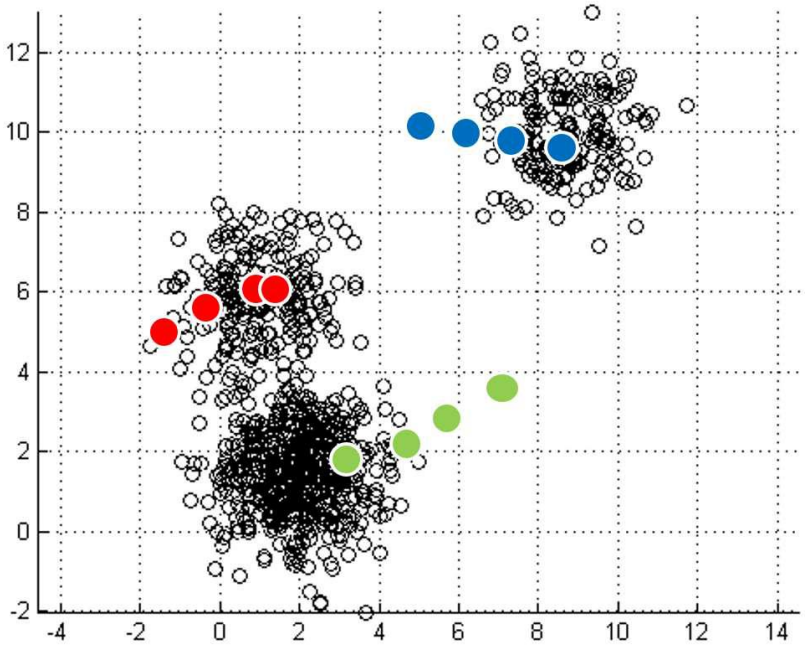


Fig 2. Example of how the centroids evolve over time

5. Repeat steps 3 and 4 until convergence, that is, some previously defined criteria is fulfilled (*e.g.* the centers of regions do not move, or a certain number of iterations is reached).

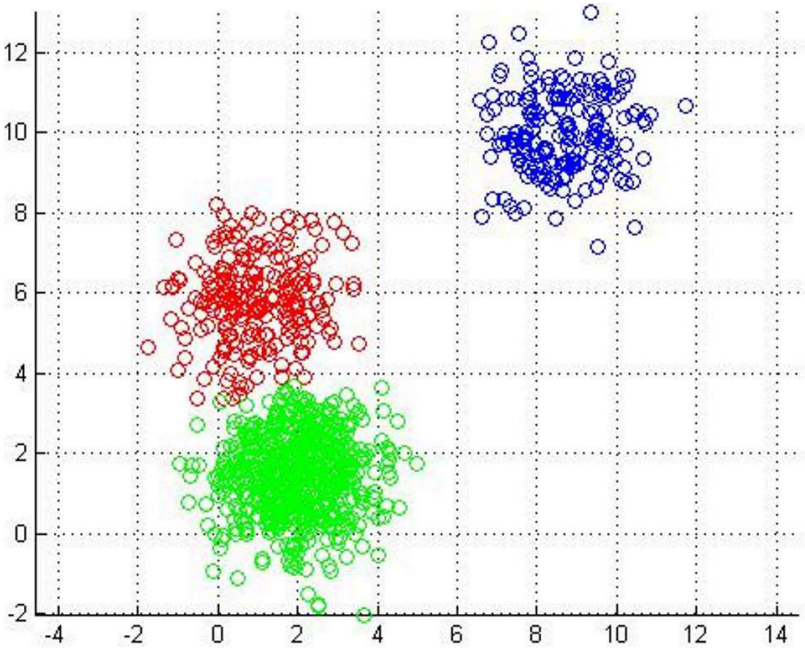


Fig 3. Final segmentation result

This procedure is the same independently of the number of dimensions in the workspace.

This technique presents a number of pros and cons:

- **Pros:**
  - It's simple.

  - Convergence to a local minima is guaranteed (but no guarantee to reach the global minima).
- **Cons:**
  - High usage of memory.
  - The K must be fixed.
  - Sensible to the selection of the initialization (initial position of centroids).
  - Sensible to outliers.
  - Circular clusters in the feature space are assumed (because of the usage of the Euclidean distance)

K-means toy example

Luckily for us, OpenCV defines a method that perform k-means: `cv2.kmeans()`, here you can find a nice explanation about how to use it. Let's take a look at a toy 1D k-means example in order to get familiar with it. The following function, `binarize_kmeans()`, binarizes an input `image` by executing the K-means algorithm, where the `it` sets its maximum number of iterations.

Note that the stopping criteria can be either:

- if a maximum number of iterations is reached, or
- if the centroid moved less than a certain `epsilon` value in an iteration.

In [2]:
```
def binarize_kmeans(image,it):
    """ Binarize an image using k-means.

        Args:
            image: Input image
            it: K-means iteration
    """

    # Set random seed for centroids
    cv2.setRNGSeed(124)

    # Flatten image
    flattened_img = image.reshape((-1,1))
    flattened_img = np.float32(flattened_img)

    #Set epsilon
    epsilon = 0.2

    # Estabish stopping criteria (either `it` iterations or moving less than `ep
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, it, epsilon)

    # Set K parameter (2 for thresholding)
    K = 2

    # Call kmeans using random initial position for centroids
    _,label,center=cv2.kmeans(flattened_img,K,None,criteria,it,cv2.KMEANS_RANDOM

    # Colour resultant labels
    center = np.uint8(center) # Get center coordinates as unsigned integers
    print(center)
    flattened_img = center[label.flatten()] # Get the color (center) assigned to
```

```
    # Reshape vector image to original shape
    binarized = flattened_img.reshape((image.shape))

    # Show resultant image
    plt.subplot(2,1,1)
    plt.title("Original image")
    plt.imshow(binarized, cmap='gray',vmin=0,vmax=255)

    # Show how original histogram have been segmented
    plt.subplot(2,1,2)
    plt.title("Segmented histogram")
    plt.hist([image[binarized==center[0]].ravel(), image[binarized==center[1]].r
```

As you can see, `cv2.kmeans()` returns two relevant arguments:

- label: Integer array that stores the cluster index for every pixel.
- center: Matrix containing the cluster centroids (each row represents a different centroid).

Attention to this!!! It is also remarkable the first function argument, which represents the data for clustering: an array of N-Dimensional points with float coordinates. Such array has the shape $num\_samples \times num\_features$, i.e., it has as many rows as samples (pixels in the image), and as many columns as features describing those samples (for example, if using the intensity of a pixel in a graysacle image, there is only one feature). For that, the code line `image.reshape((-1,1))` convert the initial grayscale image with dimensions 242×1133 into a flattened version of it with dimension 274186×1, that is, 274186 samples (or pixels) with only one feature, its intensity. Take a look at `np.reshape()` to see how it works.

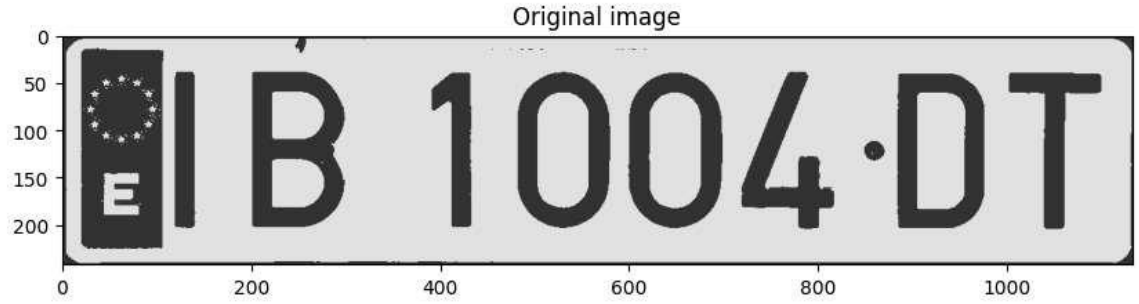
Below it is provided an interactive code so you can play with `cv2.kmeans()` by calling it with different `it` values.

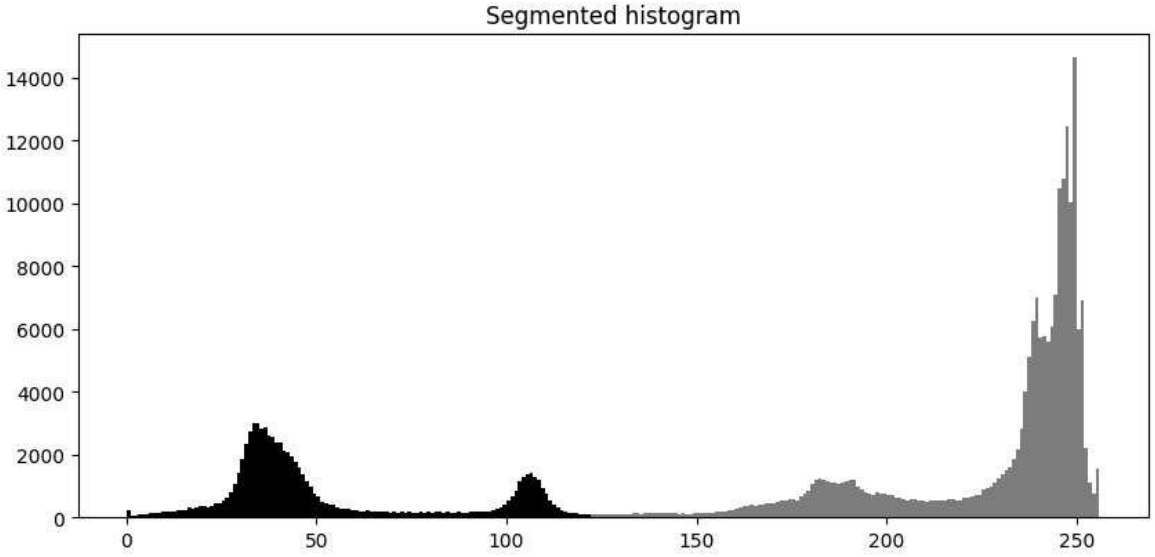
As you can see, if k=2 in a grayscale image, it is a binarization method that doesn't need to fix a manual threshold. We could have used it, for example, when dealing with the plate recognition problem!

In [3]:
```
matplotlib.rcParams['figure.figsize'] = (10.0, 10.0)

image = cv2.imread(images_path + 'plate.jpg',0)

interact(binarize_kmeans, image=fixed(image),it=(2,5,1));
```

it 3

```
[[ 52]
 [229]]
```





Notice that for 1D spaces and not high-resolution images k-means is very fast! (it only needs a few iterations to converge). What happens if k-means is applied to color images (3D space) in order to get color quantization?

Now that you know how k-means works, you can experimentally answer such question!

ASSIGNMENT 1: Playing with K-means

Write an script that:

- applies k-means to `malaga.png` with different values for K: $K = 4$, $K = 8$ and $K = 16$, setting `epsilon=0.2` and `it=10` as convergence criteria, and
- shows, in a 2×2 subplot, the 3 resulting images along with the input one.

Notice that in this case we are using 3 features per pixel, their R, G and B values, so the input data for the kmeans function has the dimensions $num\_pixels \times 3$.

Expected output:

In [4]:
```
# Assignment 1
matplotlib.rcParams['figure.figsize'] = (15.0, 12.0)

# Read RGB image
image = cv2.imread(images_path + "malaga.png")
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Flatten image
flattened_img = image.reshape((-1,3))
flattened_img = np.float32(flattened_img)

# Set criteria
it = 10
epsilon = 0.2
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, it, epsilon)

# Apply k-means. Keep the third argument as None!
_,label4,center4=cv2.kmeans(flattened_img,4,None,criteria,30,cv2.KMEANS_RANDOM_C
_,label8,center8=cv2.kmeans(flattened_img,8,None,criteria,30,cv2.KMEANS_RANDOM_C
_,label16,center16=cv2.kmeans(flattened_img,16,None,criteria,30,cv2.KMEANS_RANDC

# Colour resultant labels
center4 = np.uint8(center4)
center8 = np.uint8(center8)
center16 = np.uint8(center16)

# Get the color (center) assigned to each pixel
res4 = center4[label4.flatten()]
res8 = center8[label8.flatten()]
res16 = center16[label16.flatten()]

# Reshape to original shape
quantized4 = res4.reshape((image.shape))
quantized8 = res8.reshape((image.shape))
```

```
quantized16 = res16.reshape((image.shape))

# Show original image
plt.subplot(2,2,1)
plt.title("Original image")
plt.imshow(image)

# Show k=4
plt.subplot(2,2,2)
plt.title("k=4")
plt.imshow(quantized4)

# Show k=8
plt.subplot(2,2,3)
plt.title("k=8")
plt.imshow(quantized8)

# Show k=16
plt.subplot(2,2,4)
plt.title("k=16")
plt.imshow(quantized16);
```

Thinking about it (1)

Now, **answer the following questions**:

- What `cv2.kmeans()` is doing in each iteration?

  En cada iteración, cadapixel se asigna al cluster con el centroide más cercano a él. Una vez terminado este proceso, se recomputa el centroide y se vuelvena a asignar los píxeles a cada clúster hasta que converja,

- What number of maximum iterations did you use? Why?

  10, ya que son suficientes para llegar a la convergencia sin pasarse de iteraciones

- How could we compress these images so they require less space in memory? *Note: consider that a pixel in RGB needs 3 bytes to be represented, 8 bits per band.*

  Al reducir los colores, se podría codificar a que región pertenece cada pixel con menos bits. Por ejemplo, usando 2 bits por banda en el caso de k = 4, 3 con k = 8, y 4 con k = 16.

Analyzing execution times

In this exercise you are asked to compare the execution time of K-means in a grayscale image, with K-means in a RGB image. Use the image `malaga.png` for this task, and use the same number of clusters and criteria for both, the grayscale and the RGB images.

Tip: how to measure execution time in Python

In [5]:
```
import time

print("Measuring the execution time needed for ...")

K = 2

# Read images
image = cv2.imread(images_path + "malaga.png")
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

# Set criteria
it = 10
epsilon = 0.2
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, it, epsilon)

start = time.process_time() # Start timer

# Flatten image
flattened_img = image.reshape((-1,3))
flattened_img = np.float32(flattened_img)

# Apply k-means
_,label,center=cv2.kmeans(flattened_img,K,None,criteria,30,cv2.KMEANS_RANDOM_CEN

print("K-means in the RGB image:", round(time.process_time() - start,5), "second

start = time.process_time() # Start timer

# Flatten image
flattened_img = gray.reshape((-1,1))
flattened_img = np.float32(flattened_img)

# Apply k-means
_,label,center=cv2.kmeans(flattened_img,K,None,criteria,30,cv2.KMEANS_RANDOM_CEN

print("K-means in the grayscale image:", round(time.process_time() - start,5), "
```