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.

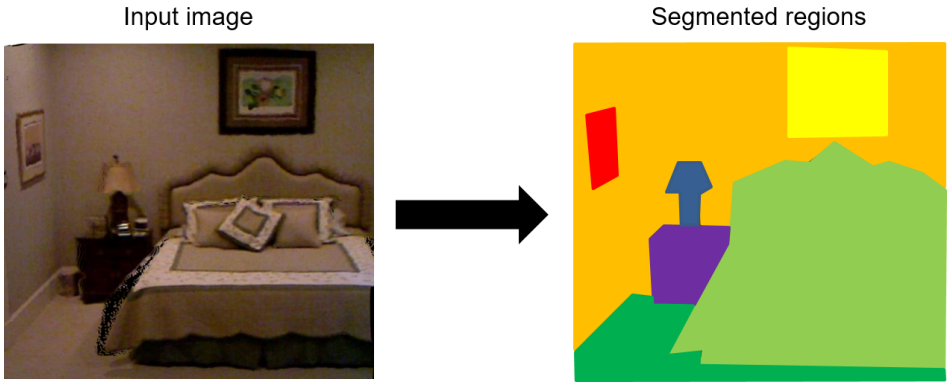


Fig. 1: Example of instance segmentation where each region corresponds to an object in the scene.

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).

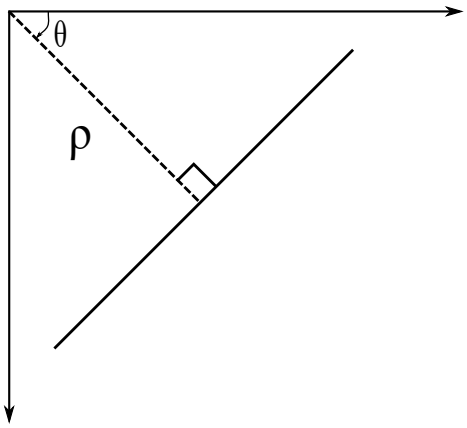


Fig. 3. Parametric representation of a line.

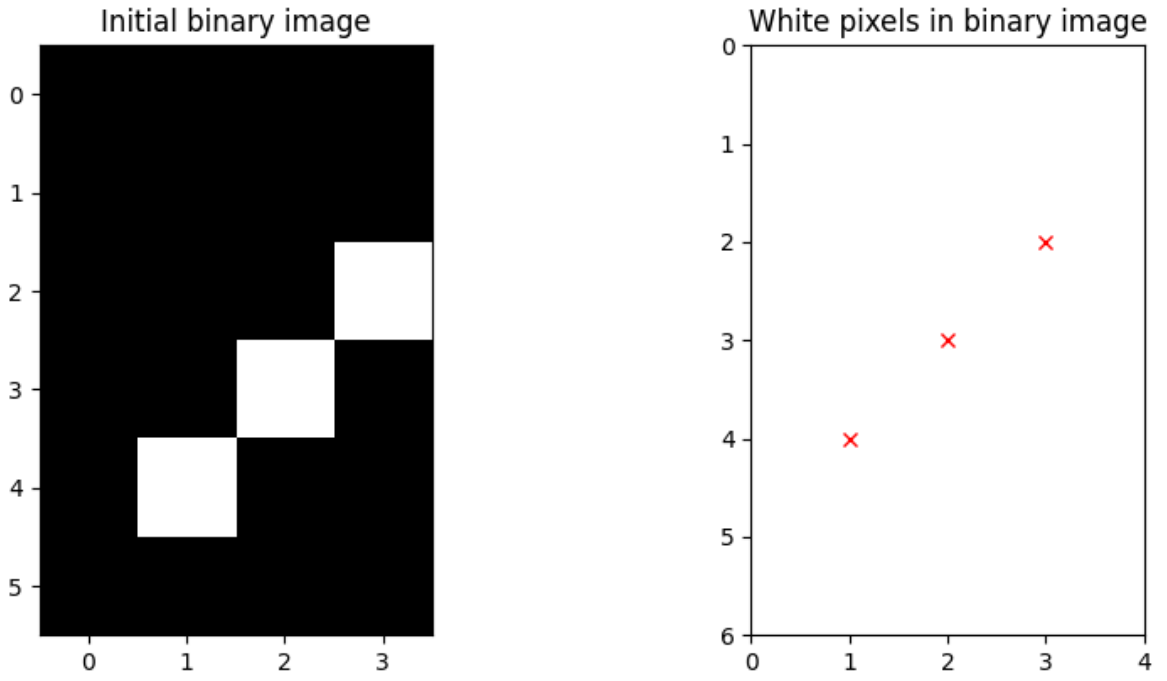
The Hough transform works by a voting procedure, which is carried out in a parameter space (ρ, θ in our case). This technique consists of the following steps:

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).

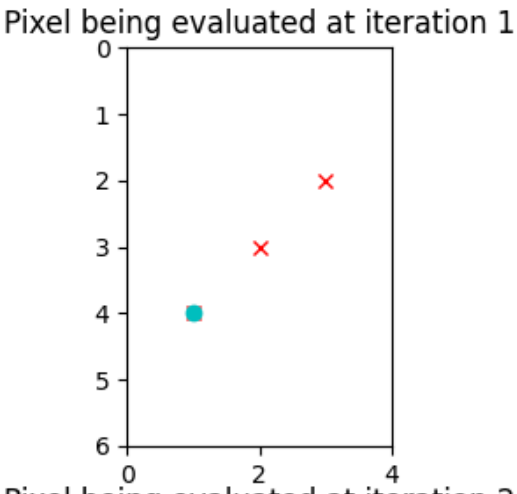
In [3]:
```
# Coordinates of white points in binary image
xs = np.array([1,2,3])
ys = np.array([4,3,2])

# 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');
```

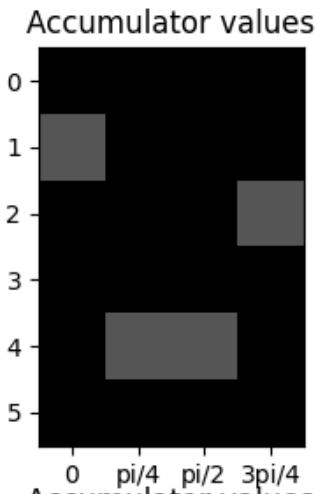


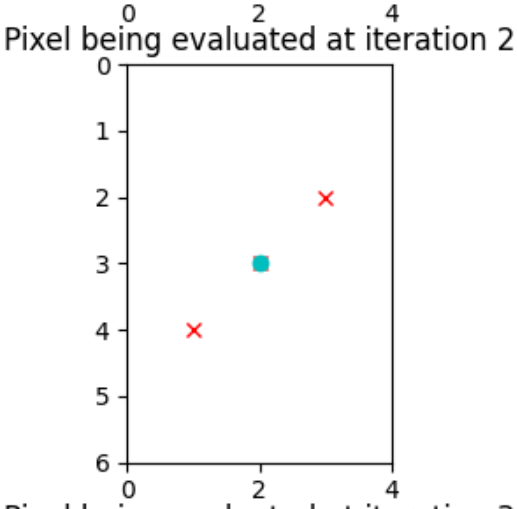
3. For each candidate (white pixel):
   A. **Evaluate**: Since the point coordinates (x, y) are known, place them in the line parametric form and iterate over the possible values of θ to obtain the values for ρ. In the previous example $\rho_i = x \cos\theta_i + y \sin\theta_i, \forall i \in [0, 180]$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

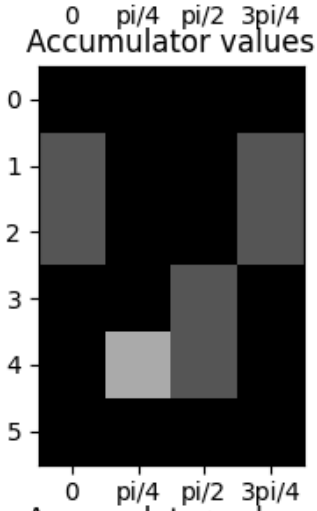
If we do this with an accumulator with an enough resolution, we would get a sinusoid.

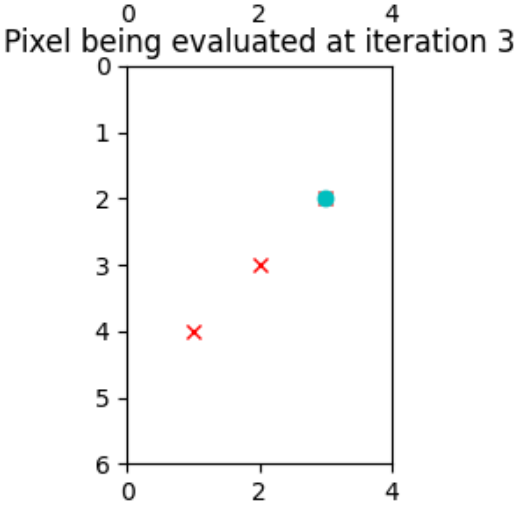
In [4]:
```
# For each white pixel

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')
```

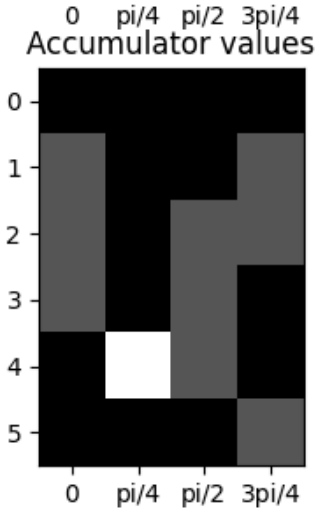












4. Finally, **obtain the shape candidates** by setting a threshold to control how many votes needs a pair (ρ, θ) to be considered a line, and by applying local maxima in the accumulator space.

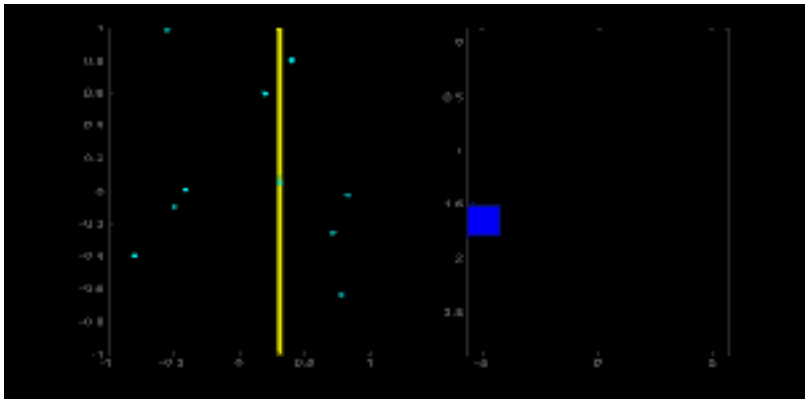


Fig. 3. Left, image space. Right, parameter space illustrating the evolution of the votes. Note that in this example θ have only 8 possible values.

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:

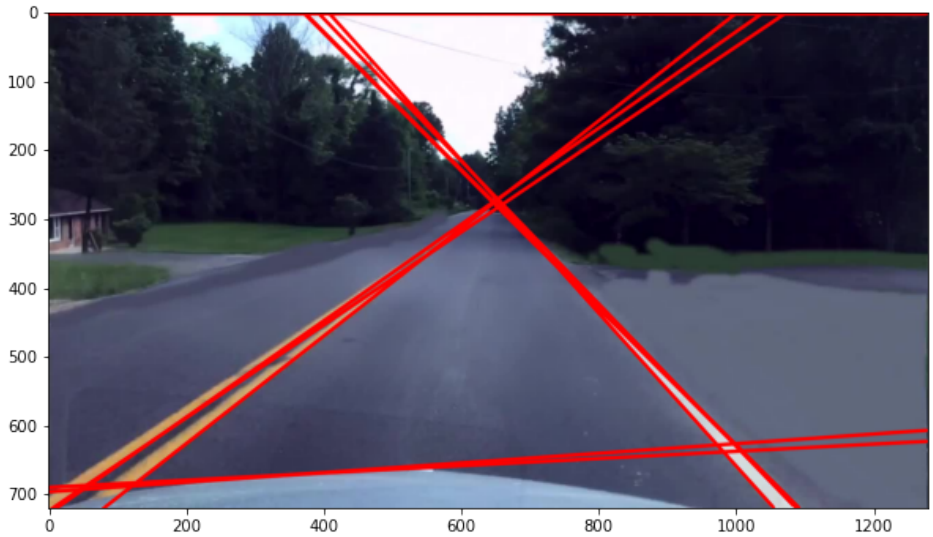
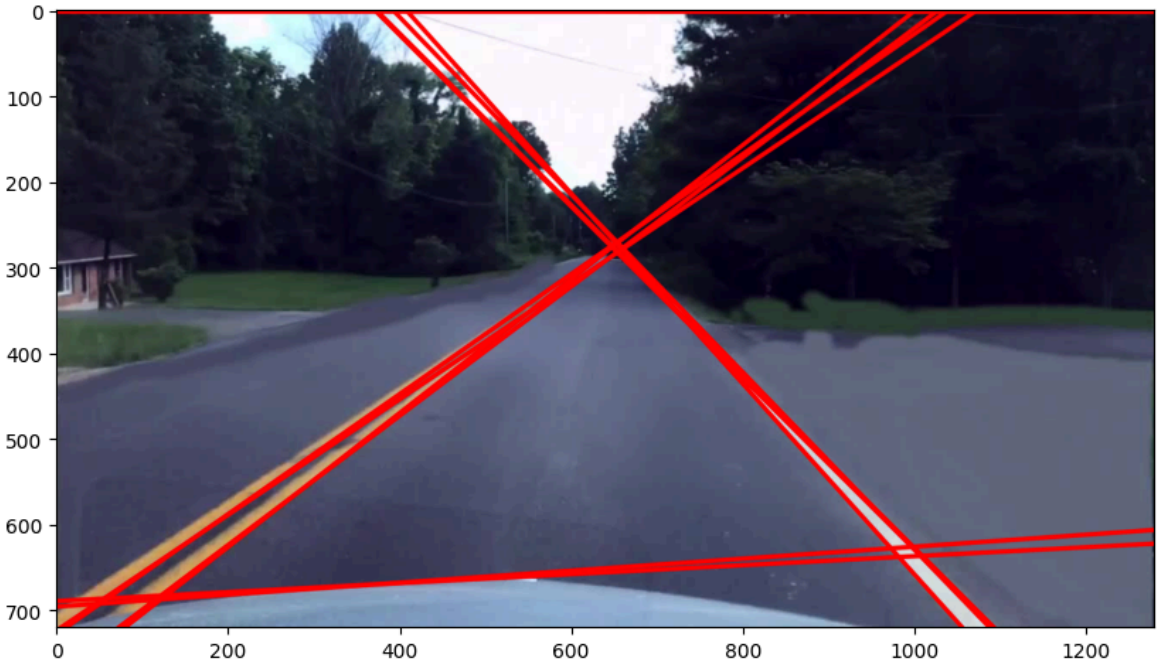


Fig. 4. Example of lines detection.

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=140)

# 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 = a * rho
    y0 = b * rho

    # 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:

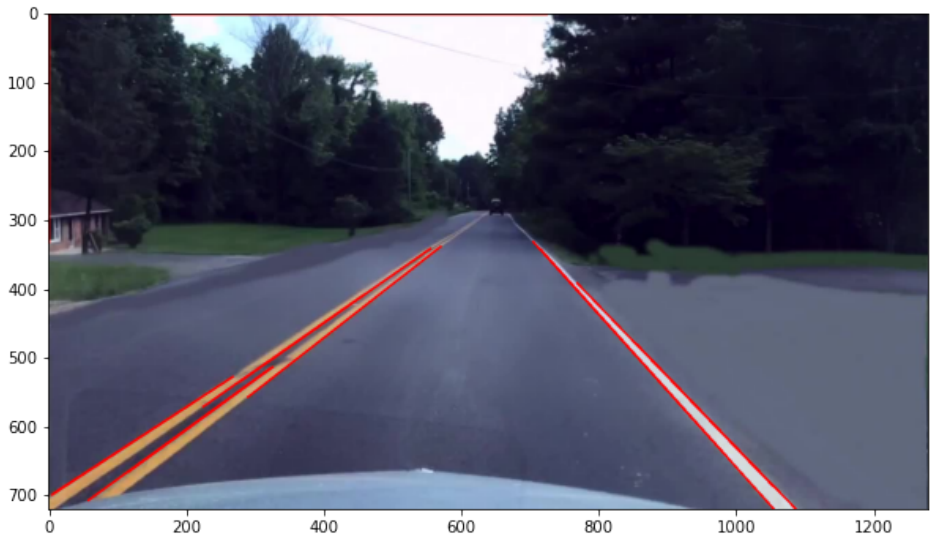


Fig. 5. Lines detection example with the probabilistic Hough Transform.

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 = 140
lines = cv2.HoughLinesP(edges, rho, theta, threshold,
                        minLineLength=230,maxLineGap=15)
# 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?

  To fix this issue, we could maybe use a cropped version of the image focusing on where the lines are as a parameter, instead of the whole original image, or maybe intersect the Canny edges image (pixels that voted for a line) with the Hough lines image.

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

  I would use the Hough Transform, as the resulting lines are well defined.
- Could there be a cell in the accumulator with a value higher than the number of white pixels in the binary image? why?

  No, there could not, as each cell represents the number of votes a pair [rho, theta] has from all the white pixels in the binary image, hence the maximum value a cell could have equals the number of white pixels.
- Which should be the maximum value for ρ in the accumulator? Why?

  The maximum value should equal the diagonal of the image, so that you do not surpass the maximum length a line can have in the image.

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.

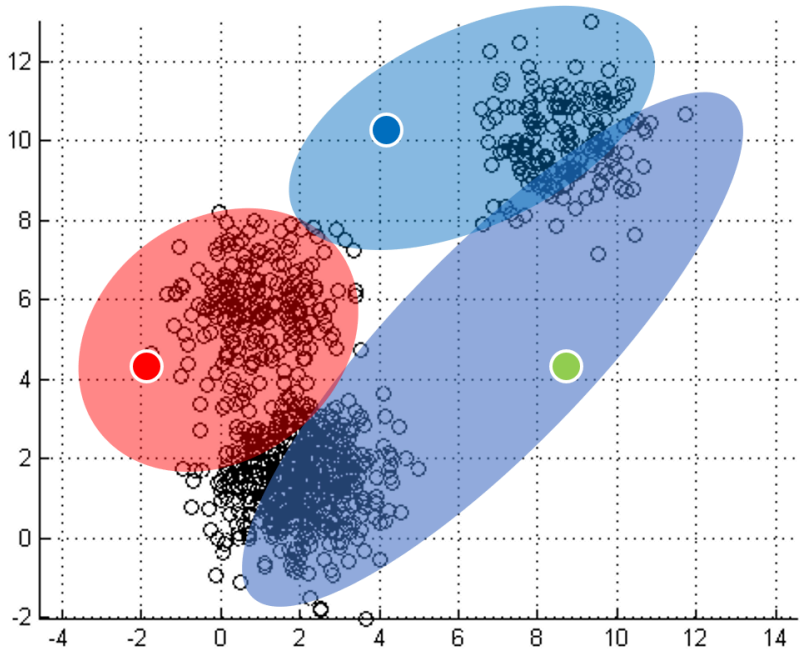


Fig 1. Example in a 2D space (e.g. YCbCr color space) with 3 clusters. Each point is assigned to its closest centroid

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

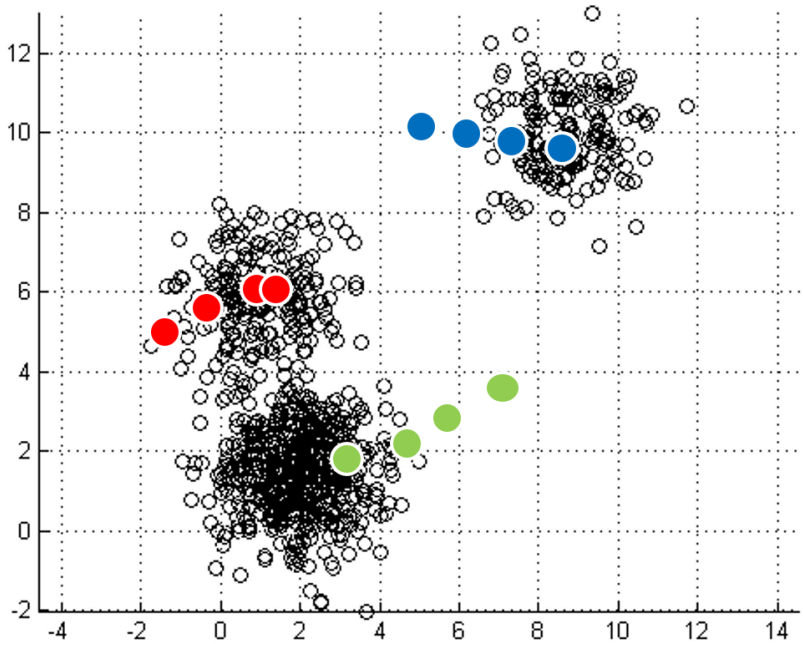


Fig 2. Example of how the centroids evolve over time

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).

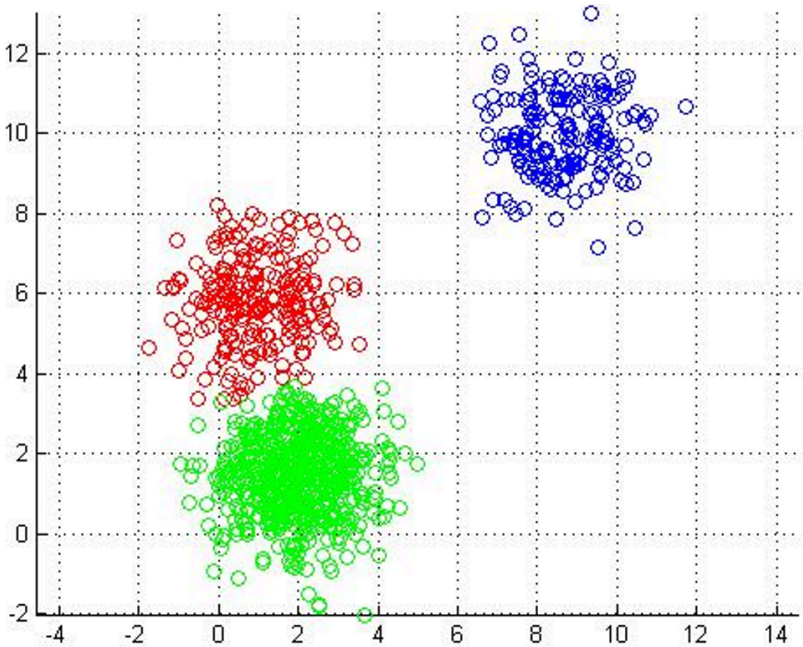


Fig 3. Final segmentation result

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.

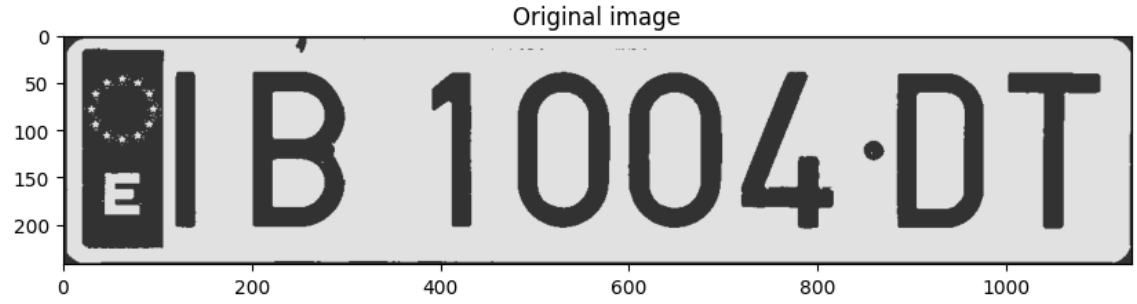
Below it is provided an interactive code so you can play with `cv2.kmeans()` by calling it with different `it` values.

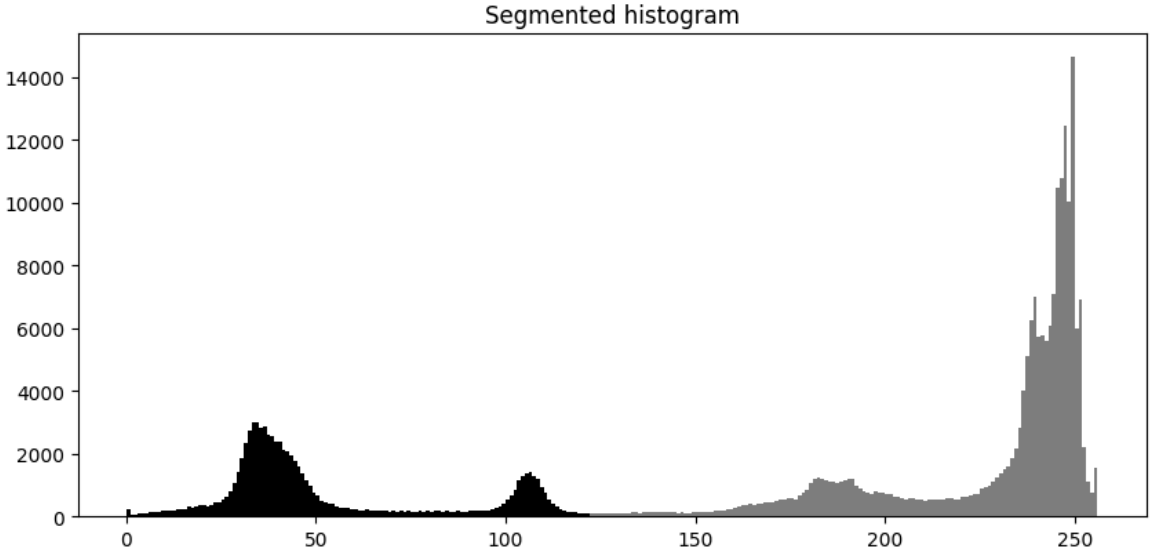
As you can see, if k=2 in a grayscale image, it is a binarization method that doesn't need to fix a manual threshold. We could have used it, for example, when dealing with the plate recognition problem!

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_RANDO

# 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?

  In each iteration, each pixel is assigned to the cluster which has the centroid with the closest distance to it. Then the new centroids (means) of each cluster are calculated.

The last iteration will be when either the maximum number of iterations is reached or the centroids move less than a given epsilon.

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

  I chose 10, so that enough iterations are performed for the centroids to be well placed thus stopping the algorithm because they barely move (or move less than the given epsilon).

- 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.*

  We could use enough bits to cover the number of clusters, that is, using a bit for every 2 clusters.

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), "
```

```
Measuring the execution time needed for ...
K-means in the RGB image: 0.3125 seconds
K-means in the grayscale image: 0.42188 seconds
```

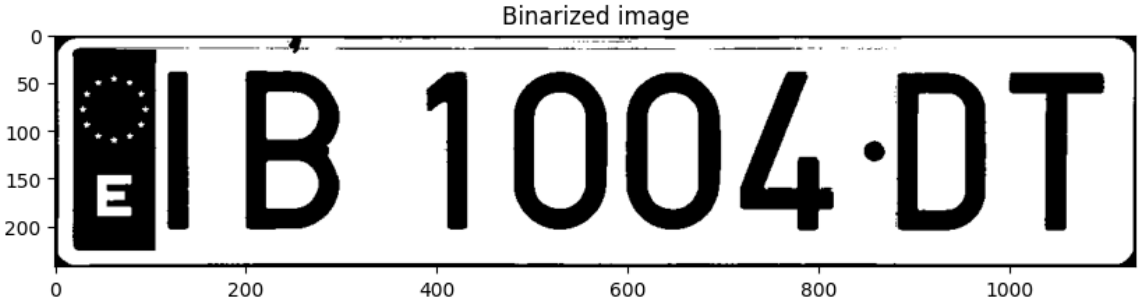
5.2.2 Expectation-Maximization (EM)

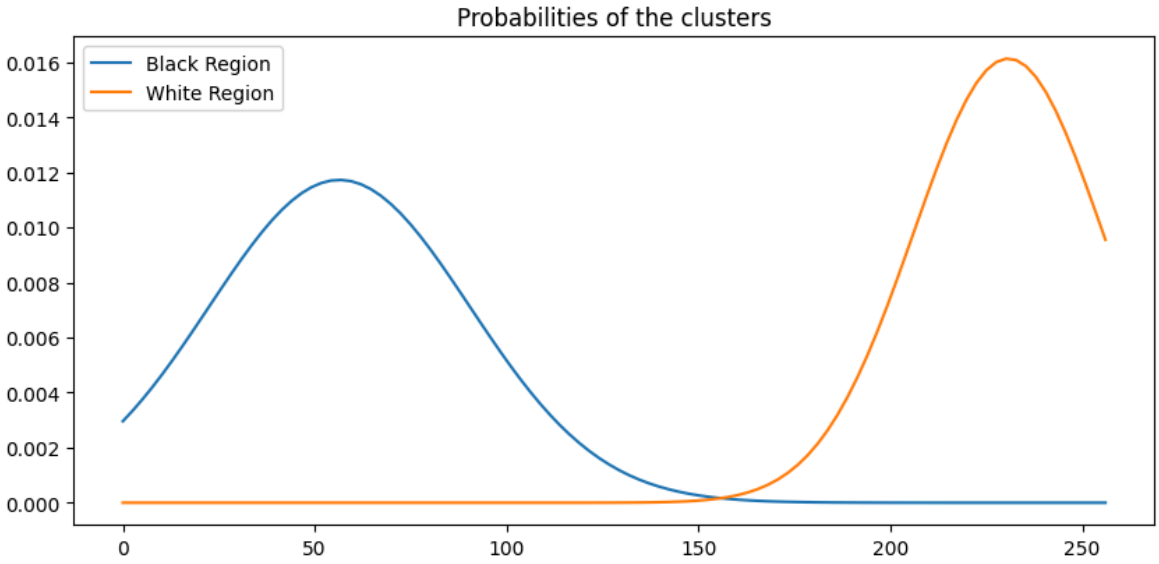
Expectation-Maximization (EM) is the generalization of the K-means algorithm, where each cluster is represented by a Gaussian distribution, parametrized by a mean and a covariance matrix, instead of just a centroid. It's a *soft clustering* since it doesn't give *hard* decisions where a pixel belongs or not to a cluster, but the probability of that pixel belonging to each cluster C_j, that is, $p(x|C_j) \sim N(\mu_j, \Sigma_j)$. This implies that at each algorithm iteration not just the mean of each cluster is refined (as in K-means), but also their covariance matrices.

Before going into detail on the theory behind EM, it is worth seeing how it performs in the car plate problem. OpenCV provides a class implementing the needed functionality for applying EM segmentation to an image, called `cv2.ml.EM()`. All methods and parameters are fully detailed in the documentation, so it is a good idea to take a look at it.

In [6]:
```
matplotlib.rcParams['figure.figsize'] = (10.0, 10.0)
cv2.setRNGSeed(5)

# Define parameters
n_clusters = 2
covariance_type = 0 # 0: covariance matrix spherical. 1: covariance matrix diago
n_iter = 10
epsilon = 0.2

# Create EM empty object
em = cv2.ml.EM_create()

# Set parameters
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, n_iter, epsilon)
em.setClustersNumber(n_clusters)
em.setCovarianceMatrixType(covariance_type)
em.setTermCriteria(criteria)

# Read grayscale image
image = cv2.imread(images_path + "plate.jpg",0)

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

# Apply EM
_, _, labels, _ = em.trainEM(flattened_img)

# Reshape labels to image size (binarization)
binarized = labels.reshape((image.shape))
```

```
# Show original image
plt.subplot(2,1,1)
plt.title("Binarized image")
plt.imshow(binarized, cmap="gray")

# --------------- Gaussian visualization ---------------

plt.subplot(2,1,2)
plt.title("Probabilities of the clusters")

# Get means and covs (for grayscale 1D both)
means = em.getMeans()
covs = em.getCovs()

# Get standard deviation as numPy array
sigmas = np.sqrt(covs)
sigmas = sigmas[:,0,0]

# Cast list to numPy array
means = np.array(means)[:,0]

# Plot Gaussians
x = np.linspace(0, 256, 100)
plt.plot(x, stats.norm.pdf(x, loc = means[0], scale = sigmas[0]))
plt.plot(x, stats.norm.pdf(x, loc = means[1], scale = sigmas[1]))
plt.legend(['Black Region', 'White Region'])

plt.show()
```





As you can see, although in OpenCV k-means is implemented as a method and EM as a class, they operate in a similar way. In the example above, we are segmenting a car plate into two clusters, and **each cluster is defined by a Gaussian distribution** (a Gaussian distribution for the black region, and another one for the white region). This is the basis of EM, **but how it works**?

EM is an iterative algorithm that is divided into two main steps:

- First of all, it **initializes the mean and covariance matrix of each of the K clusters**. Typically, it picks at random (μ_j,Σ_j) and $P(C_j)$ (prior probability) for each cluster j.

- Then, it keeps iterating doing Expectation-Maximization steps until some stopping criteria is satisfied (e.g. when no change occurs in a complete iteration):

    1. **Expectation step:** calcule the probabilities of every point belonging to each cluster, that is $p(C_j|x_i), \forall i \in data$:

$$P(C_j|x_i) = \frac{p(x_i|C_j)p(C_j)}{p(x_i)} = \frac{p(x_i|C_j)p(C_j)}{\sum_i P(x_i|C_j)p(C_j)}$$

    assign x_i to the cluster C_j with the highest probability $P(C_j|x_i)$.

2. **Maximization step:** re-estimate the cluster parameters ((μ_j,Σ_j) and $p(C_j)$) for each cluster j knowing the expectation step results, which is also called *Maximum Likelihood Estimate* (MLE):

$$\mu_j = \frac{\sum_i p(C_j|x_i)x_i}{\sum_i p(C_j|x_i)}$$

$$\sum_j = \frac{\sum_i p(C_j|x_i)(x_i-\mu_j)(x_i-\mu_j)^T}{\sum_i p(C_j|x_i)}$$

$\backslash[5pt]$

$$p(C_j) = \sum_i p(C_j|x_i)p(x_i) = \frac{\sum_i p(C_j|x_i)}{N}$$

Note that if no other information is available, the priors are considered equally probable.

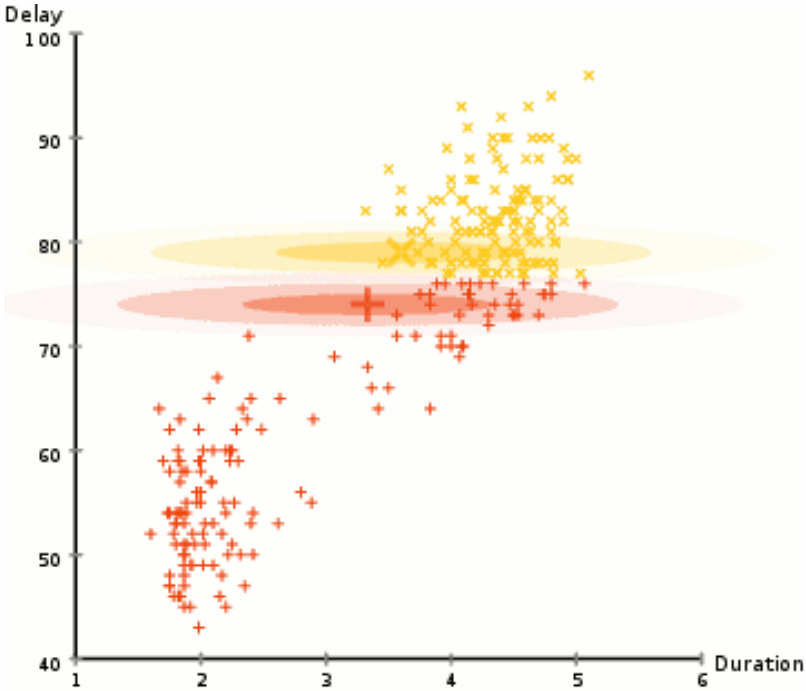


Fig 4. Example of an execution of the EM algorithm with two clusters, with details about the evolution of their associated Gaussian distributions.

Doesn't it remind you to the K-means algorithm? **What is the difference between them?**

The main difference is that K-means employs the **euclidian distance** to measure how near is a point to a cluster. In EM we use a distance in which **each dimension is weighted** by the **covariance matrix** of each cluster, which is also called **Mahalanobis distance**. Furthermore, for k-means a point of data **belongs or not to** a cluster, in EM a point of data have a higher or lower **probability** to belong to a cluster. The table below summarizes other differences:

	K-means	EM
Cluster representation	Mean	Mean, (co)variance
Cluster initialization	Randomly select K means	Initialize K Gaussian distributions (μ_j,Σ_j) and $P(C_j)$
Expectation: Estimate the cluster of each data	Assign each point to the closest mean	Compute $P(C_j\|x_i)$
Maximization: Re-estimate the cluster parameters	Compute means of current clusters	Compute new (μ_j, Σ_j), $P(C_j)$ for each cluster j

If you still curious about EM, you can find here a more detailed explanation.

OpenCV pill

Going back to code, working with EM we have to specify a covariance matrix type using `em.setCovarianceMatrixType()`. Also, when you applying `em.trainEM()` it doesn't return the centroid of the clusters, it is possible to get them calling `em.getMeans()`.

ASSIGNMENT 2: Color quantization with YCrCb color space

In the next example, color quantization is realized using the YCrCb color space instead of RGB. Recall that you have more info about such a space available in Apendix 12.2 Color spaces. In this way, color quantization is only applied to the two color bands Cr and Cb, neglecting the grayscale one Y.

Notice that in this case, the feature space has 2 dimensions, one for the Cr band, and another dimension for the Cb, hence the feature vector describing the i^{th} pixel results $x_i = [Cr_i, Cb_i]$.

Let's see how it works!

What to do? Test and understand the following code.

In [7]:
```
# Assignment 2
matplotlib.rcParams['figure.figsize'] = (15.0, 15.0)
cv2.setRNGSeed(5)

# Define parameters

n_clusters = 3 # Don't modify this parameter for this exercise

covariance_type = 0 # 0: Spherical covariance matrix. 1: Diagonal covariance mat
n_iter = 10
epsilon = 0.2

# Create EM empty object
em = cv2.ml.EM_create()
```

```
# Set parameters
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, n_iter, epsilon)
em.setClustersNumber(n_clusters)
em.setCovarianceMatrixType(covariance_type)
em.setTermCriteria(criteria)

# Read color image
image = cv2.imread(images_path + "malaga.png")

# Convert to YCrCb
image = cv2.cvtColor(image, cv2.COLOR_BGR2YCrCb)

# Take color bands (2 lasts)
color_bands = image[:,:,1:3]

# Flatten image
flattened_img = color_bands.reshape((-1,2))
flattened_img = np.float32(flattened_img)

# Apply EM
_, _, labels, _ = em.trainEM(flattened_img)

# Colour resultant labels
centers = em.getMeans()
centers = np.uint8(centers)
res = centers[labels.flatten()]

# Reshape to original shape
color_bands = res.reshape((image.shape[0:2]) + (2,))

# Merge original first band with quantized color bands
quantized = np.zeros(image.shape)
quantized[:,:,0] = image[:,:,0]
quantized[:,:,[1,2]] = color_bands

# Cast to unsigned data dype
quantized = np.uint8(quantized)

# Reconvert to RGB
quantized_rgb = cv2.cvtColor(quantized, cv2.COLOR_YCrCb2RGB)
image_rgb = cv2.cvtColor(image, cv2.COLOR_YCrCb2RGB)

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

# Show resultant image
plt.subplot(1,2,2)
plt.title("YCrCb quantized colors (3 colors)")
plt.imshow(quantized_rgb);
```

Thinking about it (2)

Once you understanded the code above, **answer the following questions:**

- What are the dimensions of the means u_j and the covariance matrices Σ_j?

  The means u_j would have dimensions 1x2, while the covariance matrices Σ_j would have dimensions 2x2, as we are using only 2 bands of the YCrCb space (Cr and Cb).

- What are the dimensions of the input to `trainEM()`? Why?

  The dimensions of the input to trainEM() are (239400,2), the first one being the number of pixels in the image, as these are all the samples that will be used in the algorithm. The other value is 2 as it is the number of features (recall that we are using 2 bands of the YCrCb colour space: Cr and Cb).

- Why are the obtained results so good using only 3 clusters?

  That is because we are using only 2 colour bands of the YCrCb colour space, while the Y band remains unchanged.

- What compression would be better in terms of space in memory, a 16-color compression in a RGB image (that is, each band uses 16 different colors instead of the original 256) or a 4-color compression in a YCrCb image? *Hint: consider the bits needed to codify such information. Hint 2: the grayscale band in YCrCb, that is, Y, is not compressed.*

  With this compression, the RGB image would use 12 bits (16 colours = 4 bits per band), while the YCrCb would use 12 bits (256 colours in the Y band = 8 bits; 4 colours in the Cr and Cb bands = 2 bits per band). It ends up they would use the same memory space.

Diving deeper into covariance matrices

There are 3 types of covariance matrices: **spherical covariances**, **diagonal covariances** or **full covariances**:

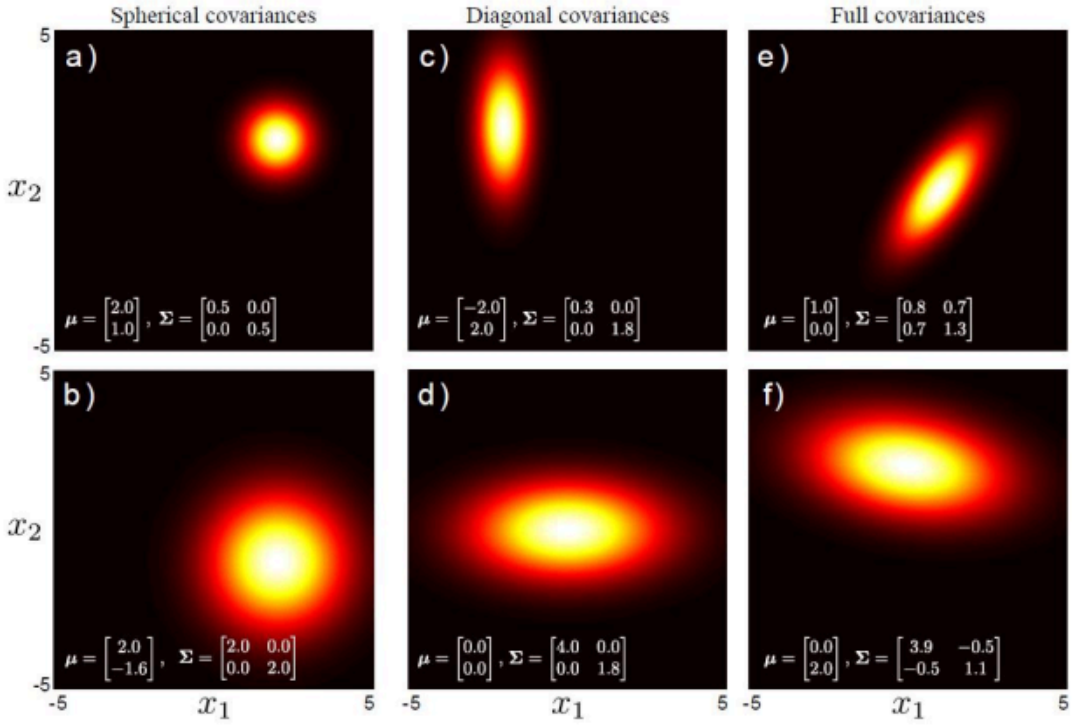


Fig 5. Examples of different types of covariance matrices.

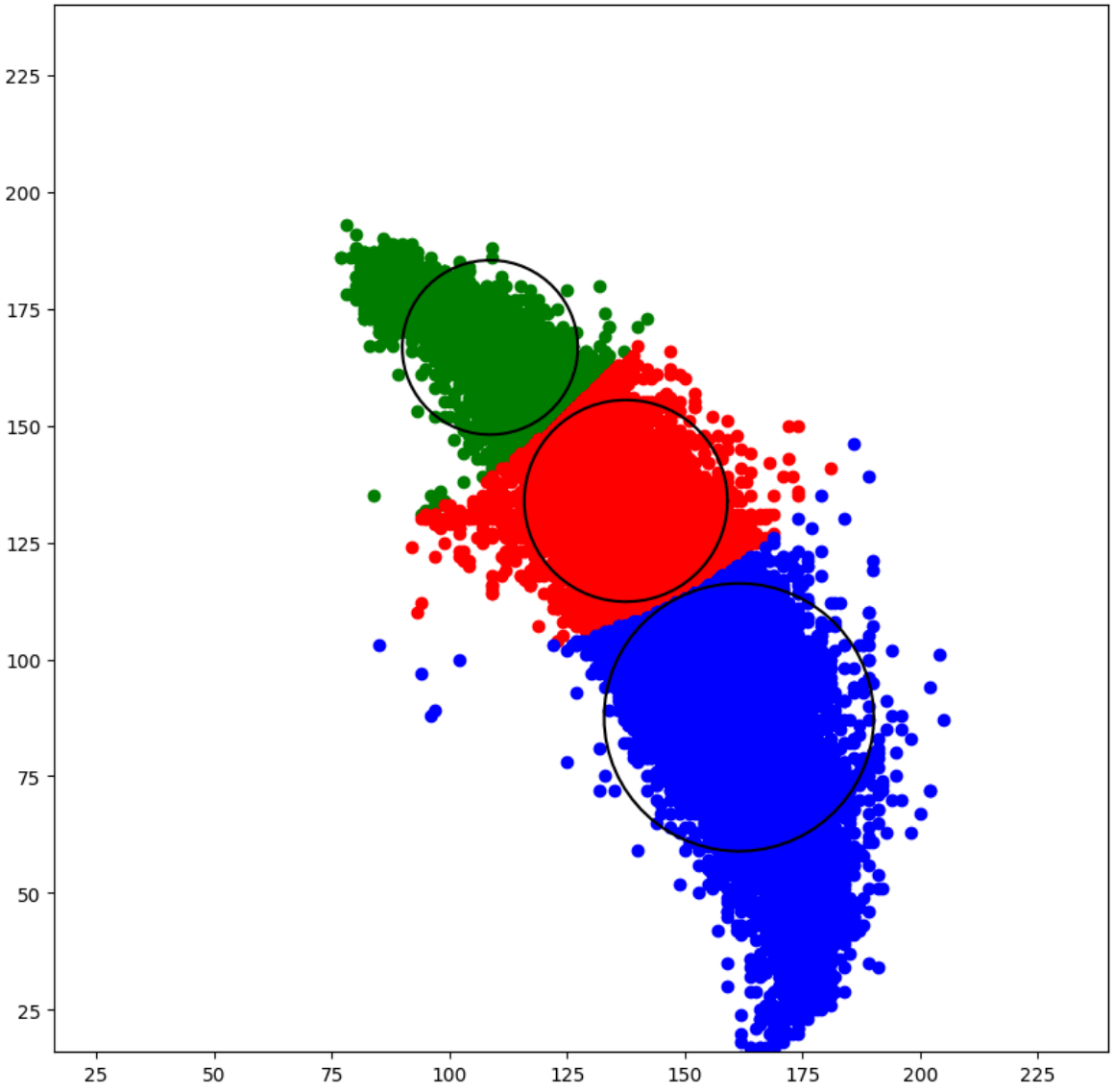
ASSIGNMENT 3: Visualizing clusters from EM

Next, you have a code for visualizing the clusters in the YCrCb color space using EM.

What to do? Run the previous example modifying the type of covariance in the EM algorithm and visualize the changes using the following code.

In [8]:
```
# Assignment 3
matplotlib.rcParams['figure.figsize'] = (10.0, 10.0)

# Get means (2D) and covariance matrices (2x2)
means = np.array(em.getMeans())
covs = np.array(em.getCovs())

# Create figure
fig, ax = plt.subplots()
plt.axis([16, 240, 16, 240])

# Get points contained in each cluster
cluster_1 = np.any(color_bands == np.unique(res,axis=0)[0,:],axis=2)
cluster_2 = np.any(color_bands == np.unique(res,axis=0)[1,:],axis=2)
cluster_3 = np.any(color_bands == np.unique(res,axis=0)[2,:],axis=2)
cluster_1 = image[cluster_1]
cluster_2 = image[cluster_2]
cluster_3 = image[cluster_3]

# Plot them
plt.plot(cluster_1[:,1],cluster_1[:,2],'go')
plt.plot(cluster_2[:,1],cluster_2[:,2],'ro')
plt.plot(cluster_3[:,1],cluster_3[:,2],'bo')

# Plot ellipses representing covariance matrices
PlotEllipse(fig, ax, np.vstack(means[0,:]), covs[0,:,:], 2, color='black')
```

```
PlotEllipse(fig, ax, np.vstack(means[1,:]), covs[1,:,:], 2, color='black')
PlotEllipse(fig, ax, np.vstack(means[2,:]), covs[2,:,:], 2, color='black')

fig.canvas.draw()
```



Thinking about it (3)

Answer the following questions about how clustering works in EM:

- What are the differences between each type of covariance?

  Spherical covariance has equal values (and not null) in the diagonal and null values otherwise, resulting in circular clusters. Diagonal covariance has different values (and not null) in the diagonal and null values otherwise, resulting in horizontally or vertically eliptical clusters. Finally, the full covariance has different values in all cells, resulting in eliptical clusters (not necessarily horizontal or vertical).

- What type of covariance makes EM equivalent to k-means?

  Spherical covariance, as it represents the euclidean distance better, which is what k-means want to minimise.

ASSIGNMENT 4: Applying EM considering different color spaces

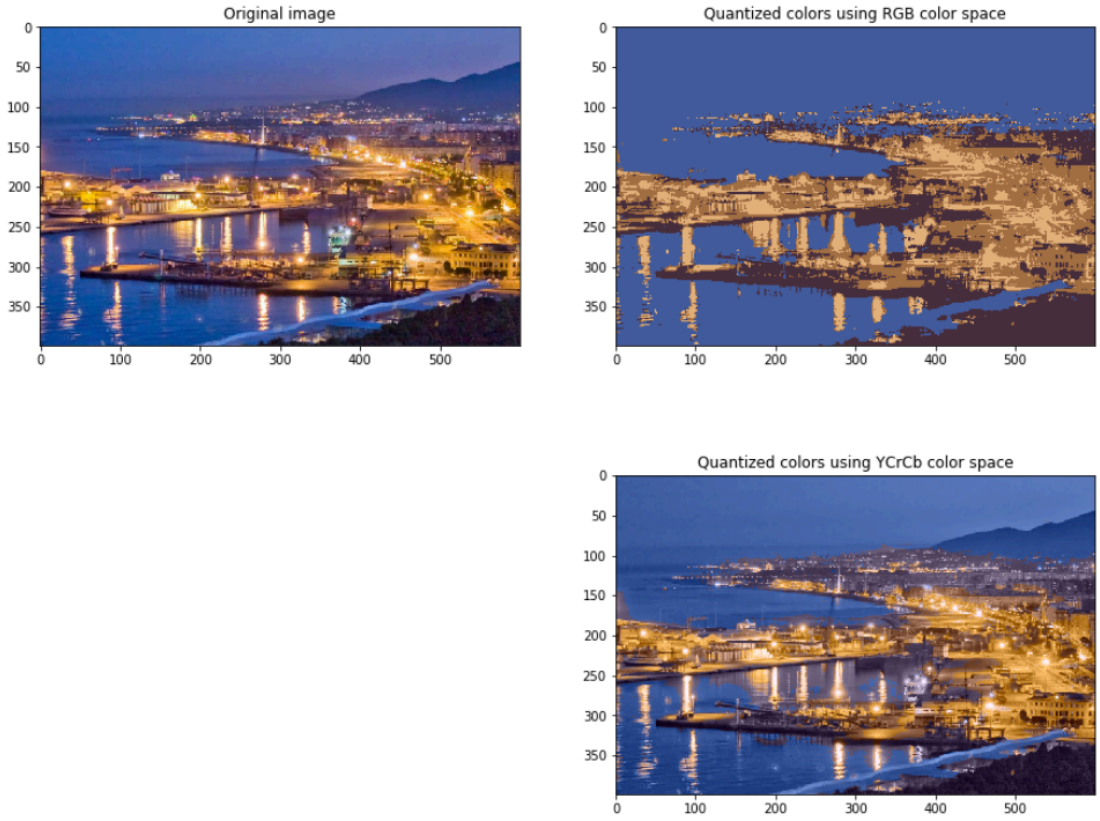
It's time to show what you have learned about **EM** and **color spaces**!

What is your task? You are asked to **compare color quantization in a RGB color space and in a YCrCb color space**.

For that:

- apply Expectation-Maximization to `malaga.png` using 4 clusters (colors) to both the RGB-space image and the YCrCb-space one,
- and display both results along with the original image.

Expected output:



In [9]:
```
# Assignment 4
matplotlib.rcParams['figure.figsize'] = (15.0, 12.0)
cv2.setRNGSeed(5)

# Define parameters
n_clusters = 4
covariance_type = 2 # 0: covariance matrix spherical. 1: covariance matrix diago
n_iter = 10
epsilon = 0.2

# Create EM empty objects
em = cv2.ml.EM_create()

# Set parameters
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, n_iter, epsilon)
```

```
em.setClustersNumber(n_clusters)
em.setCovarianceMatrixType(covariance_type)
em.setTermCriteria(criteria)

# Read image
image = cv2.imread(images_path + "malaga.png")

# Convert image to RGB
image_RGB = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Convert image to YCrCb
image_YCrCb = cv2.cvtColor(image, cv2.COLOR_BGR2YCrCb)

# Flatten RGB image
flattened_RGB = image_RGB.reshape((-1,3))
flattened_RGB = np.float32(flattened_RGB)

# Flatten color bands of YCrCb image
color_bands_YCrCb = image_YCrCb[:,:,1:3]
flattened_YCrCb = color_bands_YCrCb.reshape((-1,2))
flattened_YCrCb = np.float32(flattened_YCrCb)

# Apply EM and get centers of clusters
_, _, labels_RGB, _ = em.trainEM(flattened_RGB)
centers_RGB = em.getMeans()
centers_RGB = np.uint8(centers_RGB)

_, _, labels_YCrCb, _ = em.trainEM(flattened_YCrCb)
centers_YCrCb = em.getMeans()
centers_YCrCb = np.uint8(centers_YCrCb)

# Colour resultant labels
res_RGB = centers_RGB[labels_RGB.flatten()]
res_YCrCb = centers_YCrCb[labels_YCrCb.flatten()]

# Reshape to original shape
quantized_RGB = res_RGB.reshape((image.shape))
quantized_colors_YCrCb = res_YCrCb.reshape((image.shape[0:2]) + (2,))

# Merge original first band with quantized color bands for YCrCb image
quantized_YCrCb = np.zeros(image.shape)
quantized_YCrCb[:,:,0] = image_YCrCb[:,:,0]
quantized_YCrCb[:,:,[1,2]] = quantized_colors_YCrCb

# Cast YCrCb image to unsigned data dype
quantized_YCrCb = np.uint8(quantized_YCrCb)

# Reconvert YCrCb image back to RGB
quantized_YCrCb = cv2.cvtColor(quantized_YCrCb, cv2.COLOR_YCrCb2RGB)

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

# Show resultant quantization using RGB color space
plt.subplot(2,2,2)
plt.title("Quantized colors using RGB color space")
plt.imshow(quantized_RGB)
```

```
# Show resultant quantization using YCrCb color space
plt.subplot(2,2,4)
plt.title("Quantized colors using YCrCb color space")
plt.imshow(quantized_YCrCb);
```

Conclusion

Congratulations for getting this work done! You have learned:

- how k-means clustering works and how to use it,
- how EM algorithm performs and how to employ it,
- how to carry out color quantization and the importance of color spaces in this context, and
- some basics for image compression.

OPTIONAL

You have used YCrCb in this notebook because you were already fimiliar with it. The truth is that, in color quantization matters, Lab color space is commonly used.

Surf the internet for information about the Lab color space and then **answer the following questions**:

- How does Lab color space work?
- Why is it typically used for color quantization?

END OF OPTIONAL PART

References

[1]: Borenstein, Eran, Eitan Sharon, and Shimon Ullman. Combining top-down and bottom-up segmentation.. IEEE Conference on Conference on Computer Vision and Pattern Recognition Workshop, 2004.