5.2 Region-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

No description has been provided for this image

$\\[5pt]$

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!) or content-based image retrieval.

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

In [11]:
```
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("..")
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:

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

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.

No description has been provided for this image

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

$\\[5pt]$

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

No description has been provided for this image

Fig 2. Example of how the centroids evolve over time

$\\[5pt]$

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).$\\[10pt]$

No description has been provided for this image

Fig 3. Final segmentation result

$\\[5pt]$

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 [12]:
```
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 t
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, it, e

    # 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

    # Colour resultant labels
    center = np.uint8(center) # Get center coordinates as unsigned intege
    print(center)
    flattened_img = center[label.flatten()] # Get the color (center) assi

    # 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==cente
```

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 [13]:
```
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));
```

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.

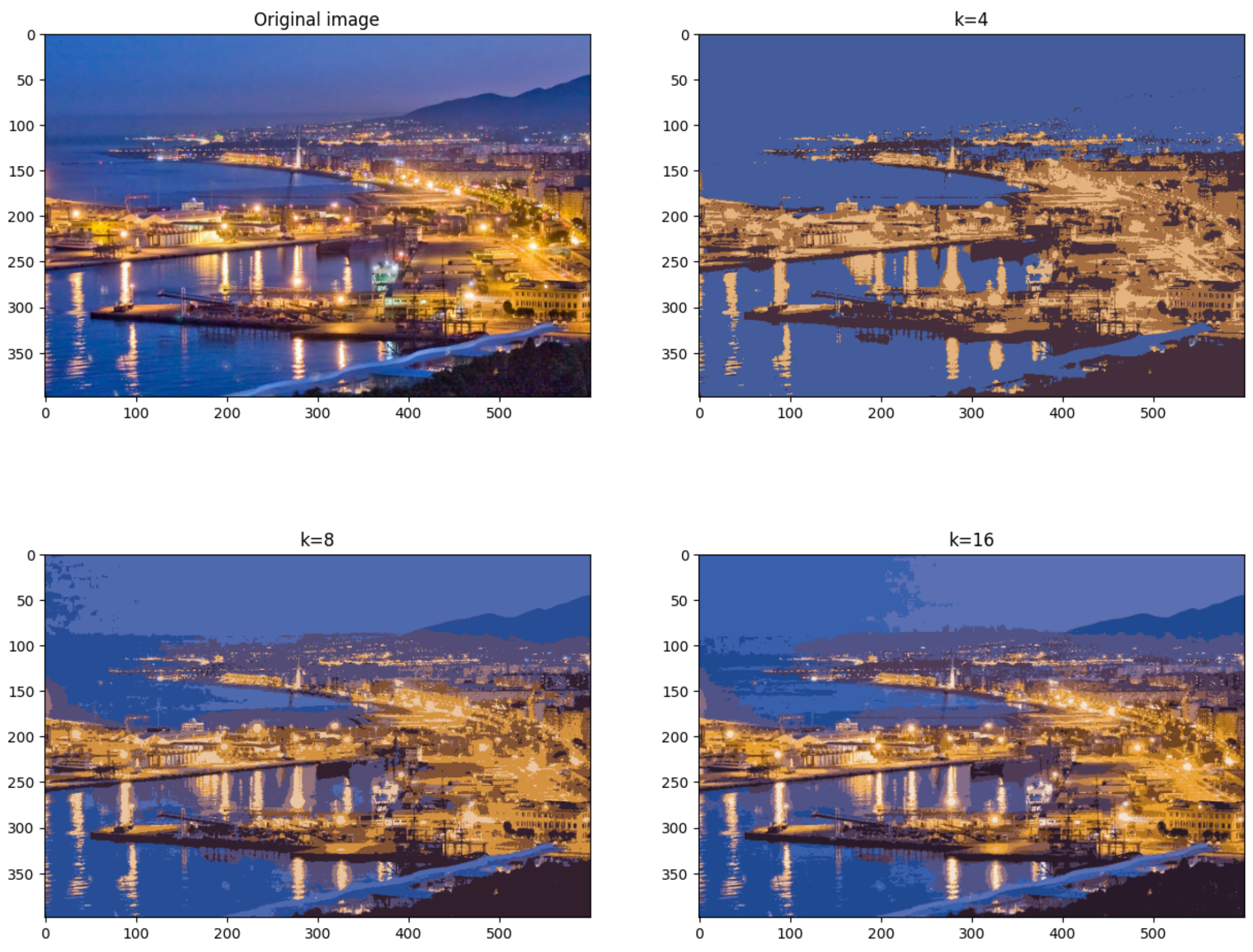
Notice that in this case we are using 3 features per pixel, their R, G and B values, so the input data for the kmeans function has the dimensions $num\_pixels \times 3$.

Expected output:

No description has been provided for this image

```
In [14]: # 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, epsil

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

# 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?

  The function is applied to the flattened image data with k clusters. When the k-means algorithm starts, the initial cluster centers are randomly initialized. In each iteration, the algorithm assigns each pixel to the cluster with the closest centroid, then it updates the cluster centers by computing the mean of the pixel colors assigned to each cluster. Updates continue till either we reach convergence or after reaching the maximum number of iterations (here 10).

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

  I set it=10. This value is appropriate since the k-means algorithm typically converges quickly, especially when using a small number of clusters. Running more iterations requires more computational resources, and in this case, it would not give the result a such a big improvement.

- 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 should apply, after color quantization, color compression, meaning that I can for exaple just take into account 16 colours and not 256. With 16 colours I would just need 4 bits to represent them instead of the 8 of the 256. Overall the total number of bits to represent the image would be 3x4 = 12 instead of 24

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 [15]:
```
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, epsil

start = time.process_time() # Start timer

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

# Apply k-means using k=8
_,label,center=cv2.kmeans(flattened_img,8,None,criteria,30,cv2.KMEANS_RAN

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

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,8,None,criteria,30,cv2.KMEANS_RAN
print("K-means in the grayscale image:", round(time.process_time() - star
```

```
Measuring the execution time needed for ...
K-means in the RGB image: 3.03412 seconds
K-means in the grayscale image: 1.80078 seconds
```

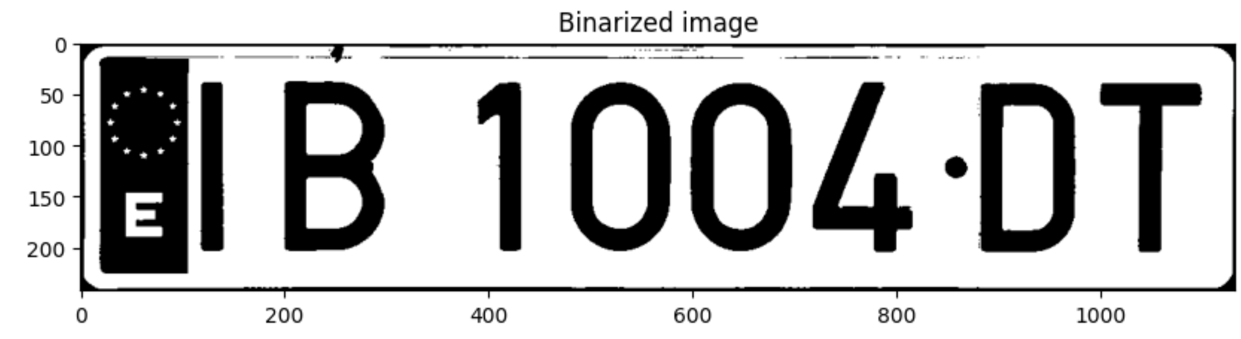
5.2.2 Expectation-Maximization (EM)

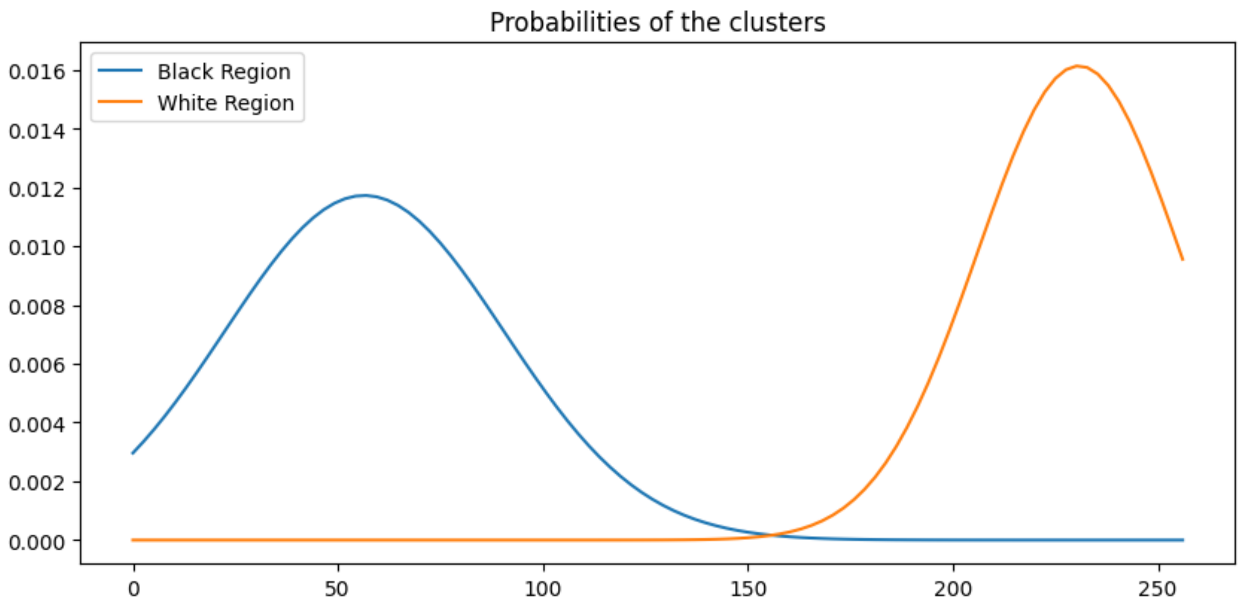
Expectation-Maximization (EM) is the generalization of the K-means algorithm, where each cluster is represented by a Gaussian distribution, parametrized by a mean and a covariance matrix, instead of just a centroid. It's a *soft clustering* since it doesn't give *hard* decisions where a pixel belongs 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 [16]:
```
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 matri
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, e
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)
```

```python
# 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):$\\[1pt]$

  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)}$$
assign x_i to the cluster C_j with the highest probability $P(C_j|x_i)$.$\\[10pt]$ 2. **Maximization step:** re-estimate the cluster parameters ((μ_j,Σ_j) and $p(C_j)$) for each cluster j knowing the expectation step results, which is also called *Maximum Likelihood Estimate* (MLE): $$\mu_j=\frac{\sum_i p(C_j|x_i)x_i}{\sum_i p(C_j|x_i)}$$ $\\[5pt]$ $$\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)}$$$\[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.

No description has been provided for this image

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

$\\[5pt]$
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 (\$\mu_j\$,\$\Sigma_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 (\$\mu_j,\Sigma_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!

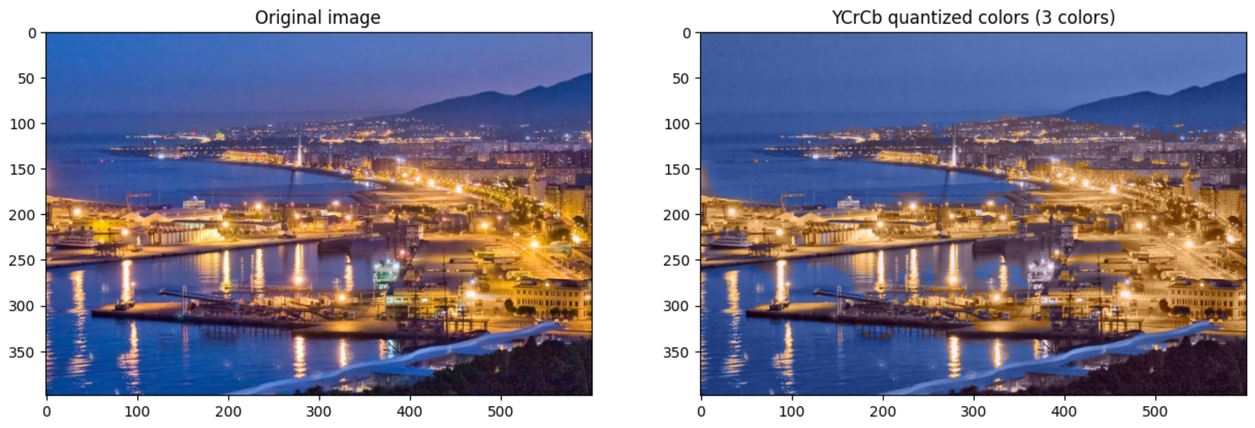
What to do? Test and understand the following code.

In [17]:
```python
# 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 = 2 # 0: Spherical covariance matrix. 1: Diagonal covaria
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, e
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)

print(flattened_img.shape)

# 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);
```

```
(239400, 2)
```



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 $\Sigma_j$?

  The means $u_j$ is be of dimension (3, 2), as we have n_clusters = 3 and 2 color channels (Cr ad Cb). The covariance matrices $|Sigma_j$ dimension is (3, 2, 2), with n_clusters = 3, and (2, 2) represents the covariance matrix for each cluster that has these dimensions since we are in a 2 dimensions feature space

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

  the input is flatten_img which has (239400, 2) as dimension. 239400 is the number of data points or samples in our dataset (the image), these are the point that we want to cluster using the EM algorithm. The samples are the pixels. the second parameter is 2 that are the number of dimensions used to describe each data point (color channels r and Cb).

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

  Our image doesn't present a lot of colors variations, there are three main color regions that are yellow, blue and the sort of brown that comes from the mix of the two. These regions are naturally separated meaning they naturally form 3 clusters

- 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.*

  16-Color Compression in RGB Image: For each of the three color channels 16 different colors are used, and to represent them we need 4 bits. So it's 12 bits for each pixel in RGB. 4-Color Compression in YCrCb Image: 4 colors are used and 2 bits to represent them.The two channels Cr and Cb require 2 bits. Additionally, we have the Y component that is not compressed meaning we need 8 bits to represent all the 256 colors. So we need 8 + 2x2 = 12 bits. They are exactly the same in terms of memory space.

Diving deeper into covariance matrices

There are 3 types of covariance matrices: **spherical covariances**, **diagonal covariances** or **full covariances**: $\\[10pt]$

No description has been provided for this image

Fig 5. Examples of different types of covariance matrices.
$\\[10pt]$

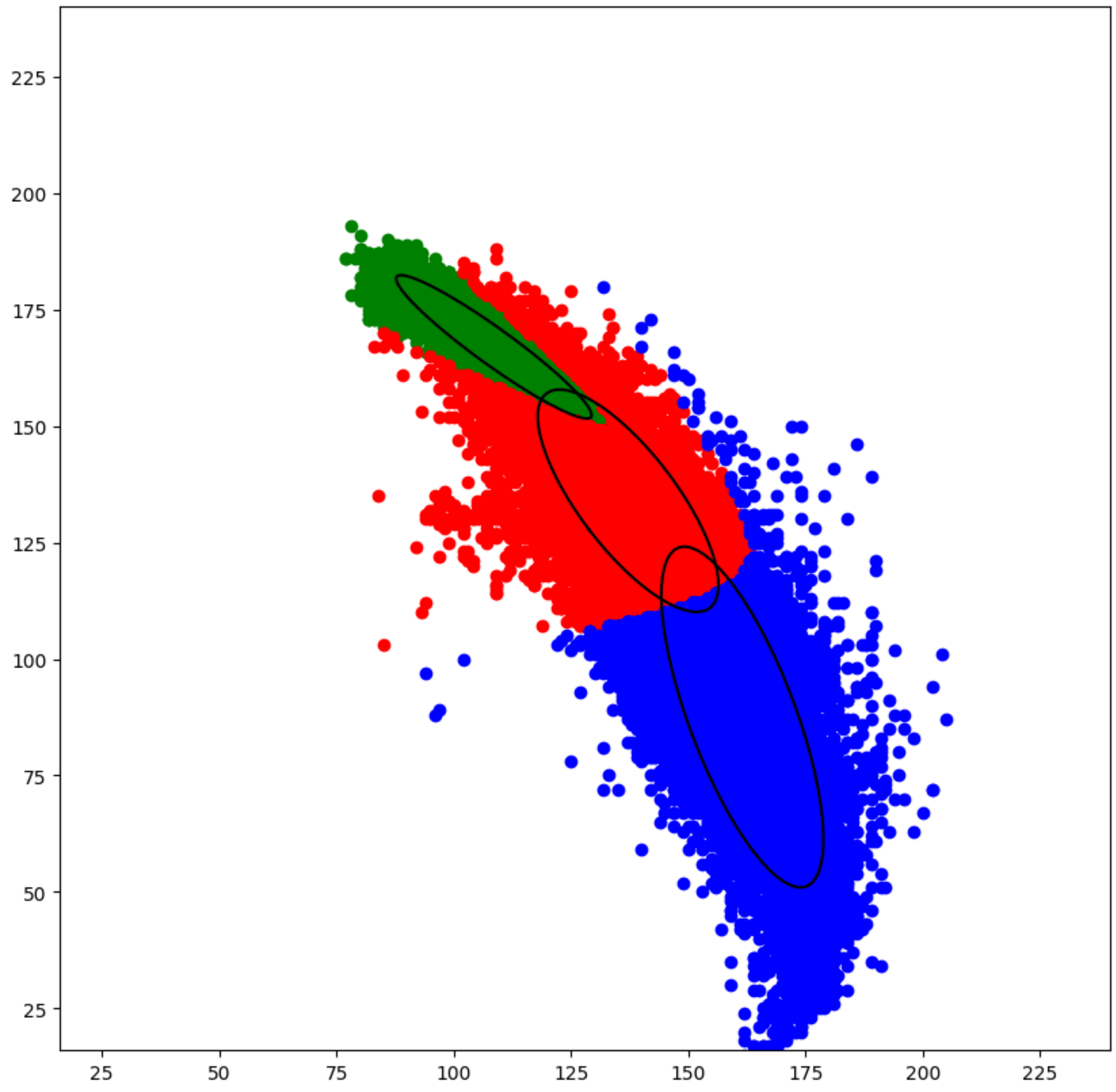
ASSIGNMENT 3: Visualizing clusters from EM

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 [18]:
```python
# 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.PlotEllipse(fig, ax, np.vstack(means[0,:]), covs[0,:,:], 2, c
PlotEllipse.PlotEllipse(fig, ax, np.vstack(means[1,:]), covs[1,:,:], 2, c
PlotEllipse.PlotEllipse(fig, ax, np.vstack(means[2,:]), covs[2,:,:], 2, c

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?

  In a spherical covariance matrix (type 0), all dimensions have the same variance.The shape of the cluster is spherical and the same for all the 3. In a diagonal covariance matrix (type 1) each dimension has its own variance, but there is no correlation between dimensions. The shape of the cluster is be elliptical. In the full covariance matrix (type = 2) each dimension has its own variance, and there can be non-zero covariances between dimensions, this makes this type the most flexible. The shape of the clusters can be elliptical or have more complex shapes and less regular, as shown in the plot above

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

Type 0 because each cluster is represented by its mean (centroid) and has a spherical shape with equal variances along all dimensions

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.

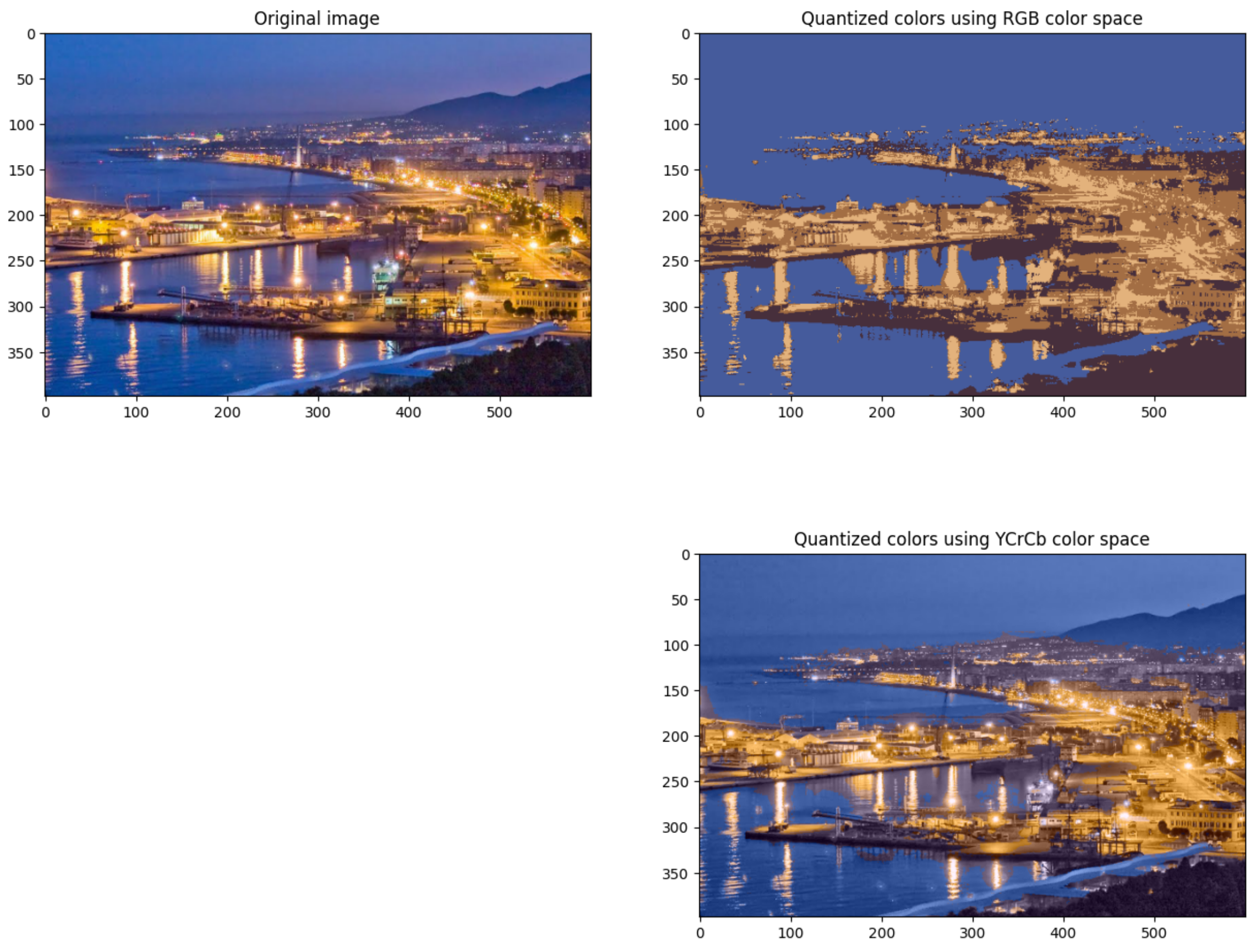
Expected output:

No description has been provided for this image

In [19]:
```
# 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 matri
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, e

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)
```

```python
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.

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.