TP2 - MRF

February 1, 2024

1 PW 2 - Bayesian analysis with MRF

For this part we will perform the binary classification of a grayscale image ***Iobservee*** (image of the observations, realization y of the field Y) using a Markovian model.

In this ideal case, we are given the ideal solution x (binary image ***IoriginalBW***), realization of the field of classes X, which will be used to evaluate the quality of the solution $\hat{x}$ that we will have obtained. (NB: In practice usually, we don't have access to x).

In this part we will use the ***display*** function to visualize the images.

[ ]:
```
import math
import os
import platform
import random
import ssl
import tempfile

import imageio
import matplotlib.pyplot as plt
import numpy as np
from bokeh.io import output_notebook
from bokeh.plotting import figure, output_file
from bokeh.plotting import show
from bokeh.plotting import show as showbokeh
from scipy import ndimage as ndi
from skimage import io

ssl._create_default_https_context = ssl._create_unverified_context
output_notebook()
```

[ ]:
```
def affiche_pour_colab(im, MINI=None, MAXI=None, titre=""):  # special colab,␣
 ↪don't look
    def normalise_image_pour_bokeh(X, MINI, MAXI):
        if MAXI == None:
            MAXI = np.max(X)
        if MINI == None:
            MINI = np.min(X)
        imt = np.copy(X.copy())
        imt = np.clip(imt, MINI, MAXI) / (MAXI - MINI)
        imt[imt < 0] = 0
        imt[imt > 1] = 1
        imt *= 255
        sortie = np.empty((*imt.shape, 4), dtype=np.uint8)
        for k in range(3):
            sortie[:, :, k] = imt
        sortie[:, :, 3] = 255
        return sortie

    img = im
    img = normalise_image_pour_bokeh(np.flipud(im), MINI, MAXI)
    p = figure(
        tooltips=[("x", "$x"), ("y", "$y"), ("value", "@image")],
        y_range=[0, im.shape[0]],
        x_range=[0, im.shape[1]],
    )
    # p.x_range.range_padding = p.y_range.range_padding = 0
    # must give a vector of images
    p.image(
        image=[np.flipud(im)],
        x=0,
        y=0,
        dw=im.shape[1],
        dh=im.shape[0],
        palette="Greys9",
        level="image",
    )
    p.xgrid.visible = False
    p.ygrid.visible = False
    showbokeh(p)


def affiche(im, MINI=0.0, MAXI=None, titre="", printname=False):
    affiche_pour_colab(
        im, MINI=MINI, MAXI=MAXI, titre=titre
    )  # under google colab many options disappear
```

The scikit-image library allows to read images from a URL. The “display” function allows to access directly to the grey levels and to the pixels positions by using the mouse.

```
[ ]: im_obs = io.imread(
         "https://perso.telecom-paristech.fr/tupin/cours/IMA203/TPMARKOV/Iobservee.
       ↪png"
     )
```

```
affiche(im_obs)
```

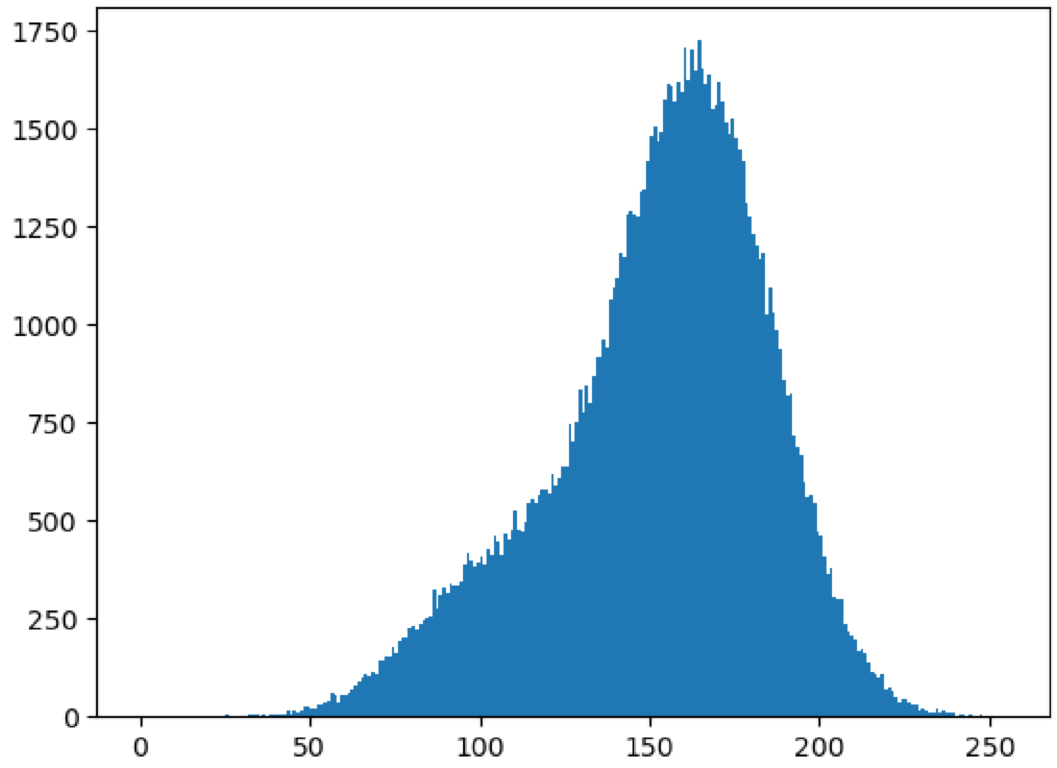
[ ]:
```
im_ori = io.imread(
    "https://perso.telecom-paristech.fr/tupin/cours/IMA203/TPMARKOV/
↪IoriginaleBW.png"
)

affiche(im_ori)
```

The objective is to estimate x from y using a prior on $P(X)$ in the form of a Markovian model. We note x_s the class of the pixel s (that we are looking for), and y_s the observed gray level. The objective is to use a global model on the random field X to classify the image. As we have seen in class, this amounts to minimizing the following energy:

$$U(x|y) = \sum_s -ln(P(Y_s = y_s|X_s = x_s)) + \sum_c U_c(x_s, s \in c)$$

[ ]:
```
# study of the global distribution of the image
# display of the histogram of the image
plt.figure()
plt.hist(im_obs.ravel(), range=[0, np.max(im_obs)], bins=np.max(im_obs))
plt.show()
```



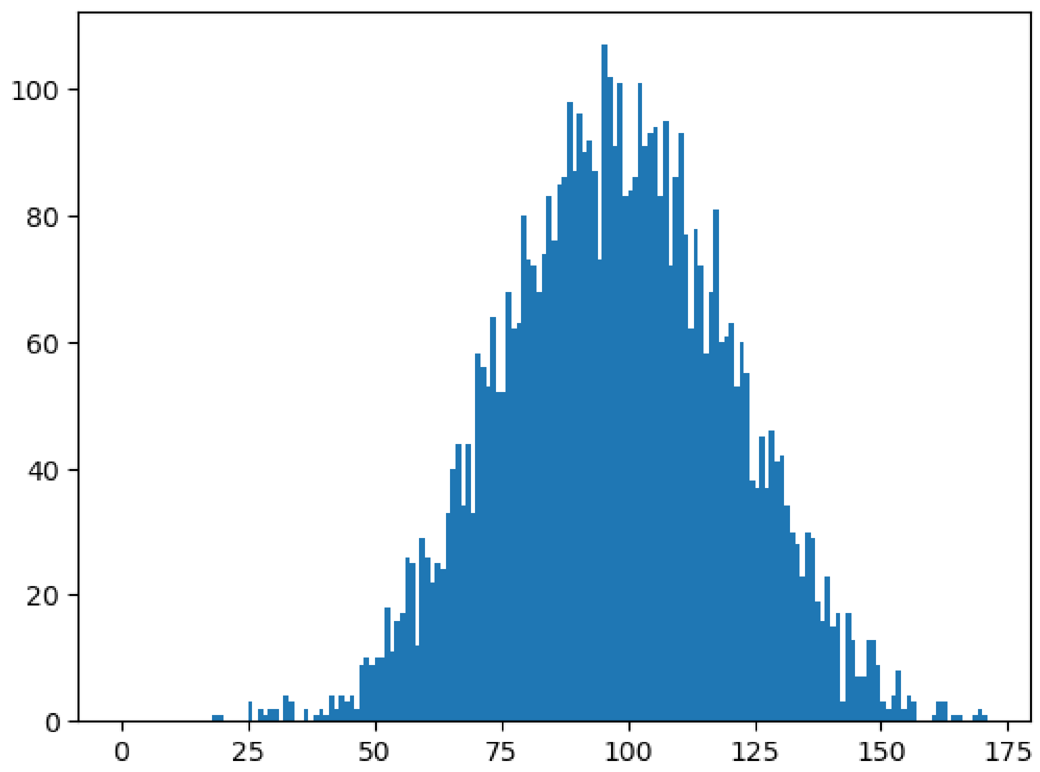
1.1 1. Analysis of the gray level distributions

In this part, we learn the probabilities $P(Y_s = y_s|X_s)$, that is to say $P(Y_s = y_s|X_s = 0)$ and $P(Y_s = y_s|X_s = 1)$. This is equivalent to studying the histogram of gray levels of pixels that are in class 0 and pixels that are in class 1.

To perform this training, we need to select pixels belonging to class 0 on the one hand (dark area of the observed image), and pixels belonging to class 1 on the other hand (light area of the observed image).

We can select pixels manually, using the command `v0=I[i1:i2,j1:j2]` in Python which puts in a vector all the values of the pixels of the image I included between the indices $i1$ and $i2$ (rows), and $j1$ and $j2$ (columns).

[ ]:
```
# select a small window in a dark region of the image (class 0)
# be careful, the ordinates correspond to the rows and the abscissas to the␣
 ↪columns
crop_classe0 = im_obs[46:121, 174:246]
# visualise the window
affiche(crop_classe0)
# plot its histogram
plt.figure()
plt.hist(
    crop_classe0.ravel(), range=[0, np.max(crop_classe0)], bins=np.
 ↪max(crop_classe0)
)
plt.show()
```



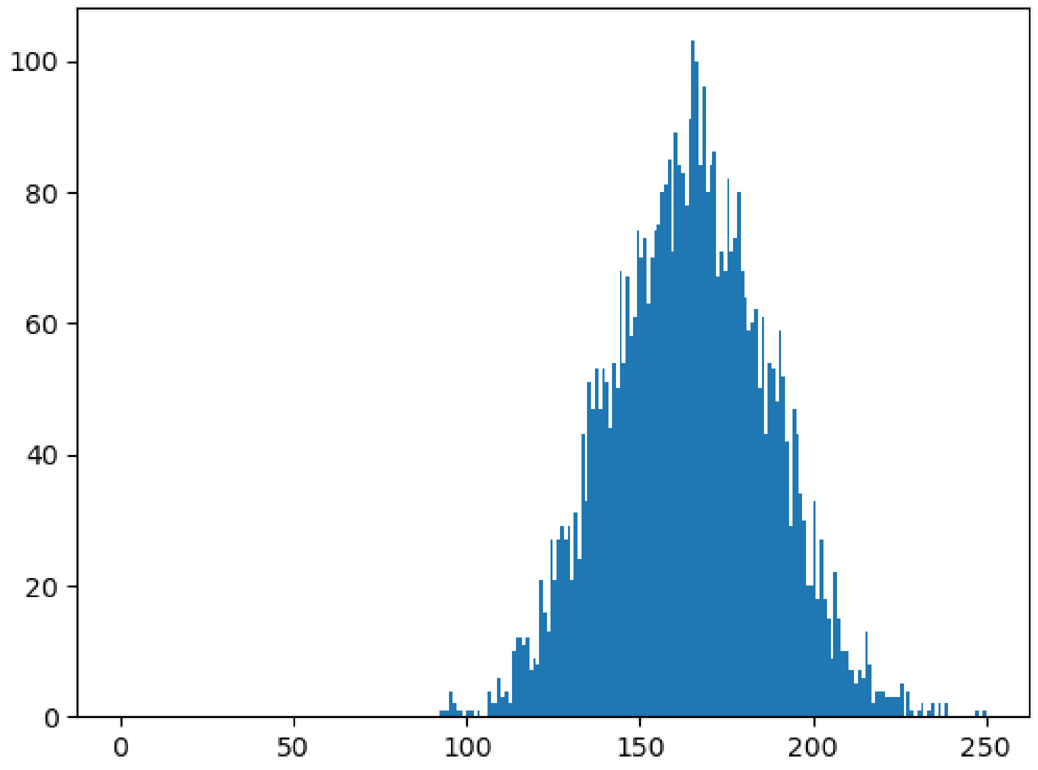
```
[ ]: # calculate its mean and variance
     #
     # By default, np.mean, np.zzz takes the two axes of the image.
     # You have to force the option if you want to do the average in row or in column
     # use np.mean and np.var
     m0 = np.mean(crop_classe0)
     var0 = np.var(crop_classe0)

     print(m0)
     print(var0)
```

```
97.40055555555556
502.5686293209876
```

```
[ ]: # select a small window in a dark region of the image (class 1)
     crop_classe1 = im_obs[230:300, 130:200]
     # plot its histogram
     plt.figure()
     plt.hist(
         crop_classe1.ravel(), range=[0, np.max(crop_classe1)], bins=np.
       ↪max(crop_classe1)
     )
```

```
plt.show()
```



[ ]:
```
# calculate its mean and variance
m1 = np.mean(crop_classe1)
var1 = np.var(crop_classe1)

print(m1)
print(var1)
```

```
163.87755102040816
512.7315368596419
```

1.1.1 Q1

What are the distributions followed by the grey levels in these two classes? Give the means and variances of the two classes that you have estimated.

1.1.2 A1

Both classes follow a Gaussian distribution:

- Class 0, corresponding to the dark regions, has a mean of a 97.40 and a variance of 502.57;
- Class 1, corresponding to the light regions, has a mean of a 163.88 and a variance of 512.73;

1.1.3 Q2

In the following, the variances will be assumed to be equal in order to simplify the energy expressions.

Suppose that we do not use a Markov model on X and that we classify a pixel only according to its grey level by comparing $P(Y_s = y_s|X_s = 0)$ and $P(Y_s = y_s|X_s = 1)$. Show that this amounts to thresholding the image and give the value of the optimal threshold as a function of the parameters found previously (we say that we are doing a classification by punctual maximum likelihood).

1.1.4 A2

When classifying a pixel based solely on its gray level, the probability distributions can be described as follows:

- $P(Y_s|0) = \mathcal{N}(97, 503)$
- $P(Y_s|1) = \mathcal{N}(163, 512)$

We express the Gaussian distribution as $P(Y_s|X_s) = \frac{1}{\sqrt{2\pi\sigma_{x_s}^2}} e^{\frac{-(y_s-\mu_{x_s})^2}{2\sigma_{x_s}^2}}$.

To classify a pixel as class 0, the condition is:

$P(Y_s|0) > P(Y_s|1)$

$\frac{1}{\sqrt{2\pi\sigma_0^2}} e^{\frac{-(y_s-\mu_0)^2}{2\sigma_0^2}} > \frac{1}{\sqrt{2\pi\sigma_1^2}} e^{\frac{-(y_s-\mu_1)^2}{2\sigma_1^2}}$

Assuming that the distributions have the same variance, we simplify to:

$(y_s - \mu_0)^2 < (y_s - \mu_1)^2$

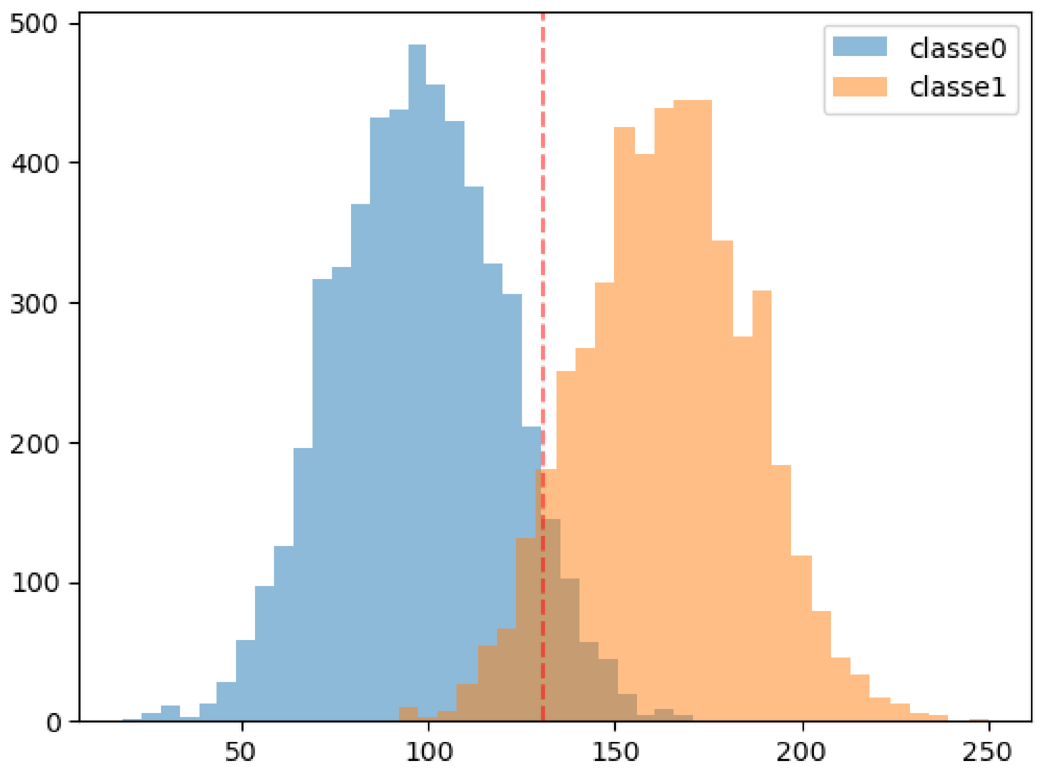
Ultimately, the threshold is determined by:

$y_s < \frac{\mu_0+\mu_1}{2}$

[ ]:
```python
# Set the threshold for classification in the ML sense and display the image
threshold = (m0 + m1)/2
im_bin = im_obs.copy()

affiche(im_bin > threshold)

plt.hist(crop_classe0.ravel(), bins=30, alpha=0.5, label="classe0")
plt.hist(crop_classe1.ravel(), bins=30, alpha=0.5, label="classe1")
plt.axvline(threshold, linestyle="--", color="r", alpha=0.5)
plt.legend()
plt.show()
```



1.1.5 Q3

From the results found for $P(Y_s = y_s|X_s)$, write the likelihood energy (data attachment):

$$U_{attdo} = \sum_s -ln(P(Y_s = y_s|X_s = x_s))$$

1.1.6 A3

As already demonstrated, we can express the Gaussian distribution as follows:

$P(Y_s|X_s) = \frac{1}{\sqrt{2\pi\sigma^2_{x_s}}} e^{\frac{-(y_s-\mu_{x_s})^2}{2\sigma^2_{x_s}}}$

Taking the negative natural logarithm of the probability, we obtain:

$-\ln P(Y_s|X_s) = \frac{(y_s-\mu_{x_s})^2}{2\sigma^2_{x_s}} + \ln(\sqrt{2\pi\sigma^2_{x_s}})$

Assuming that the distributions share the same variance, the data attachment energy is expressed as:

$U_{attdo} = \sum_s (\frac{(y_s-\mu_{x_s})^2}{2\sigma^2} + \ln(\sqrt{2\pi\sigma^2}))$

1.2 2 Ising model for regularisation and energies

To improve the thresholding results, it is necessary to introduce a regularisation (global prior model).

1.2.1 Q4

Consider the function $\Delta(x_s, x_t) = 0$ if $x_s = x_t$, and $\Delta(x_s, x_t) = 1$ otherwise. Write the second-order clique potential for this Ising model as a function of $\Delta(x_s, x_t)$ where x_s and x_t are the classes of neighbouring pixels s and t in 4-connexity and the regularisation parameter β. This model will be 0 when the two neighbouring pixels are equal and $+\beta$ otherwise. Write the global energy of the whole field and the local conditional energy for a site using the results previously established for the data attachment energy and the regularization energy defined previously.

Reminder: the global energy contains all the cliques in the image, the local conditional energy at a site contains only the cliques that contain .

Tip: the energy is defined to within one additive constant and one multiplicative constant (the minimum of K+K'U is equivalent to the minimum of U). It is better to simplify the writing of the energy as much as possible in order to do the programming afterwards.

1.2.2 A4

We have the prior equal to:

$$V_c(x_s, x_t) = \beta\Delta(x_s, x_t) = \begin{cases} 0 & x_s = x_t \\ \beta & \text{otherwise} \end{cases}$$

We know that $U(x) = \sum\limits_{c \in \mathcal{C}} V_{c=(s,t)}$. Thus:

- *global energy*:

  $\mathcal{U}(x|y) = U_{attdo} + U(x) = \sum\limits_{s} \left(\frac{(y_s - \mu_{x_s})^2}{2\sigma^2} + \ln(\sqrt{2\pi\sigma^2})\right) + \beta \sum\limits_{c \in \mathcal{C}} \Delta(x_s, x_t)$

- *local conditional energy*:

  $\mathcal{U}(x_s|y_s, V_s) = \frac{(y_s - \mu_{x_s})^2}{2\sigma^2} + \ln(\sqrt{2\pi\sigma^2}) + \beta \sum\limits_{4\text{ ppv}} \Delta(x_s, x_t)$

  However, for qualitative purposes, it is sufficient to calculate $\mathcal{U}(x_s|y_s, V_s) = (y_s - \mu_{x_s})^2 + \beta' \sum\limits_{4\text{ ppv}} \Delta(x_s, x_t)$, as the variance is constant.

1.2.3 Q5

Write the local conditional energies for classes 0 and 1 of the central pixel, using the following local neighbourhood configuration: neighbours in states 0, 1, 1, 1, and assuming that the grey level of the pixel is $y_s = 105$, and using the mean and variance values found previously.

1.2.4 A5

- Classe 0:

$$\begin{aligned}\mathcal{U}(0|105,\ 0,1,1,1) &= (105-97)^2 + \beta \cdot (0+1+1+1) \\ &= 64 + 3\beta\end{aligned}$$

- Classe 1:

$$\begin{aligned}\mathcal{U}(1|105,\ 0,1,1,1) &= (105-164)^2 + \beta \cdot (1+0+0+0) \\ &= 3481 + \beta\end{aligned}$$

1.2.5 Q6

In which class will this pixel be put if it is assigned the class that locally minimises energy?

1.2.6 A6

The class depends on the value attributed to β. The chosen class would be the class 0 if: $\mathcal{U}(0|105,\ 0,1,1,1) < \mathcal{U}(1|105,\ 0,1,1,1)$. Thus: \$64 + 3 < 3481 +   < 1708.5 \$. Consequently, for $\beta > 1708.5$ the class would be the class 1.

1.2.7 Q7

Considering the ***global*** energy of the field, what is the solution x when β is 0 ?

1.2.8 A7

When β is equal to zero, we multiply the regularization term by zero, which means no regularization. Therefore, we are in the same scenario as in Q2, where we assume that we do not use a Markovian model on X and classify a pixel based solely on its gray level.

1.2.9 Q8

Considering the ***global*** energy of the field, what is the solution x when β is $+\infty$?

1.2.10 A8

When β is equal to ∞, we heavily favor regularization, to the extent that the data attachment term becomes negligible. Consequently, the result will be equivalent to optimizing according to $P(x = \lambda|V_s)$.

1.2.11 Q9

How will the solution vary when β increases ? Comment on the interest of this Markovian model.

1.2.12 A9

As β increases, we favor regularization more in the result. As seen in Q2, if we do not use regularization, we obtain results with a lot of misclassification, especially when the image is degraded (e.g., by noise).

Therefore, the purpose of this Markovian model is to improve the classification result by striking a balance between the data attachment term and regularization based on local neighborhood.

1.3 3. Optimisation by ICM algorithm

We will optimise the global energy defined above, using the ICM (Iterated Conditional Modes) algorithm which consists of minimising the local conditional energy of the pixels one after the other, starting from a good initialisation of the classes. This algorithm converges to a local minimum but is very fast.

Complete the **iter_icm** function to program the ICM, taking into account the data attachment term you have learned.

1.3.1 ICM function

Using what you did for the Gibbs sampler, complete the following function to perform one iteration of the Iterated Conditional Modes algorithm (one pass over all pixels in the image).

```
[ ]: def iter_icm(im_bin, im_toclass, beta_reg, m0, m1):
         for i in range(im_bin.shape[0]):
             for j in range(im_bin.shape[1]):
                 i1 = (i - 1) % im_bin.shape[0]
                 i2 = (i + 1) % im_bin.shape[0]
                 j1 = (j - 1) % im_bin.shape[1]
                 j2 = (j + 1) % im_bin.shape[1]

                 # energy computation if the pixel is put in class 0
                 U0 = (
                     # (im_toclass[i, j] - m0) ** 2 / (2 * var0) + np.log(var0)
                     (im_toclass[i, j] - m0) ** 2  # simplified calcultations
                     + beta_reg
                     * (im_bin[i, j1] + im_bin[i, j2] + im_bin[i1, j] + im_bin[i2,␣
       ↪j])
                 )

                 # energy computation if the pixel is put in class 1
                 U1 = (
                     # (im_toclass[i, j] - m1) ** 2 / (2 * var1) + np.log(var1)
                     (im_toclass[i, j] - m1) ** 2  # simplified calculations
                     + beta_reg
                     * (4 - (im_bin[i, j1] + im_bin[i, j2] + im_bin[i1, j] +␣
       ↪im_bin[i2, j]))
                 )
```

```
            if U0 < U1:
                im_bin[i, j] = 0
            else:
                im_bin[i, j] = 1

    return im_bin
```

1.3.2 Q10

What do you suggest to have a good initialization of the solution? Justify your answer.

1.3.3 A10

We can use the resulting image from the classification when we consider classifying a pixel based solely on its gray level. By doing this, we initialize the algorithm with an image where the majority of terms are already optimized. The main factor that will change is the regularization.

With a well-chosen initialization, the result will converge to the global minimum.

Implement the ICM and study the influence of β.

[ ]:
```
# ICM algorithm to be implemented
# define the value of beta_reg to have a "good" regularization
beta_reg = 1.2e3
# initialise the binary image of the classes
im_bin = np.array(im_obs > threshold, dtype=np.float64)

# affiche(im_bin)
# program a loop in which we call iter_icm
# and display the class image as iterations are performed
for n in range(50):
    im_bin = iter_icm(im_bin, im_obs, beta_reg, m0, m1)

affiche(im_bin)
```

1.3.4 Q12

With what value of β do you get a good solution (i.e. the closest to the given "ideal" image **IoriginaleBW.png**). Compare this result with the result of the optimal thresholding.

1.3.5 A12

β equal to $1.2 \cdot 10^3$ gives us a result close to the ideal image.

You can compare your result with the original image to find the right regularisation parameter.

[ ]:
```
# use np.abs to calculate the absolute value of the difference
# between the original binary image and the ICM result
# note: im_bin is coded between 0 and 1 and im_ori between 0 and 255
```

```
ima_diff = 1 - np.abs(im_bin * 255 - im_ori) // 255

affiche(ima_diff)

print(
    f"Nombre de pixels différents entre l'image d'origine et l'image obtenue␣
  ↪par ICM (initialisation avec seuil): {ima_diff.size-np.
  ↪count_nonzero(ima_diff)}"
)
```

```
Nombre de pixels différents entre l'image d'origine et l'image obtenue par ICM
(initialisation avec seuil): 1523
```

1.3.6 Q13

Try with other initialisations (with a constant image, with a random image). Comment on their influence.

1.3.7 A13

[ ]:
```
im_init1 = np.ones(im_ori.shape)
im_init2 = np.random.normal(loc=0.5, scale=1, size=im_ori.shape)

for n in range(50):
    im_init1 = iter_icm(im_init1, im_obs, beta_reg, m0, m1)
    im_init2 = iter_icm(im_init2, im_obs, beta_reg, m0, m1)

affiche(im_init1)
affiche(im_init2)
```

[ ]:
```
ima_diff_init1 = 1 - np.abs(im_init1 * 255 - im_ori) // 255
ima_diff_init2 = 1 - np.abs(im_init2 * 255 - im_ori) // 255

affiche(ima_diff_init1)
affiche(ima_diff_init2)

print(
    f"Nombre de pixels différents entre l'image d'origine et l'image obtenue␣
  ↪par ICM (initialisation avec image constante): {ima_diff_init1.size-np.
  ↪count_nonzero(ima_diff_init1)}"
)

print(
    f"Nombre de pixels différents entre l'image d'origine et l'image obtenue␣
  ↪par ICM (initialisation avec image alératoire): {ima_diff_init2.size-np.
  ↪count_nonzero(ima_diff_init2)}"
)
```

```
Nombre de pixels différents entre l'image d'origine et l'image obtenue par ICM
(initialisation avec image constante): 2422
Nombre de pixels différents entre l'image d'origine et l'image obtenue par ICM
(initialisation avec image alératoire): 1587
```

1.3.8 Simulated annealing

Program the **echan_r** function of the simulated annealing which allows to update an image by sampling with the Gibbs distribution a posteriori with a fixed temperature T.

```
[ ]: def echan_r(im_bin, im_toclass, beta_reg, m0, m1, T):
         for i in range(im_bin.shape[0]):
             for j in range(im_bin.shape[1]):
                 i1 = (i - 1) % im_bin.shape[0]
                 i2 = (i + 1) % im_bin.shape[0]
                 j1 = (j - 1) % im_bin.shape[1]
                 j2 = (j + 1) % im_bin.shape[1]

                 U0 = (
                     # (im_toclass[i, j] - m0) ** 2 / (2 * var0) + np.log(var0)
                     (im_toclass[i, j] - m0) ** 2  # simplified calcultaions
                     + beta_reg
                     * (im_bin[i, j1] + im_bin[i, j2] + im_bin[i1, j] + im_bin[i2,␣
  ↪j])
                 )
                 p0 = math.exp(-U0 / T)

                 U1 = (
                     # (im_toclass[i, j] - m1) ** 2 / (2 * var1) + np.log(var1)
                     (im_toclass[i, j] - m1) ** 2  # simplified calculations
                     + beta_reg
                     * (4 - (im_bin[i, j1] + im_bin[i, j2] + im_bin[i1, j] +␣
  ↪im_bin[i2, j]))
                 )
                 p1 = math.exp(-U1 / T)

                 if p0 + p1 != 0.0:
                     if random.uniform(0, 1) < p0 / (p0 + p1):
                         im_bin[i, j] = 0
                     else:
                         im_bin[i, j] = 1

         return im_bin
```

Call the function echan_r iteratively, decreasing the temperature after each update slowly enough.

```
[ ]: # Program the simulated annealing
     # temperature initialization
     # initial temperature
     T = 3e3

     # initialization of the binary image
     im_bin = np.array(im_obs > threshold, dtype=np.float64)

     # make a loop by calling the function that does a Gibbs sampling at T
     while T > 0.1:
         # print(n)
         im_bin = echan_r(im_bin, im_obs, beta_reg, m0, m1, T)
         T *= 0.98

     affiche(im_bin)
     print(T)
```

```
0.09854757311602041
```

```
[ ]: im_diff_annealed = 1 - np.abs(im_bin * 255 - im_ori) // 255

affiche(im_diff_annealed)

print(
    f"Nombre de pixels différents entre l'image d'origine et celle obteneue␣
  ↪après recuit simulé : {im_diff_annealed.size-np.
  ↪count_nonzero(im_diff_annealed)}"
)
```

```
Nombre de pixels différents entre l'image d'origine et celle obteneue après
recuit simulé : 1450
```

1.3.9 Q14

Compare the results obtained by the Iterated Conditional Modes algorithm and by simulated annealing. Do you find the results of the course ?

1.3.10 A14

The results obtained by the Iterated Conditional Modes (ICM) algorithm and simulated annealing are similar and consistent with the course results.

A notable observation is that for the simulated annealing method, we need to start with a sufficiently high temperature and decrease it very slowly, almost to zero. It is important to notice, however, that the execution time is significantly higher (due to the greater number of calculations done, as the decreasing step of the temperature gets exponentially smaller, with each iteration recalculating the whole image), and the yielded result is relatively similar to the ICM method with a good initialization.

Therefore, for this relatively simple example, it is more advantageous to use ICM with a good initialization, providing a quicker and satisfactory result.