# INFERENCE AND REPRESENTATION: HOMEWORK 3

STUDENT: MARGARITA BOYARSKAYA

1. *Suppose that we wish to simulate random draws from the joint PMF of $Y$ for a particular known value of $\beta$. Explain how we can do this using Gibbs sampling, cycling through the pixels one by one in a fixed order.*

The Gibbs sampler sequentially samples from the full conditional distributions. For a given $\beta$, the decomposition of the joint posterior distribution into full conditional for the Ising model is:

$$\pi(y_i|y_{-i}) \propto exp(\beta \sum_{j \sim i} I_{[y_i=y_j]}).$$

Let us denote $n_i^b = \sum_{i \sim j} I_{[x_j=1]}$ the number of neighboring pixels of $i$ colored black, and $n_i^w = \sum_{i \sim j} I_{[x_j=0]}$ – the number of white colored neighbors. Then:

$$\pi(x_i = 1|x_{-i}) = \frac{exp(\beta n_i^b)}{exp(\beta n_i^b) + exp(\beta n_i^w)}.$$

The Gibbs sampler simulation for the Ising model can be executed as shown in Algorithm 1 below:

**Algorithm 1** Gibbs sampler for the Ising model, in $N$ iterations

```
procedure GIBBS(n, beta)
    Y = initialize()                    % Y can be filled with values 0 and 1 randomly
    for n in range(N) do
        for i in L do                   % Here we are updating the sites in simple numerical order. A line can be added below to call a more specific update function (e.g. j=Update(i)...)
            p = exp(βn_i^b)/(exp(βn_i^b) + exp(βn_i^w))
            Y[i] = bernoulli(p)
        end for
    end for
    return Y
end procedure
```

For any initial configuration chosen, the following statement holds:

$$Y^n \xrightarrow{n \to \infty} Y_\beta,$$

where $n$ is the number of iterations, $Y_\beta$ is a realization of the Ising model.

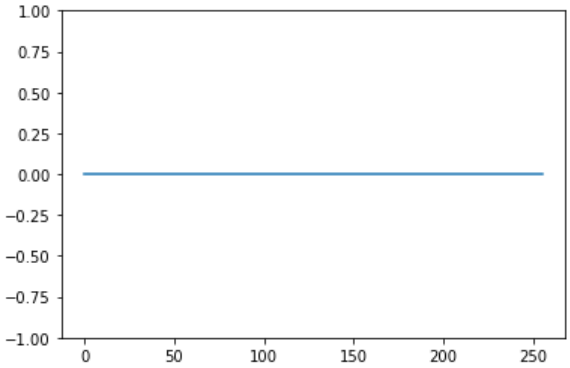
2. *Sum-product algorithm, adapted to Python.*

The code that implements the solutions for problems 2.$a$ and 2.$b$ is included in the attached files:
-*make_debug_graph.py* file contains the methods **SetMsg(g,a,b,value)**, **SumProd(g,f,n)**, **GetBeliefs**.
-*abcdefg.py* file contains code that implements the solution. Commentary accompanies both the code and its execution in the console.
The results obtained are consistent with the computation realized with the method of *fglib* package .

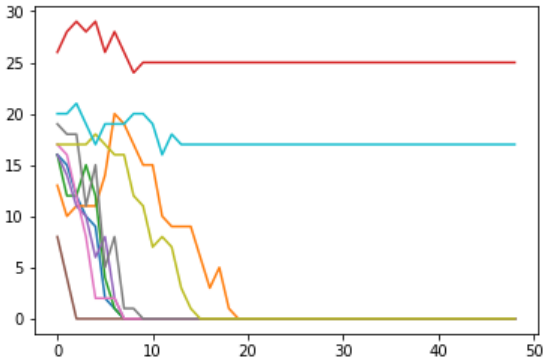
3. *LDPC algorithm, adapted to Python.*

All of the methods used in exercises $3.a - 3.f$ are defined in the file *funcs.py* in the attachment. All of the executive lines of code are marked as per the exercise in the file *abcdefg.py*, together with the commentary. Attached below are the graphics and image realizations for each exercise.

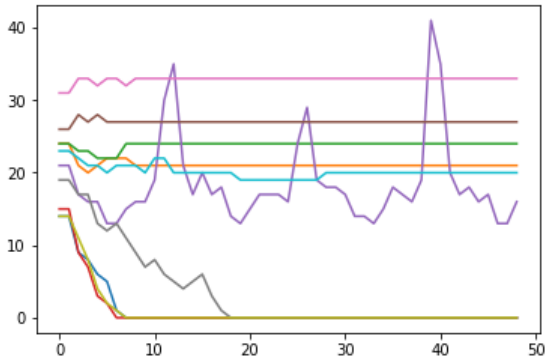
b) Estimated posterior probability that each message bit is equal to one:



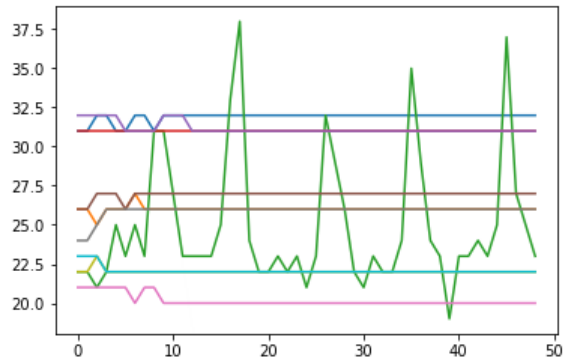

c) Ten curves showing Hamming distance versus iteration for each Monte Carlo trial, $\epsilon = 0.06$:



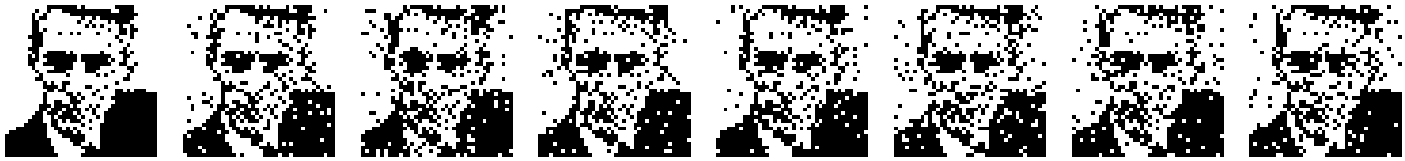
d) Ten curves showing Hamming distance versus iteration for each Monte Carlo trial, $\epsilon = 0.08$:




Ten curves showing Hamming distance versus iteration for each Monte Carlo trial, $\epsilon = 0.1$:




e) For this problem, I chose the likeness of Claude Shannon as a test image. Below are the plots of reconstructed images obtained after decoding the LDPC code computed in $i$ iterations with $\epsilon = 0.06$:



Original. $i = 1$ $i = 2$ $i = 3$ $i = 5$ $i = 10$ $i = 20$ $i = 30$

f) Reconstructed images for $i$ iterations, $\epsilon = 0.1$:

Original. $i = 1$ $i = 2$ $i = 3$ $i = 5$ $i = 10$ $i = 20$ $i = 30$

It is evident that a bigger value of $\epsilon$ renders noisier Shannons, in accordance with expectation.

4. *Implement the Chow-Liu algorithm.*

The code is provided in the attached file titled *chow-liu.py*. The results, formatted as per the requirements, are contained in the *output.txt* of the *data* subfolder.