MÁSTER INTERUNIVERSITARIO EN CIBERSEGURIDAD

Privacy and Anonymity
Homework #1: Filtering Encrypted Images

In this assignment you will follow the Learning With Errors (LWE) paradigm to filter encrypted images. There are two parties involved in the process: **Alice** who has a private image that she wants to filter at a server provided by an external company, and **Kernels de Galicia** (KDG), which owns a proprietary kernel to filter the image. Since Alice doesn't want KDG to learn the image contents, she encrypts it and sends it to KDG that operates over the ciphertexts and returns an encrypted filtered image that Alice should be able to decrypt with her secret key. You must implement the whole process making sure that privacy is preserved (i.e., that KDG doesn't use any other information from Alice than the ciphertexts and the common parameters).
Next, there is a description of the common parameters:

- The length of the **unique** secret key $\mathbf{s}$ is $n = 128$. The elements of the secret key are chosen independently and with uniform probability from the set $\{-1, 0, 1\}$.
- The modulus $q = 2^{24}$.
- Parameter $\Delta = 2^{10}$.
- All necessary vectors $\mathbf{a}$ (and you will need a very large quantity of them) are generated taking uniform samples from $\mathbb{Z}_q^n$
- Parameter α is to be specified later.

For simplicity, the image provided by Alice will be in grayscale, this means that each pixel contains only one value of luminance which is an integer in $[0, 255]$. Although your implementation should be general enough to be able to use any square kernel, the particular one that we want to use is:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

which is a *high-pass filter* that serves to extract *edges* from Alice's image.

We briefly explain the filtering process in the clear (that is, with no encrypted signals).The operation is also called a *convolution*. You can find a detailed description in https://en.wikipedia.org/wiki/Kernel_(image_processing), but there are thousands of websites and videos that explain it. Of course, there are functions to do convolutions, but you'll need to understand the process because later you'll have to do convolutions with encrypted data.

A sample code that performs the filtering, `Filtered_clear.py` is provided for reference, to be later adapted by you to operate with encrypted data. It uses a grayscale image `nopadron.png` given as example.[1] In order to keep the size of the input image after the convolution, it is customary to extend the input on its borders. In the reference implementation, this is done by *zero-padding*, that is, attaching zeros on the borders of the image; specifically, if the size of the kernel is $d \times d$, and assuming that d is odd, then $d' \doteq (d-1)/2$ all-zero columns are attached to the left and to the right of the image and d'

[1] No Padron peppers were spoiled in the making of this Homework.

all-zero rows to the top and to the bottom (see reference code). Let's call the zero-padded image P, then the (i,j) pixel of the filtered image is calculated as

$$Q(i,j) = \sum_{u=-d'}^{d'} \sum_{v=-d'}^{d'} K(-u,-v)P(i+u,j+v) \tag{1}$$

where $K(u,v), u,v = -d', \cdots, d'$ are the elements of the kernel.[2] In practice, the kernel is stored in a matrix, so its indices go from 0 to $d-1$. See the reference implementation `Filtered_clear.py` to learn how this can be implemented.

Encryption proceeds straightforwardly: Alice generates a pair $(\mathbf{a}, b)$ **for each pixel** of the zero-padded image. Let us denote this pair for the (i,j)th pixel as $(\mathbf{a}(i,j), b(i,j))$. Notice that $b(i,j)$ is generated as we already know:

$$b(i,j) = \mathbf{s}^T\mathbf{a}(i,j) + \Delta \cdot P(i,j) + e(i,j) \bmod q$$

where $e(i,j)$ is one error sample generated as in Lab assignment #4, Question **Q2**, and $P(i,j)$ is the (i,j)th pixel of the zero-padded image.

Decryption of an image Q, requires an encryption for every pixel, say $(\mathbf{a}'(i,j), b'(i,j))$, for all (i,j), and then, having the secret vector $\mathbf{s}$, doing for each pixel (i,j):

$$Q(i,j) = \lfloor (b'(i,j) - \mathbf{s}^T\mathbf{a}'(i,j) \bmod q)/\Delta \rceil$$

At this stage, it is avised that you try with a small value of α (e.g., α = 5.0e-07) to encrypt the `nopadron.png` image and decrypt it to verify that you get the original image back.

It's time for you to implement what KDG would do: given the pairs $(\mathbf{a}(i,j), b(i,j))$ for all (i,j), and knowing the parameters (see list above) and the kernel, implement the filtering. Notice that the convolution in (1) is a linear operation (i.e. sums and products), so you should be able to produce the pairs $(\mathbf{a}_f(i,j), b_f(i,j))$ corresponding to the pixels of the filtered image (I've put the subindex f here to denote 'filtered') by doing linear operations on the encryptions.

Now, given the pairs $(\mathbf{a}_f(i,j), b_f(i,j))$ for each of the output pixels, you can emulate the role of Alice: she should be able to decrypt the result using the secret key vector, and if everything goes well, you should get the same result as if the filtering were done in the clear. Besides plotting the result, measure the square error (i.e., squared ℓ_2-norm) between the two results (in the clear and after encryption-filtering-decryption); if α is sufficiently small, the error should be zero.

Once you implemented the process, repeat it for the following values of α and report the results: 5e-07, 5e-06, 5e-05, 5e-04, 5e-03. Discuss the results.

Tips:

- The coefficients of the kernel are rather small, so you can assume that they're *small constants*, and therefore there is no need to use the gadget decomposition here.
- Since a grayscale image can be seen as a 2D matrix, you can store the $\mathbf{a}$ vectors corresponding to the encryptions of every pixel in a 3D matrix (adding a third dimension to store the n elements of every vector). In the case of the b's, since they're scalar, you can store them in a matrix of the same size as the image, i.e. one per pixel.
- Try to use matrix operations as much as possible. If done properly, you don't need to have more loops than those going over the pixels, that are already in `Filtered_clear.py`.
- Try to make your `errorgen` function able to generate a matrix of errors, so you just need to call it once.
- Make sure that you work with integers and not numbers in uint8 format (this is what by default the `imread` function ouptuts). See reference code.

[2] Notice that (1) requires $K(-u,-v)$ which can be obtained from $K(u,v)$ by flipping its rows and columns.