ECE 285 – Project D
Non-Local Regularization

Written by Charles Deledalle on May 31, 2019.

You will have to submit a notebook `projectD.ipynb` and the package `imagetools/projectD.py`. Organize your notebook with headings (following the numbering of the questions). For writing questions, answer directly in your notebook in markdown cells. For each section, it is indicated in brackets how much it contributes to the grade.

This project focuses on image restoration with non-local means Before starting this project you will need to have gone through all assignments. Functions developed in this project will complete the `imagetools` package. We will be using the following assets

- `assets/jockeys.png`
- `assets/starfish.png`
- `assets/zebra.png`

1 Operators (25%)

We focus on the estimation of a clean image x_0 form its degraded observation y satisfying

$$y = \mathbf{H}x_0 + \varepsilon$$

where ε is a white Gaussian noise component with standard deviation σ, and $\mathbf{H}$ a linear operator. We will consider three types of linear operators: identity (denoising problem), convolution (deblurring problem), and random masking (inpainting problem).

We will need to be able to compute for any images x:

- the application of $\mathbf{H}$ to x: $x \mapsto \mathbf{H}x$,
- the application of its adjoint: $x \mapsto \mathbf{H}^* x$,
- the application of its gram matrix: $x \mapsto \mathbf{H}^*\mathbf{H}x$,
- the resolvent of its gram matrix: $x \mapsto (\mathrm{Id} + \tau\mathbf{H}^*\mathbf{H})^{-1}x$.

A linear operator will be represented by a Python object as an instance of a class that inherits from our homemade abstract class `LinearOperator` defined in `imagetools/provided.py`. Please have a look at the code. Note that `LinearOperator` has a method `norm2` that returns an approximation of the spectral norm of the operator $\|\cdot\|_2$ and `normfro` that returns an approximation of the Frobenius norm $\|\cdot\|_F$. It also has two properties `ishape` and `oshape`, the first one is the shape of the input of the operator, the second is the shape of the output. Any class that inherits from it must implement (at least):

- `__call__(self, x)`
- `adjoint(self, x)`
- `gram(self, x)`
- `gram_resolvent(self, x, tau)`

As an example, we provided `Grad` that reuses functions from the previous assignments to implement each of these methods for the gradient operator. An object can be instantiated as `H = im.Grad((n1, n2, 3))` for the gradient of a RGB image of shape $(n1, n2, 3)$.

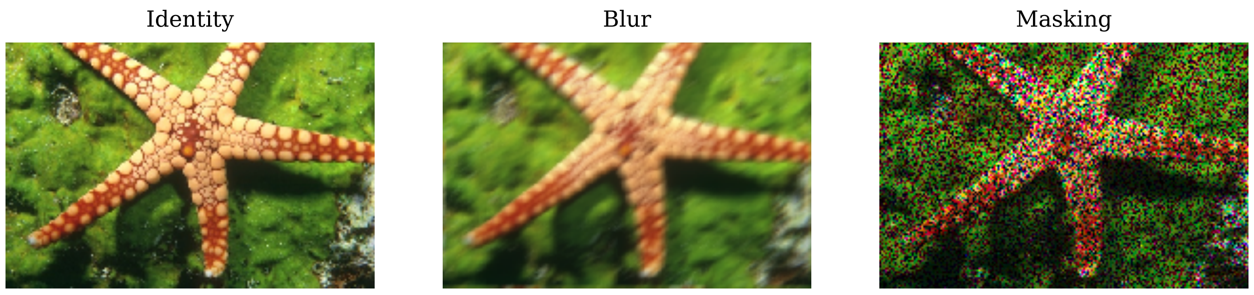
1. In `imagetools/projectD.py`, create a class `Identity` that implements the identity operator $x \mapsto x$. An object can be instantiated as `H = im.Identity(shape)`.

2. Create a class `Convolution` that implements the convolution operator $x \mapsto \nu * x$. An object can be instantiated as `Convolution(shape, nu, separable=None)`. As we will manipulate large convolution kernels ν, all operations should be implemented in the Fourier domain. Note that during this project, we will always consider periodical boundary conditions.

   Hint: reuse functions from the assignments.

3. Create a class `RandomMasking` that implements the linear operator that sets a proportion p of arbitrary pixels to zeros. An object can be instantiated as `H = im.RandomMasking(shape, p)`.

4. In your notebook, load the image `x0 = starfish`. Create a version y for each of the three operators. For the random masking we will consider $p = .4$. For the convolution we will consider the motion kernel ν. Display the result and check that they are consistent with the following ones.



5. For the three linear operators, check that $\langle \mathbf{H}x, y\rangle = \langle x, \mathbf{H}^* y\rangle$ for any arbitrary arrays `x` and `y` of shape `H.ishape` and `H.oshape` respectively (you can generate `x` and `y` randomly).

6. Check also that $(\mathrm{Id} + \tau \mathbf{H}^*\mathbf{H})^{-1}(x + \tau \mathbf{H}^*\mathbf{H}x) = x$ for any arbitrary image `x` of shape `H.ishape`.

2 Sinkhorn-Knopp NL-means filter (20%)

The Non-Local means (NL-means) filter was defined (refer to Chapter 2) as

$$x_{i,j}^{\text{(NL-means)}} = \sum_{k=-s_1}^{s_1} \sum_{l=-s_2}^{s_2} w_{i,j,i+k,j+l} \cdot y_{i+k,j+l} \tag{1}$$

$$\text{where} \quad w_{i_1,j_1,i_2,j_2} = \frac{1}{Z_{i_1,j_1}} \left\{ \begin{array}{ll} \varphi\left(\frac{1}{P}\|\boldsymbol{y}_{i_1,j_1} - \boldsymbol{y}_{i_2,j_2}\|_2^2\right) & \text{if} \quad |i_1 - i_2| \leqslant s_1 \quad \text{and} \quad |j_1 - j_2| \leqslant s_2 \\ 0 & \text{otherwise} \end{array} \right. , \tag{2}$$

where $\boldsymbol{y}_{i,j} = (y_{i+a,j+b})_{-p_1 \leqslant a \leqslant p_1, -p_2 \leqslant b \leqslant p_2}$, and s_1, s_2, p_1, p_2 and φ are as defined in assignment 4. The normalization constant Z_{i_1,j_1} is defined such that $\sum_{i_2,j_2} w_{i_1,j_1,i_2,j_2} = 1$. The weights are said to be row-stochastic. The bi-stochastic NL-means is a variant in which the weights w are replaced by bi-stochastic weights $\tilde{w}$ satisfying also $\sum_{i_1,j_1} \tilde{w}_{i_1,j_1,i_2,j_2} = 1$. A simple iterative procedure to get bi-stochastic weights is called Sinkhorn-Knopp algorithm and reads, for $t > 0$ and $\tilde{w}^0_{i_1,j_1,i_2,j_2} = w_{i_1,j_1,i_2,j_2}$, as

$$\underbrace{\tilde{w}^{t+1}_{i_1,j_1,i_2,j_2} = \frac{\tilde{w}^{t+1/2}_{i_1,j_1,i_2,j_2}}{\sum_{i_2,j_2} \tilde{w}^{t+1/2}_{i_1,j_1,i_2,j_2}}}_{\text{row normalization}} \quad \text{and} \quad \underbrace{\tilde{w}^{t+1/2}_{i_1,j_1,i_2,j_2} = \frac{\tilde{w}^{t}_{i_1,j_1,i_2,j_2}}{\sum_{i_1,j_1} \tilde{w}^{t}_{i_1,j_1,i_2,j_2}}}_{\text{column normalization}} . \tag{3}$$

As $t \to \infty$, we are guaranteed that $\tilde{w}$ is the closest bi-stochastic array from w.

7. Let $W_{i,j,k,l} = w_{i,j,i+k,j+l}$. Show that

$$\sum_{\substack{1\leqslant i_2\leqslant n_1\\1\leqslant j_2\leqslant n_2}} w_{i_1,j_1,i_2,j_2} = \sum_{\substack{-s_1\leqslant k\leqslant s_1\\-s_2\leqslant l\leqslant s_2}} W_{i_1,j_1,k,l} \quad\text{and}\quad \sum_{\substack{1\leqslant i_1\leqslant n_1\\1\leqslant j_1\leqslant n_2}} w_{i_1,j_1,i_2,j_2} = \sum_{\substack{-s_1\leqslant k\leqslant s_1\\-s_2\leqslant l\leqslant s_2}} W_{i_2-k,j_2-l,k,l}\ .$$

8. In `imagetools/projectD.py`, write the function

```
def sinkhorn_knopp(W, s1, s2, T, boundary)
```

that performs T iterations of the Sinkhorn-Knopp algorithm directly on the array W of shape $(n_1, n_2, 1, S)$ stacking for each pixel the collection of weights for all possible shifts in the search window $S = (2s_1 + 1) \times (2s_2 + 1)$. It returns the updated array W.

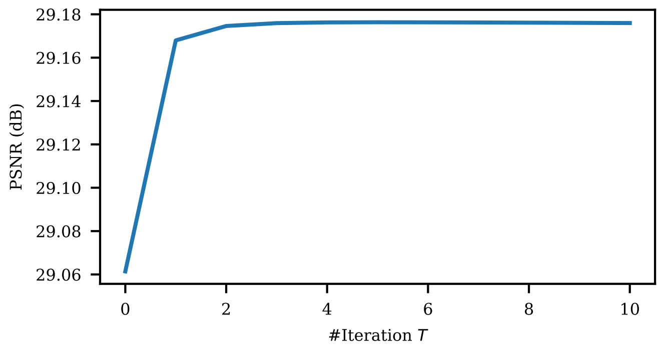
Hint: rely on the results form the previous question and use `im.shift`.

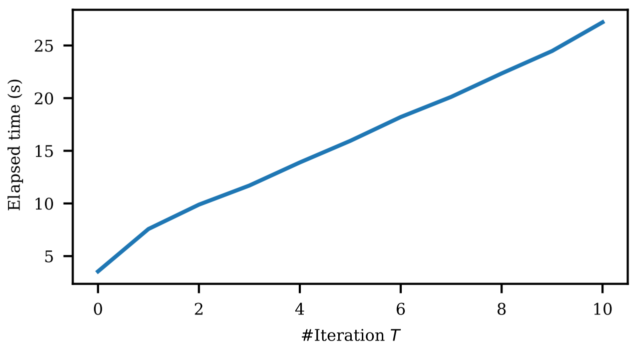
9. Recopy the function `nlmeans` from assignment 4, and add an extra optional argument `T`

```
def nlmeans(y, sig, s1=7, s2=7, p1=None, p2=None, h=1, T=1, boundary='mirror')
```

Keep the general idea of your previous implementation, but decompose the process into three parts: creation of the stack of weights W, application of Sinkhorn-Knopp algorithm, and weighted average.

10. Complete your notebook to test your new variant on the corrupted version `y` of `x0 = zebra` with additive white Gaussian noise of standard deviation $\sigma = 20/255$. Compare the quality (PSNR) and the execution times of bi-stochastic NL-means for different values of T and check that your results are consistent with the following ones:





3 Block-wise NL-means filter (15%)

The block-wise NL-means is another variant of NL-means, where instead of averaging values of pixels with similar patches, similar patches are first averaged together:

$$\boldsymbol{x}^{\text{BNLM}}_{i,j} = \sum_{k=-s_1}^{s_1} \sum_{l=-s_2}^{s_2} W_{i,j,k,l} \cdot \boldsymbol{y}_{i+k,j+l} \tag{4}$$

Next all restored patches are projected into the image domain at their original locations by averaging as

$$x^{\text{BNLM}}_{i,j} = \frac{1}{P} \sum_{a=-p_1}^{p_1} \sum_{b=-p_2}^{p_2} (\boldsymbol{x}^{\text{BNLM}}_{i+a,j+b})_{-a,-b}. \tag{5}$$

11. Plug these two equations together, and show that

$$x^{\text{BNLM}}_{i,j} = \frac{1}{P} \sum_{k=-s_1}^{s_1} \sum_{l=-s_2}^{s_2} \sum_{a=-p_1}^{p_1} \sum_{b=-p_2}^{p_2} W_{i+a,j+b,k,l} \cdot y_{i+k,j+l} \ .$$

Hint: note that $(\boldsymbol{y}_{i,j})_{-a,-b} = y_{i-a,j-b}$.

12. In `imagetools/projectD.py`, add to the function `nlmeans` an extra optional argument `block`

```python
def nlmeans(y, sig, s1=7, s2=7, p1=None, p2=None, h=1, T=1, block=True,
            boundary='mirror')
```

and implement the block-wise NL-means but without adding extra explicit loops (this must be performed after modifying the weights with Sinkhorn-Knopp algorithm).

Hint: you just need to add a single line calling the function `im.convolve`.

13. Complete your notebook to test the block-wise NL-means on y with default parameters, improving the performance by about $.2dB$.

14. Why does the block-wise NL-means show improved performance as compared to the standard NL-means?

4 Smoothed Non-Local means (20%)

A difficulty with NL-means is that some residual noise remains where not enough similar patches are found in the search window. This is known as the rare patch effect.

15. Let $x_0 \in \mathbb{R}$ and $y_i = x_0 + \varepsilon_i \in \mathbb{R}$ for $i = 1, \ldots, N$, where ε_i are independent, $\mathbb{E}[\varepsilon_i] = 0$ and $\text{Var}[\varepsilon_i] = \sigma^2 > 0$. Consider the average

$$x^{\text{AVG}} = \frac{1}{N} \sum_{i=1}^{N} y_i \ .$$

What is its bias and variance as an estimator of x_0? By how much does it reduce the noise variance?

16. Give a statistical interpretation of the rare patch effect.

17. Let $w_i > 0$ and $\sum_i w_i = 1$ be the weights and consider the weighted average

$$x^{\text{WAVG}} = \sum_{i=1}^{N} w_i y_i \ .$$

Assume that all w_i and y_i are independent. What is its bias and variance as an estimator of x_0? By how much does it reduce the noise variance?

18. Add an extra optional argument `return_noise_reduction`

```python
def nlmeans(y, sig, s1=7, s2=7, p1=None, p2=None, h=1, T=1, block=True,
            boundary='mirror', return_noise_reduction=False):
    ...
    if return_noise_reduction:
        return x, noise_reduction
    else:
        return x
```

such that your function returns a $(n_1, n_2, 1)$ array in which each value is the factor of noise variance reduction at that pixel location.

19. Show that if $w_i = \max_{1 \leqslant j \leqslant N} w_j$ for $1 \leqslant i \leqslant K \leqslant N$, then the weighted average reduces the noise by at least a factor of K. Note that $w_i = w_i / \sum_j w_j$ since $\sum_j w_j = 1$.

20. Smoothed NL means reduces the noise variance by at least K by enforcing the K most similar patches to have the same weight as the most similar one. Add an extra optional argument `K`

```
def nlmeans(y, sig, s1=7, s2=7, p1=None, p2=None, h=1, T=1, K=4, block=True,
            boundary='mirror', return_noise_reduction=False)
```

and implement the smoothed NL means of factor K for the case where $K > 0$. This has to be performed after the Sinkhorn-Knopp algorithm but before block reprojection.

Hint: use `np.argsort`, `np.put_along_axis` and `np.take_along_axis`. Don't forget to renormalize!

21. Complete your notebook to test the smoothed NL-means on y with default parameters and compare the quality (PSNR) and the execution times of `nlmeans` with Sinkhorn-Knopp ($T = 1$) or not ($T = 0$), block or not, smoothing ($K = 4$) or not ($K = 0$). For each case display the noise variance reduction map and the residue. Interpret the results. Check that your results are consistent with the following ones.

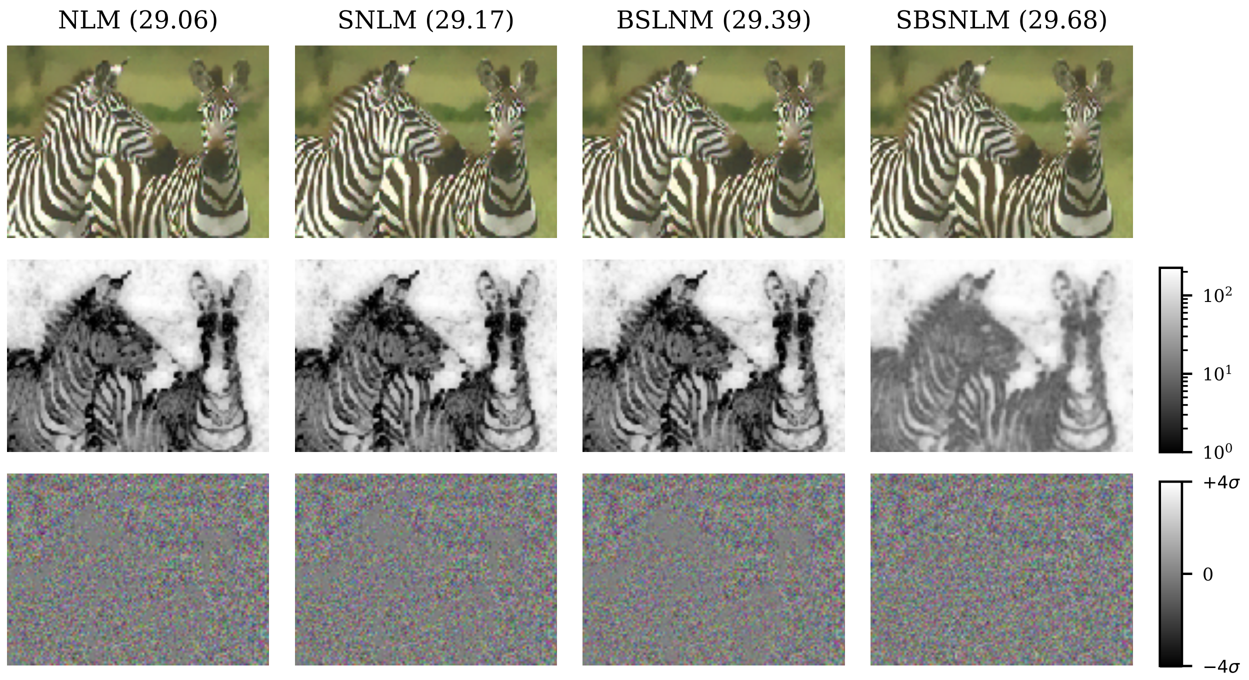


Figure 1: (Top) Denoised images x with NL-means (NLM, 3.8s), Sinkhorn NL-means (SNLM, 7.3s), Block+Sinkhorn NL-means (BSLNM, 9.0s), and Smoothed+Block+Sinkhorn NL-means (SBSNLM, 10.0s). PSNRs are in brackets. (Middle) Noise reduction map (in log scale). (Bottom) Residue $y - x$.

5 Non-Local Regularization with Plug-and-Play ADMM (20%)

Plug-and-play ADMM (Alternating Direction Method of Multipliers) is a technique that can embed a Gaussian denoiser to solve a linear restoration problem of the form: $y = \mathbf{H}x + \varepsilon$. It reads as

$$\begin{aligned} x^{k+1} &= (\mathrm{Id}_n + \gamma \mathbf{H}^* \mathbf{H})^{-1} (\tilde{x}^k + d^k + \gamma \mathbf{H}^* y) \\ \tilde{x}^{k+1} &= \mathrm{denoise}(x^{k+1} - d^k, \sigma, \gamma) \\ d^{k+1} &= d^k - x^{k+1} + \tilde{x}^{k+1} \end{aligned}$$

where $\mathrm{denoise}(y, \sigma, h)$ is a denoiser for Gaussian noise with standard deviation σ and smoothing parameter h (Refer to Chapter 7 for more details). Note that the variables $\tilde{x}$ and d are images of same shape as x.

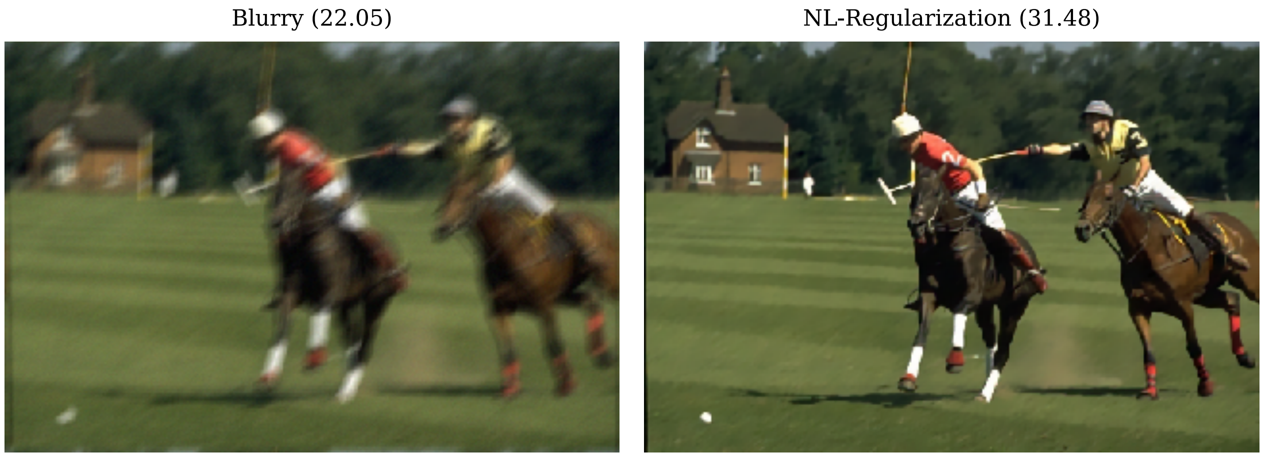
22. In `imagetools/projectD.py`, create the function

```python
def nlmeans_regularization(y, sig, gamma=None, H=None, m=10,
                           s1=7, s2=7, p1=None, p2=None, h=1, K=4, block=True,
                           boundary='mirror')
```

that implements m iterations of Plug-and-play ADMM with NL-means denoiser (or variants). When `gamma is None`, we will consider $\gamma = n_1 n_2 C / \|\mathbf{H}\|_F^2$. We will choose $x^0 = \tilde{x}^0 = \mathbf{H}^* y$ and $d = 0$.

23. In your notebook, load the image `x0 = jockeys` and create a blurry version `y` by defining $\mathbf{H}$ as the motion blur kernel, and add a Gaussian noise of standard deviation $\sigma = 2/255$. Apply your deblurring function on y.

Results may look like the following ones:



24. Repeat the experiment but with a random masking of 40% for $m = 10$ and next $m = 40$.

6 Bonus (+10% max)

- In denoising, for different noise levels and variants, compare your implementation of NL-means with the one of Scikit image: `skimage.restoration.denoise_nl_means`.
- Implement super-resolution.
- Make the algorithm faster by implementing some parts in `cypthon`.
- Implement and discuss further possible improvements.