SCC5830/0251 — Prof. Moacir Ponti

Teaching Assistant: Leo Sampaio Ferraz Ribeiro

Teaching Assistant: Flavio Salles

Assignment 4: image restoration

Code the assignment by yourself. Ask if you need help. Plagiarism is not tolerated.

1 Introduction

1.1 Task

In this assignment you have to implement two image restoration filters. Read the instructions for each step. Use `python 3` and the libraries `numpy`, `imageio` and `scipy` to complete the task.

Follow the instructions carefully:

1. **Load** the degraded image g
2. **Read the parameters** specific to the chosen restoration method F
3. **Apply the method** F as described in Sec. 2
4. **Clip** the image's values between $[0, 255]$ range
5. **Compare** your resulting image against the original image I using Root Mean Squared Error (RMSE)

1.2 Input Parameters

The following parameters will be input to your program in the following order through `stdin`, as usual for run.codes:

1. filename for the reference image f,
2. filename for the degraded image g,
3. type of filter to perform image restoration $F = 1$ (denoising) or 2 (constrained l.s.),
4. parameter $\gamma \in [0, 1]$ that is used by either filter,
5. then following parameters depend on the chosen method:
   - i **Method 1 (Adaptive Denoising)**:
     (a) row with coordinates of a flat rectangle in the image, defined as $[i_1, i_2, j_1, j_2]$
     (b) size of the filter $n \in [3, 5, 7, 9, 11]$

(c) denoising mode: "average" or "robust"

ii **Method 2 (Constrained Least Squares)**:

(a) size of the gaussian filter $k \in [3, 5, 7, 9, 11]$

(b) σ for the gaussian filter

6. **Restore image** g with filter F producing the restored image $\hat{f}$.
7. **Compute** and print the RMSE comparing $\hat{f}$ with the reference image f,

2 Image Restoration Filters

Your program should apply the restoration filter selected by the parameter F, either (1) Adaptive Denoising Filtering or (2) Constrained Least Squares Filtering. The following sections detail how each should work.

2.1 Adaptive Denoising Filtering

The first step for any restoration process is to identify the source of degradation before applying an appropriate fix. For this homework we have done this for you and will tell you which method to use, but we will also explain the idea anyway because learning is living. This section regards the method of Adaptive Denoising Filtering.

In the real world, we'd look at an image and identify that **it's degradation is uniform** (the image is equally degraded on its entirety) and that it **changes the statistics of the intensity distribution on each image region**. One such noise is known as Salt&Pepper, caused by sensor failures on a camera that make random pixels completely white or black. And one of the solutions for this family of degradations is Adaptive Denoising Filtering.

This method applies a common mean/median filter, but with a twist. The strength of the filter is weighted by how close the dispersion (i.e. deviation from mean or median) in a neighbourhood is to the estimated noise dispersion in the image. Let's break this down then. Given only an image and no previous knowledge it's very hard to have a good estimate of the distribution of noise that caused the degradation; one good solution is to crop a flat (no borders or change in colour) region of an image and estimate the noise distribution from that.

For this implementation you should only use functions you wrote or from `numpy`. For each pixel in coordinates $g(i, j)$ you should apply the following adaptive filter:

$$\hat{f}(\mathbf{x}) = g(i,j) - \gamma \frac{\text{disp}_\eta}{\text{disp}_L(i,j)} \left[g(i,j) - \text{centr}_L(i,j) \right]$$

where γ controls the filter's strength, $disp_\eta$ is the estimated noise dispersion, $disp_L$ is the local dispersion (computed over the pixels in the $n \times n$ neighbourhood) and $centr_L$ is the centrality measure for the $n \times n$ neighbourhood.

To compute the estimated noise dispersion $disp_\eta$, you should crop the image according to *the coordinates we provided as input* $[i_1, i_2, j_1, j_2]$ and compute the dispersion on top of it. These coordinates should crop a flat region of the image. The local dispersion $disp_L$ is, again, computed over the pixels in the $n \times n$ neighbourhood, as is the centrality $centr_L$.

The dispersion and centrality metrics are different depending on the mode (one of the input parameters):

(a) `'average'`: use the arithmetic mean as centrality measure and the standard deviation as the dispersion measure.

(b) `'robust'`: use the median as centrality measure, and the interquartile range ($IQR = Q_3 - Q_1$) as the dispersion measure.

Since this is a neighbourhood-based filtering operation, you should pad your image *before* applying the filter. Use "symmetric" passing for this.[1]

Things to do so that it doesn't go wrong: If the estimated dispersion measure computed for an image is $disp_\eta = 0$, manually set it to $disp_\eta = 1$ to avoid having the filter doing nothing for the whole image. If any local dispersion is $disp_L = 0$, then set it to $disp_L = disp_\eta$ to avoid division by zero.

2.2 Constrained Least Squares Filtering

Now, picture a picture that has been **degraded by convolution** and you have a way of identifying which filter was used to cause the degradation. In this context you recovery arsenal can count on a method for approximating the inverse operation of this degrading convolution: *Constrained Least Squares Filtering.*

This method considers the problem of a convolutional degradation plus some random noise. We know that if we know the degradation was caused by a convolution, inverting the degradation would be a point-wise division between image and filter in the frequency domain:

$$F(u) = \frac{G(u)}{H(u)}$$

where F and G are the original and degraded images in the frequency domain and H is the degrading filter in the frequency domain.

This would be great if not for the inherent presence of random noise common to all real images (caused by sensors, compression, corruption). Reformulating this inverse filter accounting for noise however leads to

$$\hat{F}(u) = \frac{G(u)}{H(u)} + \frac{N(u)}{H(u)}$$

[1] Notice how most other choices of padding here would disturb the dispersion of the border neighbourhoods and create artifacts.

where N is our random noise and $\hat{F}$ our original image in the frequency domain without the noise. This looks feasible, except that we know noise has, in general, high frequencies and blurring filters have generally low frequencies, setting up the second fraction to increase the noise considerably.

The solution then is to approximate the perfect solution by applying some kind of suppression on the noise. The Constrained Least Squares filter implements that using the inverse of a Laplacian filter:

$$p \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

also in the frequency domain. This is done as follows:

$$\hat{F}(\mathbf{u}) = \left[\frac{\overline{H}(\mathbf{u})}{|H(\mathbf{u})|^2 + \gamma |P(\mathbf{u})|^2} \right] \times G(\mathbf{u}),$$

where H is the filter, $\hat{F}$ the approximate restored image, G our input image and P the Laplacian operator, all of them of course in the frequency domain. $\overline{H}$ is the complex conjugate of the filter H in the frequency domain.

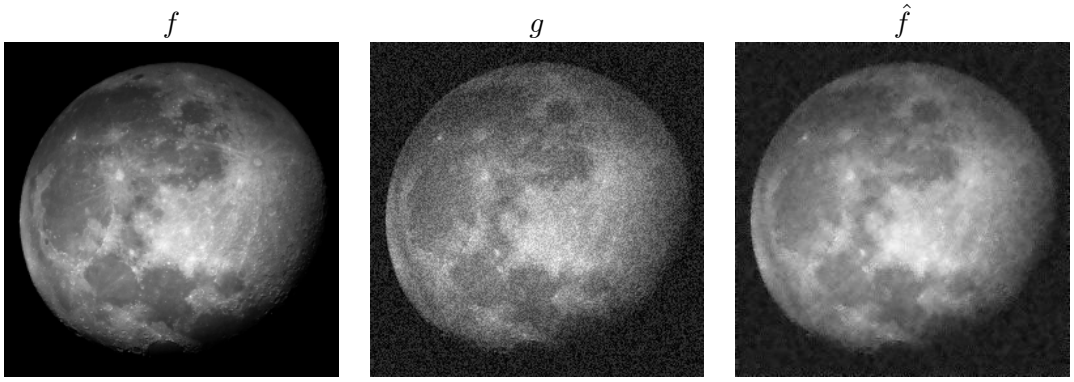
For this assignment, **we know the filter that caused the degradation: a gaussian filter**. We will provide as input the size k and the σ value for the gaussian gilter, and you can use the following code to generate it:

```
def gaussian_filter(k=3, sigma=1.0):
    arx = np.arange((-k // 2) + 1.0, (k // 2) + 1.0)
    x, y = np.meshgrid(arx, arx)
    filt = np.exp( -(1/2)*(np.square(x) + np.square(y))/np.square(sigma) )
    return filt/np.sum(filt)
```

Remember to do computations in the correct domain: **(1)** Use the discrete fourier transform (see `scipy.rfft2`, you should use it) on the image g, filter h and Laplacian operator p; **(2)** Do the filtering process in the frequency domain; **(3)** Use the inverse discrete fourier transform (see `scipy.irfft2`) to have the image $\hat{f}$ back to the spatial domain.

2.3 Clipping Instead of Normalizing

After performing the restoration steps your image will be a bit out of the $[0, 255]$. This time however we will use clipping instead of min-max normalization. This will help mitigate some of the noisy artifacts created by the imperfect constrained least squares filtering. Try to plot your resulting image before clipping to see how those spurious artifacts make flat regions less... flat.



$F = 1$, $\gamma = 0.8$, $k = 5$, *mode* ="robust"

$F = 2$, $\gamma = 0.00005$, $k = 5$, $\sigma = 1.0$

Figure 1: Examples of reference, degraded and restored images for the two image restoration methods

3 Comparing against reference

Your program must compare the restored image against reference f. This comparison must use the root mean squared error (RMSE). Print this error in the screen, rounding to 4 decimal places.

$$RMSE = \sqrt{\frac{\sum_i \sum_j (g(i,j) - f(i,j))^2}{n \cdot m}}$$

4 Submission

Submit your source code to run.codes (only the `.py` file).

1. **Comment your code**. Use a header with name, USP number, course code, year/semestre and the title of the assignment. If you are in a group, include all the info for both and submit the assignment under both of your accounts. A penalty on the evaluation will be applied if your code is missing the header and comments.

2. **Organize your code in programming functions**. Use one function for each method.