2. Spatial filtering

Abstract
We will learn how to inject different types and amounts of image noise, and to remove it or reduce it through spatial-domain filtering. In practice, introducing noise on purpose can serve two purposes: to learn about these noises, and to study the effect of different types of filtering in controlled conditions. We will look at different filters and their pros and cons. It is essential that you understand which filter is more adequate for each type of noise.

Keywords
Salt-and-pepper noise • Gaussian noise • Median, mean, and Gaussian filters

Contents

1. Generating noisy images

Salt-and-pepper noise

We will produce salt-and-pepper noise with basic NumPy operations. As in other cases where code is provided, it is very important that you understand what each part of the code does and for which purpose, so that you can learn to write or modify code for similar situations.

Gaussian noise

To produce Gaussian noise, we use the module numpy.random. Notice that we do not use a function that directly applies the noise to an image, but we first generate Gaussian noise in an array of the same size of the original image, and then add this result to the image. ▷ E_4 Try also with Gaussian noise with σ twice the maximum value used in the original code.

2. Image filtering

To assess the practical value of a filter like those studied here, we need to consider, at least: (1) its effectiveness in denoising; (2) its behaviour preserving object edges; and (3) its computational cost as a function of the mask (filter) size. ▷ E_1

Mean filter

To apply the mean filter, we will procede in a general way: the mask of the given size is first created, and then a convolution is applied. ▷ E_2 Make sure you understand how the mask is defined in the provided code. For convolution, the module scipy.ndimage.filters is used.

Note that the given code applies the mean filter to images with salt-and-peper noise. Discuss how effective is this filter for that type of noise.

Gaussian filter

For the Gaussian filter, we use the module scipy.ndimage.filters. As you can see, this filter is applied to images with Gaussian noise. Does this filter work for this noise? Experiment with several values of the standard deviation of the noise and several values of the standard deviation of the Gaussian filter. Please, pay attention to the fact that Gaussian functions are used both for the noise and for the filter, but with different meanings; **do not get confused**, and make sure you understand what the Gaussian function represent in each case.

To better appreciate the filtering result, you can zoom in on a relatively small area of the image (e.g. the nose, an eye, or a part of the hat). ▷ E_7

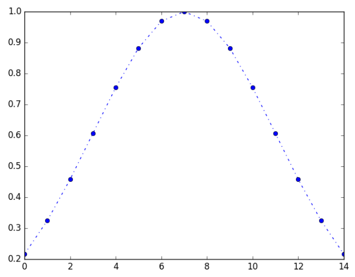
Median filter

Now we use the module scipy.signal. Observe the effect of the median filter on salt-and-pepper noise. With respect to the mean filter, is the median filter more or less effective? Why? Does the same happen if the filter size is "very high"?

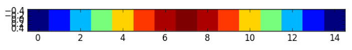
3. Exercises

1. Prepare a double-entry table with types of noise and types of filters. In each table cell, write how effective *that* filter is for *that* noise. Add some comments or notes about the noise level that can be considered, or which filter parameters can be adequate, and possible "side effects" over the image (e.g. maybe a given filter decreases the amount of noise but also has a negative impact on other aspects of the image). Finally, you may include additional comments highlighting the key features of each filter, or comparing among them.

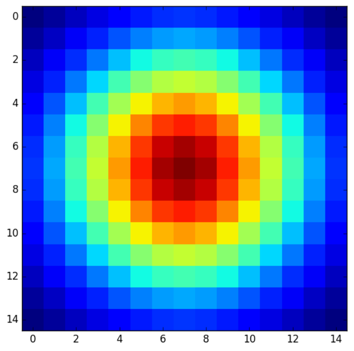
2. As you know, the mean filter is separable. What does this imply? Well, we can obtain the same result by applying sequentially two 1D filters (one over rows and the other over columns) that by applying one 2D filter, with the benefit that the cost of applying two one-dimensional filters is smaller than that of applying a single 2D filter. Write the separable version `average`



(a) `gv1d`



(b) `gv1d` as an image



(c) `gv2d` as an image

Figure 1. The discretized Gaussian: 1D (a,b) and 2D (c), for $n = 15$ y $\sigma = 4$.

`FilterSep()`, of the mean filter given in `averageFilter()`.

Check that the results are equivalent and compare the running times with increasing image and mask sizes. Plot these times using Matplotlib for a visual analysis.

3. In our code, we used directly the function gaussian_filter() to apply the gaussian filter. Now, we are going to do it with an explicit convolution, as we did with the mean filter, i.e. mask creation + convolution. Let's proceed step by step:

   (a) Use the function `gaussian()` of scipy.signal to generate an array of size $n \times 1$ with the values of a 1D Gaussian of the given size n and standard deviation σ. Let `gv1d` be this vector. Display `gv1d` with Matplotlib, as if it was an image, without any interpolation (`interpolation='none'`).

   (b) Generate an $n \times n$ matrix with the values of 2D Gaussian (as before, for the given n and σ). Let `gv2d` be this matrix. You can get `gv2d` by a matrix product of `gv1d` and its transpose, either with `gv1d * gv1d.T`, or `np.outer(gv1d, gv1d)`. Display `gv2d` as an image. Fig. 1 shows an example of both `gv1d` and `gv2d` using a color map.

   (c) Finally, use `gv2d` as a mask (kernel) to convolve the image to be filtered. Compare the result obtained now with that we obtained using `gaussian_filter()`.

   (d) The Gaussian filter is also separable, as you know. Therefore, for the sake of efficiency, instead of applying the convolution with the 2D mask, let's apply it with two convolutions of the 1D masks (`gv1d` and its transpose). Compare the result with that of the 2D convolution in the previous step.

4. Generalize the function `addGaussianNoise()` for it to work both for gray-level and color images. As usual, apply the filter to each of the color bands, separately. Then, find out whether this band-wise application of the filter is actually required, and why (not).

5. The *quotient image* is obtained by pixel-wise division of an image I and its blurred version $I * G_\sigma$, where G_σ is a 2D Gaussian kernel with standard deviation σ. Write a function `quotientImage(im,sigma)` that returns the quotient image for I=`im` with σ =`sigma`. Test it on some images and think of its possible usefulness.