Exercise 8

Spatial filtering

IAML 2020

Purpose These exercises aim to introduce you the use of spatial filters. You will be able to relate and compare filtering, templates, convolution and correlation. You will make methods to apply linear filters and template matching.

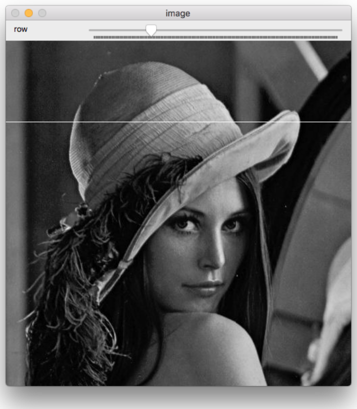
Notes

- All exercises are non-mandatory and we have purposefully added (hopefully) enough exercises to keep everyone busy. We mark less important exercises with ^{extra}. If you don't have time to do all exercises, wait until we release the solutions and use them as a guide.
- In the exercises based on script (i.e. `.py`) files, you are expected to implement code at certain points in the file. Although it is typically clear from context which piece of code you have to change, we have included specifier comments of the type `# <Exercise x.x (n)>`, where `x.x` is the exercise number and `(n)` is the letter associated with the specific task.

Exercise 8.1

Noise removal

In this exercise you work on reducing / removing noise from images. Use the script `noise.py` to answer this exercise. Figure 1 (top) shows a grayscale image with a white line in which represents a row selected by the user. Figure 1 (bottom) shows a one-dimensional signal from the selected row. You can use the 1D signal to evaluate how well noise is removed from the images.



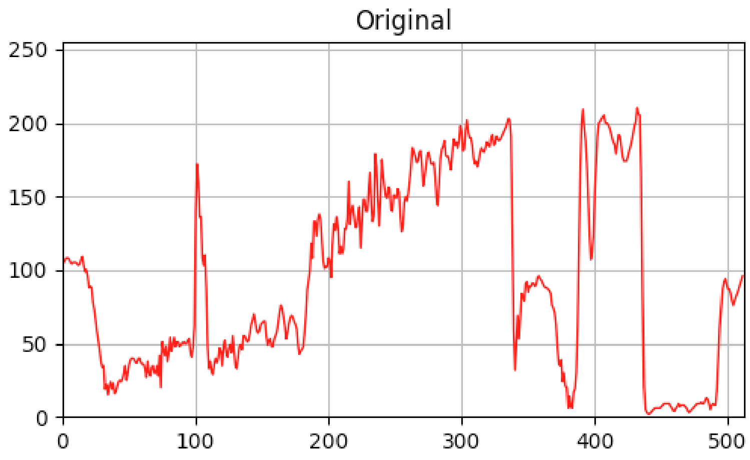


Figure 1: Example of selecting a row in the input image and converting it to an 1D signal.

The functions `salt_and_pepper_noise()`, `gaussian_noise()` and `uniform_noise()` for adding various types of noise have been implemented.

1. Study the implementations of the noise functions `salt_and_pepper_noise()`, `gaussian_noise()` and `uniform_noise()` and make sure you understand them. Notice

   - The function `cv2.add()`[1] adds two matrices of same size (e.g. the image and the random noise).
   - The function `to_uint8()` converts the type of a Numpy arrays to unsigned integer 8-bits (i.e. `np.uint8`).

[1] An OpenCV function to calculate the per-element sum of two arrays or an array and a scalar.

- The function `scipy.stats.normaltest()` tests whether a Numpy array is a normal distribution.

2. **Show noisy images:** The script `noise.py` calls the functions that add the noise ot the image and displays the noisy images, as shown in Figure 2. Run the script.

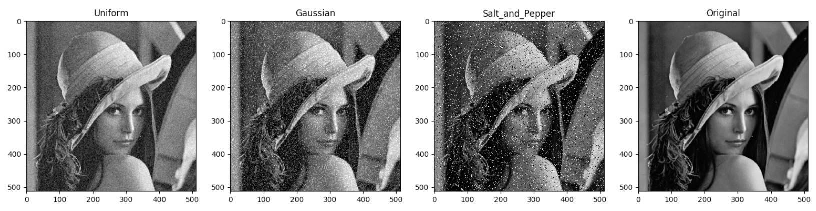


Figure 2: Example of images with simulated random noise.

3. **Reduce image noise:** For each type of noisy image (Gaussian, uniform and salt-and-pepper) determine which $N \times N$ filter best removes the noise (N is a user defined parameter). Implement the best performing technique in the respective functions:

   (a) `salt_and_pepper_filter()` to remove salt-and-peper noise from the input image.

   (b) `gaussian_filter()` to remove Gaussian noise from the input image.

   (c) `uniform_filter()` to remove uniform noise from the input image.

   Hint: OpenCV has functions for median-blur, gaussian-blur, and general convolutions[2].

[2] Link to OpenCV guide on convolutions and image blurring.

Exercise 8.2

Template matching

In this exercise, you will detect regions in the input image using template matching. Use the script `template_matching.py` to answer this exercise.

The function `select_roi()` displays the input image and allows you to select a rectangle area (i.e. a template) in the image. Use the left mouse button to select the *upper left* corner and the right mouse button to select the *lower right* corner. Thereafter, press `Enter` to make the selection. The function returns the selected region of interest (ROI).

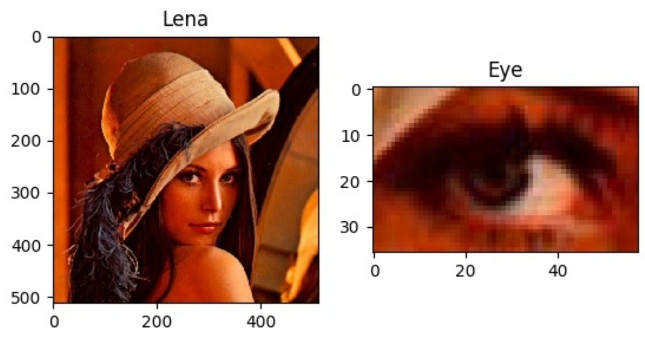


Figure 3: Example of showing multiple images with the function show_images().

Task 1: *Simple template matching*

1. **Template matching:** Use the OpenCV function `cv2.matchTemplate()` [3] to find the location of the template in the input image. Experiment with the different comparison metrics :

   (a) `cv2.TM_CCORR_NORMED`, *Normalized Cross-Correlation Matching* ;

   (b) `cv2.TM_SQDIFF_NORMED`, *Normalized Square Difference Matching*

   (c) `cv2.TM_CCOEFF_NORMED`, *Normalized Correlation Coefficient Matching* .

   How do these methods perform in identifying the template (false positive, true positive)?

2. **Matching the image:** In this exercise, you will calculate the region where the template matches in the input image and draw a red rectangle on this region. You have to:

   (a) Select a template in the input image and use the *Normalized Cross-Correlation Matching Method* (i.e., `cv2.TM_CCORR_NORMED`) to the template matching.

   (b) Use the function `cv2.minMaxLoc()` [4] to locate the maximum and minimum values of the result image.

   (c) Draw a red rectangle where the match occurs in the input image. If the image resolution is $W \times H$ and the template resolution is $w \times h$ then the result must be $(W - w + 1) \times (H - h + 1)$.

[3] An OpenCV function to compare a template against overlapped image regions.

[4] An OpenCV function to find the global minimum and maximum in an array.

3. **Blurring the images:** In this exercise, you will apply a template matching after blurring the input image and the template image. Use the function `cv.GaussianBlur()` [5] to blur images using Gaussian filters. You have to:

   (a) Blur only the template using different Gaussian kernel sizes, e.g. $N = 3, 5, 9, 11$.

   (b) Blur both input image and template image using the same Gaussian kernel sizes that you have tested in the previous step.

[5] An OpenCV function to blur an image using a Gaussian filter.

Task 2: *Image pyramid^{extra}*

In the following, you will make experiments with image pyramids by using the OpenCV functions `cv2.pyrDown()` [6] and `cv2.pyrUp()` [7] to downsample or upsample a given image. Use the script `template_matching_pyramid.py` to answer this exercise.

[6] An OpenCV function to blur an image and downsample it.

[7] An OpenCV function to upsample an image and then blurs it.

1. **Image pyramids:** Select a template in the image `pattern.jpg` and downsample both template and image. Then use *Normalized Correlation Coefficient Matching Method* for template matching. You have to evaluate the image pyramids based on the following questions.

   - What happens if you downsample the image by a factor of 2^1, 2^2, and 2^3, but use the same template for matching?
   - What does this say about template matching and scales?

2. **Match multiple targets:** Create a function `match_all()` that uses cross correlation on an input image (`image`) and a template (`template`), and shows a rectangle around each possible match in the original image. Use a `threshold` in a range from 0 to 1).

3. **Test multiple template matching:** Test your code using the image `pattern.jpg`. The result will look something similar as shown in Figure 4.

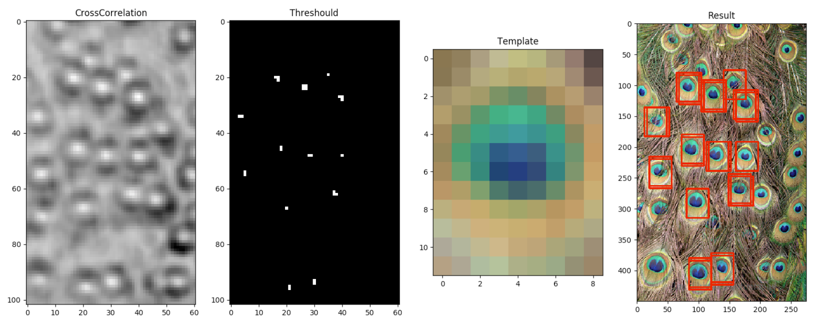


Figure 4: Result of a template matching with multiple targets.

Exercise 8.3

Sharpening

In this exercise, you will create filters to sharpen, shift and smooth the pixels of an input image. Use the function `cv2.filter2D()` [8]). Use the script `sharpening.py` to answer this exercise.

[8] A OpenCV function to convolve an image with a $N \times N$ kernel.

Spatial filtering can be used to sharpen the image making details look clearer than in the original image.

$$K_1 = \frac{1}{9}\begin{bmatrix} -1 & -1 & -1 \\ -1 & 17 & -1 \\ -1 & -1 & -1 \end{bmatrix}, K_2 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Notice that in this case the filter values sum to one.

1. **Question:** Why do sharpen filters contain a single positive value at their centers, which are completely surrounded by negative values?

2. **Create a sharpen filter:** Define the above filter kernels as Numpy arrays and create your own arbitrary 5×5 sharpen filter K_3.

Use the sharpen filters K_1, K_2 and K_3 to answer the following exercises.

3. **Sharpen an image:** Use the filter kernels K_1, K_2 and K_3 in `cv2.filter2D()` to sharpen an input image with each of the kernels K_i. Show the three filtered images and compare them with the original input image.

4. **Plot images as 2D functions:**$^{(Extra)}$ Plot the original image and the filtered image as two-dimensional functions using Matplotlib. Compare how the kernels K_1, K_2 and K_3 sharpen the input image.

Exercise 8.4

Shifting pixels with spatial filters ^{extra}

In the following exercises, you will create filters to shift the pixels of an image to the left, right, up or down n pixels. For example, the kernel K_{left} shifts the pixels of an image 1 column to the left:

$$K_{left} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Use `shifting.py` for your implementation. Have a look at the function `shift_to_left()`. Given an input image, this method uses the filter K_{left} to shifts the pixels of the input image n columns to left (e.g. $n = 10$). It is doing this by using K_{left} n times. However, this is not a clever algorithm and it slows down the Python script.

1. Why does L_{left} shift the pixels to the left?
2. **Improve the shift filter:** Change the function `shift_to_left()` to shift the pixels of an input image, n columns to the left using a single filter.
3. **Different directions:** Make the functions `shiftToRight()`, `shiftToUp()`, and `shiftToDown()` to shift the input images n pixels to the right, up and down. Use the input image to test these functions and evaluate if the images are correctly shifted.