Exercises 4 (Week 4) - Spatial Filtering

All exercises are relevant for the course but none are mandatory. You are highly advised to do as many exercises as you have time for. We have made enough exercises so that there is enough material to do within class hours as well as extra exercises (marked as *"Extra"*) for those who want to do more outside class. None of the exercises should take very long to solve so don't spend too much time not making progress. In case you need help don't hesitate to ask as we can probably give you a hint or two to guide you :). You are welcome to send questions by email.

Learning goals In these exercises you will be experimenting with spatial filtering. You will initially investigate the relationship between noise and smoothing and then you will experiment with template-matching. If you did not finish the exercises from last week, then please do this weeks exercises first. We will happily be of assistance with previous exercises.

Get Exercise Data

I Extract the zip file *"Exercises_4.zip"* and set that folder as the source folder of your project.

II The file *Exercises4_Tools.py* contains auxiliary functions (e.g., showImage functions that you were working with last week) that can become helpful in these exercises. You will need to import this file in your programs. e.g. :

```
import Exercises4_Tools as tools
tools.show_Multiple_images_pylab(imgOriginal=I1,imgFiltered=I2)
```

III Use the OpenCV reference manual [Ope11a] (that exists in the materials folder) or the online documentation for additional information and help about the OpenCV functions.

Filtering, Smoothing And Noise

Create a file called *filtering.py* in your source folder. All the exercises in this section will be done in this file. For the following exercises you may use the OpenCV function *cv2.filter2D()*.

1. In this exercise you will be experimenting with the box filter (that is all entries are 1). Remember to normalize the kernels before use.

   (a) Make a function *BoxFilter(I,n)* that given the input image *I* (grayscale) and size *n* of the box filter, returns the filtered image.

   (b) Load the *Lena.png* image and convert it to grayscale into a variable *LenaGray*.

   (c) Apply a 3×3 box filter to *LenaGray* using the *BoxFilter* function. Display the input image *I*, the filtered image and the difference image (use *cv2.absdiff()*) between the original image and the filtered image in one window. Where do the changes occur?

   (d) Try the same but now with different values of *n = 5, 9, 11*. What do you observe?

   (e) $^{(Extra)}$ Use *cv2.filter2D()* to apply the following kernels on the *LenaGray* image and observe their effect (e.g. what do they do?):

A:

1	0	0
0	1	0
0	0	1

B:

-1	0	1
-2	0	2
-1	0	1

C:

-2	-2	0
-2	6	0
0	0	0

D:

3	10	3
0	0	0
-3	-10	-3

2. In the following exercise you will experiment with noise and smoothing of images. Random noise with a uniform distribution can be generated using *numpy.rand()*. For example :

```
import Exercises4_Tools as tools
import numpy as np
M,N=imgGray.shape
noise = np.random.rand(M,N)*60
noise=tools.to_uint8(noise)# convert the noise array to unit8
```

Similarly, Gaussian random noise can be generated through *numpy.random.normal()*. e.g. :

```
import Exercises4_Tools as tools
import numpy as np
gaussianNoise = np.random.normal(mu, sigma,(M,N))
gaussianNoise=tools.to_uint8(gaussianNoise)
```

(See scipy documentation for more information.)

   (a) Create a matrix with random uniform noise of the same size as *LenaGray* and add it to the image. Call the image with noise added *LenaGray_uniformNoise*.

   (b) Do the same but this time add Gaussian noise and call the image *LenaGray_gaussianNoise*. (start with mu=0 and sigma=50)

   (c) Make a function called *SaltPepperNoise(I,density)* that given an image *I* puts salt-and-pepper noise with the density factor of *0<density<1*, and returns the result.
   (Hint: pepper noise can be done as below, think about the salt noise!)

```
I[((np.random.rand(M,N))<density)] = [0]
```

   (d) Use the *SaltPepperNoise* function to add salt-and-pepper noise to the *LenaGray* image and call the new image *LenaGray_SaltPPrNoise*.

   (e) Apply a 3×3 box filter (mean filter) to *LenaGray_uniformNoise, LenaGray_SaltPPrNoise, LenaGray_gaussianNoise* and display the results in one figure.

   (f) Try the same but with different values of *n = 5, 9, 11* for the kernel. Which type of noise does the box filter handle the best?

   (g) Use the *cv2.GaussianBlur(img, (n,n),0)* and apply a gaussian filter on *LenaGray_uniformNoise*, *LenaGray_SaltPPrNoise* , *LenaGray_gaussianNoise*. Which type of noise does the gaussian filter handle?

3. $^{(Extra)}$ In this exercise you will be experimenting with non-linear filters such as Median and Bilateral filters.

   (a) Use the *cv2.medianBlur(img, n)* and apply a median filter on *LenaGray_uniformNoise, LenaGray_SaltPPrNoise, LenaGray_gaussianNoise*. For which kind of noise does the median filter handle the best? Why does it perform better on these images?

   (b) * Extract the RGB channels of the *face1.jpg* image. Perform smoothing on each channel using the *cv2.medianBlur()* function and the kernel size of $n = 11$. Merge the channels (*cv2.merge((R,G,B))*) and display the filtered image. What do you observe? Try the other smoothing functions *cv2.bilateralFilter()* and *cv2.GaussianBlur()*.

   (c) Extract the HSV channels of the *face2.jpg* image. Perform smoothing on the gray channel (V) using the *cv2.medianBlur()* function and the kernel size of $n = 11$. Merge the channels (*cv2.merge((H,S,V))*) and display the filtered image. What do you observe? Try also *cv2.GaussianBlur()* and *cv2.bilateralFilter()*.

Template matching

1. Create *templateMatching.py* in your source folder. All the exercises in this section will be done in this file. In the following exercises you may use the function *SelectArea(img)* from the *Exercises4_Tools.py*. This function displays the image *img* and you can select a rectangle area in the image window using the

mouse buttons, and then when you press the Enter key, it returns the coordinates of the UpperLeft and the LowerRight corners of the selected area. You can read more about the function by typing *help(SelectArea)* in the python command prompt).

(a) Load the image *Lena.jpg* and use the function *SelectArea(img)* to extract the region of Lena's left eye using slicing. Call the extracted region, *template* Display *template* using pylab.

(b) Correlate the left eye *template* with the image *Lena* using *cv2.matchTemplate()* with different methods *cv2.TM_CCORR_NORMED*, *cv2.TM_SQDIFF_NORMED* and *cv2.TM_CCOEFF_NORMED*. Show the resulting images in one window using pylab and compare the methods. Do all these methods return a high value around the left eye in the resulting images?

(c) * Do the template matching on *Lena.jpg* with different templates. Compare the different methods of template matching when the extracted template contains:
(i) Few intensity changes in all directions (e.g. almost uniform intensity)
(ii) Few intensity changes in one direction and more in another direction
(iii) Many intensity changes in all directions

(d) Select a template in the *Lena* image and use the Cross Correlation (*cv2.TM_CCOEFF_NORMED*) for template matching. Use *cv2.minMaxLoc()* to locate the maximum and minimum values of the result image. Draw a rectangle on the original image with the size of the template in the original image where the best match occurs. (Hint: If image is $W \times H$ and template is $w \times h$ then the result must be $(W - w + 1) \times (H - h + 1)$. For more help see pages 215-223 of the [Ope11b])

(e) Load and select a template in *pattern1.jpg* and use different methods for template matching. What happens if you downscale the image (use *cv2.pyrDown()*) by a factor of $2, 4$ and 8 but use the same matching template? What does this say about template matching and scales?

(f) * The following code snippet rotates an image I around its center with 40 degrees and displays it.

```
import Exercises4_Tools as tools
I = cv2.imread(SOMEIMAGE);
IR = tools.rotateImage(I,40)
cv2.imshow('ROT',IR)
cv2.waitKey(0)
```

Load and select a template in *pattern1.jpg*. Rotate the original image with 10,15,20,25,30 degrees and use template matching to detect the template in the rotated images. For which degrees does template matching give the best results. Are there any differences between the methods? What does this say about template matching and rotation?

(g) Make a function called *matchAll(img,template,threshold)* that does cross correlation on an image, *img*, and a template, *template* and shows a rectangle around each possible match in the original image (larger than *threshold*).

(h) Test the program on *pattern1.jpg*. Your results should look something like figure 1h.

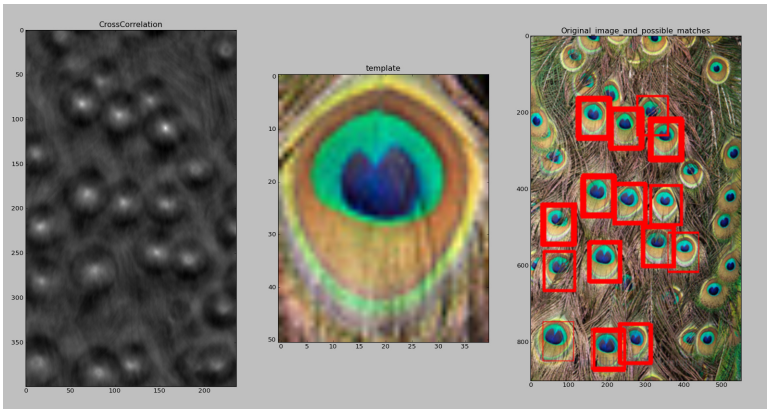

figure 1

(i) * Load the image *text.jpg*. Use *matchAll* on *text.jpg*. Select different letters as a template. Can you apply the same theshold for different letters? Are some letters easier to detect than others? You results could look something like 1i

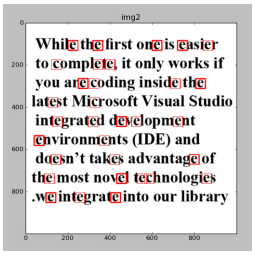

figure 2

2. $^{(Extra)}$ This exercise is about template matching in video sequences.

   (a) Open the *video.py* file which loads a video and shows it. Change the program so that by pressing the space key it stops the video and allows you to select a template in the stopped frame. Then it does template matching (Cross Correlation method) in the rest of the video and shows the best match in the video. (more help in the *video.py*)

   (b) Select the face as template and see if Cross Correlation method can detects the face in the video sequence. Does this method handle the changes in the scale?

   (c) * Downscale the image for template matching. Does it change the speed and accuracy?

References

[Ope11a] *The OpenCV Reference Manual*. 2011.

[Ope11b] *The OpenCV Tutorials*. 2011.