Practical 2:
Intensity Transformations and Spatial Filtering
Week 3

Unit Coordinator & Lecturer: Dr. Maryam Haghighat

Notes

- In general, it is recommended that you record the details of the steps you went through to obtain your results, for your personal reference.
- Whenever it appears in this document, "compare results" means: show the results, comment on them, and explain the differences (or similarities) observed (if any).
- Items in italic and parentheses indicate what is asked for (e.g., what would be required in a report on this lab), e.g. (*text*) if a written explanation is required, (*print*) if you should show a printed/displayed image of your result, (*sketch*) is a sketch or drawing that can be used to help illustrate the answer, (*code*) if you should show (print or write) the code you have written.
- Some items will be shown as ***Optional***. This means that it is recommended that you try to solve those, but they are not necessary. (Typically, students aiming for an HD should address those questions).

1 Intensity Transformation Functions

1.1 Adjust Contrast and Brightness

In Python, there are different libraries implementing functions for contrast and brightness adjustment. Please, for different implementations and functions, refer to SciPy and scikit-image. To transform intensity values (gray-scale image) via `gamma` (γ) correction, we can refer to the function

$$\hat{I} = 255 * (\frac{I}{255})^{\gamma}$$

This maps all the image values to the new gamma-adjusted values. What if you wanted to adjust only values within certain ranges?

NB: The negative image of f can also be obtained with `cv2.bitwise_not(f)`.

In a more general form, contrast and brightness of an image can be controlled via:

$$g(x) = \alpha * I(x) + \beta$$

Test values of α and β for a coloured image and observe changes.

1.2 Logarithmic and Contrast-Stretching Transformations

Logarithm transformations of an image `f` are expressed as:

$$g = c * log(1 + \texttt{float}(f))$$

Apply such a logarithm transformation on an image of a Fourier Transform, for various values of `c` (*print one*). To do so, apply a 2-D Fourier Transform to the input image using `fft2`, "arrange" the frequencies for display with `fftshift` and display the magnitude obtained using the function `abs` (NB: these operations will be detailed in the next tutorial):

```
F=fft2(f);
Fc=fftshift(F);
plt.imshow(np.abs(F))
```

Then apply the `log` transform to that image (starting with `c=1`):

```
g = log(1 + np.abs(Fc));
```

The resulting values in the matrix `g` are likely to be 'compressed' (small range of values). In order to bring those values back to the full range of the display (8-bit image), we can use normalisation. Display the normalised values `gs` and explain what that command line is doing to the image `g` (*text*).

A function for Contrast stretching (see section 3.1.1 of the lecture) can be expressed as:

$$s = T(r) = \frac{K}{1 + (m/r)^E} \tag{1}$$

where r represents the intensities of the input image, s the corresponding intensity values in the output image, E controls the slope of the function, and K can be used for normalisation purpose. Implement Equation (1) in Python, plot it for specified values of `m` and `E` to have a representation of this function, and apply this contrast stretching to your original image (*text*). Explain how you chose appropriate values for `m` and `E`, given an application (*text*).

2 Histogram Processing and Function Plotting

The focus of this section is on obtaining, plotting, and using histograms for image enhancement.

2.1 Generating and Plotting Image Histograms

The core function for dealing with image histograms is `np.histogram`:

```
h = np.histogram(f, b)
```

where `f` is the input image, `h` its histogram, and `b` the number of bins (subdivisions of the intensity scale) used in forming the histogram (if `b` is not included in the argument, `b = 256` is used by default).

The normalised histogram can be obtained by passing the argument `density=True`. Compare the following various ways to plot an image histogram (*text + optional sketch*):

- using directly `np.histogram`,
- using the `np.bar` function,
- using the `np.stem` function (similar to `np.bar`),
- simply with the function `plot`.

2.2 Histogram Equalisation

The histogram equalisation of the image `f` can be obtained by:

```
g = cv2.equalizeHist(f)
```

To obtain the transformation function of the histogram equalisation (i.e. the cumulative sum of normalised histogram values), we can use the function `np.cumsum`:

```
cdf = np.cumsum(h)
```

Perform a histogram equalisation on a couple of images (*print one, with original*), find out the transformation function (*print or draw*) and interpret its effect on the intensity levels of the image (*text*).

2.3 Histogram Matching/Specification (*Optional*)

OpenCV does not have a function for histogram matching. It can be achieved via the scikit-image package `exposure.match_histograms(f, target, channel_axis)`.

Perform a histogram matching on the same input images as before to obtain a histogram with the given square-root shape `image_data`. Compare the output image with the results obtained after histogram equalisation.

Now load `Golden.jpeg` and `flower.jpg` images and perform a histogram matching on the full RGB spectrum.

3 Spatial Filtering

3.1 Linear Spatial Filtering

OpenCV implements linear spatial filtering using the function `cv2.filter2D`:

```
g = cv2.filter2D(f, -1, w, borderType)
```

where `f` is the input image, `w` is the filter mask. The parameter `borderType` deals with the "border-padding" issue. When you apply a 3×3 filter kernel, then, when processing the left-most pixels in each row, you need pixels to the left of them, that is, outside of the image. You can let these pixels be the same as the left-most image pixels ("replicated border" extrapolation method), or assume that all the non-existing pixels are zeros ("constant border" extrapolation method), and so on.

Generate a mask `w` and study the results obtained.

For some functions, it is sometimes required to manually add some padding to an image. In OpenCV, we can do that with

```
fp = cv2.copyMakeBorder(f, 200, 200, 200, 200, cv2.BORDER_REFLECT)
```

Note that OpenCV usually applies padding automatically, but it might not be the case for other libraries or functions.

3.2 Non-linear Spatial Filtering

"Order-statistic filters" (also called "rank filters") are non-linear spatial filters whose response is based on ordering (ranking) the pixels contained in an image neighbourhood (or window), with the value determined by the ranking result. In Python, these filters are typically accessed by their specific operation names. Common types of rank filters include the median filter, minimum filter, and maximum filter.

Morphological operations, such as dilation and erosion, use maximum and minimum filters to modify the structure of objects within an image

```
cv2.erode(f, kernel) cv2.dilate(f, kernel)
```

Apply these filters and observe the effect in the image.

4 Common Spatial Filtering Operations

4.1 Linear Spatial Filters

An example of a linear filter is the Laplacian filter. Generate a Laplacian kernel, and apply this filter to an image via the function `cv2.filter2D`. Add (or subtract, as needed) the Laplacian image to the original image to restore the gray tones lost by using the Laplacian. Explore both common Laplacian kernel configurations.

4.2 Non-linear Spatial Filters

Unlike linear filters, non-linear filters often involve thresholding operations and prove particularly useful in preserving edges while reducing noise. A common example is the filter `cv2.medianBlur`. In the presence of a "salt & pepper" noise, it is particularly effective as it relies on pixel values rank instead of averaging, thus not blurring edges as much as linear filters do.