

CCIT4076 Engineering and Information Science

Laboratory 5: Digital Image Processing

HKU SPACE Community College, Fall 2022

1 Image Science

In imaging science, image processing is any form of signal processing for which the input is an image, such as a photograph or video frame; the output of image processing may be either an image or a set of characteristics or parameters related to the image. Most image-processing techniques involve treating the image as a two-dimensional signal and applying standard signal-processing techniques to it.

Image processing usually refers to digital image processing, but optical and analog image processing also are possible. The acquisition of images (producing the input image in the first place) is referred to as imaging. Image processing refers to processing of a 2D picture by a computer.

An image defined in the “real world” is considered to be a function of two real variables, for example, $I(x, y)$ with I as the amplitude (e.g. brightness/exposure to light) of the image at the real coordinate positions (x, y) , where x and y are the horizontal or vertical dimensions.

Modern digital technology has made it possible to manipulate multidimensional signals with systems that range from simple digital circuits to advanced parallel computers. The goal of this manipulation can be divided into three categories:

- Image Processing (image in \rightarrow image out)
- Image Analysis (image in \rightarrow measurements out)
- Image Understanding (image in \rightarrow high-level description out)

In this lab, we are going to use the Octave to perform some simple image processing.

2 Image Processing with Octave

To perform the image processing in Octave, we are going to use the image processing toolbox embedded with the Octave system:

```
>> pkg load image
```



Figure 1: Subject Images

from which you may read an image file as an array object in the Octave workplace by:

```
>> X = imread('myImage.png');
```

Note that the exact file name of `'myImage.png'` determines which image file you are reading from the current directory. We provide four testing images as shown in Figure 1 for your testing. After you have loaded an image data onto the array `X`, you may show the image using the command:

```
>> imshow(X);
```

which interprets the data stored in `X` and pictorialises the image in an Octave figure. Finally, one may store the image data into other image format, such as bitmap and jpeg files, using the command lines

```
>> imwrite(X, 'myImage_New.xxx');
```

where `xxx` is the file extension that determines the graphical file type. Some popular file types are:

1. `bmp` — Windows' bitmap
2. `jpg` — Joint Photographic Experts Group
3. `png` — Portable Network Graphics

You may convert, say for instance, a png file into a bmp file in this way.

3 Simple Image Operations

We now discuss some elementary image operations. First, consider a 90 degrees rotation toward the anti-clockwise direction (i.e. rotate to the left hand side). A notable example is presented in Figure 2. Suppose the original image is stored in the array `X_org`, the rotated image can be obtained by the command lines:

```
>> X_rot = rot90(X_org);
```

and you may visualize it using `imshow` as discussed.

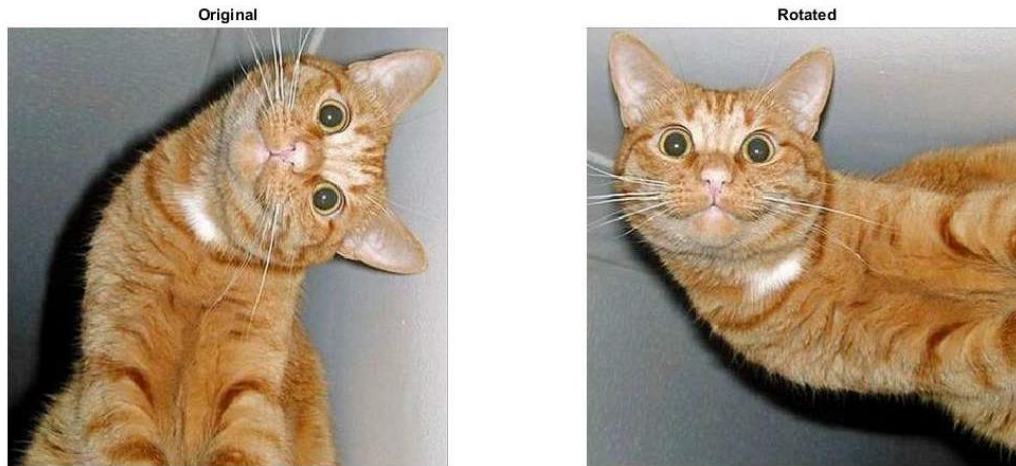


Figure 2: Image Rotation

Next, we consider symmetric flipping of images. The command line:

```
>> X_LR = fliplr(X);
```

returns an array `X_LR` that is with its columns flipped in the left-right direction; similarly:

```
>> X_UD = flipud(X);
```

returns `X_UD` with its rows flipped in the up-down direction (that is, about a horizontal axis). See the example below:

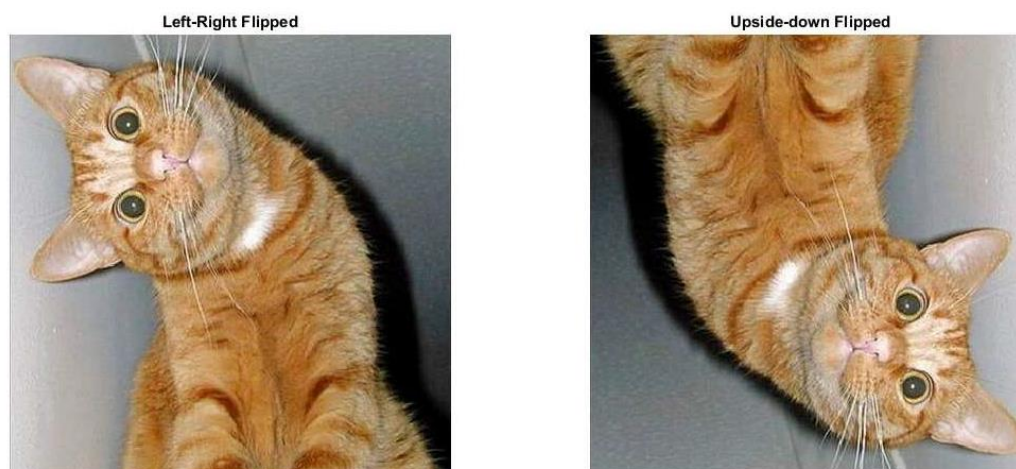


Figure 3: Image Flipping

Finally, we may tune the brightness of an image by introducing a constant value to the whole image matrix. This is easily done by

```
>> X_br = X + bias_val;  
>> X_dk = X - bias_val;
```

For example, when the bias value is `bias_val = 100`, we may obtain the following results:

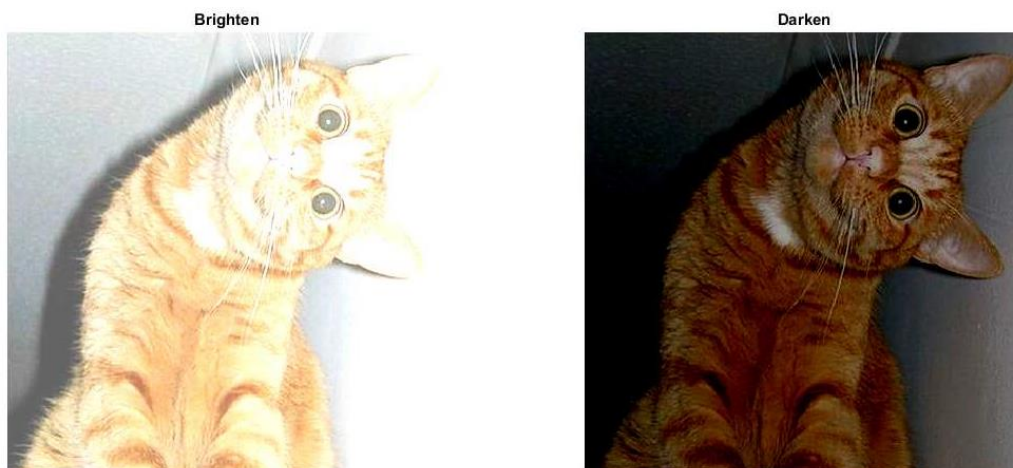


Figure 4: Image Brightness/Darkness Tuning

4 Image Filtering

As discussed in the lecture, we are able to perform filtering over image data. Important applications include image denoising, image sharpening and edge detection. We will go on to run through these experiments below. Consider the noisy image in the lab pack:



Figure 5: [lenna_noisy.png](#)

We plan to apply filtering and see if the noise can be smoothened. Recall that the filter/kernel coefficients of the average filter and the Gaussian filter are acquired by calling:

```
>> w_avg = fspecial('average', 3);  
>> w_gau = fspecial('gaussian', 3);
```

and the command line to perform filtering is

```
>> Y = imfilter(X, w);
```

assuming X is the array storing the noisy image; and w is either the average filter or the Gaussian filter. The expected output Y shall look like:



Apparently they do not work so well. Let us try for the median filtering:

```
>> Y_med = medfilt2(X);
```

The resulting image should be pretty nice.



Think: How to apply filtering on RGB images?

4 Image Sharpening

Image sharpening highlights transitions in intensity of an image. As a result, the contrast of the image is improved. In this section, we are going to perform image sharpening using the unsharp masking method. Consider the blurry moon image in the lab pack:



Figure 6: `moon_blurred.png`

We will use the following command to sharpen this image:

```
>> G = EIS_Sharpen(X, k);
```

The function `EIS_Sharpen.m` will firstly produce a smoothed version of the original image `X` by using a Gaussian filter. Then, an unsharp mask is computed. The sharpened image `G` is obtained by adding the unsharp mask to the original image. The input parameter `k` is used to control weight of the unsharp mask to be added.

Start with `k = 1` for the experiment and compare the original and sharpen images visually. Then, gradually increase the value of `k` and observe the differences.

5 Edge Detection

Edge detection is a preprocessing step for many computer vision systems. For examples, we can find edge detection in applications like medical imaging, vehicle detection, and object recognition. In this section, we are going to detect edges in an image using the widely known Sobel method and Canny method.

Consider the following image in the lab pack:



Figure 7: [building.png](#)

To find the edges in this image using the Sobel method, we can use the command:

```
>> G = edge(X, 'Sobel', threshold);
```

It will detect edges from the original image `X` by using the Sobel method. Edges that are stronger than `threshold` will be shown on the returned image `G`.

Start with `threshold = 0.0` for the experiment and compare the original and the returned images visually. Then, gradually increase the value of `threshold` with step size ≤ 0.1 and observe the differences between the images.

To find edges using the Canny method, we can use the command:

```
>> G = edge(X, 'Canny', threshold);
```

It will detect edges from the original image `X` by using the Canny method. Edges that are stronger than `threshold` will be shown on the returned image `G`.

Start with `threshold = 0.0` for the experiment and compare the original and the returned images visually. Then, gradually increase the value of `threshold` with step size ≤ 0.1 and observe the differences between the images.