Exercise 1: The basics of image processing

Multimedia systems

2018/2019

First, create a folder in which you will store all your solutions and code for this course. Inside the folder, create a separate sub-folder for each exercise.

Create a folder `exercise1` that you will use during this exercise. Unpack the content of `exercise1.zip` that you can download from the course webpage to the folder. Save the solutions for the assignments as the *Matlab/Octave* scripts to `exercise1` folder. In order to complete the exercise you have to present these files to the teaching assistant. Some assignments contain questions that require sketching, writing or manual calculation. Write these answers down and bring them to the presentation as well. The tasks that are marked with ★ are optional. Without completing them you can get at most 75 points for the exercise (the total number of points is 100 and results in grade 10). Each optional task has the amout of additional points written next to it, sometimes there are more optional exercises and you do not have to complete all of them.

If you have not used the *Matlab/Octave* environment before, take a look at the introductory tutorial, which will explain the basic syntax and properties of the language. The tutorial is available on the website of the course.

Assignment 1: Basic image processing

The aim of this assignment is to familiarize yourself with the basic functionality of *Matlab/Octave*, as well as the use of matrices for storing image information. In this assignment, you will try out the following built-in functions: `imread`, `imshow`, `imagesc` and `colormap`. Complete the following steps, and write the code to the assignment file (`exercise1_assignment1.m`):

(a) Read image from the file `umbrellas.jpg`, and display it using functions `imshow` and `imagesc`.

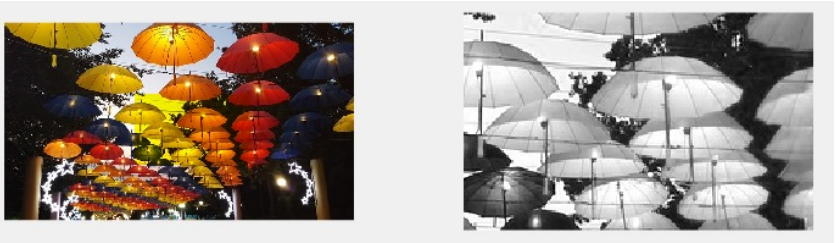
(b) At the first glance, there is no substantial difference between the two functions used to display the image, `imagesc` and `imshow`; we will demonstrate the difference between them later on. The image that you have loaded consists of three channels (**R**ed, **G**reen, and **B**lue), and is represented as a 3-D matrix with dimensions *height* × *width* × *channels*. You can query the dimensions of the matrix using the following command: `[ h, w, d ] = size(A)`. Now, convert the color image into a grayscale one by averaging all three channels[1]. When performing operations on the matrix, one needs to be careful about its type. The image is by default loaded as a matrix of type `uint8`, meaning that the pixel values are in range between 0 and

[1] A similar effect can be achieved by using the built-in `rgb2gray` function.

255. When summing up `uint8` values in *Matlab/Octave*, they *saturate on integer overflow*[2] — if the summed value exceeds 255, it will be truncated to 255. Therefore, we first need to convert the matrix to `double`, perform the averaging, and then convert it back to `uint8`. Once the image is converted, display it again using the previously introduced functions.

(c) In the previous example, the image is displayed using the default palette or *color map* (the actual color map used by default may depend on your version of *Matlab/Octave*). A color map defines the set of colors used to display each gray level in the single-channel image; for example, the low values may be displayed as dark blue, and high values as red. The color map of the displayed image can be changed using the `colormap` command (see *Matlab/Octave* help for details). Try using the following pre-defined color maps: `jet`, `bone`, and `gray`.

(d) Cut out a rectangular sub-image, and display it as a new image. Mark the same region in the original image by setting its third (the blue) color channel to 0, and display the modified image.



(e) Display a grayscale image that has the selected region negated (its values are inverted).

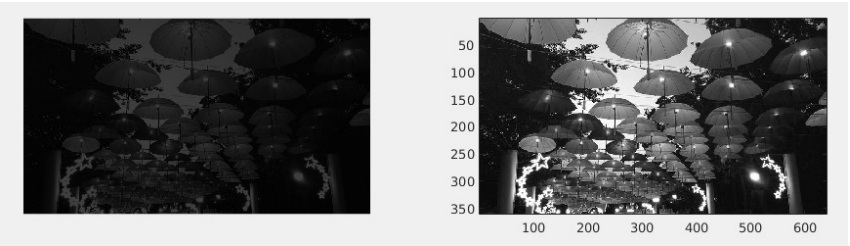
(f) Compute and display a *thresholded binary image*. Thresholding is an image operation that produces a binary image (mask) of the same size as the source image; its values are 1 (true) and 0 (false), depending on whether the corresponding pixels in the source image have values greater or lower than the specified threshold. Use a threshold of 150, and display the resulting image. Change the color map to grayscale. Use the following code-snippet as a starting point:

```
A_thr = A_gray > 150; % elements with value > 150 are set to 1, others to 0
figure(1);
imagesc(A_thr);
```

(g) ★ (5 points) Lastly, you will perform *a reduction of grayscale levels* in the image. First, read the image from `umbrellas.jpg`, and convert it to grayscale. Convert the resulting matrix to the `double` floating-point format. The highest possible grayscale level in this case is 255, and the lowest possible one is 0. Compute a new image by multiplying elements of the original image with an appropriate factor (and then round the result). Choose the multiplication factor so that the resulting image will have the highest possible grayscale level 64, while the lowest one remains at 0. Display the resulting image, again using both the `imagesc` function and the `imshow`

[2] In some other languages, such as C/C++, the value will merely overflow.

function — do you notice the difference in the way these two functions display the image? Try to explain this difference (the description can be found in the *Matlab/Octave* help).



Assignment 2: Histograms

In this assignment, we will take a look at the construction of *histograms*. Histograms are a very useful tool in image analysis; as we will be using them extensively in the later exercises, it is recommended that you pay extra attention to how they are built. In this assignment, we will focus on the construction of histograms for grayscale (single-channel) images.

(a) Create a script file `myhist.m` and use it to implement function `myhist` that is provided with a 2-D grayscale image and a number of bins, and the function returns a 1-D histogram as well as the bottom reference value for each bin. Look at the code below and explain what is the meaning of each line.

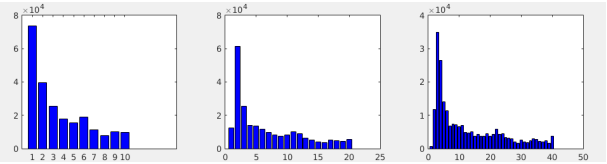
```
function [ H, bins ] = myhist(I , nbins)
I = reshape(I, 1, numel(I) ); % reshape image in 1D vector
H = zeros(1, nbins) ; % initialize the histogram
max_val_in = 255 ; % highest input value
min_val_in = 0 ; % lowest input value
max_val_out = nbins; % highest cell index
min_val_out = 1 ; % lowest cell index

% Compute bin numbers for all pixels
f = (max_val_out - min_val_out) / (max_val_in - min_val_in); % Compute the scaling factor
idx = floor(double(I) * f) + min_val_out; % Calculate indices for all pixels
idx(idx > nbins) = nbins; % Truncate the outliers
for i = 1 : length(I) % Now iterate the image and increase appropriate histogram cell for each ↩
     pixel
    H(idx(i)) = H(idx(i)) + 1;
end
% Normalize the histogram (sum of cell values equals 1)
H = H / sum(H) ;
% Compute reference values for all cells in the histogram
bins = ( (1 : nbins) - min_val_out) ./ f;
```

It is important that you know what is performed in this function, if you are unsure please check the lecture slides again. Test the code by writing a script (e.g. `exercise1_assignment2a.m`) that loads an image, converts it to grayscale, uses the `myhist` function to calculate image histogram and visualize it using the integrated function *bar*. Experiment with different numbers of bins and observe what happens, how are the histograms different?

(b) Histogram calculation is also implemented in *Matlab/Octave*in form of function `hist`, however, this function works a bit differently. To try this out, write the code

below to script file `exercise1_assignment2b.m`. Read image from the file `umbrellas.jpg` and change it to grayscale. Since the `hist` function does not work on images, but on sequences of points we have to first reshape the image matrix of size $(N \times M)$ into 1-D vector of size $NM \times 1$ and compute a histogram on this sequence. Plot a histogram for different number of bins and explain why does the shape of the histogram change with that number.

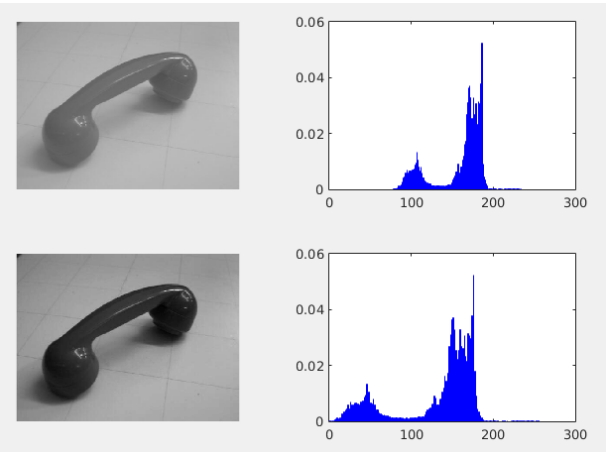


(c) ★ (10 points) Within this task, you will implement a simple image operation called *histogram stretching*. Using the pseudo-code provided below, implement the function `histstretch`, which performs the histogram stretching on the input grayscale image. For the outline of the algorithm, consult the slides from the lectures. As we are performing the same operation (with the same factors) on all image elements, the operation can be sped-up via use of matrix operations, which perform the operation on the whole image at once. Important: you should not use `imadjust` or similar integrated functions to achieve the goal but rather implement the low-level calculation yourself.

```
function S = histstretch (I)
 % 1. determine the minimum and maximum value of I
 % 2. the minimum and maximum of the output image S are known
 % 3. use the stretch formula to compute new value for each pixel
```

Hints: The maximum and the minimum grayscale value in the input image, `v_max` and `v_min`, can be determined using commands `max(I(:))` and `min(I(:))`. The new pixel value can be computed using a formula that is similar to the one that we used in the function `myhist`.

Test the function by writing a script that reads an image from file `phone.jpg` (note that it is already a grayscale image), compute the histogram with 255 bins, and displays it using `imshow`. As you can observe from the histogram, the lowest grayscale value in the image is not 0, and the highest value is not 255. Perform the histogram stretching operation and visualize the results (display the image using `imshow` and plot its 255-bin histogram).



Question: Why is the operation called *histogram stretching* if we are not even using the histogram of an image directly? Which built-in function for displaying images is doing something similar?

Assignment 3: Color spaces

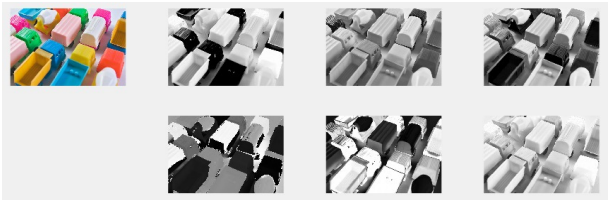
The color information can be encoded using different *color spaces*, with each color space having its own characteristics. This assignment will demonstrate how a relatively simple conversion between the RGB and the HSV color spaces helps us achieve interesting results.

(a) Read image from the file `trucks.jpg`. Display the image on screen, both as a color image in the RGB color space, and each of its channels as a separate grayscale image.

Convert the image from the RGB color space to the HSV color space, using the built-in function `rgb2hsv`, and display each channel as a separate grayscale image. When working with resulting matrices, take a note of their type; the original image in the RGB color space is stored in a matrix of type `uint8` (unsigned integers in range 0 to 255), while the converted image is stored in a matrix of type `double` (real values in range 0 to 1).

Write the code for this assignment to file `exercise1_assignment3a.m`.

Question: How do you interpret the channels of the HSV color space with respect to the original RGB channels?



Question: How would you comment the effect of changes in each component (channel)?

(b) Different color spaces are also useful when we wish to threshold the image. For example, in the RGB color space, it is difficult to determine regions that belong to a certain shade of a color. To demonstrate thiss, create a file `exercise1_assignment3b.m` and write a script that loads the image from file `trucks.jpg`, and thresholds its blue channel with the threshold value of 200. Display the original and the thresholded image next to each other.

Question: Did we achieve a good segmentation of blue regions? How come?

If we want to extract a custom color shade, it is much more intuitive to convert the image to the HSV color space. Modify your script so that the image is converted from the RGB to the HSV color space, and perform thresholding on the Hue channel. As the blue color occupies only a limited portion of the hue value range, we need to apply two thresholds — an upper and a lower one. In the *Matlab/Octave* environment, this can be done by applying a logical function of two masks (obtained with two

different thresholds), e.g., `AB = A & B`. To determine the threshold values for the blue color, you can use the following code snippet, which will display the color spectrum corresponding to the whole hue component:

```
I = ones(20, 255, 3);
I(:, :, 1) = ones(20, 1) * linspace(0, 1, 255);
image([0, 1], [0, 1], hsv2rgb(I));
```

Experiment with different threshold values to find the optimal ones, and display the resulting thresholded image. To display the masked color in an easily interpretable way, you can use the function `isolate_color` that is included in the exercise material.

(c) ★ (5 points) How would you make the thresholding in the HSV color space more robust? Hint: Why is it difficult to determine the hue for some regions? Could you use an additional information to extract such regions? Verify your solution by implementing it.

(d) ★ (5 points) How would you use the HSV color space to perform the histogram stretching operation to improve the contrast, but without distorting the colors? Find a color image with weak contrast and write a script that demonstrates your solution on it.

Assignment 4: Homography

A *homography* is a bijective transformation between two projection spaces, in our case planes; the first plane is the source image plane, while the second plane is defined by input points that denote an area in which we wish to embed the source image. A homography is described by a matrix; in case of transformation between two planes, it has dimension 3×3.

(a) Write a script in which you read and display the image from `monitor.jpg`, and prompt the user to pick four points (use the built-in `ginput` function). Define a suitable order of point selection (for example, begin at top-right corner and continue in counter-clockwise direction). Afterwards, display a polygon that is defined by the selected points (e.g. set the pixels within the polygon to white as shown in the example below). For the rest of the assignment, it is recommended that you pick the four points that correspond to the corners of the monitor in the image. Save those points to a file so manual clicking is not required every time.

(b) Create a script that reads an image and uses the its dimensions and the selected points to compute the homography, using the function `estimate_homography` (part of provided materials for this exercise). This function returns the homography matrix `H`, which can be used for transforming pixel-coordinates from the source image plane using the following formula:

$$p'_b = \mathbf{H}_{ab} p_a \tag{1}$$

with the following individual parts of the equation:

$$p_a = \begin{bmatrix} x_a \\ y_a \\ 1 \end{bmatrix}, p'_b = \begin{bmatrix} w'x_b \\ w'y_b \\ w' \end{bmatrix}, \mathbf{H}_{ab} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}. \quad (2)$$

As can be seen from (2), the resulting point p'_b is in homogenous form, i.e. before we can use its coordinates, they need to be divided by w'.

Equation (1) allows you to transform the coordinates of each pixel from the original image to its destination coordinates in the target image. Replace the pixels in the target image with the corresponding pixels from your source image.