Exercise 1: Basic image processing and histograms

Machine perception

2017/2018

First create a working folder that will contain all the code for the course. In the folder create a separate folder for each exercise.

Create a folder `exercise1` that you will use during this exercise. Unpack the content of the `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 exercise has the amout of additional points written next to it.

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

Assignment 1: Basic image processing

The purpose of this assignment is to get familiar with the basic functionality of *Matlab/Octave*as well as using matrices to describe image information. In the assignment you will test instructions such as: `imread`, `imshow`, `imagesc`, `colormap`, and `imrotate`. Write the assignment to script file. Complete the following steps:

(a) Read the image from the file `umbrellas.jpg` and display it using the following snippet:

```
A = imread('umbrellas.jpg'); % Image A is in 8-bit format (uint8)
figure(1); clf; imagesc(A); % open figure window, clear, draw
figure(2); clf; imshow(A);
```

(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

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

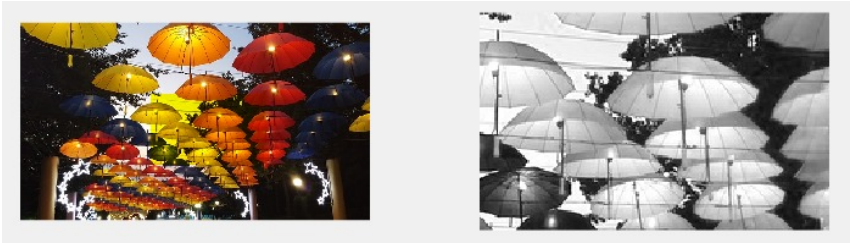
matrix, one needs to be careful about its type. The image is by default loaded as a matrix of type `uint8`, meaning that the pixel values are in range between 0 and 255. When summing up `uint8` values in *Matlab/Octave*, they *saturate on integer overflow*[2] — if the summed value exceeds 255, it will 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:

```
Ad = double(A); % division operation not defined for uint8
[h,w,d] = size(A) % output the size of the image (notice the absence of semicolon at the end of ↩
    the line)
A_gray = uint8((Ad(:,:,1) + Ad(:,:,2) + Ad(:,:,3)) / 3.0);
figure; imshow(A_gray); % change image to 8-bit
```

In the example above the image has been displayed using the default palette or color-map (depending on the version of your *Matlab/Octave* this can be a different colormap). A color map defines the colors that are used to display single-channel images. E.g. it can be used to display low values as dark blue and high values as red. Try changing the color map for the image using command `colormap`. Try pre-defined maps `jet`, `bone`, or `gray`.

(c) 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 (blue) color channel to 0 and displaying the modified image.

```
A1 = A;
A1(130:260,240:450,3) = 0 ;
figure;
subplot(1,2,1);
imshow(A1);
A2 = A(130:260,240:450, 1);
subplot(1,2,2);
imshow(A2);
```



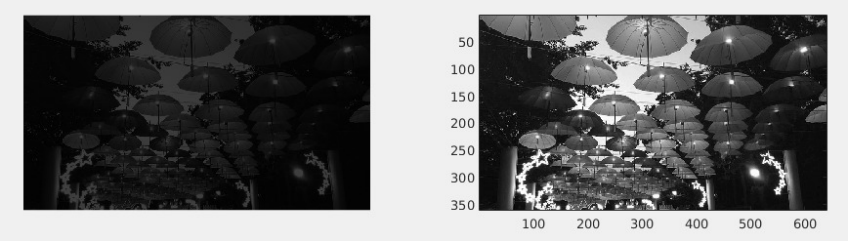
(d) Display a grayscale image that has a region negated (values are inverted).

```
A3 = A_gray;
A3(130:260,240:450) = 255 - A3(130:260,240:450) ;
figure;
imshow(A3);
```

(e) 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

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

floating point format. The highest possible grayscle level in this case is 255, and the lowest possible one is 0. Calculate a new image by multiplying the elements of the original image with appropriate factor (and then round the result). Choose the multiplication factor so that the resulting image will have the highest possible grayscale level 63 (the lowest remains 0). Display the image again, first by using the `imagesc` function and then by using the `imshow` function – do you notice the difference in the way the two functions display the image? Try to explain the difference based on the way the two functions display images (the description of this can be found in *Matlab/Octave* help).



Assignment 2: Thresholding and histograms

Thresholding an image is an operation that produces a binary image (mask) of the same size where the value of pixels is determined by whether the value of the corresponding pixels in the source image is greater or lower than the given threshold.

(a) Test the operation with a manually set threshold. Use the grayscale version of the image that you load from the file `bird.jpg`. Calculate and display a thresholded binary image, where value 1 denotes elements in the source image that have a value higher than 150 and 0 denotes elements that do not. Change the color-map to grayscale. Use the following snippet as a start:

```
A = rgb2gray(imread('bird.jpg'));
M = A > 150 ; % all elements with value > 150 are changed to 1, others are changed to 0
figure;
imagesc(M);
```

Experiment with different threshold values to obtain a reasonably good mask of the central object in the image.

(b) Setting a threshold manually can take some time, but for some types of images this can be done automatically with a bit of additional analysis of the image. For this we will use a really useful description called a *histogram*. Histograms are really useful and since we will be using them even more in the following exercises it is recommended to pay extra attention to how they are built. At this point we will look into building histograms for grayscale (single channel) images.

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.

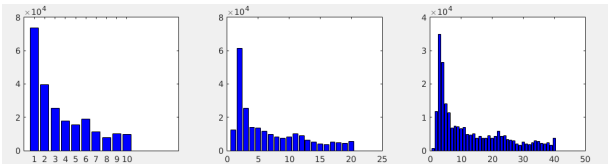
```
function [ H, bins ] = myhist(I , nbins)
I = reshape(I, 1, numel(I) ); % reshape image in 1D vector
H = zeros(1, nbins) ; % initialize the histogram
max_val_in = 255 ; % highest input value
min_val_in = 0 ; % lowest input value
max_val_out = nbins; % highest cell index
min_val_out = 1 ; % lowest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

(c) Histogram calculation is also implemented in *Matlab/Octave* in form of function `hist`, however, this function works in a bit different way. To try out the function, write the code below write to script file. 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.

```
I = double(rgb2gray(imread('umbrellas.jpg')));
P = I(:); % A handy way to turn 2D matrix into a vector of numbers
figure(1); clf;
bins = 10 ;
H = hist(P, bins);
subplot(1,3,1); bar(H, 'b');
bins = 20 ;
H = hist(P, bins);
subplot(1,3,2); bar(H, 'b');
bins = 40 ;
H = hist(P, bins);
subplot(1,3,3); bar(H, 'b');
```



(d) Visualize result of `myhist` and `hist` for other images of your choice and visualize the results. Notice that the result may be a bit different between the two functions. Do you know why? Hint: `myhist` assumes that it knows the highest and lowest values

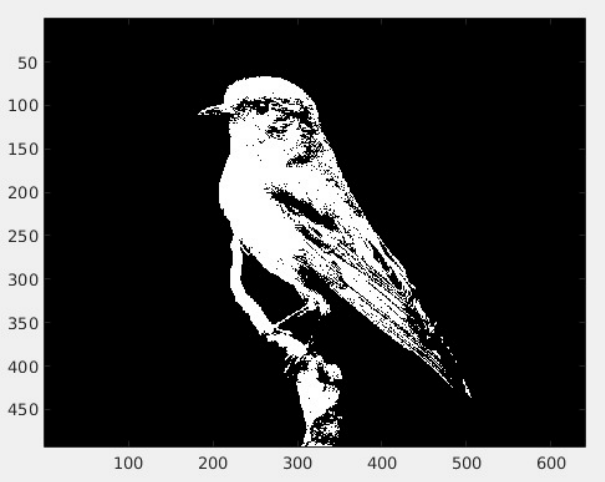
are known in advance since it operates on images of fixed range. On the other hand `hist` operates on general sequences of numbers, how would one determine the range in this case? Also, what is the meaning of values in histogram bins in one case and in the other one?

(e) ★ (10 points) Test `myhist` function on images (three or more) of the same scene in different lighting conditions. One way to do this is to capture several images using your web camera and turn ligts on and off. Visualize the histograms for all images for different number of bins and interpret the results.

(f) Using the knowledge about the construction of histograms you can now implement a simple automatic thresholding algorithm called Otsu's method[3] that works on bi-modal histograms and determines the threshold to best separate the two modes or classes in the image.

```
function threshold = otsu(I)
    nbins = 256;
    counts = myhist(I, nbins); % Obtain the histogram
    p = counts / sum(counts); % Normalize the histogram
    sigma = zeros(nbins, 1);
    for t = 1 : nbins
       qlow = sum(p(1:t));
       qhigh = sum(p(t + 1 : end));
       miu_L = sum(p(1:t) .* (1:t)) / qlow;
       miu_H = sum(p(t + 1 : end) .* (t + 1 : nbins)) ./ qhigh;
       sigma(t) = qlow * qhigh * (miu_L - miu_H) ^ 2;
    end
    [~, threshold] = max(sigma(:));
end
```

Test the automatic thresholding algorithm using the image loaded from `bird.jpg`. Convert the image to grayscale, calculate threshold using `otsu` function and apply the threshold to obtain a mask. Display the mask using `imagesc`.



Assignment 3: Morphological operations and regions

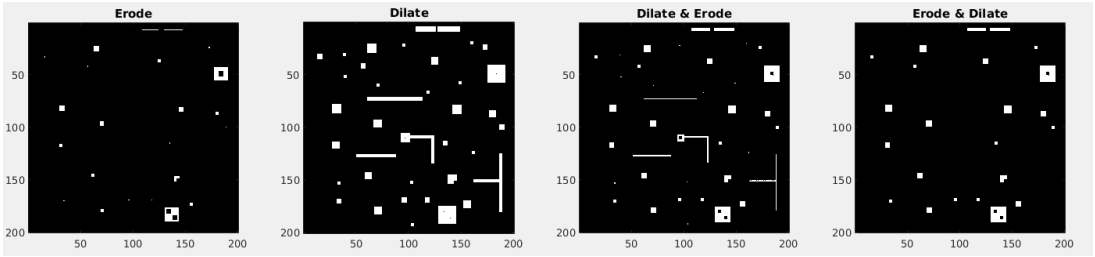
Thresholding can give you an initial mask of an object, but it is still a global technique that sometimes gives you artifacts in form of holes in the foreground object or unwanted

[3]Named after Nobuyuki Otsu.

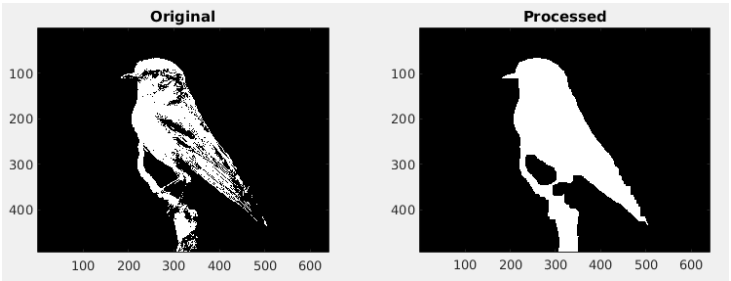
small regions around the object that you would like to remove from the mask before doing any further processing. This is where morphological operations come in handy. As you have seen at lectures these operations can be used to remove the artifacts if used correctly.

(a) We will look at two basic morphological operations: *dilate* and *erode* implemented in functions `imdilate` and `imerode`. Both operations can be combined into more complex operations. Experiment with the sequence of the two operations (just *erode* or *dilate*, *erode* then *dilate*, and *dilate* then *erode*) and display the result for all combinations. Also experiment with different structuring elements. Use the script below as a starting point.

```
M = logical(imread('mask.png')); % Load a synthetic mask
SE = ones(3); % 3x3 structuring element (also try out different sizes and shapes)
figure;
subplot(1, 4, 1);
imagesc(imerode(M, SE)); axis equal; axis tight; title('Erode');
subplot(1, 4, 2);
imagesc(imdilate(M, SE)); axis equal; axis tight; title('Dilate');
subplot(1, 4, 3);
imagesc(imerode(imdilate(M, SE), SE)); axis equal; axis tight; title('Dilate & Erode');
subplot(1, 4, 4);
imagesc(imdilate(imerode(M, SE), SE)); axis equal; axis tight; title('Erode & Dilate');
```



(b) Continuing from the mask that you have obtained at the end of the previous assignment, we will now try to improve the mask that was obtained using global thresholding. Implement a *closing* operation which is composed of two elementary operations: *dilate* and *erode* that use the same kernel. Test application of different structuring elements (different sizes) to obtain a clean mask as shown below.



(c) ★ (5 points) Create a script file `immask.m` and use it to implement function `immask` that can be given a three channel color image and a binary image of the same size and returns an image where pixel values are set to black color if their corresponding pixel in the binary image is 0. The function has to be written without explicit loops in the syntax. Hint: working with single channel images is easier than working with

all channels at once. You can then use function `cat` to combine them back together into a color image. Write a script that tests the result of the previous task: creating a mask using automatic thresholding, cleaning it up using morphological operations, and displaying a masked input color image.

(d) Now create a mask from the image in file `eagle.jpg` and visualize the result with `immask` (if you have completed the previous task, otherwise simply display the mask). Describe the cause of the inverted mask in this case. Propose a change to the code that will result in a mask that will show the object and not the background.

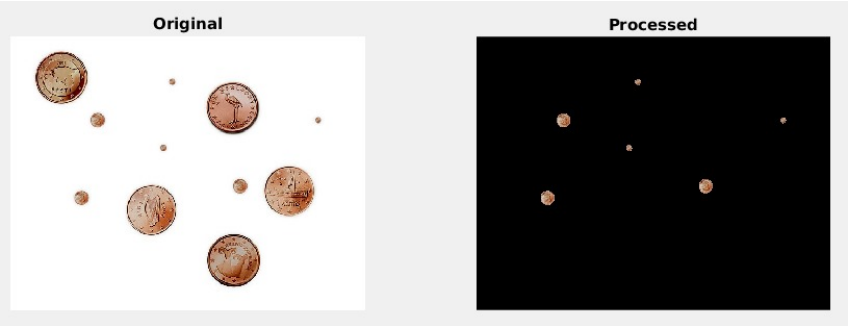
(e) Another way to process a mask is to extract individual connected components. Experiment with function `bwlabel` that returns a matrix with individual components of the mask marked with individual numbers. Write a script that loads image from file `coins.jpg`, obtains a mask, cleaning it up using morphological operations. Then use `bwlabel` to obtain the matrix of connected components and only keep components that contain not more than 700 pixels. Use the script below as a starting point.

```
I = imread('coins.jpg'); % Load a synthetic mask

% Compute mask M

L = bwlabel(M); % Use connected components algorithm to label all components
label_max = max(L(:)); % A trick to get all values present in matrix L
for i = 1:label_max
    if sum(L(:) == i) > 700 % Only process labels that have more than 700 pixels
        L(L == i) = 0;
    end;
end;
subplot(1, 2, 1);
imshow(I); title('Original');
subplot(1, 2, 2);
imshow(immask(I, L > 0)); title('Processed'); % Display the result (if you have not written immask↩
      function then simply display the mask)
colormap gray;
```

Experiment with the script to see how to isolate or remove a single component. Could you also achieve this kind of effect using morphological operations? Discuss which kind of sequence of erode and dilate together with per-pixel logical operations could you use.



(f) ★ (15 points) Write a script that loads image from file `candy.jpg` and lets user interactively select a point using function `ginput`. If the point is on one of the candy the system counts the number of candies of the same color. Segment the foreground using a threshold, process the mask with morphological operations to remove noise, determine connected components and extract the average color of each candy by computing the average color of its pixels. When a candy is clicked simply determine its color and filter out all other regions with similar color (you can use Euclidean distance and a threshold). Present the result by printing out the number of selected regions using function `text` as well as drawing markers on top of their centers using the integrated function `plot`. The number of points is dependent to the robustness of your solution.