RWTH AACHEN UNIVERSITY

Computer Vision
WS 2016/2017

Prof. Bastian Leibe<leibe@vision.rwth-aachen.de>
Stefan Breuers<breuers@vision.rwth-aachen.de>

Exercise 2: Thresholding, Morphology, Derivatives, and Edge Detection

due before 2016-11-14

Important information regarding the exercises:

- In the archive for this exercise you will find the functions `apply.m` that should be used for displaying your results. You should also use it to test your implementation and see if the results make sense. Answers are to be submitted within `answers.m` Do **not** modify the apply files in any way.
- Please do **not** include the data files in your submission!
- Please submit your code solution as a zip/tar.gz file named `mn1_mn2_mn3.{zip/tar.gz}` with your **matriculation numbers** (`mn`).
- Please submit your solutions via the L^{2}P system.

Please note:

- The exercise is not mandatory.
- There will be no corrections. If you want to verify your solutions, use the provided `apply` functions.
- Nevertheless, we encourage you to work on the exercises and present your solutions in the exercise class. For this regard the above submission rules.

Question 1: Histogramming and Thresholding

In the following, we provide a code framework. Fill in the holes in the Matlab code using the things you have learned in the lecture.

a) Implement a function which takes a grayvalue image and computes a grayvalue histogram. Do not use Matlab's built-in functions **`hist`**, `imhist` or `histc`. You can assume all pixel values fall inside the range [0,255]. Display the histogram with **`bar`**`(histogram)`.

```
function histogram = histogram(img)
```

Load the image "cells.png" and compute its grayvalue histogram. Use the built-in Matlab function **`cumsum`** to compute the cumulative histograms. Display both histograms and describe what you see.

b) Implement a function to binarize the input image `img` by thresholding at `T`. The function should return the thresholded image. Have a look at the histograms and choose a reasonable threshold. Display the thresholded image and describe how you chose the threshold.

```
function thresholded_image = thresh(img,T)
```

c) Write a function that implements Otsu's global threshold selection method to binarize the (grayvalue) input image `img`. Do not use Matlab's `graythresh` function.

```
function thresholded_image = thresh_otsu(img)
```

Implement a function to compute the weighted mean of a range of histogram cells from `rmin` to `rmax` and use it to speed up the computation.

```
function wmean = weighted_mean(h, rmin, rmax)
```

Display the thresholded image.

d) Write a function that computes the local mean and variance within the square image window of radius `r` (i.e. window size $2r + 1$) centered at pixel (`x1`,`y1`).

```
function [mean, variance] = get_window_mean_var(img, x1, y1, r)
```

Write a function that slides a square image window of radius `r` over each pixel of the input image and that applies Niblack's local threshold selection method to compute a local threshold for that pixel. The function should return the thresholded image. What does the result look like? Apply the function to the image `cells.png` with `r = 15` and `k = -0.2`. What do you observe?

```
function res = thresh_niblack(img, r, k)
```

Question 2: Morphological Operations

Try to improve the thresholding results from the previous section using morphological operators. Set up different structuring elements, e.g.

```
square_element = strel('square',5);    % 5-by-5 square
line_element   = strel('line',10,45); % line, length 10, angle 45
circle_element = strel('disk',1);      % disk, radius 1
```

and experiment with the functions `imdilate`, `imerode`, `imopen`, and `imclose`. Which combinations work best?

Question 3: Hu Moments (Bonus)

Write a simple function that is able to classify a given set of objects in a grayvalue image by using Hu Moments. In order to achieve this combine all the steps mentioned in lecture 2:

a) Use the images `pap1.gif`, `pap2.gif`, `pap3.gif`.
   1. Convert the images into binary form with Thresholding.
   2. Clean up the thresholded images with Morphological operations.
   3. Extract individual objects with connected components labeling, you can use the Matlab function `bwlabel`.

b) Use one image to learn the appearance of the pen and the piece of paper by describing them with Hu Moments and recognize them in the other images. Mark the position and name of classified objects and save your results in a new image (**`plot`**, **`text`**, `imwrite`).
   Then try this on some own grayvalue pictures (simply use your webcam or some digital camera). Which kinds of objects are easy and which are difficult to distinguish? How does the background influence this?

Question 4: Gaussian Filtering

a) In the following, you will implement a method which generates and applies a Gaussian filter for a given variance and number of samples. Start by writing a function which creates a 1D Gaussian from a given vector of integer indices $\mathtt{x} = [-\mathtt{w}, \ldots, \mathtt{w}]$:

$$\mathtt{G[i]} = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\mathtt{x[i]}^2}{2\sigma^2}\right) \tag{1}$$

where σ is the standard deviation.

```
function gaussian = gauss(x, sigma)
```

b) Use the above function to implement a function `gaussianfilter`, which first generates a Gaussian filter and then applies it to the given input image. The size of the filter should be $2 \cdot \lceil 3\sigma \rceil + 1$. Remember that the Gaussian is separable, *i.e.* that an equivalent 2D result can be obtained through a sequence of two 1D filtering operations. Do not use **`filter`**, **`filter2`**, **`conv2`** or `imfilter` in your implementation. However, you are allowed to use these functions in the following questions.

```
function outimg = gaussianfilter(inimg, sigma)
```

Matlab already provides a built-in function to construct a Gaussian filter.

```
filter_function = fspecial('gaussian', fsize, sigma);
out = imfilter(img, filter_function);
imshow(out);
```

Read the image "graf.png" and apply the filters with `sigma = 2.0`, `4.0`, and `8.0`. Again, choose `fsize` as $2 \cdot \lceil 3\sigma \rceil + 1$. What do you observe? Compare the result of this built-in filter with your own implementation by computing the difference image. Was your implementation correct?

Question 5: Image Derivatives

This exercise introduces image derivative operators.

a) Implement a function for creating a Gaussian derivative filter in 1D according to the following equation

$$\frac{d}{dx}G = \frac{d}{dx}\frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{x^2}{2\sigma^2}\right) \tag{2}$$

$$= -\frac{1}{\sqrt{2\pi}\sigma^3}x\exp\left(-\frac{x^2}{2\sigma^2}\right) \tag{3}$$

Your function should take a vector of integer values x and the standard deviation `sigma` as arguments.

```
function D = gaussdx(x, sigma)
```

The effect of applying a filter can be studied by observing its so-called *impulse response*. For this, create a test image in which only the central pixel has a non-zero value:

```
imgImp = zeros(25,25);
imgImp(13,13) = 255;
```

Now, create the following 1D filter kernels `gaussian` and `derivative`.

```
sigma = 6.0;
x = [floor(-3.0*sigma):round(3.0*floor)];
gaussian = gauss(x, sigma);
derivative = gaussdx(x, sigma);
```

What happens when you apply the following filter combinations? (For best visualization, display the result image with the **`imagesc`** command).

1. first `gaussian`, then `gaussian'`.
2. first `gaussian`, then `derivative'`.
3. first `derivative`, then `gaussian'`.
4. first `gaussian'`, then `derivative`.
5. first `derivative'`, then `gaussian`.

Describe your result.

b) Use the functions `gauss` and `gaussdx` directly in order to create a new function `gaussderiv` that returns the 2D Gaussian derivatives of an input image in x and y direction. Try the function on the given example images and describe your results.

```
function [imgDx,imgDy] = gaussderiv(img, sigma)
```

In a similar manner, create a new function `gaussderiv2` that returns the 2D second Gaussian derivatives $\frac{d^2}{dx^2}$, $\frac{d^2}{dxdy}$ and $\frac{d^2}{dy^2}$ of an input image. Try the function on the given example images and describe your results.

```
function [imgDxx, imgDxy, imgDyy] = gaussderiv2(img, sigma)
```

c) Create a new function `gradmag` that returns two images with the magnitude `imgMag` and orientation `imgDir` of the gradient for each pixel of the input image. Try the function on the given example images and describe your results.

```
function [imgMag, imgDir] = gradmag(img, sigma)
```

d) Create a new function `laplace` that returns an image with the Laplacian-of-Gaussian for each pixel of the input image. Try the function on the given example images and describe your results.

```
function imgLap = laplace(img, sigma)
```

Question 6: Edge Detection

a) Next, we will create a very simple edge detector. Write a function `getedges` that returns a binary image `imgEdge` from an input image where the color of each pixel p is selected as follows (for a given threshold `theta`):

$$p \quad = \quad \begin{cases} 1, & \text{if } |\text{grad}(\texttt{img})|\,(p) \geq \theta \\ 0, & \text{else} \end{cases} \tag{4}$$

```
function imgEdge = getedges(img, sigma, theta)
```

Experiment with the function `getedges` on the example images. Try to get good edge images for different values of `sigma`. What difficulties do you observe? (Note: it may pay off to look at the contents of the magnitude image in order to get a feeling for suitable values of `theta`).

b) Using the above function, returned edges are still several pixels wide. In practice, this is often not desired. Create a function `getedges2` that extends `getedges` by using the following function to suppress non-maximum points along the gradient direction:

```
function imgMax = nonmaxsupcanny(imgMag, imgDir)
        h = size(imgMag, 1);
        w = size(imgMag, 2);

        imgMax = zeros(h, w);

        offx = [-1 -1  0  1  1  1  0 -1 -1];
        offy = [ 0 -1 -1 -1  0  1  1  1  0];

        for y = 2:h-1
                for x = 2:w-1
                        % look up the orientation at that pixel
                        dir = imgDir(y, x);

                        % look up the neighboring pixels in direction
                            of the gradient
                        idx = floor(((dir + pi)/pi) * 4 + 0.5) + 1;

                        % suppress all non-maximum points
                        % (note: this simplified code does not
                            interpolate between neighboring pixels!)
                        if( (imgMag(y,x) > imgMag(y+offy(idx),x+offx(
                            idx))) && ...
                             (imgMag(y,x) > imgMag(y-offy(idx),x-
                                 offx(idx))) )
                             imgMax(y, x) = imgMag(y, x);
                        end
                end
        end
end
```

Note that this simplified code does not interpolate between the neighboring pixel values in order to look up the real magnitude samples along the gradient direction. This interpolation is crucial to obtain the necessary robustness for an actual implementation. Here it was left out for better readability, since the interpolation involves some extra effort in order to deal with all special cases (*e.g.* exactly horizontal or vertical gradients). If you feel motivated, you can try to add this step to make the function more robust.

Another problem is that suitable values for `theta` may vary substantially between images. Extend the function `getedges2` such that the threshold $\theta \in [0, 1]$ is defined relative to the maximal gradient magnitude value in the image. Try your function on the given example images and describe your results.

c) The function `getedges` you implemented is a simplified version of the Canny edge detection pipeline. The main step that is still missing is the edge following with hysteresis thresholding. The idea here is that instead of applying a single threshold over the entire image, the edge detector works with two different thresholds `theta_high` and `theta_low`. It starts with an edge pixel with a value above `theta_high` and then follows the contour in the direction orthogonal to the gradient until the pixel value falls below `theta_low`. Each pixel visited along the way is labeled as an edge. The procedure is repeated until no further pixel above `theta_high` remains.

Try writing a function `my_canny` that implements this procedure. Don't worry about efficiency for the moment. The following hints may help you.

- You can create a boolean array for already visited and yet-to-visit image pixels. Since we are not interested in pixels below the low threshold you can mark them as visited. In another boolean array you can flag the pixels that serve as starting points for line following.

```
visited = imgMax < theta_low;
high = imgMax >= theta_high;
```

- In order to avoid having to pass large arrays to each function, you can declare them as global.

```
global visited;
gloabl img_res;
```

- You can also avoid having to deal with special cases along the image borders by creating a 1-pixel boundary where the `visited` flag is set to `true`.

```
visited(:,1) = true;
visited(:,end) = true;
visited(1,:) = true;
visited(end,:) = true;
```

- The actual edge following part is most easily implemented as a recursive procedure. In most cases, you will have the option to choose between several possible continuation points. Again, the easiest way is to try all of them in sequence (or even all 8 neighbors) and let the recursive procedure (together with the `visited` flags) do the rest.

```
function [] = follow_edge(x, y)
        global visited
        global img_res;

        visited(x,y) = true;
        img_res(x,y) = 1;

        offx = [1 1 0 -1 -1 -1  0  1];
        offy = [0 1 1  1  0 -1 -1 -1];
        for i=1:8,
                idx_x = x + offx(i);
                idx_y = y + offy(i);
                if visited(idx_x, idx_y) == false
                        follow_edge(idx_x, idx_y);
                end
        end
end
```

```
function edge_image = my_canny(img, sigma, theta_low, theta_high)
```

Matlab already provides the built-in function `edge(I,'canny')` that implements the Canny edge detector. Try this function on the provided example images and compare the results to those of your implementation. What do you observe?

d) (Bonus) This solution gives better results, but its results still depend strongly on the maximal gradient magnitude value in the image. For a cleaner solution, we want to adapt the threshold to the distribution of all gradient magnitude values. Extend the function `getedges` by the following steps in order to do this:

1. Perform non-maximum suppression on the gradient magnitude image as shown above.
2. Transform the result image into a vector using the following command:

```
pix = reshape(imgMag, 1, size(imgMag, 1) * size(imgMag, 2));
```

3. Build a histogram of the remaining gradient magnitude values using the Matlab function **`hist`** on the vector `pix`.
4. Compute the cumulative sum over the histogram (except for the first cell) using the function **`cumsum`**.
5. The last cell of the cumulative histogram now contains the total number of edge pixels in the image. Compute the desired number of edge pixels `numedge` as the percentage `theta` of this value.
6. Find the threshold for which the cumulative histogram contains the value `numedge`.

Please turn in your solutions including all relevant files before 2016-11-14!