

Exercise 5: Robust detection of 2D objects

Created by: Tim Jerman | <http://lit.fe.uni-lj.si/RV> | Homework deadline: April 19/20, 2016

Instructions

During this exercise you will get familiar with basic methods for the detection of corners and edges in 2D images. Corners are prominent structure elements of an object in an image which can be used in a range of more complex applications: object tracking in a sequence of image frames, computation of the transformation between different views of the same object, calibration of an optical system using reference points, and many others. On the other hand, the detection of edges in a 2D image, can be used as the initial step in the detection of complex objects.

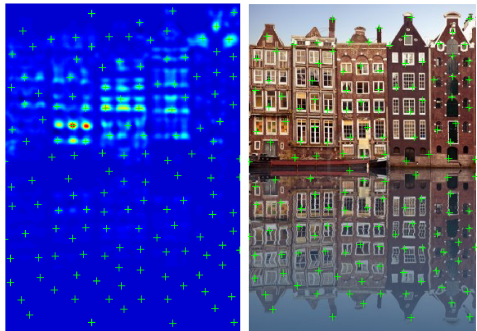
A corner is defined as an intersection of two edges in an image, where an edge can be seen as a prominent local change in grayscale values. Edges can be extracted by computing the derivatives of the image. The first derivative of an arbitrary 2D function $f(x, y)$ can be written in a vector form as:

$$\nabla f(x, y) = \vec{g}(x, y) = \begin{bmatrix} g_x(x, y) \\ g_y(x, y) \end{bmatrix} = \begin{bmatrix} \partial f(x, y) / \partial x \\ \partial f(x, y) / \partial y \end{bmatrix}.$$

A vector image of a gradient $\vec{g}(x, y)$ at each point (x, y) contains a vector of length G rotated by angle α with respect to axis x :

$$\alpha(x, y) = \arctan \frac{g_y(x, y)}{g_x(x, y)}, \quad G(x, y) = \sqrt{g_x^2(x, y) + g_y^2(x, y)}.$$

Gradient $\vec{g}(x, y)$ points in the direction of the largest change of the function $f(x, y)$ and, consequently, is perpendicular to the given edge in an the image. The two gradient components $g_x(x, y)$ and $g_y(x, y)$, which represent the two partial derivatives, are computed by filtering a given digital image using one of the standard kernels, such as **Sobel's**. The vector image $\vec{g}(x, y)$ of derivatives is computed by convolving the input image with two Sobel's kernels oriented in two different directions.

	Sobel kernels for first derivatives	Harris corner detector									
x -axis:	<table border="1" style="margin: auto;"> <tr><td>-1</td><td>0</td><td>1</td></tr> <tr><td>-2</td><td>0</td><td>2</td></tr> <tr><td>-1</td><td>0</td><td>1</td></tr> </table>	-1	0	1	-2	0	2	-1	0	1	$Q(x, y) :$ 
-1	0	1									
-2	0	2									
-1	0	1									
y -axis:	<table border="1" style="margin: auto;"> <tr><td>-1</td><td>-2</td><td>-1</td></tr> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>2</td><td>1</td></tr> </table>	-1	-2	-1	0	0	0	1	2	1	
-1	-2	-1									
0	0	0									
1	2	1									

While the edges in images have a large gradient pointing in the direction perpendicular to the edge, the gradient at the corners is usually large and comparable in multiple directions. Therefore, the majority of the corner detection methods is based on the comparison between the gradient magnitudes in image directions x and y . A commonly used method in practice is the **Harris corner detector**, where for each point (x, y) a local structrness matrix M is computed:

$$M(x, y) = \begin{bmatrix} A(x, y) & C(x, y) \\ C(x, y) & B(x, y) \end{bmatrix} * N(x, y | \sigma) = \begin{bmatrix} g_x^2(x, y) & g_x(x, y)g_y(x, y) \\ g_x(x, y)g_y(x, y) & g_y^2(x, y) \end{bmatrix} * N(x, y | \sigma)$$

Components $A(x, y)$, $B(x, y)$ and $C(x, y)$ of the local structrness matrix M represent the derivatives, which are smoothed with a Gaussian kernel $N(x, y | \sigma)$ using a 2D discrete convolution $(*)$. The score

function for the detection of corners for each image point (x, y) is determined by analyzing the relationship between the two eigenvalues λ_1 and λ_2 of matrix M :

$$\lambda_{1,2} = \frac{\text{trace}(M)}{2} \pm \sqrt{\left(\frac{\text{trace}(M)}{2}\right)^2 - \det(M)},$$

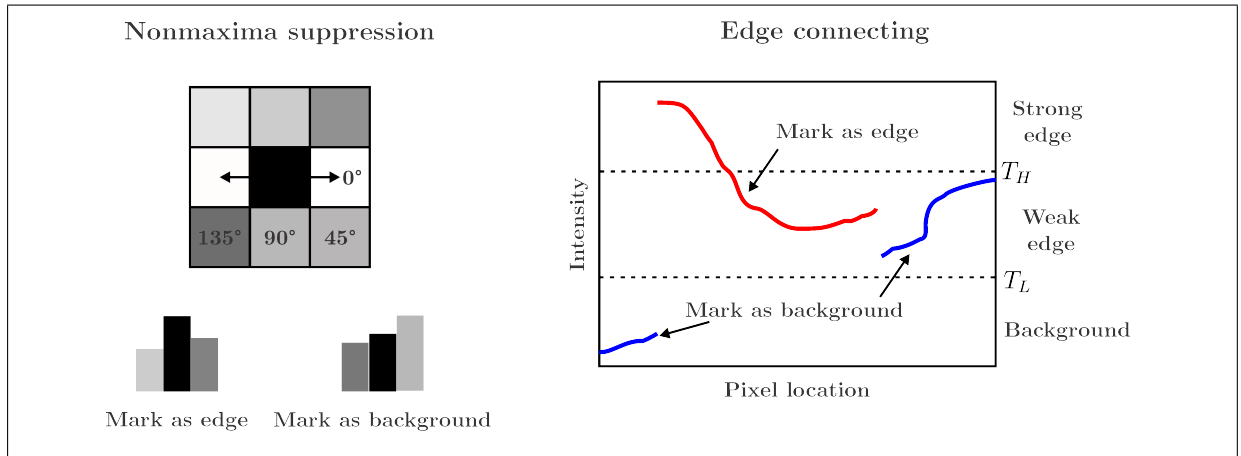
where $\text{trace}(\cdot)$ and $\det(\cdot)$ are functions defining the trace and determinant of matrix M . In image areas with constant grayscale values, $M = 0$ and $\lambda_1 = \lambda_2 = 0$. On the other hand, the eigenvalues are $\lambda_1 > 0$ and $\lambda_2 = 0$ for a monotonically increasing grayscale values in a single direction, which is expected for an edge. Therefore, eigenvalues λ_1 and λ_2 encode the prominence of the edge, whereas the corresponding eigenvectors point in the direction of the edge. The response of the two eigenvalues in the corners is similar $|\lambda_1| \approx |\lambda_2|$ and large in the direction of both eigenvectors $|\lambda_{1,2}| \gg 0$. The smaller the difference between the two eigenvalues, the higher is the prominence of the corner. Based on these conditions a score function for detecting corners is determined as:

$$Q_H(x, y) = \lambda_1 \lambda_2 - \kappa(\lambda_1 + \lambda_2)^2 = \det(M) - \kappa(\text{trace}(M))^2.$$

Parameter κ determines the sensitivity of the detector, and is selected on the interval $[0, \frac{1}{4}]$. The initial corner points are detected as the local maximums in image $Q_H(x, y)$, which are further reduced by thresholding $Q(x, y) > T_{min}$ using a threshold T_{min} between 10^4 do 10^6 .

Detection of edges is one of the most important methods in the field of image processing and analysis, and from all of them, **Canny's edge detector** is one of the most widely used. The algorithm for computing the edges using Canny's edge detector can be summarized in four basic steps:

1. Image smoothing using a Gaussian filter
2. Computation of the gradient magnitude and orientation
3. Suppression of non maximal values
4. Double thresholding and edge connecting



The selection of the smoothing and gradient filtering methods is arbitrary. The computed gradient magnitude image contains high responses around the edges, which need to be thinned to represent the exact edge. This is done by nonmaxima suppression, where pixels with a locally non maximal value in the discrete orientation ($0^\circ, 45^\circ, 90^\circ$ in 135°) of the gradient are set to zero. As the last step we do the thresholding and connection of the edges. The thresholding is done using two thresholds, where the upper threshold T_H determines the strong edges $r_H(x, y)$ which are indirectly retained. This are the edge pixels with the gradient magnitude of $G(x, y) > T_H$. On the other hand, the lower threshold T_L determines the weak edge points $r_L(x, y)$, which are later transformed into strong edges or into background by edge connection: any weak edge point $r_L(x, y)$ ($T_L < G(x, y) \leq T_H$), which is a neighbor of a strong edge point $r_H(x, y)$ is automatically converted into a strong edge point.

During this exercise you will write functions for image filtering based on 2D discrete convolution for the purpose of smoothing and sharpening the grayscale and color images and functions for image

interpolation and decimation for the purpose of magnification and minimization of grayscale and color images. Load the *RGB* color image `slika.jpg` into Spyder–Python environment and convert to grayscale image using $S = 0,299R + 0,587G + 0,114B$. You will use the color and grayscale image to verify the functions in the lab and homework assignments.

During this exercise you will write functions for computing the gradient of 2D images using 2D discrete convolution, response of Harris corner detector, and local maxima point extraction. You will also learn how the analysis of the eigenvalues of the structrness matrix M can be used for designing filters for the enhancement of tubular structures in 2D images. You will later reuse the function for the computation of the gradient to detect the edges in an image, where you will first suppress nonmaximal values and connect weak edges with the strong ones.

1. Write a function for computing the first order derivative of the input grayscale image `iImage`:

```
def imageGradient( iImage ):
    return oGx, oGy
```

The function returns the two images of the partial derivatives `oGx` in `oGy`, which are equally sized as the input image `iImage`. Load the *RGB* image `slika1.jpg` into Spyder–Python and convert it into a grayscale image using equation $S = 0,299R + 0,587G + 0,114B$. Compute the first order derivatives of this image and display them at each point as a vector using the function `matplotlib.pyplot.quiver()`.

2. Write a function to compute the response of the Harris corner detector on the input grayscale image `iImage`:

```
def responseHarris( iImage, iKappa, iSigma ):
    return oQH
```

where `iKappa` is the sensitivity of the corner detector (κ), and `iSigma` is the standard deviation of the Gaussian function used for smoothing the response. The function returns the image `oQH`, which represents the response of the corner detector. Test the function using different sets of parameters `iKappa` and `iSigma`.

3. Write a function for extracting the local maximums in an arbitrary 2D array `iArray`:

```
def findLocalMax( iArray ):
    return oLocalMax
```

The function returns a matrix `oLocalMax` of size $2 \times n$ containing (x, y) coordinates of the extracted n local maximums. The local maximums in a 2D array are those points with the value greater than the value of the 8 connecting neighbors. As a simplification do not consider the points on the edge or outside of the input image.

4. Write a function for detecting the corners in an input grayscale image `iImage`:

```
def cornerHarris( iImage, iKappa, iTmin ):
    return oCorners
```

where `iKappa` is the sensitivity of the detector and `iTmin` is the threshold for the extracted maximums. The function returns the matrix `oCorners` of dimension $2 \times n$ that contains (x, y) coordinates of the n detected corners. The function should contain the calls to previously developed functions `responseHarris()` and `findLocalMax()`. Set the parameter `iSigma` of the function `responseHarris()` to 3.

5. Write a function for the computation of the magnitude `oGAbs` and orientation `oGPhi` of the gradient of the smoothed grayscale image `iImage`:

```
def smoothImageGradientAbs( iImage, iSigma ):
    return oGAbs, oGPhi
```

where `iSigma` is the standard deviation of the Gaussian function used for the smoothing of the input image. Test the function on the *RGB* image `slika3.jpg`, which was previously transformed into a grayscale image. Show the image of the gradient magnitude for the value of `iSigma` set to 1.

6. Write a function for the suppression of nonmaxima values by employing the input images `oGAbs` and `oGPhi`:

```
def nonMaximaSuppression( iGAbs, iGPhi ):
    return oEdge
```

The function returns an image `oEdge`, where the elements with value of 1 represent the nonconnected edges in the input image. Test the function on the image gradients, which were previously computed using the function `smoothImageGradientAbs`.

7. Write a function for detecting the edges in the input grayscale image `iImage` by employing the Canny algorithm:

```
def edgeCanny( iImage, iSigma, iThreshold=0 ):
    return oEdge
```

where `iThreshold` is the threshold used to distinguish the strong edges, The function should be constructed from the previously developed functions `smoothImageGradientAbs` and `nonMaximaSuppression`, with the addition of thresholding: we retain only those detected edges which have the gradient magnitude `oGAbs` higher than the threshold `iThreshold`. The image of the gradient magnitude should be first normalized on the interval $[0, 1]$. Compare the returned image of the detected edges for different values of `iSigma` and `iThreshold`.

Homework Assignments

Homework report in the form of a Python script entitled `NameSurname_Exercise5.py` should execute the requested computations and function calls and display requested figures and/or graphs. It is your responsibility to load library packages and provide supporting scripts such that the script is fully functional and that your results are reproducible. The code should execute in a block-wise manner (e.g. `## Assignment 1`), one block per each assignment, while the answers to questions should be written in the corresponding block in the form of a comment (e.g. `# Answer: ...`).

1. Write the images of the first order derivatives of the nonsmoothed grayscale image at each point (x, y) as the orientation $\alpha(x, y)$ and magnitude $G(x, y)$ of the gradient. Display α and G as two grayscale images. For assignments 1-3 use `slika1.jpg` as the input image.
2. Compute the second order derivatives $f(x, y)$ of the grayscale image: $\partial^2 f(x, y)/\partial x^2$, $\partial^2 f(x, y)/\partial y^2$, $\partial^2 f(x, y)/\partial x \partial y$ in $\partial^2 f(x, y)/\partial y \partial x$. Display this derivatives as a grayscale image.
3. Extend the function `cornerHarris()` by adding an additional input parameter `iMinDist`. The added parameter corresponds to the minimum distance between two detected corners. Reduce the number of corners so that each pair of corners on the image has a distance of at least `iMinDist`. If the distance between a pair of corners is less than `iMinDist` retain the corner with a higher response of the Harris detector. Display the grayscale input image with the superimposed detected corners for two selections of minimum distances: `iMinDist = 0` and `iMinDist = 20`.
4. A different selection of the score function similar to the one of the Harris corner detector, which is based on the analysis of the eigenvalues λ_1 and λ_2 of matrix M , can be used for the enhancement of tubular structures in the image as for example the vessels in the retina. One of a such score function is the *linear anisotropy*:

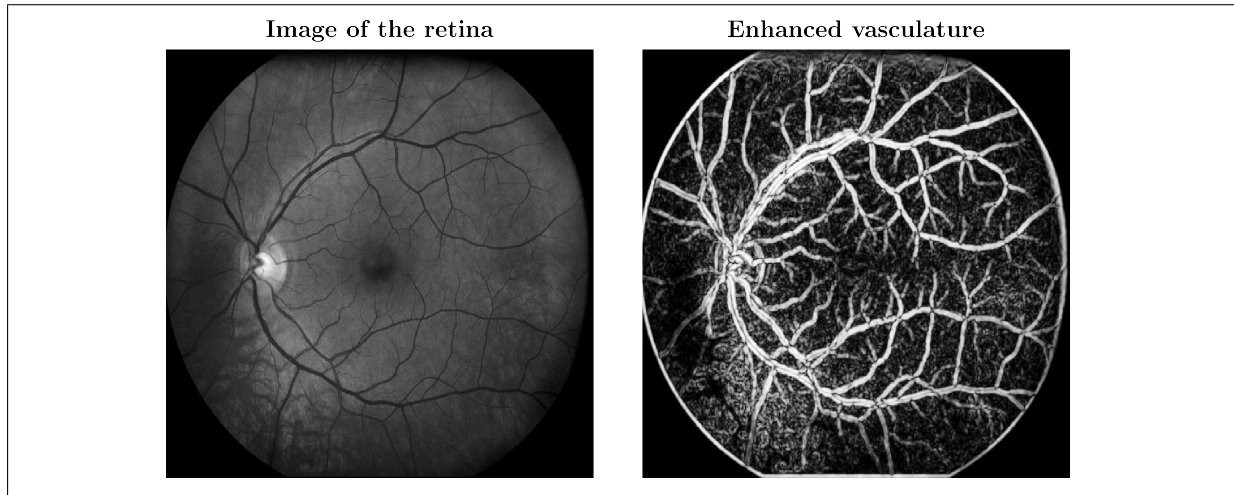
$$Q_{LA} = \frac{\lambda_1 - \lambda_2}{\lambda_1 + \lambda_2 + \beta}, \quad (1)$$

where $\beta > 0$ is an arbitrary constant. Write a function for the enhancement of tubular structures in the input grayscale image `iImage`:

```
def emphasizeLinear( iImage, iSigma, iBeta ):
    return oQLA
```

where `iSigma` is the standard deviation of the Gaussian function used for smoothing the elements of matrix M , and `iBeta` is an arbitrary constant.

Load the *RGB* color image `slika2.jpg` into Spyder–Python and convert it into a grayscale image using equation $S = 0,299R + 0,587G + 0,114B$. Use the function `emphasizeLinear()` for enhancing the vessels in the image by finding the optimal value of parameters `iSigma` and `iBeta`. Display the optimal response `oQLA` of the enhanced tubular structures and write the optimal set of parameters `iSigma` and `iBeta`.



5. The current function `edgeCanny` does not contain the last and essential step of edge connecting. Extend the function `edgeCanny` by including this step. For this purpose write a function `connectEdge` with input parameters `iEdge`, the image of edges with nonmaxima suppression, the gradient magnitude image `iGAbs`, and an array `iThreshold` containing a lower and upper threshold $[T_L, T_H]$:

```
def connectEdge( iEdge, iGAbs, iThreshold ):
    return oEdge
```

The function should be built by following the next steps:

- (a) Normalize the gradient magnitude on the interval $[0, 1]$.
- (b) Separately find those points in the edge image `iEdge` that have a value of 1 and have the values of normalized magnitude: i) higher than the upper threshold `iThreshold[1]`, and ii) lower or equal to `iThreshold[1]` and at the same time higher than `iThreshold[0]`. Mark the former as `edgeStrongIdx` and the later as `edgeWeakIdx`.
- (c) In a new empty image `oEdge` set to 1 all points that were assigned as a strong edge (points in `edgeStrongIdx`).
- (d) Edge connecting: iterate through all weak edges in the image (`edgeWeakIdx`); if any of the neighboring points is a strong edge, convert the current weak edge to a strong edge. You can simplify the function by not iterating over the points that are on the image edges. These are points with coordinates: `x==0` or `y==0` or `x==iImage.shape[1]-1` or `y==iImage.shape[0]-1`.
- (e) Repeat the step in (d) until there are still any weak edges that can be converted to the strong edges. After there are no more conversions that can be done return the image of the edges `oEdge`.

Integrate the created function into the function `edgeCanny` by replacing the current single threshold thresholding. Compare the previous and the new extended implementation on the grayscale image of the *RGB* image `slika3.jpg`. In the first case use the threshold `iThreshold=0.2`, while in the second case use `iThreshold=[0.2,0.2]`. In both cases use the same value of `iSigma=1`. Display both edge images and verify that your result is equal for both cases.

6. Compare the results of the improved edge detection function `edgeCanny` for the next selections of thresholds: `iThreshold=[0.2,0.2]`, `iThreshold=[0.05,0.05]`, `iThreshold=[0.05,0.2]`. Again set the value of `iSigma` to 1. Describe how the selection of the two thresholds affects the detection of the edges.

Image of tablets



Detected edges

