Exercise 3: Edges and Hough transform

Machine perception

2018/2019

Create a folder `exercise3` that you will use during this exercise. Unpack the content of the `exercise3.zip` that you can download from the course webpage to the folder. Save the solutions for the assignments as the *Matlab/Octave* scripts to `exercise3` 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.

Introduction

For additional explanation of the theory, required for the following assignment, check the slides from the lectures as well as scientific literature on these topics [2, 5]. A version of the book by Forsyth in Ponce can also be found on-line [3] – the related theory is located in chapters 8 and 15.

Assignment 1: Image derivatives

In the next two assignments we will deal with the problem of detecting edges in images. Edges can be determined by analyzing local changes of grayscale levels. Mathematically this means that we are computing *image derivatives*. The downside of a simple image derivation at a certain point in an image is that it is very sensitive to the presence of image noise which also affects estimation of derivatives. It is therefore common to soften the image beforehand with a narrow filter, $I_b(x, y) = G(x, y) * I(x, y)$ and only then calculate a derivative.

A Gaussian filter is usually used to soften an image. As we will use partial derivatives in the future, we will first look at the decomposition of the partial derivative of the Gaussian kernel. A 2D Gaussian kernel can be written as a product of two 1D kernels as

$$G(x, y) = g(x)g(y), \tag{1}$$

therefore image filtering for image $I(x, y)$ can be formulated as

$$I_b(x, y) = g(x) * g(y) * I(x, y). \tag{2}$$

Taking into account the property of the convolution that $\frac{d}{dx}(g * f) = (\frac{d}{dx}g) * f$, we can write a partial derivative of the *smoothed* image with respect to x can be written as

$$I_x(x,y) = \frac{\delta}{\delta x}[g(x) * g(y) * I(x,y)] = \frac{d}{dx}g(x) * [g(y) * I(x,y)]. \quad (3)$$

This means that the input image can be first filtered with a Gaussian kernel with respect to y and then filter the result with the derivative of the Gaussian kernel with respect to x. In a similar manner we can also define the second partial derivative with respect to x, however, we have to remember to always filter the image before we perform derivation. The second derivative with respect to x is therefore defined as a partial derivative of already derived image:

$$I_{xx}(x,y) = \frac{\delta}{\delta x}[g(x) * g(y) * I_x(x,y)] = \frac{d}{dx}g(x) * [g(y) * I_x(x,y)]. \quad (4)$$

(a) Follow the equations above and define the equations used to compute first and second derivative with respect to y, $I_y(x,y)$, $I_{yy}(x,y)$, as well as the mixed derivative $I_{xy}(x,y)$.

(b) Implement a function that computes a derivative of a 1D Gaussian kernel. The formula for a Gaussian kernel is (you can verify this if you want):

$$\frac{d}{dx}g(x) = \frac{d}{dx}\frac{1}{\sqrt{2\pi}\sigma}\exp(-\frac{x^2}{2\sigma^2}) \quad (5)$$

$$= -\frac{1}{\sqrt{2\pi}\sigma^3}x\exp(-\frac{x^2}{2\sigma^2}). \quad (6)$$

Implement the kernel in function `gaussdx(sigma)`. As we have (in the previous exercise) normalized a final discrete Gaussian kernel it is also recommended to normalize its derivative. However, the derivative of a Gaussian function is an odd function so we have to be careful. Normalize the kernel in a way that the sum of its absolute values is 1. This means that you have to divide each value by $\sum abs(g_x(x))$.

The properties of the filter can be analyzed using a *impulse response function* for $f(x,y)$, that is defined as a convolution of a Dirac $\delta(x,y)$ function with the kernel $f(x,y)$: $f(x,y) * \delta(x,y)$. A discrete version of the Dirac function is constructed as a finite image that has all but the central element set to 0, while the center element contains a non-zero (possibly very high) value:

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

Now, generate the following 1D kernels G and D:

```
sigma = 6.0;
G = gauss(sigma);
D = gaussdx(sigma);
```

What happens if you apply the following operations to the image `impulse`?:

(a) First convolution with G and then convolution with G^T.

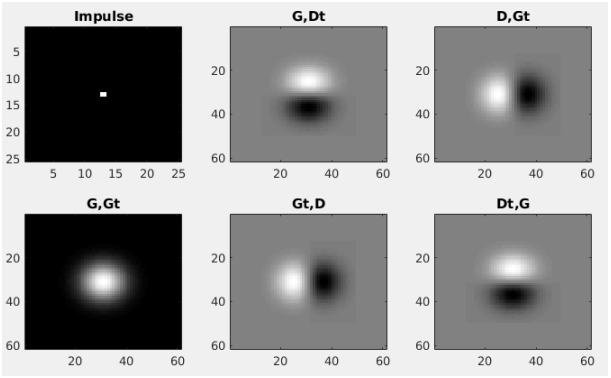
(b) First convolution with G and then convolution with D^T.

(c) First convolution with D and then convolution with G^T.

(d) First convolution with G^T and then convolution with D.

(e) First convolution with D^T and then convolution with G.

is the order of the operations important? Display images of the impulse responses for separate combinations of operations. Use `imagesc` for better visualization.



(c) Implement a function that uses functions `gauss` and `gaussdx` to compute both partial derivatives of a given image, with respect po x and with respect to y.

```
function [Ix, Iy] = image_derivatives(I, sigma)
 ...
```

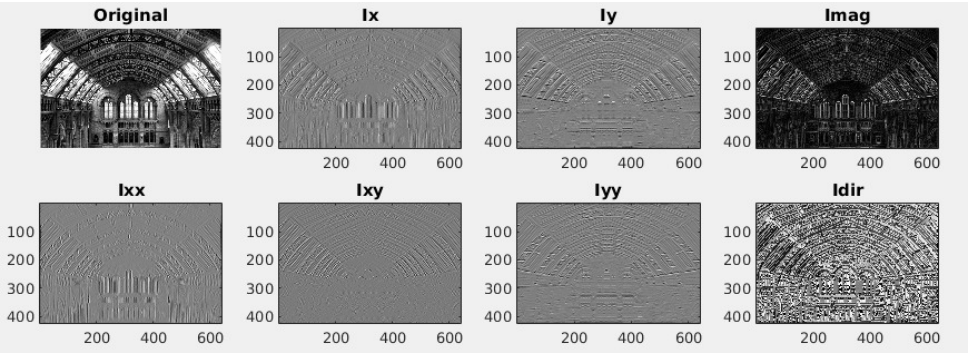
Similarly as above, implement function that returns partial second order derivatives of a given image.

```
function [Ixx, Iyy, Ixy] = image_derivatives2(I, sigma)
 ...
```

Implement function `gradient_magnitude` that accepts a grayscale input image `I`, and returns a matrix of derivative magnitudes `Imag` and a matrix of derivative angles `Idir`. Magnitudes are calculated as $m(x,y) = \sqrt{(I_x(x,y)^2 + I_y(x,y)^2)}$, angles are calculated as $\phi(x,y) = \arctan(I_y(x,y)/I_x(x,y))$. Hint: (i) use matrix calculation for better performance (ii) when calculating angles, you can avoid division by zero by using the `atan2` function.

```
function [Imag, Idir] = gradient_magnitude(I, sigma)
 ...
```

Test all three functions by visualizing their results on image from file `museum.jpg`.



(d) ★ (10 points) Gradient information is often used in image recognition. Extend your image retrieval from the previous exercise to use a simple gradient-based feature instead of color histograms. To obtain this feature, compute gradient magnitudes and angles, then divide the image in a 8×8 grid and for each cell of the grid compute a 8 bin histogram of gradient magnitudes with respect to gradient angles (quantize the angles into 8 values and for each pixel in the cell add the value of the gradient magnitude to the bin that is specified by the corresponding angle). Combine all the histograms to get a feature. Hint: you can also use integrated functions like `accumarray` to speed up feature calculation. Test the new feature on the image database that you have received in the previous exercise and compare it to the color histogram based retrieval.

Assignment 2: Edges in images

(a) Canny edge detector is one of the most widely-used detectors of edges in images. In this assignment you will implement the first steps of the Canny's algorithm that will be extended in throughout the assignment For start create function `findedges` following the example below.

```
function [Ie] = findedges(I, sigma, theta)
 ...
```

Extend the function with code that computes magnitudes of gradients `Imag` and returns a binary matrix `Ie` that shows the magnitudes that are higher than a specified threshold value `theta`:

$$I_e(x,y) = \begin{cases} 1 & ; I_{mag}(x,y) \geq \theta \\ 0 & ; otherwise \end{cases} \tag{7}$$

Test the function with the image `museum.png` and display the results for different values of the parameter `theta`. Can you set the parameter so that all the edges in the image are clearly visible?

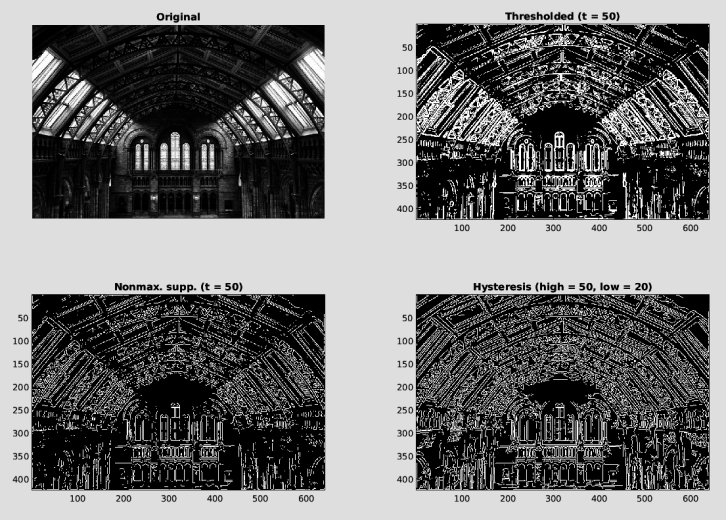
(b) The function above returns a first approximation of the detected edges. Unfortunately, many of these edges are more than one pixel wide while we would usually like to have a one pixel wide edge detections. Use the function below within the function `findedges` to filter `Imag` before the thresholding operation. The function is a edge-specific local non-maxima suppression technique that searches the neighborhood of each pixel's magnitude `Imag` with respect to the local edge angle `Idir` and sets the magnitude value to 0 if it does not contain a maximum value in respect to the neighborhood magnitudes. Go through the code and analyze each line. Test the modified function `findedges` and visualize the result.

```
function Imax = nonmaxima_suppression_line(Imag, Idir)
[h, w] = size(imgMag);
Imax = 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 = 1:h
   for x = 1:w
```

```
        dir = Idir(y, x); % check pixel orientation
        idx = round(((dir + pi) / pi) * 4) + 1; % map orientation to the lookup table
        y1 = y + offy(idx) ; x1 = x + offx(idx);
        y2 = y - offy(idx) ; x2 = x - offx(idx);
        x1 = max([1, x1]); x1 = min([w, x1]); % constrain coordinates
        y1 = max([1, y1]); y1 = min([h, y1]);
        x2 = max([1, x2]); x2 = min([w, x2]);
        y2 = max([1, y2]); y2 = min([h, y2]);
        if((Imag(y, x) >= Imag(y1, x1))&&(Imag(y, x) >= Imag(y2, x2))) % check if local maxima
            Imax(y, x) = Imag(y, x) ;
        end
    end
```

(c) ★ (10 points) Non-maxima suppression algorithm can be implemented using matrix operations that results in significant performance improvements. Hint: function `circshift` can shift the entire matrix for a given number of elements. Using this function we can make a quick comparison of all elements with their corresponding top neighbors using `I == circshift(I, [1, 0])` (we have to be careful with the border elements, however, there elements have handled separately in any case). Any innovative solution to this task will be accepted if it avoids explicit loops whenever possible. Demonstrate the new algorithm by comparison with the reference algorithm, you have to show result equality for all non-border elements and significantly increased computational performance.

(d) ★ (10 points) The last step of the Canny's algorithm is edge tracking using hysteresis thresholding. Replace the normal thresholding at the end of function `findedges` with hysteresis thresholding. This step is at first look difficult to implement, however, in *Matlab/Octave* we can accomplish in a very compact way using functions like `bwlabel`, `unique`, and `ismember`. Hint: in the edge magnitude map we are looking for connected components in which all pixel values are above t_{low} and at least one value is also above I_{high}, where $t_{low} < t_{high}$. Any innovative solution to this task will be accepted if it avoids explicit loops whenever possible.



Assignment 3: Detecting lines

In this assignment we will look at the Hough algorithm, in particular the variation of the algorithm that is used to detect lines in an image. For more information about the theory

look at the lecture slides as well as the literature [2], and web applets that demonstrate the Hough transform, e.g. [1, 4].

We have a point in the image $p_0 = (x_0, y_0)$. If we know that the equation of a line is $y = mx + c$, which are all the lines that are running through the point p_0? The answer is simple: all the lines, whose parameters m and c correspond to the equation $y_0 = mx_0 + c$. If we fix the values (x_0, y_0), then the variable parameters again describe a line, however, this time the line is in the (m, c) space that we also call the parameter space. If we consider a new point $p_1 = (x_1, y_1)$, this new point also has a line in the (m, c) space. This line crosses the p_0 line in a point (m', n'). The point (m', c') then defines a line in (x, y) space that connects the points p_0 and p_1.

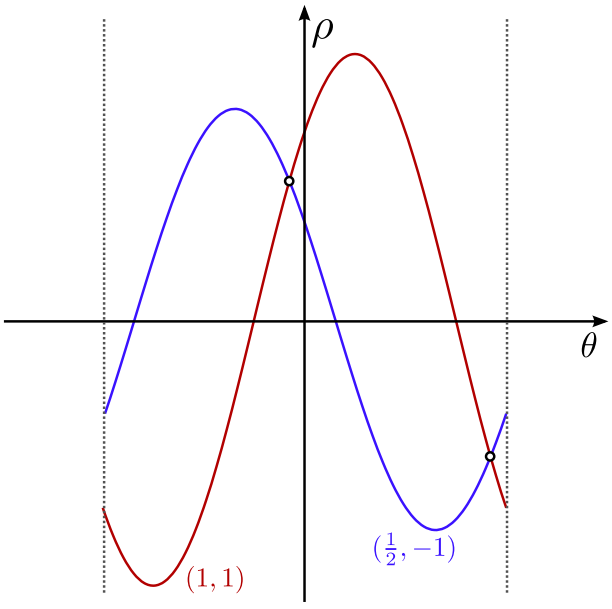
Question: Analytically solve the problem by using Hough transfom: In 2D space you are given four points $(0, 0)$, $(1, 1)$, $(1, 0)$, $(2, 2)$. Define the equations of the lines that run through at least two of these points.

If we want to find all the lines in an image using the Hough approach in a program, we have to proceed as described. The parameter space (m, c) is first quantized as an *accumulator* matrix. For each edge pixel we, *draw* a corresponding line in the (m, c) space in an additive manner (increase the value by 1). All the image elements that lie on the same line in the input image will generate lines in the (m, c) space that will cross in the same point and therefore increase the value of the same accumulator cell. This means that a local maxima in the (m, c) space define the lines in the input image that contain a lot of the detected edge pixels.

In real scenarios the $y = mx + c$ formulation of a line is inefficient in our case. This is in particular apparent in the case of vertical lines, there m becomes infinite. This problem can be simply avoided by a different line parametrization, for example using the *polar coordinates*. In this case an equation of a line looks like

$$x\cos(\theta) + y\sin(\theta) = \rho. \tag{8}$$

The algorithm remains more or less the same, the only difference is that a point in a (x, y) space generates a sinusoid in the (θ, ρ) space. For points $(1, 1)$ and $(\frac{1}{2}, -1)$ the corresponding curves in the parameter space are shown in the figure below.

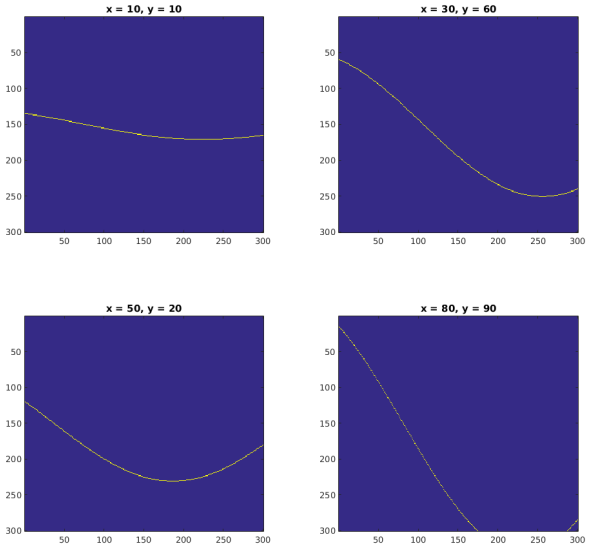


(a) To better understand how the Hough algorithm works first look at the code below that fills the accumulator array with a curve for a single point.

```
bins_theta = 300; bins_rho = 300; % Resolution of the accumulator array
max_rho = 100; % Usually the diagonal of the image
val_theta = (linspace(-90, 90, bins_theta) / 180) * pi; % Values of theta are known
val_rho = linspace(-max_rho, max_rho, bins_rho);
A = zeros(bins_rho, bins_theta);

% for point at (50, 90)
x = 50;
y = 90;
rho = x * cos(val_theta) + y * sin(val_theta); % compute rho for all thetas
bin_rho = round(((rho + max_rho) / (2 * max_rho)) * length(val_rho)); % Compute bins for rho
for i = 1:bins_theta % Go over all the points
    if bin_rho(i) > 0 && bin_rho(i) <= bins_rho % Mandatory out-of-bounds check
        A(bin_rho(i), i) = A(bin_rho(i), i) + 1; % Increment the accumulator cells
    end;
end;
imagesc(A); % Display status of the accumulator
```

First change the position of the edge and observe the changes of the resulting curve.



(b) In the following tasks we will gradually implement the Hough transform for line detection. You will implement the algorithm as a *Matlab/Octave* function that will serve as a main function of the algorithm:

```
function [out_ro, out_theta] = hough_find_lines(Ie, bins_rho, bins_theta, threshold)
 ...
```

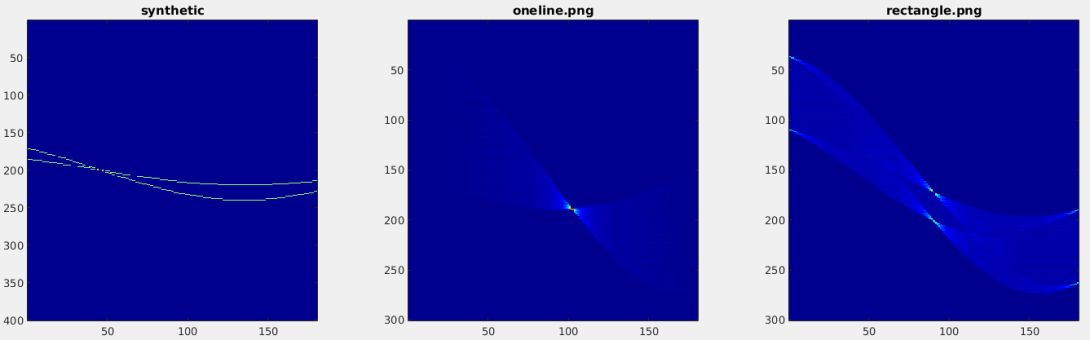
Create an accumulator matrix `A` for parameter space (ρ, θ). θ is defined in the interval from $-\pi/2$ to $\pi/2$, ρ is defined on the interval from $-D$ to D, where D is the length of the image diagonal. Parameters `bins_rho` and `bins_theta` define the number of cells (resolution) for each of the parameters ρ and θ. Initialize the accumulator by setting all cells to 0.

For each edge pixel in the image generate a curve in the (ρ, θ) space using the equation (8) for all possible values of θ and increase the values of the corresponding cells in `A`. Display the resulting accumulator matrix using the `imagesc` function. You

can test the algorithm first on a synthetic image of edges, generated by the script below:

```
E = zeros(100); % 100 x 100 pixel image
E(10, 10) = 1;
E(20, 20) = 1; % Set threshold to 2 to obtain line that travels through both points.
```

Then test the algorithm on two synthetic images, `oneline.png` and `rectangle.png`. Obtain an edge map for each image using function `findedges` and run your implementation of Hough algorithm on the edge maps.

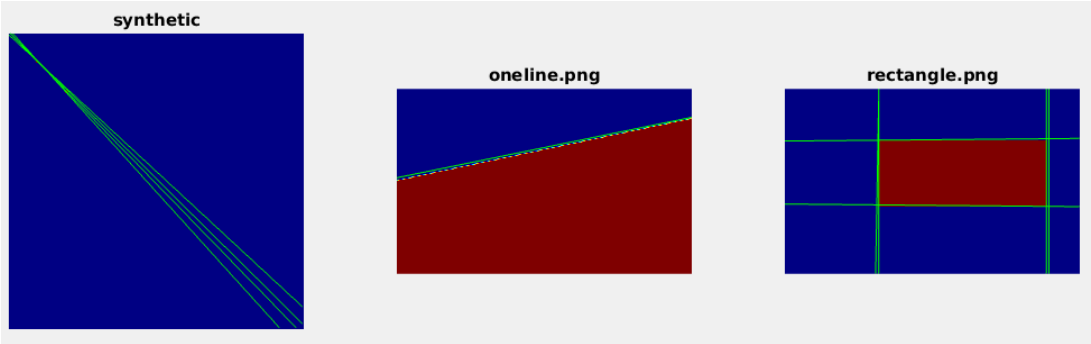


(c) Notice that the noise and the quantization cause the curves in the accumulator do not cross in one single cell but rather in multiple neighborhood cells. Therefore implement a `nonmaxima_suppression_box` function, that will set all the cells that are not local maxima (consider the neighboring 8 cells) to 0. Display the modified accumulator.

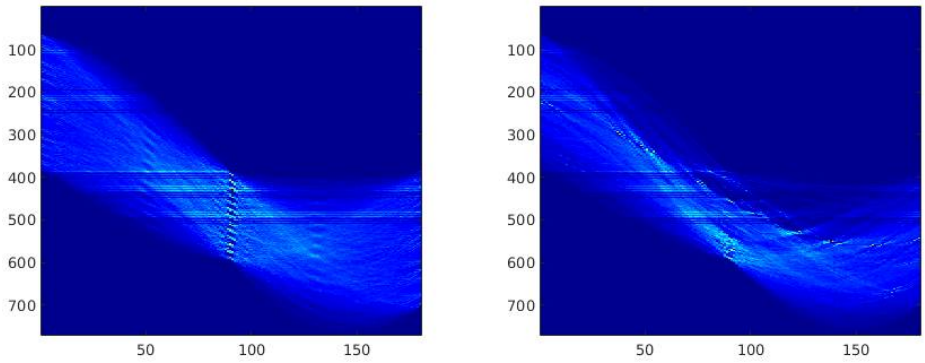
```
function B = nonmaxima_suppression_box(A)
 ...
```

(d) ★ (5 points) You have probably implemented the accumulator population and the nonmaxima suppression algorithms with nested loops. Try to reorganize the algorithms in a way that avoids explicit pixel iteration as much as possible. Consider the very useful functions `find`, `sub2ind` and `ind2sub`. Check the help section for these functions. Hint: especially for lower number of edge pixels iterating through their indices is much faster than by iterating through the entire image and checking the pixels if they are edge or not. In case of nonmaxima suppression you can use function `ordfilt2` that you can use to get the highest value in a certain region.

(e) Search the parameter space and extract all the parameter pairs (ρ, θ) whose corresponding accumulator cell value is greater than a specified threshold `threshold`. Draw the lines that correspond to the parameter pairs using the `hough_draw_lines` function that you can find in the supplementary material.



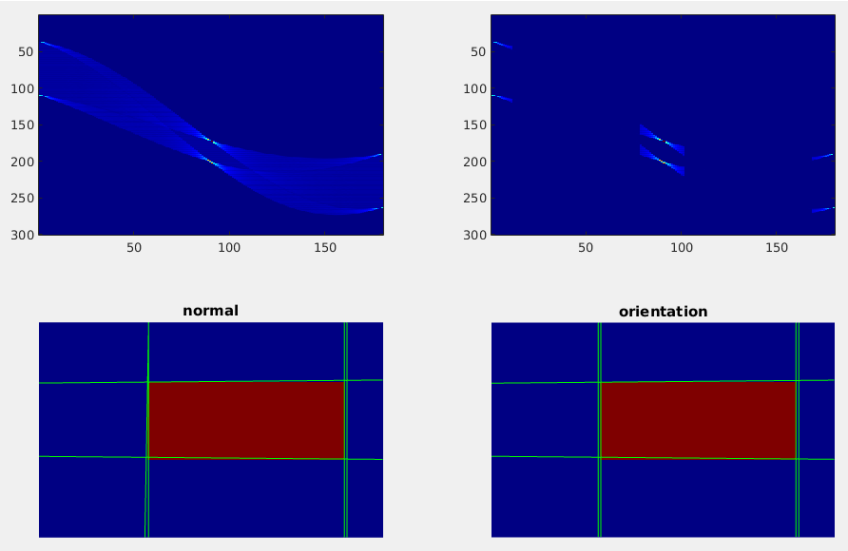
(f) Read the image from files `bricks.jpg` and `pier.jpg`. Change the image to grayscale, detect edges using the `findedges`, that you have implemented previously. Then detect lines using your algorithm. As the results will likely depend on the number of pixels that vote for specific cell and this depends on the size of the image and the resolution of the accumulator, try sorting the pairs by their corresponding cell values in descending order and only select the top $n = 10$ lines. Use function `sort` to perform sorting. Display the results and experiment with parameters of Hough algorithm as well as the edge detection algorithm, e.g. try changing the number of cells in the accumulator or σ parameter in edge detection to obtain results that are similar or better to the ones shown on the image below.



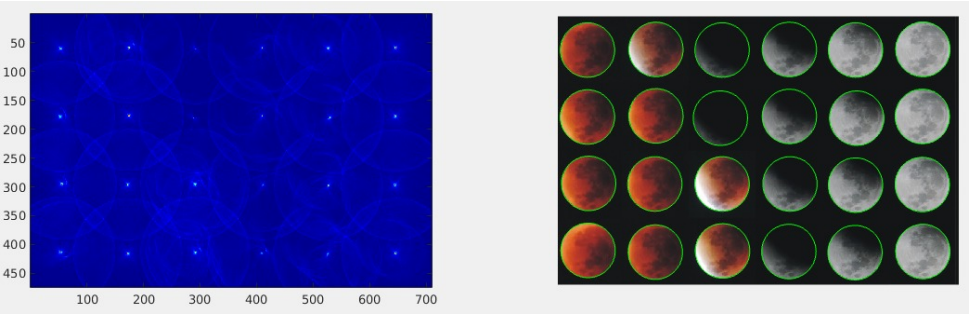
(g) ★ (10 points) A problem of the Hough transform is that we need a new dimension for each additional parameter in the model, which makes the execution slow for more complex models. We can avoid such parameters if we can reduce the parameter space, e.g. by introducing domain knowledge. Recall from the previous assignment that we can get the local gradient angle besides its magnitude. This angle is perpendicular

to the change and can be used to limit the scope of the parameter θ for a specific edge point. We therefore do not have to increase the values of the cells of the entire range of θ (calculate multiple values of ρ), but can use the local angle and only work with a single (ρ, θ) pair for each edge point.

Copy your implementation of the line detector to a new function and modify the algorithm that it also accepts the `Idir` matrix of edge angles. Note that the `Idir` values were probably calculated using the `atan2(dy, dx)` function that returns the values between $-\pi$ and π. You have to adjust the angles so that they are within the $[-\pi/2, \pi/2]$ interval. Test the modified function on several images and compare the results with the original implementation.



(h) ★ (10 points) Implement a Hough transform that detects circles of a fixed radius. You can test the algorithm on image `eclipse.jpg`. Try using radius somewhere between 45 and 50 pixels. You can use function `hough_draw_circles` to visualize the result.



(i) ★ (10 points) Not all lines can accumulate the same number of votes, e.g. if the image is not rectangular candidates along the longer dimension are in better position. Extend your algorithm so that it normalizes the number of votes according to the maximum number of votes possible for a given line (how many pixels does a line cover along its crossing of the image). Demonstrate the difference on some non-rectangular images where the difference can be demonstrated clearly.

References

[1] C. Brechbühler. Interactive hough transform. http://users.cs.cf.ac.uk/Paul.Rosin/CM0311/dual2/hough.htm.

[2] D. A. Forsyth and J. Ponce. *Computer Vision: A Modern Approach.* Prentice Hall, 2002.

[3] D. A. Forsyth and J. Ponce. Computer vision: A modern approach (online version). http://www.cs.washington.edu/education/courses/cse455/02wi/readings/book-7-revised-a-indx.pdf, 2003.

[4] M. A. Schulze. Circular hough transform. http://www.markschulze.net/java/hough/.

[5] R. E. Woods, R. C. Gonzalez, and P. A. Wintz. *Digital Image Processing, 3 ed.* Pearson Education, 2010.