

Assignment 3: Edges and Hough transform

General instructions: The assignment is composed of the compulsory tasks and optional tasks (denoted by \star). To approach the assignment defense, you are required to complete at least the compulsory parts. Successfully defending the compulsory part will give you a maximum 75 points out of 100. At the defense, the obligatory tasks **must** be implemented correctly and completely. If your implementation is incomplete or has major errors, you will not be able to successfully defend the assignment. You can choose arbitrarily among the optional tasks to reach the 100 points. The number of points for an optional task is given in the task description. It is possible to obtain more than 100 points on individual assignment. However, the maximum overall points obtained from the 6 assignments is 600 points.

Required formating and submission: Create a folder `assignment3` that you will use during this assignment. Unpack the content of the `assignment3.zip` that you can download from the course webpage to the folder. Save the solutions for the assignments as Python scripts to `assignment3` folder. In order to complete the assignment you have to present your solutions to the teaching assistant. Some assignments contain questions that require sketching, writing or manual calculation. Write these answers down and bring them to the assignment defense as well. The code must be submitted on the e-classroom **before** the defense. Submit the code as .py source files (not Jupyter notebooks). If you have more than one source file, submit a zip of .py files.

COMMON ERRORS AND DEBUG IDEAS

- check your x and y coordinates
- check your data type: float, uint8
- check your data range: [0,255], [0,1]
- perform simple checks (synthetic data examples)

Introduction

This assignment will have you construct approaches for edge detection from simple image derivatives, then the edge images will be further used to detect simple structures (lines, circles) in the images. For additional explanation of the theory required for the assignment, check the slides from the lectures as well as scientific literature on these topics [1, 3]. A version of the book by Forsyth in Ponce can also be found online [2] – the related theory is located in chapters 5 and 10.

Exercise 1: Image derivatives

The first and the second exercise will deal with the problem of detecting edges in images. A way of detecting edges is by analyzing local changes of grayscale levels. Mathematically, this means that we are computing *image derivatives*. The downside of an image derivative at a certain point in the image is that it can be sensitive to image noise. Thus, it is common to soften the image beforehand with a narrow filter $I_b(x, y) = G(x, y) * I(x, y)$ and only then calculate the derivative.

Usually, a Gaussian filter is used to smooth the image. As we will use partial derivatives in the following exercises, we will first look at the decomposition of the partial derivative of the Gaussian kernel. A 2-D Gaussian kernel can be written as a product of two 1-D kernels as:

$$G(x, y) = g(x)g(y), \quad (1)$$

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

$$I_b(x, y) = g(x) * g(y) * I(x, y). \quad (2)$$

Taking into account the associative property of the convolution $\frac{d}{dx}(g * f) = (\frac{d}{dx}g) * f$, we can write a partial derivative of the *smoothed* image with respect to x 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 . Similarly, we can 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 derive the equations used to compute first and second derivatives 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 the derivative of a 1-D Gaussian kernel. The formula for the derivative of the Gaussian kernel is:

$$\begin{aligned} \frac{d}{dx}g(x) &= \frac{d}{dx} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{x^2}{2\sigma^2}\right) \\ &= -\frac{1}{\sqrt{2\pi}\sigma^3}x \exp\left(-\frac{x^2}{2\sigma^2}\right). \end{aligned} \quad (5)$$

Implement the function `gaussdx(sigma)` that works the same as function `gauss()` from the previous assignment. Don't forget to normalize the kernel. Be careful as the derivative is an odd function, so a simple sum will not do. Instead normalize the kernel by dividing the values such that the sum of absolute values is 1. Effectively, you have to divide each value by $\sum abs(g_x(x))$.

- (c) The properties of the filter can be analyzed by using an *impulse response function*. This is performed as a convolution of the filter with a Dirac delta function. The discrete version of the Dirac function is constructed as a finite image that has all elements set to 0, except the central element, which is set to a high value (e.g. 1).

```
impulse = np.zeros((50, 50))
impulse[25, 25] = 1
```

Generate a 1-D Gaussian kernel G and a Gaussian derivative kernel D .

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

- (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 operations important? Display the images of the impulse responses for different combinations of operations.

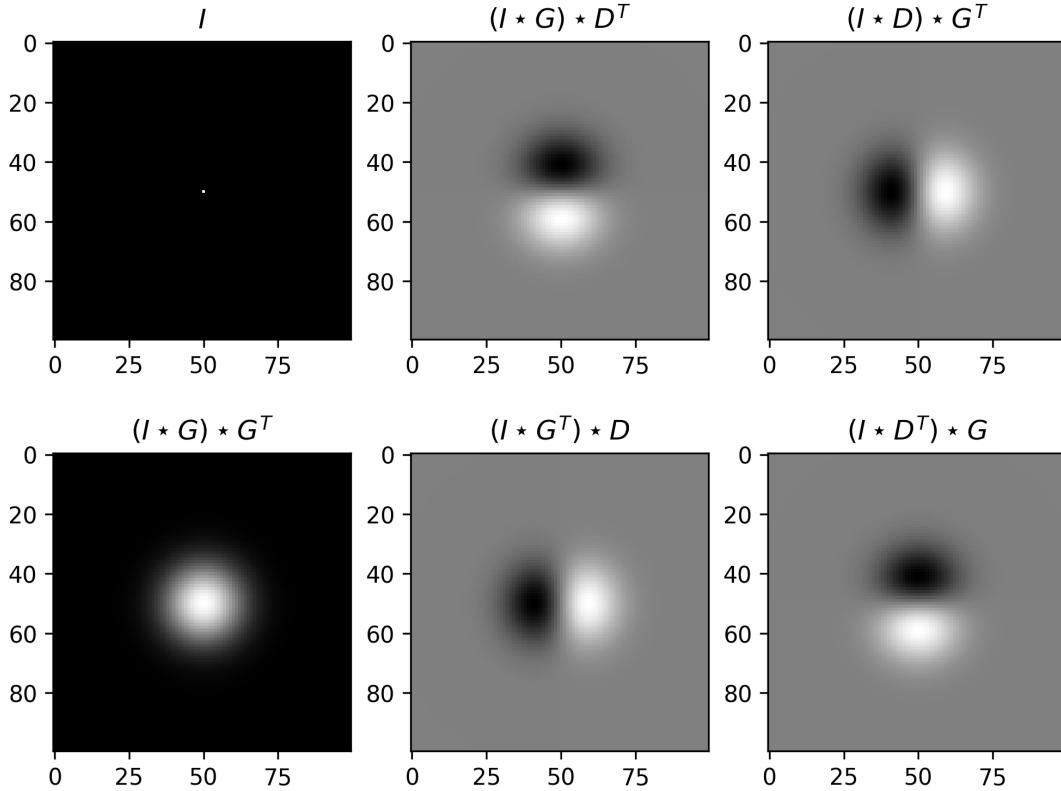


Figure 1: 1D convolution with Gaussian and Gaussian derivative kernel applied on the impulse image in different ordering.

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

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

Additionally, implement the function `gradient_magnitude()` that accepts a grayscale image I and returns both derivative magnitudes and derivative angles. Magnitude is calculated as $m(x, y) = \sqrt{(I_x(x, y)^2 + I_y(x, y)^2)}$ and angles are calculated as $\phi(x, y) = \arctan(I_y(x, y)/I_x(x, y))$

Hint: Use function `np.arctan2()` to avoid division by zero for calculating the arc tangent function.

Use all the implemented functions on the same image and display the results in the same window.

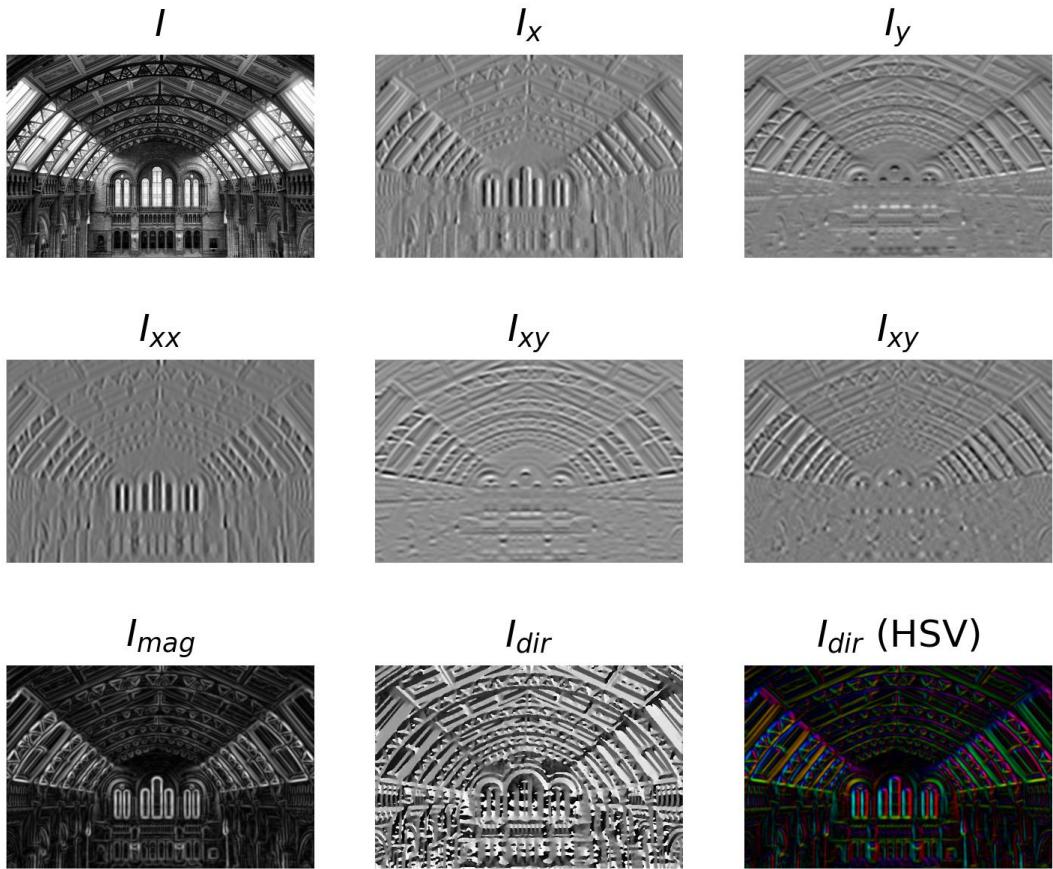


Figure 2: The results of first and second order derivatives on image `museum.jpg`, as well as the gradient direction and magnitude. In the bottom right are the gradient directions depicted with HSV color, weighted by the magnitude.

- (e) ★ (15 points) Gradient information is often used in image recognition. Extend your image retrieval system from the previous assignment to use a simple gradient-based feature instead of color histograms. To calculate this feature, compute gradient magnitudes and angles for the entire image, then divide the image in a 8×8 grid. For each cell of the grid compute a 8 bin histogram of gradient magnitudes with respect to gradient angle (quantize the angles into 8 values, then for each pixel of the cell, add the value of the gradient to the bin specified by the corresponding angle). Combine all the histograms to get a single 1-D feature for every image. Test

the new feature on the image database from the previous assignment. Compare the new results to the color histogram based retrieval.

Exercise 2: Edges in images

One of the most widely used edge detector algorithms is Canny edge detector. In this exercise, you will implement parts of Canny's algorithm.

- (a) Firstly, create a function `finedges()` that accepts an image I , and the parameters `sigma` and `theta`.

The function should create a grayscale matrix I_e that keeps pixels higher than threshold `theta` and sets others to zero:

$$I_e(x, y) = \begin{cases} I_{mag}(x, y) & ; I_{mag}(x, y) \geq \vartheta \\ 0 & ; \text{otherwise} \end{cases} \quad (6)$$

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) Using magnitude produces only a first approximation of detected edges. Unfortunately, these are often wide, and we would like to only return edges one pixel wide. Therefore, you will implement non-maxima suppression based on the image derivative magnitudes and angles. Iterate through all the edge pixels and for each search its 8-neighborhood. Check the neighboring pixels parallel to the gradient direction, and set the current pixel to 0 if it is not the largest in the neighborhood (based on gradient magnitude). You only need to compute the comparison to actual pixels, using interpolation for subpixel accuracy is not required.
- (c) ★ (10 points) The final step of Canny's algorithm is edge tracking by hysteresis. Add the final step after performing non-maxima suppression along edges. Hysteresis uses two thresholds $t_{low} < t_{high}$, keeps all pixels above t_{high} and discards all pixels below t_{low} . The pixels between the thresholds are kept only if they are connected to a pixel above t_{high} .

Hint: Since we are looking for connected components containing at least one pixel above t_{high} , you could use something like `cv2.connectedComponentsWithStats` to extract them. Try to avoid explicit `for` loops as much as possible.

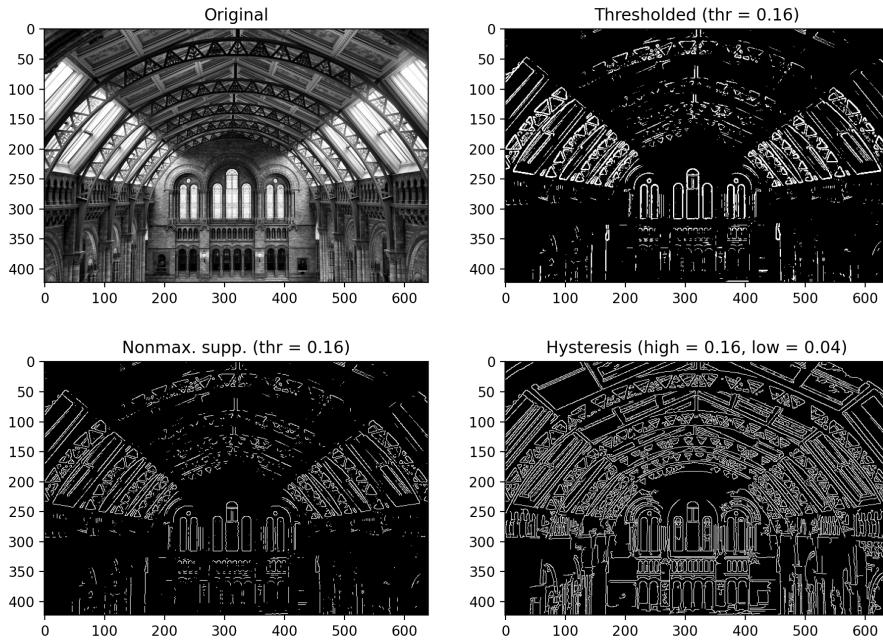


Figure 3: Intermediate steps of the Canny algorithm applied on image `museum.jpg`.

Exercise 3: Detecting lines

In this exercise, 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 [1], and a [web applet](#) that demonstrates the Hough transform.

We have a point in the image $p_0 = (x_0, y_0)$. If we know that the standard equation of a line is $y = mx + c$, which are all the lines that are running through the point p_0 ? The answer is simply: 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 at a point (m', c') . 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 transform: 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.

The Hough transform approaches finding lines from a voting perspective. Each of the edge points casts votes for all the lines that can pass through it. By accumulating the votes corresponding to edge points in an accumulator array, the cells with the highest number of votes will correspond to the lines present in the input image. However, the line equation $y = mx + c$ is not appropriate for this approach. We want to represent all line angles, but for vertical lines, the parameter m tends to infinity. Thus, we choose a different

line parametrization using *polar coordinates*. A line in polar coordinates is expressed very similarly to a single point, in this case the equation is:

$$x \cos(\vartheta) + y \sin(\vartheta) = \rho. \quad (7)$$

The parameters ρ and ϑ represent the parameters of the line in Hough parametric space. The parameter ρ represents the perpendicular distance from the line to the origin, while parameter ϑ represents the angle between the x axis and the perpendicular line. In this formulation, each line in the image space has a corresponding point in the parameter space. Conversely, all the different lines passing through a single point in the image space correspond to a sinusoid curve in the Hough space. Figure 4 displays the sinusoids corresponding to points $(1, 1)$ and $(\frac{1}{2}, -1)$. Since two 2D points only define one straight line, how can you explain the two intersections in the parameter space plot?

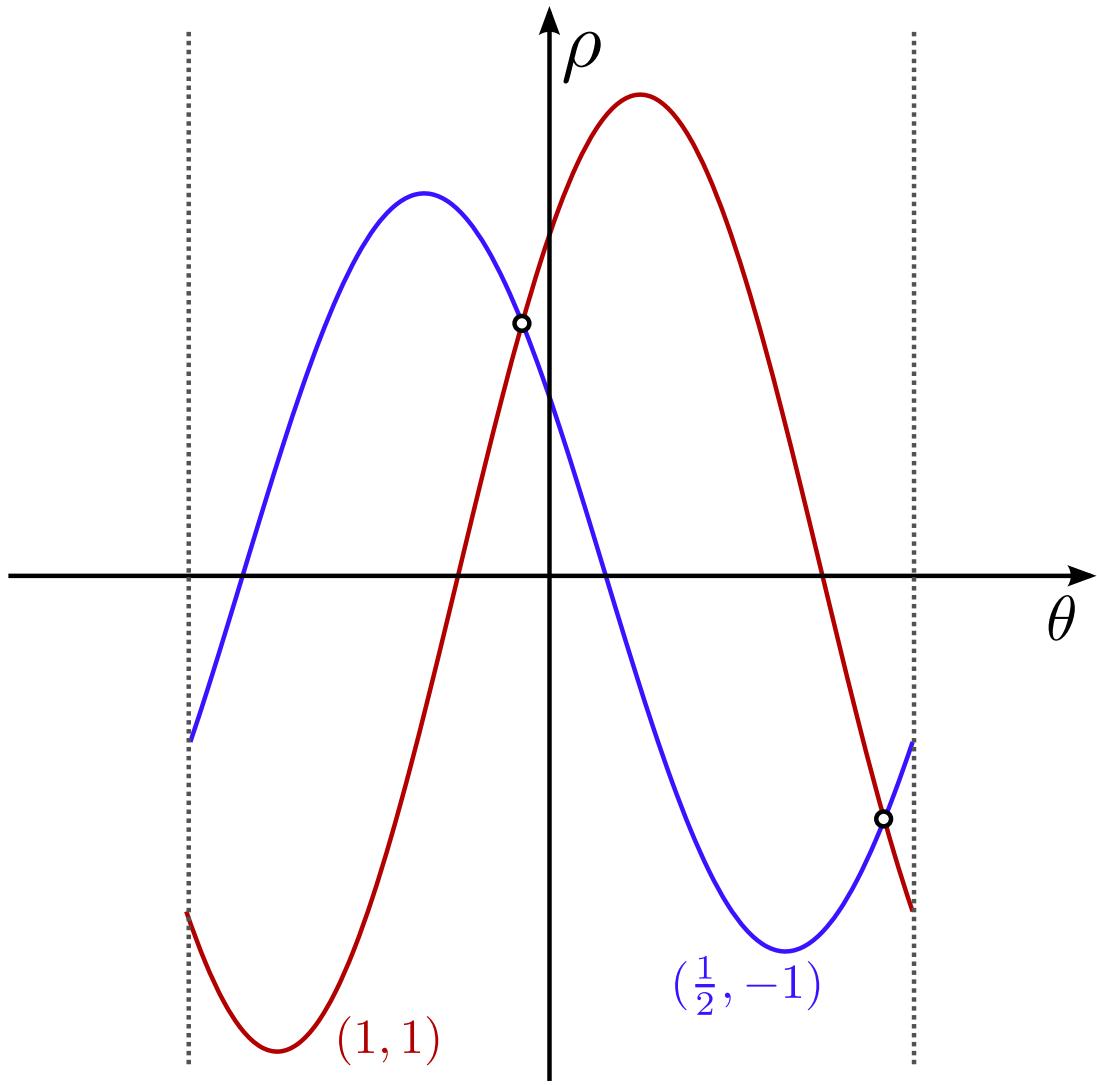


Figure 4: Sinusoids corresponding to two 2d points in the Hough parameter space. The θ axis ranges from $-\pi$ to $-\pi$.

To implement the algorithm in code, we must first define the size of the accumulator array and the interval for the parameter values. Normally, we define ϑ on the interval from $-\pi/2$ to $\pi/2$, while ρ is defined on the interval from $-D$ to D , where D is the length of the

image diagonal. In the context of polar parametrization, ρ is technically a signed distance relative to the origin, thus the range should be extended to also cover negative values of D . The ϑ interval can also be reduced from 2π to π since the lines are undirected. The parameter ρ does not need to be larger than the diagonal of the input image, since lines further away from the origin would fall outside the image plane. Since the accumulator array must be discrete, the two parameters of the Hough transform are therefore the resolution of the accumulator matrix (let's call them `bins_rho` and `bins_theta`).

The algorithm for generating the Hough accumulator is as follows. First, define an empty accumulator arrays based on your chosen parameters. Then, iterate over edge pixels and for each value ϑ calculate the corresponding value for ρ . Since the line equation has 4 variables and you have 3 of them (ϑ , x , and y) this is a simple calculation. Then, increment the cells that correspond to different (ρ, ϑ) pairs by 1.

- (a) Define an empty accumulator array, then choose a nonzero point $p = (x, y)$ and calculate the sinusoid that represents all lines that pass through p . Increment the corresponding cells in the accumulator array. Experiment with different positions for p and different accumulator resolutions. To verify this first step, use the provided function `draw_line()` to draw all the lines. If done correctly, you should get something similar to Figure 5.

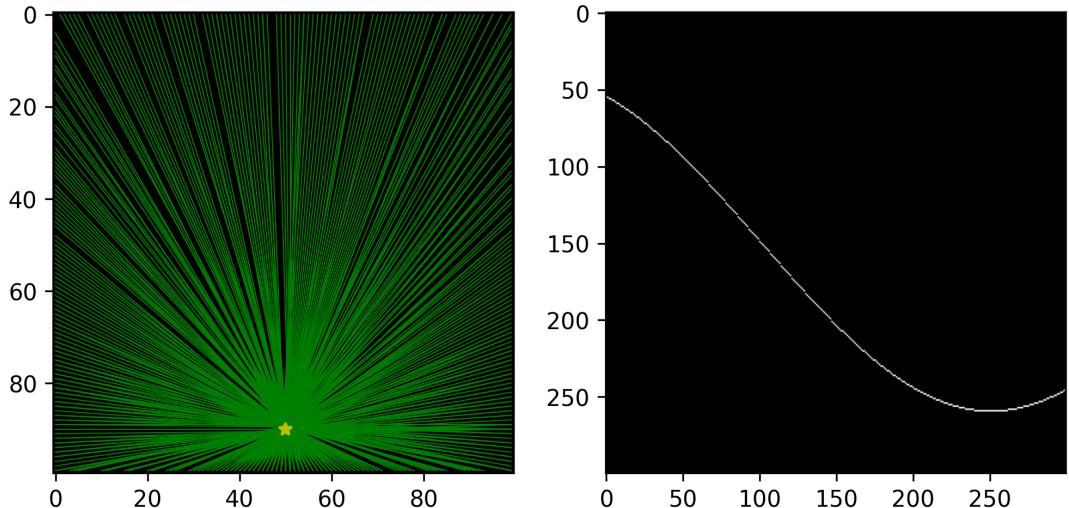


Figure 5: Hough accumulator array for a single point $p = (50, 90)$ and the corresponding lines in image space.

Note: If you defined both parameter intervals as a vector of values, the correct inputs for the function `draw_line()` should be the vectors indexed by the position of the accumulator cell, as well as the image height and width.

- (b) Implement the function `hough_find_lines()` that accepts a binary image, the number of bins for ϑ and ρ (allow the possibility of them being different) and a threshold. Create an accumulator matrix A for the parameter space (ρ, ϑ) . For each nonzero pixel in the image, generate a curve in the (ρ, ϑ) space by using the equation (7) for all possible values of ϑ and increase the corresponding cells in A . Display the accumulator matrix. Test the method on your own synthetic images (e.g. a 100×100 black image, with two white pixels at $(10, 10)$ and $(10, 20)$).

Finally, test your function on two synthetic images `oneline.png` and `rectangle.png`. First, you should obtain an edge map for each image using either your function `finedges()` or some inbuilt function. Run your implementation of the Hough algorithm on the resulting edge images.

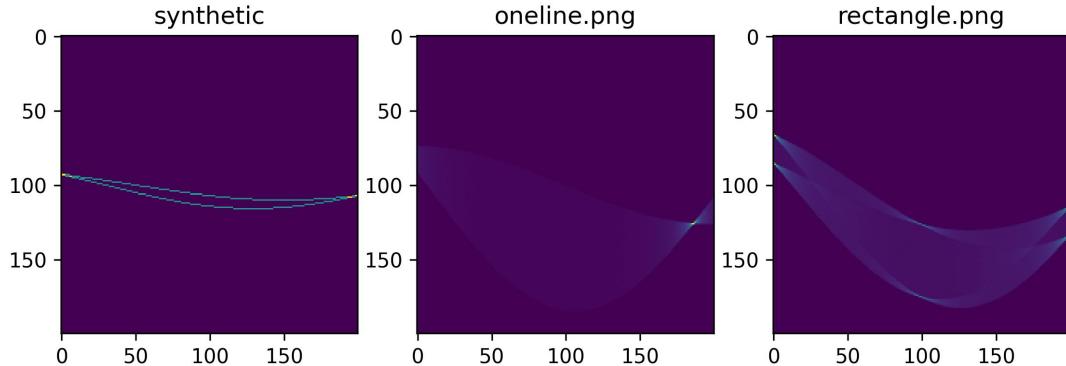


Figure 6: Accumulator values for various synthetic images.

- (c) The sinusoids don't usually intersect in only one point, resulting in more than one detected line. Implement a function named `nonmaxima_suppression_box()` that checks the neighborhood of each pixel and set it to 0 if it is not the maximum value in the neighborhood. Define the neighborhood as parameter k , determining the size of the square region that will be checked for each pixel. To speed up the process, use something like `np.nonzero()`. Also note that if more neighboring pixels have the maximum value, keep only one.
- (d) 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 `draw_line()` function that you can find in the supplementary material. Your results should look similar to Figure 7 (depending on the accumulator resolution and threshold).

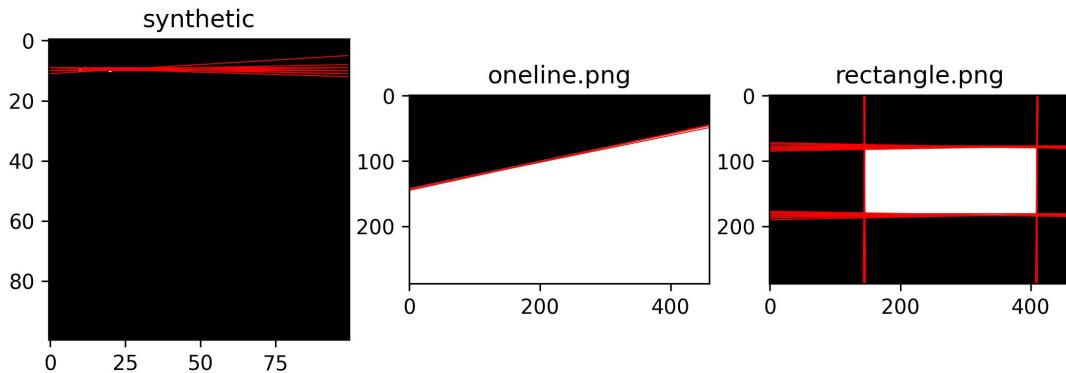


Figure 7: Lines resulting from Hough transform for various synthetic images.

- (e) Read the image from files `bricks.jpg`, `building.jpg`, and `pier.jpg`. Change the image to grayscale and detect edges. Then detect lines using your algorithm. As the results will likely depend on the number of pixels that vote for a 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. Display the results and experiment with parameters of the Hough algorithm as well as the edge detection algorithm, e.g. try changing the number of cells in the accumulator or parameter σ in edge detection to obtain results that are similar to the ones shown in Figure 8.

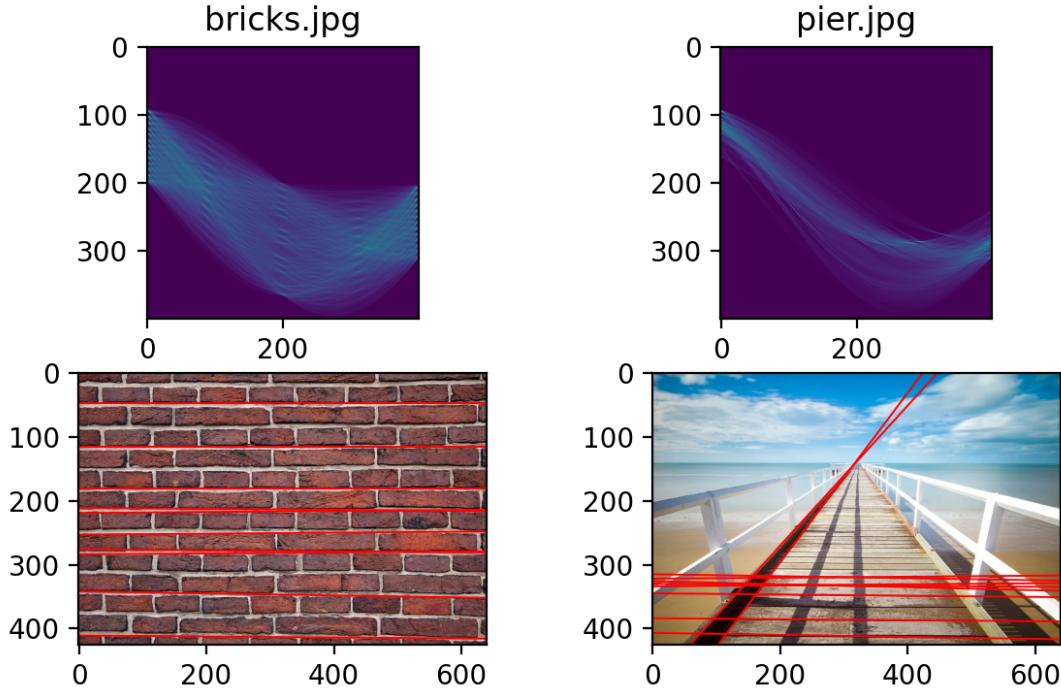


Figure 8: Accumulators and lines resulting from Hough transform of images `bricks.jpg` and `pier.jpg`.

- (f) ★ (10 points) In order to reduce the amount of votes and include additional domain knowledge, we can modify the Hough voting scheme. When we calculate the edges of the image, we can also obtain the magnitude and angle of the gradient of each pixel. The direction of the gradient of an edge pixel is perpendicular to the edge and can be used to limit the scope of the parameter ϑ . Not all cells on the ϑ interval need to be incremented, i.e. lines in the direction of the gradient are unlikely to be good candidates, while those perpendicular to the gradient are much more likely to have a higher support.

Modify your line detection algorithm so that it also accepts the matrix of edge angles. Note that the angle values were probably calculated using the `np.arctan2(dy, dx)` function that returns the values between wider range. 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. Note that you should define a range of angles around the chosen direction for which each pixel votes (e.g., a neighborhood of $\pm 5^\circ$ around the gradient-based direction). The approximate change in accumulator appearance is shown in Figure 9.

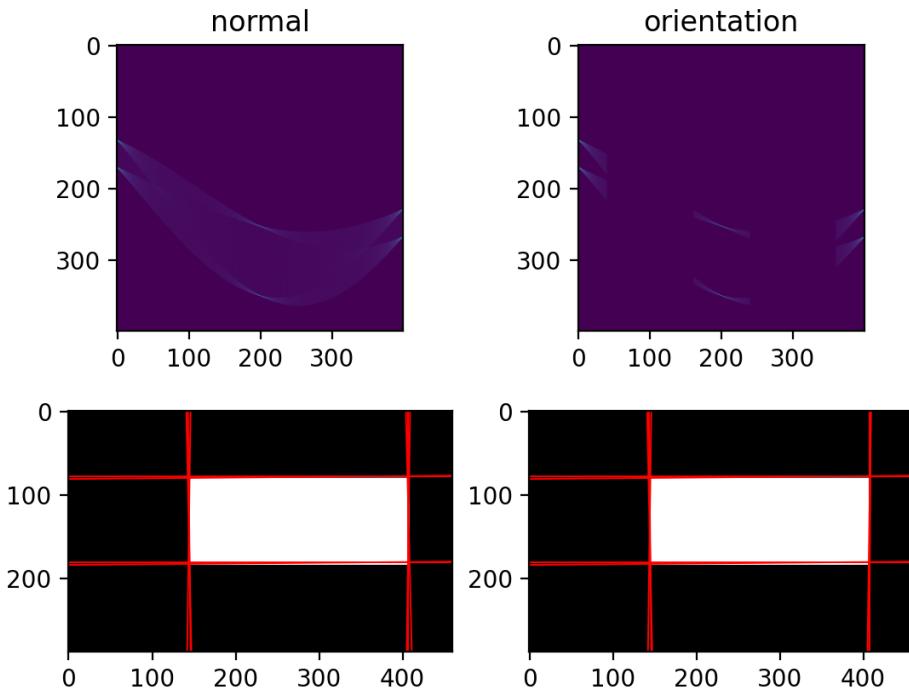


Figure 9: Resulting accumulators and lines for the baseline and gradient-filtered versions of the Hough transform.

- (g) ★ (5 points) Implement a Hough transform that detects circles of a fixed radius. You can test the algorithm on images `eclipse.jpg` and `coins.jpg`. Try using a radius somewhere between 45 and 50 pixels. You do not need to detect every circle, just show that you can reliably detect most. Report how the formulation of the Hough voting scheme changes for detecting circles. The approximate accumulator matrix and detected circles are shown in Figure 10.

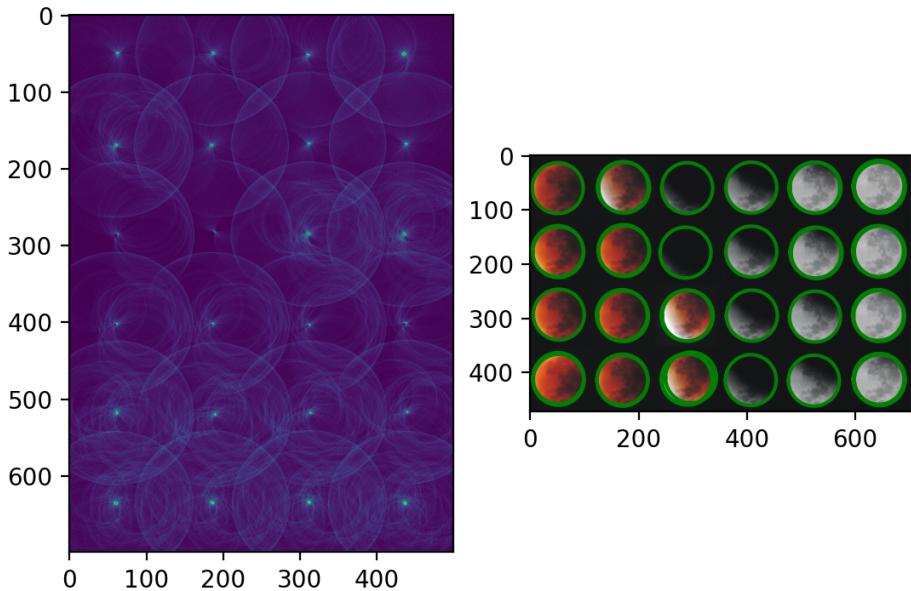


Figure 10: Accumulator and resulting circles for image `eclipse.jpg`.

- (h) ★ (15 points) Not all lines can accumulate the same number of votes, e.g. if the image is not square, candidates along the longer dimension are in better position since there are more pixels that can vote for them. 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 images containing objects with high aspect ratios (e.g. `rectangle.png`), where the difference can be shown clearly.

References

- [1] D. A. Forsyth and J. Ponce. *Computer Vision: A Modern Approach*. Prentice Hall, 2002.
- [2] D. A. Forsyth and J. Ponce. Computer vision: A modern approach (online version). <https://eclasse.hmu.gr/modules/document/file.php/TM152/Books/Computer%20Vision%20-%20A%20Modern%20Approach%20-%20D.%20Forsyth%2C%20J.%20Ponce.pdf>, 2003.
- [3] R. E. Woods, R. C. Gonzalez, and P. A. Wintz. *Digital Image Processing, 3 ed.* Pearson Education, 2010.