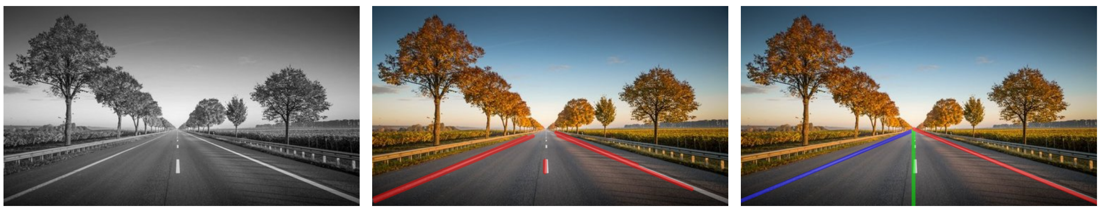


(a) Input image. (b) Hough Transform. (c) Detected road markings.

Figure 1: The lane detection algorithm applied to an input image. On the left the original image, in the middle detected lines using the Hough Transform and on the right the detected road markings.

CV/task1 — Lane Detection

1 Overview

This task serves as an introduction to working with the popular OpenCV library. The goal is to implement a pipeline for automatic lane detection in road images which localizes the road markings in a given image. Fig. 1 shows a stripped down example of the pipeline in Fig. 2.

In order to find the lanes, we first extract the edges of the input image using the Canny edge detection algorithm [1]. This method returns an image where for each pixel, it is indicated whether it lies on an edge or not. Then we apply the Hough Transform [2] to extract lines. This algorithm uses a representation of the edge image in a parameter space where high density regions indicate the presence of a line. The found lines are then grouped into three classes, based on their slope, and averaged to get an estimate for the left, middle and right road marking.

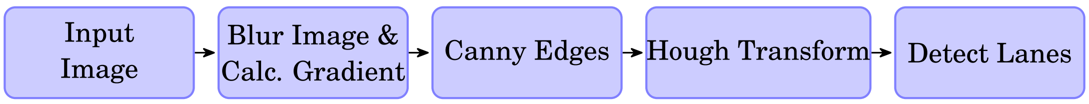


Figure 2: Overview of the individual steps of the algorithm.

2 Tasks

The provided framework contains a `main.cpp` (which must not be edited as it is used for automatic execution) and an `algorithms.cpp` where the tasks should be implemented in the respective functions (`compute_grayscale`, `gaussian_blur`, `compute_gradient`, `non_maxima_suppression`, `hysteresis`, `canny_edges`, `apply_convex_mask`, `bin_lines`, `average_lines` , `compute_intersection` and for the bonus task `hough_fill_accumulator`, `hough_get_local_maxima` and `hough_convert_to_pts`).

The required parameters for each test case (e.g. `road`) are stored in separate JSON files (e.g. `road.json`) located in the folder `tests` in your repository. The file `road.json` has to be passed to the application to execute the test case `road`. Then the input data is fetched automatically from the JSON files and should be used directly for the calculations in the respective functions. Do not modify these parameters. They are also passed to the respective subroutines for you. The content of this task is limited solely to the functions in `algorithms.cpp`. You do not have to modify any other provided file. The framework is built such that only the previously mentioned functions have to be implemented to achieve the desired output.

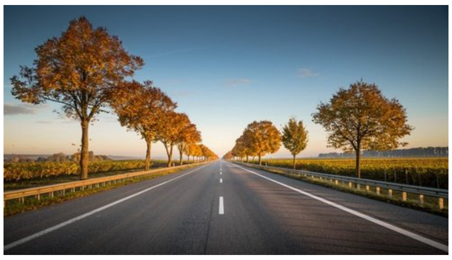
The example is divided into several sub-tasks. The pipelines consists of the following steps:

- Task 1
  - Compute Grayscale [1pt]
  - Gaussian Blur [1pt]
  - Compute Gradient [1pt]
  - Canny Edge Detector [5pts]
    - Non Maxima Suppression [2pt]
    - Hysteresis [2pt]
    - Canny Edges [1pt]
  - Apply Convex Mask [1pt]
  - Bin Lines [3pts]
  - Average Lines [1pt]
  - Compute Intersection [3pts]
  - Bonus - Hough Transform [3pts]
    - Fill Accumulator [1pt]
    - Get Local Maxima [1pt]
    - Convert to Cartesian Coordinates [1pt]

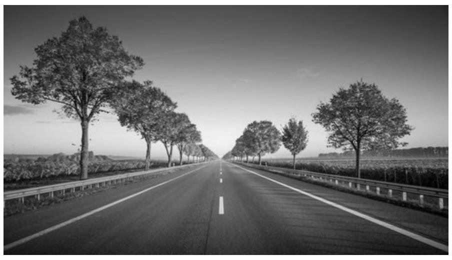
This task has to be implemented using OpenCV[1] 4.5.4. Use the functions provided by OpenCV and pay attention to the different parameters and image types. OpenCV uses the BGR color format instead of the RGB color format for historical reasons, i.e., the reversed order of the channels. This has to be considered in the implementation.

Notation We denote images as upper-case letter (e.g. G for gradients, A for the hough accumulator matrix) and image locations as lower-case letters in bold typeface (e.g. $\mathbf{p}$). For pixel locations, we denote the x, y components as p_x, p_y. We denote index i as $\mathbf{p}^{(i)}$ for sets of points. We denotes lines as ℓ. We denote rounding to the nearest integer value as $\lfloor\cdot\rceil$.

[1] http://opencv.org/



(a) RGB input image.



(b) Grayscale output image.

Figure 3: Expected output for `compute_grayscale(...)`.

2.1 Compute Grayscale (1 Point)

In the function `compute_grayscale(...)`, you have to generate a grayscale image from a 3-channel RGB input image. For this purpose, the information of the three color channels must be extracted for each pixel of the input image. We recommend the access to the pixel at location `(row, col)` via `input_image.at<cv::Vec3b>(row, col)`. This command returns a vector of 3 byte values containing the channel intensities.

There are two things to keep in mind when fetching color information from `cv::Mat` objects in OpenCV:

- OpenCV uses a row-major order, which means that the first index always refers to the row that should be accessed and the second index to the column of interest, respectively.
- OpenCV uses the BGR channel order. As a consequence, the first element of an extracted `cv::Vec3b` object is related to blue, the second to green, and the third one to red.

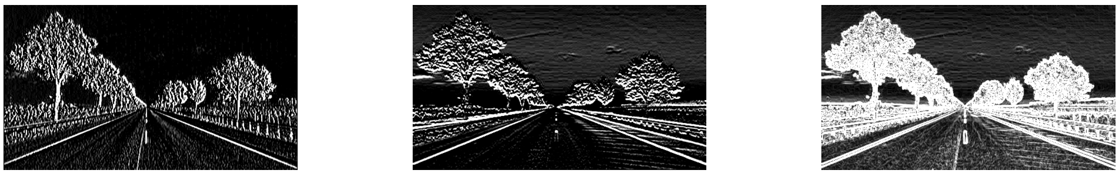
After the color information is obtained, it has to be combined according to

$$I = 0.2989 \cdot R + 0.5870 \cdot G + 0.1140 \cdot B \tag{1}$$

to retrieve the intensity value for the corresponding pixel in the output image. This formula considers that the human eye is more sensitive to green components in the spectrum of light. That is the reason for giving the green color channel the highest weight. After the calculation, the result can directly be written to the passed single-channel `cv::Mat` reference `grayscale_image` after the conversion. Fig. 3 shows an example for the expected output.

Forbidden Functions:

- `cv::cvtColor(...)`



(a) G_x. (b) G_y. (c) G.

Figure 4: Expected output of function `compute_gradient(...)`.

2.2 Gaussian Blur (1 Point)

In the function `gaussian_blur(...)`, you have to blur the image according to the parameters σ, k_s. This is done by convolving the image with a gaussian kernel $K \in \mathbb{R}^{k_s \times k_s}$, where k_s denotes the kernel size. Get a 1D gaussian kernel using `cv::GetGaussianKernel(...)`. This returns a filter $\mathbf{k} \in \mathbb{R}^{k_s \times 1}$. Next, compute K using an outer product:

$$K = \mathbf{k}\mathbf{k}^T. \tag{2}$$

Then, call `cv::filter2D(...)` using K and the provided σ as parameters. An example can be seen in Fig. 5a.

Useful Functions:

- `cv::GetGaussianKernel(...)`
- `cv::filter2D(...)`
- `cv::mulTransposed(...)`

Forbidden Functions:

- `cv::GaussianBlur(...)`

2.3 Compute Gradient (1 Point)

This task is to be solved in the function `compute_gradient(...)`. The gradient calculation is done by applying the Scharr operator. This should be done by utilizing the function `cv::Scharr`, which detects directed changes in the input image with the help of derivatives. A large gradient value in certain image regions indicates strongly pronounced intensity changes caused by edges, color transitions, exposure differences, and so forth. For the derivative in x-direction G_x, the image I is convolved with the kernel S_x. Analogously, for the derivative in y-direction G_y, the image I is convolved with the kernel S_y. This can be formally written as

$$G_x = S_x * I = \begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix} * I \tag{3}$$

and

$$G_y = S_y * I = \begin{bmatrix} -3 & -10 & -3 \\ 0 & 0 & 0 \\ 3 & 10 & 3 \end{bmatrix} * I. \tag{4}$$

The convolution operation is denoted with $*$.

First, you have to determine the respective first-order derivative in the directions x and y. When calling the function `cv::Scharr` set the parameter describing the bit depth of the output image to the type `CV_32F`, which corresponds to a 32-bit floating point value. After that, combine the two gradient images according to

$$G = \sqrt{G_x^2 + G_y^2}, \tag{5}$$

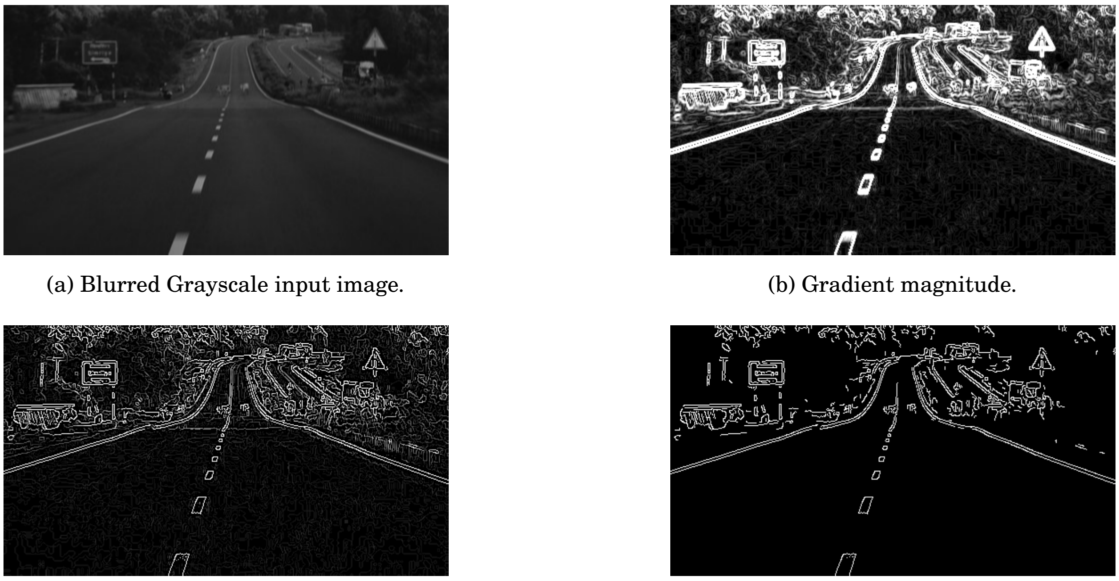
where the resulting image G stores the per-pixel ℓ_2-norm of the gradient. The three generated images (x-derivative, y-derivative, and gradient norm) are supposed to be returned via the provided `cv::Mat` references. In Fig. 4 you can see an example of the expected output.

Useful Functions:

- `cv::Scharr(...)`
- `cv::pow(...)`
- `cv::add(...)`
- `cv::sqrt(...)`

2.4 Canny Edge Detector (5 Points)

In this task, you will implement the famous Canny edge detection algorithm [1]. The method is an essential part of many computer vision algorithms. An example of this pipeline can be seen in Fig. 5.



(a) Blurred Grayscale input image.

(b) Gradient magnitude.

(c) Non-maxima suppression.

(d) Detected Edges.

Figure 5: Pipeline of the Canny Edge Detector

2.4.1 Non-Maxima Suppression (2 of 5 Points)

This part of the task has to be implemented in the function `non_maxima_suppression(...)`. Here, local maxima in the gradient should be found. This leads to the fact that in the resulting image, only the detected edges remain. For each pixel **p**, the already calculated gradient from Sec. 2.3 is used to calculate the angle

$$\alpha = \angle G = \arctan \frac{G_y}{G_x}. \tag{6}$$

Afterwards, transform all angles smaller than 0° to the interval $[0, 180]$:

$$\beta = \begin{cases} (\alpha + 360) \mod 180 & \text{if } \alpha < 0, \\ \alpha & \text{otherwise.} \end{cases} \tag{7}$$

The calculated angle β is now assigned to one of the four classes (1: horizontal, 2: diagonal-1, 3: vertical, 4: diagonal-2) according to Table 1. Depending on the class of the angle β, the focus lies on neighboring pixels in different directions. If either the value of pixel **q** or **r** is larger than the value of pixel **p**, $\hat{G}(\mathbf{p}) = 0$ because it is not the largest value in the gradient direction:

$$\hat{G}(\mathbf{p}) = \begin{cases} 0 & \text{if } G(\mathbf{p}) < G(\mathbf{q}) \text{ or } G(\mathbf{p}) < G(\mathbf{r}), \\ G(\mathbf{p}) & \text{otherwise.} \end{cases} \tag{8}$$

The non-maxima suppression result $\hat{G}(\mathbf{p})$ should be written to `non_max_sup`.

Table 1: Classification table of the angles and relationship to the neighboring pixels **q** and **r**.

Class	Angle	Neighbor Relation $(\mathbf{p} = (p_x, p_y))$
1	$\beta \leq 22.5$ or $\beta > 157.5$	$\mathbf{q} = (p_x - 1, p_y)$, $\mathbf{r} = (p_x + 1, p_y)$
2	$22.5 < \beta \leq 67.5$	$\mathbf{q} = (p_x - 1, p_y - 1)$, $\mathbf{r} = (p_x + 1, p_y + 1)$
3	$67.5 < \beta \leq 112.5$	$\mathbf{q} = (p_x, p_y - 1)$, $\mathbf{r} = (p_x, p_y + 1)$
4	$112.5 < \beta \leq 157.5$	$\mathbf{q} = (p_x - 1, p_y + 1)$, $\mathbf{r} = (p_x + 1, p_y - 1)$

Useful Functions:

- `cv::copyTo(...)`
- `std::atan2(...)`

2.4.2 Hysteresis Thresholding (2 of 5 Points)

This part of the task must be implemented in the function `hysteresis(...)`. The already found and refined edges should be compared with the thresholds τ_{min} and τ_{max}. Each pixel **p** is classified, whether it belongs to a weak, a strong, or no edge. If the value of pixel **p** is greater than the upper threshold τ_{max} then it is part of a strong edge. Strong edges are certainly part of the final result. If the value is smaller than the lower threshold τ_{min} then this pixel is certainly not part of an edge and therefore not part of the final result. If the value of pixel **p** is between the two thresholds, it is part of a weak edge. The value assignment in the hysteresis thresholded image H is as follows:

$$H(\mathbf{p}) = \begin{cases} 255 \text{ (strong edge)} & \text{if } \hat{G}(\mathbf{p}) \geq \tau_{\text{max}}, \\ \text{weak edge} & \text{if } \tau_{\text{min}} \leq \hat{G}(\mathbf{p}) < \tau_{\text{max}}, \\ 0 \text{ (no edge)} & \text{otherwise.} \end{cases} \tag{9}$$

After all non-edges and strong edges in the image have been found and marked, weak edges are classified iteratively or recursively depending on their neighborhood. If in the 8-neighborhood of a pixel **p**, which belongs to a weak edge, at least one pixel is part of a strong edge, then the pixel **p** also becomes a strong edge (=255). If this is the case, the weak edge becomes a strong edge. It is recommended to save all pixels that have already been classified as strong edges and then examine their 8-neighborhood to search for a weak edge in this area. Furthermore, search in the 8-neighborhood of the freshly found strong edge as well. Using a recursive approach makes sure to not miss any weak edges. At the end of this function, $H(\mathbf{p}) \in \{0, 255\}\ \forall\ \mathbf{p}$.

2.4.3 Canny Edges (1 of 5 Points)

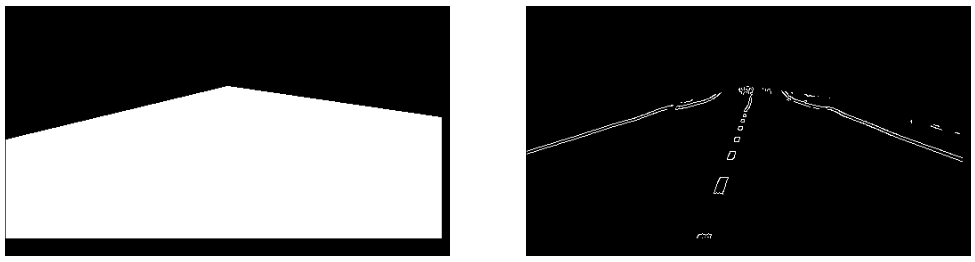
In the `canny_edges(...)` function, all previously implemented parts of the algorithm should now be applied to create an edge image. The function gets a blurred gray-scale image and the two thresholds τ_{min} and τ_{max} as input. Use the previously implemented function `compute_gradients(...)` to obtain G_x,G_y,G. Afterwards, call the functions `non_maxima_suppression(...)` and `hysteresis(...)` with according parameters. The result is the finished edge image which should be returned in `output_image`.

Useful Functions:

- `cv::Mat().copyTo(...)`

Forbidden Functions:

- `cv::Canny(...)`



(a) Convex bit-mask obtained from the sets of points.

(b) Applied mask on the Canny edges image.

Figure 6: Applying a mask to the Canny edge image to specify the region in which to search for lines. This is a form of outlier filtering.

2.5 Apply Convex Mask (1 Point)

To ease the task of lane detection, we provide the vertices $\{\mathbf{v}_i\}_{i=1}^{N}$ of a convex polygon, which specifies the region in which to search for lines. This removes the need for more sophisticated outlier filtering later on. In the function `apply_convex_mask(...)`, use the vertices provided in `std::vector<cv::Point2i> pts` to create a mask with `cv::fillConvexPoly(...)`. Make sure to set the parameter `cv::Scalar& color` to a value greater than 0. Next, apply the mask to the given `input_image` to obtain the edges in the region of interest only. We visualize both steps in Fig. 6.

Useful Functions:

- `cv::fillConvexPoly(...)`
- `cv::bitwise_and(...)`

Forbidden Functions:

- `cv::fillPoly(...)`

2.6 Bin Lines (3 Point)

After the Hough Transform is applied, we obtain a set of lines L, which we now aim to bin according to the slope in `bin_lines(...)`. Each line in `input_lines` is a `cv::Vec4i` with parameters $\begin{pmatrix} a & b & c & d \end{pmatrix}$. We obtain two points on the line $\mathbf{p}^{(0)} = \begin{pmatrix} a & b \end{pmatrix}^T$, $\mathbf{p}^{(1)} = \begin{pmatrix} c & d \end{pmatrix}^T$.

We need to be careful, as the pixel $\begin{pmatrix} 0 & 0 \end{pmatrix}^T$ is located in the upper left corner of the image. We transform this to a more intuitive coordinate system $\left(\text{with the bottom left corner at } \begin{pmatrix} 0 & 0 \end{pmatrix}^T\right)$ with

$$p_y^{(i)} = h - p_y^{(i)} \qquad \forall\ i, \tag{10}$$

where h denotes the height of the image.

We now transform this representation to a linear equation

$$p_y = w \cdot p_x + b. \tag{11}$$

To do this, we estimate the slope w and the intercept/bias b. In our case, b is the y-coordinate of the intersection of the line with the leftmost column of the image.

We can estimate the slope of the line with the two points using

$$w = \frac{p_y^{(0)} - p_y^{(1)}}{p_x^{(0)} - p_y^{(1)} + \epsilon}, \tag{12}$$

where ϵ is a small constant to avoid division by 0. To estimate the intercept, we use

$$b = \left\lfloor p_y^{(0)} - w \cdot p_x^{(0)} \right\rceil. \tag{13}$$

We define a new line with these points $\ell = \begin{pmatrix} w & b \end{pmatrix}^T$.

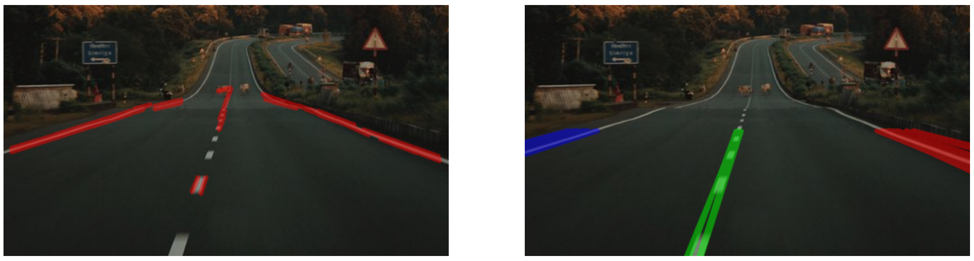
Table 2: Classification table of the lines according to the estimated slope w.

Class	Condition
center	$\|w\| > \tau_u$
right	$w < \tau_l$
left	otherwise

After we have computed the parameters w, b, we bin the lines according to w into left, center, and right. We use an upper threshold τ_u and a lower threshold τ_l. Bin the lines according to the relations given in Table 2. In our framework, the parameters `slope_upper_threshold`, `slope_lower_threshold` are given for this purpose. We show the outputs obtained from this in Fig. 7.

Useful Functions:

- `Line {slope, intercept}`
- `round(...)`



(a) Obtained Hough Lines. (b) Binned lines according to w.

Figure 7: Clustering the lines according to slope w to obtain three sets of lines for averaging later.

2.7 Average Lines (1 Point)

This task should be implemented in the function `average_lines(...)`. As we can see in Fig. 8, we get three sets of lines. We now average the parameters for each set of lines to obtain three lines, which we will process further. For each set of lines L_i, $i \in \{0,1,2\}$, accumulate the intercept and the slope and divide by the number of lines in the set $|L_i|$.

$$\bar{w}_i = \frac{1}{|L_i|} \sum_{j=1}^{|L_i|} w_j, \tag{14}$$

$$\bar{b}_i = \frac{1}{|L_i|} \sum_{j=1}^{|L_i|} b_j. \tag{15}$$

For the intercept b, perform the accumulation in `int`, the division by $|L_i|$ in `float`, then round to the nearest `int`. For the slope w, perform all computations in `float`.

Useful Functions:

- `round(...)`
- `std::accumulate(...)`

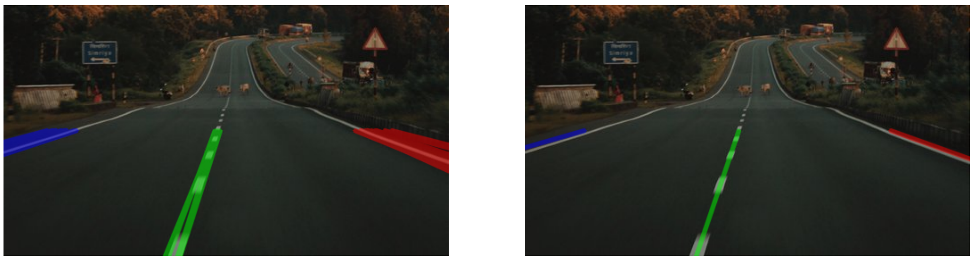

(a) Clustered lines from the previous step. (b) Averaging of the three sets of lines.

Figure 8: For each set of lines L_i, we average all parameters w, b to obtain a single line per set.

2.8 Compute Intersection (3 Point)

We now have two lines, specified by $\ell^{(1)}, \ell^{(2)}$. For an intersection point, we need two sets of two points each, $\{\mathbf{p}^{(1)}, \mathbf{p}^{(2)}\}, \{\mathbf{p}^{(3)}, \mathbf{p}^{(4)}\}$. We can obtain points from a line equation by defining p_y and computing p_x using

$$p_x = \frac{p_y - b}{w}. \tag{16}$$

Compute points for $p_y \in \{0, \mathtt{y}\}$, where `y` is of type `int`, given in the framework (to avoid numerical issues). $\{\mathbf{p}^{(1)}, \mathbf{p}^{(2)}\}$ denote the points for $\ell^{(1)}$, $\{\mathbf{p}^{(3)}, \mathbf{p}^{(4)}\}$ denote the points for $\ell^{(2)}$.

We now try to find an intersection point of these two lines. To do this, we transform the points, which are $\in \mathbb{R}^2$ to projective space $\mathbb{P}^2$. Here, each point is a ray in 3d-space and parallel lines intersect at a point at infinity $\left(\text{at } \begin{pmatrix} x & y & 0 \end{pmatrix}^T\right)$. Transforming points to projective space works by appending a 1, i.e.

$$\begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \ \mathbb{R}^2 \to \mathbb{P}^2. \tag{17}$$

In $\mathbb{P}^2$, lines are also defined as 3-vectors. We can compute a line by two points with

$$\mathbf{l}^{(1)} = \mathbf{p}^{(1)} \times \mathbf{p}^{(2)}, \tag{18}$$

$$\mathbf{l}^{(2)} = \mathbf{p}^{(3)} \times \mathbf{p}^{(4)}, \tag{19}$$

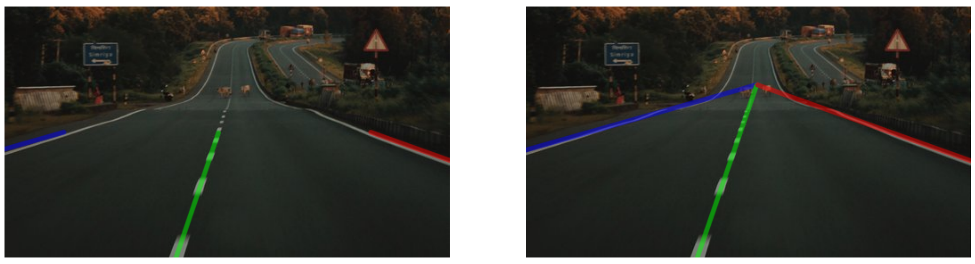
where $\mathbf{l}^{(i)} \in \mathbb{P}^2$ denotes a line in projective space. In $\mathbb{P}^2$, lines and points are dual. This means that we can get a new equation by substituting points for lines in the original equation. In our case, the cross product of two points computes a line, and the cross product of two lines computes a point. This point $\mathbf{p}^{\text{int}}$ is the intersection of the two lines in $\mathbb{P}^2$:

$$\mathbf{p}^{\text{int}} = \mathbf{l}^{(1)} \times \mathbf{l}^{(2)}. \tag{20}$$

We now need to transform $\mathbf{p}^{\text{int}}$ back to $\mathbb{R}^2$. We do this with perspective division

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \mapsto \begin{pmatrix} \frac{x}{z} \\ \frac{y}{z} \end{pmatrix}, \ \mathbb{P}^2 \to \mathbb{R}^2. \tag{21}$$

Finally, return $\begin{pmatrix} \lfloor x \rceil & \lfloor y \rceil \end{pmatrix}^T$ (after perspective division). This way, we can visualize the lines until they intersect. The output image for this algorithm is shown in Fig. 9.

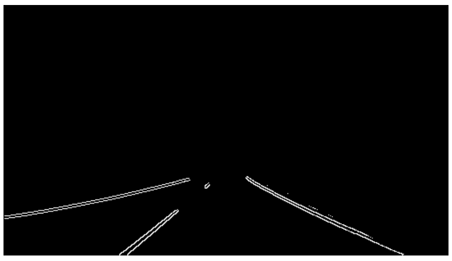


(a) 3 lines visualized. (b) Lines end at computed intersections.

Figure 9: Calculation of the line intersections for visualization. Due to perspective distortion, the intersection is in the image.

Useful Functions:

- `cv::Point3f::cross(...)`


(a) An edge image.

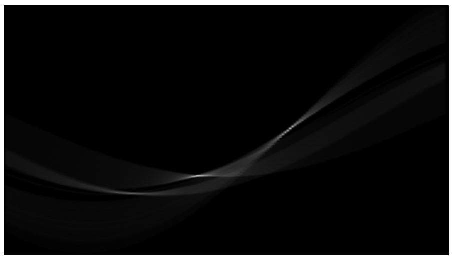

(b) The Hough transform of the image.

Figure 10: The three prominent intersections in Hough space correspond to the three lines in the edge image.

2.9 Bonus - Hough Transform (3 Points)

So far we used the probabilistic Hough Transform which was first introduced in [3], in the bonus task you will implement the traditional Hough Transform. The Hough Transform is a classical feature extraction algorithm that can be used to detect certain classes of shapes within an image, given that some (simple) parametric equation describing them is known (e.g. lines, circles). Using the polar representation we can define a line as a set of points

$$L = \{(x, y) : x \cdot \cos\theta + y \cdot \sin\theta = r\} \tag{22}$$

with some parameters r and θ. This means that a line can be seen as a point in a 2-dimensional parameter space. In this space, a single pixel **p** represents the set of solutions to Eqn. (22) where $x = p_x$ and $y = p_y$, which corresponds to a sinusoidal curve. An example is illustrated in Fig. 10.

The Hough Transform proceeds by mapping every pixel in the input image into the chosen parameter space, resulting in an image like the one in Fig. 11a. Then, high-density regions where many intersections are located are returned as candidate lines. These two parts are to be implemented in Sec. 2.9.1 and Sec. 2.9.2 respectively.

For a more detailed explanation of the algorithm please refer to lecture 5.

2.9.1 Fill Accumulator (1 Point)

At first, we calculate the number of angles n_θ and the number of radii n_ρ, this is already done for you. We calculate them as follows:

$$n_\theta = \left\lfloor \frac{\pi}{\theta} \right\rceil \tag{23}$$

$$n_\rho = \left\lfloor \frac{(w + h) \cdot 2 + 1}{\rho} \right\rceil, \tag{24}$$

where w, h denotes the dimension of the input image. This part of the solution has already been done for you, use num_angle, num_rho.

In order to effectively iterate through all theta values for all the edge pixels, we pre-compute all corresponding $cos(\theta)$ and $sin(\theta)$ values. They are then multiplied with $\frac{1}{\rho}$ and stored in the vectors $\mathbf{\Theta}_{\text{cos}} \in \mathbb{R}^{n_\theta}, \mathbf{\Theta}_{\text{sin}} \in \mathbb{R}^{n_\theta}$. This will later on save lots of redundant calculations. Formally, the entries look like the following:

$$\mathbf{\Theta}_{\text{sin}} = \left\{ \sin(\theta \cdot i) \cdot \frac{1}{\rho} \right\}_{i=0}^{n_\theta}, \tag{25}$$

$$\mathbf{\Theta}_{\text{cos}} = \left\{ \cos(\theta \cdot i) \cdot \frac{1}{\rho} \right\}_{i=0}^{n_\theta}, \tag{26}$$

where θ is given as the parameter theta.

In the next step, we iterate over each pixel $\mathbf{p}$ of the edge image. If $I(\mathbf{p}) \neq 0$, the pixel $\mathbf{p}$ is considered a candidate for a line, $\mathbf{p} \in K$. Therefore, we test $\forall\ \mathbf{p} \in K$ all angles θ, to get all possible lines through the point (with the chosen parameters). We then increase the corresponding entry in the accumulator matrix A (also known as the voting matrix) by one. Assume, that we want to calculate the accumulator entries for the i-th angle. The corresponding index r for the accumulator matrix is calculated as

$$r = \left\lfloor \left(p_x \cdot \Theta_{\text{cos}}^{(i)} + p_y \cdot \Theta_{\text{sin}}^{(i)} \right) + d \right\rceil, \tag{27}$$

where p_x and p_y represent the coordinates of the pixel $\mathbf{p}$ in the image. The diagonal of the image is represented by d, it is used to shift the ρ-values to the positive area, hence allowing for more easy access. The diagonal d is defined as

$$d = \frac{(n_\rho - 1)}{2}. \tag{28}$$

The result of a filled accumulator matrix is shown in Fig. 11a.

Useful Functions:

- round(...)

2.9.2 Get Local Maxima (1 Point)

In this step, we will iterate over the entries $\mathbf{a}$ of the accumulator matrix $A \in \mathbb{R}^{n_\rho \times n_\theta}$. A local maximum is defined by the brightest pixel in a local neighborhood of A. Therefore, we slide a Cross-Window (see Fig. 12) over the accumulator A and look for local maxima. For each location $\mathbf{a} = \begin{pmatrix} a_x & a_y \end{pmatrix}^T$, we define the neighborhood of $\mathbf{a}$ as

$$N(\mathbf{a}) = \left\{ \begin{pmatrix} a_x - 1 & a_y \end{pmatrix}^T, \begin{pmatrix} a_x & a_y - 1 \end{pmatrix}^T, \begin{pmatrix} a_x + 1 & a_y \end{pmatrix}^T, \begin{pmatrix} a_x & a_y + 1 \end{pmatrix}^T \right\}. \tag{29}$$

(a) Hough Accumulator.

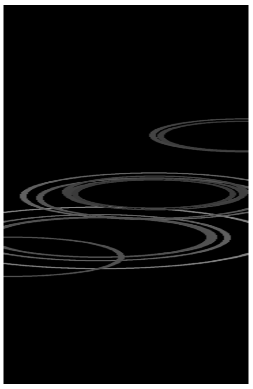

(b) Hough Local Maxima.

Figure 11: Visualisation of the filled 2D-accumulator matrix and the local maxima.

If $A(\mathbf{a})$ is strictly larger than all its neighbors $\mathrm{N}(\mathbf{a})$, **a** fulfills the condition to be a local maximum. Be aware to only iterate over pixels **a**, where each pixel of the neighborhood N is defined.

Additionally, a value is considered a local maximum only if it exceeds a certain threshold τ_{max}:

$$A(\mathbf{a}) > \tau_{\text{max}}. \tag{30}$$

We store the index **a** and the value $A(\mathbf{a})$ for all relevant local maxima (above τ_{max}) in a `std::vector<LocalMaximum>`. Each local maxima represents a detected line.

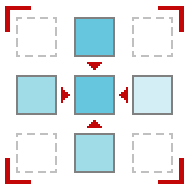


Figure 12: Cross-Window approach for identification of local maxima.

Now we sort and store the maximally first `max_num_lines` sorted entries in `local_maxima`. The entries should be sorted in descending order according to their respective accumulator value. The local maxima are visualized in Fig. 11b. We visualize the locations as circles with a radius proportional to the accumulated value for easier debugging.

Useful given Struct:

- `LocalMaximum { theta_index, rho_index, accumulator_value}`

2.9.3 Convert to Cartesian Coordinates (1 Point)

Finally, we use the stored and sorted values in `local_maxima` to calculate the corresponding lines in the hessian normal form (see Eqn. (32)). We denote the entries of `local_maxima` as $\mathbf{m} = \begin{pmatrix} m_\theta & m_\rho & m_a \end{pmatrix}^T$:

$$\ell_\rho = \left(m_\rho - \frac{n_\rho - 1}{2}\right) \cdot \rho, \tag{31}$$

$$\ell_\theta = m_\theta \cdot \theta. \tag{32}$$

Now, we convert the polar representation of each line (ℓ_ρ, ℓ_θ) to a representation with two points ($\mathbf{p}^{(0)}$, $\mathbf{p}^{(1)}$). This is done by calculating the adjacent and opposite lines of the angle and then calculating the point coordinates.

$$u = \ell_\rho \cdot \cos(\ell_\theta) \tag{33}$$

$$v = \ell_\rho \cdot \sin(\ell_\theta) \tag{34}$$

The x, y-coordinates of the respective points are then given as:

$$p_x^{(0)} = \lfloor u + k \cdot \cos(\ell_\theta) \rceil, \tag{35}$$

$$p_y^{(0)} = \lfloor v + k \cdot (-\sin(\ell_\theta)) \rceil, \tag{36}$$

$$p_x^{(1)} = \lfloor u - k \cdot \cos(\ell_\theta) \rceil, \tag{37}$$

$$p_y^{(1)} = \lfloor v - k \cdot (-\sin(\ell_\theta)) \rceil, \tag{38}$$

where k is a positive integer constant. In our case, we provide k as the function argument `factor`. The point coordinates ($p_x^{(0)}, p_y^{(0)}, p_x^{(1)}, p_y^{(1)}$) should then be stored in the `std::vector<cv::Vec4i> line_pts` (in the order specified above).

3 Framework

The following functionality is already implemented in the program framework provided by the ICG:

- Processing of the passed JSON configuration file.
- Reading of the corresponding input image.
- Iterative execution of the functions in `algorithms.cpp`.
- Writing of the generated output images to the corresponding folder.

4 Submission

The tasks consist of several steps, which build on each other but are evaluated independently. On the one hand, this ensures objective assessment and, on the other, guarantees that points can be scored even if the tasks are not entirely solved.

We explicitly point out that the exercise tasks must be solved independently by each participant. If source code is made available to other participants (deliberately or by neglecting a certain minimum level of data security), the corresponding part of the assignment will be awarded 0 points for all participants, regardless of who originally created the code. Similarly, it is not permitted to use code from the web, books, or any other source. Both automatic and manual checks for plagiarism will be performed.

The submission of the exercise examples and the scheduling of the assignment interviews takes place via a web portal. The submission takes place exclusively via the submission system. Submission by other means (e.g. by email) will not be accepted. The exact submission process is described in the `Readme.md` in your repository.

The tests are executed automatically. Additionally, the test system has a timeout after seven minutes. If the program is not completed within this time, it will be aborted by the test system. Therefore, be sure to check the runtime of the program when submitting it.

Since the delivered programs are tested semi-automatically, the parameters must be passed using appropriate configuration files exactly as specified for the individual examples. In particular, interactive input (e.g. via keyboard) of parameters is not permitted. If the execution of the submitted files with the test data fails due to changes in the configuration, the example will be awarded 0 points.

The provided program framework is directly derived from our reference implementation, by removing only those parts that correspond to the content of the exercise. Please do not modify the provided framework and do not change the call signatures of the functions.

References

[1] John Canny. A Computational Approach to Edge Detection. IEEE Transactions on Pattern Analysis and Machine Intelligence, 8(6):679–698, 1986.

[2] Richard O Duda and Peter E Hart. Use of the hough transformation to detect lines and curves in pictures. Communications of the ACM, 15(1):11–15, 1972.

[3] Jiri Matas, C. Galambos, and J. Kittler. Robust detection of lines using the progressive probabilistic hough transform. Computer Vision and Image Understanding, 78:119–137, 04 2000.