TTK4255 Robotic Vision, Spring 2020

Exercise 5: Image Filtering and Line Detection

Authored by Simen Haugo (simen.haugo@ntnu.no)

Due: See Blackboard

This exercise is a mix of programming problems and theory questions. You are free to use any programming language, but solutions will be provided in Python and Matlab, and we may not be able to help you effectively if you use a different language. We therefore recommend that you use either Python or Matlab. You should also do exercise 0 to get familiar with language features and libraries that are useful for the exercises.

Instructions

This exercise is **approved** / **not approved**. To get your exercise approved, submit your answers to the questions as a **PDF** to Blackboard. You **don't** have to submit your code. Feel free to use LaTeX, Word, handwritten scans, or any other tool to write your answers.

Getting help

If you have questions about the exercise, you can:

- Post a question on Piazza (you can post anonymously if you want)
- Ask for help during the tutorial sessions (see Piazza for time and place)

Primary resources

- Lecture 6.
- *Computer Vision: Algorithms and Applications* by Richard Szeliski.
  - Ch. 3.2: Linear filtering
  - Ch. 4.2.1: Edge detection
  - Ch. 4.3.2: Hough transforms

Other relevant resources

- *Computer Vision: A Modern Approach* by Forsyth and Ponce.
  - Ch. 7: Linear Filters (up to section 7.3)
  - Ch. 8: Edge Detection
- *Robotics, Vision and Control 2nd Edition.*
  - Ch. 12.5: Spatial Operations
  - Ch. 13.2: Line Features

Overview

In this exercise you'll implement a line detection algorithm! It works by first filtering the image to extract pixels that lie on strong brightness discontinuities (edges). Then, using the edges in a Hough transform (pronounced "huff") to vote for the lines that may be present in the image. Fig. 1 shows a sample output, but your results may be different depending on your choice of thresholds and your implementation.

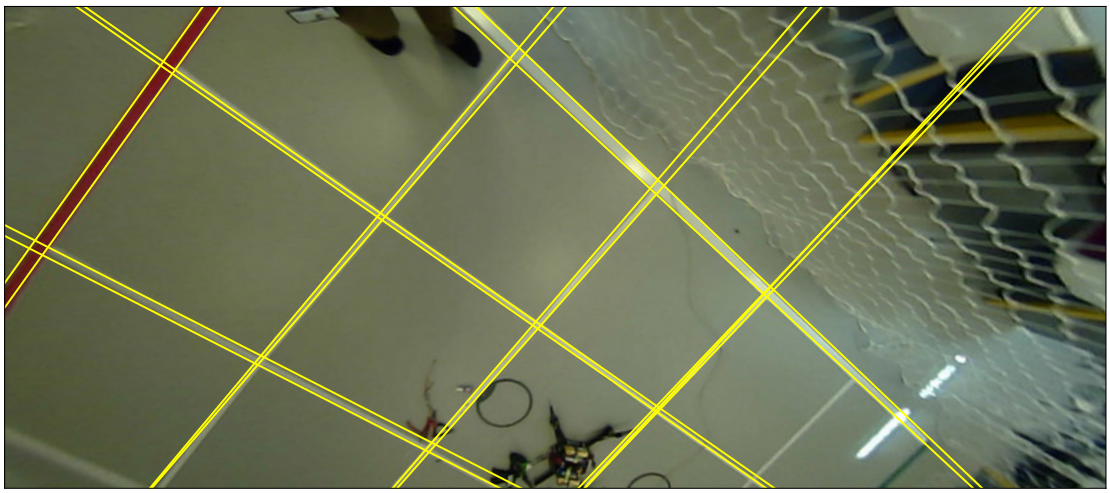

Figure 1: Example output using the steps described in this exercise.

The zip includes two scripts

- `task1.m/py`
- `task2.m/py`

that take care of reading in an image, making function calls to the various steps of task 1 and task 2 (the functions you will be implementing) and generates images showing the output and some of the intermediate steps.

The remaining files contain function declarations where you have to fill in the correct implementation. Once you have done this you should be able to run the above scripts to generate the output.

You are free to modify the scripts or write your own as long as your figures display what is asked for in task 1 and 2.

1 Edge detection

The functions in this task use grayscale images as input. The included script does this conversion for you.

(a) Write a function that takes a grayscale image I and computes the horizontal and vertical gradient images by convolving with the 1-D central difference kernel:

$$\begin{bmatrix} +\frac{1}{2} & 0 & -\frac{1}{2} \end{bmatrix} \tag{1}$$

Return the two gradient images and the gradient magnitude image. To compute the gradient image I_u in the horizontal direction, convolve the kernel with each row of the input. Similarly, for the gradient image I_v in the vertical direction, convolve with each column. You can use `conv` (Matlab) or `numpy.convolve` (Python) for 1-D convolution. The magnitude image can be computed as $I_m = \sqrt{I_u^2 + I_v^2}$.

To make later tasks easier, ensure that the result images have the same size as the input. For this you need to handle border effects (Szeliski p. 101). A simple way to use the zero-padding strategy described in the book is to specify the `"same"` option when calling `conv` or `numpy.convolve`.

(b) Write a function that blurs a grayscale image by convolving with the Gaussian

$$\frac{1}{2\pi\sigma^2} \exp\left(-\frac{u^2+v^2}{2\sigma^2}\right) \tag{2}$$

taking the standard deviation parameter σ ("sigma") as the blur strength. Again, ensure that the result image has the same size as the input. Note that because it is separable, convolution with a 2-D Gaussian can be computed by convolving each row with a 1-D Gaussian, and then each column. The 1-D kernel is obtained from the 1-D Gaussian: $\exp(-u^2/2\sigma^2)/\sqrt{2\pi}\sigma$.

(c) Write a function that detects edges based on the gradient images and returns their pixel coordinates (u, v) and orientation θ. The orientation should be computed as the angle of the image gradient: $\theta = \angle[I_u \quad I_v] = \text{atan2}(I_v, I_u)$.

Note: The included script provides a solution based on thresholding the magnitude image. You may use this and be done with this task. Otherwise, you may want to try to implement the non-maximum suppression technique described in section 8.3.5 of Forsyth and Ponce to get better results.

(d) Test your functions on one of the included images and generate a figure showing the blurred image, the gradient direction and magnitude images, and the extracted edges. Apply the blur before computing the gradients. **Include the figure in your submission.**

Note: If you wrote your solutions using the provided script files, you should be able to run the `task1` script to generate the necessary figure.

2 The Hough transform

This exercise uses the orientation-aware version of the Hough transform described in Ch. 4.3.2 in Szeliski. Recall that in the original algorithm, a point votes for *all* possible lines passing through it. Instead of doing this, we'll take advantage of having computed the orientation of each detected edge, which together with the edge's position determines a unique line. In other words, an edge (u, v, θ) votes for a *single* line (θ, ρ) that passes through it. The equation for this line is

$$u \cos\theta + v \sin\theta = \rho \tag{3}$$

For the tasks below you should write your implementation in the `task2` script.

(a) Compute and display the Hough accumulator array for one of the sample images.

Start by determining the minimum and maximum value possible for θ and ρ. Keep in mind that your edges are in pixel coordinates, not normalized pixel coordinates, as they are in Szeliski's book.

Next, choose an appropriate resolution for the accumulator array (the number of bins used for θ and ρ to store votes). If the resolution is too low, the estimated lines can be wildly inaccurate; if the resolution is too high, the run time will make you sad and votes for one line can get split into multiple bins, weakening its response.

To compute the array you may follow the procedure described in Szeliski:

1. Create and set the accumulator array to zero
2. For each edge (u, v, θ), compute the value of $\rho = u \cos\theta + v \sin\theta$
3. Increment the accumulator bin corresponding to (θ, ρ)

Alternatively, since this is the same as computing a histogram, you can use the `histogram2d` function (in Numpy/Python) or `histcounts2` (in Matlab).

(b) Find the dominant lines in the image by finding peaks in the accumulator array. Peaks are points in $\theta\rho-$space where a large number of votes are clustered. In the Hough accumulator array, these peaks are local maxima.

A commonly used method to extract local maxima in an array is non-maximum suppression. You can use the provided function `extract_peaks` which implements this. You need to specify an appropriate neighborhood size and the minimum number of votes (threshold). Otherwise, you can look into better peak extraction methods (see Appendix J in Corke's book).

(c) Use the provided function `draw_line(theta, rho)` to draw the dominant lines back onto the input image. You need to convert the discrete row and column indices returned from `extract_peaks` into real θ, ρ values, using your chosen number of bins N_θ, N_ρ, and the minimum and maximum values of θ and ρ. Make sure that you don't confuse between rows and columns in the array and θ and ρ.

Include a figure showing the Hough accumulator array and the dominant lines in your submission.