COMPUTER VISION COURSE (2020/21)
Integrated Master in Electrical and Computers Engineering
Laboratory Assignment 1

Image Filtering and Hough Transform

Due Date : Sunday, March 14th, 2021, 23h59m

1 Learning Goals and Description

This is the first assignment of the course that will be evaluated by the instructors. In this assignment you will be implementing some basic image processing algorithms and putting them together to build a Hough Transform based line detector. Your code will be able to find the start and end points of straight line segments in images. Like most vision algorithms, the Hough Transform uses a number of parameters whose optimal values are data dependent, that is, a set of parameter values that works really well on one image might not be best for another image. By running your code on the test images you will learn about what these parameters do and how changing their values effects performance.

Many of the algorithms you will be implementing as part of this assignment are functions in the Matlab image processing toolbox. You are not allowed to use calls to functions in this assignment. You may however compare your output to the output generated by the image processing toolboxes to make sure you are on the right track.

Your submission for this assignment should be a zip file, `<ClassIDNamesID.zip>`, composed of your report, your Matlab implementations (including any helper functions), and, if the case, your implementations and results for extra credits (optional). Please make sure you include the `data/` and `result/` folders, the `houghScript.m` and `drawLine.m` scripts, and remove any other temporary files you generated.

Your final upload should have the files arranged in this layout:

`<ClassIDNamesID>.zip`

- `<NamesID>`
  - `<NamesId.pdf> - PDF Assignment report`
  - `Matlab`
    - `ImageFilter.m`
    - `EdgeFilter.m`
    - `HoughTranform.m`
    - `HoughLines.m`
    - `HoughLineSegments.m`
    - `Other functions needed`

Please make sure that any file paths that you use are relative and not absolute. Not `imread('/name/Documents/hw1/data/xyz.jpg')` but `imread('../data/xyz.jpg')`.

We have included a main script named `houghScript.m` that takes care of reading in images from a directory, making function calls to the various steps of the Hough transform (the functions that you will be implementing) and generates images showing the output and some of the intermediate steps. You are free to modify the script as you want, but note that Instructors will run the original `houghScript.m` while grading. Please make sure your code runs correctly with the original script and generates the required output images.

Every script and function you write in this section should be included in the `matlab/` folder. Please include resulting images in your report.

2 Assignment Pipeline

Convolution Write a function that convolves an image with a given convolution filter

```
function [img1] = ImageFilter(img0, h)
```

As input, the function takes a greyscale image (`img0`) and a convolution filter stored in matrix `h`. The output of the function should be an image `img1` of the same size as `img0` which results from convolving `img0` with `h`. You can assume that the filter `h` is odd sized along both dimensions.

The process of performing a convolution requires K^2 (multiply-add) operations per pixel, where K is the size (width or height) of the convolution kernel. In many cases, this operation can be significantly sped up by first performing a one-dimensional horizontal convolution followed by a one-dimensional vertical convolution (which requires a total of $2K$ operations per pixel). A convolution kernel for which this is possible is said to be *separable*. It is easy to show that the two-dimensional kernel K corresponding to successive convolution with a horizontal kernel h and a vertical kernel v is the *outer product* of the two kernels, $K = vh^T$. How can we tell if a given kernel K is separable? A direct method is to treat the $2D$ kernel as a $2D$ matrix K and to take its singular value decomposition (SVD),

$$K = \sum_i \sigma_i u_i v_i^T. \tag{1}$$

If only the first singular value σ_0 is non-zero, the kernel is separable and $\sqrt{\sigma_0}u_0$ and $\sqrt{\sigma_0}v_0^T$ provide the vertical and horizontal kernels.

You will also need to handle boundary cases on the edges of the image. For example, when you place a convolution mask on the top left corner of the image, most of the filter mask will lie outside the image. One solution is to output a zero value at all these locations, the better thing to do is to pad the image such that pixels lying outside the image boundary have the same intensity value as the nearest pixel that lies inside the image. You can call Matlab′s `padarray` function to pad your array or matrix. However, your code can not call on Matlab′s `imfilter`, `conv2`, `convn`, `filter2` functions, or any other similar functions. You may compare your output to these functions for comparison and debugging.

This function should be vectorized. Examples and meaning of vectorization can be found here. Specifically, try to reduce the number of `for loops` that you use in the function as much as possible.

Edge detection

Write a function that finds edge intensity and orientation in an image. Display the output of your function for one of the given images in the handout.

`function [img1] = EdgeFilter(img0, sigma)`

The function will input a greyscale image (`img0`) and scalar (`sigma`). `sigma` is the standard deviation of the Gaussian smoothing kernel to be used before edge detection. The function will output `img1`, the edge *magnitude* image.

First, use your convolution function to smooth out the image with the specified Gaussian kernel. This helps reduce noise and spurious fine edges in the image. Use Matlab′s `fspecial` function to get the kernel for the Gaussian filter. The size of the Gaussian filter should depend on `sigma` (e.g., `hsize = 2 * ceil(3 * sigma) + 1`).

The edge magnitude image `img1` can be calculated from image gradients in the x direction and y direction. To find the image gradient `imgx` in the x direction, convolve the smoothed image with the x-oriented Sobel filter. Similarly, find image gradient `imgy` in the y direction by convolving the smoothed image with the y-oriented Sobel filter. You can also output `imgx` and `imgy` if needed.

In many cases, the high gradient magnitude region along an edge will be quite thick. For finding lines it is best to have edges that are a single pixel wide. Towards this end, make your edge filter implement non-maximum suppression, that is for each pixel look at the two neighboring pixels along the gradient direction and if either of those pixels has a larger gradient magnitude then set the edge magnitude at the center pixel to zero. Map the gradient angle to the closest of 4 cases, where the line is sloped at almost 0^o, 45^o, 90^o, and 135^o. For example, 30^o would map to 45^o.

For more details about non-maximum suppression, please refer to the last page of this handout.

Your code cannot call on Matlab′s `edge` function, or any other similar functions. You may use `edge` for comparison and debugging. A sample result is shown in Figure 1.

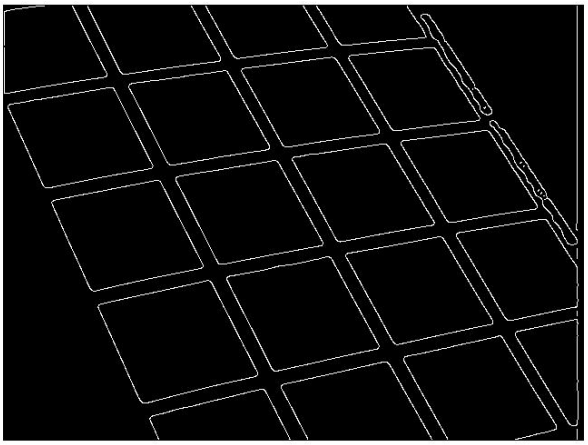

Figure 1: Edge detection result.

The Hough transform

Write a function that applies the Hough Transform to an edge magnitude image. Display the output for one of the images in the report.

```
function [H, rhoScale, thetaScale] =
HoughTransform(Im, threshold, rhoRes,thetaRes)
```

`Im` is the edge magnitude image, `threshold` (scalar) is a edge strength threshold used to ignore pixels with a low edge filter response. `rhoRes` (scalar) and `thetaRes` (scalar) are the resolution of the Hough transform accumulator along the ρ and θ axes respectively. `H` is the Hough transform accumulator that contains the number of ″votes″ for all the possible lines passing through the image. `rhoScale` and `thetaScale` are the arrays of ρ and θ values over which `HoughTransform` generates the Hough transform matrix `H`. For example, if `rhoScale(i)` $= \rho_i$ and `thetaScale(j)` $= \theta_i$, then `H(i,j)` contains the votes for $\rho = \rho_i$ and $\theta = \theta_i$.

First, threshold the edge image. Each pixel (x, y) above the threshold is a possible point on a line and votes in the Hough transform for all the lines it could be a part of. Parametrize lines in terms of θ and ρ such that $\rho = x \cdot cos\theta + y \cdot sin\theta$, $\theta \in [0, 2\pi]$ and $\rho \in [0, M]$. M should be large enough to accommodate all lines that could lie in an image. Each line in the image corresponds to a unique pair (ρ, θ) in this range. Therefore, θ values corresponding to negative ρ values are invalid, and you should not count those votes.

The accumulator resolution needs to be selected carefully. If the resolution is set too low, the estimated line parameters might be inaccurate. If resolution is too high, run time will increase and votes for one line might get split into multiple cells in the array. Your code cannot call Matlab′s `hough` function, or any other similar functions. You may use hough for comparison and debugging. A sample visualization of `H` is shown in Figure 2.

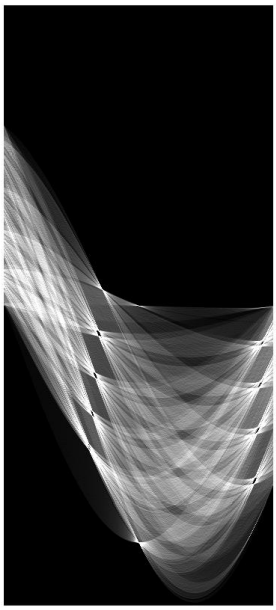

Figure 2: Hough transform result.

Finding lines

Write a function that uses the Hough transform output to detect lines,

```
function [rhos, thetas] = HoughLines(H, nLines)
```

where `H` is the Hough transform accumulator, and `nLines` is the number of lines to return. Outputs `rhos` and `thetas` are both `nLines`$\times 1$ vectors that contain the row and column coordinates of peaks in `H`, that is, the lines found in the image.

Ideally, you would want this function to return the ρ and θ coordinates for the `nLines` highest scoring cells in the Hough accumulator. But for every cell in the accumulator corresponding to a real line (likely to be a locally maximal value), there will probably be a number of cells in the neighborhood that also scored high but should not be selected. These non maximal neighbors can be removed using non maximal suppression. Note that this non maximal suppression step is different from the one performed earlier. Here you will consider all neighbors of a pixel, not just the pixels lying along the gradient direction. You can either implement your own non maximal suppression code or find a suitable function on the Internet (you must acknowledge and cite the source in your report, as well as hand in the source in your `matlab/` directory).

Once you have suppressed the non maximal cells in the Hough accumulator, return the coordinates corresponding to the strongest peaks in the accumulator.

Your code can not call on Matlab′s `houghpeaks` function or other similar functions. You may use `houghpeaks` for comparison and debugging.

Fitting line segments

Now you have the parameters ρ and θ for each line in an image. However, this is not enough for visualization. We still need to prune the detected lines into line segments that do not extend beyond the objects they belong to. This is done by `houghlines` and `drawLines.m`. See the script `houghScript.m` for more details. You can modify the parameters of `houghlines` and see how the visualizations change. As shown in Figure 3, the result is not perfect, so do not worry if the performance of your implementation is not good. You can still get full credit as long as your implementation makes sense.

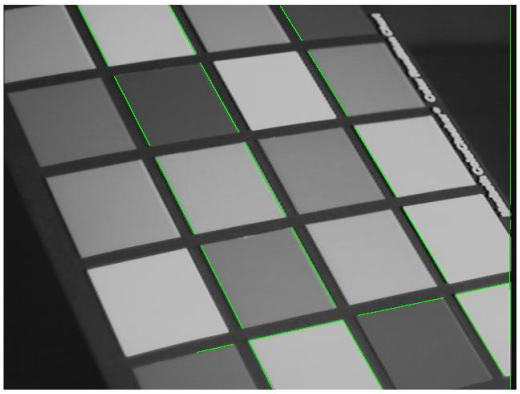

Figure 3: Line segment result.

3 Experiments

In folder `data/` you can find in a number of images for you to test your line detector code on. Use the script included to run your Hough detector on the image set and generate intermediate

output images. Include the set of intermediate outputs for one image in your report.

Evaluate the impact of the parameters of the algorithm in the final result.

- Did your code work well on all the images with a single set of parameters?
- How did the optimal set of parameters vary with images?
- Which step of the algorithm causes the most problems?
- Did you find any changes you could make to your code or algorithm that improved performance?

In your report, you should describe how well your code worked on different images, what effect do the parameters have and any improvements you made to your code to make it work better.

4 Non-maximmun supression

Non-maximum suppression (NMS) is an algorithm used to find local maxima using the property that the value of a local maximum is greater than its neighbors. To implement the NMS in 2D image, you can move a 3×3 (or 7×7, etc.) filter over the image. At every pixel, the filter suppresses the value of the center pixel (by setting its value to 0) if its value is not greater than the value of the neighbors.

To use NMS for edge thinning, you should compare the gradient magnitude of the center pixel with the neighbors along the gradient direction instead of all the neighbors. To simplify the implementation, you can quantize the gradient direction into 8 groups and compare the center pixel with two of the 8 neighbors in the 3×3 window, according to the gradient direction. For example, if the gradient angle of a pixel is 30^o, we compare its gradient magnitude with the north-east and south-west neighbors and suppress its magnitude if it$'$s not greater than these two neighbors.