DIGITAL IMAGE PROCESSING REPORT

MATLAB ASSIGNMENT 2

NIKOLAOS GIAKOUMOGLOU
AEM 9043

1.1 HOUGH TRANSFORM

ALGORITHM

myHoughTranform takes a binary image img_binary as an input (which is an output of an edge detector, e.g. Canny where 1 represent edges and every other element is 0) and calculates the accumulator matrix of Hough Transform H in the ρ,θ plain where ρ<ρmax (ρmax = sqrt($N1^2$ +$N2^2$), N1, N2 is the size of the binary image) and |θ|<90 degrees. In order to have intervals of ρ and θ, we use a step of *Drho* and *Dtheta* for ρ and θ respectively which represent the step in going from −ρmax to ρmax and from -90 to 90 degrees. Lastly, the function calculates the positions in (ρ,θ) of the n maximum elements of the accumulator matrix aka strongest lines, given in a nx2 matrix L (those pairs represent a line in the ρ,θ plain) and the number of pixels that do not belong in any of the n lines.

First, we convert the binary image to double, for easier handling and then we rotate the image because the following procedure calculates the accumulator matrix of the flipped and rotated image. We define N1, N2 the size of the image and then we define the θ and ρ as theta and rho as a vector of values from -90 to 90 with step *Dtheta* and from −ρmax to ρmax with step *Drho* (the intervals are different than the one from the notes; intervals match what MATLAB uses). Note that ρmax=*rhomax* is rounded before setting the intervals. In the Hough Transform, only the "1" vote so we find all the "1" of the binary image using the find method which by default finds all the non-negative values and sets x as the row, y as the column - (x,y) are the coordinates and val the value of the image (here val is a column vector of 1). After that we are ready to calculate the accumulator matrix but first, we initialize it to 0. In order to make the loop faster, we parallelize the loop for every 1000 elements. We set by first the start of the indexing and last the last of the index in the loop (note that in the last loop, the elements might be less than 1000 so last is the min for first+999 and length(x)). Then we replicate *x, y, val, theta* length(theta)-times which is (length(theta) copies of the initial vectors *x, y, val, theta* of size last−first (e.g. 1000) creating the *x_mat, y_mat, val_mat, theta_mat* and then we calculate *rho_mat* as the result of ρ=n1·cos(θ) + n2·sin(θ), where each row is the result of the pair (x_i,y_i) for every θ in theta. Lastly, we have to find the ρ and θ bin indexes defined as rho_bin_index, theta_bin_index, where θ is obvious and ρ occurs from the difference of *rho_mat* and rho(1)=−ρmax and update the accumulator H with the *rho_bin_index theta_bin_index*. The calculation of res is very simple: we find the 2 edge values of the lines – in the image's borders and then the length of the line equals to sqrt[((x_{end}−x_0)^2+(y_{end}−y_0)^2]. That way we might calculate some pixels twice, but the difference is insignificant.

DELIVERABLE_1

In order to make things a little clearer, we defined a class *myFunctions* with a set of functions.

The first function *plotHough*, takes a grayscale image BW, the coordinates of the lines in polar system L, and the set of ρ, θ that the Hough transform was calculated and prints the Hough transform in gray colors with the maximum (ρ,θ) in green squares as defined from the matrix L, which holds the (ρ,θ) of the n strongest lines.

The second function is the *plotLines*, which, given a grayscale image BW and a set of lines in polar system, plots the image, and next to it the image with the given lines in green. This function calculates the first and last pixel of the image in the horizontal dimension (x_0, x_{end}) and for every ρ,θ (degrees) prints the line from y_0 to y_{end} in vertical dimension using the following formula, with a special case if the line is vertical thus $\theta=0°$

$$y = \left(-\frac{\cos\theta}{\sin\theta}\right)x + \left(\frac{r}{\sin\theta}\right)$$

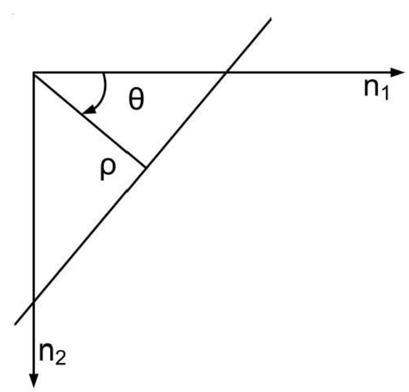

Figure 0

For the given images, we applied the algorithm by first converting the image to grayscale and then using an edge detector like Canny. By default, we used *Drho=1*, *Dtheta=1* and *n=5*. Results, after calls to functions *myHoughTransform*, *plotHough* ***and*** *plotLines* for *im.2.jpg* are as following:

Figure 1: Gray scale of `im2.jpg`

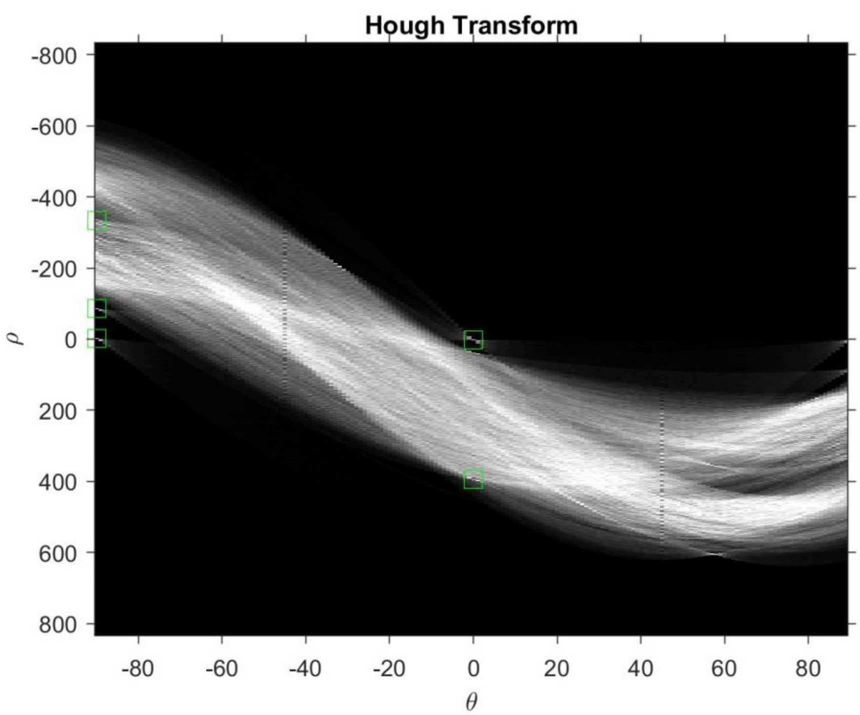

Figure 2: Hough Transform of `im2.jpg` in `(ρ,θ)` plain (green squares indicate the maximal of the accumulator)

Figure 3: 5 strongest lines of `im2.jpg`

COMPARISON TO MATLAB hough, houghpeaks and houghlines

Here we present the results of the build-in implementation of Hough's algorithm.

For details see links below:

https://www.mathworks.com/help/images/ref/hough.html

https://www.mathworks.com/help/images/ref/houghpeaks.html

https://www.mathworks.com/help/images/ref/houghlines.html

Results are as following:

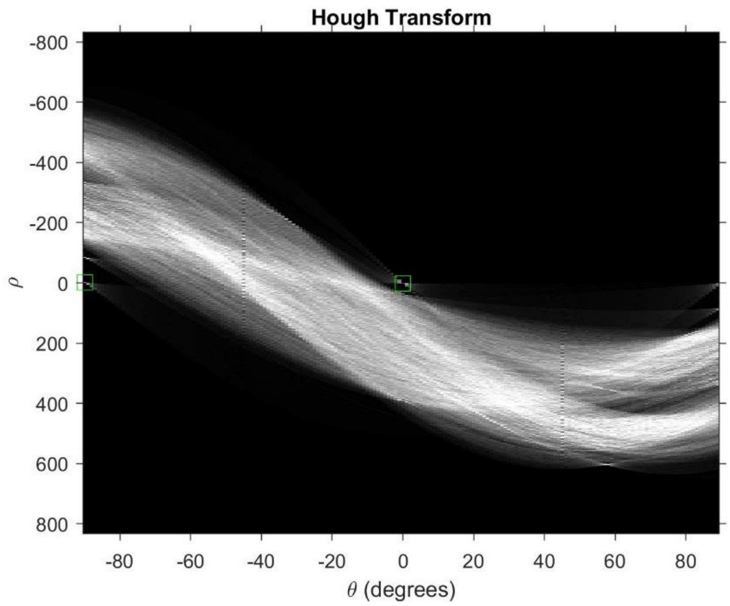

Figure 5: Hough Transfrom of `im.2jpg` in `(ρ,θ)` plane using MATLAB's `hough` function (green squares indicate the maximal of the accumulator)

Figure 6: 5 strongest lines of `im2.jpg` using MATLAB's `houghpeaks` and `houghlines` functions

1.2 HARRIS CORNER DETECTOR

ALGORITHM

`myDetectHarrisFeatures` takes a gray scale image as an input and outputs the corners of the image using Harris.

The algorithm is a simple implementation of the paper's procedure. In order to calculate the partial derivatives of the image in horizontal and vertical dimension, we choose a Prewitt mask defined as `Gx` and `Gy` respectively, where

$$\text{Gx} = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \text{ and } \text{Gy}=\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

We applied those two masks in our image `I` and took `Ix` and `Iy`. For next, we calculated Ix^2, Iy^2 and $Ixy=Ix \cdot Iy$. Then we arbitrarily defined a gaussian filter as a `9x9` mask with `sigma=2` and applied it to Ix^2, Iy^2 and `Ixy`. We are ready to calculate the `M` and `R` matrices. For that purpose, we iterate through every index of the image and set

$$M = \begin{bmatrix} Ix(n1,n2)Ix(n1n2) & Ixy(n1,n2) \\ Ixy(n1,n2) & Iy(n1,n2)Iy(n1,n2) \end{bmatrix}$$

$R(n1,n2) = \det(M) - k \cdot \text{trace}(M)^2$, `k` is set arbitrarily as a very low number

While iterating through each pixel, we simultaneously trace the maximum value of `R`, denoted by `Rmax`.

Now we are ready to choose among all pixels, which one are true corners. For that reason, we have a threshold at $0.1 \cdot Rmax$ and additional 9 requirements that the pixel is a local maximum on a `3x3` neighborhood

DELIVERABLE_2

We applied our alogirthm to the image `img2.jpg` with the following results. Note that we did not red-out a `5x5` squared as asked in the assignment, but instead we marked the pixel with a star (*). The image is gray-scaled to better understand the corners.

Figure 8: Corners of `img2.jpg` using `myDetectHarrisFeatures`

COMPARISON TO MATLAB detectHarrisFeatures

To check our algorithm, we compare the `im2.jpg` with corners of our implementation (red) with corners of MATLAB's build-in function `detectHarrisFeatures`. We observe that our implementation finds less corners, but they are precise.

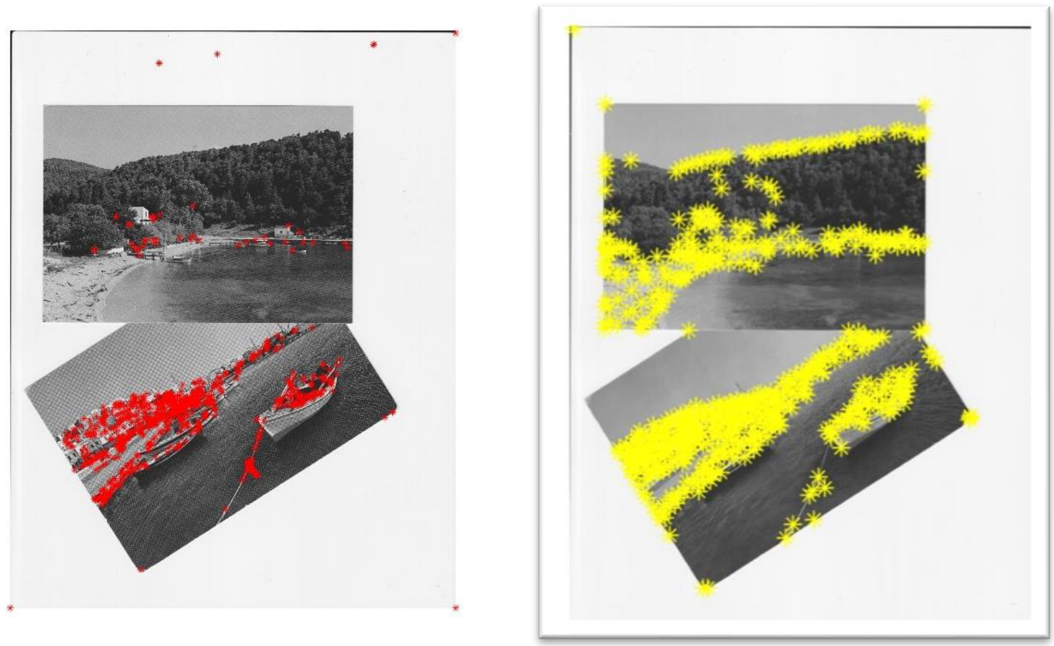

Figure 10: Corners of `img2.jpg` using `myDetectHarrisFeatures` (left) and `detectHarrisFeatures` (MATLAB, right)

1.3 IMAGE ROTATION

ALGORITHM

myImgRotation takes an image RGB or grayscale as an input alongside with an angle and rotates that image by angle degrees.

In our implementation, first we calculate the new dimensions `M1, M2` of the rotates image such that the rotated image fits in exactly. We use the absolute, so we get an absolute value in any case. The new image is initialized as an RGB of dimensions `[M1 M2 N3]` where `N3` is 1 if the image is black and white and 3 if the image is RGB and then we initialize all the values to 0(black). Before we start iterating through the main loop, we are going to need the center of the initial image x_0, y_0 and the center of the rotated image x_{final}, y_{final}. In the loop, we calculate the corresponding coordinates of pixel of the image for each pixel of the rotated image, and its intensity will be assigned after checking whether it lies in the bound of the original image. More specifically, each pixel (x_{new}, y_{new}) of the rotated image is

$$x_{new} = x_0 + (x-x_{final})\cdot\cos(\theta) + (y-y_{final})\cdot\sin(\theta)$$
$$y_{new} = y_0 - (x-x_{final})\cdot\sin(\theta) + (y-y_{final})\cdot\cos(\theta)$$

where (x, y) is the pixel of the initial image.

DELIVERABLE_3

We applied our implementation for the image rotation to `img2.jpg`. The algorithm is significant slower in comparison to MATLAB's build-in function `imrotate`, the rotated image is always alike though. Results are as following:

Figure 11: Rotation of `img2.jpg` by 54 degrees

Figure 12: Rotation of `img2.jpg` by 213 degrees

2. LAZY SCANNER

An important assumption before proceeding, is that the background of the scanned photos is white.

In the last part of our assignment, we combine the output of the functions implemented so as to distinguish sub-images in one image, crop them, rotate them, and save them as new images with suffix _1, _2, etc.

For that reason, we will use the `myFunctions.m` (additional m-file) and `myLazyScanner.m`.

Before we expain our algorithm, we defined a set of functions to do some pre-work and for better understanding of the code. Those functions are declared inside `myFunctions.m`, inside a class called `myFunctions`.

We already explain the use of `plotLines` and `plotHough` so they are assumed as known. The first function is `houghTransform`. This is an implementation of `myHoughTransform` but using the build-in functions `hough`, `houghpeaks` and `houghlines`. The reason we will not use the function we made is that our implementation was made later and it was an obstacle for moving forward. We will ignore the function `findEdges` due to the fact that is not used. However, it is a well-structured function that tries to find the corners of an image after binning it with a threshold of 0.90. Next we created the function `removeWhiteLines` that searches each row and column of an image and deletes it, if the mean of that row or column respectively is higher than a threshold, defined arbitrarily at 240. In a similar way we have defined a function `removeBlackLines`. The function `makeBackgroundWhite` searches for total black spots (brightness equal to 0) and make them total white (brightness 255). This faction is called after a rotation of an Image to turn the background from black by default to white, because the printers background is also white. More details about that later. The function that is most important for our algorithm is `removeBadLines`. This function identifies lines that are near the edges, e.g. up of an image, down of the image, right or left of the image and deletes them. We will refer to those lines as "bad" lines and to the rest of them as "good" lines. We do not wish to cut the image in a "bad" line because we will get a "white" image and a smaller image of the initial. For that reason, we have other functions to do that work (see `removeWhiteLines`). This function is our key and could be implemented to seek lines that cut a sub-image in the middle which means it's bad line. However due to the complexity of that, the current implementation satisfies our input and we will keep it as it is. Another simple function is `removeDuplicateLines` that deletes lines with the same rho and theta. The final function that we are going to need is `rotImageFinal`. This function is called after we have found all the sub-images of the initial image and will try to rate them further using Hough Transform and the strongest line. This is another important function that could find the orientation of the image but we will not use it. We are ready to explain our solver denoted as recursiveFinder3 (3 because it was the 3rd implementation of the algorithm). The inputs are the image – BW, a flag named `edgeFlag` that either uses a binary image as the input to the edge detector (`edgeFlag`=1) or the image itself (`edgeFlag`=0). This flag exists for the only reason that the edge detector might have better results if the image is binary and thresholded to 0.90 arbitarily. This was a “black” rectangle appears where the sub-image is, and the Hough transform might find lines in this case, where we will not find otherwise. The step `Drho` and `Dtheta` are also denoted as an input, however we always set them equal to 1 for simplicity. The last input is a 3-D array `IMAGES_FOUND` where all the sub-images that are identified by the algorithm are stored in the 3rd dimension, hence `IMAGES_FOUND(:,:,i)` denote the sub-image `i`. The number of sub-images found is `NO_IMAGES_FOUND`. The last attributes are also the output of the algorithm. The reason an input is an output as well is due to the fact that the algorithm is recursive, and we want to store any finding of a sub-tree of the search. Before we start searching, we check that the n is not zero (just to be sure) and the possibilities than the sub-image we are about to check is almost full white, hence the mean of the brightness is higher than a threshold `whiteGarbageThreshold` e.g. 230. We also check if the sub-image has indeed a content which occur if we binarize the image and we have enough “black” pixels. In that cause we add our image to the `IMAGE_FOUND` array and we increase the `NO_IMAGES_FOUND` by 1. After those checks, we remove any possible white or black lines (useless pixels) and we apply the Hough Transform to find the lines, either using the edge detector to the initial image or to the binarized image. The use of the binarized image as an input to our algorithm can only happen once in the first call. Any other call that happens recursively, uses `edgeFlag`=0. As explained previously, Hough’s lines might have “garbage” lines, so we get rid of them by calling `removeDuplicateLines` and `removeBadLines`. Before we iterate through every line, we set the value of the attribute `isEnd` equal to the number of lines. The reason we do that is that in case we don’t have any lines, or we do not call recursively the algorithm the image we currently examine might be sub-image with no other sub-images. Inside our main loop, we test 3 cases: if a line is vertical, horizontal or none of them. In every case we crop the image into 2 sub-images that must be further examined and call our algorithm to test them. If line has a slope different that 0 or 90 (and -90 since theta is between -90 and 90), we rotate the image and call the algorithm again, but in this case with n reduced by 1. The reason we do that is because the algorithm might keep calling itself and keep rotating an image (sloppy implementation but a loophole is found). After all the loops are finished, we are ready to judge the image. If in the end of the loop (or in the root of the tree since we can imagine the recursiveness of the algorithm a tree) if we do not have any “good” lines, due to the fact that all the lines are around the image and hence the function `deleteBadLines` delete them all, we can safely say that is surely a sub-image and we can add it to IMAGES_FOUND array. Note that if we do not find any “good” lines the initial image will be the only sub-image found but we will deal with it in our main script `myLazyScanner.m.`

In our main script, the user adds the name of the image without the suffix (we assume the suffix is .jpg). Then we read the image (we assume that the image is in the current directory) and resize it to 10% of the original just because of the time complexity of the algorithm (the given images are high dimensional). We define the set `IMAGES_FOUND` and `NO_IMAGES_FOUND` and initialize our parameters with some values. The important choice is the n. We choose n=5 because in the latter case, the Hough transform will identify the 4 edges of the image and 1 line that separates the sub-images. We only need 1 line for the first call, because the algorithm will call itself recursively with n=5 again and might identify sub-sub-images (did not try it though). In the next section we call the recursiveFinder3. If the finder fails to find any image, we call it with the flag `edgeFlag` equal to 1 and try again. We save the images to a set `ALL_IMAGES` for easier handling later where `ALL_IMAEGS{i}` represent the `i` sub-image. We mentioned before that the algorithm might return the initial image but also it might return the same image twice or more. That happened because of the calls to rotation. The best implementation would crop the image to 2 sub-images and call itself recursively for each sub-image so no duplicates would appear. But we rotate the image and call the function with n reduced by 1 because of simplicity. For that reason, we are obligated to search for duplicates, or the initial image repeated. We use the corr2 function to check the correlation of 2 images. We iterate through every image we found plus the initial. In case the correlation `R` is higher that a threshold, here arbitrarily is set to 0.5 after trial and error (of course we cannot expect `R`>0.9 if an image is the same with another, so we have lower expectations), we delete the image. After that we are in the position to plot the sub-images we fund and save them with a suffix _1, _2, etc. In case no sub-images are found, it is plotted in the screen. We should mention that in the screen we can see the calls to our algorithm and the correlation among the images found.

Update: After implementing the Hough Transform from 1.1, we can use our implementation to identify sub-images. We change the above algorithm to `recursiveFinder4`. This algorithm behaves differently than `recursiveFinder3` due to the fact that different Hough lines are identified. The effectiveness of the algorithm relies on the ability of the Hough transform to find "good" lines. For our `LazyScanner` we are going to use `recursiveFinder3` for the first and second if needed try (if the first time the number of images found is 1, then we call the finder again with `edgeFlag=1`). If `recursiveFinder3` fails (it fails in `im2.jpg`) then we call `recursiveFinder4`, twice if needed aswell. If that fails too, we display the message of failure (it succeeds in `im2.jpg`).

Results are as following (with a white outline added from word):

Figure 11: `im1_1.jpg & im1_2.jpg`

Figure 12: `im2_1.jpg & im2_2.jpg`

Figure 13: `im3.jpg`

Figure 14: `im4_1.jpg & im4_2.jpg`

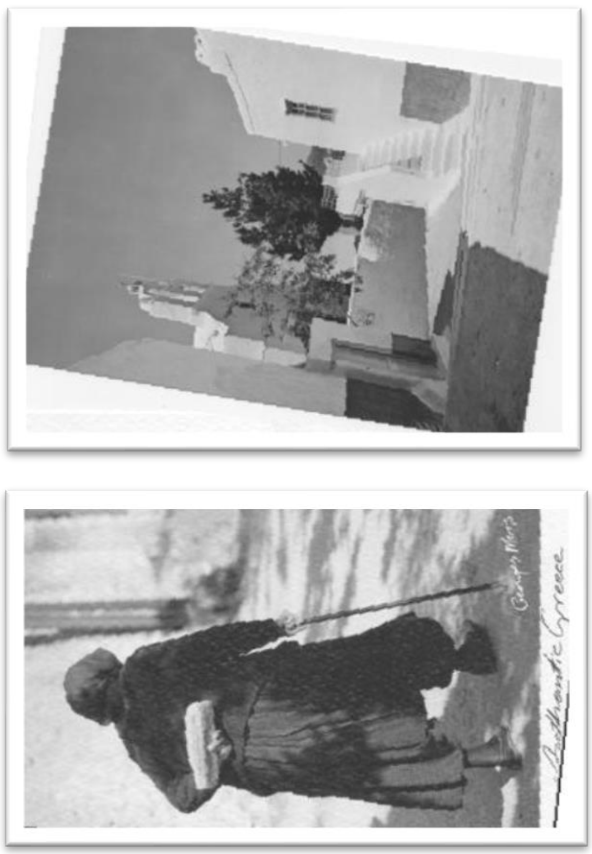

Figure 15: `im5_1.jpg & im5_2.jpg`

In order to understand how effective each algorithm is (`recursiveFinder3,4`) based on the Hough lines that are identified, we present those lines for the initial image. Of course, due to the recursive nature of the algorithm, new lines can be found in sub-images. For example, `recursiveFinder3` fails in `im2.jpg` but succeeds in `im4.jpg` where `recursiveFinder4` fails.

Figure 16: Hough lines of `im1.jpg` (MATLAB left, `myHoughTransform` right)

Figure 17: Hough lines of `im2.jpg jpg` (MATLAB left, `myHoughTransform` right)

Figure 18: Hough lines of `im3.jpg jpg` (MATLAB left, `myHoughTransform` right)

Figure 19: Hough lines of `im4.jpg jpg` (MATLAB left, `myHoughTransform` right)

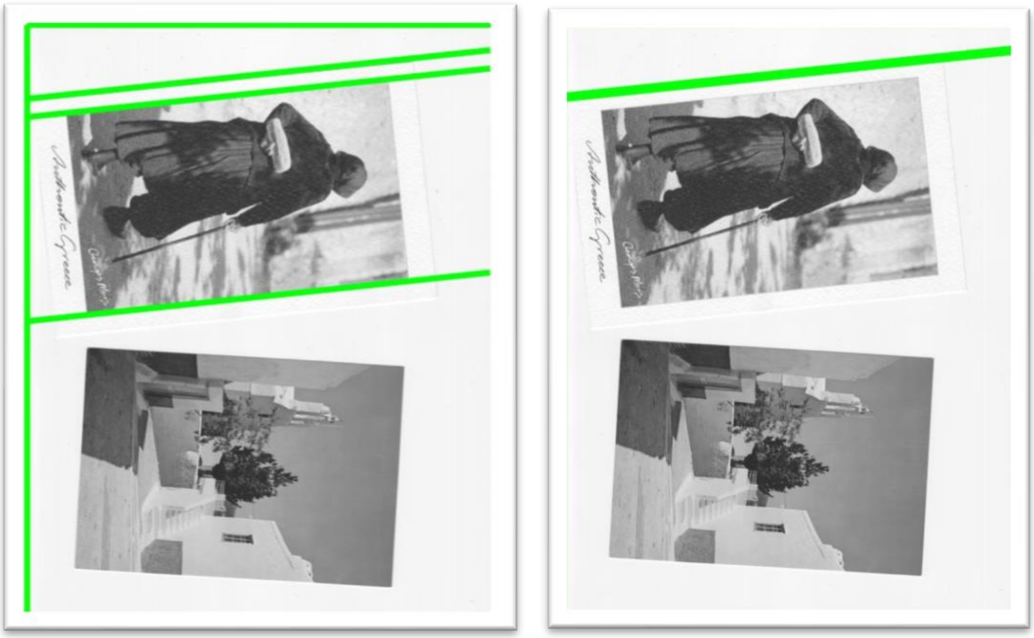

Figure 20: Hough lines of `im5.jpg jpg` (MATLAB left, `myHoughTransform` right)

One can try how effective is each algorithm separately, but removing the other algorithm from the MATLAB script `myLazyScanner.m` (lines 18-38).