**1 Edge Detection**

* 1. **Reading the Image**

We used our function “imread\_normalized\_1” to read and normalize the “camaraman.tif” image. You can see our function, it is the same as in Ex3, but we excluded the line “%I = rgb2gray(I);”:

function I\_normalized = imread\_normalized\_1(src)

I = imread(src);

%I = rgb2gray(I);

I = im2double(I);

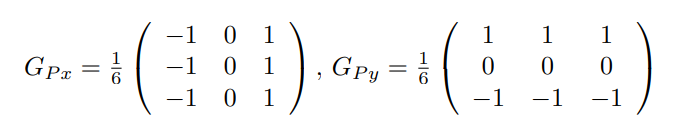
I\_min = min(I(:));

I\_max = max(I(:));

I\_normalized = (I - I\_min) / (I\_max - I\_min);

end

* 1. **Prewitt Edge Detector**

1.1.2 We wrote our own function named dip\_prewitt\_edge(img,thresh) that will apply the Prewitt edge detector on ’img’ and output the edge image with the same size as the input image. The ’thresh’ parameter will determine the gradient magnitude cutoff threshold.

The function is designed to perform edge detection on an input image using the Prewitt operator. This operator utilizes two convolution kernels above, prewitt\_kernel\_x and prewitt\_kernel\_y, to compute the horizontal and vertical gradients of the image, respectively. The resulting gradients are combined to calculate the gradient magnitude, representing the strength of edges in the image. The function then applies a user-specified threshold to the gradient magnitude, producing a binary edge image. Pixels with gradient magnitudes above the threshold are considered part of an edge and set to 1, while others are set to 0. The output is a binary image highlighting significant edges in the input image based on the chosen threshold.

function edge\_image = dip\_prewitt\_edge(img, thresh)

% Prewitt filter kernels for horizontal and vertical edges

prewitt\_kernel\_x = (1/6)\*[-1, 0, 1; -1, 0, 1; -1, 0, 1];

prewitt\_kernel\_y = (1/6)\*[-1, -1, -1; 0, 0, 0; 1, 1, 1];

% Convolve the image with the Prewitt kernels

gradient\_x = conv2(img, prewitt\_kernel\_x, 'same');

gradient\_y = conv2(img, prewitt\_kernel\_y, 'same');

% Compute the gradient magnitude

gradient\_magnitude = sqrt(gradient\_x.^2 + gradient\_y.^2);

% Apply threshold to the gradient magnitude

edge\_image = uint8(gradient\_magnitude > thresh) \* 255;

end

1.2.2 Displaying 2 edge images generated using dip\_prewitt\_edge(img,thresh) with 2 different thresholds 0.05 and 0.15:



As you can see in the images above, using a lower threshold results in more edges being considered in the edge map. In The image of threshold=0.05 we can see more lines representing edges of vegetation and background buildings for example than in the image with threshold=0.15.

* 1. **Canny Edge Detector**

1.3.1. We read about the MATLAB function edge(I,’Canny’).

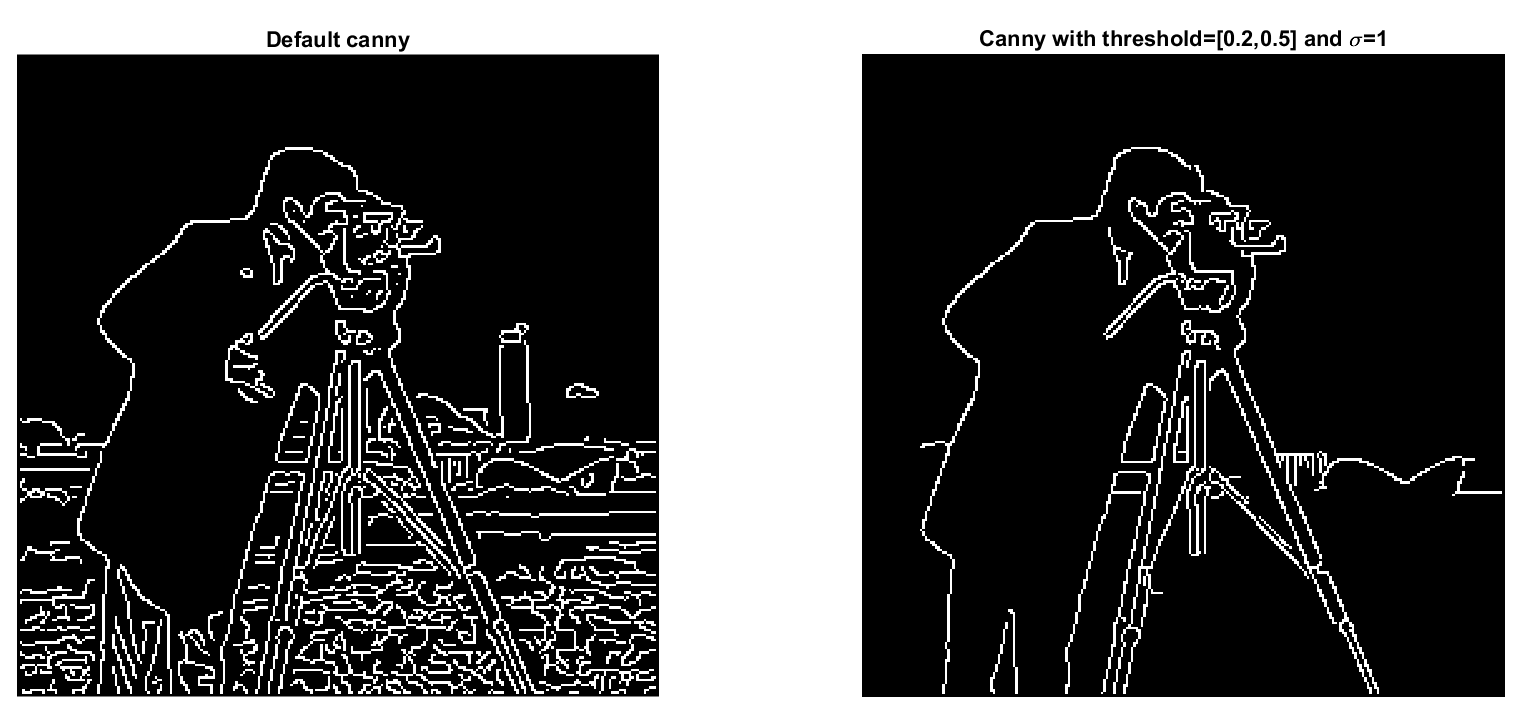
The process of Canny edge detection algorithm can be broken down to five different steps:

1. Apply **Gaussian filter** to smooth the image in order to remove the noise.
2. **Find the intensity gradients** of the image.
3. **Apply gradient magnitude thresholding** or lower bound cut-off suppression to get rid of spurious response to edge detection.
4. Compare the edge strength of the current pixel with the edge strength of the pixel in the positive and negative gradient directions.
5. If the edge strength of the current pixel is the largest compared to the other pixels in the mask with the same direction (e.g., a pixel that is pointing in the y-direction will be compared to the pixel above and below it in the vertical axis), the value will be preserved. Otherwise, the value will be suppressed.
6. **Apply double threshold** to determine potential edges.  
   Pixels with a gradient higher than the high threshold are marked as strong edges, those between the high and low thresholds are marked as weak edges, and those below the low threshold are suppressed.
7. **Track edge by hysteresis**: Finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges.  
   Strong edge pixels are essential in the final image, representing true edges. For weak edge pixels, connectedness to strong edges is considered. Blob analysis examines a weak edge pixel and its 8-connected neighbors. If at least one strong edge pixel is part of the blob, the weak edge is preserved and considered a strong edge. This identifies weak edge pixels from true edges, eliminating those associated with noise or color variations.

**Optional parameters for the 'Canny' method:**

**threshold:** Specifies the threshold for edge detection. It is a two-element vector [low high], where edges with gradient magnitudes within this range are considered. **Default values: [0.1 0.2]**

**sigma:** Specifies the standard deviation of the Gaussian filter applied to the image before computing gradients. A larger sigma results in a smoother image and potentially detects longer edges. **Default value: 1.0**

1.3.2. We used the MATLAB functions edge(I,’Canny’) with two sets of parameters: the default set and another set threshold=[0.2,0.5] and sigma=1 :

We believe our version is better because it's more concise. The default one has too many distracting edges that aren't important. Our version keeps things simple and focuses on the main details, making it less cluttered and easier to understand the image. A lot of edges were suppressed due to the increase in the higher threshold to 0.5.

1.3.3. We achieved better results with our settings in 1.3.2.

**2 Hough Transform**

**1. Hough line transform**

2.1.a. We read the floor.jpg image, converted it to grayscale and normalize it to [0,1] using our function “imread\_normalized()”. You can see our function, it is the same as in Ex3 :

function I\_normalized = imread\_normalized(src)

I = imread(src);

I = rgb2gray(I);

I = im2double(I);

I\_min = min(I(:));

I\_max = max(I(:));

I\_normalized = (I - I\_min) / (I\_max - I\_min);

End

2.1.b. The edge detector that is used by default is the “Sobel edge detection model” (as written in the function documentation).

**Threshold:**

It is a scalar threshold value that determines the sensitivity of the edge detection. Edges with gradients above this threshold are considered.

**Direction:**

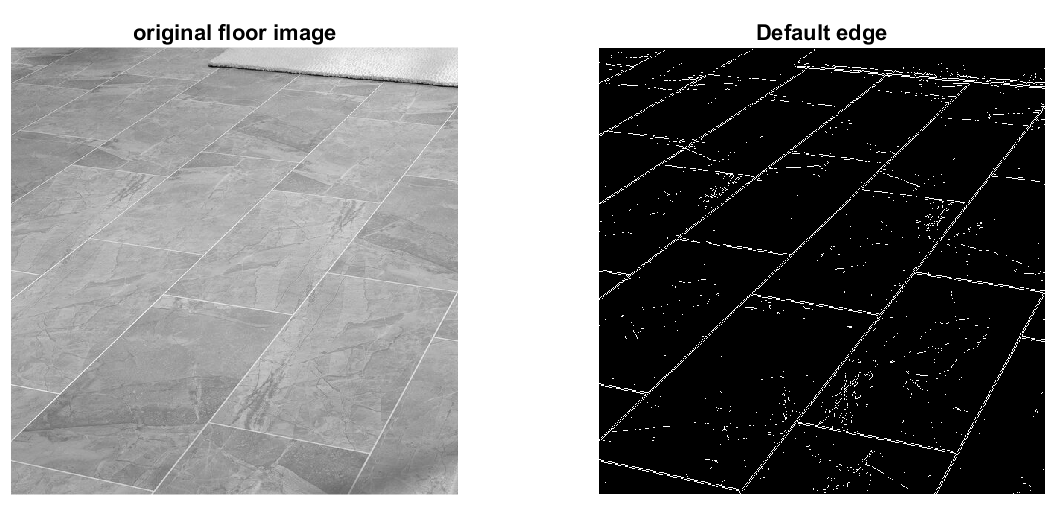
Specifies the direction of the Sobel operator. It can be 'horizontal', 'vertical', or 'both' to indicate the desired direction of edge detection.

The function’s default parameters are:

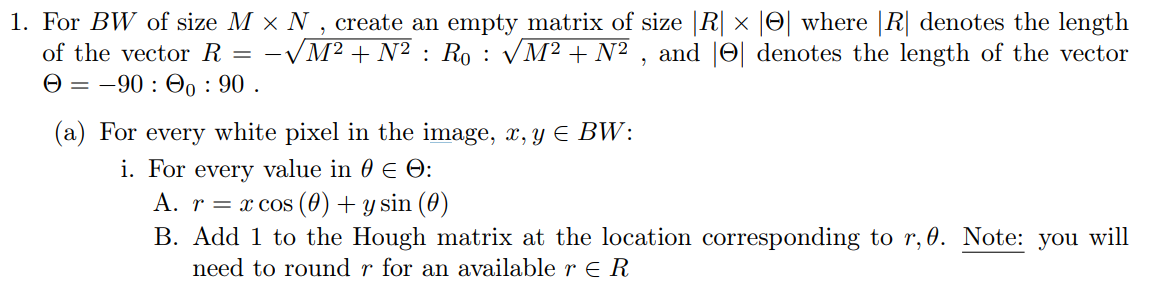
- Method = “Sobel edge detection model”

- Threshold = edge chooses the default threshold heuristically, depending on the input data. The best way to vary the threshold is to run edge once, capturing the calculated threshold as the second output argument. In our case TH = 0.0565.

- Direction = horizontal and vertical.



2.1.c. We wrote our own n dip\_hough\_lines(BW,R0, Θ0) function that calculates the Hough Matrix for finding lines. We followed the abstract algorithm mentioned below this section:



And in a simplified writing:

1. Creates an empty Hough matrix of size |R| × |Θ|.
2. For every white pixel in the binary image, it calculates r for every value in Θ.
3. Adds 1 to the corresponding cell in the Hough matrix.

Our function:

function HoughMat = dip\_hough\_lines(BW, R0, teta0)

[M, N] = size(BW); % Get the size of the input image

% Quantize parameter space

R = fix(-sqrt(M^2 + N^2):R0:sqrt(M^2 + N^2)); % R range

theta = fix(-90:teta0:90); % θ range in degrees

HoughMat = zeros(length(R), length(theta)); % Create Accumulator Array initialized to 0

% Loop over each pixel in the image

for x = 1:M

for y = 1:N

if BW(x, y) == 1 % Check if the pixel is an edge pixel

% Loop over all possible θ values

for t = 1:length(theta)

% Calculate the corresponding r for the current (x, y) and θ

r = fix(x \* cosd(theta(t)) + y \* sind(theta(t)));

% Find the index in R corresponding to the calculated r

rho\_index = find(r == R);

% Increment the accumulator array for the current (r, θ) pair

HoughMat(rho\_index, t) = HoughMat(rho\_index, t) + 1;

end

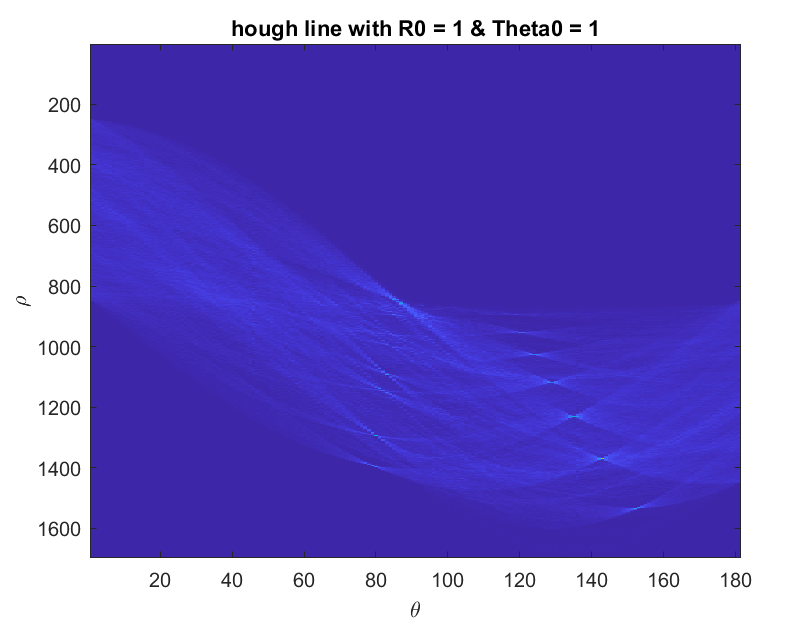
end

end

end

end

2.1.d. Using imshow(M,[]) did work but the result was less visible for us that's why we used imgsec. In the code both options appear, here we display the results with imgsec:



A blue and white graph

Description automatically generated

**\*\*\*\* Explain the results\*\*\*\*\***