**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);

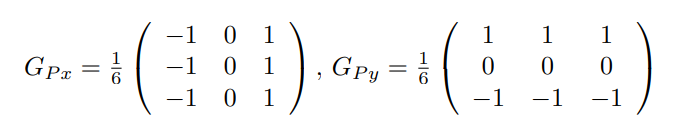
I\_min = min(I(:));

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’).

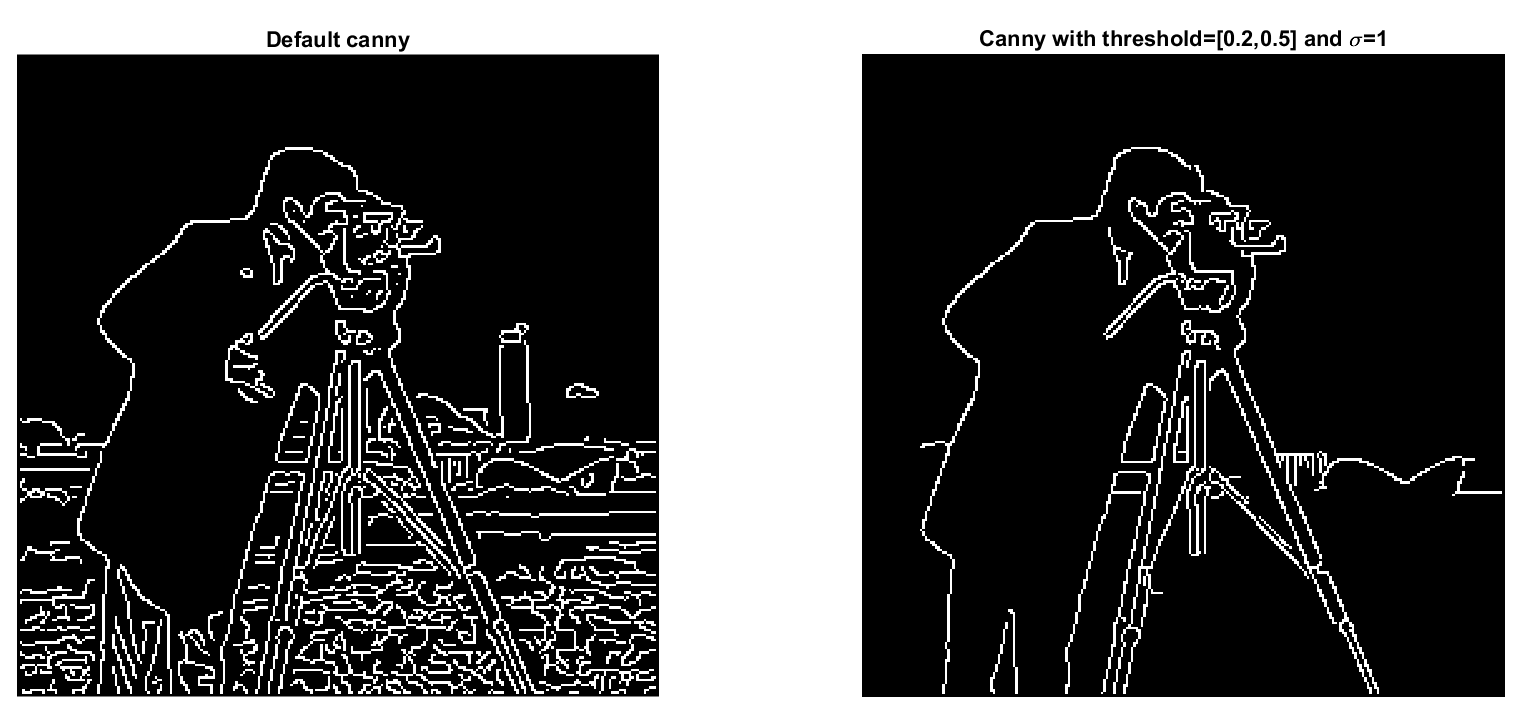
The process of Canny edge detection algorithm can be broken down to five different steps:

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:**

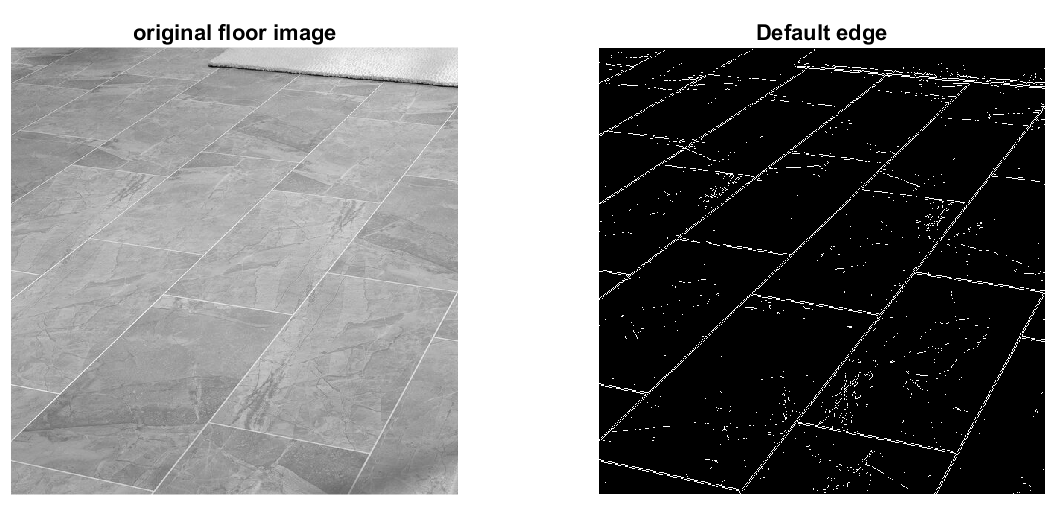
Specifies the direction of the Sobel operator. It can be 'horizontal', 'vertical', or 'both' to indicate the desired direction of edge detection.

The function’s default parameters are:

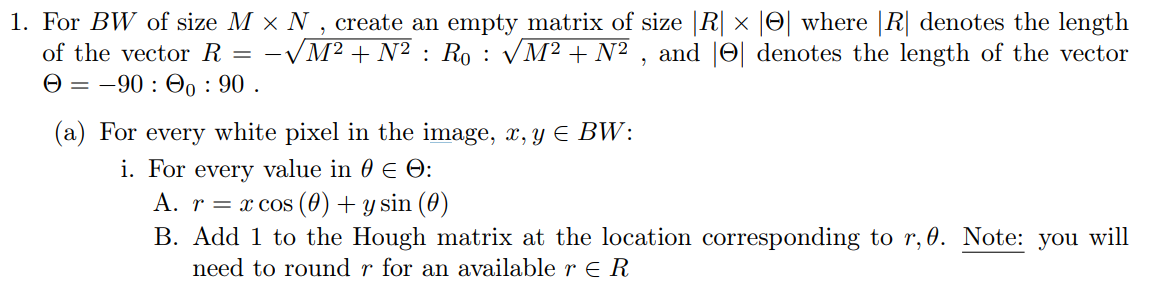
- Method = “Sobel edge detection model”

- 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

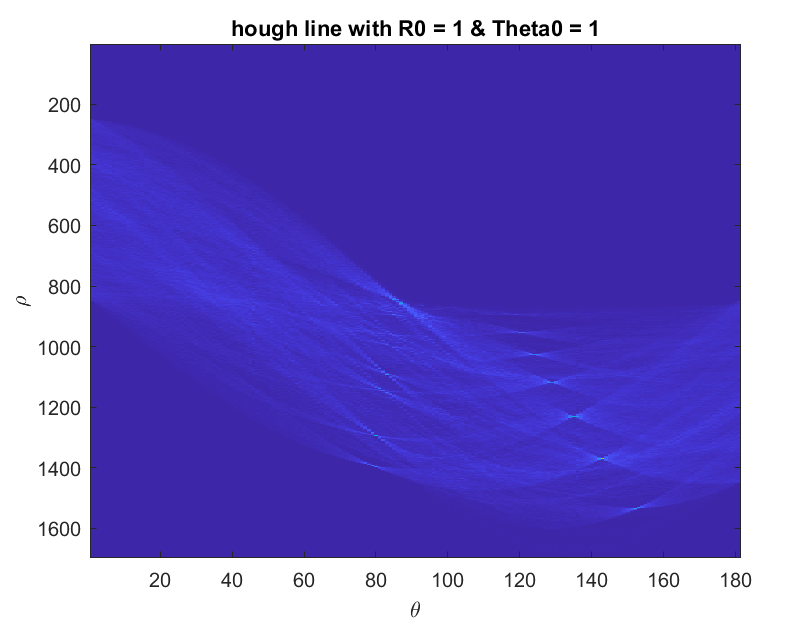
end

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

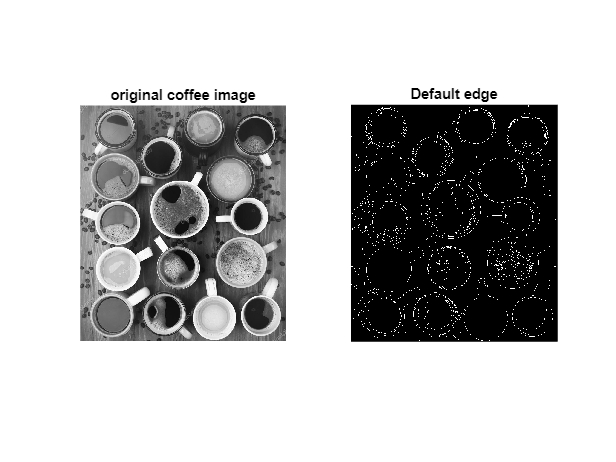
Description automatically generated

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

2.2 Hough circle Transform

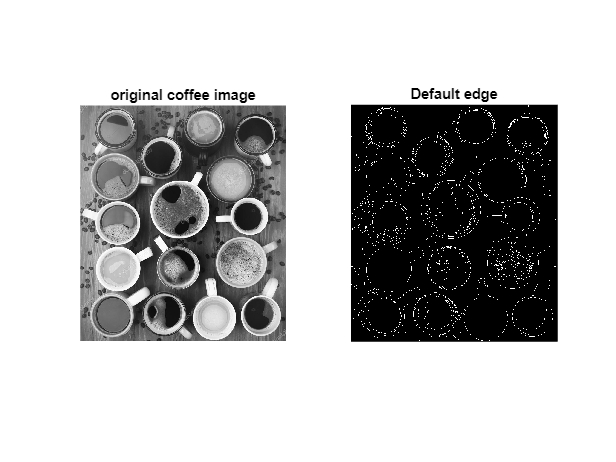
1. **read the coffee image, converted it into grayscale image and normalize it**

we used the `` imread\_normalized`` function as previously, to get the grayscale representation of the coffee image



#### Extract the edges using MATLAB’s edge()

#### Here is the result of the edge function, compared to the original image:



#### (c) Write your own dip\_hough\_circlesfunction that calculates the Hough Matrix

#### The function ``dip\_hough\_circles(BW, R\_0, theta\_0)`` generates a 3D Hough matrix, as thaught in class. This function uses the Hough formula to guess the responsibility of a circle of radius R, and origin (a,b) with every point (x,y) on an edge image. Here, R is adjusted to range from 80 to 100, following the question's guidelines.

#### For each point (x,y) with a non zero entry in the edge image (meaning there is an edge there) , the function maps out all possible circles.

#### for every radius value within the specified range, multiple points will "vote" making the 3D point representing it "pop", in other words, making a "peak".

#### This process is repeated for all edge points, indicating circles of parameters R,a,b, resulting in a 3D matrix.

#### Our implementation of the circle Hough transform is as follows:

%Iris Eting 209027333

%Nadav Orenstein 312349509

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

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

% Define the ranges for the parameters a (x-center), b (y-center), and r (radius)

A = 1:M; % Range of x-center

B = 1:N; % Range of y-center

R = fix(80:R0:100); % Range of radius

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

cos\_vals = cosd(theta);

sin\_vals = sind(theta);

% Create Accumulator Array initialized to 0

HoughMat = zeros(length(A), length(B), length(R));

% Loop over each edge 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 radius values

for rIndex = 1:length(R)

r = R(rIndex);

% Loop over a range of angles to cover the entire circle

for t = theta

% Calculate potential center (a, b) for this radius and angle

a = fix(x - r \* cosd(t));

b = fix(y - r \* sind(t));

% Check if the calculated center is within bounds

if a >= 1 && a <= M && b >= 1 && b <= N

% Increment the accumulator for the current (a, b, r) triplet

HoughMat(a, b, rIndex) = HoughMat(a, b, rIndex) + 1;

end

end

end

end

end

end

1. **Measure the Run-Time of your function**

tic;

H\_circles\_1 = dip\_hough\_circles(BW2, 1, 1);

toc;

Elapsed time is 9.533131 seconds.

tic;

H\_circles\_2 = dip\_hough\_circles(BW2, 4, 10);

toc;

Elapsed time is 0.387609 seconds.

tic;

H\_circles\_speedup =dip\_hough\_circles(BW2, 20, 1);

toc;

Elapsed time is 0.578993 seconds.

Choosing r0=1 and theta0=20 we got a good time-quality tradeoff. By keeping theta0 low, we save the angular resolution, preserving the quality of circle detection. On the other hand, increasing R0,the algorithm needs to iterate over fewer radii, which saves computation time.

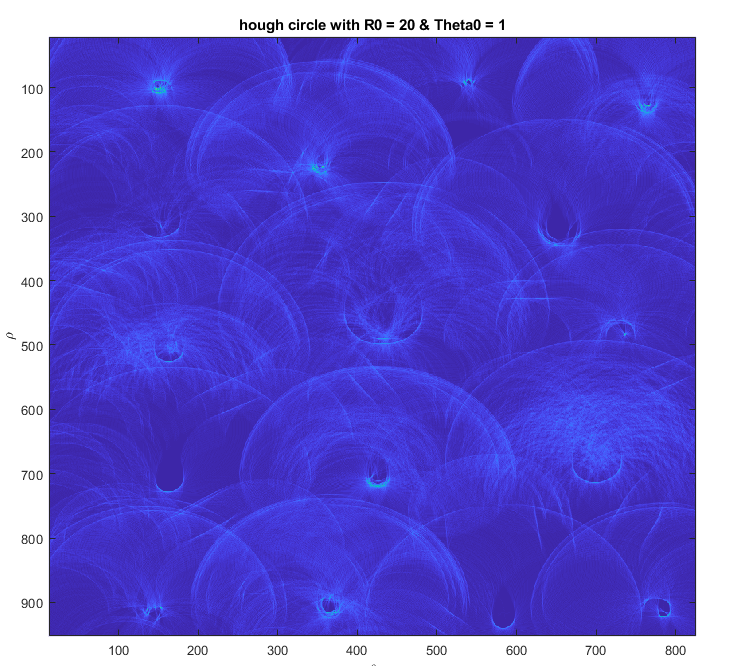
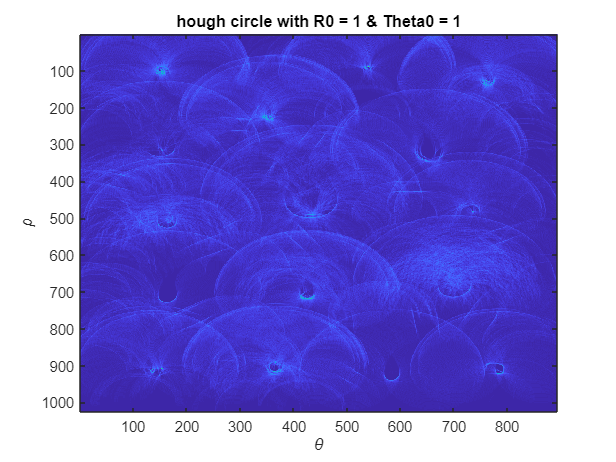
These values gives significant computational efficiency without compromising the detection quality.

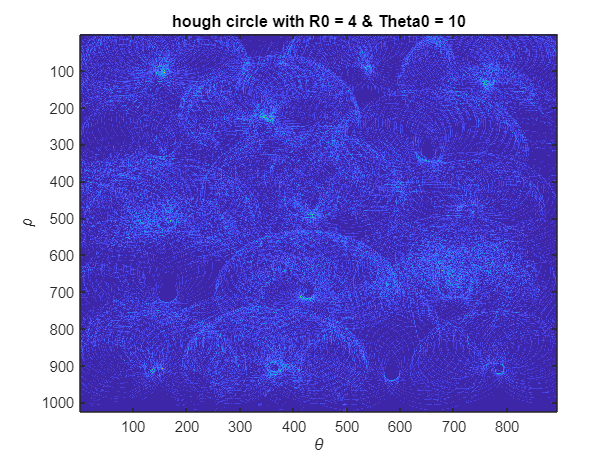
The speedup is roughly 10/0.5 = 20 (!)

We can be impressed by the quality of the results below:

**(e)display one slice (2D image)**

To visualize the Hough matrix,lets display one 2D slice from the matrix (at index 1)

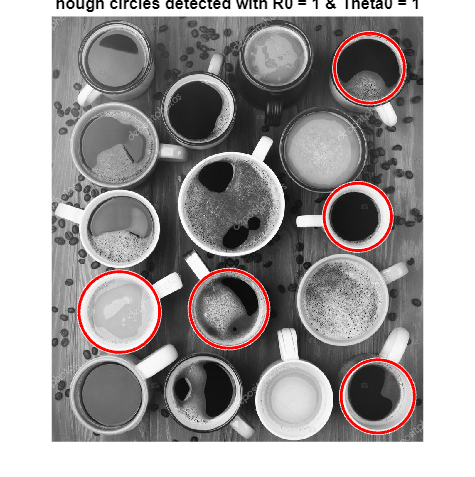
****

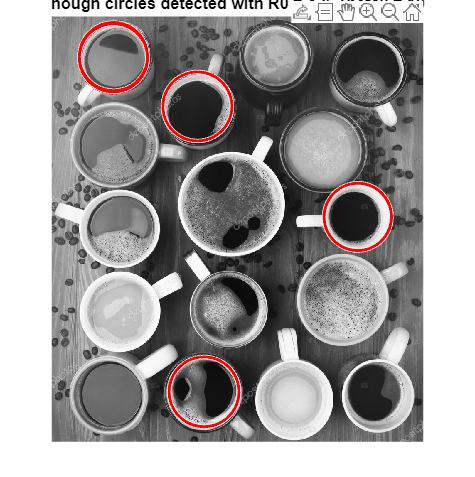


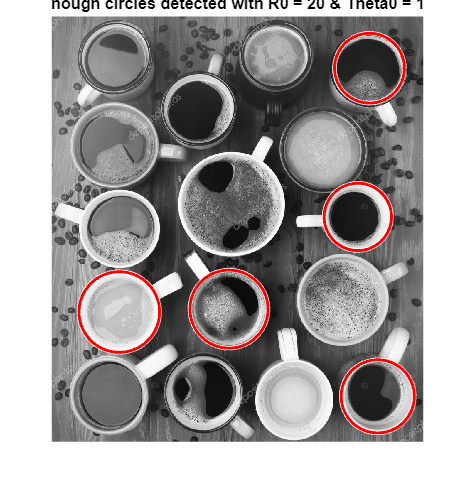
#### (f) Write you own dip\_houghpeaks3d

We implemented the function peaks3d = ``dip\_houghpeaks3d(H\_circles\_1)`` as suggested, and used it to mark the 5 most significant circles detected by the Hough algorithm:

For minimal R0, theta0 (1,1):







the effect of different R0 and Theta0 values:

- With larger R0, we check more radii, enhancing results but also adding runtime (linearly). but, too large R0 might miss some circles, or make some seem less intense. On the other hand, increasing Theta0 results in less smooth circles, missing our goal, and longer runtime.

We can see that our result made 2 circles apper at almost identicle coordinates and radii, making them overlap!