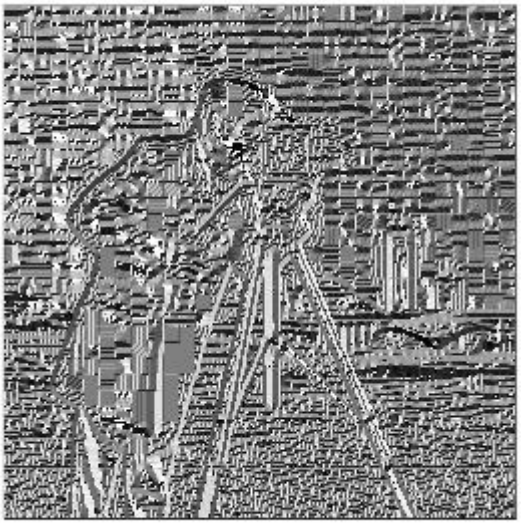
Roberts Cross Operator Results

Image 1: Cameraman

cameraman.jpg

Original Image

Gradient Magnitude



Gradient Direction

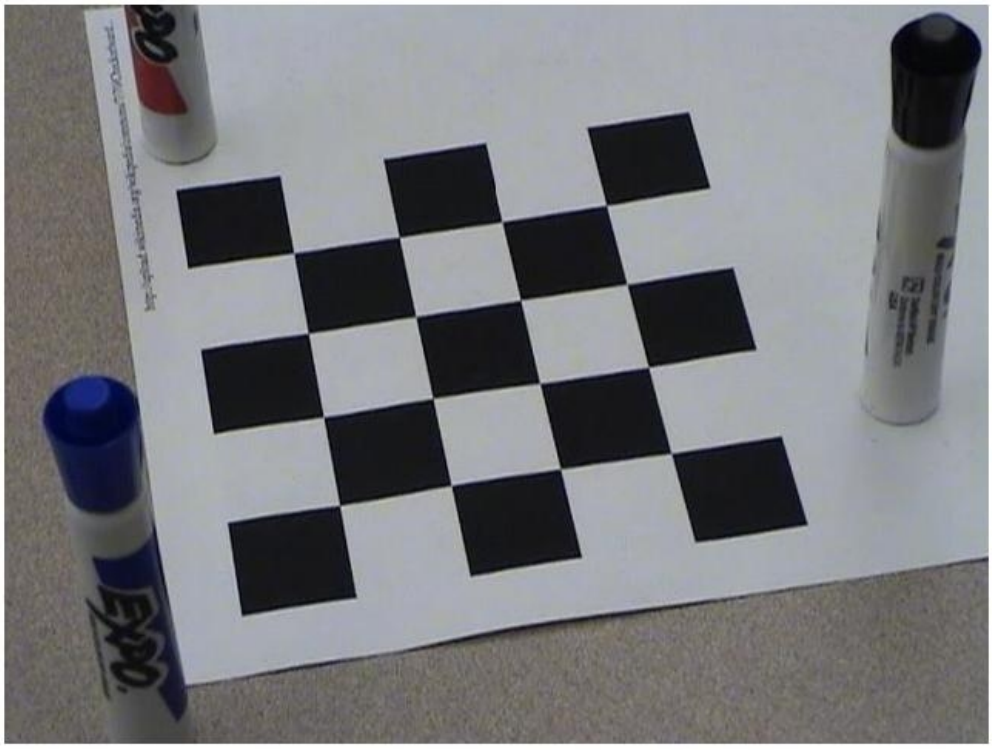
Thresholded Magnitude (100)

Thresholded Direction (100)

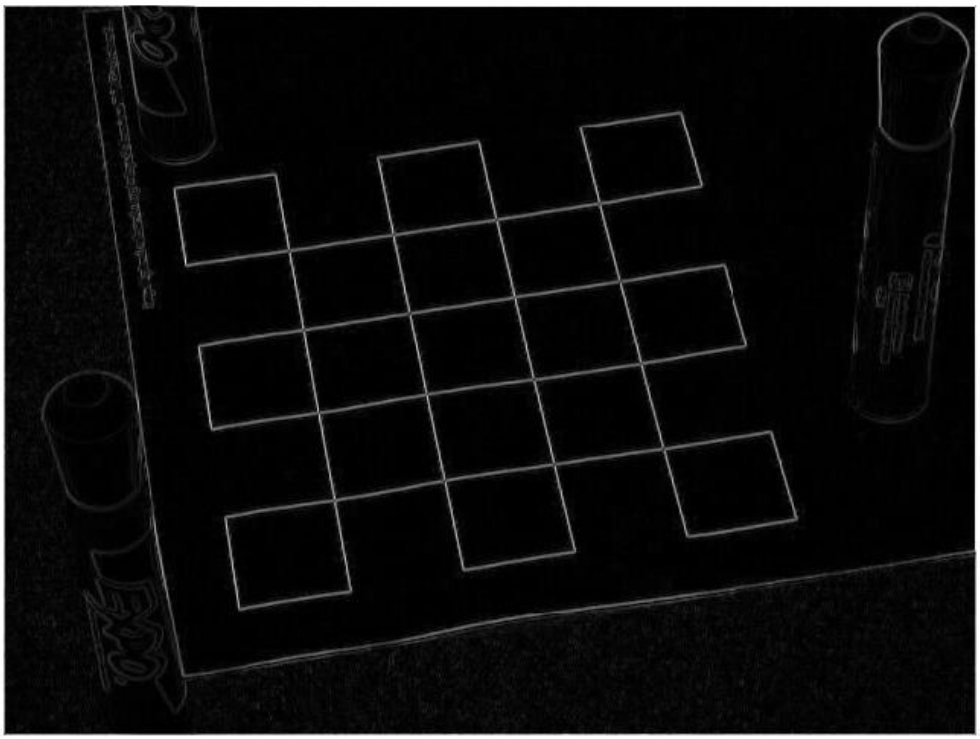
Magnitude after Thinning
(5 NSEW parses)

Image 2: Grid

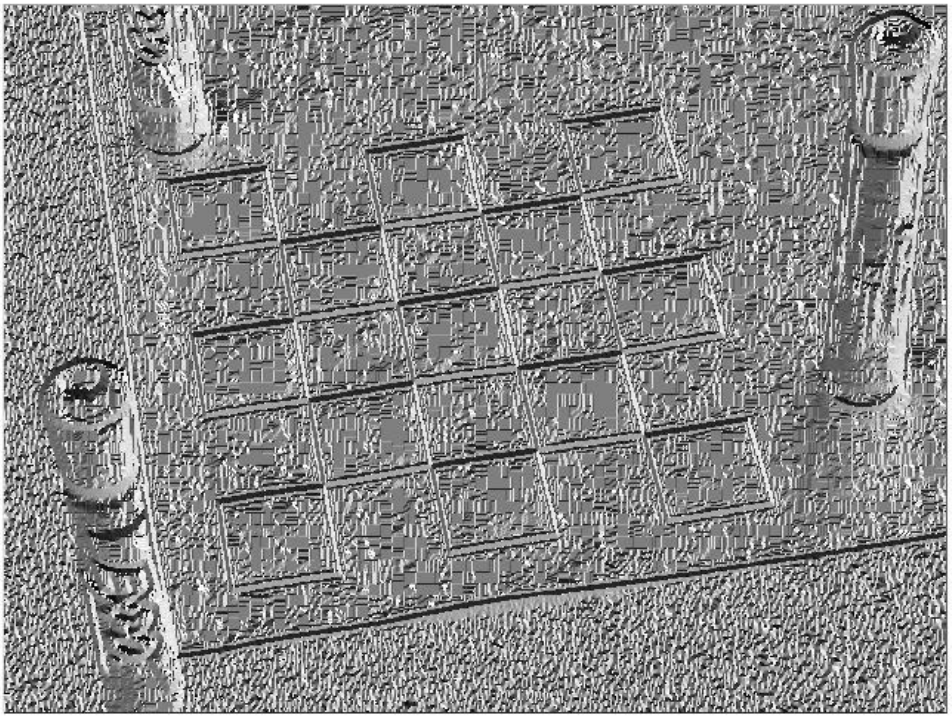
grid.jpg



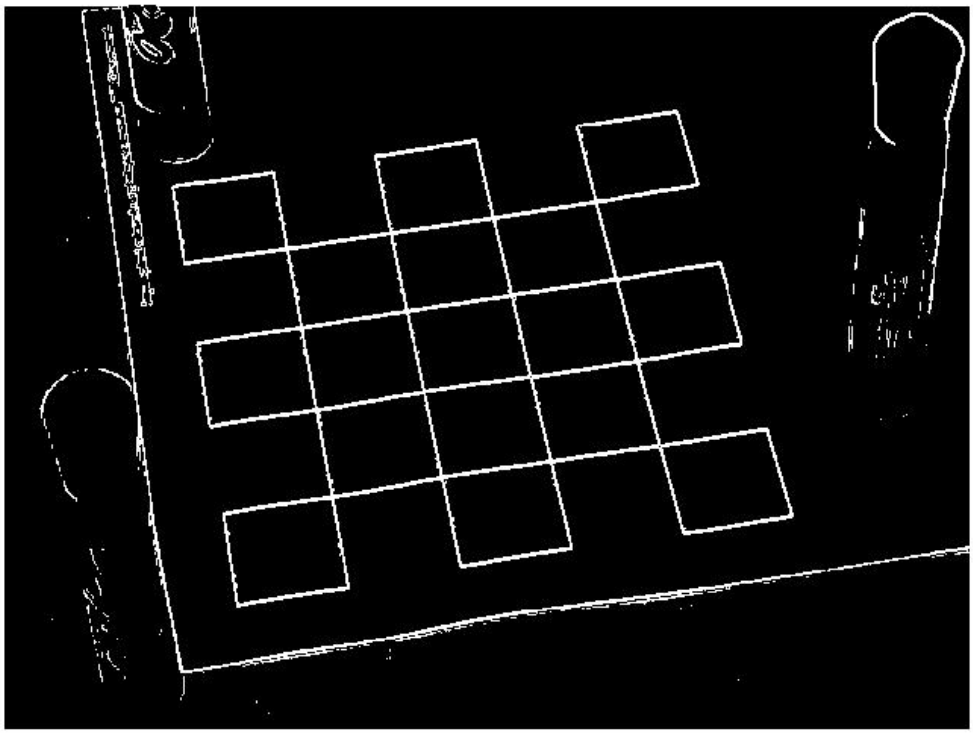
Original Image



Gradient Magnitude



Gradient Direction



Thresholded Magnitude (30)

Thresholded Direction (150)

Sobel Operator Results

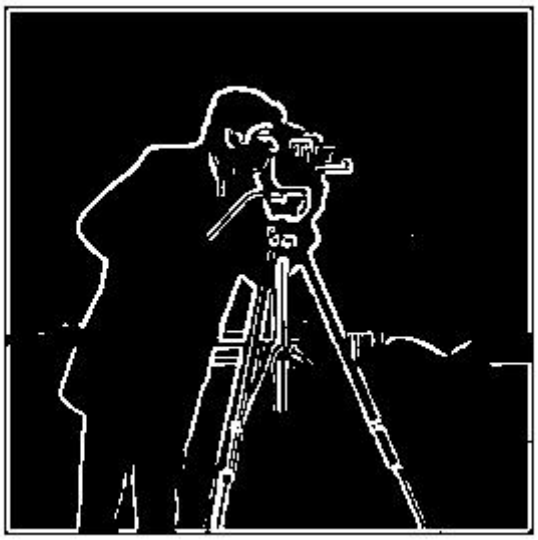
Image 1: Cameraman

cameraman.jpg

Original Image

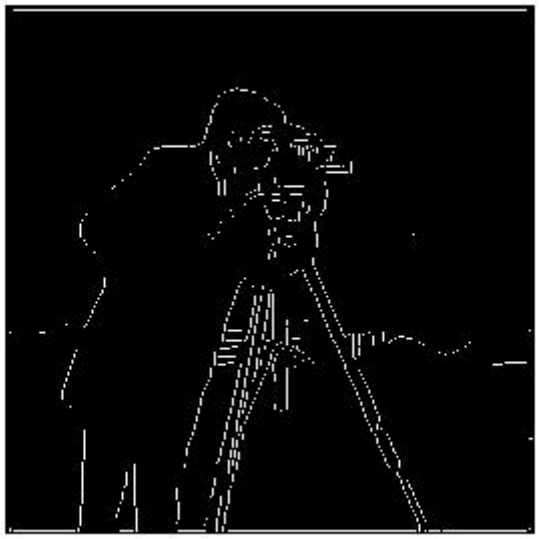
Gradient Magnitude

Gradient Direction



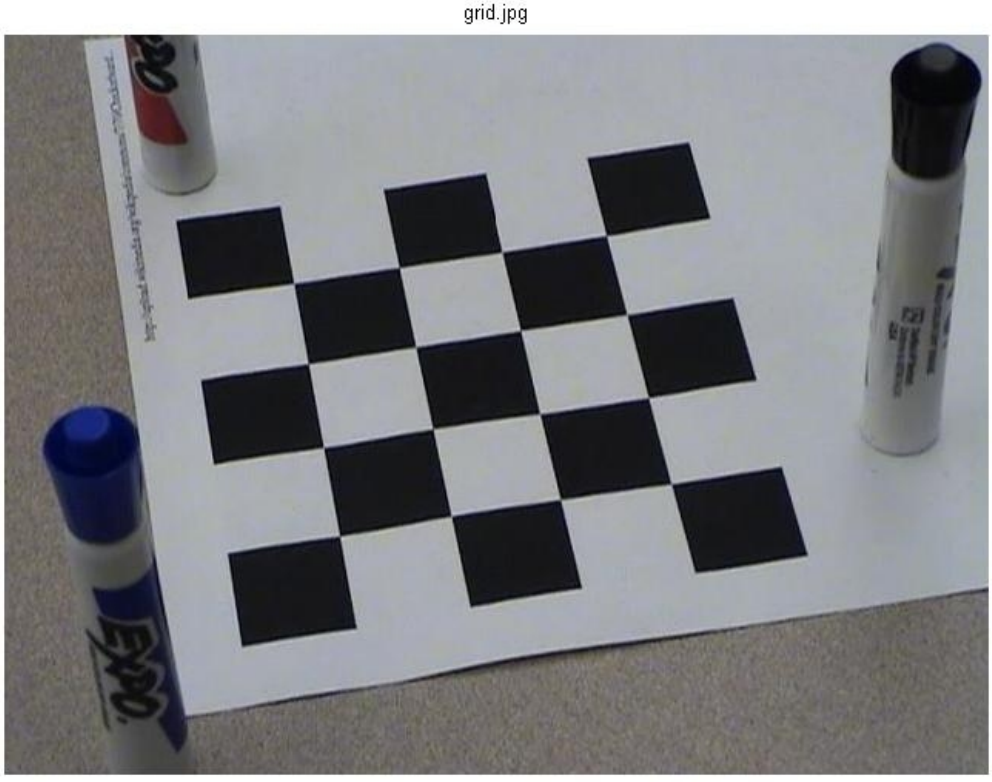
Thresholded Magnitude (100)

Thresholded Direction (100)

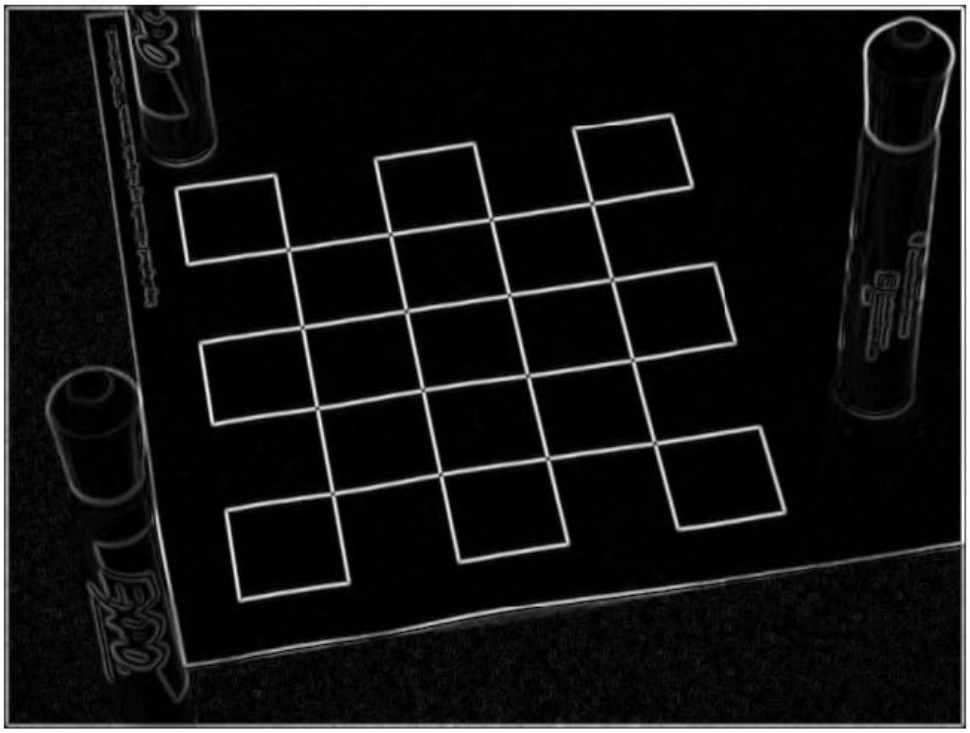


Magnitude after Thinning
(163 NSEW parses)

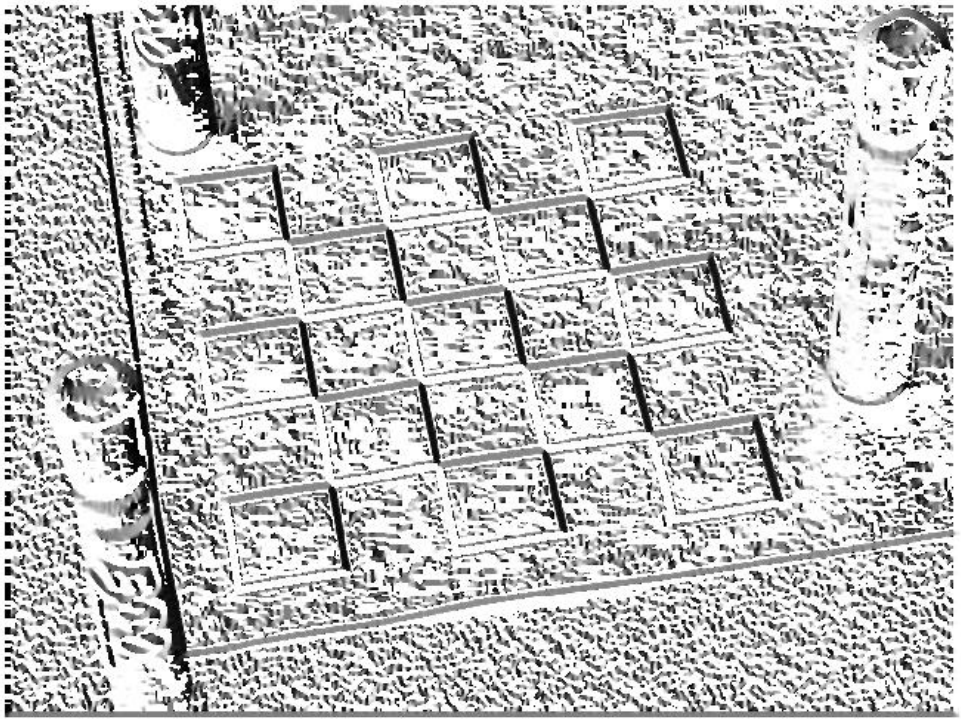
Image 2: Grid



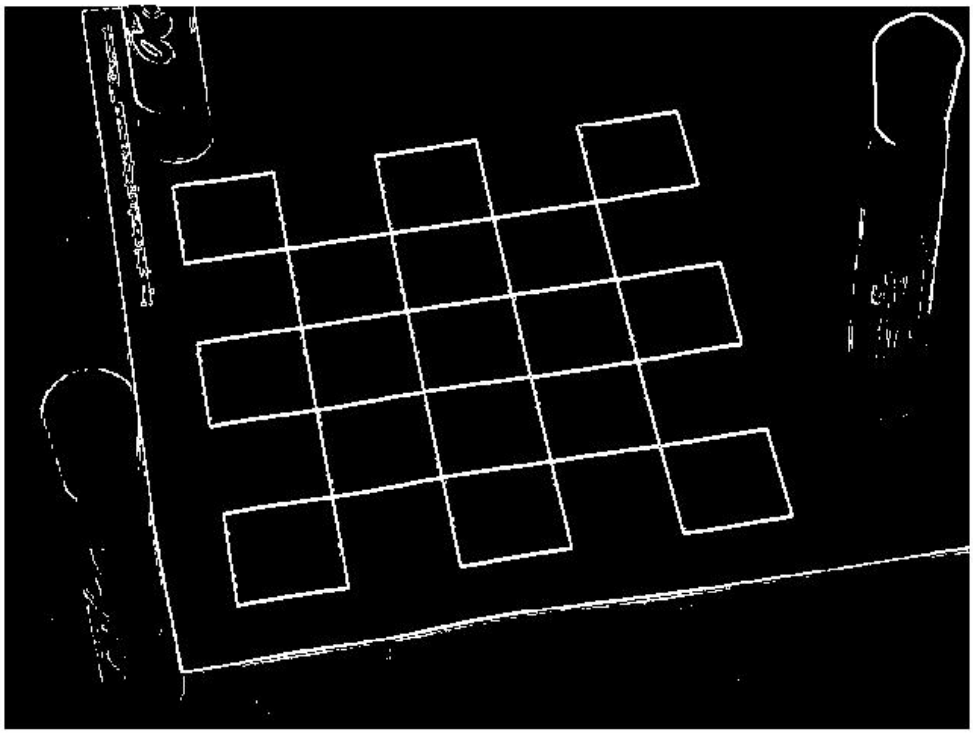
Original Image



Gradient Magnitude



Gradient Direction



Thresholded Magnitude (100)

Thresholded Direction (100)

Code

- **Roberts Cross Edge Operator**

```
%-------------------------------------------------------------------------
% Reference: http://homepages.inf.ed.ac.uk/rbf/HIPR2/roberts.htm
%-------------------------------------------------------------------------

% Implementation of Roberts Operator for cross edge detection
clear all;
close all;

% Load a gray scale image
I = imread('cameraman.jpg');
% I = rgb2gray(I);

% Converting the image to double values for better accuracy in
% mathematical processing
%I = im2double(I);
I = double (I) / 255;


% Creating kernels for detecting diagonal edges and convolving them over
% the original image
K1 = [0 1; -1 0];
K2 = [1 0; 0 -1];
imgK1 = conv2(I, K1);
imgK2 = conv2(I, K2);

% Merging the two outputs to get the gradient magnitude
mag = sqrt(imgK1.^2 + imgK2.^2);

% Calculating the gradient direction
dir = atan2(imgK2, imgK1) - ((3*pi)/4);


%Scale all values in mag between 0 and 1 to ensure that imshow can display
%them correctly
mag = scale(mag);
figure;
imshow(mag);

%Scale all values in dir between 0 and 1 to ensure that imshow can display
%them correctly
dir = scale(dir);
figure;
imshow(dir);
```

```
% Thresholding Magnitude
binMag = threshold(mag, 100);
figure;
imshow(binMag);

% Thresholding Direction
binDir = threshold(dir, 100);
figure;
imshow(binDir);


% Thinning based on 8 connectivity of the magnitude
thinnedImg = thin(binMag);
figure;
imshow(thinnedImg);
```

- **Function for Thresholding**

```
% This function creates a binary image based on thresholded value of a
% given image

function output = threshold(inputImg, value)

% Converting the image back to uint8 format for thresholding
inputImg = im2uint8(inputImg);

% Create an array that holds index of all pixels above the threshold
list = find(inputImg > value);

% Create a binary image of the same size as the original with all pixels
% turned OFF
binaryImg = zeros(size(inputImg));

% Switch ON all pixels which are greater than the threshold
binaryImg(list) = 1;

output = binaryImg;
```

- **Function for Scaling**

```
% This function scales the values of an image to between 0 & 1

function output = scale(inputImage)

minValue = min(inputImage);              % returns row as a vector
minValue = min(minValue);                % returns max value from the row
inputImage = inputImage - minValue;      % This will scale the lowest value
to 0

maxValue = max(inputImage);              % returns row as a vector
maxValue = max(maxValue);                % returns max value from the row
inputImage = inputImage / maxValue;      % This will scale the highest
value to 1

output = inputImage;
```

- **Sobel 5x5 Edge Operator**

```
%-------------------------------------------------------------------------
% Reference: http://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm
%-------------------------------------------------------------------------

% Implementation of Sobel 5x5 Operator for edge detection
clear all;
close all;

% Load a gray scale image
I = imread('cameraman.jpg');
%I = rgb2gray(I);

% We add 2 rows and 2 columns of padding to the array to allow the 5x5
% Sobel operator to take the border rows and columns into parsing as well
I = padarray(I, [2 2],'replicate');

% Converting the image to double values for better accuracy in
% mathematical processing
% I = im2double(I);
I = double (I) / 255;

% Creating kernels for detecting diagonal edges and convolving them over
% the original image
K1 = [-1 -2 0 2 1; -2 -3 0 3 2; -3 -5 0 5 3; -2 -3 0 3 2; -1 -2 0 2 1];
K2 = -K1';
```

```
imgK1 = conv2(I, K1);
imgK2 = conv2(I, K2);

% Merging the two outputs to get the gradient magnitude
mag = sqrt(imgK1.^2 + imgK2.^2);

% Calculating the gradient direction
dir = atan2(imgK2, imgK1);


%Scale all values in mag between 0 and 1 to ensure that imshow can display
%them correctly
mag = scale(mag);
figure;
imshow(mag);

%Scale all values in dir between 0 and 1 to ensure that imshow can display
%them correctly
dir = scale(dir);
figure;
imshow(dir);


% Thresholding Magnitude
binMag = threshold(mag, 100);
figure;
imshow(binMag);

% Thresholding Direction
binDir = threshold(dir, 100);
figure;
imshow(binDir);


% Thinning based on 8 connectivity of the magnitude
thinnedImg = thin(binMag);
figure;
imshow(thinnedImg);
```

- **Function for Counting No. of Neighbors of a given pixel in an image**

```
% This function accepts an image and returns no. of neighbors a given
% pixel has

function [number] = countNeighbors(img, a, b)

count = 0;
```

```
% We pad the image with 1 row and column of 0's to reduce complexity in
% counting and scale i & j for that
img = padarray(img, [1 1]);
a = a + 1;
b = b + 1;

% Sum of all 9 cells
for m=a-1:a+1
    for n=b-1:b+1
        if img(m,n) == 1
            count = count+1;
        end
    end
end

% As we cannot count the pixel itself
number = count - 1;
```

- **Function for Checking the 6 possibilities where a pixel must be retained**

```
function [flag] = checkConditions(img, p, q)

flag = 0;
conditions = [0 0 0; 0 1 0; 0 0 0];

% padding the original image with row and col containing 0's
img = padarray(img, [1 1]);
p = p + 1;
q = q + 1;

% The 6 kernels
k1 = [0 0 0; 1 1 1; 0 0 0];
k2 = [0 1 0; 0 1 0; 0 1 0];
k3 = [0 0 0; 0 1 0; 1 0 0];
k4 = [0 0 1; 0 1 0; 0 0 0];
k5 = [0 0 0; 0 1 0; 0 0 1];
k6 = [1 0 0; 0 1 0; 0 0 0];

for m=1:3
    for n=1:3
        conditions(m,n) = img(p-2+m, q-2+n);
    end
end

if isequal(conditions, k1) || isequal(conditions, k2) ||
isequal(conditions, k3) || isequal(conditions, k4) || isequal(conditions,
k5) || isequal(conditions, k6)
    flag = 1;
end
```

- **Function for Thinning**

```
function output = thin(inputImg)

% clear all;
% close all;
% inputImg = [1 1 0 0; 0 1 1 0; 0 0 1 1; 0 1 0 0];

mask = ones(size(inputImg));
[rows,cols] = size(inputImg);

complete = 0;
noOfParses = 1;
previousScan = inputImg;

while(complete == 0)
    % North Scan
    for i=1:rows
        for j=1:cols
            if inputImg(i,j) == 1
                noOfNeighbors = countNeighbors(inputImg, i, j);

                % If no. of neighbors is greater than 1
                if (noOfNeighbors > 1)
                    flagMatch = checkConditions(inputImg, i, j);

                    % If no patterns matched & this is not the first row
                    if (i == 1)
                        mask(i,j) = 0;
                    elseif (flagMatch == 0) && (inputImg(i-1,j) ~= 1)
                        mask(i,j) = 0;
                    end
                end
            end
        end
    end

    inputImg = inputImg .* mask;

    % South Scan
    for i=rows:1
        for j=cols:1
            if inputImg(i,j) == 1
                noOfNeighbors = countNeighbors(inputImg, i, j);

                % If no. of neighbors is greater than 1
                if (noOfNeighbors > 1)
                    flagMatch = checkConditions(inputImg, i, j);
```

```
                % If no patterns matched & this is not the first row
                if (i == rows)
                    mask(i,j) = 0;
                elseif (flagMatch == 0) && (inputImg(i+1,j) ~= 1)
                    mask(i,j) = 0;
                end
            end
        end
    end
end

inputImg = inputImg .* mask;

% West Scan
for j=1:cols
    for i=rows:1
        if inputImg(i,j) == 1
            noOfNeighbors = countNeighbors(inputImg, i, j);

            % If no. of neighbors is greater than 1
            if (noOfNeighbors > 1)
                flagMatch = checkConditions(inputImg, i, j);

                % If no patterns matched & this is not the first row
                if (j == 1)
                    mask(i,j) = 0;
                elseif (flagMatch == 0) && (inputImg(i,j-1) ~= 1)
                    mask(i,j) = 0;
                end
            end
        end
    end
end

inputImg = inputImg .* mask;

% East Scan
for j=cols:1
    for i=1:rows
        if inputImg(i,j) == 1
            noOfNeighbors = countNeighbors(inputImg, i, j);

            % If no. of neighbors is greater than 1
            if (noOfNeighbors > 1)
                flagMatch = checkConditions(inputImg, i, j);

                % If no patterns matched & this is not the first row
                if (j == cols)
                    mask(i,j) = 0;
                elseif (flagMatch == 0) && (inputImg(i,j+1) ~= 1)
                    mask(i,j) = 0;
```

```
                    end
                end
            end
        end
    end

    inputImg = inputImg .* mask;

    if isequal(previousScan, inputImg)
        complete = 1;
    else
        previousScan = inputImg;
        noOfParses = noOfParses + 1;
    end
end

output = inputImg;

noOfParses
```

Analysis

Padding

While implementing the Sobel operator it was necessary to pad the image by 2 rows and columns on each side to ensure that the 5x5 operator would not overflow outside when the border elements of the image were being analyzed.

I ended up replicating the border rows and columns in the padded elements by using Matlabs inbuilt function
`I = padarray(I, [2 2],'replicate');`

Here is what a sample array would look like after padding by replication for a single row and column:

1	2
3	4

Original Array

1	1	2	2
1	1	2	2
3	3	4	4
3	3	4	4

Padded Array

Since the images are really big, a ghosting of 2 rows is not really evident but this is still an important step.

Scaling

While implementing the Sobel 5x5 operator, I had initially not scaled the values between 0 and 1 as required for the imshow() function in Matlab to work correctly. As a result, I was seeing a lot of noise in the image. Here are the results before and after scaling the values.

Before Scaling (Lot of noise)

After Scaling (Less noise)

Thinning

My implementation of thinning was based on the 2nd approach discussed in class i.e. on the algorithm to retain points. This technique basically requires us to parse the cells in a row-major fashion where each iteration requires a scan from the North, South, East and West directions of the image. Parsing stops when the resultant output image of 2 successive iterations is the same. As a result, this is a very performance intensive implementation and demands some tweaking.

For each scan we pick a pixel and retain it:

a. If it is an isolated pixel or the end of a line. In other words we have to retain the point if it has 0 or 1 neighbors
b. If it matches one of the 6 conditions as illustrated in the class notes
c. If it has a N, S, E, W neighbor respective to the scan taking place

For each, scan I created a mask image of the same size with all the pixels turned on. If the three steps of my scan indicated that the pixel needs to be switched off, I made a corresponding note in the mask. At the end of the operation I did an AND operation between the input image and the mask to get an output. To illustrate how this works, please see the following example:

1	1	0	0
0	1	1	0
0	0	1	1
0	1	0	0

1. Input binary image

1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1

2. Initial mask

0	0	1	1
1	1	0	1
1	1	1	0
1	1	1	1

3. Mask after a N scan

0	0	0	0
0	1	0	0
0	0	1	0
0	1	0	0

1 AND 3 gives the thinned image for the S scan

Roberts vs. Sobel Operator

The Roberts Cross operator is more sensitive to noise than the Sobel operator which is obvious when we apply both operators to the same image. This is because the Roberts operators take a less number of localized pixels into consideration (2x2 kernel) vs. the Sobel operator (5x5 kernel). However, the greater the kernel size, the slower the algorithm will be.

While thinning, I could see that there is a big difference between the times consumed on the binary outputs of the magnitude threshold of the two operators. While Roberts took only 5 parses, Sobel took a lot of time with 163 parses in the cameraman example.

Overall, Sobel's output is usually better considering noise rejection and edge detection but it is a lot slower than Roberts operator. Based on the application requirements, both the operators can be useful.

References:

http://homepages.inf.ed.ac.uk/rbf/HIPR2/roberts.htm
http://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm