Visual Computing Lab 3

CHAKKA Karthik Subramanyam

CHATTERJEE Jit

TP3 - Edge Detection

Exercise 1: Gradient

1.a) Write a program that computes the images Ix and Iy of the gradients in x and y of a PGM image (filtered) using the Scharr operators.

1.b) Display the image of the gradient magnitudes.

1.a) Introduction:

```
Edge detection can be done using image gradient technique. Gradient is a
 vector which has certain magnitude and direction,
```

$$g(x,y) \cong (\Delta x^2 + \Delta y^2)^{1/2}$$

$$\Theta(x,y) \cong \tan^{-1}(\Delta y/\Delta x)$$

```
where, g(x,y) is the gradient magnitude and Θ(x,y) is the gradient
 direction.
```

$$\Delta x = f(x+n,y) - f(x-n,y)$$

$$\Delta y = f(x,y+n) - f(x,y-n)$$

```
and n is a small integer, usually equal to 1.
```

```
The magnitude of gradient provides information about the strength of
 the edge. The direction of gradient is always perpendicular to the
 direction of the edge (the edge direction is rotated with respect to
 the gradient direction by -90 degrees).
```

Goal:

```
To apply edge detection using image gradient technique with Scharr
 operators.
```

Procedure:

```
The Scharr Operators is used as a filter to generate the Ix and Iy
 components of the gradients in x and y of a PGM image. It is obtained
 by optimizing the gradient estimation in the Fourier domain:
```

$$h1 = 1/16\begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix} \qquad h2 = 1/16\begin{bmatrix} -3 & -10 & -3 \\ 0 & 0 & 0 \\ 3 & 10 & 3 \end{bmatrix}$$

Steps:

1. Define the Scharr Operators to generate the filters scharrX (h1) and scharrY (h2):

```
mat scharrX[9] = {-3, 0, 3, -10, 0, 10, -3, 0, 3};
mat scharrY[9] = {-3, -10, -3, 0, 0, 0, 3, 10, 3};
*divFactor = 16;
```

2. Read pgm image

```
We need to read the pgm image and store the pixel values into a 1D
 memory buffer.
```

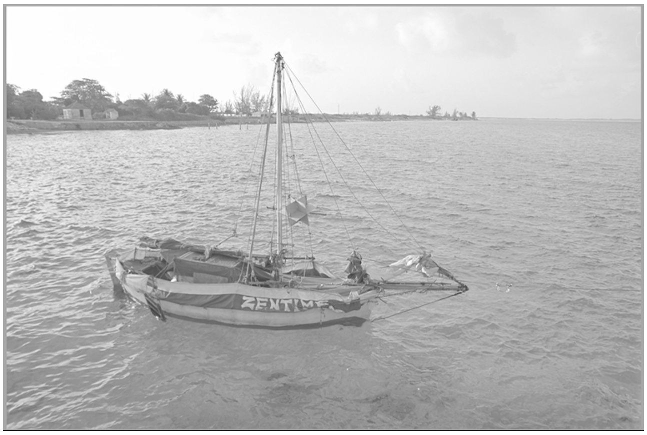
3. Convolution with h1 and h2 seperately

```
Now, we have to convolute the image pixels with h1 and h2 filter
 matrices seperately to get first derivative in horizontal direction
 and vertical direction.
```

Code Snippet:

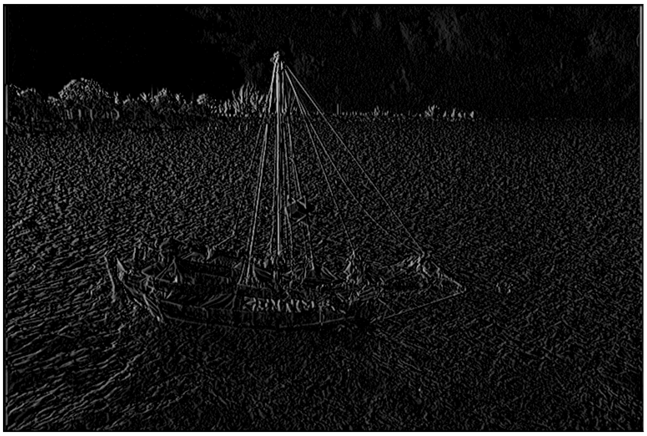
```
void convol(pix *dstBuff, pix *srcBuff, size_t row, size_t col, pix *h,
 pixprod divFactor){
    pix *revH = allocFilterMem(sizeof (pix));
    revFilter(revH, h);
    for (size_t i = 0; i < row; ++i) {
        for (size_t j = 0; j < col; ++j) {
            *(dstBuff + ( (i * col) + j) ) = convXY(srcBuff, i, j, row,
 col, revH, divFactor);
        }
    }
    free(revH);
    return;
}
```

Input Image:

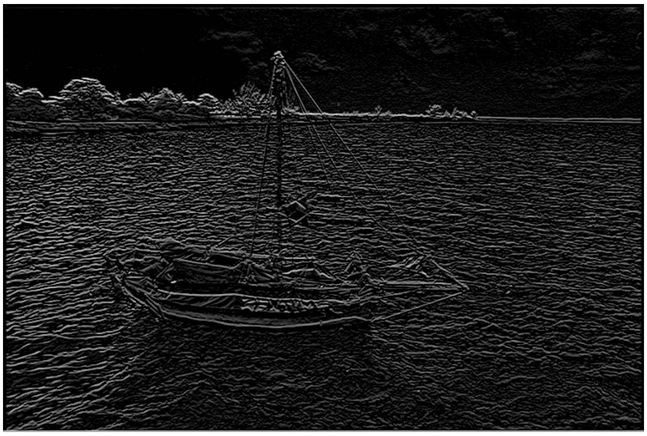


Input image of a boat (Fig 1.1)

Output Image:

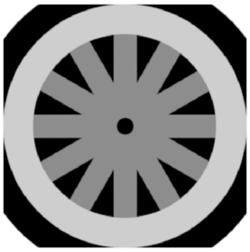


```
   Boat image with first derivative in horizontal direction Ix after
applying Filter using scharr operators
                               (Fig 1.2)
```



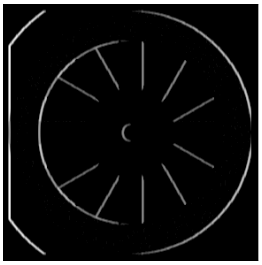
```
   Boat image with first derivative in vertical direction Iy after
applying Filter using scharr operators
                               (Fig 1.3)
```

Input Image:

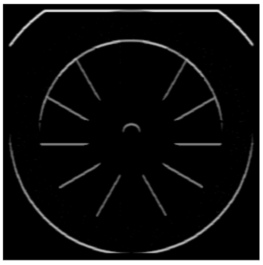


```
Input image of a wheel (Fig 1.4)
```

Output Images:



```
   Wheel image with first derivative in horizontal direction Ix after
applying Filter using scharr operators
                              (Fig 1.5)
```



```
   Wheel image with first derivative in vertical direction Iy after
applying Filter using scharr operators
                              (Fig 1.6)
```

1.b) After computing the vertical and horizontal components of the image we have to generate the gradient magnitudes.

Steps:

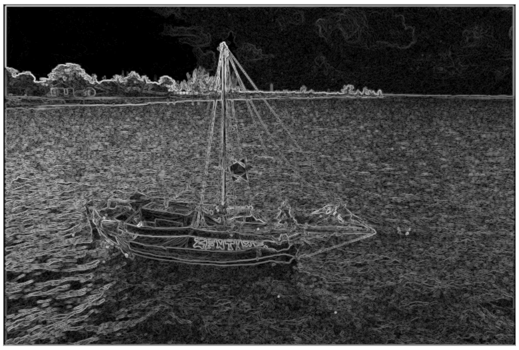
1. Multiply the vertical component with itself to get the square. Similarly for the horizontal component.
2. Summation of the square of both horizontal and vertical component.
3. Square root of the values which we get at step 2.

Code Snippet:

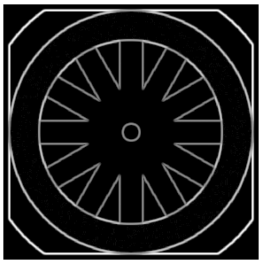
```
void getMagnitudeMat(mat *xComp, mat *yComp, size_t row, size_t col, mat
 *mag){
    mat *x2Comp = allocMatMem(row, col);
    prodMatrix(xComp, xComp, row, col, x2Comp);
    mat *y2Comp = allocMatMem(row, col);
    prodMatrix(yComp, yComp, row, col, y2Comp);
    mat *sum2 = allocMatMem(row, col);
    sumMatrix(x2Comp, y2Comp, row, col, sum2);
    free(x2Comp);
    free(y2Comp);

    matrixSqrt(sum2, row, col, mag);
    free(sum2);
}
```

Output Images:



```
Image of the gradient magnitudes (boat) (Fig 1.7)
```



```
Image of the gradient magnitudes (wheel)
(Fig 1.8)
```

Exercise 2 : Edge Detection

2.a) Display pixels with gradient magnitudes above a threshold defined by the user. Comment on the results.

2.b) Canny's approach:

i. Non maximum suppression: extract local gradient extrema in the gradient direction.
ii. Hysteresis thresholding: edge pixels correspond to pixels with a gradient magnitude above a low threshold that are connected to at least one pixel with a gradient magnitude above a high threshold.

Introduction:

Canny's approach is a multi-stage edge detector. Here, a Gaussian filter is used to remove noise from the initial image. Smoothened image is then filtered with a Scharr kernel in both horizontal and vertical direction. Then, potential edges are thinned down to 1-pixel curves by removing non-maximum pixels of the gradient magnitude. Finally, edge pixels are kept or removed using hysteresis thresholding on the gradient magnitude.

Goal:

i. To implement user defined threshold for gradient magnitudes.
ii. To implement Edge detection using Canny’s approach.

Procedure:

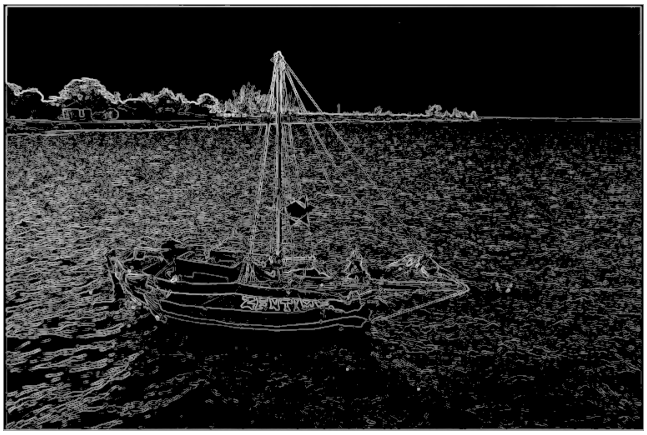
2.a) Steps to implement the threshold for gradient magnitudes:

1. Iterate over memory buffer containing gradient magnitudes.
2. If the value is less than the threshold, replace the value with 0.
3. Else, keep the original value.

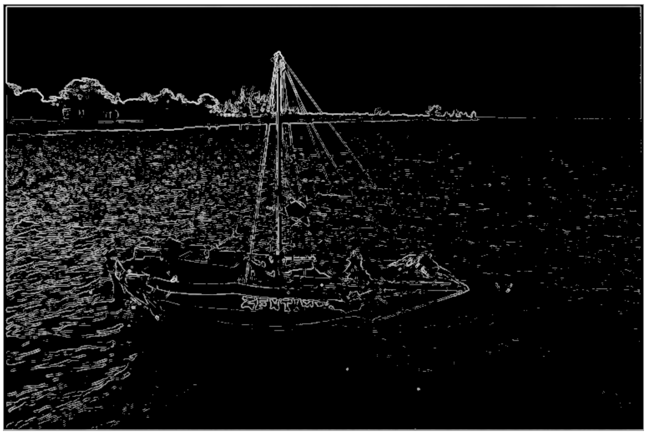
Code Snippet:

```
void matrixThresholding(mat *x, size_t r, size_t c, size_t th, mat *y){
    assert(th <= (unsigned)MAX_PIX_VAL);
    for (size_t i = 0; i < r; ++i) {
        for (size_t j = 0; j < c; ++j) {
            if( x[(i * c) + j] < (signed)th){
                y[(i * c) + j] = 0;
            } else {
                y[(i * c) + j] = x[(i * c) + j];
            }
        }
    }
}
```

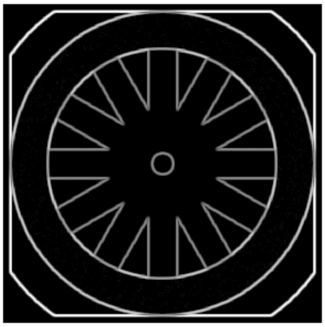
Output Images:



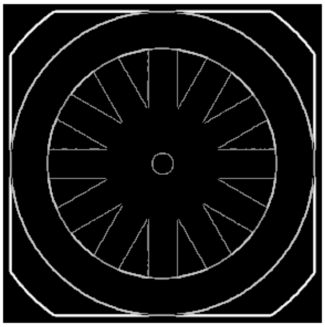
```
Image of the gradient magnitudes (boat) where threshold = 20
                          (Fig 2.1)
```



```
Image of the gradient magnitudes (boat) where threshold = 50
                          (Fig 2.2)
```



```
Image of the gradient magnitudes (wheel) where threshold = 1
                         (Fig 2.3)
```



```
Image of the gradient magnitudes (wheel) where threshold = 50
                          (Fig 2.4)
```

Results:

```
From the above figures (Fig 2.1, 2.2, 2.3 and 2.4), we can infer that
 increasing the threshold removes the thin edges. Comparing, Fig 2.1
 with Fig 2.2 clearly depicts that when the threshold is increased from
 20 to 50, some of the thin edges in the image becomes black.
```

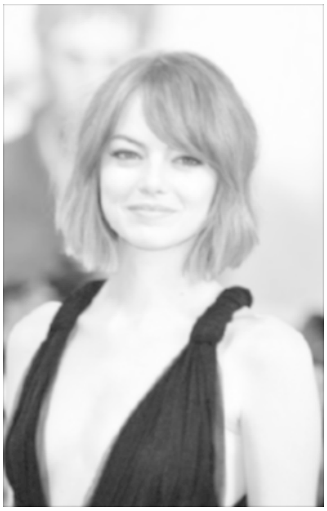
2.b) Steps to implement Edge detection using Canny's approach:

1) Noise Reduction of the image by using **Gaussian Filter**. Generate the gaussian filter with binomial coefficients and perform convolution with the image matrix.

Input Image:

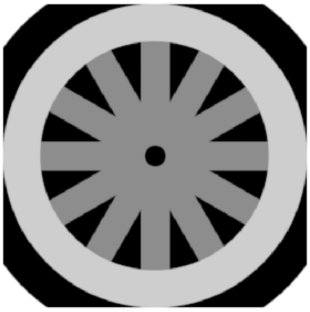
```
Input image (Fig 2.5)
```

Output Image:



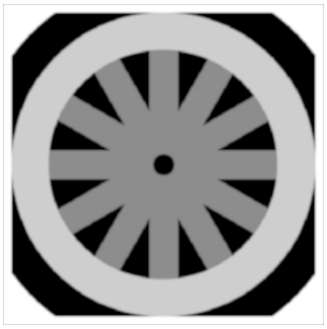
```
Applying Gaussian Filter on the image Fig 2.5 to remove noise
(Fig 2.6)
```

Input Image:



```
Input image of a wheel (Fig 2.7)
```

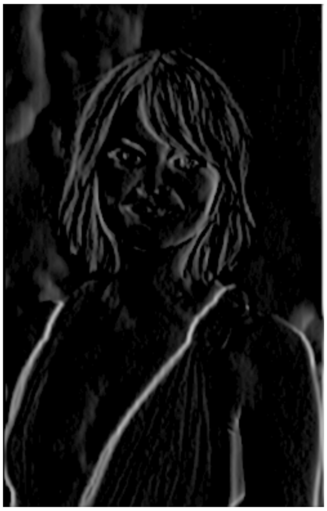
Output Image:



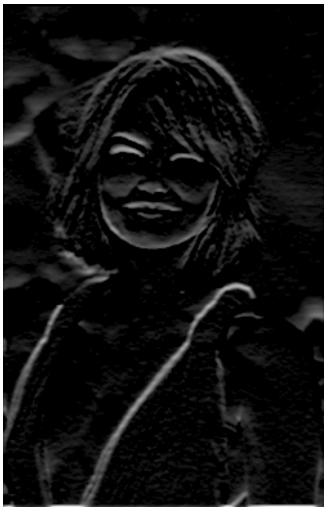
```
Applying Gaussian Filter on the image Fig 2.7 to remove noise
(Fig 2.8)
```

2) Apply filter (using **Scharr Operator**) to generate the vertical and horizontal components of the image.

Output Images:



```
   Applying Filter (using Scharr Operators) on the image Fig 2.6 to
generate first derivative in horizontal direction Ix
                               (Fig 2.9)
```



```
   Applying Filter (using Scharr Operators) on the image Fig 2.6 to
generate first derivative in vertical direction Iy (Fig 2.10)
```

3) Generate the **intensity gradient** of the image (magnitude and direction): For direction we need to calculate the angle.

Code Snippet:

```
void matrixArcTan(mat *y, mat *x, size_t r, size_t c, mat *theta){
    for (size_t i = 0; i < r; ++i) {
        for (size_t j = 0; j < c; ++j) {
            double atanVal = atan2( y[(i * c) + j], x[(i * c) + j] );
            atanVal *= 180;
            atanVal /= PI;
            if(atanVal < 0){
                atanVal += 180;
            }
            theta[((i * c) + j)] = (mat) atanVal;
        }
    }
}
```

Output Image:

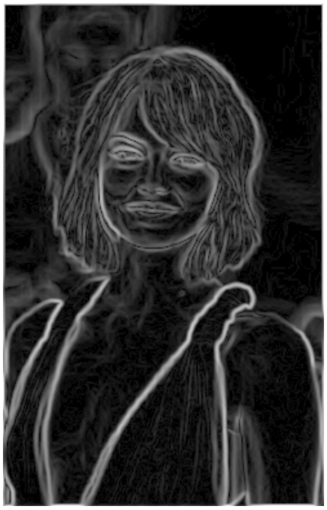


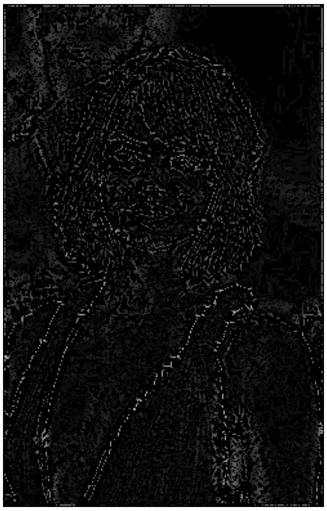
Image of the gradient magnitudes for Fig 2.9 and Fig 2.10 (Fig 2.11)

4) Apply **non-maximum suppression** to get rid of spurious response to edge detection by removing any unwanted pixels which may not constitute the edge. Each pixel is checked if it is a local maximum in its neighborhood in the direction of gradient. For the pixels at the edge in the image matrix, set the values to 255.

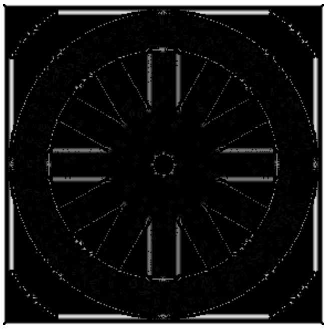
Code Snippet:

```
void nonMaxSuppression(mat *imgMat, mat *imgDirec, mat *nmsMat, size_t
 row, size_t col){
    for (size_t i = 0; i < row; ++i) {
        for (size_t j = 0; j < col; ++j) {
            mat q, r;
            mat curImgMat = imgMat[(i * col) + j];
            if( ( ((signed)i - 1) < 0 ) || ( ((signed)j - 1) < 0 ) ||
 ( ((signed)i + 1) > (signed)(row - 1) ) || ( ((signed)j + 1) > (signed)
(col - 1) ) ){
                q = 255;
                r = 255;
            } else {
                mat curAng = imgDirec[(i * col) + j];
                if((curAng >= 0 && curAng < 22.5) || (curAng >= 157.5 &&
 curAng <= 180)){
                    q = imgMat[(i * col) + (j + 1)];
                    r = imgMat[(i * col) + (j - 1)];
                } else if(curAng >= 22.5 && curAng < 67.5) {
                    q = imgMat[((i + 1) * col) + (j - 1)];
                    r = imgMat[((i - 1) * col) + (j + 1)];
                } else if(curAng >= 67.5 && curAng < 112.5) {
                    q = imgMat[((i + 1) * col) + j];
                    r = imgMat[((i - 1) * col) + j];
                } else if(curAng >= 112.5 && curAng < 157.5) {
                    q = imgMat[((i - 1) * col) + (j - 1)];
                    r = imgMat[((i + 1) * col) + (j + 1)];
                } else {
                    assert(0);
                }
            }
            if((curImgMat >= q) && (curImgMat >= r)){
                nmsMat[(i * col) + j] = curImgMat;
            } else {
                nmsMat[(i * col) + j] = 0;
            }
        }
    }
}
```

Output Images:



```
Applying non-maximum suppression to Fig 2.11
(Fig 2.12)
```



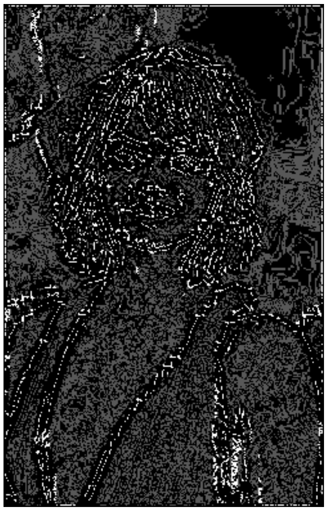
```
Applying non-maximum suppression to Fig 1.8
(Fig 2.13)
```

5) **Track edge by hysteresis**: Finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges. We are considering pixel value above 25 as weak pixels. We will set all values below 25 as 0. For tracking the edges, further for each weak pixel, we will check each of the surrounding 8 pixels and if any one of the pixels are strong pixel then the weak pixel will be treated as a strong pixel, else, the weak pixel will be set to 0.

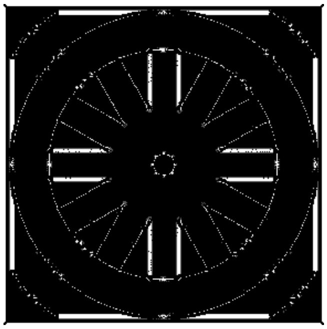
Code Snippet: For Hyteresis Thresholding

```
#define STRONG_PIX_VAL 255
#define WEAK_PIX_VAL 25
void hysteresisThresholding(mat *nmsImg, mat *hysThImg, size_t row,
 size_t col, double lowThRatio, double highThRatio){
    double highTh = getMatrixMax(nmsImg, row, col) * highThRatio;
    double lowTh = highTh * lowThRatio;
    assert(highTh > lowTh);
    for (size_t i = 0; i < row; ++i) {
        for (size_t j = 0; j < col; ++j) {
            mat curVal = nmsImg[(i * col) + j];
            if(curVal > highTh){
                hysThImg[(i * col) + j] = STRONG_PIX_VAL;
            } else if(curVal < lowTh){
                hysThImg[(i * col) + j] = 0;
            } else {
                hysThImg[(i * col) + j] = WEAK_PIX_VAL;
            }
        }
    }
}
```

Output Images:



```
Applying Hysteresis Thresholding to Fig 2.12
(Fig 2.14)
```

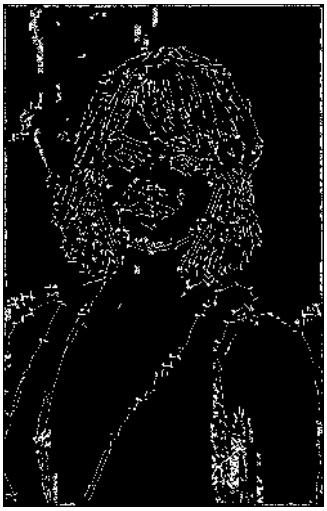


```
Applying Hysteresis Thresholding to Fig 2.13
(Fig 2.15)
```

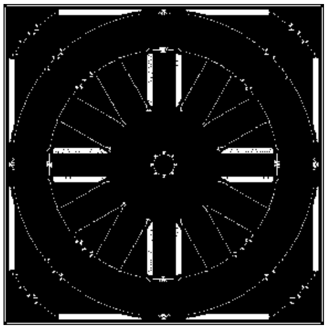
Code Snippet: For tracking edges by hyteresis

```
void trackEdges(mat *hysThImg, mat *resImg, size_t row, size_t col){
    for (size_t i = 0; i < row; ++i) {
        for (size_t j = 0; j < col; ++j) {
            if( ( ((signed)i - 1) < 0 ) || ( ((signed)j - 1) < 0 ) ||
 ( ((signed)i + 1) > (signed)(row - 1) ) || ( ((signed)j + 1) > (signed)
(col - 1) ) ){
                resImg[(i * col) + j] = 0;
            } else {
                if(hysThImg[(i * col) + j] == WEAK_PIX_VAL){
                    if( (hysThImg[((i + 1) * col) + (j - 1)] ==
 STRONG_PIX_VAL) || (hysThImg[( (i + 1) * col) + j] == STRONG_PIX_VAL)
 || (hysThImg[((i + 1) * col) + (j + 1)] == STRONG_PIX_VAL) ||
                            (hysThImg[(i * col) + (j - 1)] ==
 STRONG_PIX_VAL) || (hysThImg[(i * col) + (j + 1)] == STRONG_PIX_VAL) ||
                            (hysThImg[((i - 1) * col) + (j - 1)] ==
 STRONG_PIX_VAL) || (hysThImg[((i - 1) * col) + j] == STRONG_PIX_VAL) ||
 (hysThImg[((i - 1) * col) + (j + 1)] == STRONG_PIX_VAL) ){
                        resImg[(i * col) + j] = STRONG_PIX_VAL;
                    } else {
                        resImg[(i * col) + j] = 0;
                    }
                } else if(hysThImg[(i * col) + j] == STRONG_PIX_VAL){
                    resImg[(i * col) + j] = STRONG_PIX_VAL;
                } else if(hysThImg[(i * col) + j] == 0){
                    resImg[(i * col) + j] = 0;
                }
            }
        }
    }
}
```

Output Images:



Tracking edges by Hysteresis to Fig 2.14
(Fig 2.16)



Tracking edges by Hysteresis to Fig 2.15
(Fig 2.17)

Results:

```
Finally, we get Fig 2.16 and Fig 2.17 edge detection using Canny's
 approach on Fig 2.5 and Fig 2.7 respectively.
```

Conclusion:

```
    Hence, we can conclude with the help of edge detection of an image
 we can discover the information about the shapes and the reflectance or
 transmittance in an image.
```