Homework – 6

Name: Sridhar Mocherla

Part A

1. Computing gradient using finite differences

We compute the gradient in the X-direction and the Y-direction using the masks and applying convolution to the original image.

$$\begin{bmatrix} 0.5 & 0 & -0.5 \end{bmatrix} \begin{bmatrix} 0.5 \\ 0 \\ -0.5 \end{bmatrix}$$

Figure 1: Masks for computing X and Y gradients

Figure 2: Original image

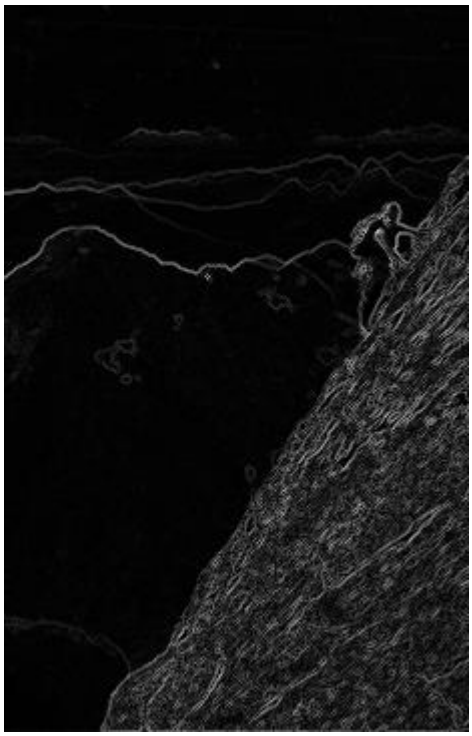

Figure 3: Gradient magnitude

MATLAB code snippet:

```
hx = [0.5 0 -0.5]; %X mask
hy = [0.5;0;-0.5]; %Y mask
```

```
Fx = conv2(double(image),hx,'same'); %Find gradient in X direction
Fy = conv2(double(image),hy,'same'); %Find gradient in Y direction

grad_mag = zeros(size(image,1),size(image,2));
grad_dir = zeros(size(image,1),size(image,2));
for i=1:size(image,1)
    for j=1:size(image,2)
       grad_mag(i,j) = sqrt(Fx(i,j).^2+Fy(i,j).^2); %Gradient magnitude at
each pixel
       grad_dir(i,j) = atan(Fy(i,j)/Fx(i,j)); %Gradient direction at each
pixel
    end
end
```

2. Detecting edges from gradient

By trial and error, we tweak with the various thresholds for the gradient magnitude and arrive on a value which gives optimum edge points. Threshold = 60.

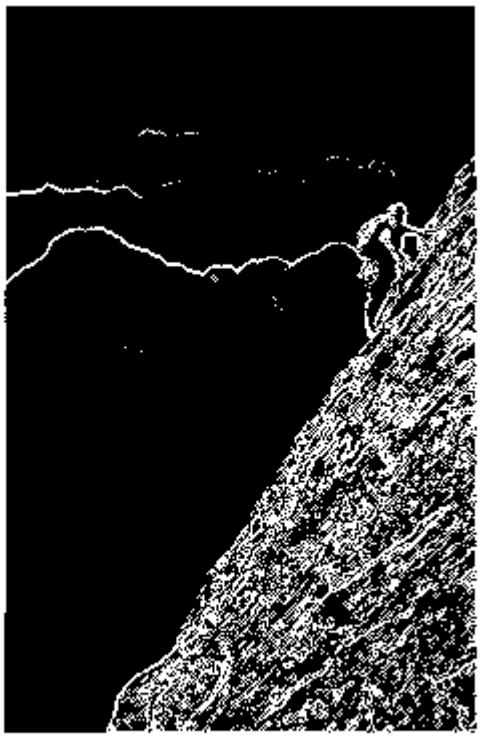

Figure 4: Edge detection from gradient

3. Smoothing using Gaussian filter

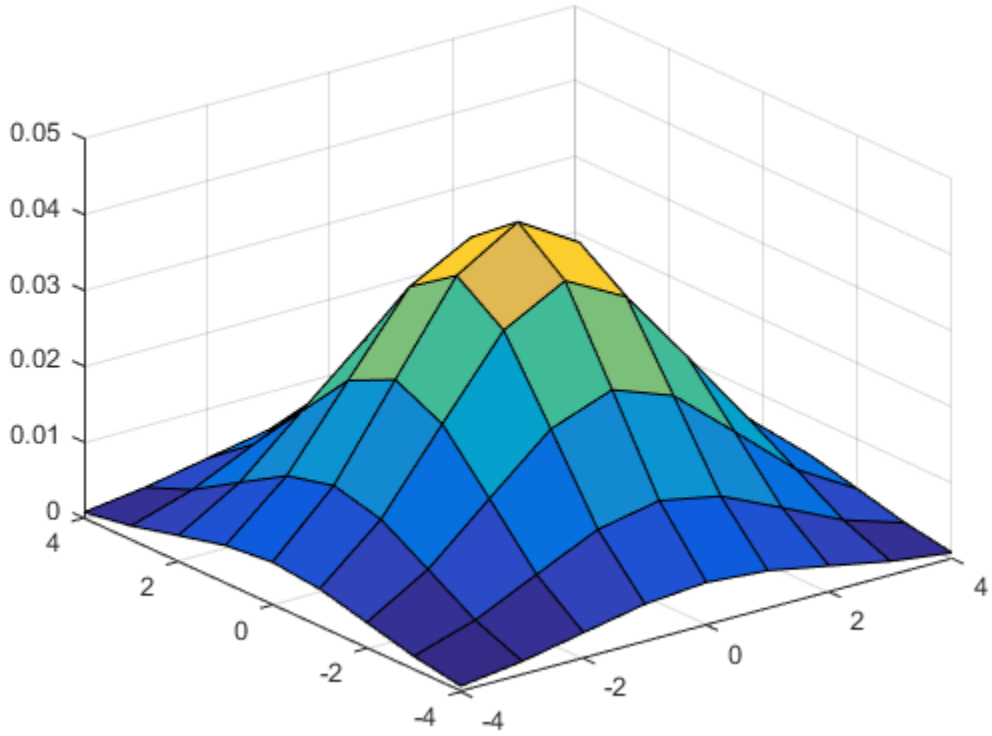

Figure 5: Surface plot of Gaussian filter with sigma 2.

We use the Gaussian function on values generated from meshgrid() to obtain the values as shown in code below:

```
n=9; %range -4 to 4
coords = -floor(n/2):floor(n/2);
[x,y] = meshgrid(coords,coords);
sig =2;
h = exp(-(x.^2+y.^2)/(2*sig*sig)); %use gaussian function
h = h/sum(h(:)); %filter should sum to 1
figure('Name','Surface plot of gaussian filter');
surf(x,y,h);
```

On applying this smoothing filter, we get a blurred image as show in Figure 6.

Figure 6: Blurring image using Gaussian filter

The code snippet for this is below:

```
result3 = conv2(double(image),h);
figure('Name','Gaussian filter with sigma 2');
imshow(uint8(result3));
```

4. Edge detection after blurring

Figure 7: Edges detected on blurred image

We use the blurred image in Figure 6 and obtain the gradient magnitude by applying the finite differences method. Then by tweaking the gradient magnitude threshold, we create a black and white image set to white at pixels where gradient magnitude is above the threshold. We determine threshold here as 8.

The code for obtaining these edges is below:

```
hx_4 = [0.5 0 -0.5];
hy_4 = [0.5;0;-0.5];
Fx_4 = conv2(double(result3),hx_4,'same');
Fy_4 = conv2(double(result3),hy_4,'same');

new_image_4 = zeros(size(result3,1),size(result3,2));
threshold = 8;
for i=1:size(result3,1)
    for j=1:size(result3,2)
        gradient = sqrt(Fx_4(i,j).^2+Fy_4(i,j).^2);
```

```
        if gradient > threshold
           new_image_4(i,j)=255;
        end
    end
end

figure('Name','Edge detection after blurring using gaussian filter');
imshow(new_image_4);
```

5. Combining blurring and edge detection

We combine blurring and edge detection into a filter. We apply a convolution (smoothing) on another convolution (gradient by finite differences) and then tweak the threshold to get the edges. Here the sigma is set to 4 pixels. The result is displayed in Figure 8.

Figure 8: Edges obtained on combining filters.

The code to perform this combination of filters is:

```
n=9;
coords = -floor(n/2):floor(n/2);
[x,y] = meshgrid(coords,coords);
sig =4;
h = exp(-(x.^2+y.^2)/(2*sig*sig));
h = h/sum(h(:));
image_5_x = conv2(conv2(double(image),hx,'same'),h,'same');%applying
smoothing and finite difference filter in X direction
```

```
image_5_y = conv2(conv2(double(image),hy,'same'),h,'same');%applying smooting
and finite difference filter in Y direction

new_result_5 = zeros(size(image,1),size(image,2));
threshold = 8; %threshold
for i=1:size(image,1)
    for j=1:size(image,2)
        gradient = sqrt(image_5_x(i,j).^2+image_5_y(i,j).^2);
        if gradient > threshold
           new_result_5(i,j)=255;
        end
    end
end
figure('Name','Problem 5');
imshow(new_result_5);
```

6. Decomposition of smoothing filter

This can be implemented by decomposing the Gaussian filter in to two filters by singular value decomposition (svd). SVD decomposes a **mxn** matrix into – an **mxm** matrix, an **mxn** matrix and a **nxn** matrix. We obtain the two decomposed filters as below and use a variation of conv2 function to obtain the result. The code snippet is below:

```
[A,B,C] = svd(h);
K1 = A(:,1)*sqrt(B(1,1));
K2 = C(:,1)*sqrt(B(1,1));
tic
result_1d = conv2(K1,K2,double(image),'same');
toc
figure('Name','Gaussian filter using combination of 1-D convolutions');
imshow(uint8(result_1d));
```

Using a stop watch (tic and toc functions MATLAB), we evaluate the time taken to do smoothing directly and as a combination of two filters. The direct method (used in 3) took 0.001687 seconds whereas doing it as combination of two filters was slightly faster as it took 0.001392 seconds for sigma=4. It's a similar trend when sigma is 6 – direct filter took 0.001667 seconds whereas the combination of two decomposed filters took 0.001636 seconds. The result is displayed in Figure 9.

Figure 9: Gaussian smoothing at sigma 2 as a combination of two filters

PART B:

B1. Gradient-based edge detection

The steps involved in detecting edges from gradient are:

- Smooth the image using a Gaussian filter to remove noise.
- Obtain the gradient magnitude by using the finite difference masks
- Find the gradient magnitude and direction at each pixel.
- Then we implement the non-maximal suppression as below
    - At each pixel, get the angle of direction of the gradient. And find the nearest rounded angle (0 or 45 or 90 or 135). If the rounded angle is 0 degrees, then edge is in the north—south direction and take the magnitude of the pixels in the perpendicular direction (east-west) and find if the pixel at current location is greater than them. If yes, then we store the value as is, otherwise we suppress it by making it zero
    - We determine similarly for all other pixels similarly if they must be suppressed depending on the magnitude of the neighboring pixels in the direction perpendicular to the gradient at that pixel.
- Display the image with various sigma values and thresholds.

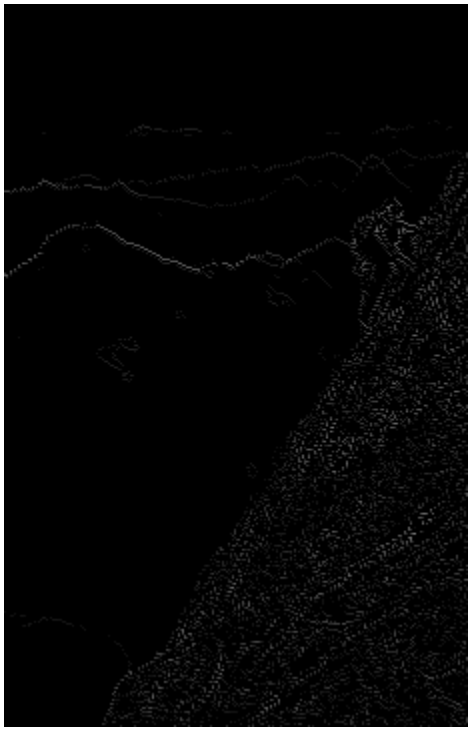

Figure 10: Edges detected with sigma = 2 and threshold = 8

In Figure 10, we see the result for a specific choice of sigma and the gradient threshold. We see that lot of the edges are visible and sometimes even non-edge points get more prominence.

On the other hand, if we observe in Figure 11, edges are finer but some prominent edges in the background get wiped as the threshold is higher.

Figure 12 is somewhere in the between and probably the more acceptable result.

Figure 11: Edges detected with sigma=4 and threshold = 18

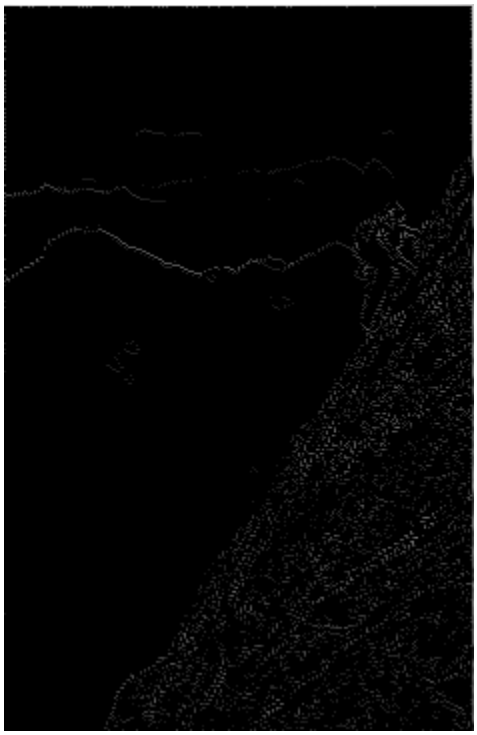

Figure 12: Edges detected with sigma = 6 and threshold = 12

The code snippet for non-maximal suppression is below:

```
threshold = 12;
for i=2:m
    for j=2:n
        if vals(i,j) == 0
            dir = Gdir(i,j);
            rounded_angle = findRoundedAngle(dir);
            if rounded_angle==0
                left = Gmag(i-1,j);
                right = Gmag(i+1,j);
                max_val = max(Gmag(i,j),max(left,right));
                if max_val ~= Gmag(i,j) || Gmag(i,j)<threshold
                    Gmag(i,j)=0;
                end
            elseif rounded_angle==90
                up = Gmag(i,j-1);
                down = Gmag(i,j+1);
                max_val = max(Gmag(i,j),max(up,down));
                 if max_val ~= Gmag(i,j) || Gmag(i,j)<threshold
                    Gmag(i,j)=0;
                 end

            elseif rounded_angle==135
                northwest = Gmag(i-1,j-1);
                southeast = Gmag(i+1,j+1);
                max_val = max(Gmag(i,j),max(northwest,southeast));
                 if max_val ~= Gmag(i,j) || Gmag(i,j)<threshold
                    Gmag(i,j)=0;
                 end
            elseif rounded_angle==45
                northeast = Gmag(i+1,j-1);
                southwest = Gmag(i-1,j+1);
                max_val = max(Gmag(i,j),max(northeast,southwest));
                 if max_val ~= Gmag(i,j) || Gmag(i,j)<threshold
                    Gmag(i,j)=0;
                 end
            end
        end
    end
end
```

B2. OCR

For the given image, we choose the below 5 templates for matching.

Figure 13: Templates used in OCR process for matching.

We use correlation (corr2) between each of the templates and the portions of sub image (portion of the image which is the size of the template). We determine the areas with maximum correlation and highlight the region with rectangle.

We detect all the templates in the image accurately as shown in Figure 14. We ensure that each of the sub image templates sum upto zero if their mean is equal to each of the pixels.

The code snippet for template matching is:

```
for i=1:num
    template_image = rgb2gray(images{i});
    [h,w] = size(template_image);

    corr_matrix = zeros(H,W);

    for j=1:H-h
        for k=1:W-w
                sub_image = double(image(j:(j+h-1),k:(k+w-1)));
                if mean(sub_image(:)) == sub_image(1,1)
                    for s=1:h
                        for t=1:w
                            if mod(t,2)==0
                                sub_image(s,t) = (-1)*sub_image(s,t);
                            end
                        end
                    end
                end
                result = corr2(template_image,sub_image);
                if isnan(result)
                   result=0;
                end
                corr_matrix(j,k)=result;
            end
         end
        % max_corr = max(abs(corr_matrix(:)));
         [maxCorrValue, maxIndex] = max(abs(corr_matrix(:)));
```

```
    [yPeak, xPeak] = ind2sub(size(corr_matrix),maxIndex(1));
        rectangle('Position',[xPeak yPeak h w],'edgecolor','r');
    end


end
```

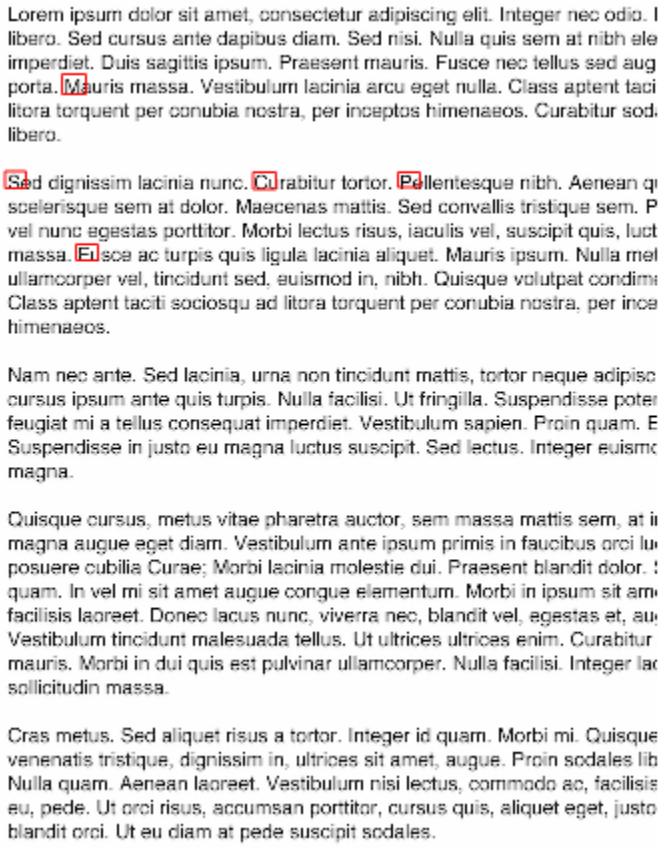

Figure 14: Red highlighted boxes indicating template matches

The confusion matrix C is perfectly diagonal as all the letters are accurately classified. Therefore, the matrix would be as in Figure 15.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 15: Confusion matrix – rows would indicate letters P, S, M, F and C, all of which are accurately classified.

NOTE: The corr2 function is very slow and hence the time taken for the letters to be identified and the rectangular boxes to be overlaid over them can be in the order of minutes.