CZ4003 Computer Vision

Lab 2 Report

Zhang Yuhan

U1823060F

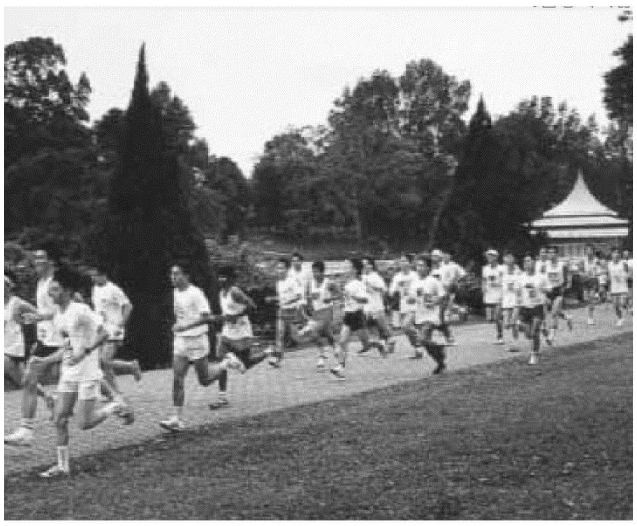
17 November 2020

Table of Contents

Edge Detection

a) Download `macritchie.jpg' from edveNTUre and convert the image to grayscale. Display the image.

Input	Output: Figure 1.1 Original Image (Grayscale)

`P=imread('maccropped.jpg');`
`Picture = rgb2gray(P);`
`imshow(Picture);`
	

b) Create 3x3 horizontal and vertical Sobel masks and filter the image using conv2. Display the edge-filtered images. What happens to edges which are not strictly vertical nor horizontal, i.e. diagonal?

Sobel operator/filter is used in image processing within edge detection algorithms where it is effective in emphasising edges within an image. It consists of a 3x3 mask for both horizontal and vertical components which can then be combined together. The concept of Sobel masks/filters are shown below:

-1	0	+1
-2	0	+2
-1	0	+1

x filter

+1	+2	+1
0	0	0
-1	-2	-1

y filter

Figure 1.2: Sobel masks

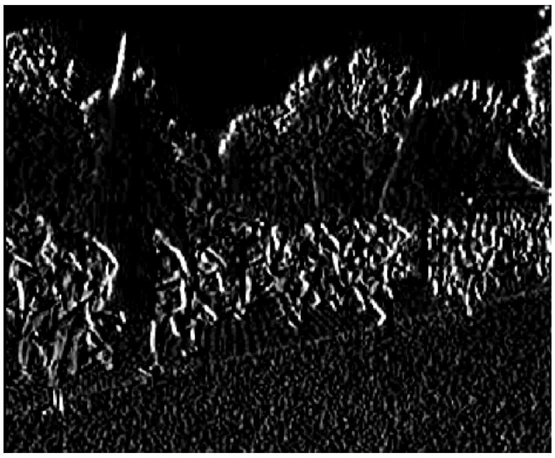
The implementation of Sobel masks in matlab is as follows:

Vertical jumps	Horizontal jumps

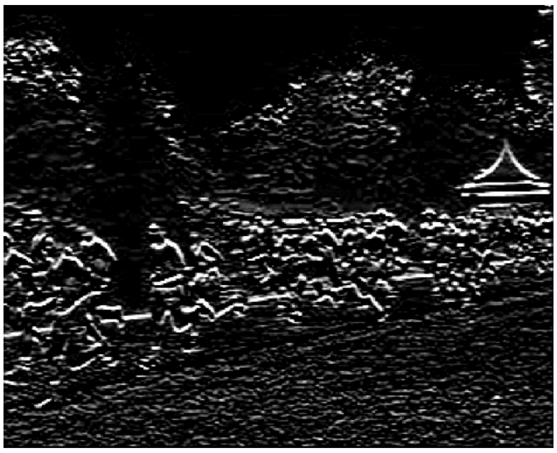
`sobel_mask_h = [`
`    -1 0 1;`
`    -2 0 2;`
`    -1 0 1];`
	`sobel_mask_v = [`
	`    1 2 1;`
	`    0 0 0;`
	`    -1 -2 -1;];`

These Sobel masks are then applied to the image in figure 1.1 using the **conv2** function to obtain the edge-filtered images:

Input	Output: Figure 1.3 horizontal edge-filtered

`pic_h = conv2(Picture, sobel_mask_h);`
`figure, imshow(uint8(pic_h))`
	

Input	**Output: Figure 1.4 vertical edge-filtered**

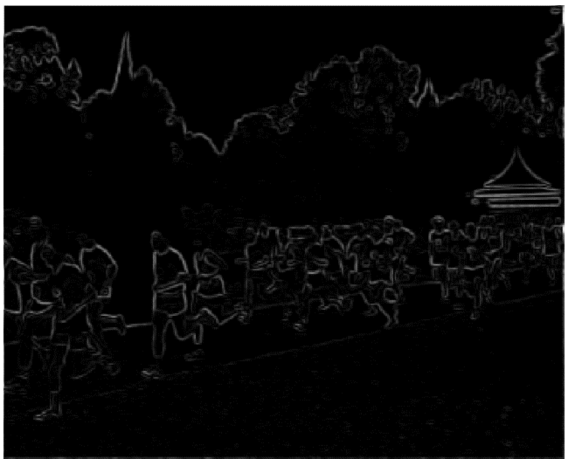
`pic_v = conv2(Picture, sobel_mask_v);`
`figure, imshow(uint8(pic_v))`
	

As can be observed from figures 1.3 and 1.4, the horizontal and vertical edges in the image is detected by their respective filters. Most of the diagonal lines disappears in both of the outputs since their filter will only result in the horizontal and vertical edges.

c) Generate a combined edge image by squaring (i.e. .^2) the horizontal and vertical edge images and adding the squared images. Suggest a reason why a squaring operation is carried out.

The image generated by combining the horizontal and vertical edge images is shown below:

Input	Figure 1.5 Normalized edge detection

`E = pic_v.^2 + pic_h.^2;`
`imshow((E),[])`
	

Input	Figure 1.6 $\lvert G \rvert$ edge detection

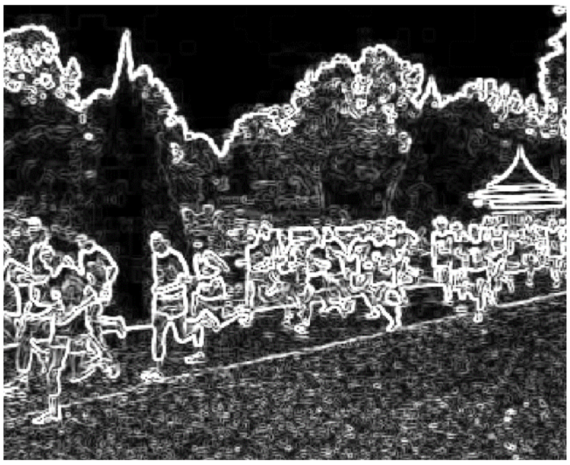
`E = pic_v.^2 + pic_h.^2;`
`figure, imshow(uint8(sqrt(E)))`
	

Figure 1.5 shows the normalized image of the combined edge image. Normalization is necessary in this case because many of the values in original combined edge image far exceeds the range supported by **imshow**. Since the squaring operation is done to obtain the gradient magntitude, as shown in the input fields above. The general equal for this is shown below:

$$|G| = \sqrt{Gx^2 + Gy^2}$$

Where Gx and Gy are the results from horizontal and vertical Sobel filters, and G is result of the combined edge filtering. This is applied in figure 1.6 where the square root of E is taken to find the gradient magnitude at each point.

d) Threshold the edge image E at value t by:

```
>> Et = E>t;
```

This creates a binary image. Try different threshold values and display the binary edge images. What are the advantages and disadvantages of using different thresholds?

Various different thresholds (t) were used to analyse its effects on the output. The array of t values and their associated implementations are shown below:

```
t = [10,10000,20000,40000,80000];
Et = E>t(1);
figure, imshow(Et)
Et = E>t(2);
figure, imshow(Et)
Et = E>t(3);
figure, imshow(Et)
Et = E>t(4);
figure, imshow(Et)
Et = E>t(5);
figure, imshow(Et)
```

The output for each case is shown below:

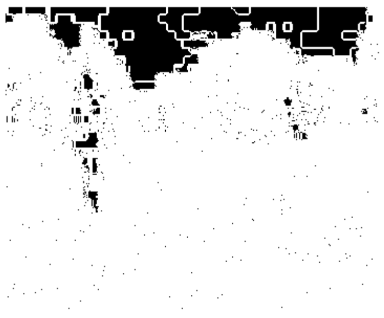
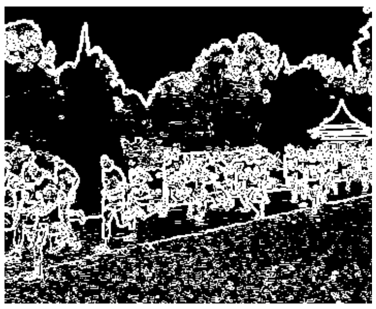
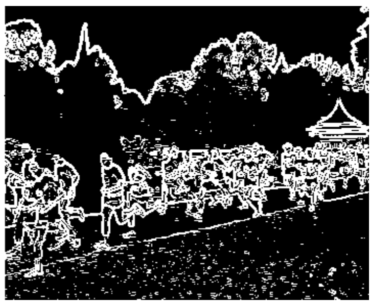
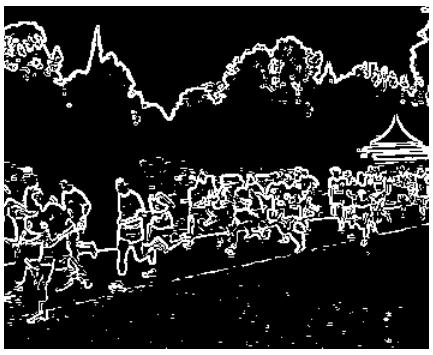
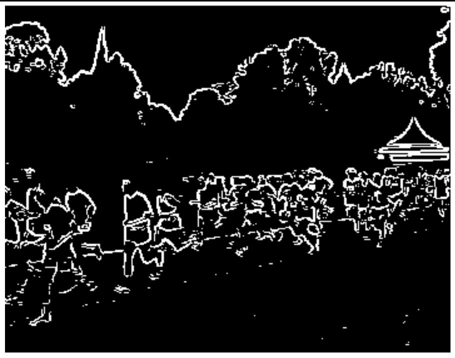
Figure 1.7: t = 10	Figure 1.8: t = 10000	Figure 1.9: t = 20000
		

Figure 1.10: t = 40000	Figure 1.11: t = 80000
	

For extremely small values of t, the threshold is not effective at all since most of the values in the image is significantly larger, as can be seen in figure 1.7. As can be observed from figures 1.8 to 1.11, the number of edges shown decreases as the threshold value, t, increases. For smaller values of t, more details of the image are shown, but the output is also subject to more noise, while for larger values of t, noise is reduced at the cost of potentially important edges.

e) Recompute the edge image using the more advanced Canny edge detection algorithm with tl=0.04, th=0.1, sigma=1.0

>> E = edge(I,'canny',[tl th],sigma);

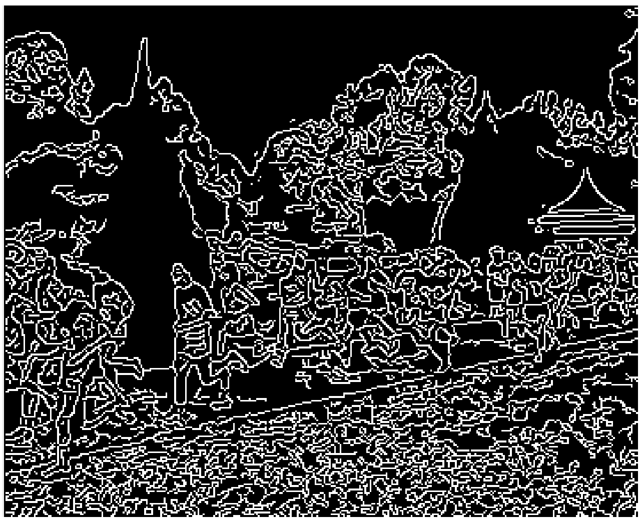
This generates a binary image without the need for thresholding.

i. Try different values of sigma ranging from 1.0 to 5.0 and determine the effect on the edge images. What do you see and can you give an explanation for why this occurs? Discuss how different sigma are suitable for (a) noisy edgel removal, and (b) location accuracy of edgels.

ii. Try raising and lowering the value of tl. What does this do? How does this relate to your knowledge of the Canny algorithm?

Canny edge detection algorithm is a multi-stage algorithm that is effective in removing noise while retaining information. The output generated by canny edge detection with tl = 0.04, th = 0.1, and sigma = 1.0 is shown below:

Input	Figure 1.12 Canny edge detection

`tl = 0.04;`
`th = 0.1;`
`sigma = 1.0;`
`E = edge(Picture,'canny',[tl th],sigma);`
`imshow(E);`
	

Various different sigma values were used to analyse its effects on the output. The array of sigma values and their associated implementations are shown below, the tl and th values remain constant at 0.04 and 0.1 respectively:

```
sigma_l = [1.0, 2.0, 3.0, 4.0, 5.0];
E_canny = edge(Picture,'canny',[tl th],sigma_l(1));
figure, imshow(E_canny)
E_canny = edge(Picture,'canny',[tl th],sigma_l(2));
figure, imshow(E_canny)
E_canny = edge(Picture,'canny',[tl th],sigma_l(3));
figure, imshow(E_canny)
E_canny = edge(Picture,'canny',[tl th],sigma_l(4));
figure, imshow(E_canny)
E_canny = edge(Picture,'canny',[tl th],sigma_l(5));
figure, imshow(E_canny)
```

The output for each case is shown below:

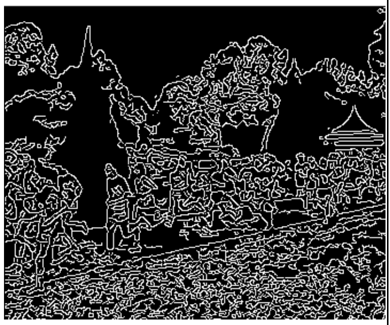
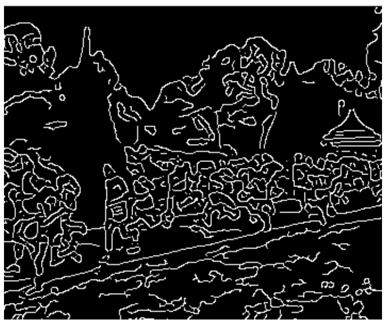
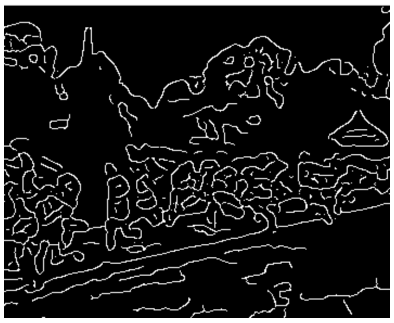
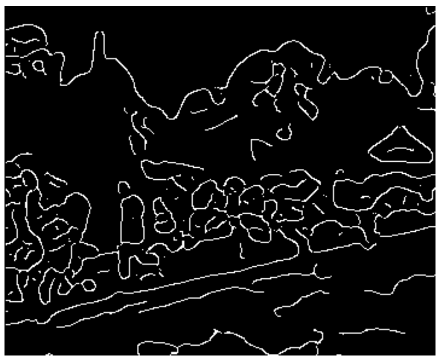
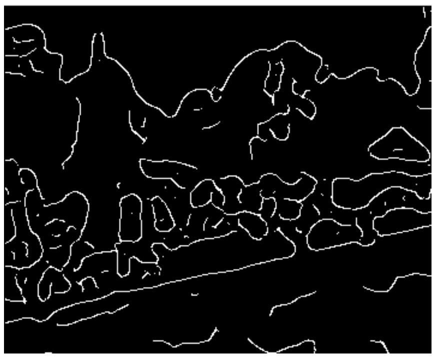
Figure 1.13: sigma = 1.0	Figure 1.14: sigma = 2.0	Figure 1.15: sigma = 3.0
		

Figure 1.16: sigma = 4.0	Figure 1.17: sigma = 5.0
	

As can be observed from figures 1.13 to 1.17 above, the noise in the output for increasing values of sigma decreases. However, this comes at the cost of loss of edges (information) as well as less accurate location of edges. Thus, it is important to find a balance between noise removal and edge accuracy.

Various different threshold low (tl) values were also used to analyse its effects on the output. The array of tl values and their associated implementations are shown below, the sigma and th values are constant at 1.0 and 0.1 respectively:

```
tl_l = [0, 0.02, 0.04, 0.06, 0.08];
E_canny = edge(Picture,'canny',[tl_l(1) th],sigma);
figure, imshow(E_canny)
E_canny = edge(Picture,'canny',[tl_l(2) th],sigma);
figure, imshow(E_canny)
E_canny = edge(Picture,'canny',[tl_l(3) th],sigma);
figure, imshow(E_canny)
E_canny = edge(Picture,'canny',[tl_l(4) th],sigma);
figure, imshow(E_canny)
E_canny = edge(Picture,'canny',[tl_l(5) th],sigma);
figure, imshow(E_canny)
```

The output for each case is shown below:

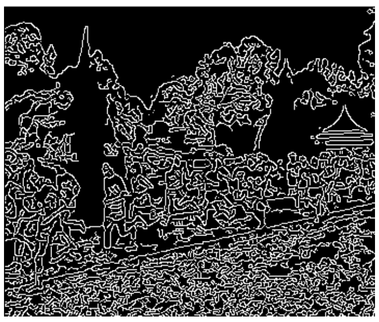
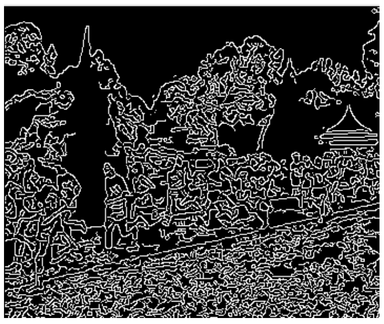
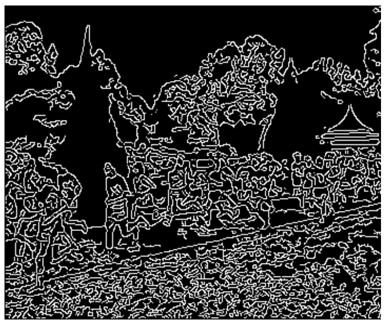
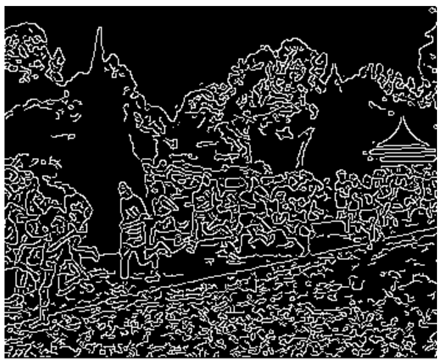
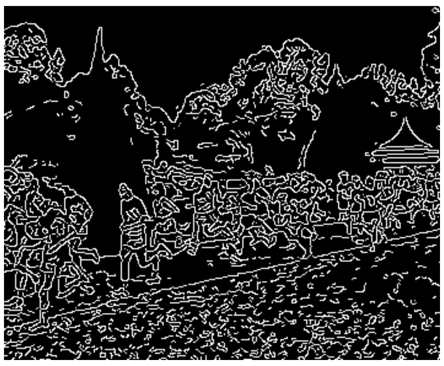
Figure 1.18: tl = 0	Figure 1.19: tl = 0.02	Figure 1.20: tl = 0.04
		

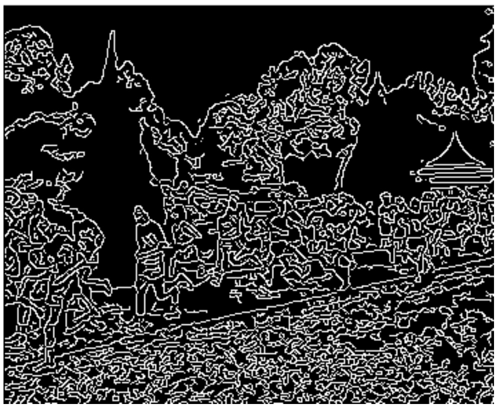
Figure 1.21: tl = 0.06	Figure 1.22: tl = 0.08
	

Canny edge detection uses hysteresis thresholding whereby only pixels in between the threshold low and threshold high will be considered an edge. By keeping the threshold high (th) constant and increasing the threshold low (tl), we are effectively reducing the number of pixels that will be considered an edge, this can be seen in figures 1.18 to 1.22. This method of thresholding is effective in reducing noise while retaining information, depending on the threshold values.

Line Finding Using Hough Transform

a) Reuse the edge image computed via the Canny algorithm with sigma=1.0.

Input	Output: Figure 2.1 Edge image with Canny

`P=imread('maccropped.jpg');`
`Picture = rgb2gray(P);`
 	
`tl = 0.04;`
`th = 0.1;`
`sigma = 1.0;`
`E = edge(Picture,'canny',[tl th],sigma);`
`imshow(E);`
	

b) As there is no function available to compute the Hough transform in MATLAB, we will use the Radon transform, which for binary images is equivalent to the Hough transform. Read the help manual on Radon transform and explain why the transforms are equivalent in this case. When are they different?

>> [H, xp] = radon(E);

Display H as an image. The Hough transform will have horizontal bins of angles corresponding to 0-179 degrees, and vertical bins of radial distance in pixels as captured in xp. The transform is taken with respect to a Cartesian coordinate system where the origin is located at the centre of the image, and the x-axis pointing right and the y-axis pointing up.

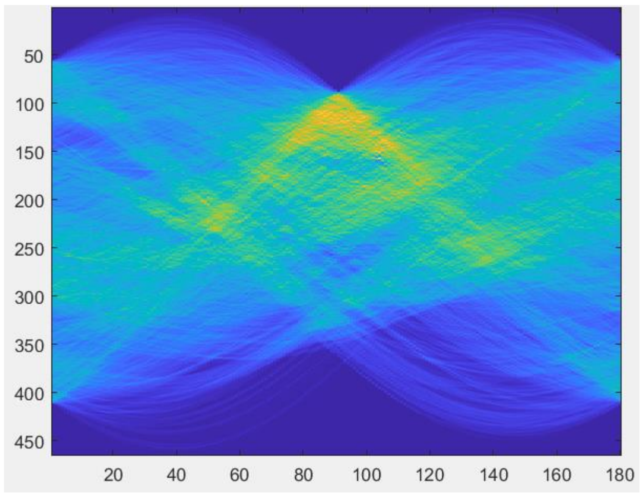
The help manual on Radon transform is as follow:

```
radon Radon transform.
   The radon function computes the Radon transform, which is the
   projection of the image intensity along a radial line oriented at a
   specific angle.

   R = radon(I,THETA) returns the Radon transform of the intensity image I
   for the angle THETA degrees. If THETA is a scalar, the result R is a
   column vector containing the Radon transform for THETA degrees. If
   THETA is a vector, then R is a matrix in which each column is the Radon
   transform for one of the angles in THETA. If you omit THETA, it
   defaults to 0:179.

   [R,Xp] = radon(...) returns a vector Xp containing the radial
   coordinates corresponding to each row of R.
```

Radon transform is defined for continuous functions on hyperplanes while Hough transform is a discrete algorithm. Thus, Radon transform will achieve more accurate results compared to Hough transform but is inherently much slower because of its continuous nature. The applications of Hough transform are generally in image processing (line detection) while applications of Radon transform are generally in fields that require more precision. In this case, the Radon transform will be the same as Hough transform since the inputs are binary number, which are discrete. If continuous numbers are used instead, the outputs of the two transform methods will differ slightly (radon will be more accurate).

The inputs and results of running Radon transform on the image in Figure 2.1 are shown below:

Input	Output: Figure 2.2 Radon transform

`[H, xp] = radon(E);`
`class(H)`
`% Convert H from type 'double' to 'uint8' to display`
`figure,imagesc(uint8(H));`
`colormap('default');`
	

c) Find the location of the maximum pixel intensity in the Hough image in the form of [theta, radius]. These are the parameters corresponding to the line in the image with the strongest edge support.

The inputs and results for finding maximum pixel intensity are shown below:

Input	Output: Figure 2.2 Maximum pixel intensity

`% Finding maximum pixel intensity`
`maxValue = max(H(:));`
`[radius, theta] = find(H == maxValue);`
	`theta =`
	
	`104`
	
	`radius =`
	
	`157`

As shown in Figure 2.2, the maximum pixel intensity is found at theta = 104 and radius = 157 of the Hough image from Figure 2.1.

d) Derive the equations to convert the [theta, radius] line representation to the normal line equation form Ax + By = C in image coordinates. Show that A and B can be obtained via

```
>> [A, B] = pol2cart(theta*pi/180, radius);
>> B = -B;
```

B needs to be negated because the y-axis is pointing downwards for image coordinates. SC437 Computer Vision & Image Processing Experiment 2 Page 3 Find C.

Reminder: the Hough transform is done with respect to an origin at the centre of the image, and you will need to convert back to image coordinates where the origin is in the top-left corner of the image.

Since the general equations for converting from polar form to cartesian form are:

$$x = rcos(\theta)$$
$$y = rsin(\theta)$$
$$xcos(\theta) + ysin(\theta) = r$$

We can say that:

$$A = radius \times \cos\left(\frac{\theta\pi}{180}\right)$$
$$B = radius \times \sin\left(\frac{\theta\pi}{180}\right)$$
$$\therefore xr\cos\left(\frac{\theta\pi}{180}\right) + yr\sin\left(\frac{\theta\pi}{180}\right) = r^2$$
$$Ax + By = r^2$$

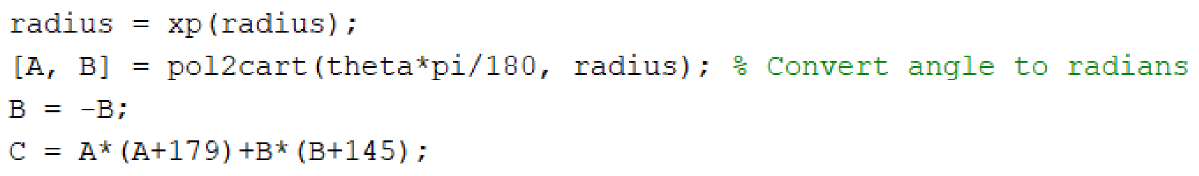
After this, we must transform the origin back to the top-left of the image, dimension of the image is (358x290):

$$A(x - 179) + B(y - 145) = r^2$$

Which simplifies to:

$$C = r^2 + 179A + 145B$$
$$\therefore C = A^2 + 179A + B^2 + 145B$$

The implementation of this is shown below:

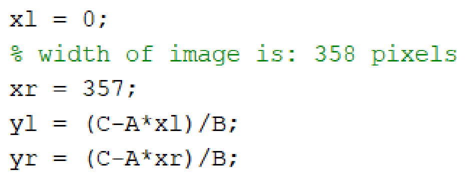
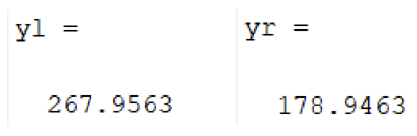


```
radius = xp(radius);
[A, B] = pol2cart(theta*pi/180, radius); % Convert angle to radians
B = -B;
C = A*(A+179)+B*(B+145);
```

e) Based on the equation of the line Ax+By = C that you obtained, compute yl and yr values for corresponding xl = 0 and xr = width of image - 1.

The input and result of this is shown below:

Input	Output: yl and yr values

 `xl = 0;`
`% width of image is: 358 pixels`
`xr = 357;`
`yl = (C-A*xl)/B;`
`yr = (C-A*xr)/B;`
	 `yl =`
	
	`267.9563`
	
	`yr =`
	
	`178.9463`

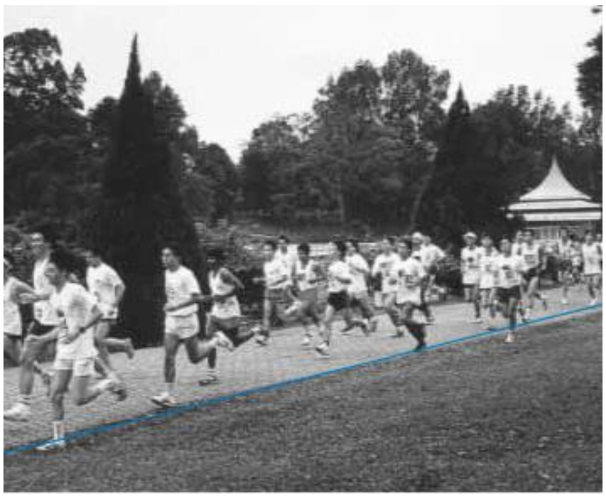
f) Display the original ‘macritchie.jpg’ image. Superimpose your estimated line by

>> line([xl xr], [yl yr]);

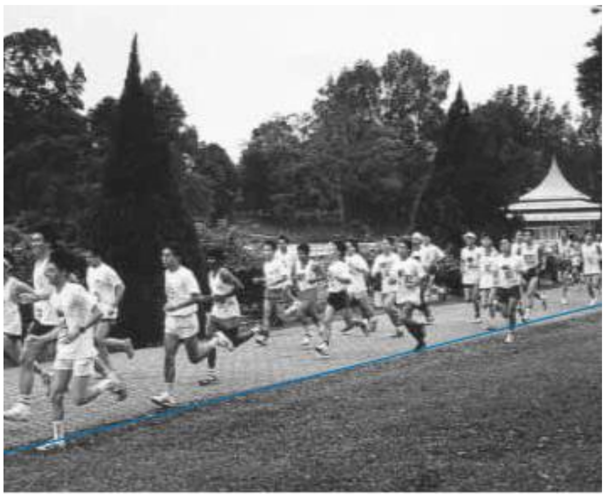
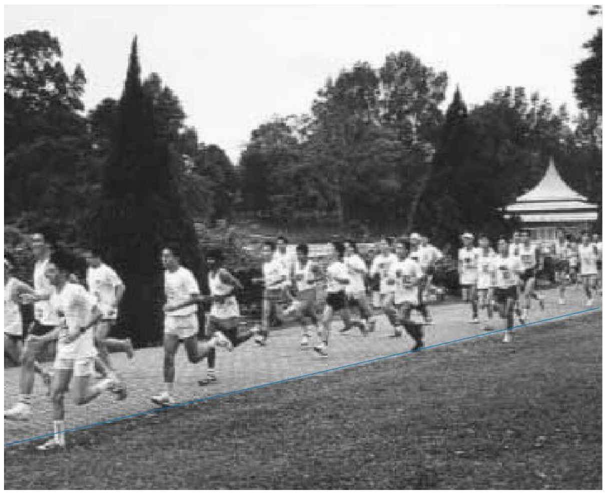
Does the line match up with the edge of the running path? What are, if any, sources of errors? Can you suggest ways of improving the estimation?

The input and results of this is shown below:

Input	Output: Figure 2.3 Estimated line

`figure, imshow(Picture)`
`line([xl xr], [yl yr])`
	

The estimated line shown in Figure 2.3 matches the edge of the running path relatively well. As can be observed, the slope of the line is slightly larger than that of the running path. This may be due to some noise and disturbances when running the Canny edge detection algorithm. To reduce the noise, I attempted to increase the value of sigma from 1.0 to 5.0, the results and comparisons with the previous results are shown below:

Figure 2.3	Output: Figure 2.4 Estimated line with sigma = 5.0
	

As can be observed from figures 2.3 and 2.4, the line with sigma = 5.0 matches the running path slightly better than that of the line with sigma = 1.0. This may be further optimized by tweaking other parameters such as the hysteresis threshold and other values that may have an effect on denoising the image.

3D Stereo

The overview of the algorithm is: for each pixel in Pl,

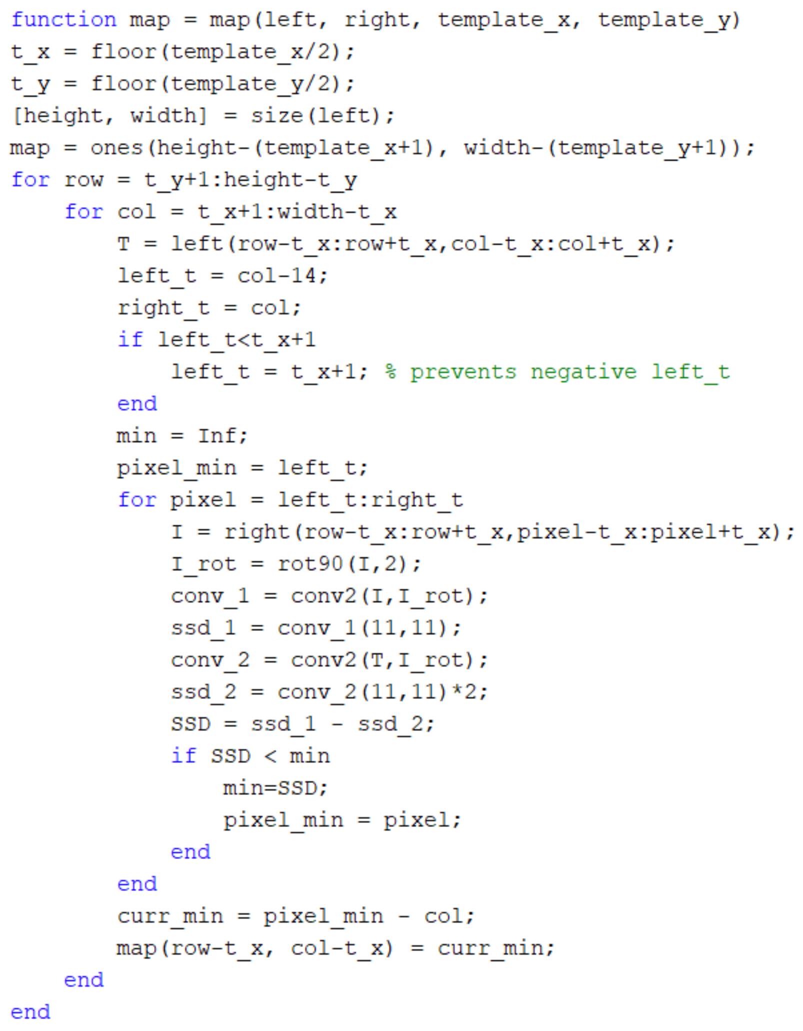
i. Extract a template comprising the 11x11 neighbourhood region around that pixel.
ii. Using the template, carry out SSD matching in Pr, but only along the same scanline. The disparity is given by

$$\mathrm{d}(x_l, y_l\ ) = x_l - \hat{x}_r$$

where xl and yl are the relevant pixel coordinates in Pl, and r $\hat{x}_r$ is the SSD matched pixel's x-coordinate in Pr. You should also constrain your horizontal search to small values of disparity (<15). Noted that you may use **conv2**, **ones** and **rot90** functions (may be more than once) to compute the SSD matching between the template and the input image. Refer to the equation in section 2.3 for help.
iii. Input the disparity into the disparity map with the same Pl pixel coordinates.

a) Write the disparity map algorithm as a MATLAB function script which takes two arguments of left and right images, and 2 arguments specifying the template dimensions. It should return the disparity map. Try and minimize the use of for loops, instead relying on the vector / matrix processing functions.

The implementation of the disparity map algorithm is shown below:



```
function map = map(left, right, template_x, template_y)
t_x = floor(template_x/2);
t_y = floor(template_y/2);
[height, width] = size(left);
map = ones(height-(template_x+1), width-(template_y+1));
for row = t_y+1:height-t_y
    for col = t_x+1:width-t_x
        T = left(row-t_x:row+t_x,col-t_x:col+t_x);
        left_t = col-14;
        right_t = col;
        if left_t<t_x+1
            left_t = t_x+1; % prevents negative left_t
        end
        min = Inf;
        pixel_min = left_t;
        for pixel = left_t:right_t
            I = right(row-t_x:row+t_x,pixel-t_x:pixel+t_x);
            I_rot = rot90(I,2);
            conv_1 = conv2(I,I_rot);
            ssd_1 = conv_1(11,11);
            conv_2 = conv2(T,I_rot);
            ssd_2 = conv_2(11,11)*2;
            SSD = ssd_1 - ssd_2;
            if SSD < min
                min=SSD;
                pixel_min = pixel;
            end
        end
        curr_min = pixel_min - col;
        map(row-t_x, col-t_x) = curr_min;
    end
end
```

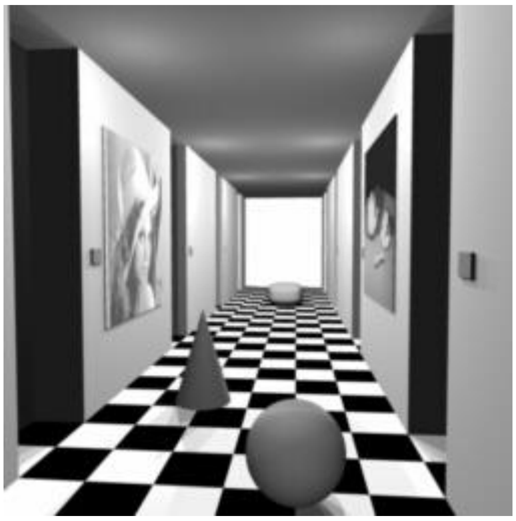
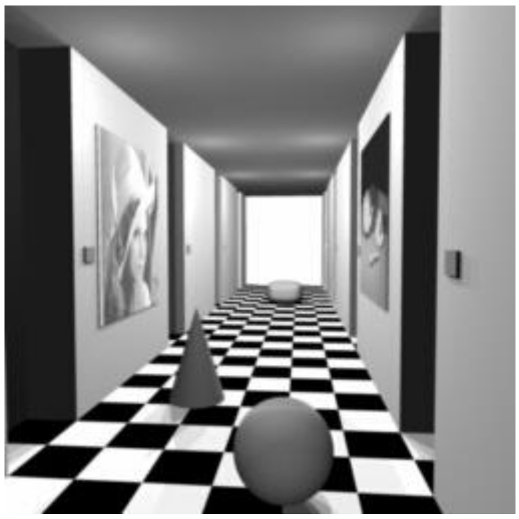
The code for the above map function is present in the file map.m.

b) Download the synthetic stereo pair images of ‘corridorl.jpg’ and ‘corridorr.jpg’, converting both to grayscale.

The inputs and outputs of this is shown below:

Inputs

```
left = imread('corridorl.jpg');
left = rgb2gray(left);
figure,imshow(left);
right = imread('corridorr.jpg');
right = rgb2gray(right);
figure,imshow(right);
```

Figure 3.1: greyscale image of corridorl.jpg	Figure 3.2: greyscale image of corridorr.jpg
	

c) Run your algorithm on the two images to obtain a disparity map D, and see the results via

>> imshow(-D,[-15 15]);

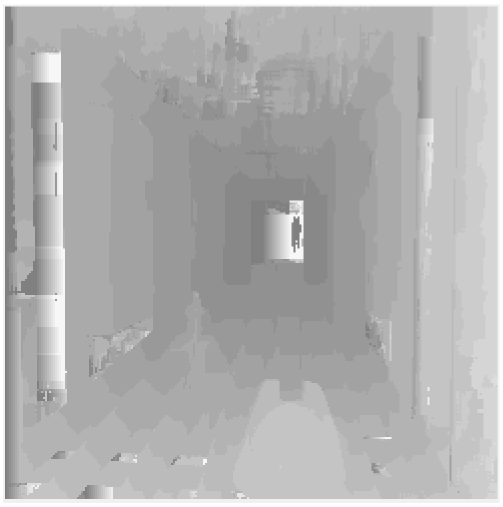
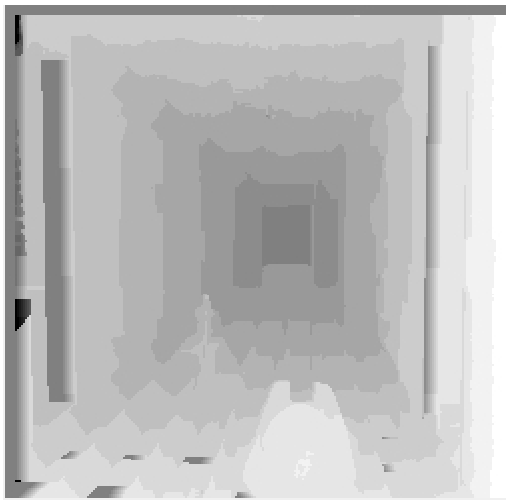
The results should show the nearer points as bright and the further points as dark. The expected quality of the image should be similar to `corridor_disp.jpg’ which you can view for reference.

Comment on how the quality of the disparities computed varies with the corresponding local image structure.

The inputs and outputs of running the disparity map algorithm is shown below with a comparison with ‘corridor_disp.jpg’:

Inputs

```
D = map(left,right,11,11);
figure,imshow(-D,[-15,15])
disp = imread('corridor_disp.jpg');
figure, imshow(disp);
```

Figure 3.3: Resultant image from map function	Figure 3.4: 'corridor_disp.jpg'
	

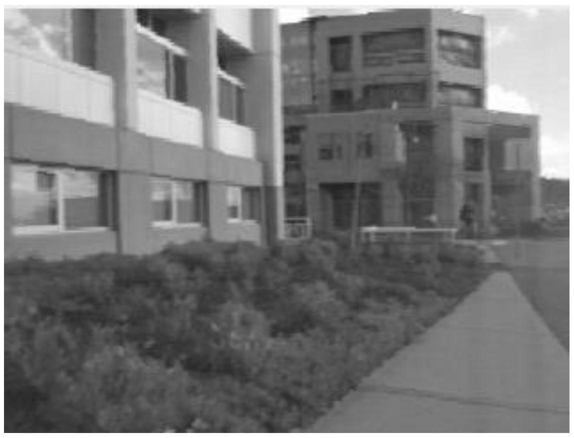
The resultant image from Figure 3.3 is generally similar to the optimal disparity map in Figure 3.4. However, it can be observed that the centre of the resultant image should be dark instead of bright. This may be due to the fact that in the SSD algorithm, summation is carried out over the squared differences in pixel intensities between two corresponding pixels in the window. This means that occlusion error may occur, especially in cases with large spots with similar intensities.

d) Rerun your algorithm on the real images of 'triclops-i2l.jpg' and triclops-i2r.jpg'. Again you may refer to 'triclops-id.jpg' for expected quality. How does the image structure of the stereo images affect the accuracy of the estimated disparities?

The images used as input into the algorithm is shown below:

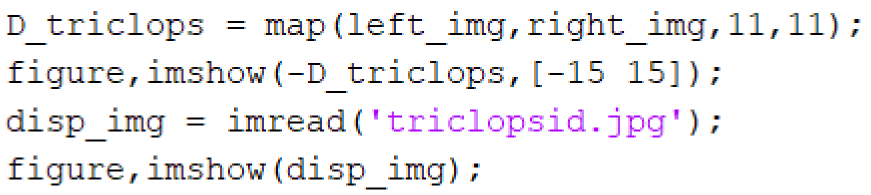
Inputs

```
left_i = imread('triclopsi2l.jpg');
left_img = rgb2gray(left_i);
figure,imshow(left_img);
right_i = imread('triclopsi2r.jpg');
right_img = rgb2gray(right_i);
```

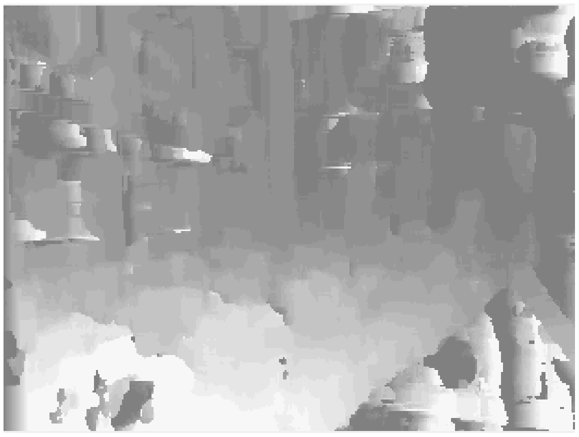
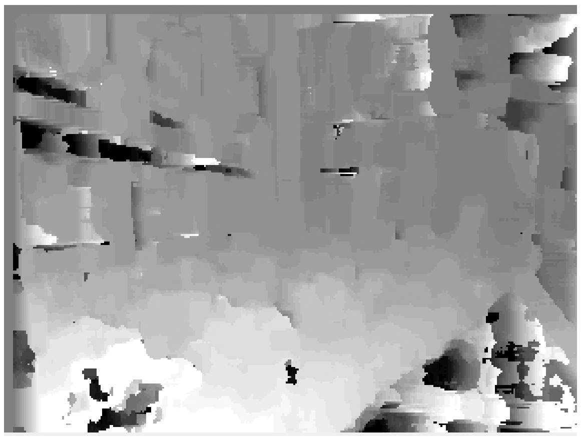
Figure 3.4: ‘triclops-i2l.jpg’	Figure 3.5: ‘triclops-i2r.jpg;
	

The inputs and outputs of running the disparity map algorithm on ‘triclops-i2l.jpg’ and ‘triclops-i2r.jpg’ is shown below with a comparison with ‘triclops-id.jpg’:

Inputs



```
D_triclops = map(left_img,right_img,11,11);
figure,imshow(-D_triclops,[-15 15]);
disp_img = imread('triclopsid.jpg');
figure,imshow(disp_img);
```

Figure 3.6: Resultant image from map function	Figure 3.7: ‘triclops-id.jpg’
	

As can be observed from comparing Figure 3.6 against the optimal disparity map in Figure 3.7, there are some spots that were mapped wrongly. For example, there are some sections in the left and bottom right of the resultant map that were mapped much lighter than their corresponding sections in the optimal map. This may be due to the occlusion error as mentioned in the previous question. A possible solution to this problem is to reduce the window size during mapping. This may increase the accuracy of the resultant disparity map but this also means that the algorithm is subject to more noise in the images.