I'm going to try and make this seem a little bit like a lab report, including the Exercises et cetera. This is to make it clear and easy to read, rather than plastering several pages of code. I will be testing for trees, blood, CL3 and CL1. Though I will not test circle detection for trees for obvious reasons.

Exercise 1 – Edge Detection

Write a first Matlab function that finds the gradient M for a given input image. The function should work for images of different sizes and produce an image representation of the gradient, the same size as the original image, where the value of each point is the gradient at that point.

Note that the Matlab function filter2 will perform a two-dimensional filter for you. Your function should not use filter2 or any equivalent functions but should implement a two-dimensional sliding window filter using for loops or similar. You can, however, use filter2 to check the results your function produces. Don't forget to think about what happens at the image border.

Your function should have the form

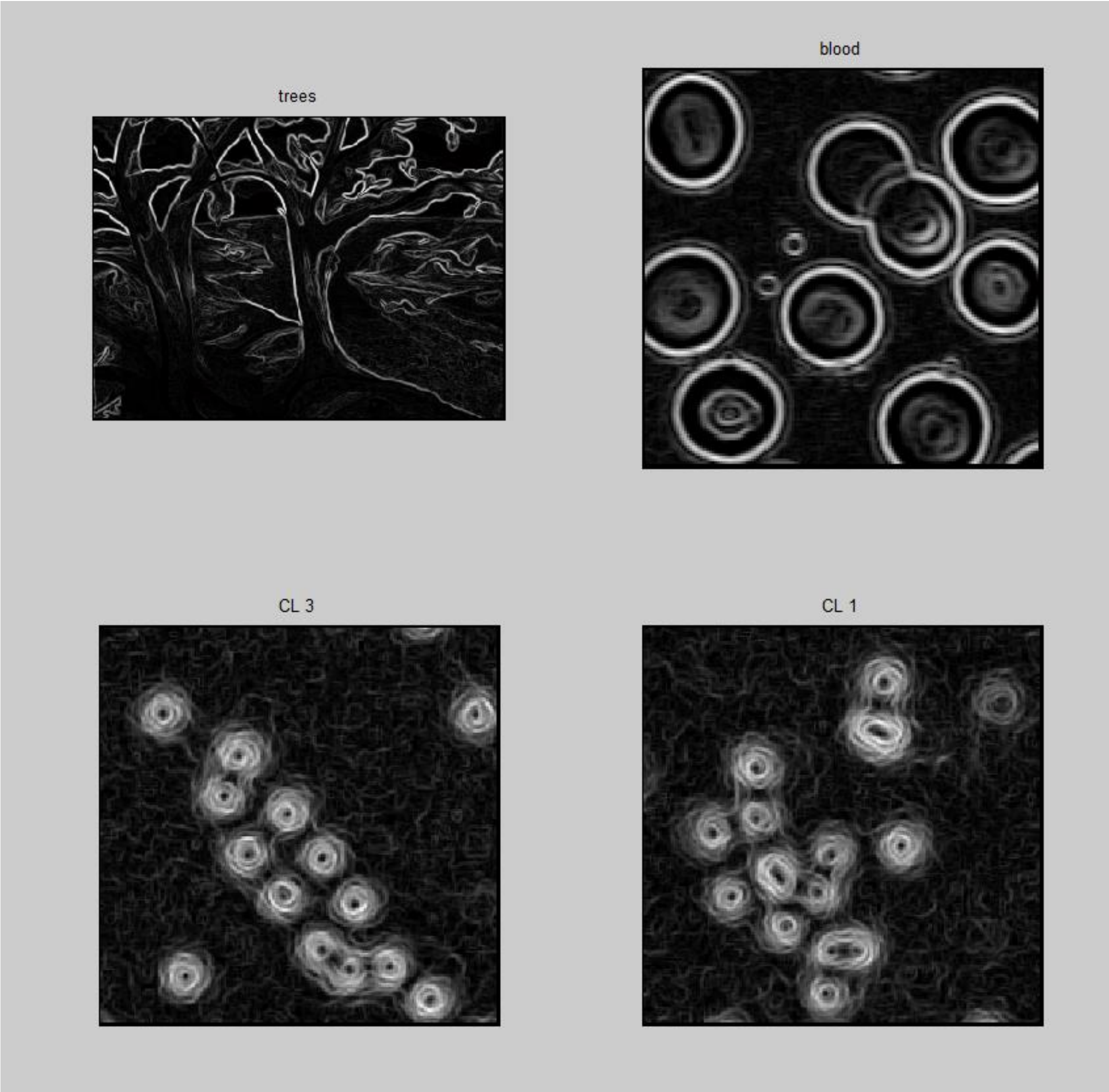
function [out_image]=gradient_of_image(in_image,filter,maxvalue)

where in_image is the input image (a 2D matrix), filter are the number of pixels at the edge that are given a gradient of zero, and maxvalue gives the maximum value of the gradient after normalisation.

File used: gradient_of_image.m
Inserted in command window:

```
>> load trees
>> load blood.mat
>> load CL_1.mat
>> load cl_3.mat
>> subplot(2,2,1)
>> imshow(gradient_of_image(X,0.5,1))
>> title('trees');
>> subplot(2,2,2)
>> title('blood');
>> imshow(gradient_of_image(blood,0.5,1))
>> subplot(2,2,3)
>> title('CL 3');
>> imshow(gradient_of_image(CL_3,0.5,1))
>> subplot(2,2,4)
>> title('CL_1');
>> imshow(gradient_of_image(CL_1,0.5,1))
```

Resulted images:



Code used for gradient_of_image.m:

Start of function

```
function [out_image]=gradient_of_image(in_image,filter,maxvalue)
```

Putting in the Sobel Masks basis multipliers

```
Mx = [-1, 0, 1; -2, 0, 2; -1, 0, 1];    % Horizontal sobelmask
My = [-1, -2, -1; 0, 0, 0; 1, 2, 1];    % Vertical sobelmask
```

Loading in necessary variables

```
loadedImage = double(in_image);         % Loading in the image as a double
[loop1,loop2] = size(loadedImage);      % Sets the length of the loop
Mcoll = zeros(loop1, loop2);            % pre-making the variable Mcoll
```

Edge Mask creation loop

```
for i=1:loop1
    for j=1:loop2
```

If the coordinate falls out of the given size, it will skip the current loop and go to the next one

```
           if i-1 < filter              % Filtering out the negative x boundary
                continue                % Continue makes the for loop go to the "next
step"
           elseif j-1 < filter          % Filtering out the negative y boundary
               continue
           elseif i+1 > (loop1 - filter) % Filtering out max+1 x boundary
                continue
           elseif j+1 > (loop2 - filter) % Filtering out max+1 y boundary
               continue
           end
```

Determining the mask coordinates

```
           val1 = loadedImage(i-1 ,j-1);    %Top Left
           val2 = loadedImage(i   ,j-1);    %Top Mid
           val3 = loadedImage(i+1 ,j-1);    %Top Right
           val4 = loadedImage(i-1 ,j  );    %Mid Left
           val5 = loadedImage(i   ,j  );    %Mid Mid
           val6 = loadedImage(i+1 ,j  );    %Mid Right
           val7 = loadedImage(i-1 ,j+1);    %Bot Left
           val8 = loadedImage(i   ,j+1);    %Bot Mid
           val9 = loadedImage(i+1 ,j+1);    %Bot Right
```

Putting the values in a vector, preparing for the for loop

```
           posColl = [val1, val2, val3, val4, val5, val6, val7, val8, val9];
```

MxMerge (horizontal image gradient)

```
            totMX = 0;                                 % Creating the variable totMX
            for mxloop=1:9                             % Looping through it 9 times
(once for every mask coordinate)
               tMerge = posColl(mxloop) * Mx(mxloop);    % tMerge (temporary merge) is
made by putting the collection of positions times the Sobel Mask Multiplier
               totMX = totMX + tMerge;                 % Putting them together (this
happens 9 times)
            end
```

MyMerge (vertical image gradient)

```
            totMY = 0;                                 % Creating the variable totMY
            for myloop=1:9                             % Looping through it 9 times
(once for every mask coordinate)
               tMerge = posColl(myloop) * My(myloop);    % tMerge (temporary merge) is
made by putting the collection of positions times the Sobal Mask
               totMY = totMY + tMerge;                 % Putting them together (this
happens 9 times)
            end
```

Formula to combine horizontal and vertical image gradient

```
            M = sqrt(totMX^2+totMY^2);
```

Putting it all together into one variable, Mcoll

```
            Mcoll(i, j) = M;
```

```
    end
end
```

Applying the maxvalue to every part of the image

```
highestNo = max(max(Mcoll));                           % Finding the absolute maximum
value in the gradient image
for i=1:loop1
    for j=1:loop2
        Mcoll(i,j) = (Mcoll(i,j)/highestNo) * maxvalue; % Multiplying the maximum value
by the variable location over the maximum value in the existing gradient image
    end
end
```

Displaying the gradient image

```
out_image = Mcoll;
```

```
end
```

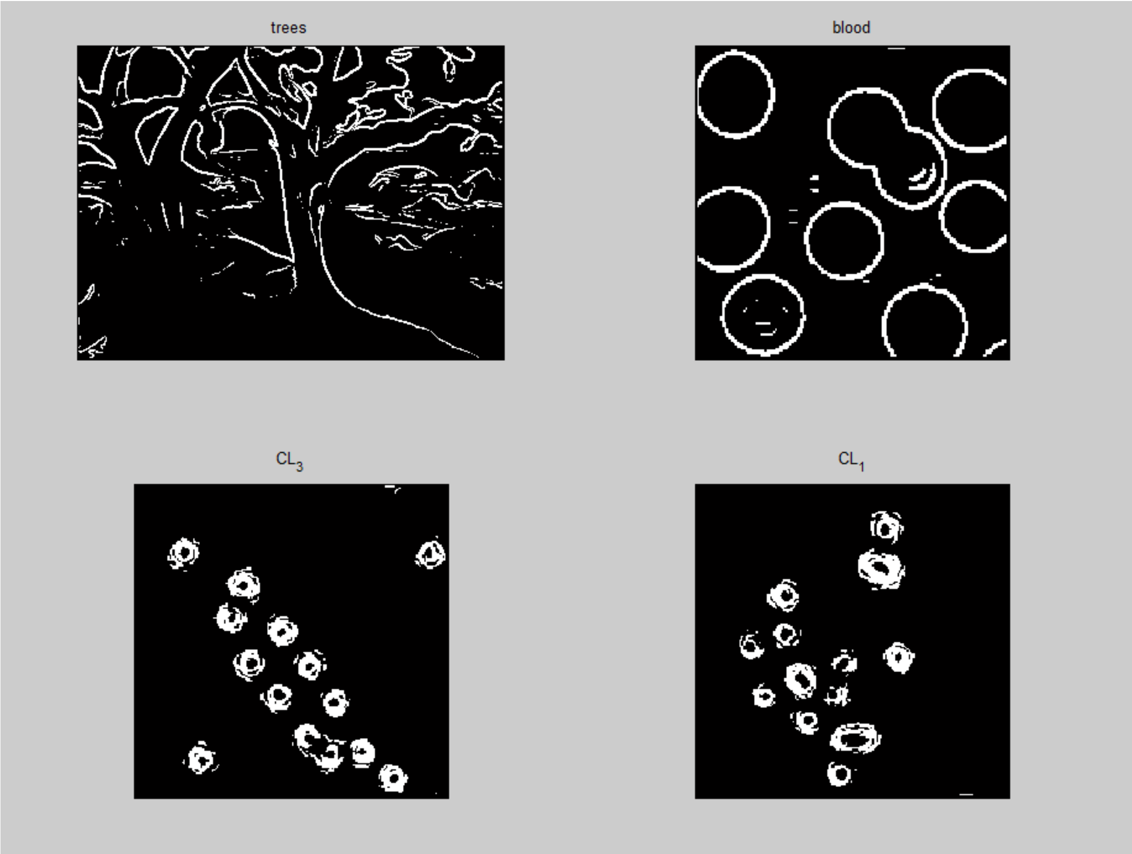
Part two of Exercise one required:

> *Now, investigate the effect of thresholding the gradient image with different values to produce a binary image in which pixels are either edge or non-edge.*
>
> *Create a second function with the form*
>
> ***function [edge_image]=edge_detect(in_image, threshold)***
>
> *where in_image is the input image, threshold is a value between 0-1, where 1 corresponds to the maximum gradient in the image. edge_image is an output matrix with values of 0 or 1 depending on whether a pixel is at a feature edge or not.*

The following was input in the command window:

```
>> subplot(2,2,1)
>> edgeX = edge_detect(X, 0.3);
>> imshow(edgeX)
>> title('trees')
>> subplot(2,2,2)
>> edgeblood = edge_detect(blood, 0.5);
>> imshow (edgeblood)
>> title('blood')
>> subplot(2,2,3)
>> edgeCL_3 = edge_detect(CL_3, 0.5);
>> imshow(edgeCL_3)
>> title('CL_3');
>> subplot(2,2,4)
>> edgeCL_1 = edge_detect(CL_1, 0.5);
>> imshow(edgeCL_1)
>> title('CL_1');
```

Resulted image:



Code used for edge_detect.m:

Start of function

```
function [edge_image]=edge_detect(in_image, treshold)
```

Determining necessary variables

```
gradientImage = gradient_of_image(in_image, 1, 1);  %Taking the gradientof the input
image
[loop1,loop2] = size(gradientImage);                 % setting the length of the loop to
the x and y size of the image
```

Looping through every part of the gradient image

```
for i=1:loop1
    for j=1:loop2
        if gradientImage(i,j) > treshold             %Checking if the position
corresponds to above or below the treshold
        gradientImage(i,j) = 1;   %Setting it to black if it's above the treshold
        else
        gradientImage(i,j) = 0;   %Setting it to white if it's under the treshold
        end
    end
end
```

Outputting the image seperately for increased encapsulation

```
edge_image = gradientImage;    % Simply outputting the gradient
```

```
end
```

Exercise two was as follows:

Exercise 2 – Circle Drawing

A key step in the Hough Transform is the ability to draw circles in an image (i.e. a circle in a matrix). Therefore Exercise 2 is to write a MATLAB function that will draw a circle of a given intensity value in an image. The format of the first line of the function should be as below

```
function [out_image] = draw_circle(in_image,x0,y0,r,value)
```

where value is the greyscale intensity value of the circle that is to be drawn in in_image, (x0,y0) are the coordinates of the centre of the circle, and r is the radius.

The circle that is drawn should ideally be a single-pixel-thickness, continuous line without any gaps in its border. Another point to consider is error checking to determine whether points are contained within the image.

File used: draw_circle.m

For this exercise I feel perhaps some explanation would be nice to go with the code. Of course all code is fully commentated and explained, but for extra information:

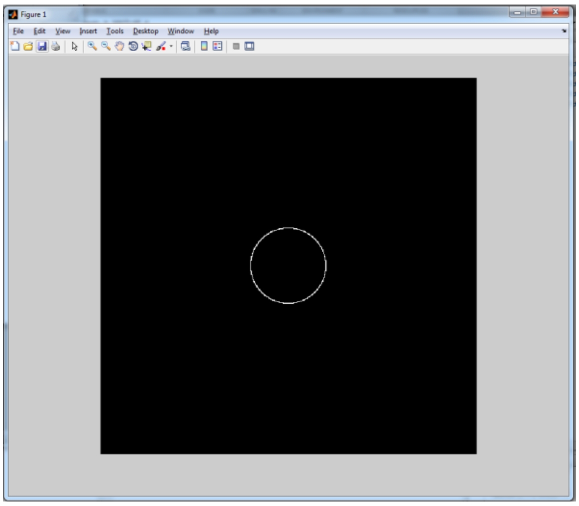
To draw the circle image, I made an angle by vectorising a variable called “angle”, from 0 until 2, with steps of 0.005, then multiplied all variables inside of the vector by pi. Then sin and cos are used, then merged and a circle is formed.

Because of how circles work, if they are out of the image, they will often either give errors or make the image look odd or incorrect. For this, if-elseif-else statements and try catches were used to prevent this from happening.

Rather than in the previous exercises, where I take all code and show it, to maintain the full aliasing of the circles, I will use a separate window for examples every time.

When the following is input:

```
>> Ex2Example = zeros(500,500);
>> circle1 = draw_circle(Ex2Example, 250, 250, 50, 1)
>> imshow(circle1)
```



The following is output:

This is just a simple example to show it works.

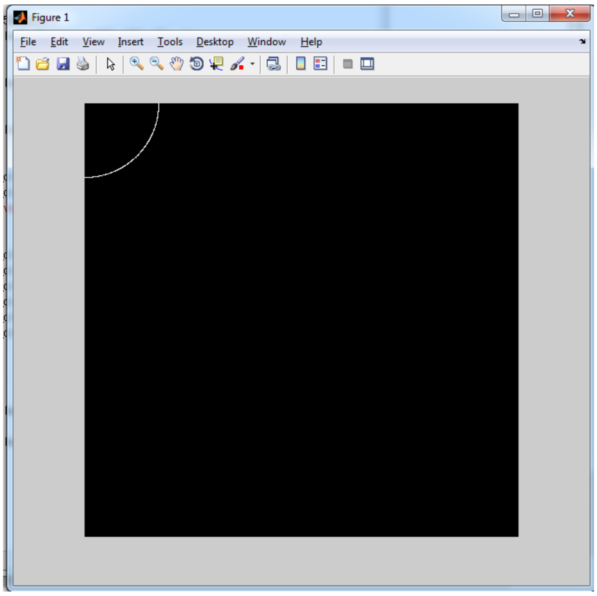
Now to show that if you go over the borders, it will not error or break the image:

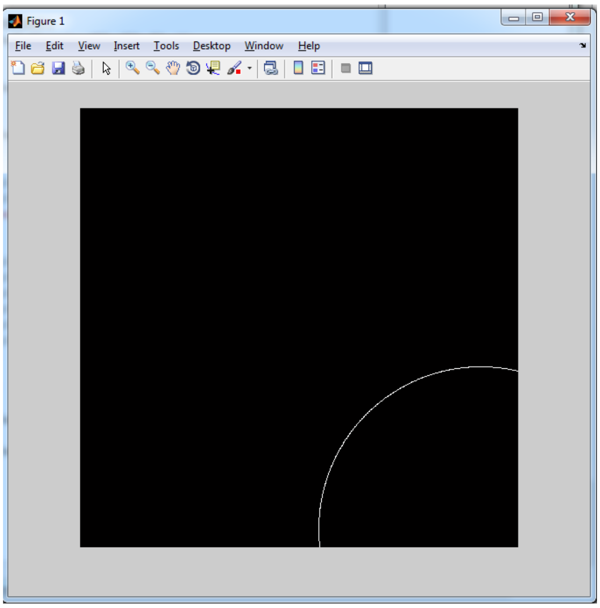
```
>> circle2 = draw_circle(Ex2Example, 1,1, 85, 1);
>> imshow(circle2)
```

and

```
>> circle3 = draw_circle(Ex2Example, 480,458, 185, 1);
>> imshow(circle3)
```

Display the following two results:

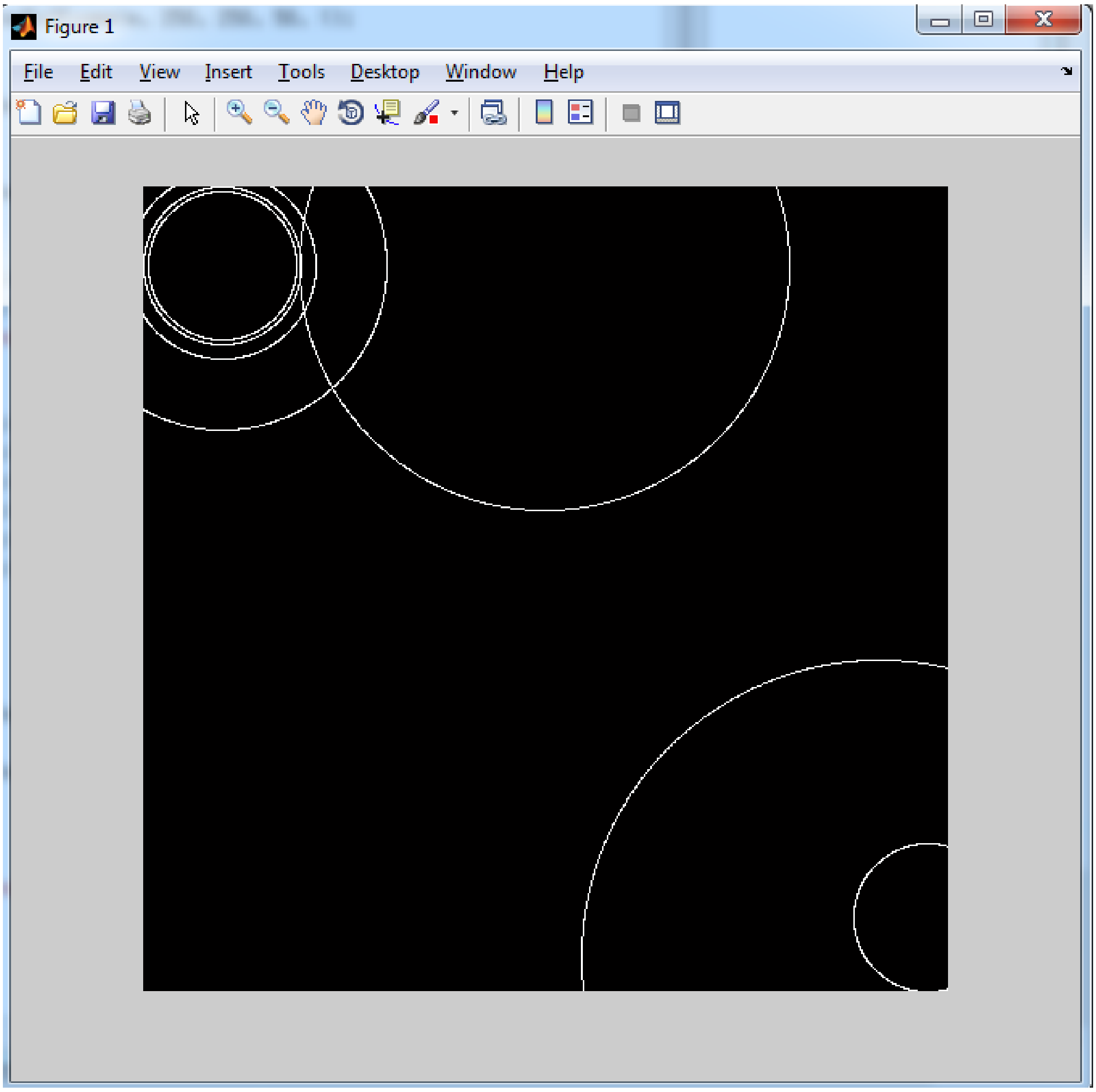




Now to check if overlapping and everything else works properly, I made a collection of circles on an image by inputting:

```
>> circle4Coll = draw_circle(Ex2Example, 480,458, 185, 1);
>> circle4Coll = draw_circle(circle4Coll, 455,488, 46, 1);
>> circle4Coll = draw_circle(circle4Coll, 50,50, 46, 1);
>> circle4Coll = draw_circle(circle4Coll, 50,50, 49, 1);
>> circle4Coll = draw_circle(circle4Coll, 50,50, 58, 1);
>> circle4Coll = draw_circle(circle4Coll, 50,50, 102, 1);
>> circle4Coll = draw_circle(circle4Coll, 50,250, 152, 1);
>> imshow(circle4Coll)
```

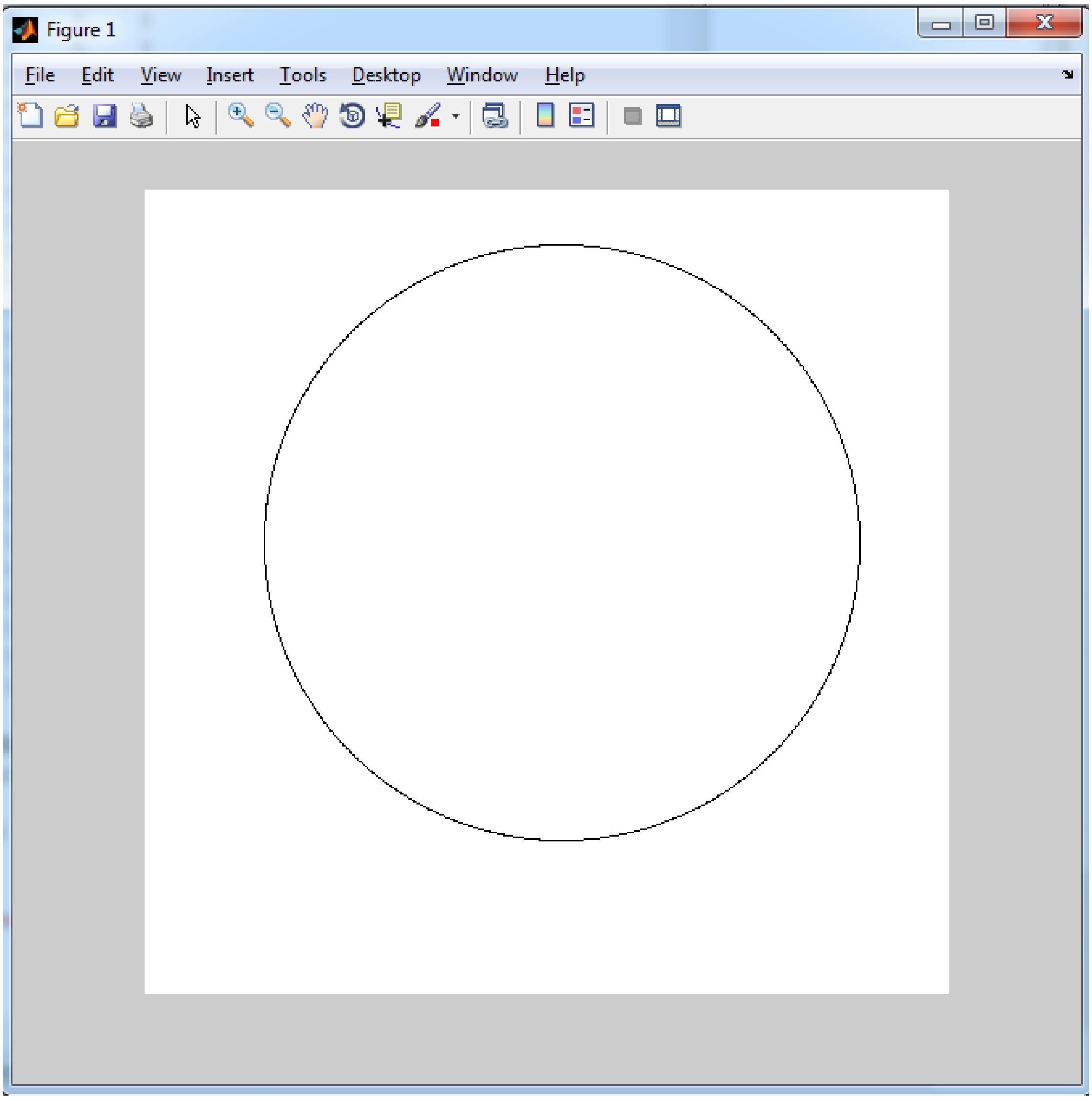
The output was as follows:



From the examples above we can clearly see that circles can be drawn given different radiuses, going out of bounds, et cetera.
As a final example, I will show you how value is implemented, by putting black circles on a white background (using zeros on a background of ones).

```
>> Ex2Example2 = ones(500,500);
>> circleZeros = draw_circle(Ex2Example2, 220,260, 185, 0);
```

The output was as follows:



Note: Values other than 1 and 0 can also be used. These are examples.

The code used for draw_circle.m:

Start of function

```
function [out_image] = draw_circle(in_image,x0,y0,r,value)
```

Determining useful variables

```
angle = 0:0.005:2*pi;             % Determining the sharpness of the angle with a
vector
circleHalfOne = r*cos(angle);     % Taking "half one" (not the true half ofcourse)
using the cosine function
circleHalfTwo = r*sin(angle);     % Taking "half two" using the sine function
[a,b] = size(in_image);           % Determining the size, mainly for the loop
angLength = length(angle);        % Using the length of the angle, also mainly for the
loop
fullcircle = [];                  % Pre-making fullcircle
```

Determining the actual full circle "coordinates"

```
for i = 1 : angLength
    x = circleHalfOne(1, i);        % Taking what's in position (1,i) of circleHalfOne
    y = circleHalfTwo(1, i);        % Taking what's in position (1,i) of circleHalfTwo
    fullcircle = [fullcircle;x,y];  % Merging it into a two-dimensional array
end
```

Inserting the circle into the image

```
for i = 1 : angLength
    try % A try catch is used for any negative out of bounds
```

RealX and RealY create the position

```
    realX = round(fullcircle(i,1)) + x0;    % Rounding helps prevent boundary being a
non-integer error
    realY = round(fullcircle(i,2)) + y0;    % Rounding helps prevent boundary being a
non-integer error
```

preventing "out of bounds" Positions to turn into huge black blocks

```
        if realY > b            % If it's bigger than the full length of the imgae
            continue            % Skip this loop iteration
        elseif  realX > a       % If it's bigger than the full width of the image
            continue            % Skip this loop iteration
        end
```

Putting the value into the actual image coordinates

```
in_image(realX,realY) = value;
```

```
    catch
    %Negative out of bounds are caught by matlab, so a try catch without error message
suffices
    %to suppress any possible and redundant error output
    end

end
```

Finally giving out the result for the function to output

```
out_image = in_image;
```

```
end
```

Exercise 3 – Detection of Circles

Exercise 3 – Detection of circles

Once you have an image containing a set of edge points, you now need to create a function to identify the possible centres of any circles with a certain radius.

Exercise 3 is to create a function with the form below that creates a new accumulator matrix containing the votes for the possible centres of the circles.

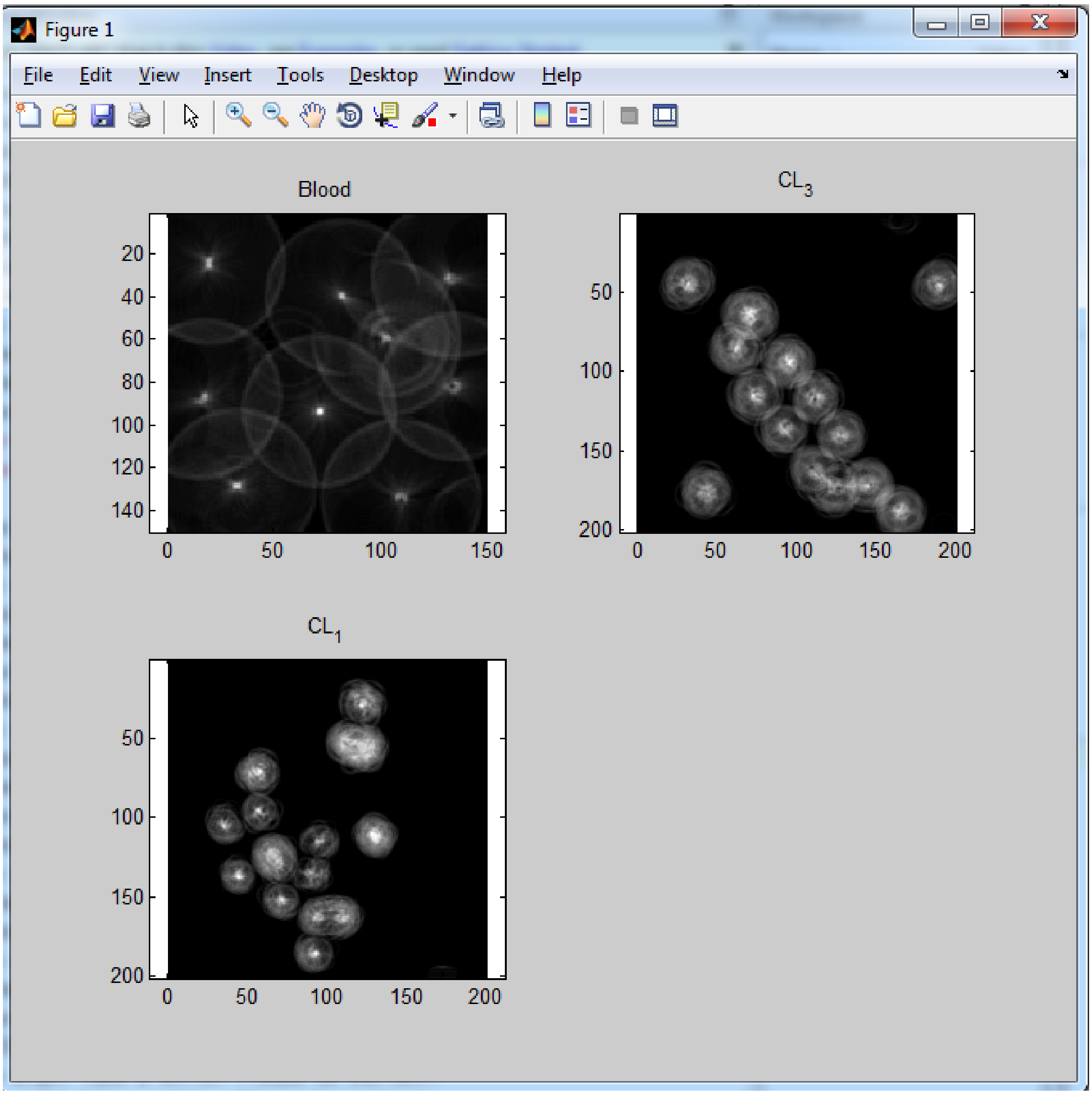
function [accumulator]=Hough_Transform(edge_image, radius)

edge_image is a matrix where each pixel has a value of 1 or 0 depending on whether it is an edge point or not (1=true), whilst radius is the radius of the circle whose centre is being sought.

Note that imagesc is used here to express the "density" of overlapping circles, rather than imshow.
The following script was ran:
Script file name: EXSCRIPTex3.m

```
%% Loading in all necessary values
load CL_1.mat
load CL_3.mat
load blood.mat
edgeCL_1 = edge_detect(CL_1,0.5);
edgeCL_3 = edge_detect(CL_3,0.5);
edgeBlood = edge_detect(blood,0.5);
%% Blood image Hough Transform
houghBlood = Hough_Transform(edgeBlood,18);
subplot(2,2,1)
imagesc(houghBlood);
axis('equal');
title('Blood');
%% CL_3 Hough Transform
houghCL_3 = Hough_Transform(edgeCL_3, 8);
subplot(2,2,2)
imagesc(houghCL_3)
axis('equal');
title('CL_3');
%% CL_1 Hough Transform
subplot(2,2,3)
houghCL_1 = Hough_Transform(edgeCL_1, 5);
imagesc(houghCL_1)
axis('equal');
title('CL_1');
```

The following images were output:



The following code was used:

Start of Hough Transform function

```
function [accumulator]=Hough_Transform(edge_image, radius)
```

```
[x,y] = size(edge_image);       % finding x and y, for pre-allocating and the loops
addition = zeros(x,y);          % Preallocating the array with zeros additon for speed

disp('This might take a while. Please be patient.'); % Hough_Transform takes around 4
seconds, so this is displayed to be user-friendly
```

Finding circles and inserting votes

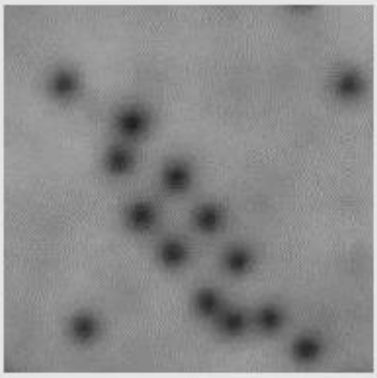
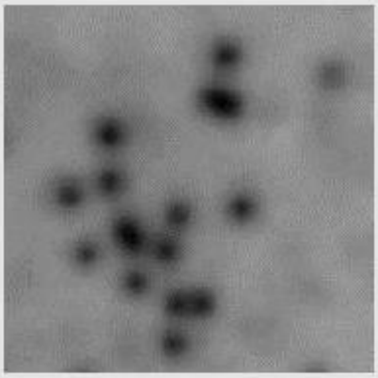
```
for i = 1:x
      for j = 1:y
        if edge_image(i,j) == 1         %If the part in the edge_image is one (white),
so an edge is found
            new = zeros(x,y);           %Create a new array of zeros, every loop
iteration
            new = draw_circle(new,i,j,radius,1); % Drawing a circle of the given radius
at the point where the edge is found
            addition =  addition + new; % Putting together the new array of zeros with
the one in the previous loop(s)

        end
      end
  end

    accumulator = addition;             % Showing the function the addition is what is
meant to be output
```

```
end
```

Exercise 4 – Counting Dislocations (part 1)

Exercise 4 – Counting Dislocations

The final exercise is to write a MATLAB script or function to automatically find and count the dark spots from a cathodoluminescence image of a semiconductor surface. Two such images, with which you will be provided, are shown below.

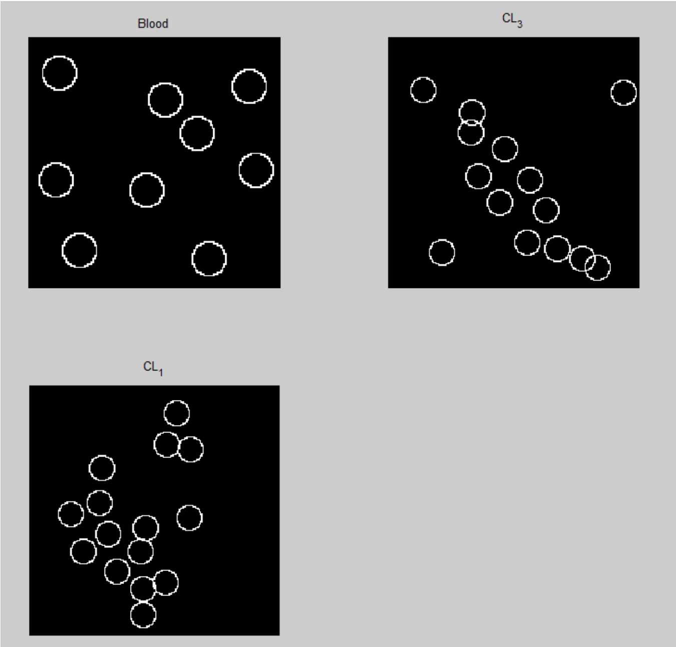
The script/function should automatically find the best-fit circle for every dark spot, superimpose these circles on the original image and count the number of circles found.

For this exercise, I wrote two functions. One simply counts the dislocations and shows an image of the dislocation circles of a given Hough Transform, while the other image merges all of the code used before, meaning it edge detects, applies the hough transformation, then counts and shows the dislocations.

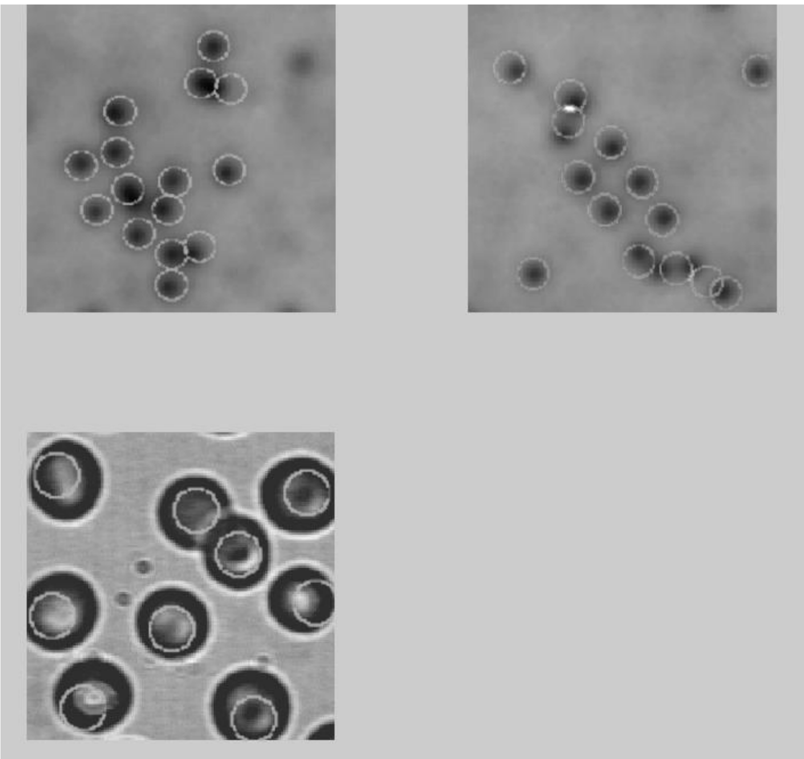
I will be using the “collective” function, (called DIslocationTotal), because this integrates the function CountDislocations into it, so seperately showing it would be redundant.
The following variables are meant to be input:
(input_image, edge_treshold, hough_radius, dislocation_treshold, filterRadius)
The following script was used:

```
%% Loading in all necessary values
load CL_1.mat
load CL_3.mat
load blood.mat
%% Blood image Dislocations
subplot(2,2,1)
bloodDislocations = DislocationTotal(blood,0.5,18,0.4,18);
title('Blood');
%% CL_3 Dislocations
subplot(2,2,2)
CL_3Dislocations = DislocationTotal(CL_3, 0.5,8,0.6,13);
title('CL_3');
%% CL_1 Dislocations
subplot(2,2,3)
CL_1Dislocations = DislocationTotal(CL_1, 0.5,5,0.6,18);
title('CL_1');
```

Which gave as output:



CL_1Dislocations = 15
CL_3Dislocations = 14
bloodDislocations = 9
Which of course does not look very clear, so I superimposed them on to the original images.



Which shows a relatively precise circle count and position.

The code used (CounDislocations) is as follows:

Start of function

```
function [ amountOfDislocations ] = CountDislocations( input_image, treshold,
filterRadius )
```

Loading in necessary variables

```
tTreshold = max(max(input_image)) * treshold; % The treshold is a combination of the
maximum varibale found times the treshold (between 1 and 0)
[x, y] = size(input_image);                   % Determining the size for the loop
newImage = zeros(x,y);                        % Part of the to-display image
totalDots = 0;                                % Allocating the variable "totalDots" to
show the amount of dislocations
```

Finding and showing the amount of dislocations

```
for i = 1:x
      for j = 1:y
          if input_image(i,j) > tTreshold % If the coordinate is over the threshold
              for k = 1:filterRadius      % Within a certain radius, everything is
deleted
                                          % To prevent detecting the same
                                          % circle several times.
                  input_image = draw_circle(input_image,i,j,k,0);
              end
            newImage = draw_circle(newImage,i,j,10,1); % Drawing a circle where a
dislocation is found
            totalDots = totalDots +1;                  % Adding one to the total
dislocation counter
          end
      end
end
```

Output

```
imshow(newImage);                 % Showing where the dislocations are on an image
amountOfDislocations = totalDots; % Outputting the total dislocation amount
```

```
end
```

The code used for DislocationTotal is a lot simpler, but it is worth noting because it collects everything as a whole.

Showing a total of dislocations in an image, and displaying them

```
function [ TotalDots ] = DislocationTotal( input_image, edge_treshold, hough_radius,
dislocation_treshold, filterRadius  )
  Y=edge_detect(input_image, edge_treshold);        % Edge detection
  A=Hough_Transform(Y, hough_radius);               % Hough trasnform
  TotalDots = CountDislocations( A, dislocation_treshold, filterRadius); % Counting
dislocations and displaying them
end
```

Exercise 4 (continued)

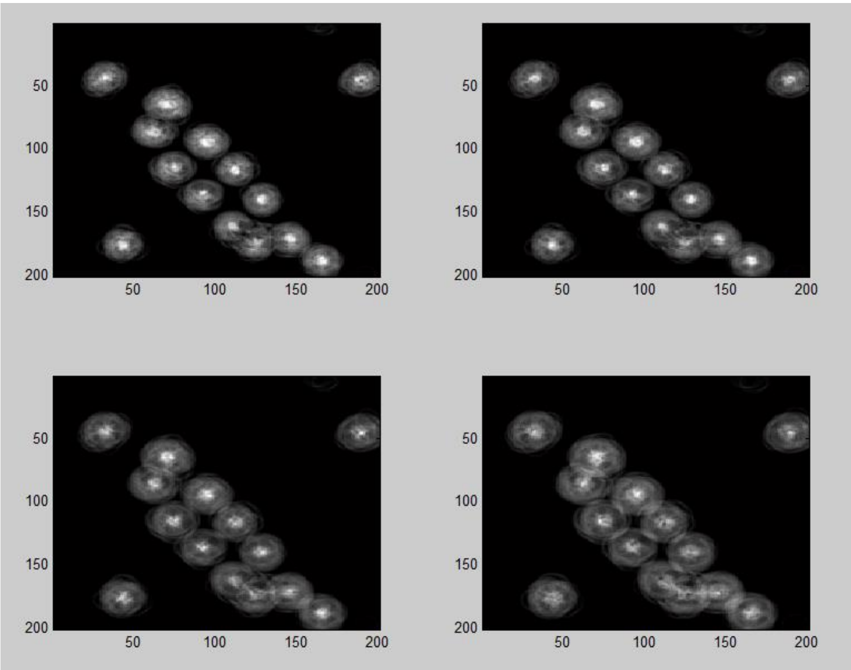
Finally, as the radius of the defects varies slightly between individual cells, a better performance can be achieved if a range of radii is used instead on one fixed value. Extend the files you have written to detect circles over a range of values, e.g. between r_min and r_max, the minimum and maximum radii of the circles you wish to find.

The effect of this is to extend the accumulator space from two to three dimensions where r_min to r_max is the third dimension. Any accumulator space filtering will now need to be done in three dimensions instead of two. As before, superimpose the circles found on the original image and produce a count for the number found.

The following was input:

```
A = Exc4continued(CLEdge, 5, 9);
subplot(2,2,1)
imagesc(A(:,:,1))
subplot(2,2,2)
imagesc(A(:,:,2))
subplot(2,2,3)
imagesc(A(:,:,3))
subplot(2,2,4)
imagesc(A(:,:,4))
```

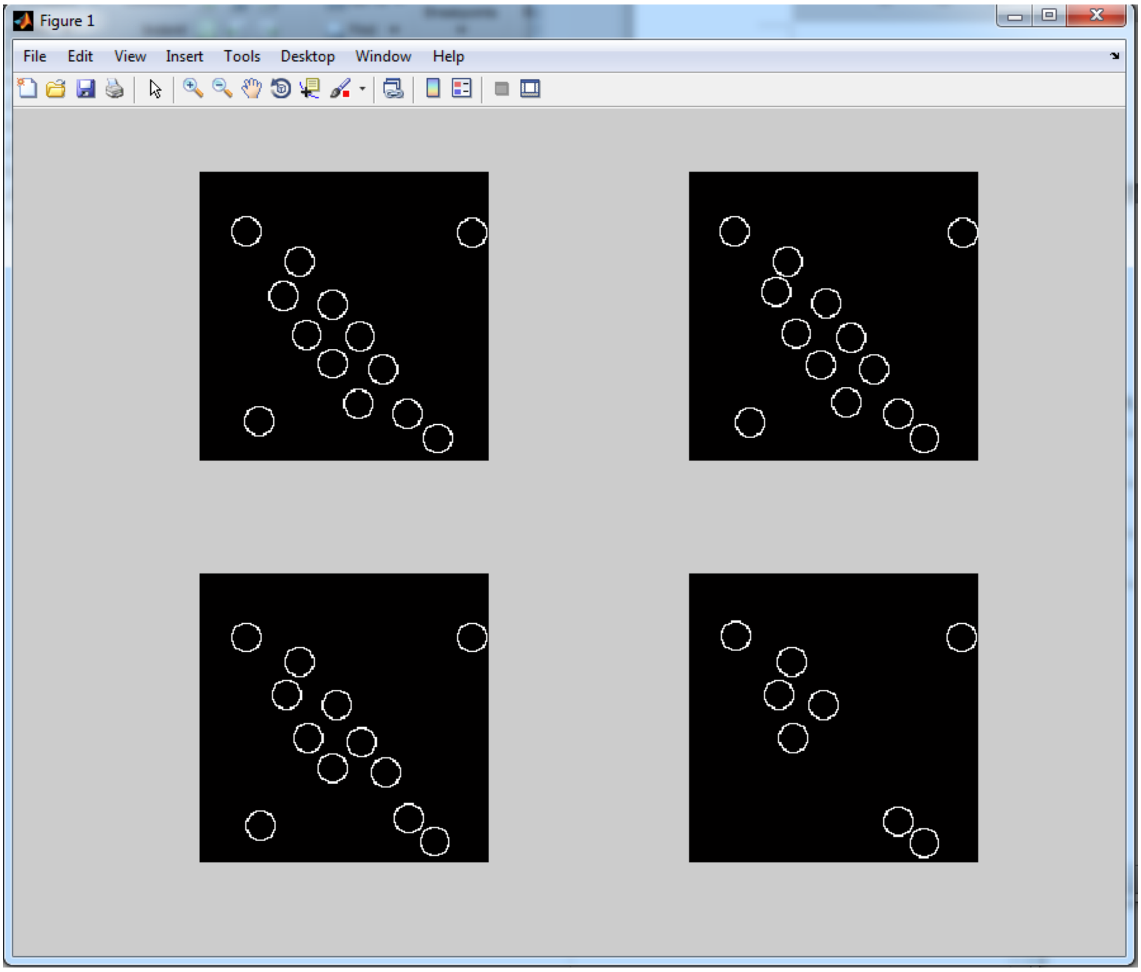
These 4 images were output:



Then CountDislocations was applied to it.

```
A1 = CountDislocations(A(:,:,1), 0.8,12);
subplot(2,2,2)
A2 = CountDislocations(A(:,:,2), 0.8,12);
subplot(2,2,3)
A3 = CountDislocations(A(:,:,3), 0.8,12);
subplot(2,2,4)
A4 = CountDislocations(A(:,:,4), 0.8,12);
```

Of which the results were:



A1	13
A2	13
A3	12
A4	8

Which clearly shows how changes in range can allow you to more accurately measure, or measure over a wider range, rather than going through it one by one.

The code behind this:

File used: Exc4continued.m

Start of function

```
function [ MDOutput ] = Exc4continued( input_image, r_min, r_max )
```

Loading in necessary variables

```
z = r_max - r_min;          % Finding the size of the loop using the max r to be found -
the min
[x,y] = size(input_image);  % finding the size of the input image to pre-allocate
MDOutput = zeros(x,y,z);    %MDOutput means MultipleDimensionOutput, this is a
preallocation for speed
radius = r_min;
```

Hough transformation in 3D

```
    for z = 1:z
    MDOutput(:,:,z) = Hough_Transform(input_image,radius); % Applying the Hough
Transformation for the page z is at
    radius = radius + 1;    % Adding 1 to radius every loop iteration
    end

end
```