UNIVERSITY OF
LINCOLN

Image Processing

CMP3108M

Liam T. Berridge BER14475946

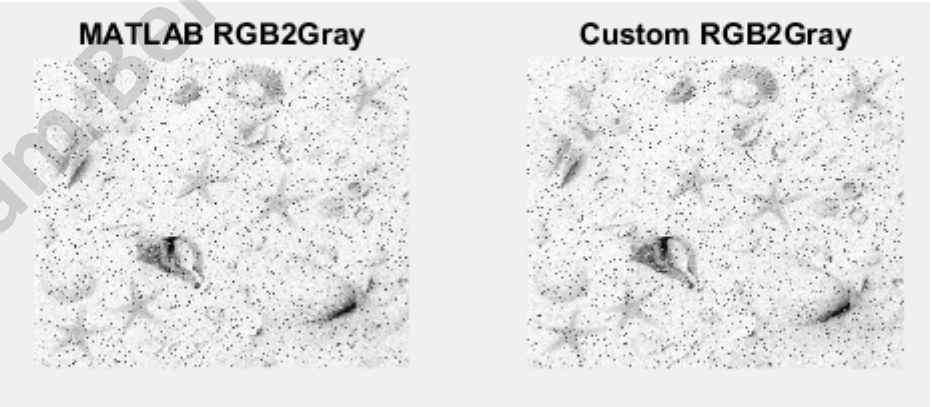
Step 1: Image Loading

For this step, the image is loaded into the program by taking every value from the image file and placing it into a matrix matching the image dimensions. For colour images, RGB values are stored by creating a 3-D matrix, with a separate matrix for each colour value. The MATLAB command imread(); allows for easy importing of images however so for the sake of simplicity it has been used in the submitted code. Below is the loaded image.

Step 2: Image Conversion

The second step was to convert the image to grayscale. Grayscale is defined as “Each pixel is a shade of grey, normally from 0 (black) to 255 (white). This range means that each pixel can be represented by eight bits, or exactly one byte”. (McAndrew, 8). This also means compressing the three-dimensional RGB matrix, to a two-dimensional one; to do so you must get the luminance value of each pixel. This can be done by multiplying each of the RGB values for a pixel to the corresponding luminance value, and then adding all resultant values together. Luminance values in this case were sourced from Rec. 601, a standard issued by the CCIR (Recommendation ITU-R BT.601-7, 2016). The formula used is as follows. $G = 0.299 * R + 0.587 * G + 0.114 * B$.

Below are both images and a sample of the code used.




```
grayscaleIm2 = 0.299*inputImage(:,:,1)+0.587*inputImage(:,:,2)+0.114*inputImage(:,:,3);
```

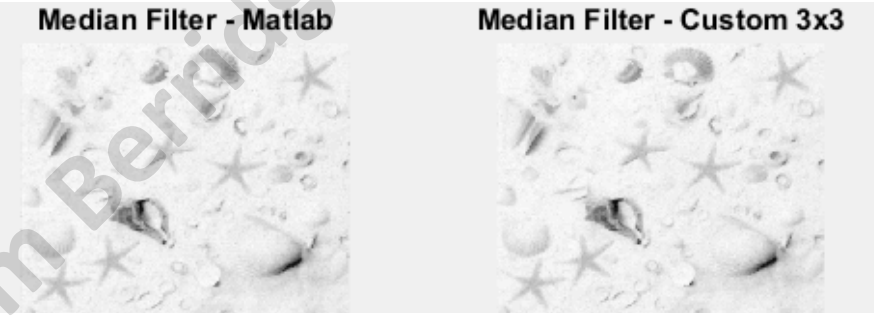
Step 3: Noise Identification and Removal

The third step was to reduce and potentially remove noise present in the image if possible. McAndrew explains this concept as “The idea is to move a “mask”: a rectangle (usually with sides of odd length) or other shape over the given image. As we do this, we create a new image whose pixels have grey values calculated from the grey values under the mask” (McAndrew, 75). To do this, we have three potential methods. The first is Mean filtering, which takes the mean value of the surrounding 3x3 area and sets the middle values as this. It produces inaccuracies in images as it can produce values that are considered outliers to the local area’s values. This is most noticeable around edges or sharp changes in either contrast or colour.

Median filtering is the second, which avoids this issue by sorting the values from the surrounding 3x3, and then overwriting the central value over the current pixel. This means the central pixel will always match another pixel in the local area and not become an anomalous value.

The third and final method is adaptive filtering, which considers local variance, smoothing values more where they are closer, and less so when values are widely varied in the local area. This acts as a form of edge detection and avoidance, making sure edges stay pronounced within the image.

For this step, I used median filtering, which I have written my own function for to compare against MATLAB’s own. The filter works by taking local variables from the given surrounding area, and then placing them into a 1 dimensional array and sorting them. It then reshapes the values to the original area and takes the central value to place back into the image. Below are comparison images as well as the code used. I have also taken a sample of image values side by side, to show how accurate my result was to MATLAB’s implementation. The values are identical, except for shifting to the left. This is due to not padding my implementation.



```
[r,c] = size(grayscaleIm);
blank = zeros(r,c);
cleanIm = grayscaleIm;

% The function loops over every column, then every row, until it has
% covered every pixel in the image.
for row = 2:(r - 1)
    for col = 2:(c - 1)
        % Write the pixels local area to the grid, and sort it ascendingly.
        filterGrid = grayscaleIm(row-1:row+1,col-1:col+1);
        sortArray = sort(reshape(filterGrid,1,[]));
        % Copy the sorted grids central value over to a new image.
        for a = row-1:row+1
            for b = col-1:col+1
                cleanIm(a,b) = sortArray(5);
            end
        end
    end
```

362x438 uint8

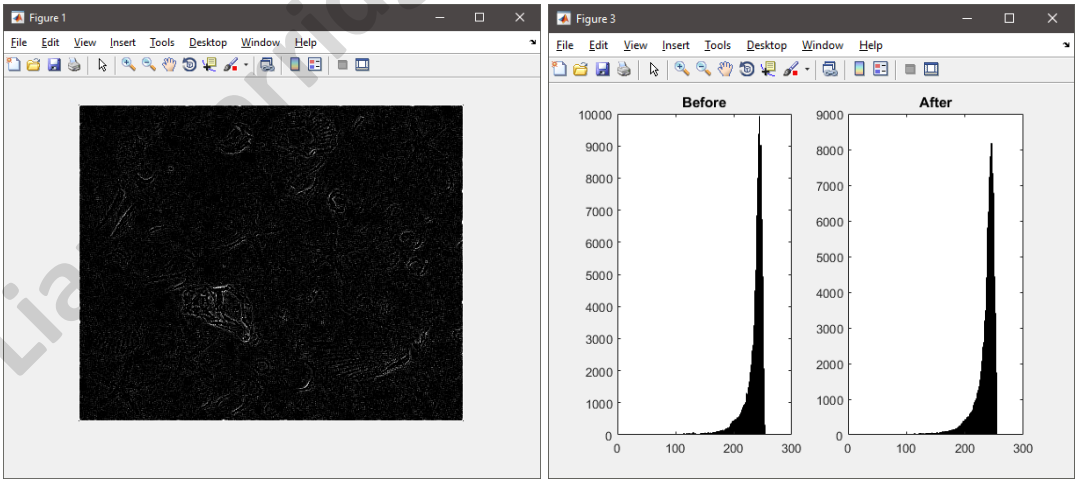
	1	2	3	4	5	6	7
1	247	244	238	238	243	243	
2	249	250	250	250	248	249	
3	249	250	251	250	250	249	

362x438 uint8

	1	2	3	4	5
1	0	247	244	238	238
2	244	249	250	250	250
3	238	249	250	251	250

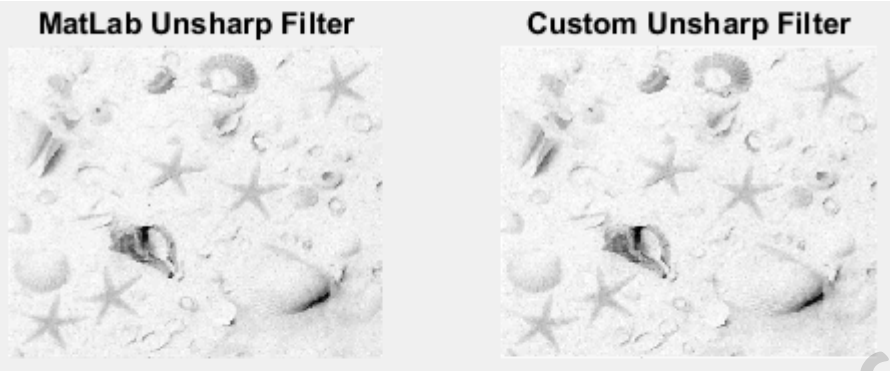
Step 4: Image Enhancement

The fourth step was image enhancement, and given that we had already smoothed the image to erase the noise, it seemed apt to attempt restoration of the images finer detail. To do this we apply image sharpening. As explained by McAndrew, there are various methods for doing so, however for this step unsharp masking was used. McAndrew addresses this as the following process “The idea of unsharp masking is to subtract a scaled “unsharp” version of the image from the original.” (McAndrew, 122). This process blurs the image with a mean neighbour filter, and then if we subtract the resulting blurred image from the original, it produces an edge map. An edge map shows areas of high-contrast and allows us to then define which areas show significant detail. Below is the resultant edge map, it has been amplified for visibility.



If we then add this image to the original, we get more pronounced edges or high-contrast areas. This allows us to capture shapes in the image that may have been affected by noise. Below are comparison images of my sharpening versus that implemented by MATLAB. The code used is also supplied. A histogram has also been supplied above, as it shows the effect sharpening has on the images luminance distribution. The image overall lowers in intensity,

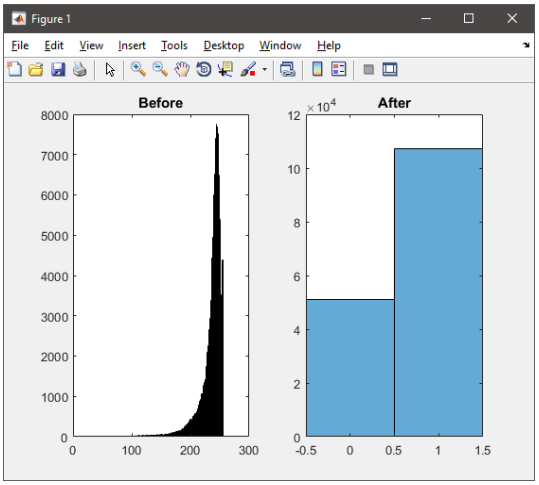
however the shape produced becomes more pronounced, this is most noticeable at the lower end where a small curve becomes a noticeable bump in the graph.




```
% My own implementation of an unsharp mask.
% First we smooth the image
sum = 0;

for row = 2:(r - 1)
    for col = 2:(c - 1)
        % We add together the values of a 3x3 grid around the current pixel
        for a = row-1:row+1
            for b = col-1:col+1
                sum = sum + doubleIm(a,b);
            end
        end
        % Here we divide the sum of all local pixels by 9, to get the mean.
        sum = sum/9;
        % We then set the corresponding pixel in the output image to the
        % mean.
        smoothIm(row,col) = sum;
        sum = 0;
    end
end

% We then subtract the blurred image from the original to produce an edge
% image. The multiplier is added to increase the edge significance.
edgeIm =(doubleIm*10 - smoothIm*10);

% To produce a sharper image, we add the edge image to the original.
sharpIm2 = doubleIm + edgeIm;
```

Step 5: Image Segmentation

For the fifth step, we need to split the image into foreground and background, and to do this we can threshold the image. This is the process of splitting the image based on a given threshold value. Anything either side of the value is then forced to an extreme set value. Usually 0 and 1 for images being converted to binary. Otsu's method is implemented by MATLAB to give a better output, which also includes the histogram weighting instead of a direct split based on the mean. To better demonstrate this concept, I have included two histograms below, showing before and after thresholding. The values have been split to 0 and 1, however we see that due to a higher weighting to lighter values, a larger proportion of the image is forced to 1 (White). The following formula shows the most basic level of thresholding, where T is the given threshold (McAndrew, 67).



$$\text{A pixel becomes } \begin{cases} \text{white if its grey level is } > T, \\ \text{black if its grey level is } \leq T. \end{cases}$$

Due to time constraints, I was not able to implement my own version of Otsu's method, however the concept is as follows. The weight, variance and mean is calculated for each threshold value. The lower half becomes the background, and the upper half the foreground (As luminance is used to determine foreground objects). We then calculate the in-class variance by multiplying the two variances against the associated weightings. Finally, we get the sum of the two produced in-class variances. For each threshold, this is measured against the last thresholds in-class variance, and the lowest value is used as the threshold for the image.

Step 6: Morphological Processing

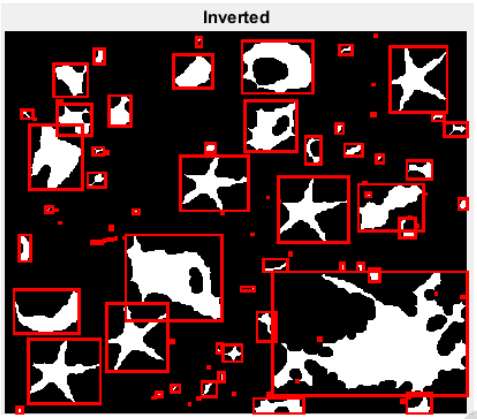
The sixth step is Morphological processing, and is defined by as the following "Morphology, or mathematical morphology is a branch of image processing which is particularly useful for analysing shapes in images" (McAndrew, 195). We use it in this stage to minimise anomalies and assist us in step 7, where we then detect and display shapes as desired. The two operations used are dilation and erosion. They have the purpose of growing shapes (dilation) or shrinking them (erosion); this can be used to shrink and remove noise, and then regrown to the objects initial dimensions with varying degrees of accuracy. Below are images of this process in effect, as well as a full explanation of the actions that occur.

First the image is closed, where the image is dilated then eroded, and then when the image is opened the opposite takes place, with erosion and dilation. Finally, we dilate the image twice more in the third image to remove as many artefacts as possible and give us only the larger shapes. Erosion and Dilation have the opposite effect on objects than the processes

intended purpose, this is due objects being a background colour, and the background space being a foreground colour. This could be corrected by inverting the image before processing, however it would only be a cosmetic change to the program, not a functional one regarding the output. The processes take place by placing a mask over the image, and then checking where the perimeter is and extending it outward or retreating it inwards towards the object.

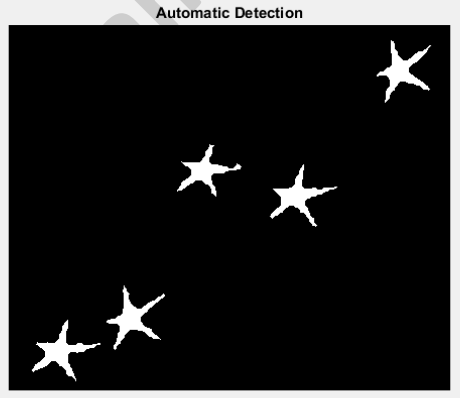
Step 7: Automated Recognition

The seventh and final step was to automatically recognised shapes and display them. To do this we require a few key pieces of information. Firstly, we need to know how many objects are in the image. Using the bwlabel command, which finds all four or eight connected shapes, we can create a list of objects. Following this we then get the area and perimeter for all the objects in the list and store that information for use. To demonstrate this I have included an image which displays the bounding box for all detected objects. The bounding box is simply placed at the edge of the object to show its boundaries. The code for this is also displayed.




```
% Then we get the statistics for all objects present in the labelled image.
objects = regionprops(label,'BoundingBox','Area','Perimeter');

% The below code then shows each of the labelled objects by drawing a
% rectangle around them based on a bounding box. However the above needs to
% be changed from 'centroid' to 'BoundingBox'.
for list = 1:objCount
   drawBox = objects(list).BoundingBox;
   rectangle('Position', [drawBox(1),drawBox(2),drawBox(3),drawBox(4)],...
   'EdgeColor','r','LineWidth',2 )
end
```

Then finally we use the circularity formula from shape factor theory to determine the roundness of the objects. By specifying the range, we want based on the first star, we can then find the other stars in the image and output them to a separate image using ismember, which can move array members from one set array to another (Uk.mathworks.com, 2016). The formula for circularity is shown in the code sample below, as well as the output image where all stars have been captured and moved to a separate image; most stars have retained all points, with some having slightly shortened points. This is due largely to the noise present in the original image.




```
starIm = blank;

% Using the statistics, we can search through and determine which values we
% want to use. We can do this by determining the roundness of objects and
% then keeping those that match the chosen object.
for count = 1:objCount
   metric = 4*pi*objects(count).Area/objects(count).Perimeter^2;
   if metric < 0.23 && metric > 0.16
       starIm = starIm + ismember(label,count);
   end
end
```

Bibliography

McAndrew, Alasdair. *An Introduction to Digital Image Processing with MATLAB*. 1st ed. Victoria University of Technology, 2016. Web.

Uk.mathworks.com. (2016). *Array elements that are members of set array - MATLAB ismember*. [Online] Available at: https://uk.mathworks.com/help/matlab/ref/ismember.html [Accessed 8 Dec. 2016].

Recommendation ITU-R BT.601-7. (2016). 1st ed. [ebook] Geneva: The ITU Radiocommunication Assembly, p.2. Available at: https://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.601-7-201103-I!!PDF-E.pdf [Accessed 8 Dec. 2016].