Image Processing

Nicholas Venis
VEN15565251

November 2017

Task 1

Task one highlighted the differences and challenges of two different interpolation methods, bilinear and nearest neighbour. Interpolation is the process of constructing new data based on previous, already known data. In this case interpolation is applied by using old pixel values from a smaller image and constructing a new larger image based on the aforementioned pixel values; in image processing this process is called resampling.

Before any processing can take place, the image must be converted from RGB to greyscale. RGB images are stored as 3 matrixes with red green and blue values respectively. Converting to greyscale from RGB cuts the number of matrixes down to one, this allows for easier transformations of images.

In the brief, the new image was defined as being 1668x1836 pixels large with the original image being 556x612. The image size of the original image is found using the size() function and stored in two variables: x and y. By dividing the new image's x and y values by the original image's we can find the scaling factor needed for resampling. In this case 1668/556 & 1836/612 are both equal to three, therefore we have to scale the original image by a factor of three.

We then create a new image to serve as the output image to be displayed by creating an unsigned 8-bit integer matrix with the same dimensions as the new image detailed in the brief. Unsigned 8-bit integers are used as we cannot have a pixel value above 255 and therefore would not need a type that supports longer numbers. This is then populated with 0 values to fill the matrix with pixel values to change at a later point.

Nearest neighbour interpolation is useful when you want to preserve the data from the input as no pixel values are actually changed or created. This therefore means that the same values are present in the output image only spread out further.

We use a nested for loop to loop through every pixel in the image in order to change its value accordingly. The two loops conditions to be met are the x and y values of the output image, therefore when these are reached the loop ends. The loop increments by one each cycle and is initialised at 1. Indexes start at 0 usually however images start at 1 therefore the loop has to reflect this to avoid errors.

The value of the new pixel is then calculated as follows. The variable newx stores the x position of the pixel value to be used in the new image. This x positon corresponds to a pixel from the old image. The next variable newy stores the y position of the of the pixel value to be used in the new image, also from the old image.

To calculate the value of newx we first take the current index of the x loop. We use this as this is equal to the position of the x value of the new image. This value, for instance 5 is then divided by the scaling factor (3). However, this can result in decimal values such as in this example (our result is 1.66 recurring). Pixels values cannot support decimals, only whole integers, for this reason we round. We use the function floor() to round towards negative infinity (down) preventing the index from increasing beyond the size of the matrix. newx

now represents the x value of the pixel value from the original image to be used at the current x index in the new image.

This exact process is then repeated for the y axis with the second loop index being used as the y value to be divided by scale. This gives us the output newy representing the y value of the pixel value from the original image to be used at the current y index in the new image.

The new image pixel value at current position (index) is then said to be equal to the pixel value of the original image at the coordinates calculated earlier and stored in newx and newy. The loop iteration then ends and newx and newy are recalculated at the current index values. However, there is a problem; if the value of newx or newy is equal to 0 the loop will break. This is due to the previously mentioned fact images start at 1,1. To combat this if statements check the values of newx and new, if found true each are set to one so the index is a real positive integer and the loop can continue.

Figure 1: input image after Nearest Neighbour interpolation.

The next interpolation method is bilinear interpolation. Bilinear interpolation is extension of linear interpolation whereby instead of creating data based off a straight line in one direction we then repeat the process in another direction. As we are working with a 2d matrix (image) we therefore should complete the process for one axis, then another.

First we need to create an output image like before that is the exact dimensions of the brief, then fill with 0 values. However, in this interpolation method we first need the original data to interpolate with. To accomplish this we scale the original image so that it fits into the new image equally. This is achieved by creating a nested for loop that iterates for every pixel in the original image, end condition being the respective x and y sizes of the original image. to

ensure equal spacing and correct resizing of the image the indexes of the loop representing the x and y values of the original image are multiplied by the scale factor to get the indexes (x and y values) of the new, larger image. The pixel value is then copied across to the new image.

The next set of loops are where the initial interpolation takes place. These loops however are slightly different. The end condition of these loops are the x and y sizes of the new image minus the scale factor (3). This is due to the fact that one of the points of reference used is the current pixel value at current index (n,m) PLUS the scale factor to find the next data value from the original image. if the loop was not created this way on the last iteration of the loop the index of the point of reference would jump beyond the size of the matrix. Another irregularity with this loop is the fact instead of iterating by one it iterates by the scale factor (xscale and yscale), this is due to how the original data is spread in the new image and therefore maintains that the references used later in the loop always land on a value containing original data.

The loop starts its index at 0 however as images cannot start at 0 two if statements are used to check if the index is 0 and if so set it to 1.
The next set of variables are used to hold pixel values at specific points. Pointa holds the pixel value at current index (x,y) and will always be from the original image. Pointb holds the pixel value at the current x index however the y index is added to the scaling factor to hold the pixel value of the next point of original data along the y axis.

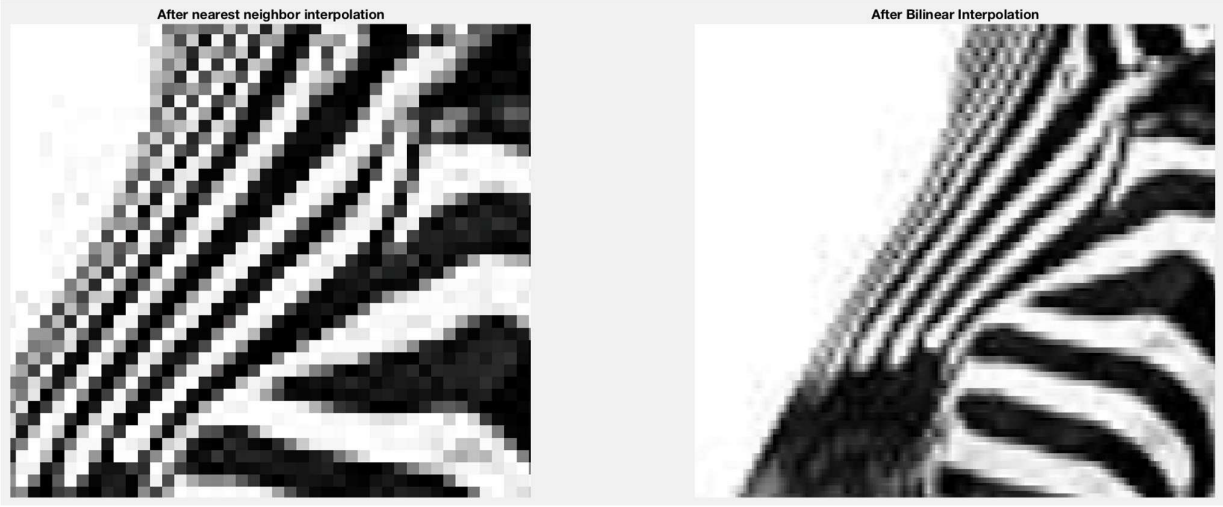
Using these two points we can interpolate the pixels in-between missing values. To do this we run another loop (k) incrementing by one and ending at the end of the scaling factor minus one. We subtract one as there are two missing values, not three (scaling factor). AB is variable used to store the calculated pixel value of the current index. AB is calculated using two variables, a and b. A is calculated as the pixel value at pointa (first original value) multiplied by the relative distance to the second original value (pointb) divided by the scaling factor. B is calculated as the value at the second original pixel value (pointb) multiplied by the relative distance to the first original pixel value divided by the scaling factor. A is then added to B and the result is rounded as pixel values can’t be decimals. The pixel value is added to the new image at the current x index and current y index with the index of the loop k. The image is then populated with columns of interpolated data.

Finally, we run a near identical loop to the last however instead of incrementing by the scaling factor we increment by one so that the loop runs for every row of the image (y) to create a completed image. instead of pointb being once increment down the x axis instead it is one increment down the y axis. This creates a completed image shown below.

Figure 2: Image after bilinear interpolation.

However, the two interpolation methods create some discrepancies between images. Each has its own respective uses and drawbacks. Nearest neighbour interpolation is extremely useful if you need a fast algorithm or want you want to preserve the original dataset, no pixel values are changed and so no data is changed. A real-world use of this type of interpolation is pixel art. Artists want to preserve the art and scale regardless of display or resolution and so resample via NN. However, the main drawbacks of nearest neighbour are that images become badly pixelated (distorted) and are prone to edge halos (Titus and Geroge, 2013).

Bilinear interpolation is a more complex process and so is both more demanding resource wise and slower. This method has properties similar to low pass filtering therefore meaning high frequency components appear faded and image outlines appear blurry or faded and images are prone to interpolation artefacts such as blurring (Dianyuan Han, 2013).



Task 2

Task 2 focused on grey scale slicing. For this transformation, we need to preserve the integrity of the original image in two ranges and shift a third range to be one value. For all pixel values from 0 to 80 we want to retain the values from the original image, however for all values from 80 to 100 we want to shift any of these values that fit the criteria to equal 220. This in turn highlights any pixels in that range as a near to bright white. We declare 80 as variable a and 100 as variable b.

To do this we must first establish a new matrix the size of the original image to serve as the output. We create a integer matrix the exact dimensions of the original image and then populate with 0's. To find these dimensions we use the size() function to develop two variables; ux and uy. We then declare the data value we wanted the pixels that fit the criteria to (220) as c and declare the range of greys in use as 255.

We then run the original image through 2 nested for loops to loop through every pixel value on the original image running for the length of the x and y axis (ux and uy). We use an if else block for the actual filtering of pixel values. The if statement uses the condition that if the pixel value is equal to or greater than 80 (a) AND equal to or less than 100 (b) it should be changed. We then change this value to 220 at the current index (positon) of the new image.

We use an else block to handle any other data not in this range. This therefore keeps the integrity of the original image and only changes a fairly small, noisy region. The output image is then divided by the amount of greys in a greyscale image to then be parsed to an unsigned 8-bit image matrix. For each transformation, a histogram is included to highlight the pixel values in a region being changed. As the noisy regions contain a compacted level of contrast if there was detail we wanted highlight further we could use contrast stretching to stretch out the values so they were scaled from 1-255, this further transformation would be useful to retrieve hidden detail. Furthermore, one issue with this transformation is that the grey levels existed also in the non-noisy image areas however these values have also been shifted, this therefore mean as a side effect some integrity has been lost.

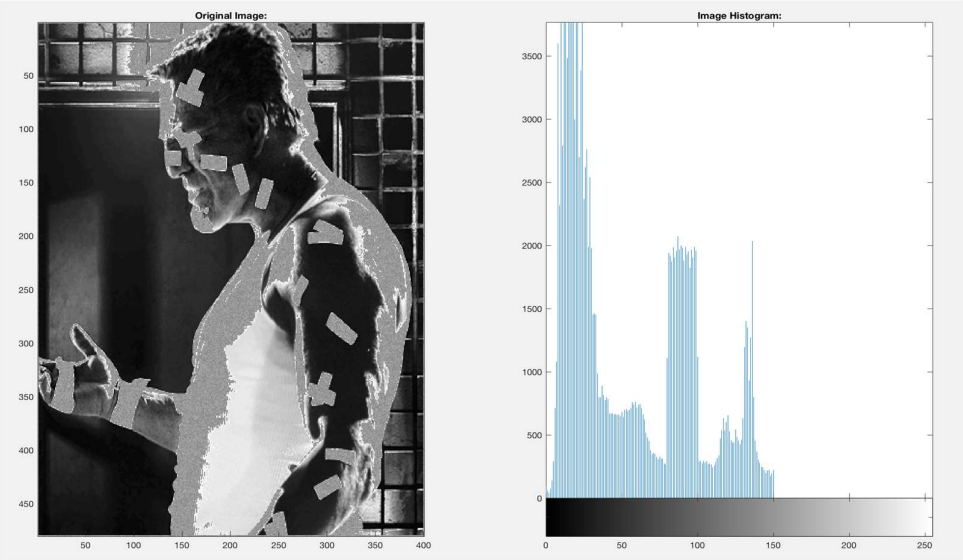


Figure 3: original image before grey level slicing.

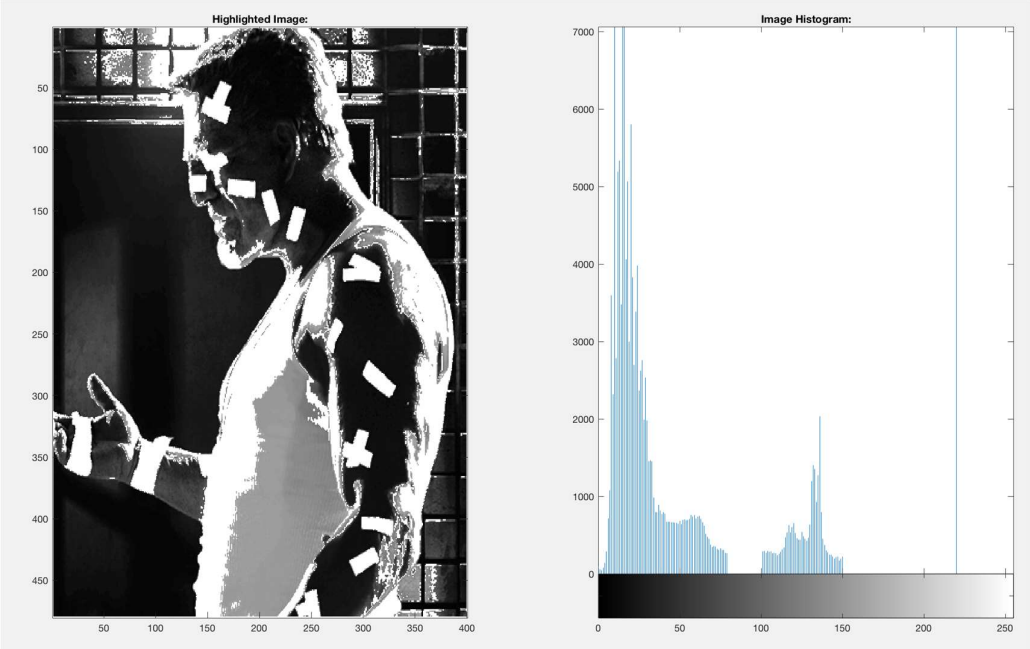


Figure 4: image after grey level slicing

Task 3

For this task, we used neighbourhood processing to remove noise and smooth an image. We used a kernel sized 5x5 and zero padding around the image to compensate for this. We start off by converting the image to greyscale to only worry about transforming a single 2d matrix. The size of this matrix is then recorded in variables x and y. We declare kernel the kernel size of 5 and use an inbuilt function “padarray” to pad the newly created output image matrix with 2 zero columns and rows so that the first original pixel of the input image is the starting point of the kernel. This also means the end point for the x and y axis is the edge of the original image. A further image is created once again the size of the original image without the 0 padding to serve as the output.

We create a new image for the output imaging filter and created two nested for loops running for the length of x and y, incrementing by one. We declare the variable sum to equal 0 so that after the kernel loops have finished running the variable is reset to 0 on the next iteration. We then create 2 further nested for loops that run for the length of the kernel (therefore running 25 times).

Inside these kernel loops the variable sum is updated each iteration and added to itself. Sum is equal to itself plus 1/25 ($kernel^2$) multiplied by the current value of the padded image matrix. This additive process repeats for the kernel size to give a sum of all pixel values inside the kernel. As we have already divided the pixel value by 1/25 we do not need average the sum. We save this completed kernel sum as the current value at the index of the initial for loops in the output image.

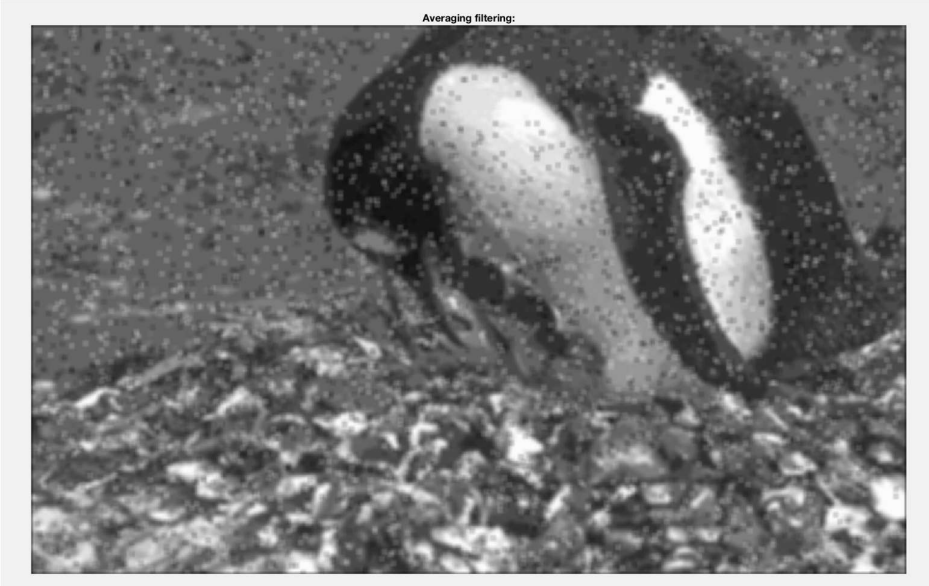


Figure 5: Output after Averaging filtering

Following this we create a new image matrix to store the result of Median filtering. We once again create two for loops to run the length of the image on both y and x axis. These too start at one rather than 0. We then create a new array: store. We create an array as it allows us to push data to the end after every kernel iteration and also reset it after the full kernel loop has finished.

We then create two further for loops to iterate for the kernel size and therefore run for each value of the kernel. We push to the end of the store array after each loop for each pixel value at the current value in the padded image to create an array of 25 pixel values. After the two kernel for loops end we use an inbuilt function “median” to calculate the median of the array (store) and then save this value as the pixel value at the current position of the output image (first two for loop indexes).

Median filtering is most suited for removing salt and pepper noise such as in this image. it does this whilst still retaining edges to an extent without smoothing and blurring the image too much. This is one advantage of averaging, even though noise remains edges are sharper and the image is less blurred. The processing times and costs have to be considered especially with a large kernel size, in averaging filtering a simple sum and division has to be calculated whereas with median filtering pixel values have to be ordered and chosen which even when using algorithms such as bubble sort or quicksort is still demanding on the processor. A further drawback with median filtering is that although it works well with salt and pepper noise Gaussian noise is not removed effectively and instead a Gaussian filter should be used.

It should also be noted that the kernel could also could be resized to affect the output image. The larger the kernel the smoother and less sharp the image however the better noise removal; especially for salt and pepper noise. A smaller kernel would result in a

sharper image however less noise removed so we have to find a happy median between sharpness and noise removal relative to kernel size.

Figure 6: output after median filtering

Task 4

First, we have to convert the image to greyscale so we can transform it. From previous tasks, we have shown that median filtering is best for removing salt and pepper noise whilst retaining image sharpness to an extent. As the image contains a lot of salt and pepper noise median filtering is preferable over Gaussian or averaging, however if the image contained these we would use a different filtering technique. The inbuilt function “medfilt2” is used to filter the image. Medfilt2 uses a 3x3 structuring element to perform kernel operations to smooth the image and remove noise, before this however it pads the image with 0’s in order for the kernel to fit. 0 padding is then removed after the image is smoothed.

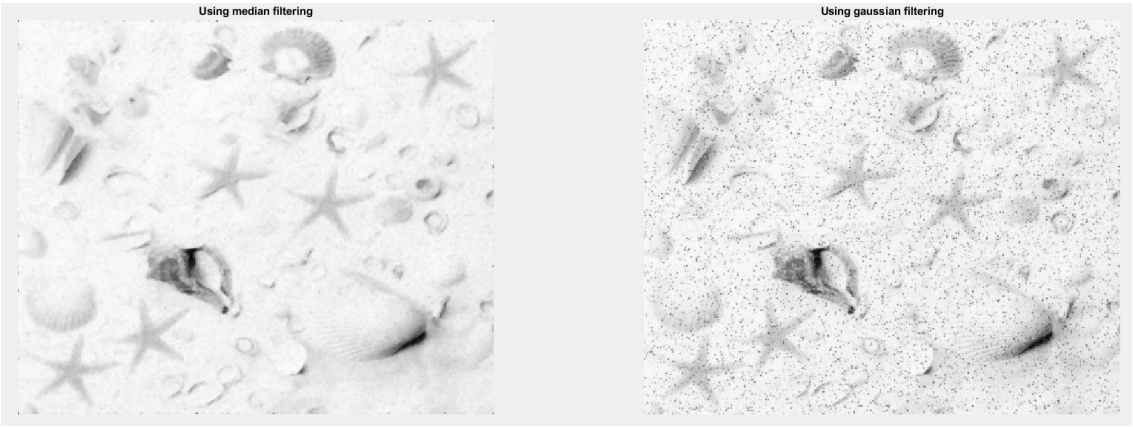


Figure 7: Effects of Gaussian Vs Median filtering

Next we need to separate the foreground and background. To do this we need convert the image from greyscale to a binary image. A binary image is different to a greyscale as a greyscale image ranges pixel values from 1 to 255, a binary image on the other hand only uses 1 to designate white and 0 to designate black. In image processing traditionally most images are converted to binary in order to recognise distinct regions, in a greyscale image this would not be computationally efficient, however it does have some uses in RGB images such as skin tone recognition.

To binarize the image we need to threshold it. Thresholding in its simplest form is a conditional if statement. If the current pixel value falls below the constant set the value to 0, if it falls above the constant set it to a 1. In our task we use a global limiting threshold of 0.9. Compared to a local limiting factor this uses up less computation power however is affected greater by salt and pepper noise in the final output (K. Chaubey, 2016). We could use Otsu's method of global thresholding however Otsu is flawed in the fact although it draws from histogram data to segment twice we would need a greater range as most of the greys in the image are very low contrast, therefore we could use contrast stretching at the cost of performance however this is unnecessary.

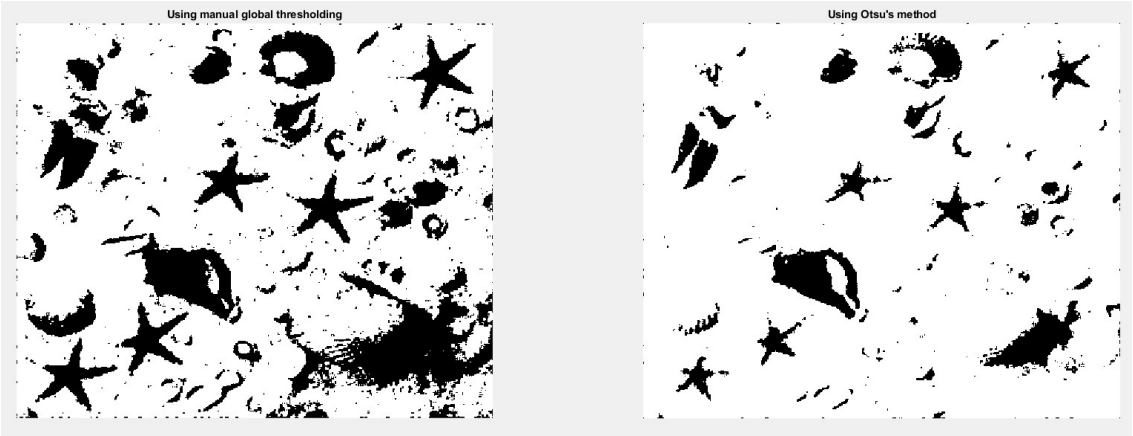


Figure 8: A comparison of manual vs Otsu's method of thresholding

We can see now there is still both noise and protrusions from shapes. Fortunately, there is a mathematic morphologic operator to solve this. Closing is a morphological process where first we dilate the image, then erode the image. First we dilate; dilation is the process of using a structuring element to make the boundaries of regions larger whilst also filling in any holes in regions, as a by-product this gives shapes a more rounded smoother edge. We then erode the image. Erosion is the process of using a structuring element to reduce object boundaries whilst also making holes smaller and removing protrusions, erosion can also result in removing some degree of noise. In the closing operation the same sized structuring element is used for both operations, in our task we use the square 3x3 structuring element to retain some sharpness of edges whilst also using a small enough kernel that our shapes don't become too round. However, as you can see as the edges aren't 0 padded some noise is left on the boundaries.

Figure 9: image after closing operation.

Now that regions are visible and noise has been reduced we try to find the edges of objects. We use the inbuilt function “edge” with the condition of canny. We use canny edge detection over sobel because canny implements a further step over sobel by first applying a low pass filter to filter out some noise then applying the sobel detection method to detect contrast differences. As the low pass filter has been run more edges are found as there is a larger difference in contrast.

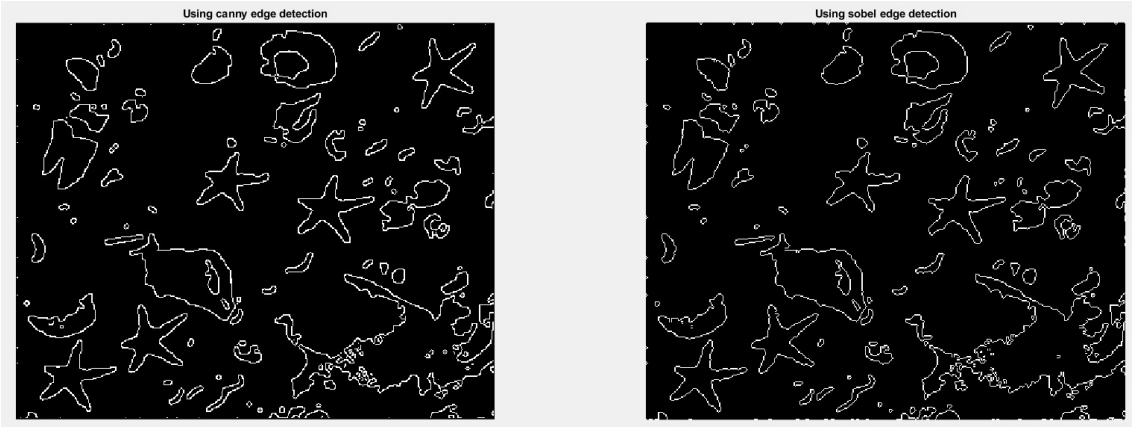


Figure 10: a comparison of canny vs sobel edge detection

Now that boundaries are found we need to now fill in these objects to create usable regions. We use the function “bwfill” to fill each object to make a region, this is further specified by the condition “holes”. However, as some shapes do not have complete boundaries they are not filled in.

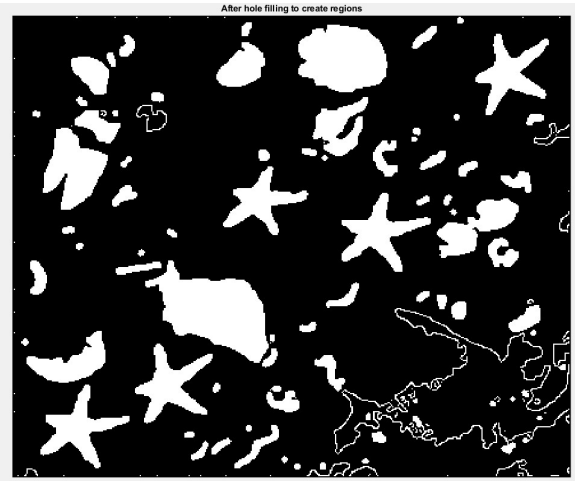


Figure 11: after hole filling to create regions.

Now we have complete regions we need to collect data about each. We use the function "regionprops" to get data about each individual region. We specify that we only want information about the area and perimeter of each object and store it in a 146x1 struct named p. Furthermore, now that we have regions to identify quantitative data we need to store the position of each region and its edges. We use the function "bwboundaries" with the condition "noholes" to store the coordinates of each region in a cell array named b. We also want a label for each region and so store this as variable L. These two variables are used later on to reference and add regions to the output image.

How do we quantify what a starfish is? A starfish, learnt from trial and error have two distinct identifiers. First, roundness; a starfish when measured for roundness should measure between 0.014 and 0.02. Secondly the area of a starfish is always above 230, however to separate starfish both criteria must be used otherwise we end up with objects that are vaguely the same area or as round as a starfish.

To compute this process we must first run a for loop that iterates over every found region stored in the cell array b. We must then separate the Area and Perimeter of the current object (found by using current index) into two separate variables for calculations. We then use a metric supplied to us in the brief to calculate roundness of the given region based on area and perimeter.

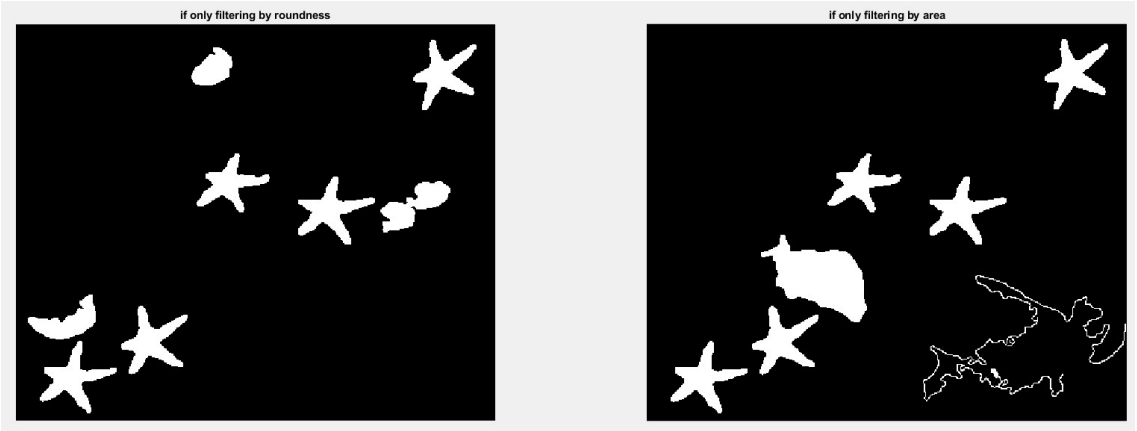


Figure 12: a comparison of only filtering by roundness and area

We use a further conditional if statement to check if the region fits the criteria for a starfish. Is the region between the upper and lower bounds for roundness and above the minimum area? If the answer is yes, we want to store this region in a new binary image. We do this by storing the current location of the region in question in a temporary variable intuitively named “temp”. Temp is then added to itself to make sure the starfish is added to the final output image. The final image is then displayed against the original to show found starfish.

Figure 13: original image Vs final image of found starfish

References

Dianyuan Han. (2013). In: International Conference on Computer Science and Electronics Engineering. Shandong: Atlantis Press, pp.1556-1559.

Titus, J. and Geroge, S. (2013). A Comparison Study On Different Interpolation Methods Based On Satellite Images. International Journal of Engineering Research & Technology (IJERT), Vol 2(Issue 6), pp.83-84.

K. Chaubey, A. (2016). Comparison of The Local and Global Thresholding Methods in Image Segmentation. World Journal of Research and Review (WJRR), 2(1), pp.3-4.