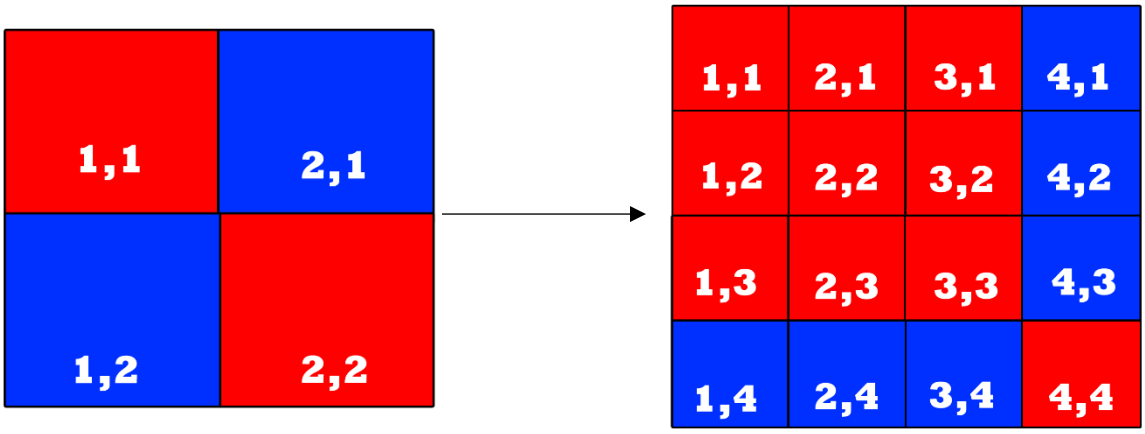
Image Processing Assessment Item 1

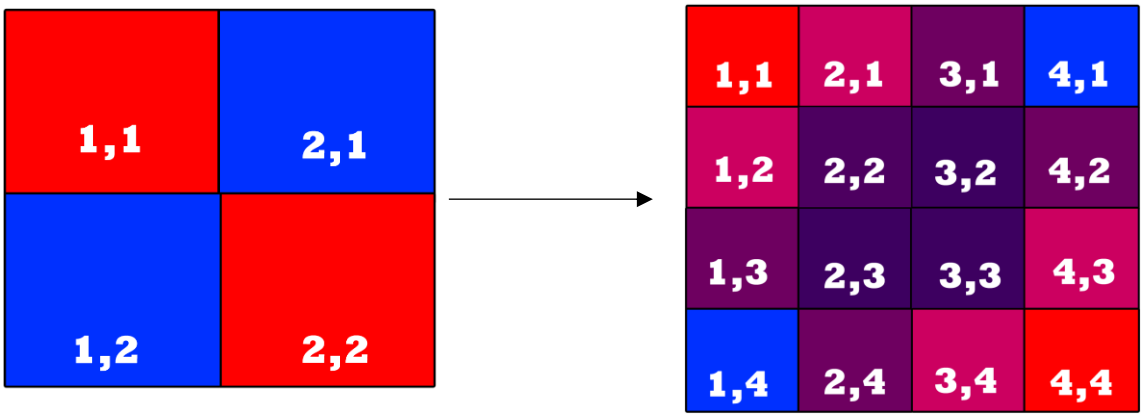
Task 1

Task 1 required the use of interpolation to resize an image. Two different techniques were used, Nearest Neighbour and Bilinear. To implement Bilinear Interpolation, I first implemented Linear Interpolation, and have included it in my code to show the steps that were taken to go from one method to the next, since Bilinear Interpolation is an extension of Linear.
I implemented Nearest Neighbour by looping through the values of the original image, and for each value in the original, replaced it with the same value, but in a 3x3 grid, meaning each pixel from the original image was then placed in the new image 9 times. This created an image which was technically the same as it was, but larger. The following image shows how an individual pixel (1,1) is expanded, as the red pixel is replicated 8 times to fill up the new 3x3 grid.



Similarly, I implemented bilinear interpolation with a nested for loop, going through each value and adding each value from the original image in its relative position in the new image, but then applying a formula on the values in-between to get their value. The formula is as follows: X = (1/3 * X1) + (2/3 * X2), where X1 is the pixel being examined, and X2 is the next pixel along to the right, or underneath, depending on what value is being searched for. This is actually the formula for linear interpolation, but it is the values not on the X/Y axis of the examined pixel ((1,1) in the example) that use bilinear interpolation, as they linearly interpolate in a 3x3 grid, instead of in a straight line. Linear Interpolation is relatively simple, but for values further into the grid, such as the middle pixel, it requires more work. To acquire these deeper pixels, linear interpolation must first be performed on the values above and below (so if the algorithm is searching for (2,2), then (2,1) and (2,4) need to be found first. And both of those pixels first require linear interpolation, so for example, (2,1) will need to be linearly interpolated between (1,1) and (4,1) to get its value. So once all these values are found, (2,2) can be found by

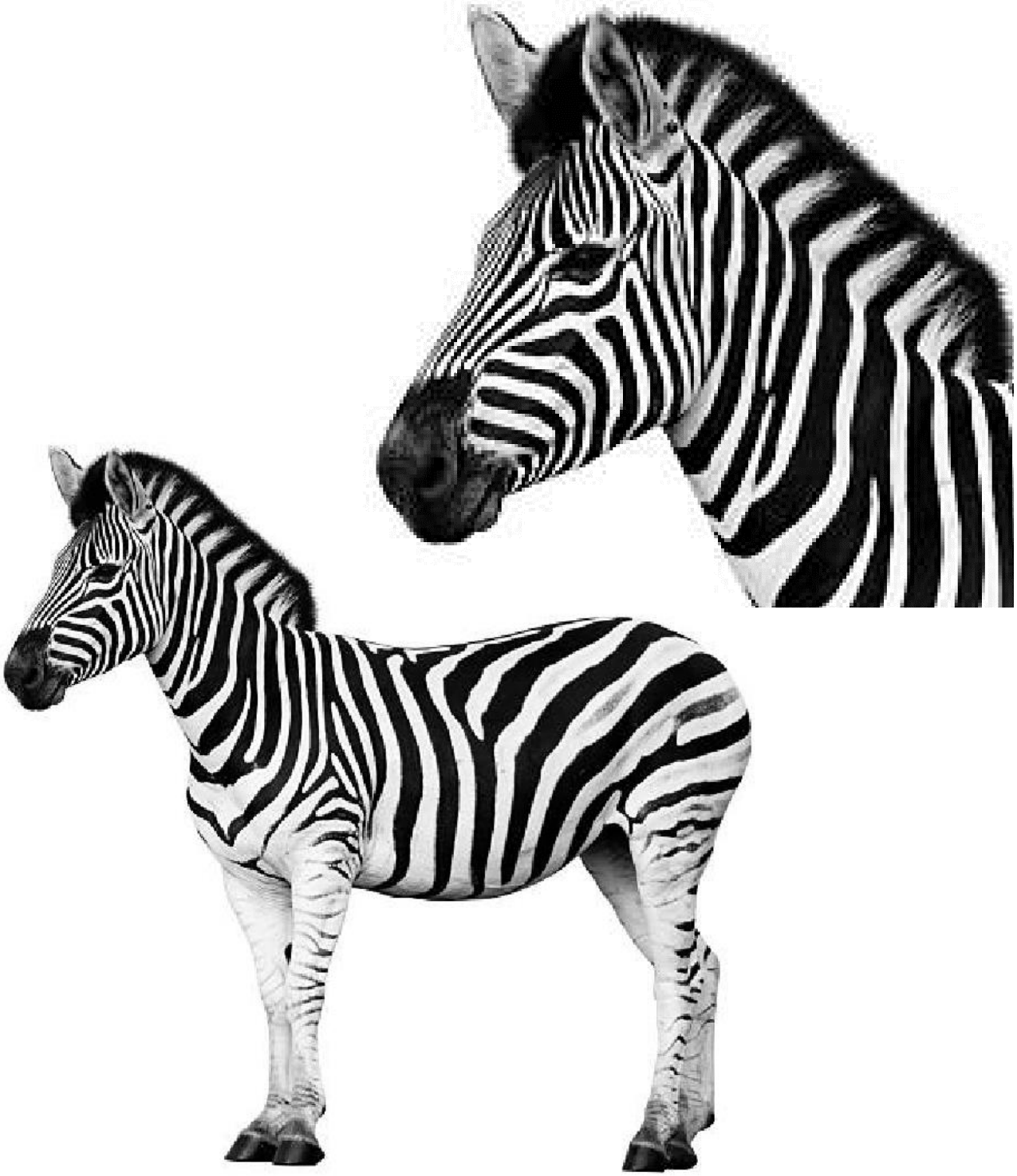
performing the formula: (2,2) = (1/3 * (2,1)) + (2/3 * (2,4)). This is represented in the image below:



Typically, Bilinear is considered a better form of interpolation for resizing processes than Nearest Neighbour. For example, resizing a photograph should yield better results when using Bilinear Interpolation than when using Nearest Neighbour, as it provides a gradient effect in the unknown spaces via the four known values around it, instead of filling them in with the same value of the closest neighbour, like Nearest neighbour would. This is seen in the outputs, as the image scaled with Bilinear Interpolation has retained a larger amount of its quality. However, Nearest Neighbour still has its uses, such as resizing pixel art. This is because resizing pixel art requires it to remain the same quality as it was, whereas Bilinear will cause a blurring effect, which will reduce the images sharpness. The blurring should make a typical image look better, as it will reduce jagged edges (as seen in the outputs below), but those sharper edges are a requisite of pixel art, and therefore should be retained.

Outputs

Nearest Neighbour Interpolation



Bilinear Interpolation

Task 2

For this task, we were simply required to use the transformation function in the Piecewise Linear function on our provided image. This boiled down to editing any pixels with a value in the range of 80 – 100, and changing them to 220. This was a simple function to do, and could have some interesting consequences. For example, this could be used for green screens – to ensure that every value in a range of green becomes one value, which can then be removed in place of the image which would be inserted in its place.

To achieve this output, I used a nested for loop to cycle through each individual pixel, and change its' value.

Outputs

Original:

Edited:

Task 3

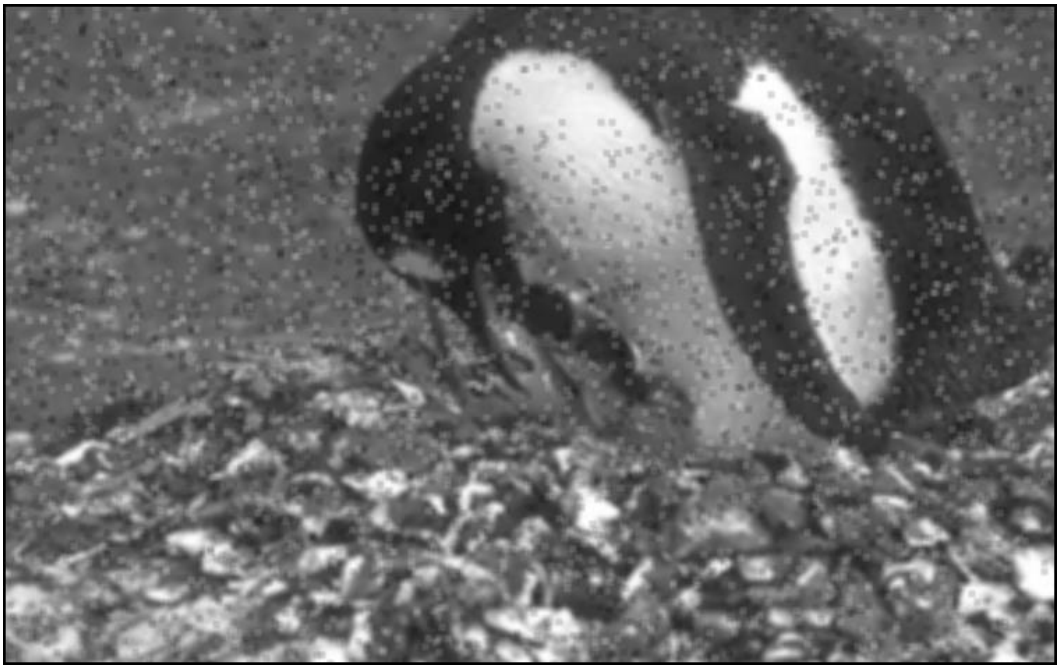
This task involved the smoothing of an image, using the smoothing filters Averaging and Median Filter, to remove the salt and pepper noise that had been added to the image. I first performed the mean filter by looping through every value in the image, and in each 5x5 grid, I calculated the mean value, and assigned it to the centre pixel. To perform the median filter, I followed the same steps, but instead of calculating the mean, I first had to order all the values in the 5x5 kernel, and then find the median value, and assign the centre value to that instead.

As you can see from the outputs below, the Median Filter was a significant improvement over the Averaging filter, as the Averaging filter didn't necessarily remove the noise, but instead spread it across the image (as using the mean makes the algorithm sensitive to outliers, which is what noise tends to be, as black/white values are at the extreme ends of the spectrum). Using the median allows the algorithm to completely ignore these values, as the median is unaffected by outliers, and replace it with the median value of the kernel/mask instead. In fact, reducing salt and pepper noise is one of the greatest strengths of median filtering (Gonzalez and Woods, 2017, 175), on top of its ability to reduce noise with less blurring than using the mean smoothing filter.

The main issue with using the median instead of the mean is the computation time, as calculating the median requires the data to be sorted. This can be done using efficient algorithms like the Quicksort, however this still brings with it a computational burden, and therefore for high quality, high resolution images, this may case issues.

Outputs

Averaging/Mean Filter



Median Filter

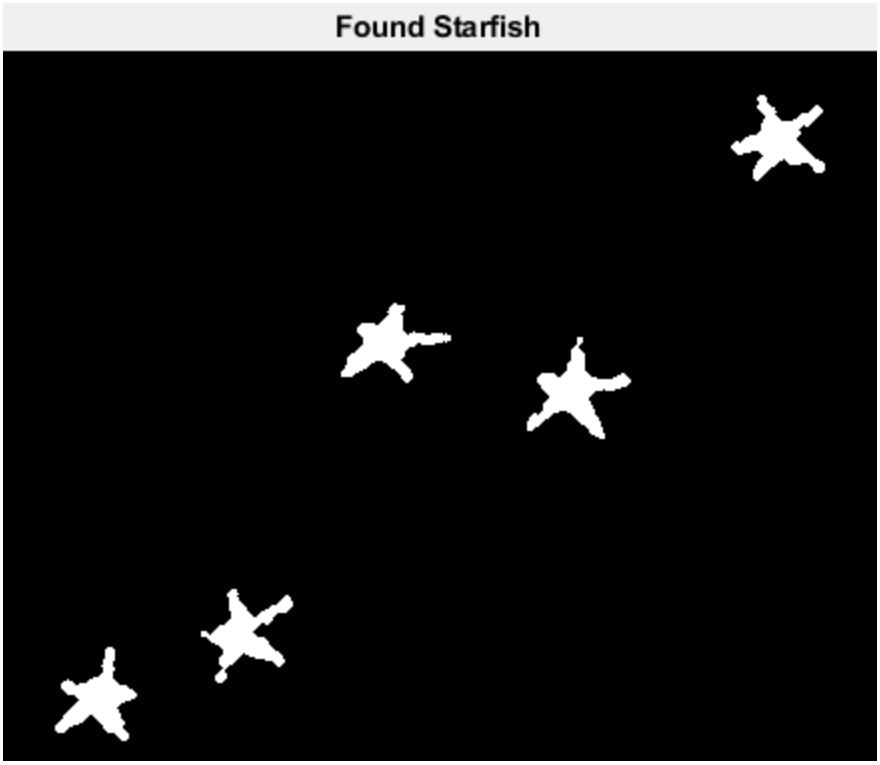
Task 4

This task took a lot more consideration than the first three. Multiple steps were needed to reach the required output. First, the image needed to be turned into a black and white image. It then needed multiple operations to make it usable in future calculations. For example, the morphological operation 'imclose' was used to fill in the holes. This was used instead of 'imdilate', which would have expanded the starfish and made them harder to find, whereas 'imclose' just fills in any holes between objects. I used the disk structuring element, as this allowed me to retain the most detail in the image whilst filling in the holes. Also, 'imadjust' was used to saturate the bottom and top 1% of pixels, allowing the image to appear sharper, and therefore the starfish to stand out more.

Following this, I turned the image into black and white, by using the 'imbinarize' function, and then used 'imcomplement' to invert the values, giving me the black background and white stars that I needed.

I then moved onto giving each object in the image a boundary, and using this to find out which was a starfish, based on its area, and its roundness. I found that the roundness of the starfishes was between 0.1 and 0.3, and the area was between 750 and 900, so after the loops I used 'bwareaopen' to remove the objects smaller than the starfishes, based on the range of areas I knew there would be. Similarly, I then did the same thing on values below 900, meaning the stars would be removed, leaving only the objects larger. I removed this from the original 'bwareaopen', which gave me a result of the starfish, as seen in the output.

Output



References

Gonzalez, R and Woods, R. (2017) Digital Image Processing, 4th edition. Harlow: Pearson.