**Task1 – Pre-processing**

**Task 1.1 Load input image**

For this task we can simply load our desired image from our image folder utilizing the “imread()” function, by passing in the full image name (including the extension e.g. .jpg).

**Task 1.2 Convert input image to greyscale**

For this task the “rgb2gray()” function was used, where we pass in the image we got as an output from task 1.1 into this function. This function converts the original image to greyscale by eliminating “the hue and saturation information while retaining the luminance”1. Converting to greyscale makes the image simpler to process and overall requires less memory than coloured images.

**Task 1.3 Rescale image using bilinear interpolation**

Interpolation is “process of using known data to estimate unknown values at other locations [essentially] an image method to increase (or decrease in our case) the number of pixels in a digital image” 2. It is useful for recalling a given image, as it produces smoother results then methods such as nearest neighbour interpolation and is quicker then bicubic interpolation. We can perform bilinear interpolation on our image using the “imresize()” function, passing in the greyscale image from task 1.2 as the input, setting the 2nd parameter of the function to “0.5” to indicate we want to reduce image size by half, and finally the 3rd parameter we set too “bilinear” to indicate that we want to use the bilinear interpolation method to carry out the resizing.

**Task 1.4 Produce histogram for the resized image** **before enhancing**

The “histogram()” function was used to generate a histogram of the image from task 1.3. Histograms are a great way to be able to visualise the image in terms of bins as rectangular bars, the height of the bar indicating the number of elements (number of pixels at each brightness level) in the bin, helping show the underlying data distribution of the image.

**Task 1.5 Enhance the image**

Image enhancement allows us to create a image that has better visual representation e.g. brightening, sharpening etc. 2 popular methods are contrast stretching and histogram equalization. Contrast stretching is a technique that seeks to increase picture contrast by extending the range of intensity values included in the image to cover a desired range of values. On the other hand, histogram equalisation, modifies the intensity values of all the pixels in the image such that the histogram is flattened. Both methods where trialled upon our scenario image. However the output of histogram equalization using the “imhisteq()” function, was rather harsh and over exaggerated the image darkness and contrast. Instead contrast starching via the “imadjust()” function was preferred as it give a more subtle enhancement but helped bring the details of the image out more. In the future I would also like to apply gamma correction, in order to lighten the left-hand side of the image, as we can observe there is a strong light coming from the east direction onto the image. This could be done using “imlocalbrighten”. However, this would not be robust as you would have to manually adjust the values based on every image.

**Task 1.6 Histogram after enhancement**

Like step 1.4 the “histogram()” method was used again but using the output image from task 1.5 as the input to the function. We can observe when we compare the histogram before and after enhancement (*figure 1 in task 1.8 section*), that there is little overall difference between the 2 graphs. This is because “contrast stretching maps the minimum and maximum intensity values in an image to the minimum and maximum values in the desired range, respectively. The overall shape of the histogram remains the same after contrast stretching. On the other hand, histogram equalization modifies the intensity values of all the pixels in the image such that the histogram is flattened. During histogram equalization, the overall shape of the histogram changes” 3.

**Task 1.7 Image Binarization**

Binarization is the process of converting greyscale / coloured images into a binary image (black and white pixels). It can be useful to separate the image foreground and background to pick out key objects, in our example being the washers and screws. The “imbinarize()” function was used, it takes several parameters, including the input grayscale image, the method of thresholding, the polarity of the foreground, and the sensitivity. For the threshold value we set it to “adapative”, this method computes a threshold for each pixel based on the local image contrast. Alternatively, we can use the histogram from task 1.6 to determine a threshold value, which is our example we could use a value of 120 (divided by 255), as this is the point on the graph where the peak starts the rise. However manually determining the threshold for an image is not very robust and so this was avoided. The polarity of the foreground was set to “dark”, this means that the foreground is darker then the background. Finally, the sensitivity was set to 0.5, as this value allowed the screws and washers to be seen the most.

Furthermore, for the threshold value, other methods such as the “Otsu method” can be used, “In the simplest form, the algorithm returns a single intensity threshold that separate pixels into two classes, foreground and background” 4. This method was also trialled using the “graythresh()” function, however the output was not ideal, and gave a heavily black pixel dominated image and distorted the objects. The Otsu method is mainly preferable in scenarios we have a bimodal image histogram.]

A screenshot of a computer screen

Description automatically generated**Task 1.8 Display the resultant images for task 1**

Figure 1. View the re-sized image, histograms before and after enhancement, enhanced image and the binarized image.

**Task 2 – Edge Detection**

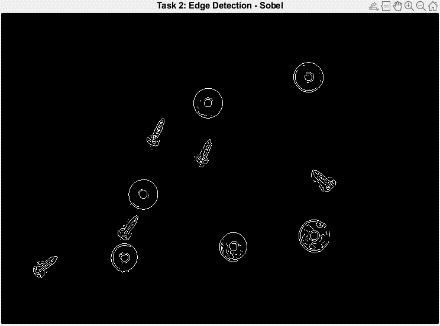
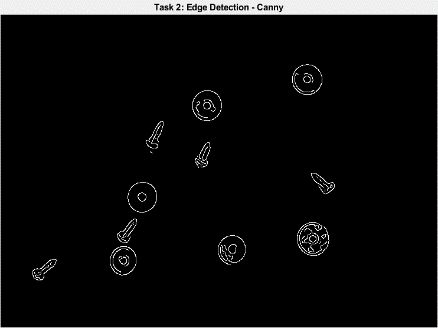
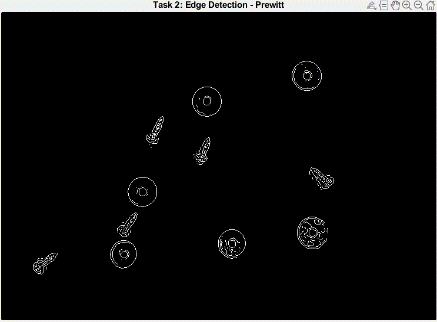
Edge detection is a common technique used to find places in the image where the intensity changes quickly. Before applying edge detection, it is a common practise to perform noise reduction in the image, as noisy images prompt variability in the local contrast along an edge. However, if we denoise an image to much it can blur the image and weaken the contrast around the edges, making it harder to perform edge detection. 2 noise reductions where experimented with both median filtering using the “medfilt2()” function and also gaussian filtering using the “imgaussfilt()”. From the experimenting median filtering was found to provide better results for edge detection in our scenario, as when applying gaussian filtering it would smooth the image too much making the edges harder to detect. However, the gaussian filtering function in MATLAB has a sigma parameter (that controls the amount of blurring) which can be adjusted to give better results, however due to the fact median filtering worked out the box and showed promising results, it was chosen to stick with this. After noise reduction, 3 edge detection algorithms using the “edge()” function was used to find the best edge detection result. The algorithms being Canny, Sobel, and Prewitt. The results can be seen in figure 2 (please zoom in to view details).

Figure 2. Image results using different edge detection algorithms (left to right, sobel, canny, prewitt)

Canny was found to give the best result, as all the edges it detected were smooth and more accurate. The Prewitt and Sobel methods gave very similar results but were missing some edge values and also showed more noise in the resultant image compared to canny. This was expected as “canny uses hysteresis to thresholding to detect edges allowing for more accurate detection” 5. On the other hand, canny can be more computationally expensive then Sobel and Prewitt so if computational resource was a key factor canny would not be a great choice.

**Task 3 – Simple Segmentation**

A screenshot of a computer

Description automatically generatedTo segment the image, we first must ensure all edges of the screws and washers are connected in order to be able to fill them so we can segment the objects. We can use the morphology method of closing which performs dilation then erosion of the image, essentially allowing any broken edges to be connected. The “imclose()” method can achieve this, and we also define and pass in a structing element to the closing function, the structuring element is binary shape we can use to interact with an input image. In our scenario we used the “strel()” function to create a structuring element the shape of a disk of radius 3, as it fits the fits the general shape of the screws and washers. To then segment the objects we can use the “imfill()” function and passing “holes” as a parameter option, that will in then fill in the holes/regions in a binary image. We then get the screws and washers filled in as white and the background as black, we can then easily distinguish the objects. Finally to remove any small objects (generally caused by noise) that are not screws/washers we can use the “bwareopen()” method with a threshold value of 20. Giving the result shown in figure 3.

Figure 3. Segmented image result

**Task 4 – Object Recognition**

To recognise the specific objects and label code them with colour into either a washer or a screw, we have to have a certain feature/value we can use to differentiate the 2 possible object categories. It was decided to use each segmented objects aspect ratio to identify if it is a screw or washer, the aspect ratio being ratio of the width to the height of an object. Aspect ratio was chosen as the comparative feature as our sample images are all taken from the same birds view angle, and are similar zoom levels, so the size and shapes of each screw and washer are on the same level, making them easy to differentiate via aspect ratio. We first use the “regionprops()” function that allows the extraction of specific properties for each object in a segmented image, in particular we want the 'MajorAxisLength' and 'MinorAxisLength' properties. The aspect ratio is then calculated used the major and minor axis length properties and is used to assign a colour to each object in the image. [If the aspect ratio of an object is greater than 2, it is assigned the colour red for indicate a screw, otherwise, it is assigned the colour orange](https://www.mathworks.com/help/images/ref/regionprops.html) to indicate a washer. The colours are stored in the “cmap” variable which is used as the colourmap, [that is used to assign colours to each object in the image based on its aspect ratio](https://www.mathworks.com/help/matlab/ref/colormap.html)6.

**Task5 – Robust Method**

For task 5 to achieve a robust method to accurately segment and recognise the three types of objects (i.e. long screws, short screws, and washers) from all provided images, we have to identify the key points of code from tasks 1 – 4, that are hard coded to our first sample image. Once the hard coded elements are adapted the code from task 1-4 can be implemented into a function and we can loop through all 10 samples images, apply our custom function, and return the results. From tasks 1-5 there are no major hard coded elements, however for task 4 it works only for given screw and washers but cannot differentiate large screws to small screws. We opted to add an additional “elseif” statement, that checks if the aspect ratio of an object is greater then 1.8 but less then 4 it is classified as a small screw, however if the ratio is greater then 4 it is a large screw. We then take the code from task 1-4 with the modified task 4 and create a custom function named “screw\_washer\_detection“, that takes a single argument that is the input image you wish to segment, and returns 2 objects “labeled\_image” (segmented image) and “cmap” (colourmap of the segmented image). A loop is then created from 1 – 10 (for all 10 images in our dataset), that extracts the image name and attempts to read in each image in a “try,catch” block, catching resulting errors in the catch block such as a invalid file name or file was not found error 7. If the image was successfully read, we can pass it through the custom function we defined that will go through the task 1-4 code and segment the objects and return a successfully segmented binary image, along with its colourmap to distinguish between washer, small and long screw. Example of the full process of a randomly picked image from our dataset (IMG\_07) is displayed in figure 4.

“asked which processing stages bashir wants to see as figures (wait for email)”

**Task 6 - Performance Evaluation**

This task involved calculating the dice score which calculates the Sørensen-Dice similarity coefficient between a image an its given ground truth. In MATLAB we can utilise the “dice()” method to compare our segmented images from task 5 to its ground truth, a dice score of 1 indicating a perfect overlap and a score of 0 showing no overlap. The formula for dice score being:

*dice(A,B) = 2 \* | intersection(A,B) | / ( | A | + | B | )*

where | *A* | and | *B* | are the number of elements of each set 8.Also the precision and recall values was calculated for each segmented image and ground truth using “bfscore()”, precision being “the fraction of detections that are true positives rather than false positives” and recall being “the fraction of true positives that are detected rather than missed” 9. Formula for precision and recall being:

*Precision = True Positive / True Positive + False Positive*

*Recall = True Positive / True Positive + False Negative*

The evaluation metric scores can be seen in figure 5, along with the mean and standard deviations for each metric in table 6.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **IMG\_01,**  0.945,  1,  1 | **IMG\_02,**  0.928,  0.969,  0.996 | **IMG\_03,**  0.941,  1,  1 | **IMG\_04,**  0.910,  0.957,  0.989 | **IMG\_05,**  0.898,  0.943,  0.995 |
| **IMG\_06,**  0.870,  0.942,  0.988 | **IMG\_07,**  0.910,  0.928,  0.999 | **IMG\_08,**  0.938,  0.982,  1 | **IMG\_09,**  0.917,  0.951,  0.999 | **IMG\_10,**  0.940,  0.995,  1 |

|  |  |  |
| --- | --- | --- |
|  | **Mean** | **SD** |
| **Dice** | 0.920 | 0.023 |
| **Precision** | 0.967 | 0.026 |
| **Recall** | 0.996 | 0.004 |

Figure 5. Table of metric scores [image, dice, precision, recall]. Figure 6. Table of mean & SD.

REFERENCES:

1. [Convert RGB image or colormap to grayscale - MATLAB rgb2gray - MathWorks United Kingdom](https://uk.mathworks.com/help/matlab/ref/rgb2gray.html)
2. Image processing lecture slides 36 week 3
3. [image processing - Difference between contrast stretching and histogram equalization - Stack Overflow](https://stackoverflow.com/questions/41118808/difference-between-contrast-stretching-and-histogram-equalization)
4. [Otsu's method - Wikipedia](https://en.wikipedia.org/wiki/Otsu%27s_method)
5. [Comparing Edge Detection Methods. There are many different edge detection… | by Nika Tsankashvili | Medium](https://medium.com/@nikatsanka/comparing-edge-detection-methods-638a2919476e)
6. [View and set current colormap - MATLAB colormap - MathWorks United Kingdom](https://uk.mathworks.com/help/matlab/ref/colormap.html)
7. [Execute statements and catch resulting errors - MATLAB try catch - MathWorks United Kingdom](https://uk.mathworks.com/help/matlab/ref/try.html)
8. [Sørensen-Dice similarity coefficient for image segmentation - MATLAB dice - MathWorks United Kingdom](https://uk.mathworks.com/help/images/ref/dice.html)
9. [Contour matching score for image segmentation - MATLAB bfscore - MathWorks United Kingdom](https://uk.mathworks.com/help/images/ref/bfscore.html)