Image Processing Summative- Report

System design

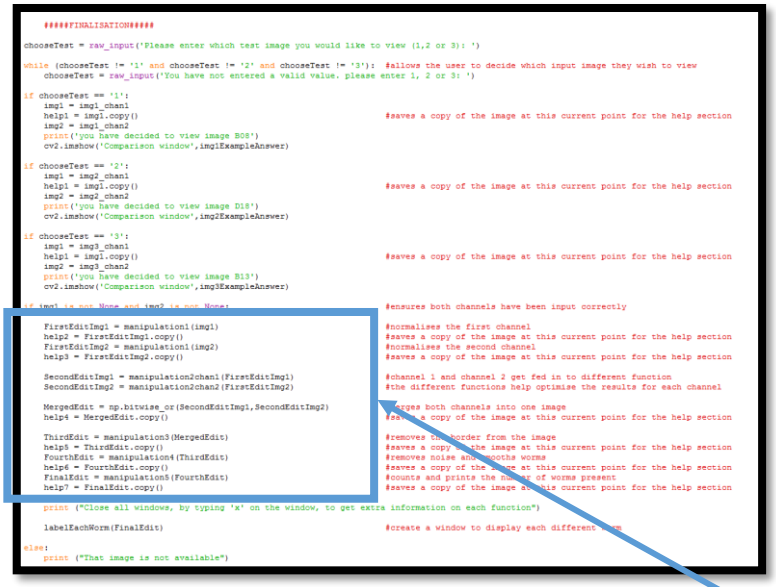
I decided to design my system such that the different functions would chronologically manipulate the inputted images to produce the final edited version.

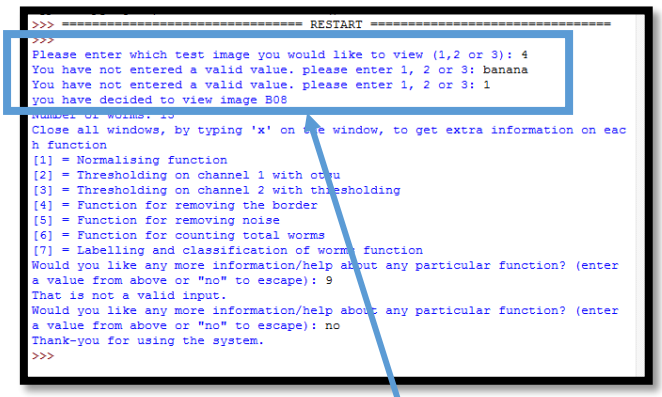
The different functions are all at the start of the document, in order and appropriately labelled, with the functions sequentially being called at the bottom of the file. This allows for a clean and easy to understand demonstration of exactly which manipulations are going on at which point.

After the functions have been called, a help option is made available. This offers the user the ability to view exactly what Is happening at each function during the manipulation on the input image.

I have supplied 3 reference images for the system to be tested on, which can be chosen by the user when the program is initially ran. For error correction, if an incorrect input was entered, the user would be prompted again to enter a valid value.

Demonstrations of how my system is designed and how it runs (including error handling) is shown below:



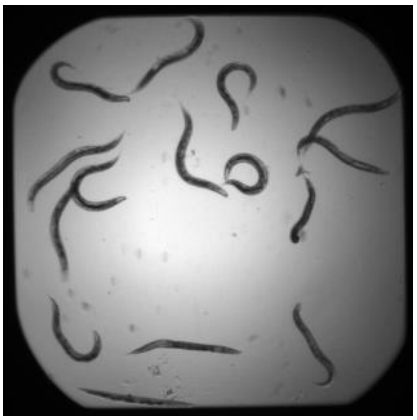


Error handling in action

Each function sequentially being called

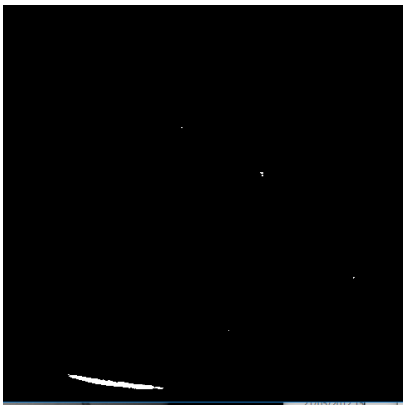
The system in use and problem approach

The following set of images demonstrates how the system manipulates the input images for each channel at each step of the process. The explanations describe how approached the problem and how I came to the solution.



manipulation1(img): Normalise the image

I initially realised that the input file was barely visible and would need some manipulation to stretch the 'non-black' pixels in order to make the image able to be processed properly. I achieved this by using the 'cv2.normalize()' function on a 16 bit and then converting the image to an 8-bit image with 'img = (img/256).astype(np.uint8)'. Both channels are input to this function.

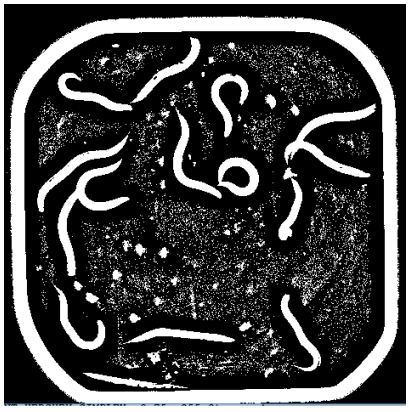


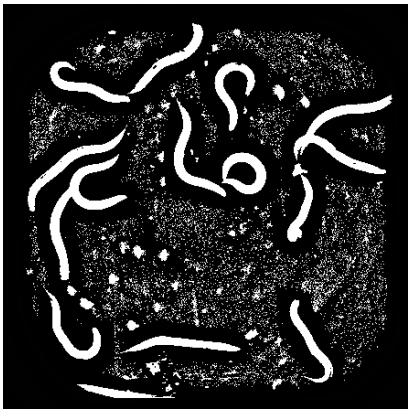
manipulation2chan1(img): Binary thresholding for channel 1

After normalising, I knew that the image would need to be converted to a binary one to allow for further manipulation. I found the first channel particularly difficult to turn in to a binary image of great quality however I found that the function 'cv2.threshold(img,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)' worked better than most others attempted

manipulation2chan2(img): Binary thresholding for channel 2

Despite 'Otsu' working better for channel 1, I found that the function 'cv2.adaptiveThreshold(img,255,cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY,31,2) ' . The adaptive threshold allowed for a very clear image with a significant blank border around each worm, however the downside to this function is that it created a large border which proved an issue for some input images.

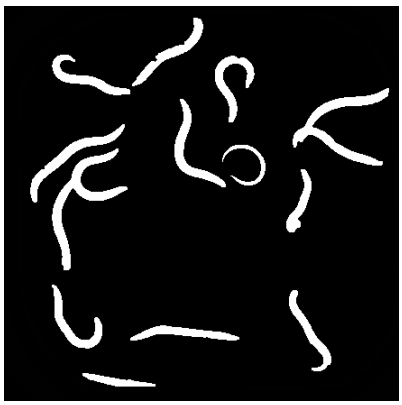




manipulation3(img): Merge channels and remove border

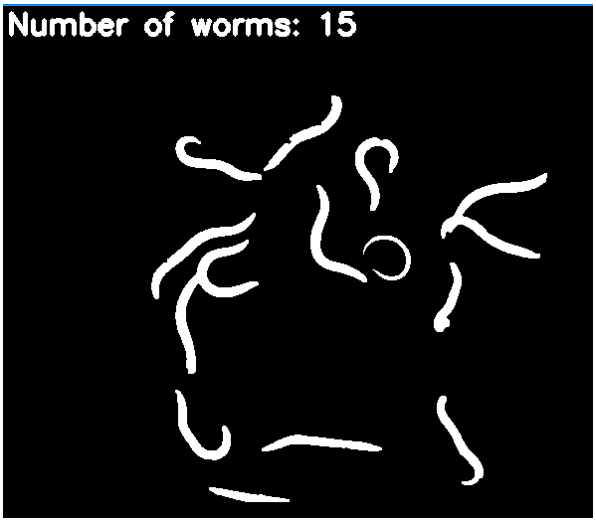
To optimise results, I decided to merge both channels at this point. I achieved this by using '.bitwise_or(SecondEditImg1,SecondEditImg2)'. The border was removed by identifying the largest worm by using contours and colouring it black. I used the contours functions: 'cv2.findContours()' and 'cv2.drawContours()' and created the following code to identify the largest worm (the border):

```
im2, contours, hierarchy = cv2.findContours(img.copy(),cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
height,width = img.shape[:2]
blank = np.zeros((height,width),np.uint8)
largestLength = 0
CurrentLargest = None
for i in contours:
    length = cv2.arcLength(i,True)
    if length > largestLength:
        largestLength = length
        CurrentLargest = i
cv2.drawContours(blank, [CurrentLargest], -1,255, 3)
blank = cv2.dilate(blank,np.ones((35,35)),iterations = 1)
cv2.drawContours(img, [CurrentLargest], -1,0, 3)
```



manipulation4(img): Remove noise by colouring black

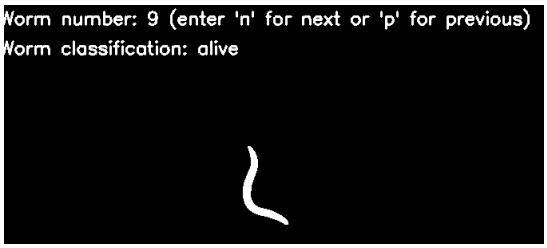
At the next step I knew that the noise produced from the previous edits needed to be removed. I achieved this by again using the contours functions and removing any collection of white pixels that had a size smaller than the specified amount of 100. The size of each contour was found by 'cv2.arcLength()'.

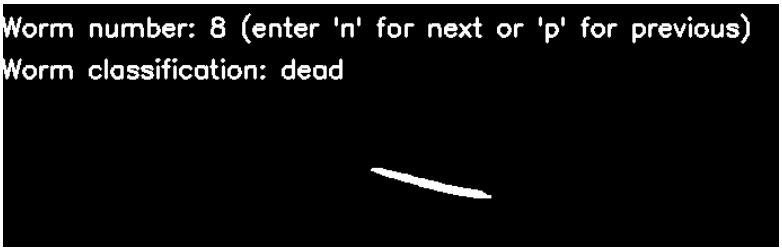


manipulation5(img): Counts worms

To accurately count the number of worms I had to use a bit of intuition. For any collection or overlap of worms, I dictated whether there was more than 1 worm present by comparing the length of its contour against the average size of a worm. If the contour was bigger than the specified amount, then +2 was added to the counting total number of worms which yielded accurate results across the different images.

labelEachWorm(img): Identifies, classifies and labels each worm





To display each worm, I simply iterated through the contours previously found and displayed each on a new blank window. To evaluate if each individual worm was dead or alive I used the function cv2.minAreaRect() for each contour- If the height and width differed by a ratio greater than 1:6 than the worm was evaluated to be dead, and if not, then it was evaluated to be alive.

For this section I wanted the user to be able to iterate through each worm identified by entering a 'next' and 'previous' key, and for an evaluation on whether the worm was dead or alive to be displayed. I achieved this by using the following code to display the information:

```
cv2.putText(display,("Worm number: " + str(wormNumber + 1) +  " (enter 'n' for next or 'p' for previous)"),
if((width/height)> 6 or (height/width)> 6):
    cv2.putText(display,("Worm classification: dead"), (5,65), cv2.FONT_HERSHEY_SIMPLEX, 0.75, 255,2)
else:
    cv2.putText(display,("Worm classification: alive"), (5,65), cv2.FONT_HERSHEY_SIMPLEX, 0.75, 255,2)
cv2.imshow('Individual worms',display)
key = cv2.waitKey(0)
if (key == ord('x')):
    keepWindow = False
    cv2.destroyAllWindows()
elif (key == ord('n')): #right key
    wormNumber +=1
    wormNumber %= numberOfWorms
elif (key == ord('p')):
    wormNumber -= 1
    wormNumber %= numberOfWorms
```

Comparisons against the ground truth data for each example image

Image B08:

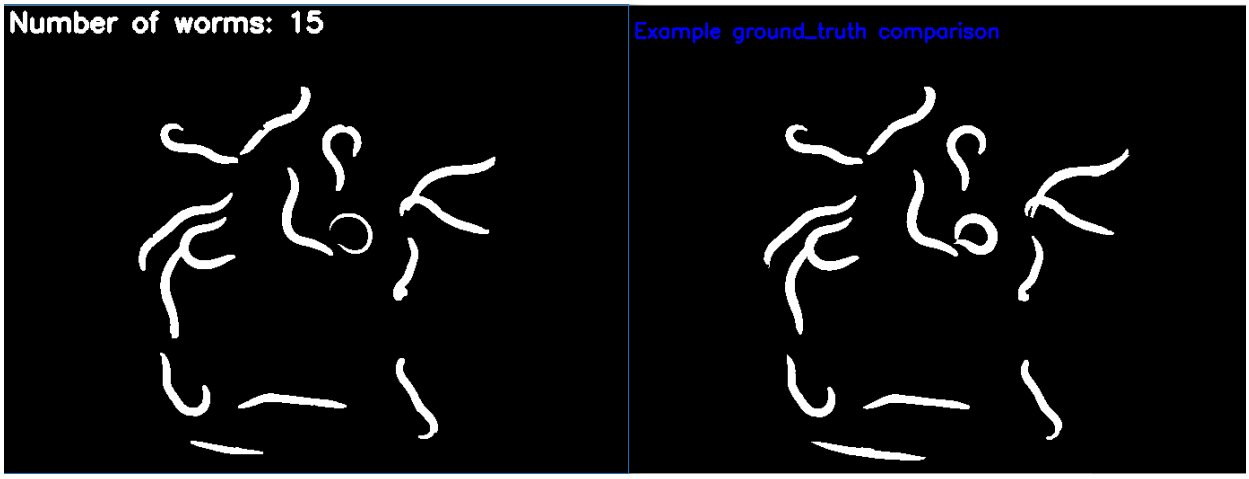


Image D18:

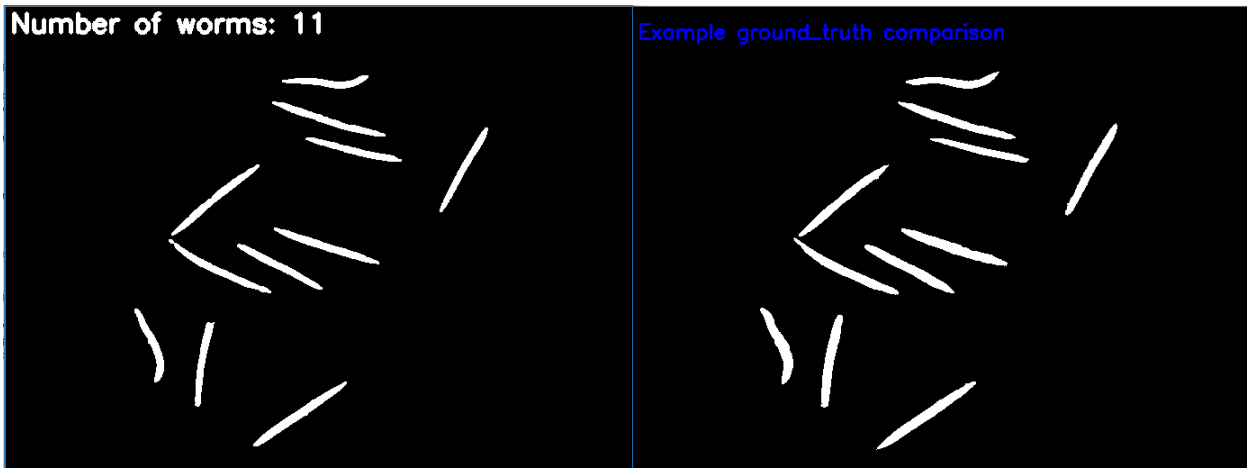


Image D13: