# Computer Vision Assignment 1 – CMP9135

# Task 1: Image Segmentation and Detection

## 1A – Plant Segmentation

The code for this section is “Task1Run” and “segmentPlant”.

We first read the plant image in using the “imread” function and the specified name of the plant image, this is out RGB image. We convert this image to grayscale using the “rgb2gray” function, specifying the RGB image as the argument and it will return a grayscale image. With a grayscale image, we can apply a mask to it to find the edges of the objects within, we apply a Sobel mask for this, the directional masks are:

X = [-1 0 1;-2 0 2;-1 0 -1]

Y = [1 2 1;0 0 0;-1 -2 -1]

As we apply each mask, we emphasize different sectors of the image, by adding them both together we form a final edge map with the object edges identified. This is still a gray image however the edges of the plant object now appear lighter than the background, therefore when binarizing the image using “imbinarize”, the resulting image will more accurately display the edges of the plant object.

We can further this by removing the small objects within the image using “bwareaopen”, specifying a size to remove, which we set to 100. Some of the plant images however do not have edges that fully connect therefore, we can use morphological functions to connect the edges. We specify a structuring element for the morphological function using the “strel” function, we specify a disk element with the size of 1, this size is large enough to close the edges without distorting the image. We can use this with the “imclose” function to join the edges. With the edges now closed we can use the “imfill” function to fill the holes of the image. Afterwards, we use the “imerode” function with the structuring element specified before to reduce the size of the image closer to the original as previous functions have slightly largened it. With this, the image is now fully segmented with the plant object appearing white and the background as black. Examples of this for images 2 and 5 can be seen in the top left corner of figure 1.

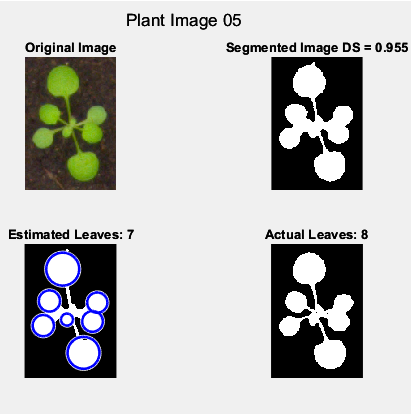


Figure 1 Plant Images 2 and 5

## 1B – Dice Score

The code related to this section is “Task1Run”, “truthIMG” and “diceScore”.

With the image now segmented we can compute the dice score. The dice score will calculate the similarity between the segmented image and the truth image by overlapping them and returning a percentage of what overlaps.

We first read the truth image using the “imread” function, however we will use a differing binarization method to the previous image read. Due to some images struggling to binarize properly, we will calculate a threshold to use to binarize the image. We use the “adaptthresh” function, we specify the truth image, the strength which we set to 1, the Foreground Polarity to Bright, meaning that the foreground is brighter than the background and the Statistic used to calculate the threshold to medium. We use this threshold with the “imbinarize” function to binarize the truth image.

Now with both images we can compute the Dice Score using this equation:

Where M represents the segmented image and S is the truth image. In terms of Matlab code, we will use the “sum” function, “(:)” and “&” operations to find the number of pixels that match within both images, multiply this by 2 and divide by the total number of pixels for both images.

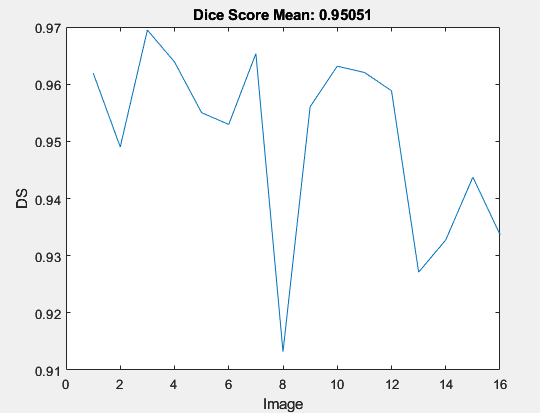


Figure 2 Dice Score for all Images

The dice score for images 2 and 5 can be seen in figure 1 as 0.94902 and 0.955 respectively. The dice score for all images can be seen in figure 2 with the mean being 0.95051.

## 1C – Leaf detection

The code for this section is “Task1Run” and “findLeaves”.

Due to the circular nature of the leaves, we found it useful to the use the “imfindcircles” function to find them. We specify the radius range for the circles we want to find including the minimum and maximum which we set to 5 and 100 respectively, we found these values best to identify the leaf objects without creating circles that do not represent anything. We identify 3 returns for the function including “centers”, “radii” and “metric”. The “centers” and “radii” will be used with the “viscircles” function to visualise the found circles on the last figure that was created, which is the segmented image which can be seen in figure 1. The length of the return “metric” can inform us on how many circles or leaves were found. In figure 1 we can see that for plant images 2 and 5, we can see that it identified 5 and 7 leaves which is close to the 7 and 8 actual leaves respectively.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Plant Image | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| Actual Leaves | 7 | 7 | 8 | 8 | 8 | 7 | 8 | 6 | 6 | 7 | 8 | 9 | 8 | 8 | 6 | 7 |
| Detected Leaves | 7 | 5 | 7 | 7 | 7 | 6 | 8 | 2 | 6 | 6 | 7 | 7 | 7 | 8 | 6 | 6 |
| Difference | 0 | 2 | 1 | 1 | 1 | 1 | 0 | 4 | 0 | 1 | 1 | 2 | 1 | 0 | 0 | 1 |

Table 1 Actual versus Detected Leaves

From table 1, we can calculate that the program missed 16 total leaf objects, meaning on average the function missed 1 leaf per image.

We could have performed this task using morphological operations to remove the stems of the leaves and subsequently count the numbers of remaining leaf heads using “bwlabel” with “regionprops” and bounding boxes to display them. This method however would have performed significantly worse on images where the leaf nodes were smaller or similar size to the stems and would have struggled identifying multiple leaf heads in a singular object.

# Task 2: Feature Calculation

## 2A – Shape features

The code for this section is “Task2Run”, “formatImages” and “patchImages”.

We first read in each image using the “imread” function, saving each of the “depth” “rgb” and “truth” images separately. Using these images, we use the function “formatImages” passing the truth image. Given that the weed and onion classes are identified using colour, we can separate out the image by colour to attain the different classes, for this we use “(:,:,3)” and “(:,:,1)” on the truth image to attain the weed and onion class respectively. The third argument in this code specifies the colour channel to keep, whilst the others are discarded. With this we use “bwlabel” to label each object in the image separately to then be used in “regionprops” to return the properties specified for each object independently. We specify within the “regionprops” arguments "BoundingBox", "Circularity", "Eccentricity", "Solidity", "Area", "Perimeter", "PixelList", and "Centroid". These will be used to calculate the shape features as well as create the patches. The shape features we want to calculate are “Solidity”, “Eccentricity”, “Circularity” and “Noncompactness”.

We will then return and use the “patchImages” function passing as arguments the objects related to the classes, the “rgb” image, “depth” image and the cells or arrays that will be used to store the patches and the shape feature values. This function will iterate through all of the objects passed, by using the syntax “Objects(i).x” where x represents the regionprops argument, we can return a value related to that “regionprops” argument for the specified object in the iteration “i”, for example “Objects(i).BoundingBox” will return the bounding box values for the object “i”. We use this bounding box value to crop both the “depth” and “rgb” image to create the patch for that object. We calculate both the “Solidity” and “Eccentricity” using the values returned from “regionprops”. Next, we calculate the second moments in the x and y direction for use in calculating the circularity and noncompactness. The x direction second moment is calculated using this formula:

Where x represents all the x values in the object and represents the value of the x coordinate for the centroid. We can get all the x values in the object using the “PixelList(:,1)” and we can get the value for the x coordinate of the centroid using “Centroid(1)”. The y direction second moment is calculated using:

Where y represents all the y coordinates for each pixel in the object and represents the value of the y coordinate for the centroid. We can use the same methods to attain the respective values “PixelList(:2)” and “Centroid(2)”.

We can use this information to calculate the circularity, the standard formula for circularity in this case being:

Where A(R) is the area and P(R) is the perimeter. However, this method of calculation is limited when objects have regions missing throughout, therefore we use another method using moments which is defined as:

Where is the area of the object squared, is the second moment in the x direction and is the second moment in the y direction. This calculation performs significantly better on objects with regions missing. We can calculate the noncompactness with the inverse of the circularity formula:

This noncompactness formula also uses the moments in both the x and y direction as well as the area squared. This formula however can perform poorly on objects that are very small, therefore we include a correction factor that will replace the section :

This correction factor is a sixth of the area of the object and can be included into both the noncompactness and circularity formula. This is now means that the value for circularity will be in a range of 0-1 as it should be, whereas before using the formula without the moments meant that the value could exceed 1.

With all information calculated regarding the shape features, we can store them in the respective arrays for future use, we can also add to the index related to each class, the number of objects just objects just calculated, this will be used later to distinguish between the train and test class.

We will test if the picture in focus is the 19th which would represent the start of the test data, if the condition is met, we will store the current index in an index related to the test data.

Figure 3 displays the distribution and mean for each of the shape feature for both classes.

From this we can deduce that the “Eccentricity” metric is worst for distinguishing the classes due to the distributions and mean being similar, whilst “Solidity” and “Circularity” would be best given the largest change in distribution and mean.

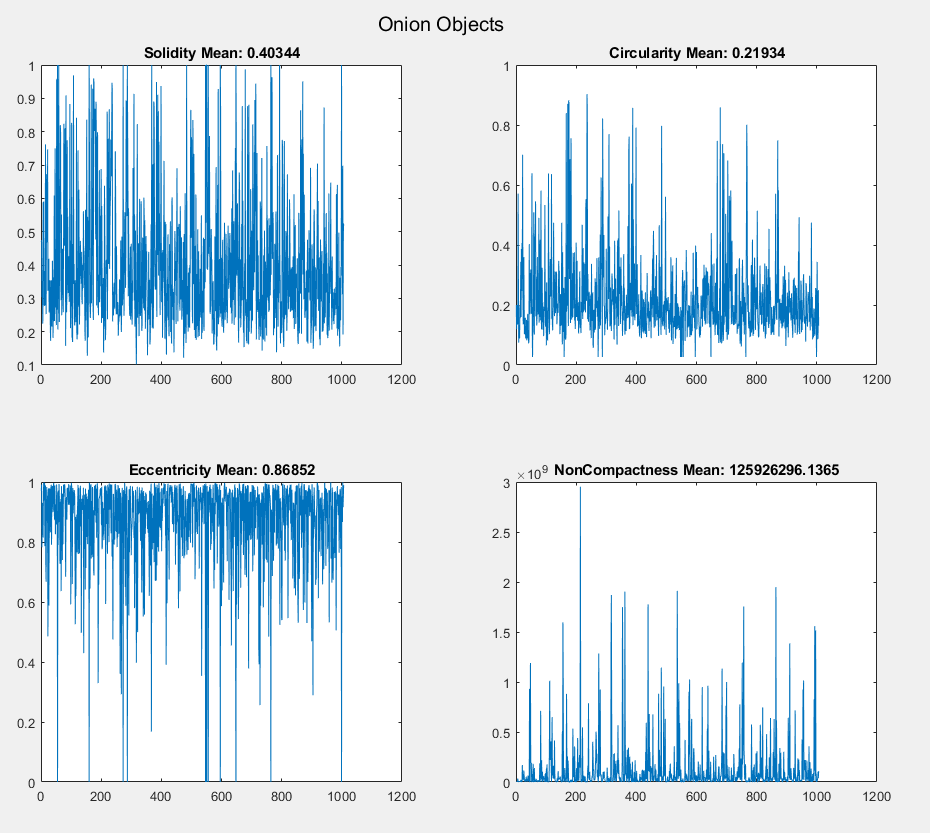
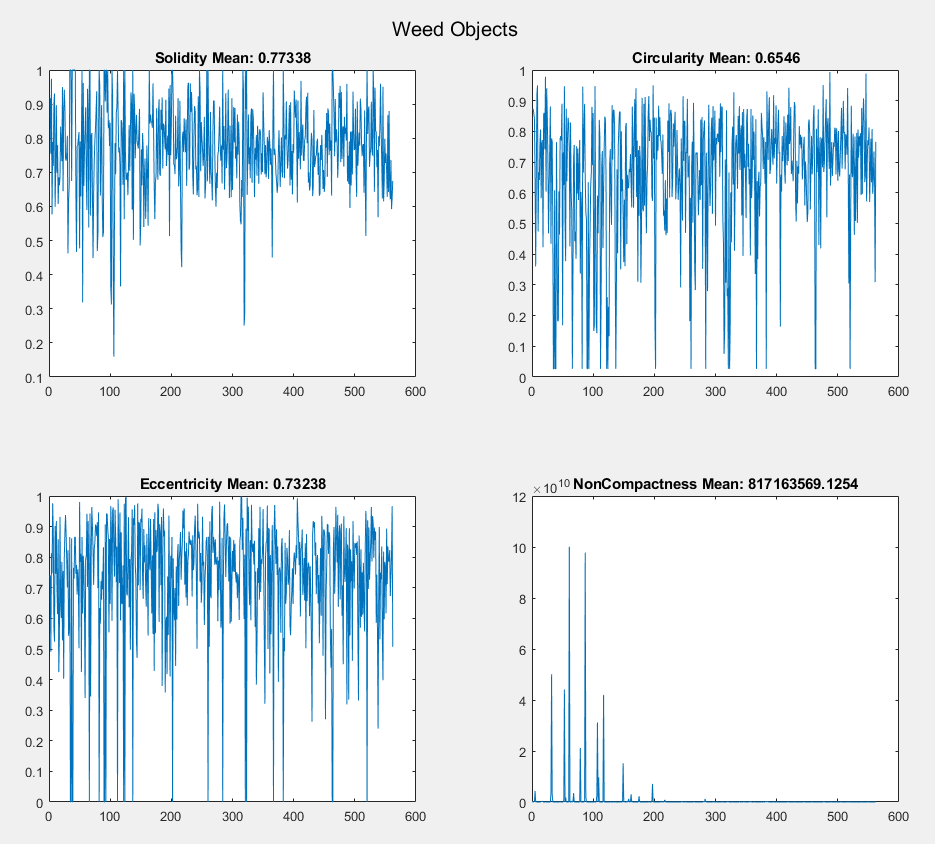


Figure 3 Shape Feature plots for weed and onion objects.

## 2B – Texture features

The code for this section is “Task2Run”, and “textureFeatures”.

We can pass both the weed patches and onion patches that were created and saved in the previous section, to the function “textureFeatures” as well as all the relevant arrays to store information to to begin this section. The function first saves the offsets for the 0, 45, 90 and 135 degree rotation as [0 1;-1 1;-1 0;-1 -1] respectively. We can iterate through all the patches and use functions “graycomatrix” and “graycoprops” to gather the texture features information we want including the “Contrast”, “Correlation” and “Angular Second Moment”. As we want the information for four colour channels, we will need to split the patches related to the “rgb” into the red, green, and blue using the “imsplit” function, we can use the patches of the “depth” image for the infra-red channel. For each of the colour channels, we will pass the respective patch to “graycomatrix” as well as the offsets identified. This function will calculate the gray-level co-occurrence matrix (GLCM) for each of the offsets. The GLCM will represent how often a pixel with gray-level value i occurs horizontally adjacent to a pixel with the value j (MathWorks,2023). We can pass the related GLCM to the function “graycoprops” as well as arguments for features we want to calculate, these include “Contrast”, “Correlation” and “Energy” where “Energy” is the argument for the angular second moment. The equation the function uses for the “Contrast” is (Mathworks, 2023):

The equation for “Correlation” is (Mathworks, 2023):

The equation for “Energy” is (Mathworks, 2023):

The value that the contrast property will return is a measure of the intensity contrast between the pixels and the image, the correlation property will return a measure of correlaiton between the pixels and the image, and the energy property is the sum of squared elements in the GLCM (Mathworks, 2023).

We can use the syntax “x.y” where x represent the variable for the “graycoprops” and y represents a property, for example “redStats.Contrast” will return the contrast values for all orientations for the current image patch. As we want both the average and range of the values, we can use the “mean(redStats.Contrast)” for the average and “max(redStats.Contrast) - min(redStats.Contrast)” for the range.

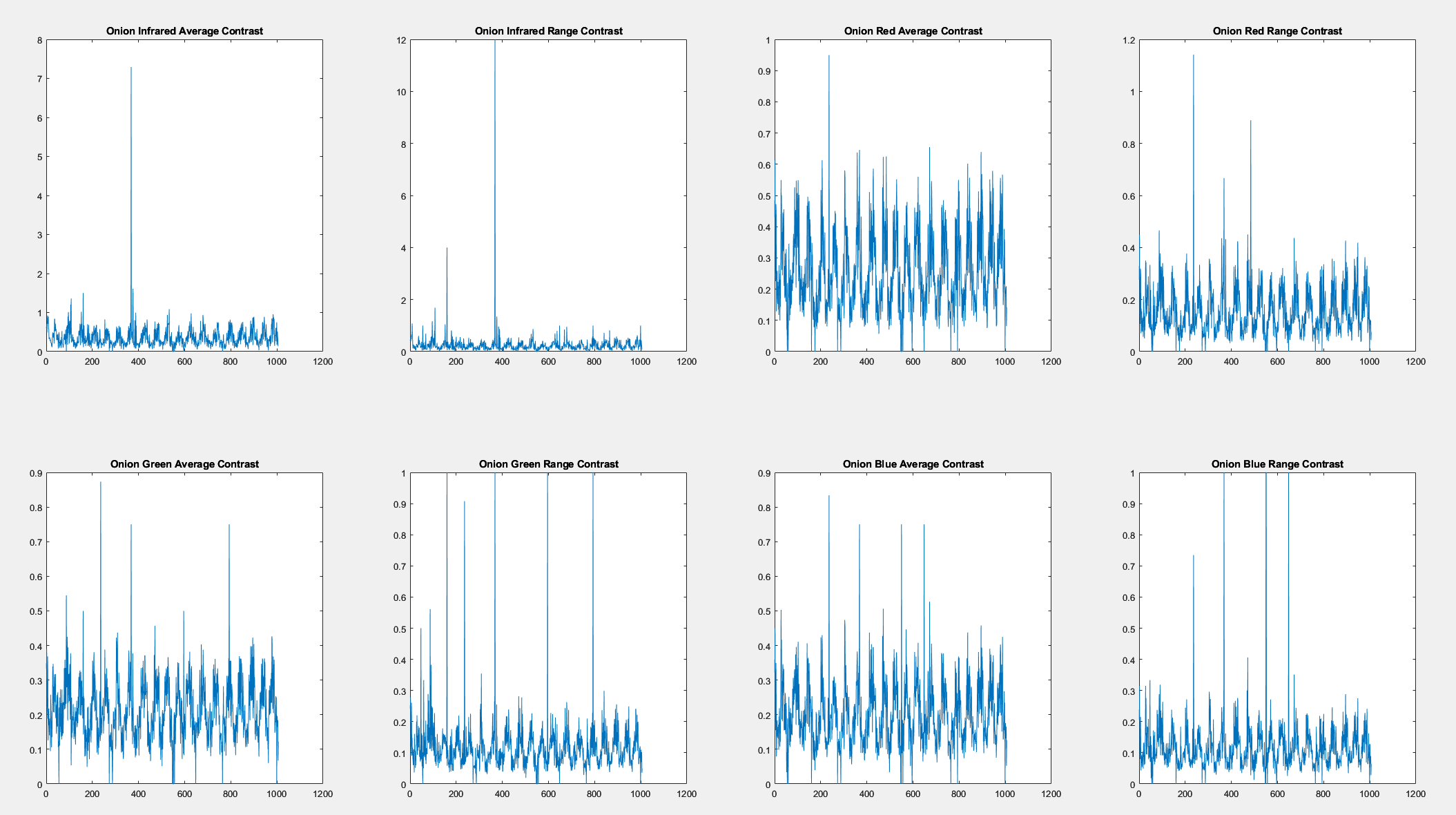


Figure 4 Average and Range for the Contrast of the onion objects

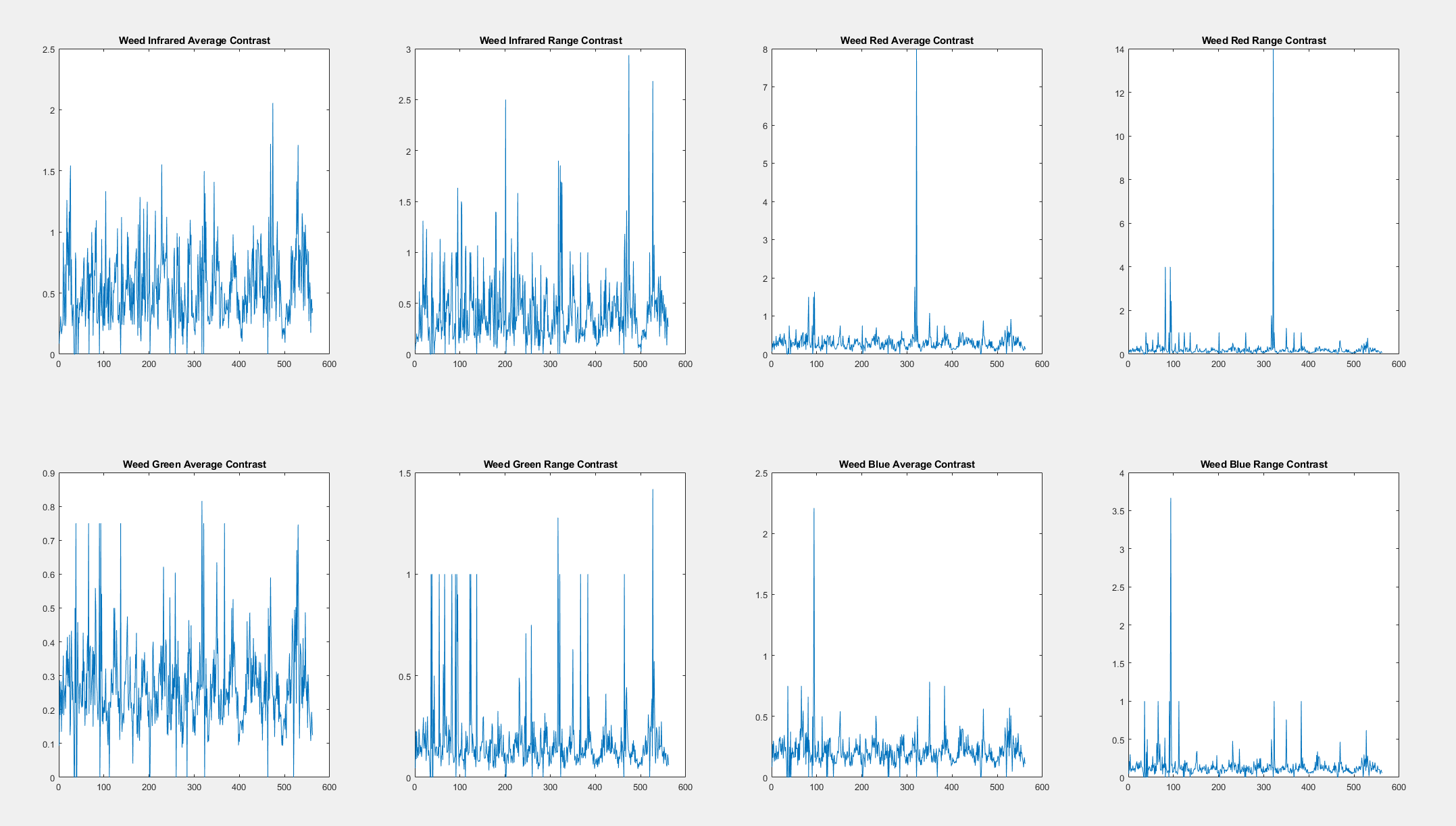


Figure 5 Average and Range for the Contrast of the weed objects

## 2C – SVM

The code for this section is “Task2Run”, “splitTrainTest”, “precisionRecall”, “testOutputPredictions”.

Having saved all shape features and texture features from previous sections, we first need to split the data into the train and test, for this we use the “splitTrainTest” function. We will pass the information we want to split as well as the testing index for both weed and onion. This function will create two arrays, one for train and another for test, before iterating through both the weed and onion data separately. If the iteration value “i” is more than or equal to the related testing index, it is saved into the testing array, if not then it is saved to the train array. As we always iterate through weed related data first, the train and test data will always store all weed data before onion data. We will perform this for all the arrays that correspond to each shape and texture feature. A similar approach is used to create and array for the “y” values.

Once all data is split, we can create relevant tables for each of the models, we want four models, a model trained using only shape data, a model trained using only feature data, a model trained using both data and a model trained using only the 10 most important features from the model with all data included. We will create separate train and test tables for all models we want to train, we pass to the “fitcvsm” function the train table and the train “y” variables. This returned model is then passed to the “predict” function along with the test table, this will return the prediction labels for each row of the table. This returned label array can be used in the function “precisionRecall” and “testOutputPredictions”.

“precisionRecall” is a basic function to calculate the model’s precision and recall metrics from the actual values and the predicted values. Precision is calculated using:

Recall is calculated using:

TP represents True Positive, FP is False Positive and FN is False Negative. We will want to calculate the metrics for both classes separately therefore the value that is “Positive” and “Negative” will change during calculation. TP are cases where the actual value and predicted value are the same “Positive” value, FP are cases where the actual value is “Negative” and predicted value is “Positive” and FN are cases where the actual value is “Positive” and the predicted value is “Negative”. We can view the precision and recall metrics in Table 2.

“testOutputPredictions” will recalculate for the 19th and 20th images, the objects related to each class and there bounding boxes. By doing this we can identify what the test and predicted values relate to, which means we can traverse the arrays and find the correct pairing to accurately display the predication and calculate the precision and recall metrics. We will calculate the precision and recall metrics for each class for each image, whilst the previous function calculated it for both images. We will display both the truth image and use the “hold on” and “rectangle” function to place the bounding boxes all on the correct image, when done with the first image, image 19, we can use “hold off” and create a new figure. We can view the precision and recall metrics for each class for each image in Table 3.

To determine the 10 most important features, we can use “fscchi2” which will use individual chi-squared tests to determine importance, we will pass the table for the combined model and the y variables. To view the top 10 features we can use “idx(1:10)”.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Model | Weed Precision | Weed Recall | Onion Precision | Onion Recall |
| Shape Features | 0.5789 | 1 | 1 | 0.0909 |
| Texture Features | 0.9434 | 0.4545 | 0.5862 | 0.9659 |
| Combined Features | 0 | 0 | 0.4180 | 0.8977 |
| Reduced Features | 0.6089 | 0.9909 | 0.9474 | 0.2045 |

Table 2 – Precision and recall for each class and model

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Model | Shape Features | Texture Features | Combined Features | Reduced Features |
| Weed Precision (Image 19) | 0.5244 | 1 | 0 | 0.5600 |
| Weed Recall (Image 19) | 1 | 0.4419 | 0 | 0.9767 |
| Onion Precision (Image 19) | 1 | 0.6471 | 0.4756 | 0.9167 |
| Onion Recall (Image 19) | 0.1136 | 1 | 0.8864 | 0.2500 |
| Weed Precision (Image 20) | 0.6204 | 0.9118 | 0 | 0.6442 |
| Weed Recall (Image 20) | 1 | 0.4627 | 0 | 1 |
| Onion Precision (Image 20) | 1 | 0.5325 | 0.3738 | 1 |
| Onion Recall (Image 20) | 0.0682 | 0.9318 | 0.9091 | 0.1591 |

Table 3 – Precision and recall for each class for each image and model

Table 2 gives us an understanding of the metrics for the entirety of the dataset whilst Table 3 gives detailed information into how the metrics are calculated for the images individually. If the class we want to measure is “weed” then precision is an indicator that there was less of the onions identified as weeds, whilst recall signifies that that there was less of the weeds identified as onions. Knowing this we can analyse that the model for only shape features correctly identified all the weeds in both images, however also predicted that many of the onions were also weeds. Texture features performed well in all aspects across both images. The Combined feature model performed poorly having not correctly identified a weed and predicting mostly onions. The Reduced feature model performed well at identifying objects that were weeds as weeds given the recall statistic as well as being good at not identifying weeds as onions given the high onion precision however in general made many weed predictions and could not identify onions consistently. Given this we would argue that the model using Texture features was the best. Figures 6 and 7 visualise the prediction onto the truth images, where a blue box means weed prediction and red box means onion prediction.

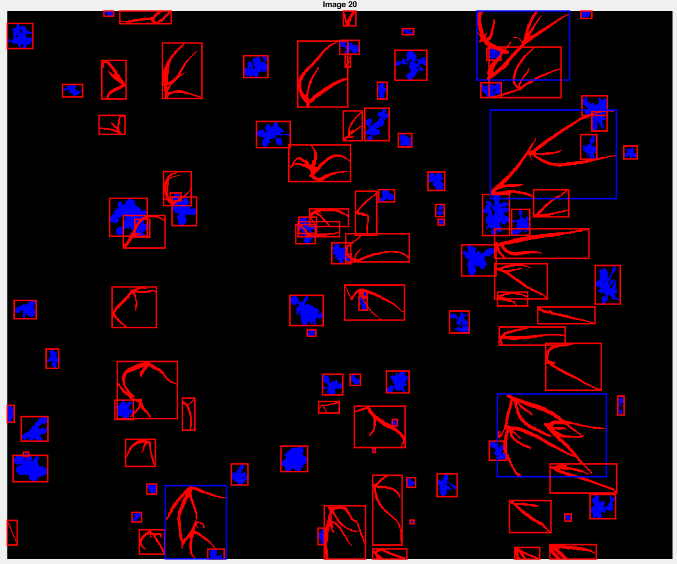
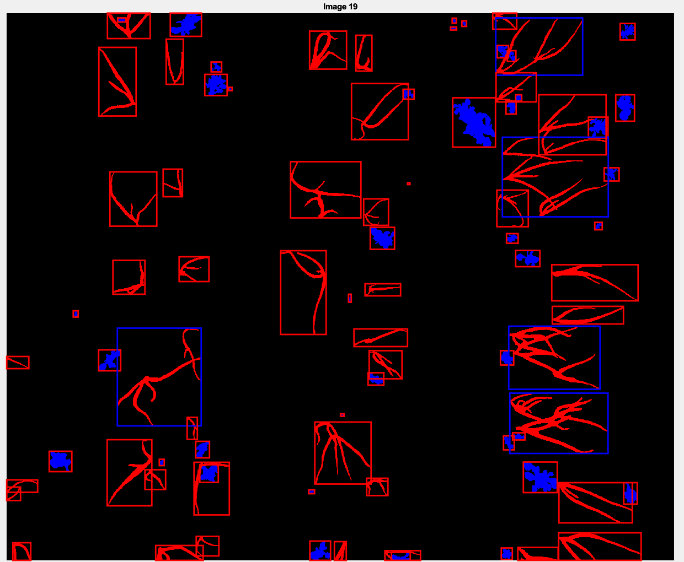


Figure 6 Combined Model Predictions

Task 3: Object Tracking

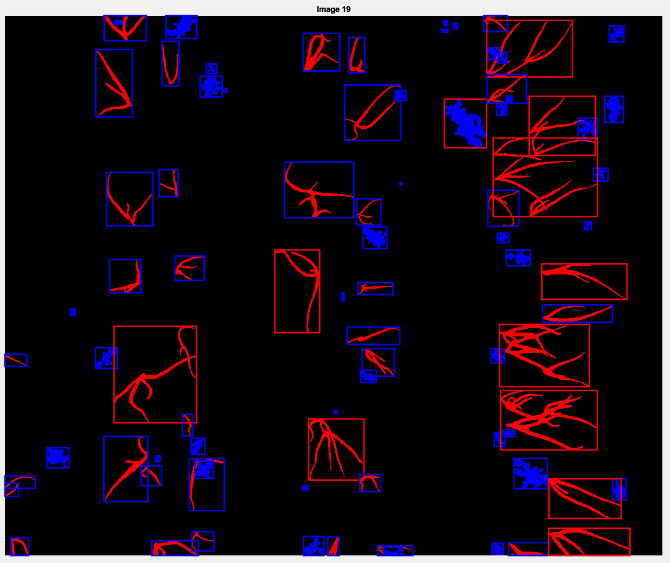
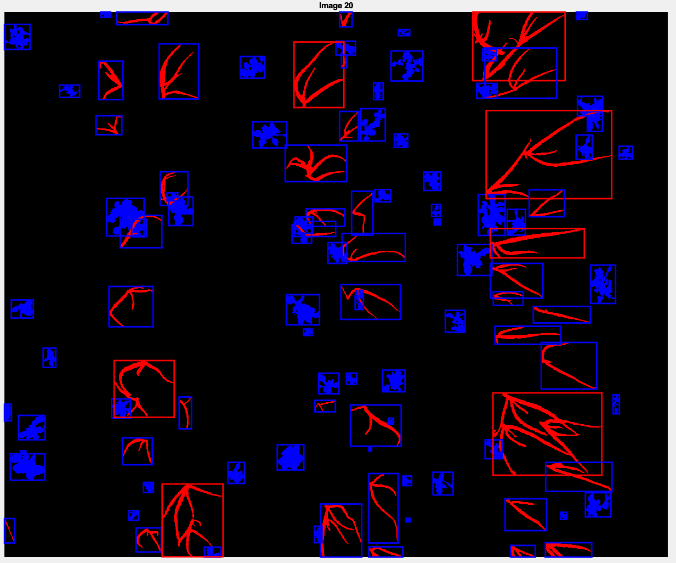


Figure 7 Reduced Model Predictions

## 3A

The code for this section is “Task3Run”, “kalmanTracking”, “kalmanPredict”, “kalmanUpdate”.

A Kalman filter implement a recursive cycle of predicting and updating values based on a previous observation returns. The filter requires a set of parameters including the delta time, number of samples, Constant velocity (CV) motion model, motion noise, Cartesian observation model, observation noise, initial state, and initial state covariance, these parameters are based on previous observation, known elements of the model or defaults for no knowledge of the model. The delta time determines the time between measurements and is a component in the CV motion model. The number of samples is based on the data. The CV motion model parameters are set based on the model itself. The motion noise is based on the constant velocity model. The cartesian observation model and related noise are mathematical values meant to represent an observational sensor. The initial state and covariance are both unknown therefore are set to 0 and the motion noise respectively.

“kalmanTracking” is the main function where the related prediction and update function are called from as well as the parameters initialised. We iterate through the number of samples, calling the predict function, then the update function before saving the state and continuing to the next sample. The “kalmanPredict” function takes as arguments the current state, the covariance matrix, the CV motion model and the related motion noise. This function will return two values, a predicted state and predicted covariance matrix. The predicted state is calculated by multiplying the current state and the covariance matrix. The predicted covariance matrix is calculated by multiplying together the CV motion model with the covariance matrix and the conjugate transpose of the CV motion model before adding the motion noise.

These returned values are then used as arguments for the “kalmanUpdate” function in conjunction with the Cartesian observation model, observation noise and the coordinates for the sample. The function first calculates the innovation covariance by multiplying together the Cartesian observation model with the predicted covariance matrix and the conjugate transpose of the Cartesian observation model before adding the observation motion noise. It then calculates the Kalman gain using the predicted covariance matrix multiplied by the conjugate transpose of the Cartesian observation model which is multiplied by the inverse of the innovation covariance. It then makes a predicted observation using the Cartesian observation model multiplied by the predicted state.

With validation gating a gate is calculated based on the predicted observation, the sample coordinates and the inverse of the innovation covariance. This gate is used against a threshold, values that are larger than the threshold are not accepted and are treated as outliers, samples such as this are not used in updating the state or covariance. Without validation gating the function will return two values, an estimated state and covariance. The estimated state is calculated using the predicted state added to the kalman gain which is multiplied by the sample coordinates minus the predicted observation. The estimate covariance is calculated using the predicted covariance minus the Kalman gain, multiplied by the innovation covariance multiplied by the conjugate transpose of the Kalman gain.

The respective state and covariance are updated for future samples and the state is saved in an array. Once all samples are completed, we will take only the x and y coordinates stored in the array to return back to the “Task3Run” file. The real, predicted and noisy coordinates can be seen on figure 8.

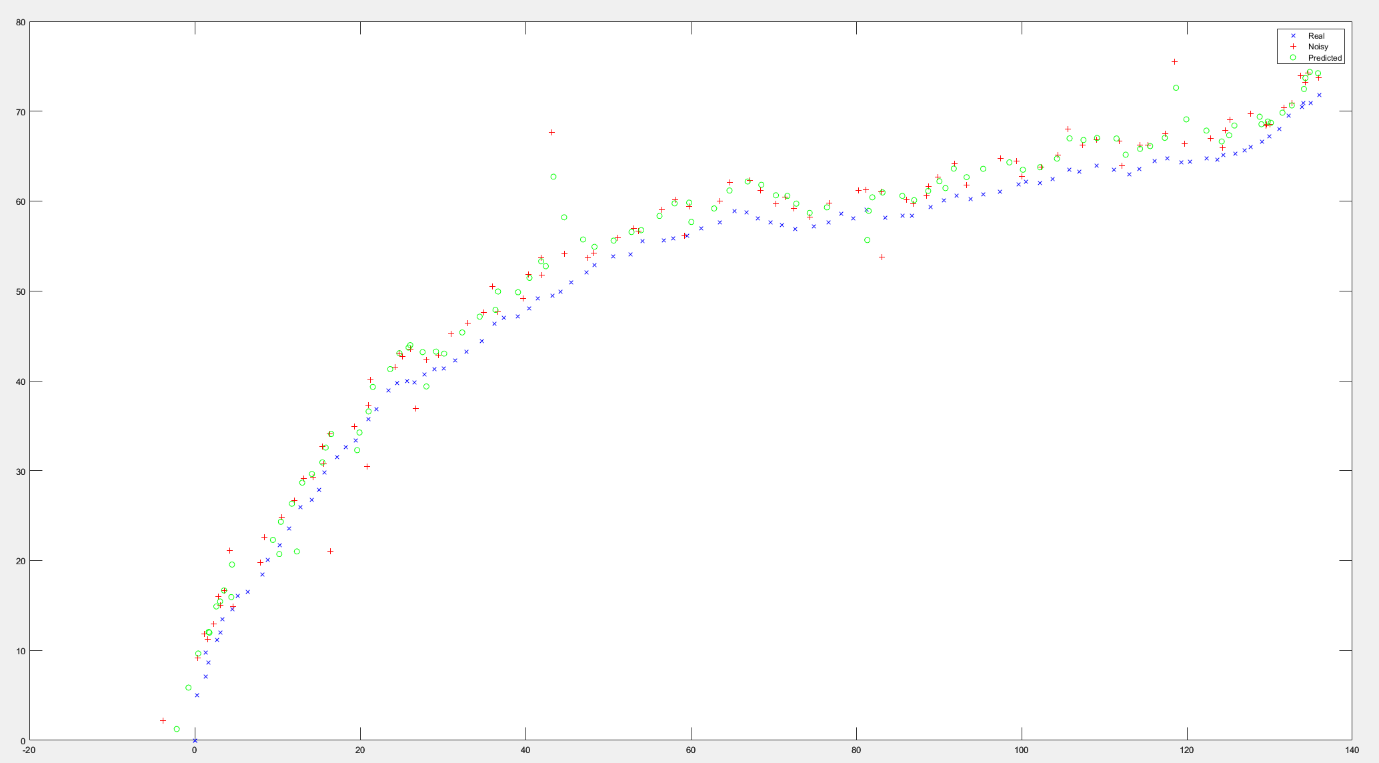


Figure 8 Kalman Filter plot

## 3B

The code for this section is “Task3Run”, “calculateError”.

We can calculate the mean and the standard deviation of the error as well as the root mean squared error (RMSE) for quality analysis of the tracking, table 3. To calculate the error, we use:

Where x and px represent the real x coordinates and the measured x coordinates and y and py measure the same respectively. The RMSE we can calculate using:

|  |  |  |  |
| --- | --- | --- | --- |
| Coordinates | Mean | Standard Deviation | RMSE |
| Noisy | 3.1637 | 2.2069 | 3.8511 |
| Predicted | 2.8945 | 1.569 | 3.2887 |

Table 3 – Statistic for kalman tracking

From this table we can deduce that the kalman filter did perform better on all metrics than the comparison between the noisy coordinates and the real coordinates.

# References

Mathworks (2023). *Create Gray-Level Co-occurrence Matrix for Grayscale Image*. [online] Available at: https://uk.mathworks.com/help/images/ref/graycomatrix.html [Accessed 7 May 2023].

Mathworks (2023). *Calculate Statistics from Gray-level Co-occurrence Matrix*. [online] Available at: https://uk.mathworks.com/help/images/ref/graycoprops.html [Accessed 7 May 2023].

‌

# Appendix

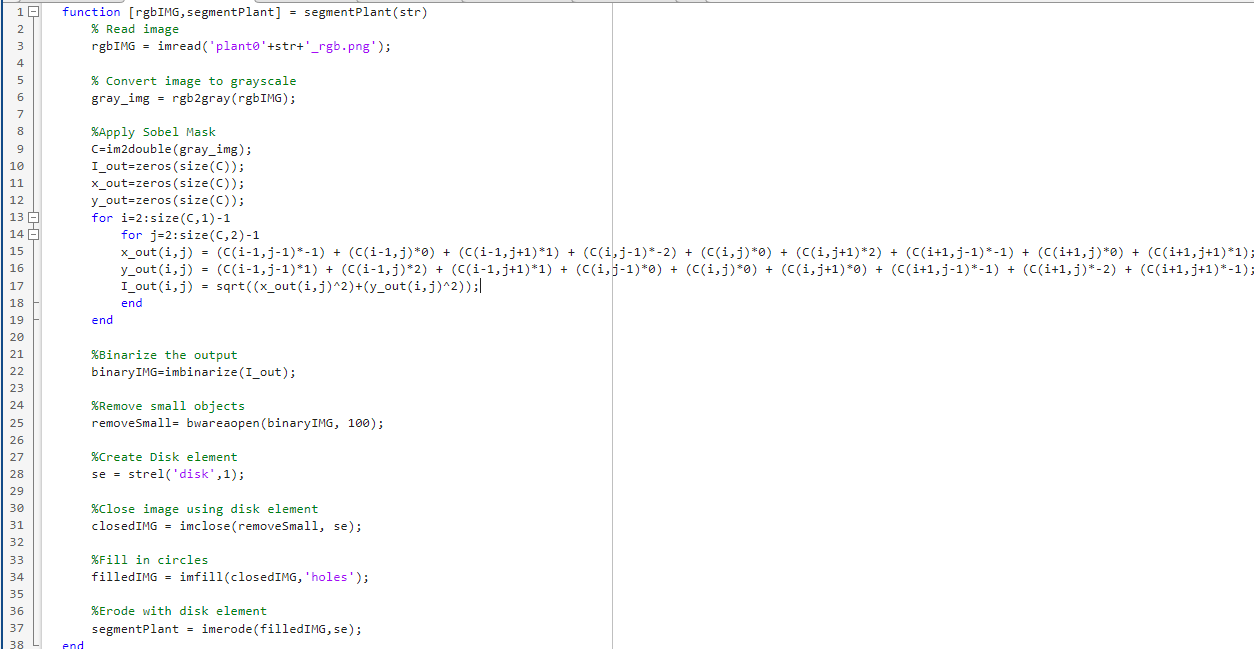


Figure 9 "segmentPlant" Source Code

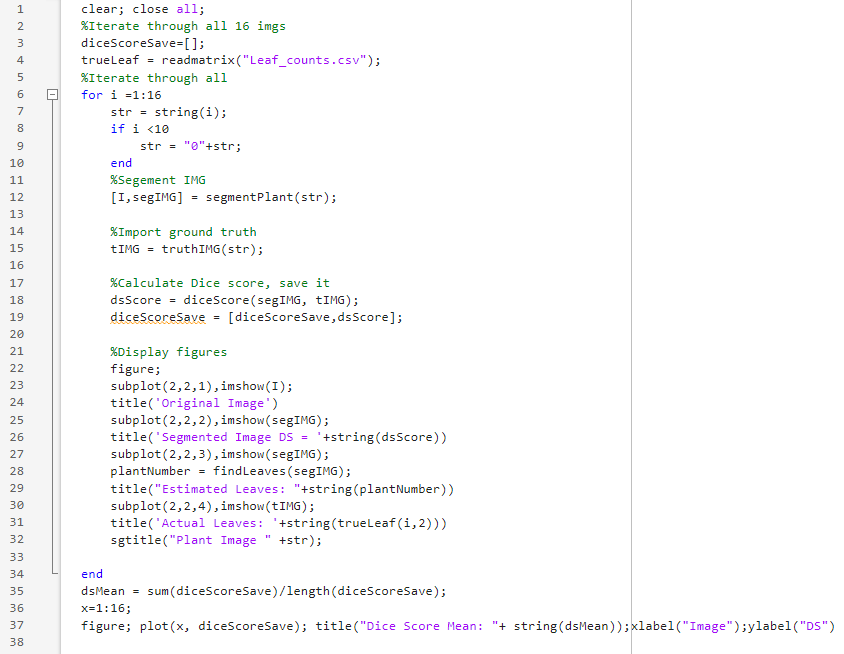


Figure 10 "Task1Run" Source Code

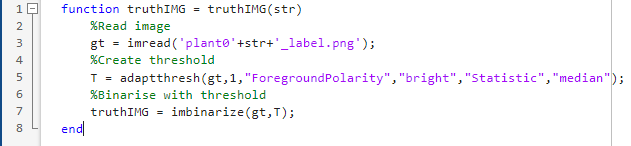


Figure 11 "truthIMG" Source Code

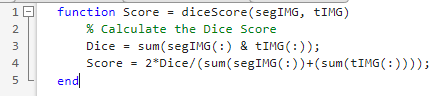


Figure 12 "diceScore" Source Code

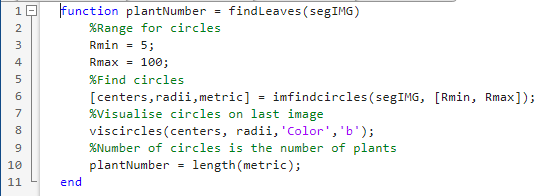


Figure 13 "findLeaves" Source Code

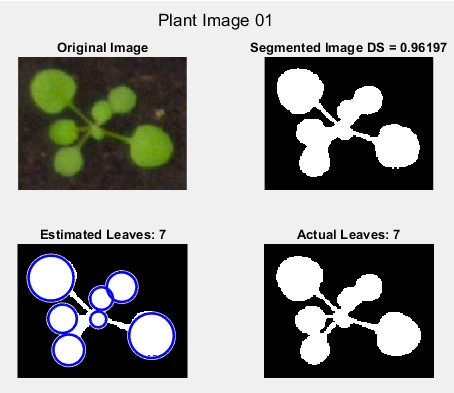


Figure 14 Plant Images 1 And 2

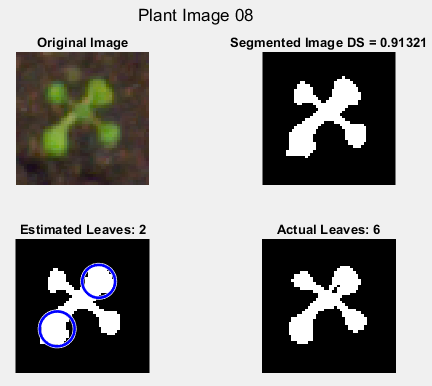
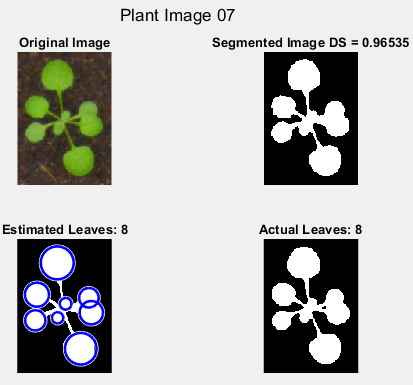


Figure 15 Plant Images 7 And 8

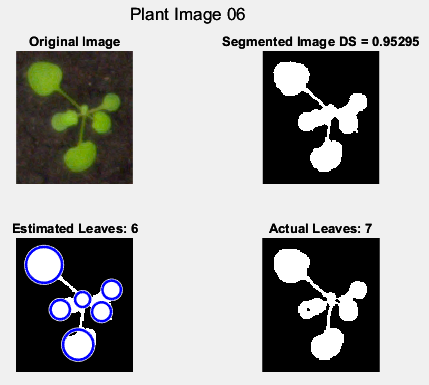
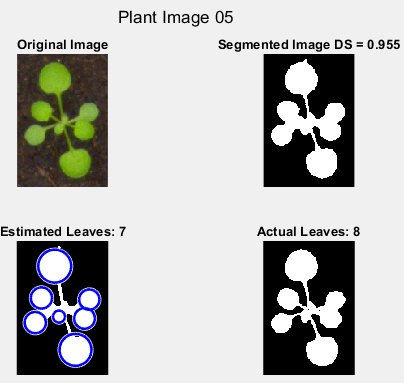


Figure 16 Plant Images 5 And 6

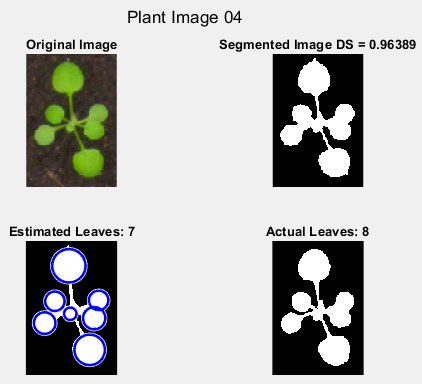
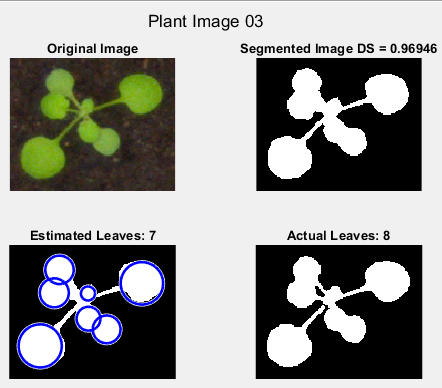


Figure 17 Plant Images 3 And 4

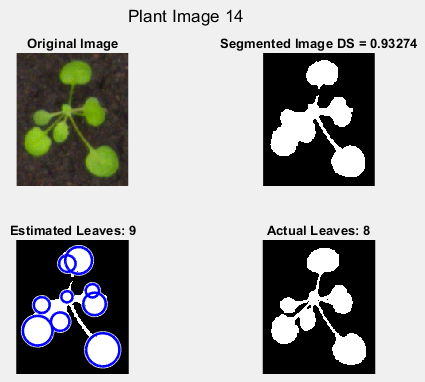
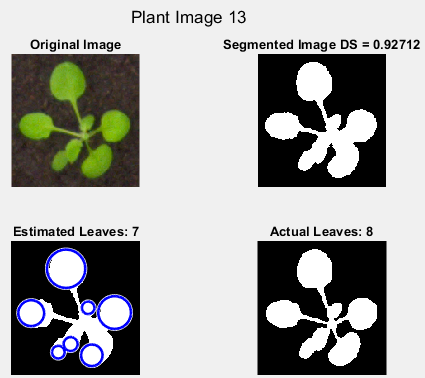


Figure 18 Plant Images 13 And 14

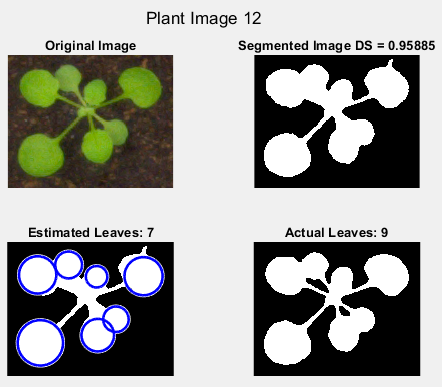
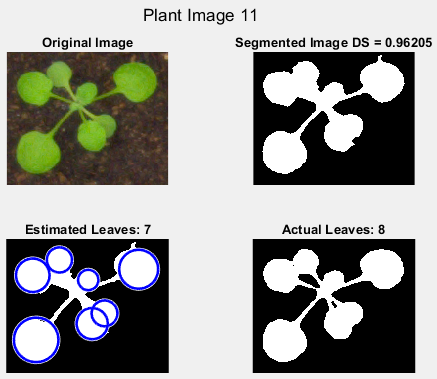


Figure 19 Plant Images 11 And 12

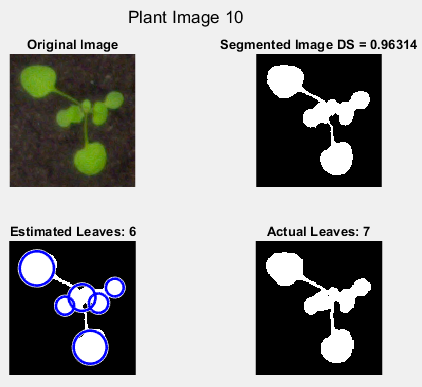
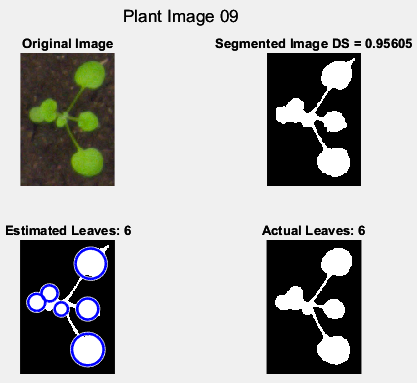


Figure 20 Plant Images 9 And 10

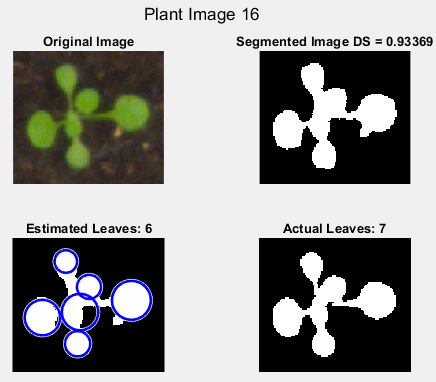
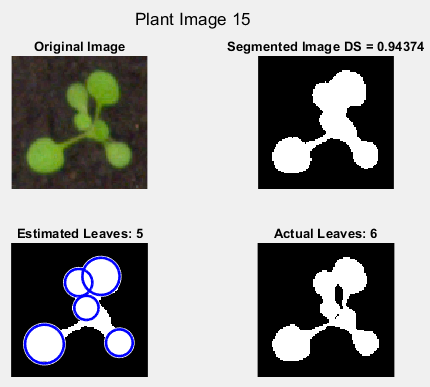


Figure 21 Plant Images 15 And 16