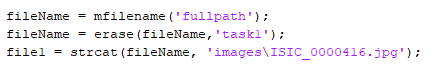
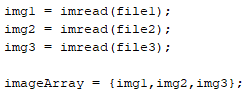
# Computer Vision Assessment Item 1

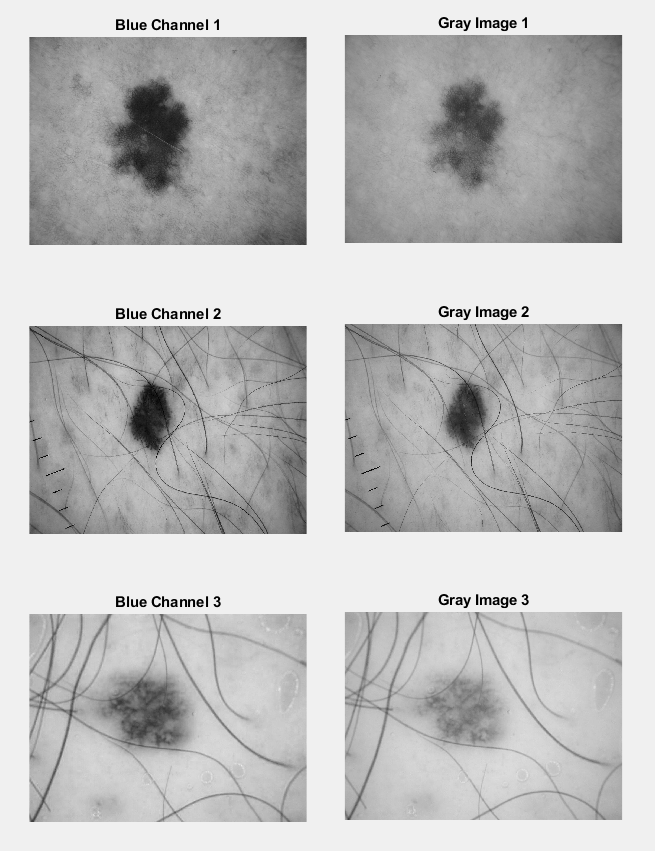
## Task 1

For this task, we were required to separate the skin legion from the background of the image, as well as any external features, such as hairs or shadows, to isolate the legion for further viewing. This boils down to an image segmentation problem, which can be solved in a variety of different ways, such as thresholding and region growing, however, I shall be using morphological operations in order to achieve a high accuracy with these images. Other methods could’ve achieved a higher average accuracy on a larger data set, however for these specific images, a good set of parameters was found to segment all three of the legions.

The first step was to read in the images, using the code shown below, which gets the path of the folder containing the images, then adds them all to an array of the images, so that the operation can be performed on all the images.



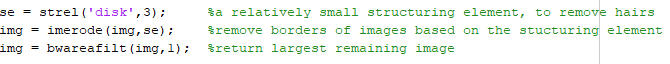


Once the images were loaded, I needed to convert the image into a single dimension (as opposed to RGB). Usually, this would be done by converting to greyscale, however I noticed that converting to the blue channel of the image made the legions more identifiable, as shown in the images to the left, and so I used that so that binarizing the image wouldn’t retain as much of the background. The code below shows how this was achieve; the blue channel was the third dimension of the dataset, the image was then binarized (split to black and white) based on a threshold of 0.51, and then had black and white flipped with the last line to make it match the ground truth.



Now that the image was in the format I needed, I could start to edit it to separate the legion from the background. First, I created a small structuring element, in order to remove the fine hairs around the image. I then used this to perform an ‘imerode’, which removes the area on the edge of the elements inside the image, based on the size and shape of the structuring element. A disk was used because the legion is round, and using something with straight edges would create flat edges on it, which would alter its original shape.

Now that the hairs were removed, I was able to use the function ‘imareafilt’, which returns the single largest element in the image. It should be noted that if any hairs are still left touching the legion at this point, they would be considered part of the legion, which would obviously lead to a reduced accuracy. However, as a result of the earlier eroding, the images have all now been segmented from the background, which technically means the program has already succeeded in its aim – albeit with a low dice score (this would yield a dice score of 0.91 for the first two images (in the order of earlier diagram), however image three would have a score of 0.62, which is too low). The code for this stage is shown below:



From here, I made small adjustments to the images to improve the score. Firstly, I used ‘imfill’ to fill in any holes in the image. I then created a second structuring element, with a much larger size than the first, to increase the size of the legions. This is to rebuild the legion, as the binarizing and the eroding have taken off several layers of it. Although it won’t keep the exact details it had before, this method will retain the overall shape and size of the original image. The code for this is shown below, with a diagram showing each of the stages in figure 1 of the appendix. The dice score for the images now is 0.97, 0.86 and 0.83, showing a marked increase in the last images dice score as a result of the dilation. The dice score is calculated at the end, using Matlabs inbuilt function ‘dice’, which performs the calculation itself, using two matrixes of doubles, and providing a value between 0 and 1 which represents the percentage of accuracy.



## Task 2

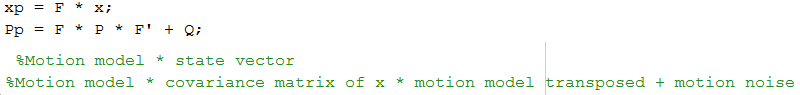
For part one of this task, we were asked to read in an image, and perform spectral feature extraction, in order to describe the texture of the image, or part of the image.

## Task 3

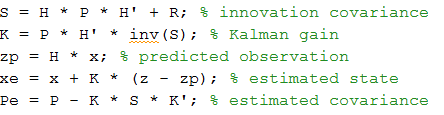
For this task, we were asked to estimate the trajectory of a moving target, by using Kalman filtering, based on a dataset of real and noisy coordinates.

Kalman filtering is a Recursive Bayesian Estimator, a very popular technique used for target tracking. Recursive Bayesian Estimators have two steps: predicting (to compute the prior probability) and updating (to calculate the posterior probability). The posterior probability is calculated by finding the probability of a target state, given the set of observations. Any type of Recursive Bayesian Estimator has to abide by 2 assumptions; the target state can only depend on the previous target state (1st order Markov assumption) and the observation can only depend on the current target state (sensor Markov assumption).

If the model is linear, and noise has a Gaussian (normal) distribution, a Kalman Filter should be used, which is why it is used for this task. A Kalman filter performs a series of matrix operations, using the state vector, the observation vector, the matrix of the observation model, the matrix of the observation noise, the innovation matrix, the Kalman gain and the covariance matrix of the state vector.

The code to implement the Kalman filter consists of three functions; Kalman Tracking, Kalman Predict and Kalman Update. Kalman Tracking is the script which manages the algorithm, and iteratively Predicts and Updates based on the number of samples. As mentioned earlier, the first stage is the Prediction, which is a matrix operation shown in the code below.

As shown in the comments, this is a matrix operation which returns ‘xp‘ as the predicted state, and ‘Pp’ as the predicted state covariance to Kalman tracking, and then gets used in the Kalman Update function, which uses ‘xp’ as the state vector, and ‘Pp’ as the covariance matrix of x, in order to make an update to the tracking. The code for Kalman Update is shown below. As you can see, the innovation covariance is calculated by the matrix of the observation model \* the covariance matrix of x \* the transposed matrix of the observation model + the matrix of observation noise. To get the Kalman gain, I multiply the covariance matrix of x by the transposed matrix of the observation model, and then times this by the inverse matrix of the innovation covariance. To get the predicted observation, I multiplied the matrix of the observation model by the state vector. The formula for the estimated state is the state vector + the Kalman gain \* the observation vector – the predicted observation. Finally, to get the estimated covariance, I did the covariance matrix of x minus the Kalman gain, multiplied by the innovation covariance and the transposed Kalman gain. The estimated state and covariance are then returned to Kalman Tracking, which stores the estimated state as the predicted path, and used the estimated covariance for the next iteration of the prediction.



## Appendix

### Figure 1

