# Lazy Snapping Image Segmentation

Maximum amount of code was completed in different functions that made it easier to call the required functions whenever necessary. It also made it easier to run a specific part of the program when building. The start running from the main function. Firstly, **coordinates** function is called in order to get the x and y coordinates of the seed image. Since we needed to separate the foreground and background from the colour of the strokes, we noticed in the seed image array that blue colour has a value of [255, 0, 6] while that of red colour is [0, 0, 255] in BGR (OpenCV RGB format) format. To separate the foreground, we simply passed the 255 to the **coordinates** function. For the background, we passed 6 as a separating characteristic to the function. After getting the coordinates, we passed them into the **k_means** function that returns three clusters namely red, blue and green. a = len(x)/k gives us the length which is then used in the for loop as gap at which to place centroids for RGB centres. Then these centroids and x, y coordinates are sent to the **clustering** function. This function takes the difference of each pixel in image from each centroid and stores the minimum of each pixel and its centroid distance in clusters array. Program then goes back to the **k_means** function which then updates the centroids inside the while loop. Update only take place if the old and new centroid difference is more than 2. Difference value 2 basically controls the number of iterations during which the centroids will be updated. When k has reached its maximum value, the while loop will break. These clusters are then passed to the **weights** function which gives the weighted sum of the clusters as the portion of the seed pixels in the clusters among the total seed pixels. The weights are then passed to the likelihood part of the code which is within the main function. The reason we did not make a different function of the likelihood part was that there were a lot of variables passed into the function which only increased the number of lines of codes. Therefore, we simply did the likelihood within the main function. We used a tqdm in the first for loop of the likelihood algorithm in order get the loading bar in the output terminal just so we can track the progress. In the likelihood part, first two loops are same as before, two access each particular pixel of the image and then there is a third loop for each centroid. Inside that loop, there is simply the implemented equation for the likelihood of the foreground and background separately. If probability of the front is higher the resulting image pixel is set to 255 and 0 if the probability is true other way around. Last two for loops are two apply the resulting black and white segmented image to the original image to visualise better.


Image

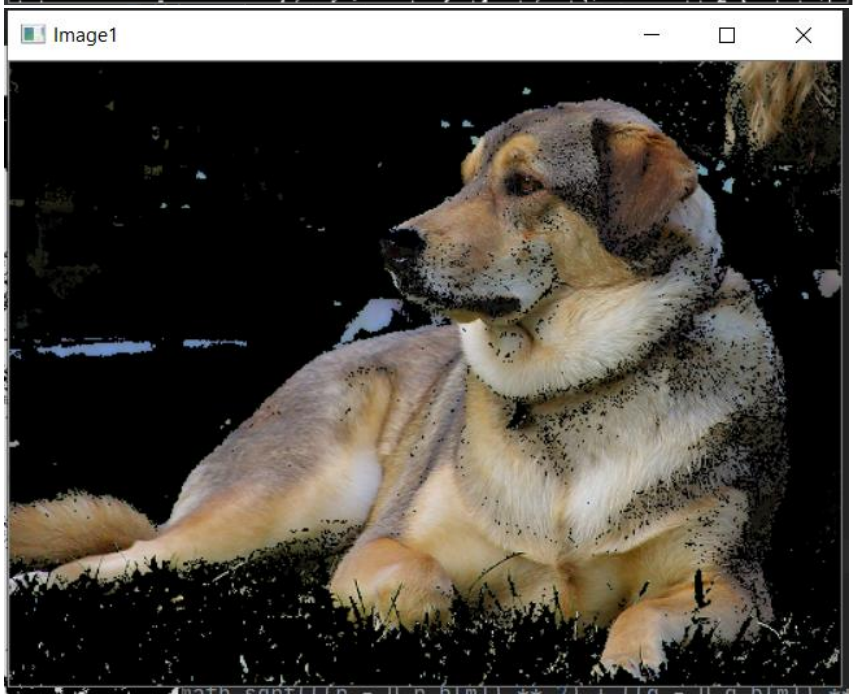
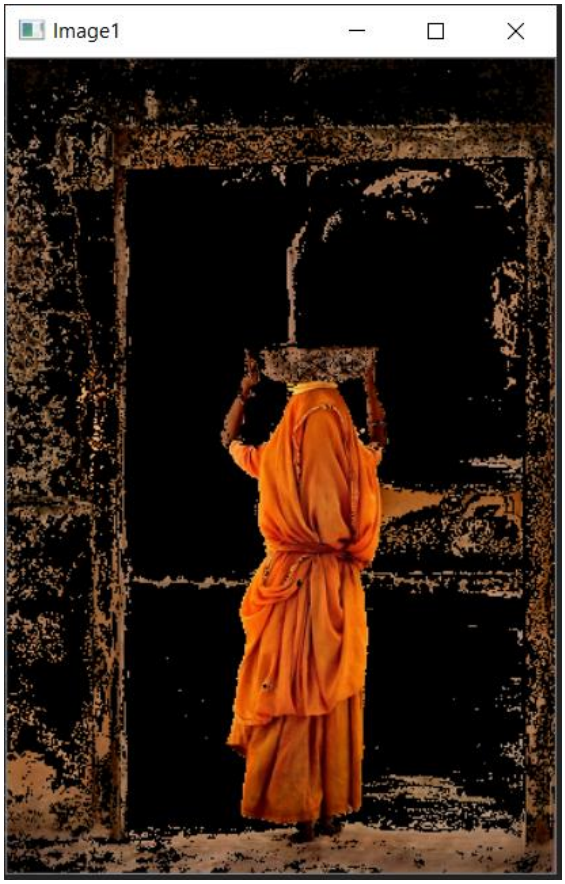


Image1

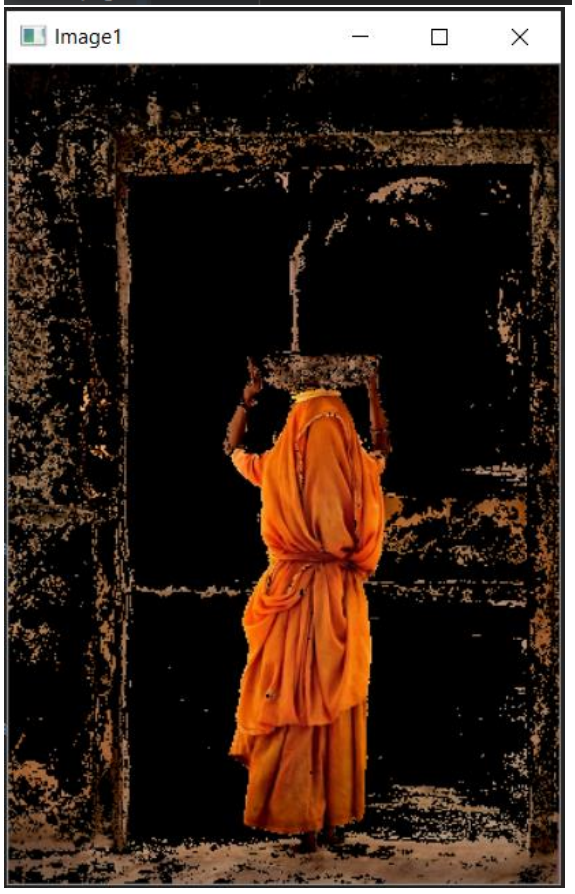
# Using stroke 1


# Using stroke 2

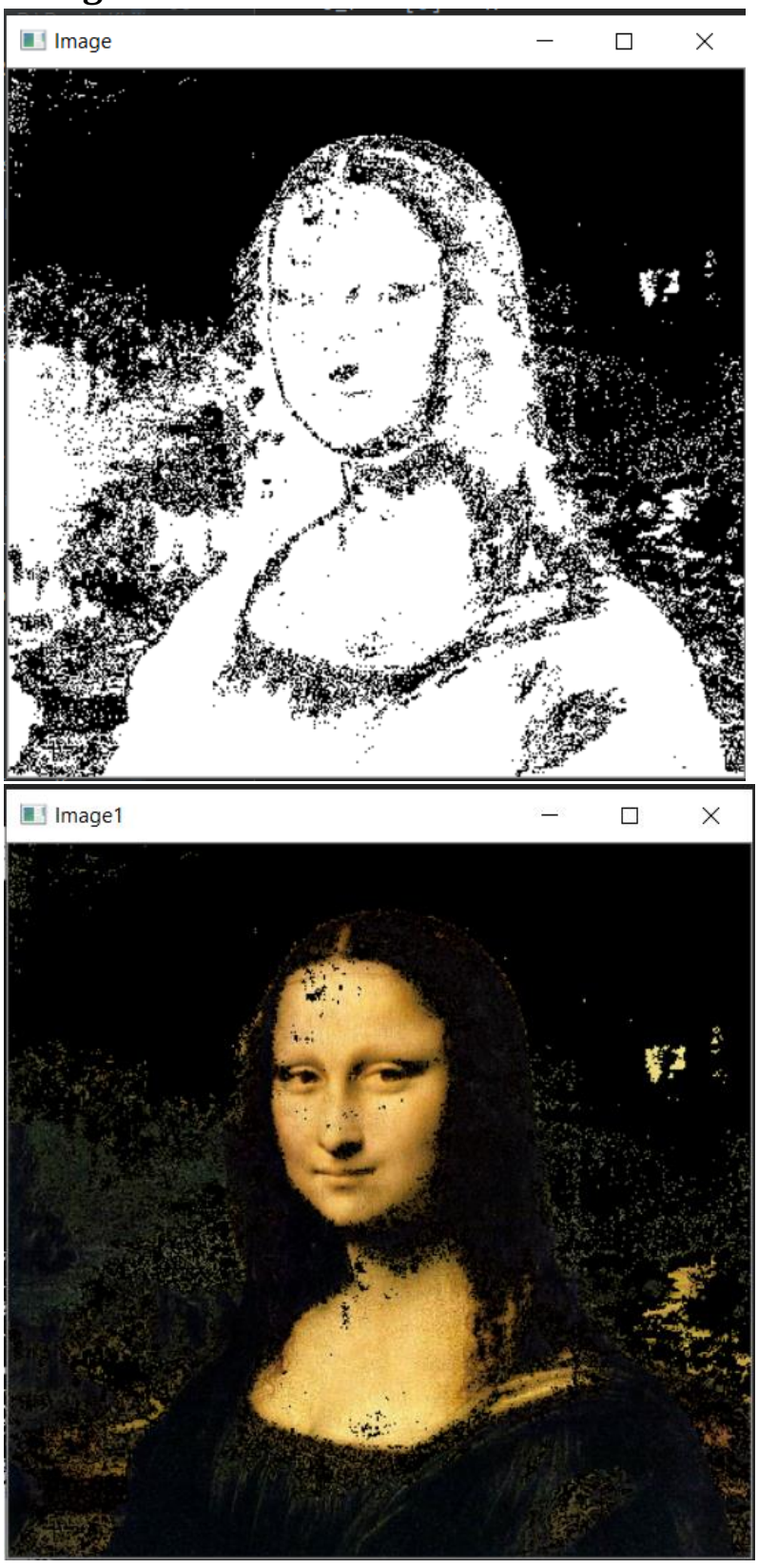

# Using stroke 1

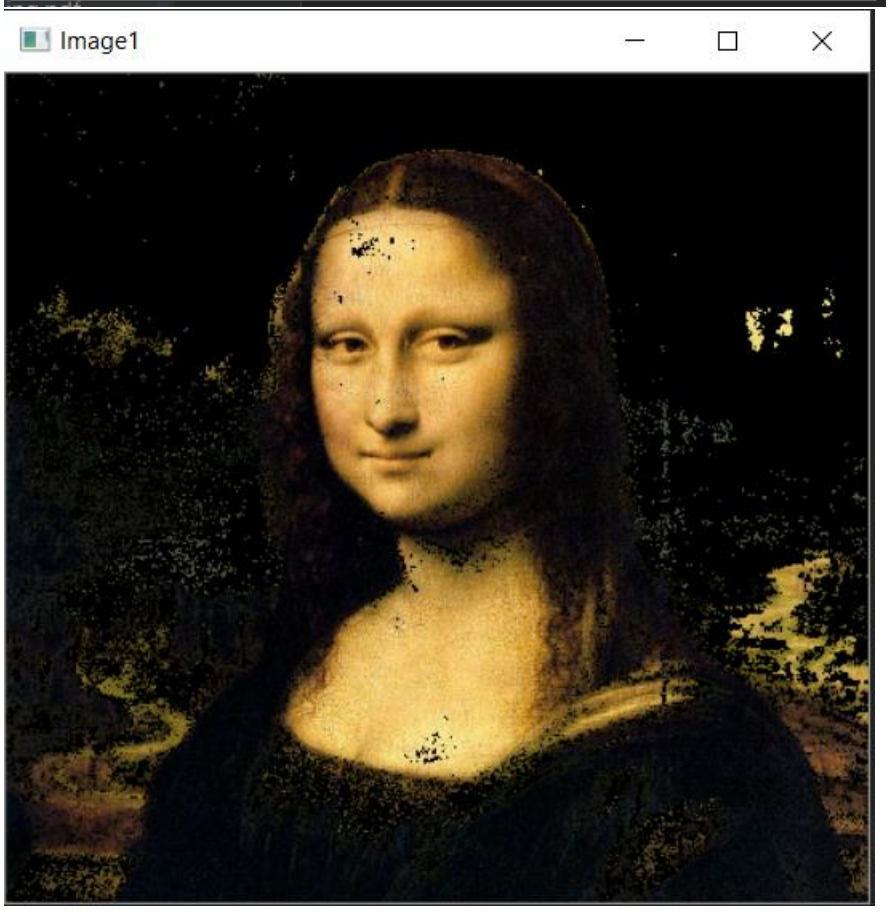
# Using stroke 2

# Using stroke 1



## Using stroke 2


Image

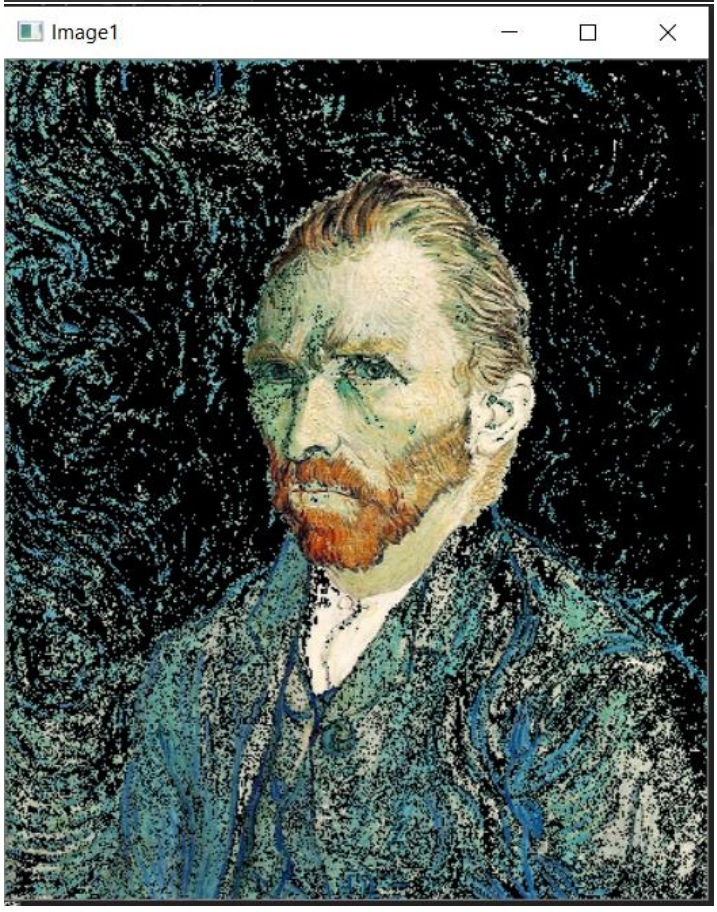


Image1

# Conclusions

In our case, we have used K=64 as it gave the most optimised results. We also tried at K=128 which made the program considerably slow up to 13 minutes. While running program at K= 32 gives the unsatisfactory results such as removing the women's face in the dance.png image. At K=64, the run time is roughly 5 minutes which is optimum for the results generated. The most accurate results were generated in case of a dog image. It is due to the reason that the image background and foreground pixel colours are quite distinct. If the pixel colours of foreground and background are similar the resulting segmentation is not satisfactory as in the case of the Van Gogh image. The use of different strokes is clearly visible in case of the dance image. In which there is a much darker area below her arms and on the ground when stroke 1 image is used. The Mona-Lisa and Lady images gave similar results when different stroke images were used.