For this lab a mean shift algorithm needed to be implemented. When called, 3 variables are set, the size of the image (number of pixels) and the toggle and flip and a size two Pixel pointer array (Pixel *image[2]). One of the 2 is initialized to the values of the array and across the loops, the toggle is used to switch between the current and future array. This is simply doe with bitwise xor: toggle ^= flip. In actuality, toggle and flip are initially 1 and when xor-ed toggle will go between 0 and 1, giving us a flip flopping index for the image array.

The algorithm used goes as follow, the image are all dots in a 3d matrix. To save on computations, the radius of the pixel to compute around the initial position can be set, as opposed to calculate the whole of the image but setting to the size of the greatest dimension will include the whole of the image. Then for all the the pixel in radius, the weight is calculate with the Gaussian distribution with the distance and the kernel bandwidth. This weighted mask is added to the mean and at the end averaged out to set the future pixel. Then the current is flipped with the future and the algorithm starts again until the maximum movement of a single pixel goes to 0.25 or the maximum iteration is reached. This achieve the goal by having every points in the image go toward the densest area in the image. Then, these points that moved, again move towards the densest region again. Since all the points are mapped to a pixel in the image, the values of their final position is copied back in the image.

Since the program takes a significant time to run, the program prints the max distance as well as the old and new point values. This enables the user to see the progress in real time and to see if any infinite looping is occurring (although it will be stopped one day with max iteration).

In practice the amount of computations required with this approach are high, but even with as little as 3 iteration of the algorithm, most important segments are well separated as we can see with the plane picture (figure 1), where , the only remaining pixels are on the bottom left. At this point a simple linear filter can be applied to make the background uniform. This method produces significantly better results than the mean shift as it does not suffer from the artifacts of the large rectangles like we can see in figure 2.

At first glance bounding the radius to only a few around the staring position I thought might’ve led to bad results since a single point could never reach a mode out of the the radius, but since all pixels move at once, the ones closest to the mode will become themselves modes, which makes it so that even points outside the modes radius, will reach the true modes. This lead fewer calculations per point, trading it for a few more iteration. See figure 3 which was calculated with a radius of 20 pixels, with around 200 iterations.

As for the filter, it works by taking a pixel, and taking all the values in the radius in an array. Then the array is sorted and the values of the middle +/- the delta are averaged out and returned to the image.

For libraries, math, malloc and stdlib were used. The rest was made for this project. The image’s struct pointer is passed around to avoid the memory overhead needed when doing a calling function and passing an object. To keep the code portable to other operating systems other than linux, no automatic opening of the image was implemented. The image will instead be stored in the root folder as a file called out.<extension> .

To use these function, one needs to enter 4 at the first assignment prompt. Then, choose what type of image is used, the type of filter or segmentation, then the radius for each pixel. Then depending on previous choice, either the bandwidth kernel or the delta of the filter.

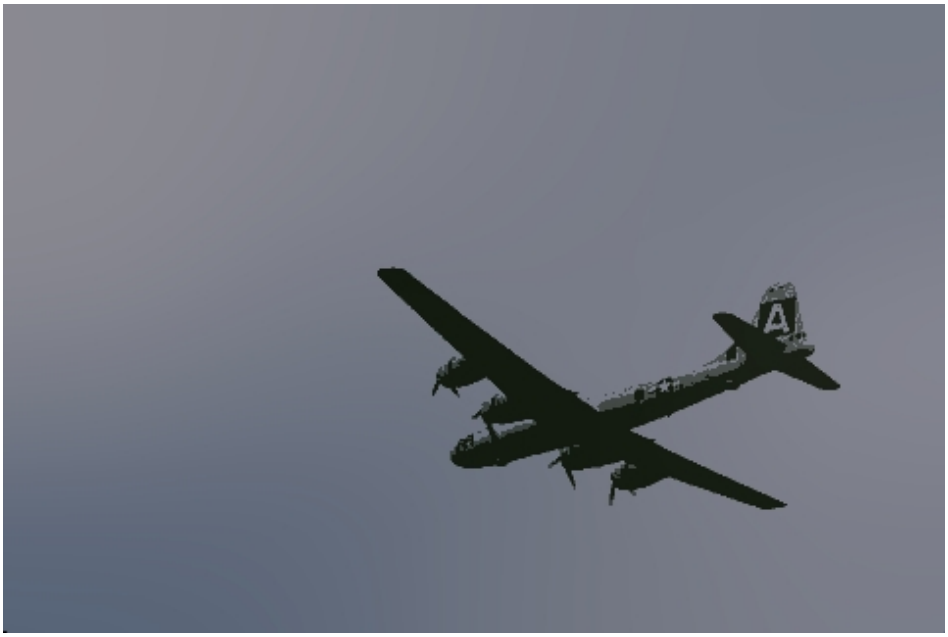

Figure 1. 3 iteration plane

figure 2. kmean plane

figure 3. lena low radius mean shift