2) We have morphed two images here. Below are the input images.

a)

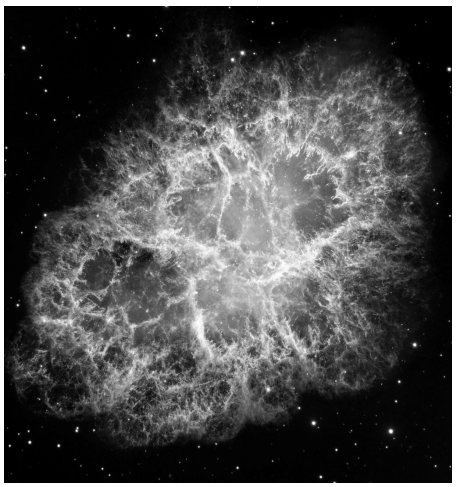
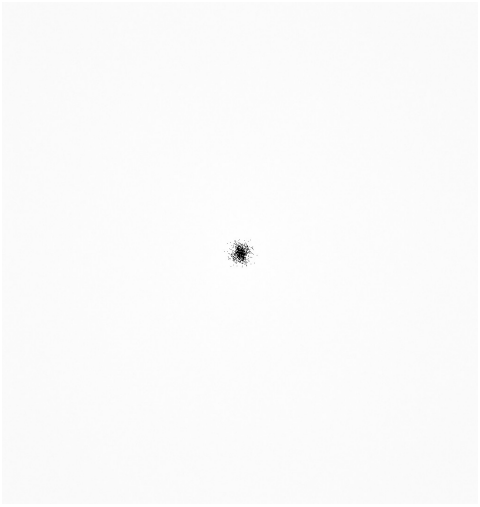
b)

The resulting morphed image:

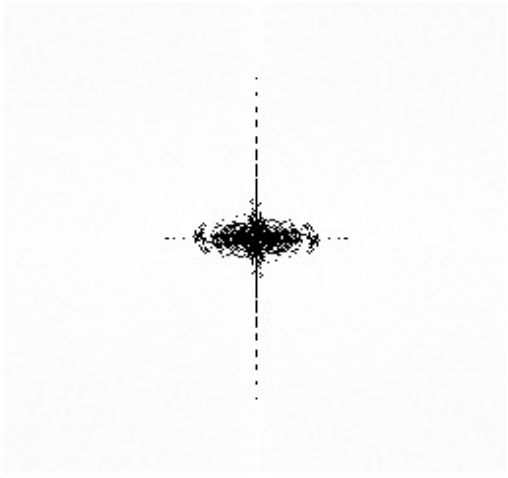
Here, the first image is passed through a low pass filter (gaussian filter with sigma 6), and the second image is passed through a high pass filter (gaussian filter with sigma 3). Note that, both the images must be of the same size for this code to work.

3.1)
The image spectograms of some of the images are presented below.

a)

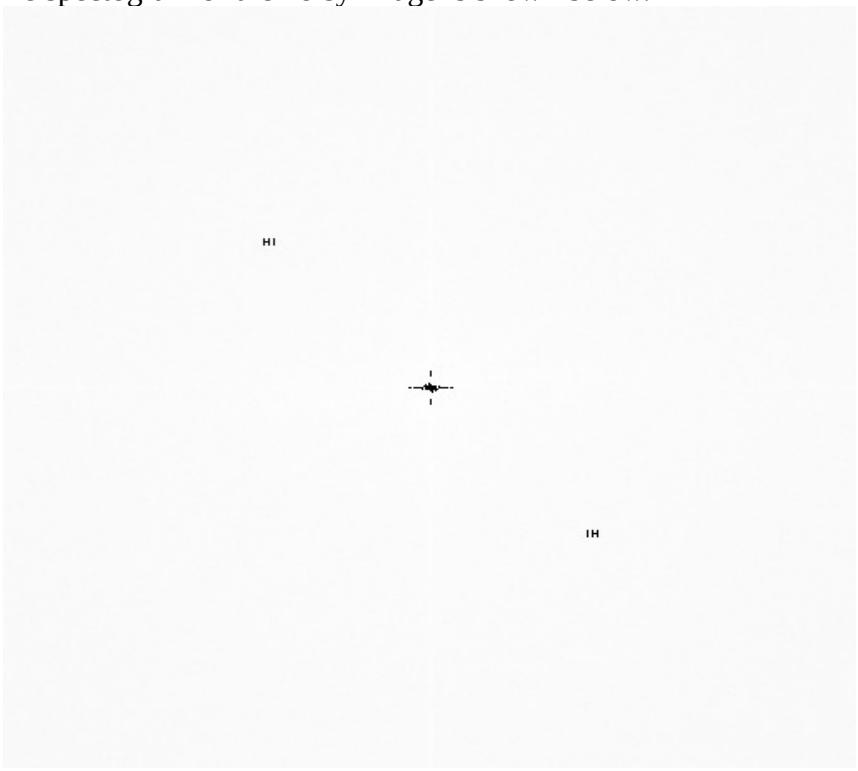

==>


b)


==>


3.2)

The spectogram of the noisy image is shown below.



As, can be seen in the above spectogram, we can see two regions symmetric to origin, which is the added noise. We have identified those two areas, and removed the noise by making the “real” values at those places as zero. We get the below cleaned image.

3.3)

Below are some results of the code. In each example, the left hand side contains the original image, and the right hand side corresponds to the filtered image.

Input Image:

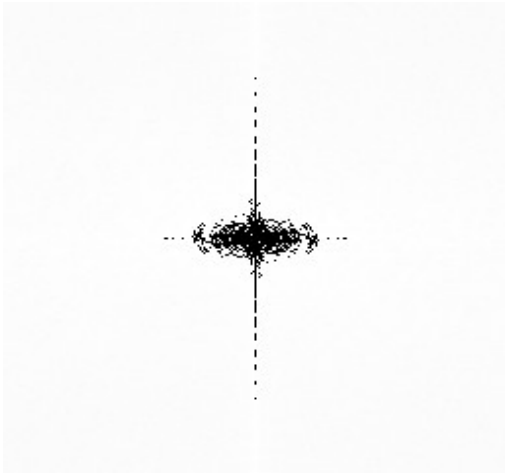
Ouput Images:

1) Sigma =2

2) Sigma =1

Changes in image spectograms:

Input Image and its spectogram:



2) Filtered Image (with sigma 2) and its spectogram:

It can be seen that, the gaussian filter removes the areas far from the origin.

3.4:

The following experiments are done using sigma as 2.

Input Image:

Output Image:

Input Image:

Output Image:

Mean filters also work in a similar way (Convolution and then de-convoltution is implemented as “3.4a” in the code.)

4. Watermarking:

For this implementation, we created two methods:

- **mark_image():** Adds a watermark to the DFT of the image and generates a copy of the same image, given the key "**N**". We generate a random binary sequence "V" of size "**L = 10**", using the key "**N**". We place this sequence on "**L**" bins along a circle with radius "**r = 128**" from the center of "**R**", which is the real part of the input image obtained after applying DFT. We use the following formula to do this:

$$R'(x,y) = R(x,y) + \alpha|R(x,y)|v_i$$

where **R'** is the real part of output image with the watermark, **R** is the real part of the original image, "**α = 10**" is a parameter and "$\mathbf{v_i}$" is the $\mathbf{i^{th}}$ element of the binary sequence "**V**".

- **check_image():** Checks whether a watermark is present. This is done by generating the same binary sequence "**V**" using key "**N**" and obtaining the sequence "**C**" from the pixel values at the same positions where the watermark was added in the real part of the image. We the calculate the Pearson Correlation Coefficient to check whether the sequence "**C**" has any linear dependence on the sequence "**V**". If the coefficient is greater than some threshold "**t = 0.3**", the we consider the watermark to be present.

We tested our implementation on two 512x512 gray scale images and observed the following results:

Test Image – 1:

Before Watermarking

After Watermarking (N = 8)

We proceeded to do some quantitative tests on Test Image – 1. We added the watermark by using key "**N = 8**". We

then used the check_image() function to see if the watermark is detected for the correct "**N**", by trying values 1 to 25. We noticed that the watermark was detected 6 times:

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Watermark Detected	T	T	F	F	T	F	F	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	T	T	F

There were 5 false positive among 25 tests. We speculate that this is dependent on the threshold value "**t**" since different threshold values produced different number of false positives. We use the threshold "**t = 0.3**" since it produced the least number of false positives, i.e. 5.

After Editing

Finally, we edited the watermarked image by adding some distortions and again used the check_image() function to see if the watermark is detected. We tried the values 1 to 25 for "**N**" and noticed that the watermark was detected 6 times, with the same number of false positives as detected on the original watermarked image.

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Watermark Detected	T	T	T	F	F	F	F	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	T	T	F

Test Image – 2:

Before Watermarking

After Watermarking (N = 25)

We did similar quantitative tests on Test Image – 2 as well. We marked this image with "**N = 25**" and noticed the following observations after trying values 1 to 25 for "**N**" in check_image(). This time, the watermark was detected 8 times, with 7 false positives:

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Watermark Detected	F	F	F	T	F	F	F	F	T	T	T	T	F	F	T	F	F	F	T	F	F	F	T	T	T

We applied the same tests on the watermarked image after adding distortions to it:

After Editing

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Watermark Detected	F	F	F	T	F	F	F	F	T	F	F	F	F	T	F	F	F	F	T	F	F	F	F	F	T

This time, we noticed that the number of false positives decreased considerably to 4 from 7.

In both the cases, there were about 15 – 20% of false positives before and after distortion of the watermarked image. Hence we proceeded to change the values of the threshold "**t**". We used the value "0.45" for the and conducted more quantitative analysis. This time, we tried the values 1 to 1000 for "**N**" and noticed that for both images there were about 11% false positive. This is a considerable decrease in the false positives. To further decrease the false positives, we can keep changing the values of threshold "**t**", however, as we noticed with "**N**" values **8** and **65**, there is a possibility that the actual watermark might not get detected, given high threshold value "**t**".

5) Two approaches were taken to solve this problem.
Approach-1:
The noisify application gives also the grayscale version as the output file. So, the blurred image is de-convolved with the blurred image, to get the motion kernel. Then, this kernel is used for inverse convolution to get the original image.

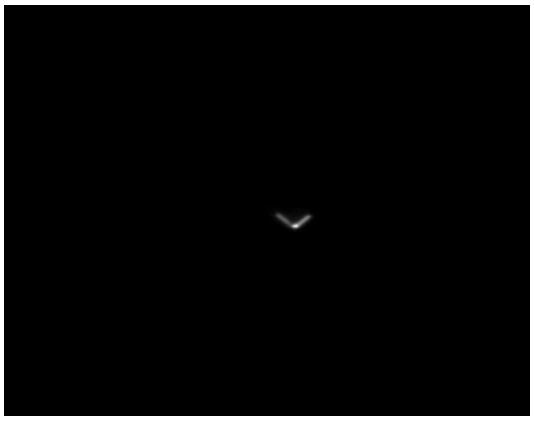
Input Image:

Motion Kernel:

Ouput Image:

Approach -2:
We created 20 sample RGB images of size 255*255. We made sure that all the values in these images are '0', except the middle pixel. Note that, these images work as a identity kernel, and they give the original kernel back on convolution. We have used 20 images to reduce the effects of quantization in the images. We averaged the kernels obtained from those 20 images, and used this for inverse convolution on the new input images. In order to further reduce the noise in the kernel, all values below a certain threshold are removed. (This code is implemented as part 5a and 5b inside main function)

Motion Kernel:



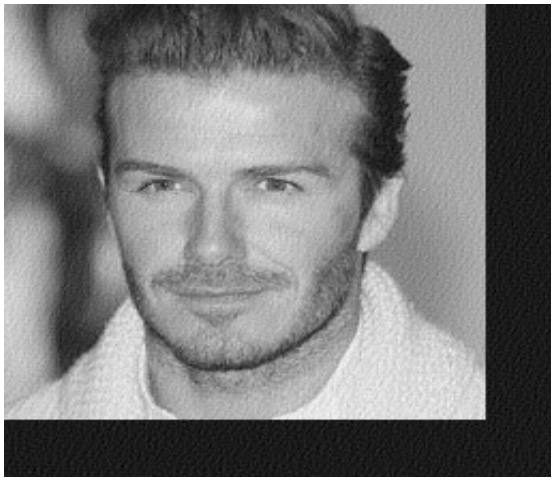
Output Image:

In this case, approach-1 is better than the second. But, it should be noted that approach-1 is not always possible if the original grayscale image is not present for some blurred image.

Effects of Gaussian Noise:

1) Input Image(sigma =0.8)

Output Image



2) Input Image(sigma =0.9)

Output Image