Image Processing (CSPE31) Assignment - I

Image Restoration in Spatial Filtering Implementation

28.03.2020

Ranga Vamsi
106116019

Madhu Babu Naik
106116043

Introduction

Image restoration is the operation of taking a corrupt/noisy image and estimating the clean, original image. Corruption may come in many forms such as motion blur, noise and camera mis-focus. Image restoration is performed by reversing the process that blurred the image and such is performed by imaging a point source and using the point source image, which is called the Point Spread Function (PSF) to restore the image information lost to the blurring process. Image restoration is done in two domains: spatial domain and frequency domain. In this project, we consider the case of restoration of image through spatial filtering.

Spatial filtering is an image processing technique for changing the intensities of a pixel according to the intensities of the neighboring pixels.

Image Restoration Model :

An image can be defined as a two dimensional function I = f(x,y) where x and y are spatial coordinates. (x,y) represents a pixel. I is the intensity or grey level value which is the amplitude of f at any point (x,y). If the values of the coordinates (spatial coordinates) and the amplitude are finite and discrete, then it is called digital image.

The degraded image g(x,y) can be represented as

$$g(x,y) = h(x,y) * f(x,y) + \eta(x,y) \text{ ---> eq.1}$$

where h(x,y) is the degradation function , f(x,y) is the original image, the symbol * indicates convolution and $\eta(x,y)$ is the additive noise.

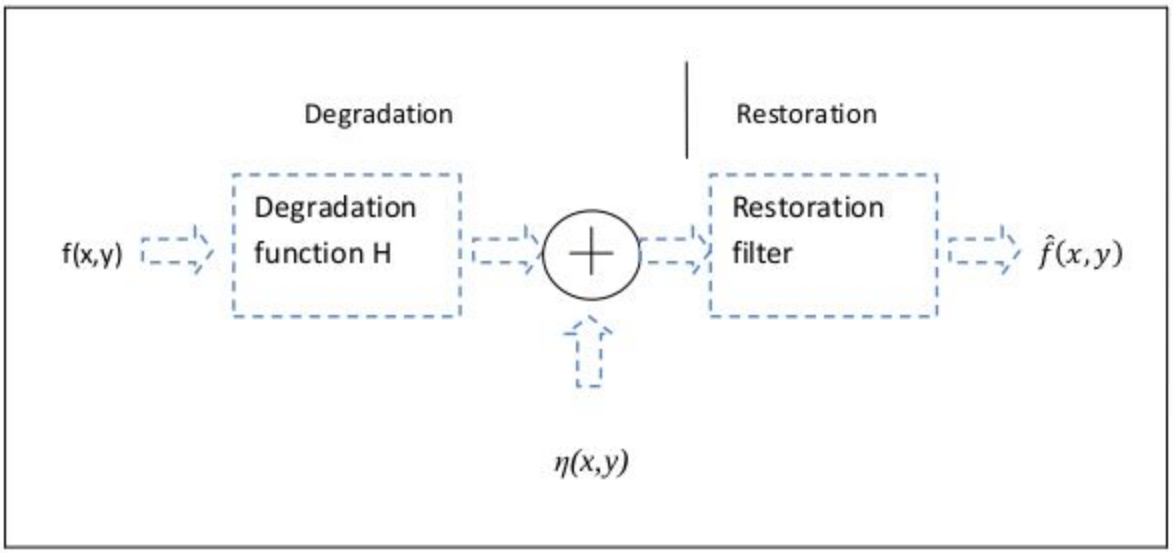
The Blurring Process :

h is a blurring function in eq.1. The process of applying the blur function is convolution, i.e., some path of the image convoles into a pixel of the blurred image. If no noise is added to the system then we can restore the image by simply reversing the convolution process i.e. by using deconvolution. In this project we used **optical blur, gaussian blur, motion blur functions.**

The Noise Process :

η is noise function in eq.1. There can be many reasons for noise in a model such as thermal vibration, magnetic influence, temperature, etc. Noise can be considered as a random process and noise is additive because it is added to noise that might be intrinsic in the system. In this project we used **salt and pepper noise, salt noise, pepper noise, gaussian noise, uniform noise, exponential noise.**

A model of image degradation and restoration process :



Spatial filters used in restoration process :

The main idea behind Spatial Domain Filtering is to convolve a mask with the image. The convolution integral of two functions f(x) and g(x) is defined as

$$f(x) * g(x) = \int_{-\infty}^{\infty} f(u)g(x-u)du$$

In this project we used

1. **Mean filters** (Arithmetic Mean Filter, Geometric Mean Filter, Harmonic Mean Filter, Contraharmonic Mean Filter)

2. **Order statistics filters** (Median Filter, Max Filter, Min Filter, Midpoint Filter, Alpha-Trimmed Mean Filter)

3. **Adaptive filters** (Local noise reduction Filter) for both colored images and gray scale images.

Goals

Implement **Image Restoration** using any three spatial filtering techniques, over a simple image without using in-built functions.

Methods:

1. Arithmetic Mean Filter

Explanation:

This is the simplest of the mean filters. Let Sxv represent the set of coordinates in a rectangular subimage window of size m X n, centered at point (x, y).The arithmetic mean filtering process computes the average value of the corrupted image g(x, y) in the area defined by Sxy.The value of the restored image at any point (x, y) is simply the arithmetic mean computed using the pixels in the region defined by S.

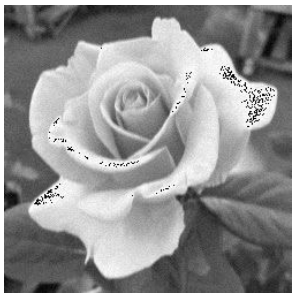
$$\hat{f}(x,y) = \frac{1}{mn} \sum_{(s,t)\in S_{xy}} g(s,t).$$

Input: Distorted image which is a result of the convolution of the perfect image with blurring model and the addition of noise.

Code:

```
def apply_filter(filter_class,img,gray):

    out_img = img.copy()
    print("Input Filter size(odd numbers) : ")
    filter_size = int(input())
    x = (int)(filter_size/2)
    n = filter_size * filter_size

    if gray == "False":
        pad_img = np.pad(img,((x,x),(x,x),(0,0)), mode='constant')
    else:
        pad_img = np.pad(img,((x,x),(x,x)), mode='constant')

    r = pad_img.shape[0]
    c = pad_img.shape[1]

    if filter_type == 1:
        for i in range(x,r-x):
            for j in range(x,c-x):
                if gray == "False":
                    for t in range(0,3):
                        Sum = 0
                        for k in range(-1*(x),filter_size-x):
                            for l in range(-1*(x),filter_size-x):
                                Sum = Sum + pad_img.item(i+k,j+l,t)

                        Avg = float(Sum/n)
                        out_img.itemset((i-x,j-x,t),Avg)

                else:
                    Sum = 0
                    for k in range(-1*(x),filter_size-x):
                        for l in range(-1*(x),filter_size-x):
                            Sum = Sum + pad_img.item(i+k,j+l)

                    Avg = float(Sum/n)
                    out_img.itemset((i-x,j-x),Avg)

        return out_img,filter_
```

Output:



→

Motion blur + uniform noise

Arithmetic Mean filter(3*3)

2. Geometric Mean Filter

Explanation:

Here, each restored pixel is given by the product of the pixels in the subimage window, raised to the power 1/mn. A geometric mean filter achieves smoothing comparable to the arithmetic mean filter, but it tends to lose less image detail in the process.

$$\hat{f}(x,y) = \left[\prod_{(s,t)\in S_{xy}} g(s,t) \right]^{\frac{1}{mn}}.$$

Code:

```
def apply_filter(filter_class,img,gray):

    out_img = img.copy()
    print("Input Filter size(odd numbers) : ")
    filter_size = int(input())
    x = (int)(filter_size/2)
    n = filter_size * filter_size

    if gray == "False":
        pad_img = np.pad(img,((x,x),(x,x),(0,0)), mode='constant')
    else:
        pad_img = np.pad(img,((x,x),(x,x)), mode='constant')

    r = pad_img.shape[0]
    c = pad_img.shape[1]

    for i in range(x,r-x):
        for j in range(x,c-x):
            if gray == "False":
                for t in range(0,3):
                    Mul = 1
                    for k in range(-1*(x),filter_size-x):
                        for l in range(-1*(x),filter_size-x):
                            Mul = Mul * pad_img.item(i+k,j+l,t)

                    Geo = Mul**(1.0/(float(n)))
                    out_img.itemset((i-x,j-x,t),Geo)
            else:
                Mul = 1
                for k in range(-1*(x),filter_size-x):
                    for l in range(-1*(x),filter_size-x):
                        Mul = Mul * pad_img.item(i+k,j+l)

                Geo = Mul**(1.0/(float(n)))
                out_img.itemset((i-x,j-x),Geo)

    return out_img
```

Output:

Motion blur + Gaussian Noise

->

Geometric Mean Filter(3*3)

3. Harmonic Mean Filter

Explanation:

The harmonic mean filter is a member of a set of nonlinear mean filters which are better at removing Gaussian type noise and preserving edge features than the arithmetic mean filter. The harmonic mean filter is very good at removing positive outliers.

$$\hat{f}(x,y) = \frac{mn}{\sum_{(s,t)\in S_{xy}} \frac{1}{g(s,t)}}.$$

Code:

```
def apply_filter(filter_class,img,gray):

    out_img = img.copy()
    print("Input Filter size(odd numbers) : ")
    filter_size = int(input())
    x = (int)(filter_size/2)
    n = filter_size * filter_size

    if gray == "False":
        pad_img = np.pad(img,((x,x),(x,x),(0,0)), mode='constant')
    else:
        pad_img = np.pad(img,((x,x),(x,x)), mode='constant')

    r = pad_img.shape[0]
    c = pad_img.shape[1]

    for i in range(x,r-x):
        for j in range(x,c-x):
            if gray == "False":
                for t in range(0,3):
                    Isum = 0.00000000000001
                    for k in range(-1*(x),filter_size-x):
                        for l in range(-1*(x),filter_size-x):
                            Is = 0.0 if pad_img.item(i+k,j+l,t) == 0 else 1/float(pad_img.item(i+k,j+l,t))
                            Isum = Isum + Is

                    Har = float(n/Isum)
                    out_img.itemset((i-x,j-x,t),Har)
            else:
                Isum = 0.00000000000001
                for k in range(-1*(x),filter_size-x):
                    for l in range(-1*(x),filter_size-x):
                        Is = 0.0 if pad_img.item(i+k,j+l) == 0 else 1/float(pad_img.item(i+k,j+l))
                        Isum = Isum + Is

                Har = float(n/Isum)
                out_img.itemset((i-x,j-x),Har)

    filter_ = "Harmonic_mean_"+str(filter_size)+"*"+str(filter_size)
    return out_img,filter_
```

Output:

->

Gaussian blur + Salt Noise

Harmonic Mean Filter (5*5)

4. Contraharmonic Mean Filter

Expression:

$$\hat{f}(x,y) = \frac{\sum_{(s,t)\in S_{xy}} g(s,t)^{Q+1}}{\sum_{(s,t)\in S_{xy}} g(s,t)^{Q}}$$

where Q is called the order of the filter. This filter is well suited for reducing or virtually eliminating the effects of salt-and-pepper noise. For positive values of Q, the filter eliminates pepper noise. For negative values of Q it eliminates salt noise. It cannot do both simultaneously. Note that the contraharmonic filter reduces to the arithmetic mean filter if Q = 0, and to the harmonic mean filter if Q = - 1.

Code:

```
def apply_filter(filter_class,img,gray):
     out_img = img.copy()
    print("Input Filter size(odd numbers) : ")
    filter_size = int(input())
    x = (int)(filter_size/2)
    n = filter_size * filter_size
    if gray == "False":
        pad_img = np.pad(img,((x,x),(x,x),(0,0)), mode='constant')
    else:
        pad_img = np.pad(img,((x,x),(x,x)), mode='constant')
    r = pad_img.shape[0]
    c = pad_img.shape[1]
    print("Input 'Q' :")
    Q = int(input())
    for i in range(x,r-x):
        for j in range(x,c-x):
            if gray == "False":
                for t in range(0,3):
                    num = 0.0
                    den = 0.00000000000001
                    for k in range(-1*(x),filter_size-x):
                        for l in range(-1*(x),filter_size-x):
                            val = 0 if (pad_img.item(i+k,j+l,t)) == 0 else (pad_img.item(i+k,j+l,t)**(Q))
                            num = num + (val*pad_img.item(i+k,j+l,t))
                            den = den + (val)
                    ratio = float(num/den)
                    out_img.itemset((i-x,j-x,t),ratio)
            else:
                num = 0.0
                den = 0.00000000000001
                for k in range(-1*(x),filter_size-x):
                    for l in range(-1*(x),filter_size-x):
                        val = 0 if (pad_img.item(i+k,j+l)) == 0 else (pad_img.item(i+k,j+l)**(Q))
                        num = num + (val*pad_img.item(i+k,j+l))
                        den = den + (val)
                ratio = float(num/den)
                out_img.itemset((i-x,j-x),ratio)
    filter_ = "Contraharmonic_mean_"+str(filter_size)+"*"+str(filter_size)
    return out_img,filter_
```

Output:

->

Motion blur + Pepper Noise

Contraharmonic Mean(3*3) with Q=3

5. Median Filter

Explanation:

The original value of the pixel is included in the computation of the median. Median filters are quite popular because, for certain types of random noise, they provide excellent noise-reduction capabilities, with considerably less blurring than linear smoothing filters of similar size. Median filters are particularly effective in the presence of both bipolar and unipolar impulse noise.

$$\hat{f}(x,y) = \underset{(s,t)\in S_{xy}}{median}\{g(s,t)\}$$

Code:

```
def apply_filter(filter_class,img,gray):
    out_img = img.copy()
    print("Input Filter size(odd numbers) : ")
    filter_size = int(input())
    x = (int)(filter_size/2)
    n = filter_size * filter_size
    if gray == "False":
        pad_img = np.pad(img,((x,x),(x,x),(0,0)), mode='constant')
    else:
        pad_img = np.pad(img,((x,x),(x,x)), mode='constant')
    r = pad_img.shape[0]
    c = pad_img.shape[1]

    for i in range(x,r-x):
        for j in range(x,c-x):
            if gray == "False":
                for t in range(0,3):
                    List = []
                    for k in range(-1*(x),filter_size-x):
                        for l in range(-1*(x),filter_size-x):
                            List.append(pad_img.item(i+k,j+l,t))

                    out_img.itemset((i-x,j-x,t),median(List))
            else:
                List = []
                for k in range(-1*(x),filter_size-x):
                    for l in range(-1*(x),filter_size-x):
                        List.append(pad_img.item(i+k,j+l))

                out_img.itemset((i-x,j-x),median(List))

    filter_ = "Median_"+str(filter_size)+"*"+str(filter_size)
    return out_img,filter_
```

Output:

->

Gaussian Blur + Salt pepper Noise Median Filter (5*5)

6. Max and Min Filters

Expression:

FORMULA:

$$\hat{f}(x,y) = \min_{(s,t)\in S_{xy}} \{g(s,t)\}$$

FORMULA:

$$\hat{f}(x,y) = \max_{(s,t)\in S_{xy}} \{g(s,t)\}$$

Code:

Max Filter :

```
def apply_filter(filter_class,img,gray):
    out_img = img.copy()
    print("Input Filter size(odd numbers) : ")
    filter_size = int(input())
    x = (int)(filter_size/2)
    n = filter_size * filter_size
    if gray == "False":
        pad_img = np.pad(img,((x,x),(x,x),(0,0)), mode='constant')
    else:
        pad_img = np.pad(img,((x,x),(x,x)), mode='constant')
    r = pad_img.shape[0]
    c = pad_img.shape[1]

    for i in range(x,r-x):
        for j in range(x,c-x):
            if gray == "False":
                for t in range(0,3):
                    Max = 0
                    for k in range(-1*(x),filter_size-x):
                        for l in range(-1*(x),filter_size-x):
                            Max = pad_img.item(i+k,j+l,t) if (pad_img.item(i+k,j+l,t)) > Max else Max

                    out_img.itemset((i-x,j-x,t),Max)
            else:
                Max = 0
                for k in range(-1*(x),filter_size-x):
                    for l in range(-1*(x),filter_size-x):
                        Max = pad_img.item(i+k,j+l) if (pad_img.item(i+k,j+l)) > Max else Max

                out_img.itemset((i-x,j-x),Max)

    filter_ = "Max_"+str(filter_size)+"*"+str(filter_size)
    return out_img,filter_
```

Output:

->

Optical blur + Salt Pepper Noise

Max Filter (3*3)

Min Filter :

```
def apply_filter(filter_class,img,gray):
    out_img = img.copy()
    print("Input Filter size(odd numbers) : ")
    filter_size = int(input())
    x = (int)(filter_size/2)
    n = filter_size * filter_size
    if gray == "False":
        pad_img = np.pad(img,((x,x),(x,x),(0,0)), mode='constant')
    else:
        pad_img = np.pad(img,((x,x),(x,x)), mode='constant')
    r = pad_img.shape[0]
    c = pad_img.shape[1]

    for i in range(x,r-x):
        for j in range(x,c-x):
            if gray == "False":
                for t in range(0,3):
                    Min = 255
                    for k in range(-1*(x),filter_size-x):
                        for l in range(-1*(x),filter_size-x):
                            Min = pad_img.item(i+k,j+l,t) if (pad_img.item(i+k,j+l,t)) < Min else Min

                    out_img.itemset((i-x,j-x,t),Min)
            else:
                Min = 255
                for k in range(-1*(x),filter_size-x):
                    for l in range(-1*(x),filter_size-x):
                        Min = pad_img.item(i+k,j+l) if (pad_img.item(i+k,j+l)) < Min else Min

                out_img.itemset((i-x,j-x),Min)

    filter_ = "Min_"+str(filter_size)+"*"+str(filter_size)
    return out_img,filter_
```

Output :

Optical blur + Salt Pepper Noise

->

Min filter (3*3)

7. Midpoint Filter

Explanation:

In the midpoint method, the color value of each pixel is replaced with the average of maximum and minimum (i.e. the midpoint) of color values of the pixels in a surrounding region. ... The midpoint filter is typically used to filter images containing short tailed noise such as Gaussian and uniform type noise.

$$\hat{f}(x,y)=\frac{1}{2}\left[\max_{(s,t)\in S_{xy}}\{g(s,t)\}+\min_{(s,t)\in S_{xy}}\{g(s,t)\}\right]$$

Code:

```
def apply_filter(filter_class,img,gray):
    out_img = img.copy()
    print("Input Filter size(odd numbers) : ")
    filter_size = int(input())
    x = (int)(filter_size/2)
    n = filter_size * filter_size
    if gray == "False":
        pad_img = np.pad(img,((x,x),(x,x),(0,0)), mode='constant')
    else:
        pad_img = np.pad(img,((x,x),(x,x)), mode='constant')
    r = pad_img.shape[0]
    c = pad_img.shape[1]

    for i in range(x,r-x):
        for j in range(x,c-x):
            if gray == "False":
                for t in range(0,3):
                    Max = 0
                    Min = 255
                    for k in range(-1*(x),filter_size-x):
                        for l in range(-1*(x),filter_size-x):
                            Max = pad_img.item(i+k,j+l,t) if (pad_img.item(i+k,j+l,t)) > Max else Max
                            Min = pad_img.item(i+k,j+l,t) if (pad_img.item(i+k,j+l,t)) < Min else Min

                    out_img.itemset((i-x,j-x,t),(Max+Min)/2)
            else:
                Max = 0
                Min = 255
                for k in range(-1*(x),filter_size-x):
                    for l in range(-1*(x),filter_size-x):
                        Max = pad_img.item(i+k,j+l) if (pad_img.item(i+k,j+l)) > Max else Max
                        Min = pad_img.item(i+k,j+l) if (pad_img.item(i+k,j+l)) < Min else Min

                out_img.itemset((i-x,j-x),(Max+Min)/2)

    filter_ = "Midpoint_"+str(filter_size)+"*"+str(filter_size)
    return out_img,filter_
```

Output:

->

Optical blur + Salt Pepper Noise

Midpoint Filter (3*3)

8. Alpha-trimmed Mean Filter

Explanation:

To define the alpha-trimmed mean filter, all pixels surrounding the pixel at the coordinate (x,y) in the image A which are specified by an input N x N size square mask A(i) are ordered from minimum to maximum value.

Formula:

$$\hat{f}(x,y) = \frac{1}{mn-d} \sum_{(s,t)\in S_{xy}} g_r(s,t)$$

where $g_r(s,t)$ represent the remaining mn-d pixels after removing the $d/2$ highest and $d/2$ lowest values of $g(s,t)$.

Code:

```
def apply_filter(filter_class,img,gray):
    out_img = img.copy()
    print("Input Filter size(odd numbers) : ")
    filter_size = int(input())
    x = (int)(filter_size/2)
    n = filter_size * filter_size
    if gray == "False":
        pad_img = np.pad(img,((x,x),(x,x),(0,0)), mode='constant')
    else:
        pad_img = np.pad(img,((x,x),(x,x)), mode='constant')
    r = pad_img.shape[0]
    c = pad_img.shape[1]

    for i in range(x,r-x):
        for j in range(x,c-x):
            if gray == "False":
                for t in range(0,3):
                    Sum = 0
                    for k in range(-1*(x),filter_size-x):
                        for l in range(-1*(x),filter_size-x):
                            Sum = Sum + pad_img.item(i+k,j+l,t)
                    d = random.randint(0,n-1)
                    #d = 5
                    avg = float(Sum/(n-d))
                    out_img.itemset((i-x,j-x,t),avg)
            else:
                Sum = 0
                for k in range(-1*(x),filter_size-x):
                    for l in range(-1*(x),filter_size-x):
                        Sum = Sum + pad_img.item(i+k,j+l)
                d = random.randint(0,n-1)
                #d = 5
                avg = float(Sum/(n-d))
                out_img.itemset((i-x,j-x),avg)

    filter_ = "Alphatrimmed_"+str(filter_size)+"*"+str(filter_size)
    return out_img,filter_
```

Output:

->

Gaussian blur + Gaussian noise

Alpha-Trimmed Mean Filter(3*3) with d=5

9. Adaptive Filter (Local Noise Reduction Filter)

Explanation:

Adaptive filter is performed on the degraded image that contains the original image and noise. The mean and variance are the two statistical measures that a local adaptive filter depends with a defined mxn window region.

Adaptive Expression:

$\hat{f}(x,y)=g(x,y)-(\sigma_\eta^2/\sigma_L^2)[g(x,y)-m_L]$

where σ_L^2 - Local variance of the local region

m_L - Local Mean

σ_η^2 -Variance of overall noise

g(x,y)- Pixel value at the position (x,y)

Code :

```
def apply_filter(filter_class,img,gray):
    out_img = img.copy()
    print("Input Filter size(odd numbers) : ")
    filter_size = int(input())
    x = (int)(filter_size/2)
    n = filter_size * filter_size
    pad_img = np.pad(img,((x,x),(x,x)), mode='constant')
    local_mean = np.zeros((img.shape[0],img.shape[1]),dtype='float')
    local_variance = np.zeros((img.shape[0],img.shape[1]),dtype='float')

    r = pad_img.shape[0]
    c = pad_img.shape[1]

    n = filter_size * filter_size
    for i in range(x,r-x):
            for j in range(x,c-x):
                sr = i-x
                er = i+filter_size-x
                sc = j-x
                ec = j+filter_size-x
                tmp = pad_img[sr:er,sc:ec]
                mean = np.mean(tmp)
                var = np.var(tmp)

                local_mean.itemset((i-x,j-x),mean)
                local_variance.itemset((i-x,j-x),var)

    r = img.shape[0]
    c = img.shape[1]

    avg_noise_var = np.mean(local_variance[:,:])
    for i in range(0,r-1):
        for j in range(0,c-1):
            if avg_noise_var > local_variance[i,j]:
                local_variance.itemset((i,j), avg_noise_var)
    avg_local_var = np.mean(local_variance[:,:])
    out_img[:,:] = img[:,:] - (float(avg_noise_var/avg_local_var)*(img[:,:] - local_mean[:,:]))

    filter_ = "Adaptive_"+str(filter_size)+"*"+str(filter_size)
    return out_img,filter_
```

Output:

->

Gaussian Blur + Gaussian Noise

Adaptive Filter (3*3)

Result

Image Restoration is implemented successfully through various filters in the spatial domain.

Technology stack :

The above code doesn't use any in built functions to perform the required tasks.

Python Libraries Used :

numpy - to perform basic array operations

cv2 - to read,write,show an image and to convert color image to gray scale image

random - to generate random values

Functions created from scratch(for both color and gray images) to perform operations :

def degrade(degrade_type,img,gray) : Function performs distortion of a perfect image through convolution with given degradation type.

def noise(noise_typ,img,gray): This function takes input from the above function and adds the given noise type to further distort the image.

def apply_filter(filter_class,img,gray): This function restores the distorted image through application of various filters in spatial domain.