Lecture 12: Understanding and applying Advanced image filtering techniques

- **Advance Level Image Filters**
  - There is a long list of advanced level filtering techniques, some of the filters are

1.1 : Bilateral Image Filtering

1.2 : Conservative Filters

1.3 : Wiener Filter

1.4 : CCC Algorithm

1.1 : Bilateral Image Filtering

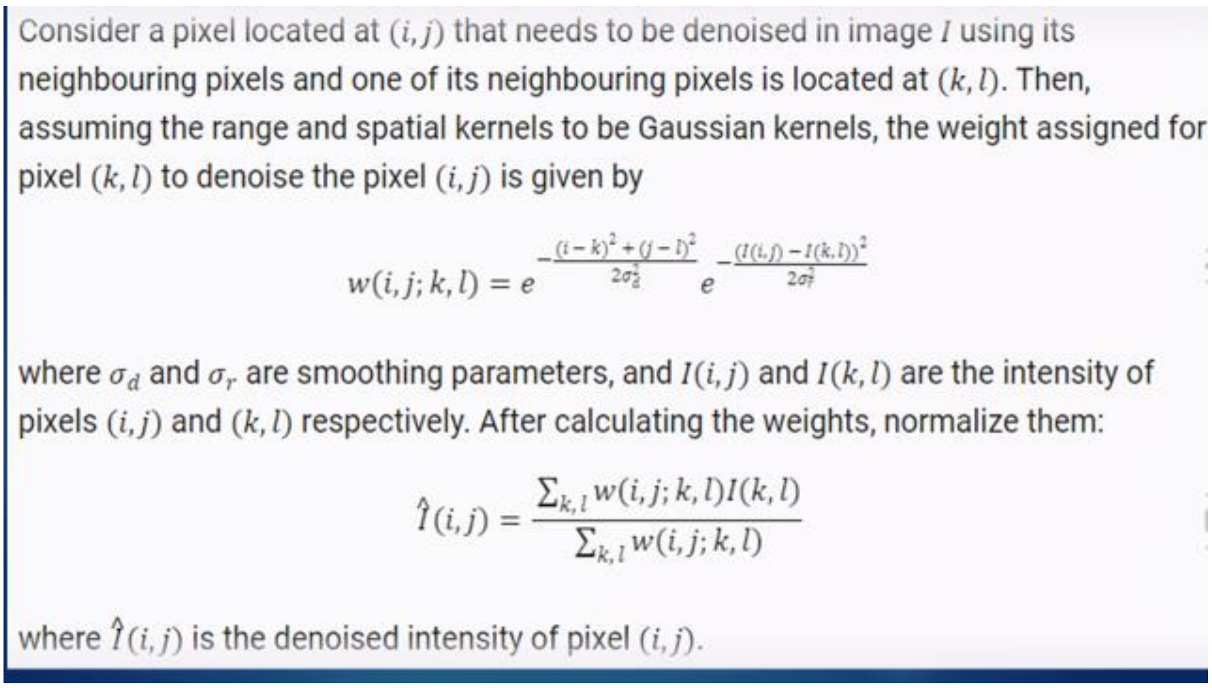
- while blurring the image, the bilateral filter considers the nearby pixel intensity values and considers whether the pixel is on edge or not.
- It Basically replaces the intensity of each pixel with a weighted average of intensity values from nearby pixels.

1.1 Bilateral Image Filtering

- Pixel depth distances are the distances between light and dark pixels of the neighborhood.
- The bilateral filter is given as

$$I^{\text{filtered}}(x) = \frac{1}{W_p} \sum_{x_i \in \Omega} I(x_i) f_r(\|I(x_i) - I(x)\|) g_s(\|x_i - x\|)$$

$$W_p = \sum_{x_i \in \Omega} f_r(\|I(x_i) - I(x)\|) g_s(\|x_i - x\|)$$



Consider a pixel located at (i, j) that needs to be denoised in image I using its neighbouring pixels and one of its neighbouring pixels is located at (k, l). Then, assuming the range and spatial kernels to be Gaussian kernels, the weight assigned for pixel (k, l) to denoise the pixel (i, j) is given by

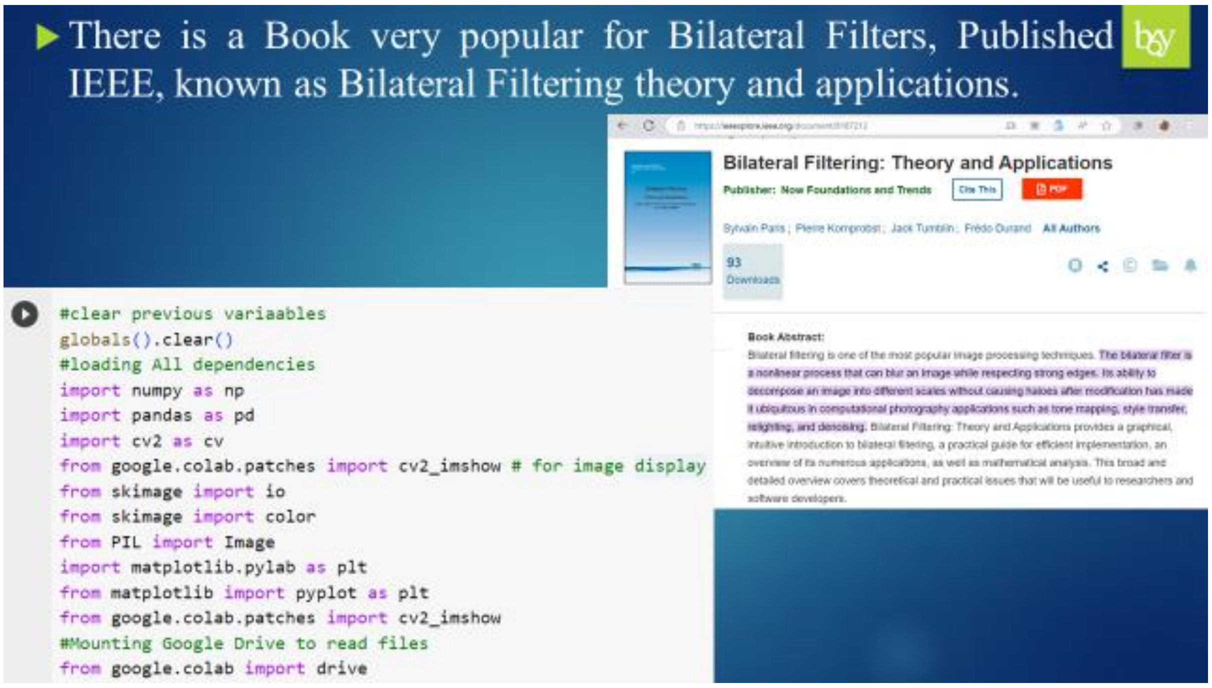
$$w(i,j;k,l) = e^{-\frac{(i-k)^2+(j-l)^2}{2\sigma_d^2}} e^{-\frac{(I(i,j)-I(k,l))^2}{2\sigma_r^2}}$$

where σ_d and σ_r are smoothing parameters, and $I(i,j)$ and $I(k,l)$ are the intensity of pixels (i,j) and (k,l) respectively. After calculating the weights, normalize them:

$$\hat{I}(i,j) = \frac{\sum_{k,l} w(i,j;k,l) I(k,l)}{\sum_{k,l} w(i,j;k,l)}$$

where $\hat{I}(i,j)$ is the denoised intensity of pixel (i,j).

▶ There is a Book very popular for Bilateral Filters, Published by IEEE, known as Bilateral Filtering theory and applications.



▶ There is a Book very popular for Bilateral Filters, Published by IEEE, known as Bilateral Filtering theory and applications.

Bilateral Filtering: Theory and Applications

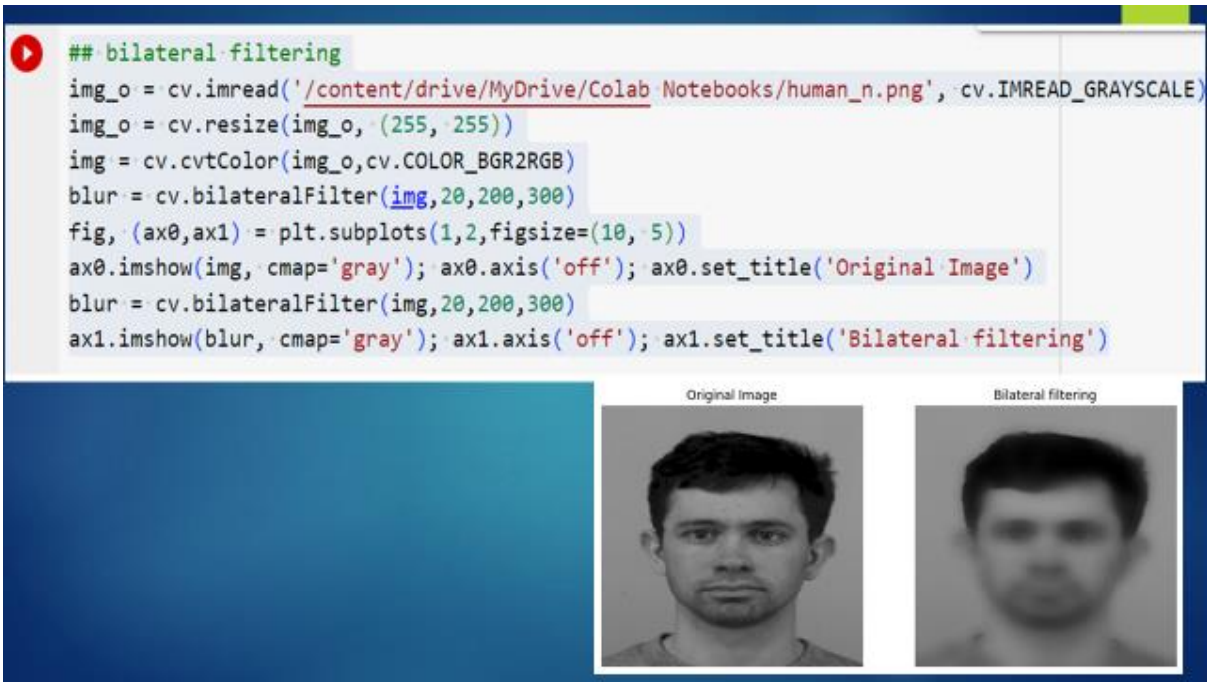
Publisher: Now Foundations and Trends

Sylvain Paris ; Pierre Kornprobst ; Jack Tumblin ; Frédo Durand All Authors

93 Downloads

Book Abstract:
Bilateral filtering is one of the most popular image processing techniques. The bilateral filter is a nonlinear process that can blur an image while respecting strong edges. Its ability to decompose an image into different scales without causing haloes after modification has made it ubiquitous in computational photography applications such as tone mapping, style transfer, relighting, and denoising. Bilateral Filtering: Theory and Applications provides a graphical, intuitive introduction to bilateral filtering, a practical guide for efficient implementation, an overview of its numerous applications, as well as mathematical analysis. This broad and detailed overview covers theoretical and practical issues that will be useful to researchers and software developers.

```
#clear previous variaables
globals().clear()
#loading All dependencies
import numpy as np
import pandas as pd
import cv2 as cv
from google.colab.patches import cv2_imshow # for image display
from skimage import io
from skimage import color
from PIL import Image
import matplotlib.pylab as plt
from matplotlib import pyplot as plt
from google.colab.patches import cv2_imshow
#Mounting Google Drive to read files
from google.colab import drive
```



```
## bilateral filtering
img_o = cv.imread('/content/drive/MyDrive/Colab Notebooks/human_n.png', cv.IMREAD_GRAYSCALE)
img_o = cv.resize(img_o, (255, 255))
img = cv.cvtColor(img_o,cv.COLOR_BGR2RGB)
blur = cv.bilateralFilter(img,20,200,300)
fig, (ax0,ax1) = plt.subplots(1,2,figsize=(10, 5))
ax0.imshow(img, cmap='gray'); ax0.axis('off'); ax0.set_title('Original Image')
blur = cv.bilateralFilter(img,20,200,300)
ax1.imshow(blur, cmap='gray'); ax1.axis('off'); ax1.set_title('Bilateral filtering')
```

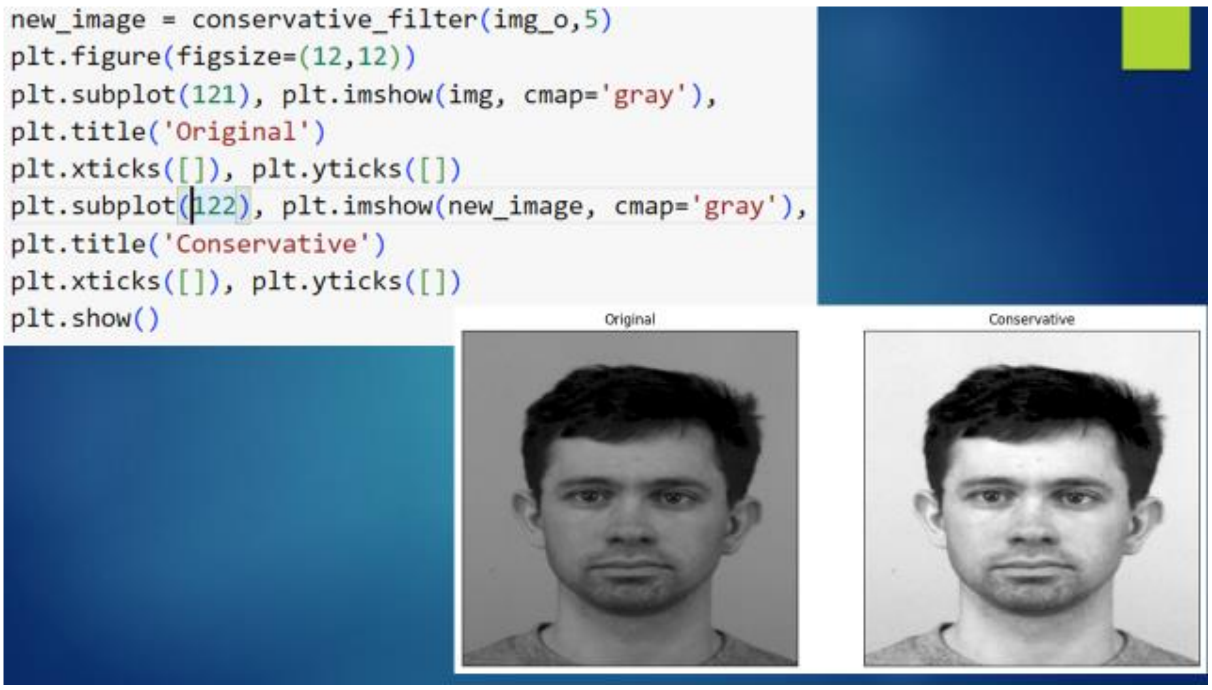
Conservative Image Filtering

- The conservative filters are preferred to remove salt and pepper noise. Determines the minimum intensity and maximum intensity within a neighborhood of a pixel.
- If the intensity of the center pixel is greater than the maximum value it is replaced by the maximum value.
- If it is less than the minimum value than it is replaced by the minimum value.
- The conservative filter preserves edges of the images but does not remove speckle noise. the filter is not able to remove as much salt-and-pepper noise as a median filter (although it does preserve more detail.)

# Let us first define a conservative filter

```
• def conservative_filter(data, filter_size):
•     temp = []
•     indexer = filter_size // 2
•     new_image = data.copy()
•     nrow, ncol = data.shape
•     for i in range(nrow):
•         for j in range(ncol):
•             for k in range(i-indexer, i+indexer+1):
•                 for m in range(j-indexer, j+indexer+1):
•                     if (k > -1) and (k < nrow):
```

- if (m > -1) and (m < ncol):
- temp.append(data[k,m])
- temp.remove(data[i,j])
- max_value = max(temp)
- min_value = min(temp)
- if data[i,j] > max_value:
- new_image[i,j] = max_value
- elif data[i,j] < min_value:
- new_image[i,j] = min_value
- temp =[]
- return new_image.copy()



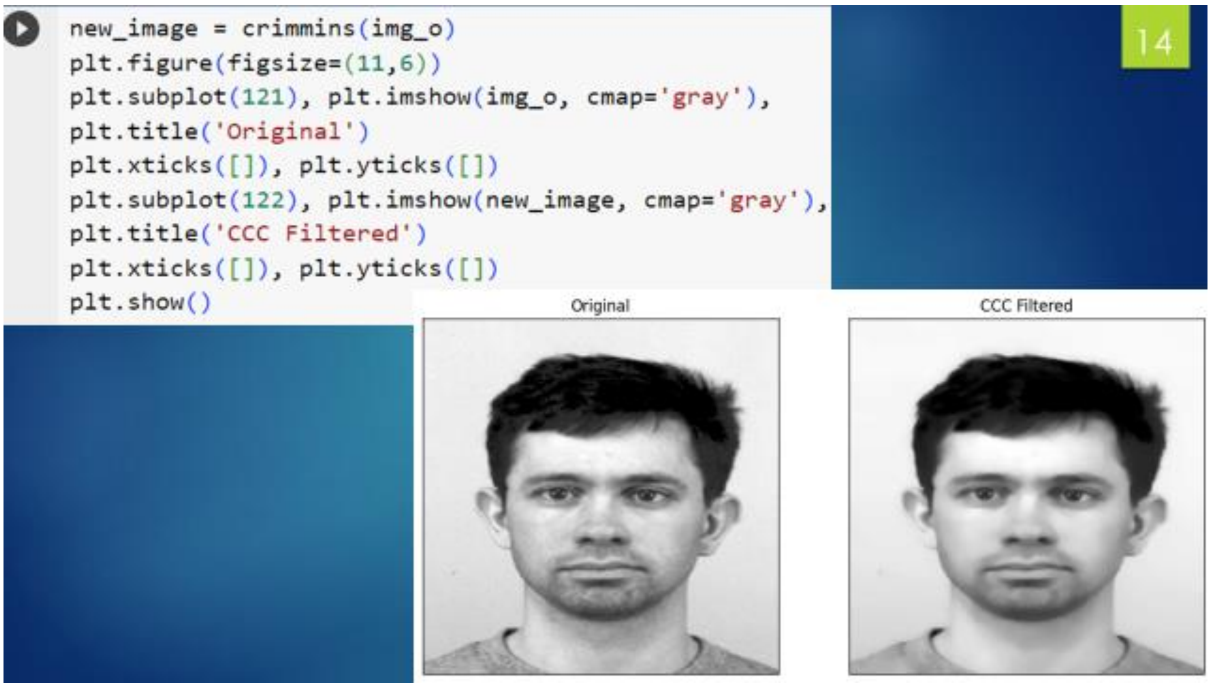
```
new_image = conservative_filter(img_o,5)
plt.figure(figsize=(12,12))
plt.subplot(121), plt.imshow(img, cmap='gray'),
plt.title('Original')
plt.xticks([]), plt.yticks([])
plt.subplot(122), plt.imshow(new_image, cmap='gray'),
plt.title('Conservative')
plt.xticks([]), plt.yticks([])
plt.show()
```

Crimmins complementary culling algorithm

- The Crimmins complementary culling algorithm is used to remove speckle noise and smooth the edges. It also reduces the intensity of salt and pepper noise. The algorithm compares the intensity of a pixel in a image with the intensities of its 8 neighbors.
- The algorithm considers 4 sets of neighbors (N-S, E-W, NW-SE, NE-SW.) Let a,b,c be three consecutive pixels (for example from E-S). Then we can explain the algorithm for each iteration:
- Crimmins complementary culling algorithm
- **a) Dark pixel adjustment: For each of the four directions**

- 1) Process whole image with: if a ≥ b+2 then b = b + 1
- 2) Process whole image with: if a > b and b ≤ c then b = b + 1
- 3) Process whole image with: if c > b and b ≤ a then b = b + 1
- 4) Process whole image with: if c ≥ b+2 then b = b + 1

- **b) Light pixel adjustment: For each of the four directions**
  - 1) Process whole image with: if a ≤ b — 2 then b = b — 1
  - 2) Process whole image with: if a < b and b ≥ c then b = b — 1
  - 3) Process whole image with: if c < b and b ≥ a then b = b — 1
  - 4) Process whole image with: if c ≤ b — 2 then b = b — 1

- 

```
new_image = crimmins(img_o)
plt.figure(figsize=(11,6))
plt.subplot(121), plt.imshow(img_o, cmap='gray'),
plt.title('Original')
plt.xticks([]), plt.yticks([])
plt.subplot(122), plt.imshow(new_image, cmap='gray'),
plt.title('CCC Filtered')
plt.xticks([]), plt.yticks([])
plt.show()
```

Transform Domain Filtering

- In transform domain filtering becomes easier because the convolution in space domain is more complex as compared to multiplication in frequency domain
- Because of complexity it requires more computational time in frequency domain filtering

Conversion to frequency domain

- The reason why we are interested in an image’s frequency domain representation is that it is less expensive to apply frequency filters to an image in the frequency domain than to apply the filters in the spatial domain.
- This is due to the fact that each pixel in the frequency domain representation corresponds to a frequency rather than a location of the image. The ‘dft’ function determines the discrete Fourier transform of an image.

- This is due to the fact that each pixel in the frequency domain representation corresponds to a frequency rather than a location of the image. The ‘dft’ function determines the discrete Fourier transform of an image.
- The Fourier transform (which decomposes a function into its sine and cosine components) can be applied to an image in order to obtain its frequency domain representation.

Conversion to frequency domain

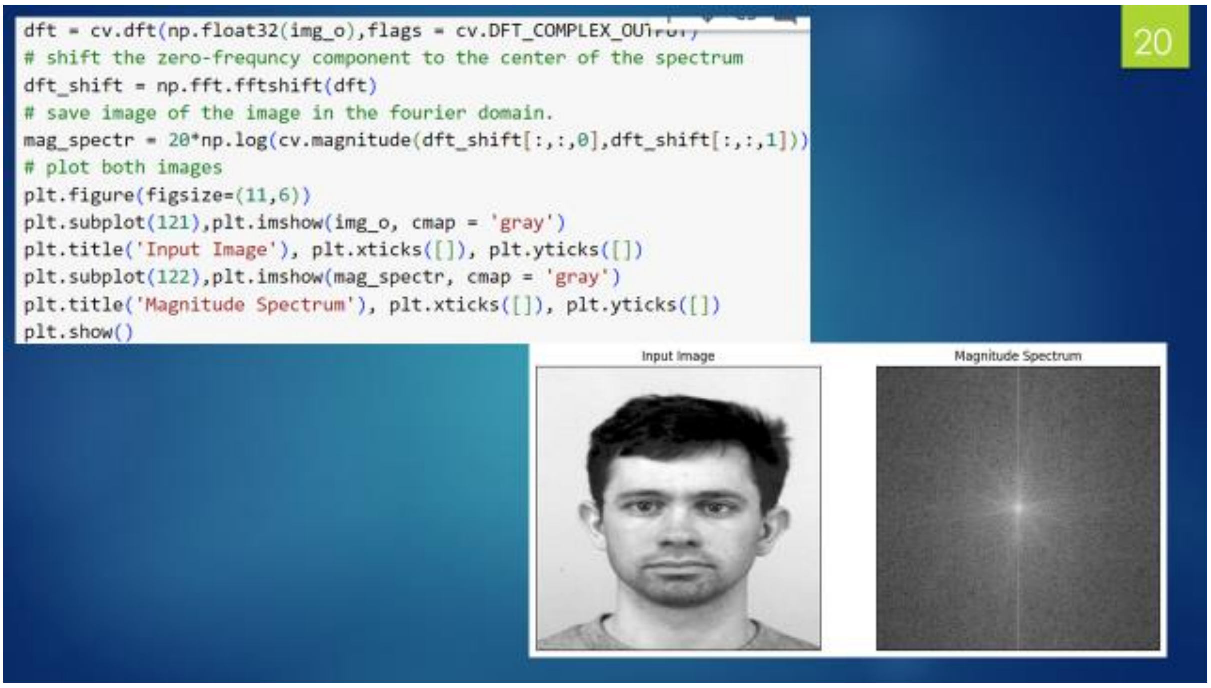
- For a M x N image the two dimensional discrete Fourier transform is given by

$$F(r,c) = \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} f(i,j) e^{-2\pi\left(\frac{ki}{M}+\frac{lj}{N}\right)}$$

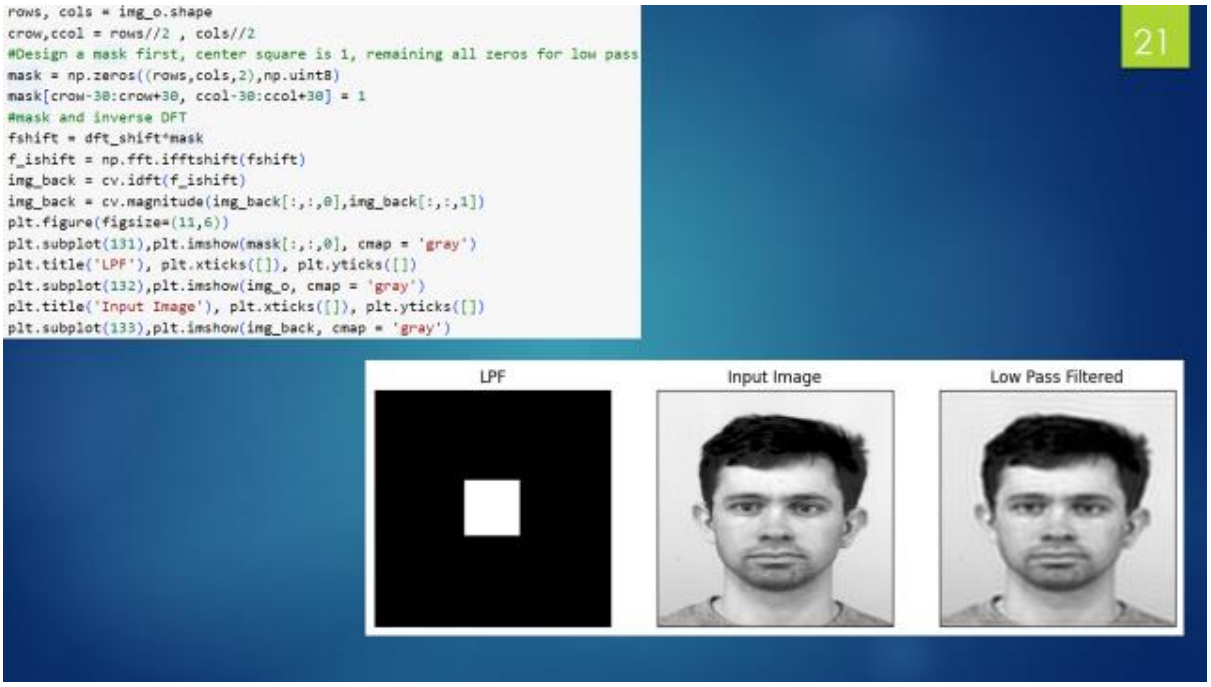
- If we use imshow function then there will be amplitude specturum instead of image.
- In order to get our image back to space domain, inverse Fourier transform is needed
- Converting back to Space domain
- For a M x N image the two dimensional discrete inverse discrete Fourier transform

$$f(a,b) = \frac{1}{M\times N} \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} f(k,l) e^{2\pi\left(\frac{ka}{M}+\frac{lb}{N}\right)}$$

- After converting back to space domain, if we use imshow function then original image can be visualized.



```
dft = cv.dft(np.float32(img_o),flags = cv.DFT_COMPLEX_OUTPUT)
# shift the zero-frequncy component to the center of the spectrum
dft_shift = np.fft.fftshift(dft)
# save image of the image in the fourier domain.
mag_spectr = 20*np.log(cv.magnitude(dft_shift[:,:,0],dft_shift[:,:,1]))
# plot both images
plt.figure(figsize=(11,6))
plt.subplot(121),plt.imshow(img_o, cmap = 'gray')
plt.title('Input Image'), plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(mag_spectr, cmap = 'gray')
plt.title('Magnitude Spectrum'), plt.xticks([]), plt.yticks([])
plt.show()
```

- 

```
rows, cols = img_o.shape
crow,ccol = rows//2 , cols//2
#Design a mask first, center square is 1, remaining all zeros for low pass
mask = np.zeros((rows,cols,2),np.uint8)
mask[crow-30:crow+30, ccol-30:ccol+30] = 1
#mask and inverse DFT
fshift = dft_shift*mask
f_ishift = np.fft.ifftshift(fshift)
img_back = cv.idft(f_ishift)
img_back = cv.magnitude(img_back[:,:,0],img_back[:,:,1])
plt.figure(figsize=(11,6))
plt.subplot(131),plt.imshow(mask[:,:,0], cmap = 'gray')
plt.title('LPF'), plt.xticks([]), plt.yticks([])
plt.subplot(132),plt.imshow(img_o, cmap = 'gray')
plt.title('Input Image'), plt.xticks([]), plt.yticks([])
plt.subplot(133),plt.imshow(img_back, cmap = 'gray')
```

Wiener Filter

- The filtered image using Wiener filter is given by

$$\hat{F}(u, v) = \left[\frac{H^*(u, v)}{|H(u, v)|^2 + K}\right] G(u, v)$$

- In which, $F(u, v)$ = the estimate, $G(u, v)$ = degraded image, $H(u, v)$ = degradation function, $H * (u, v)$ = complex conjugate of $H(u, v)$ K = constant

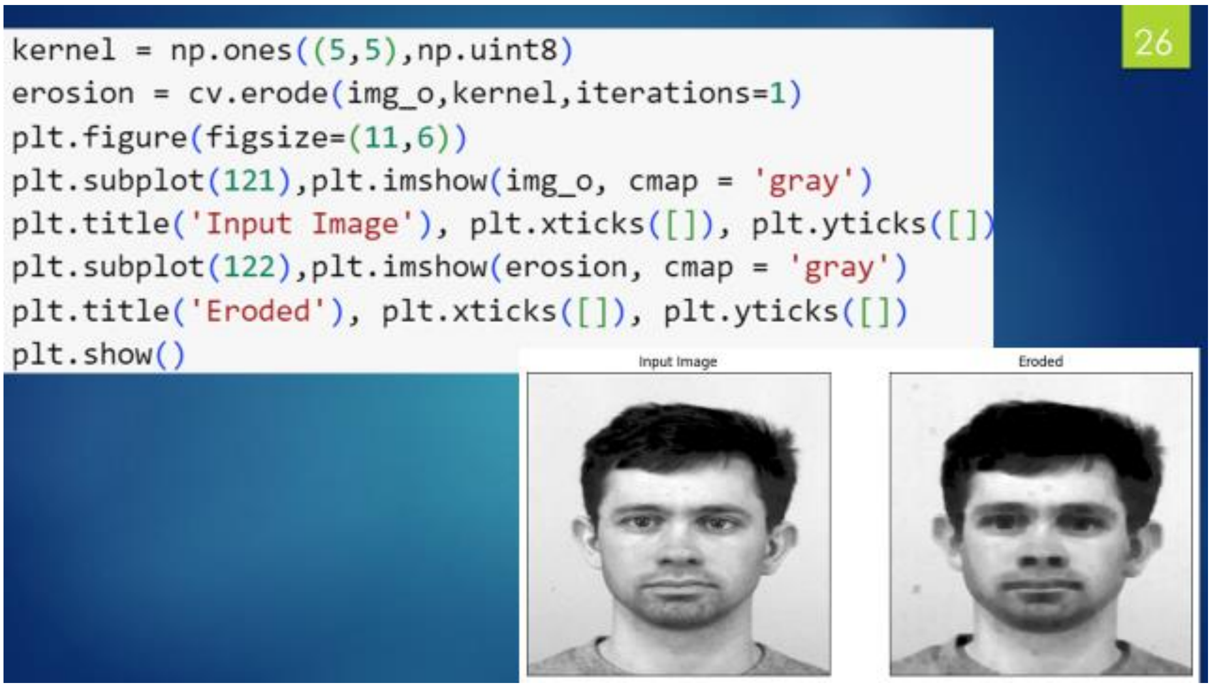
```
def wiener_filter(img, kernel, K):
    kernel /= np.sum(kernel)
    dummy = np.copy(img)
    dummy = fft2(dummy)
    kernel = fft2(kernel, s = img.shape)
    kernel = np.conj(kernel) / (np.abs(kernel) ** 2 + K)
    dummy = dummy * kernel
    dummy = np.abs(ifft2(dummy))
    return dummy
```

- Well I am leaving this for exercise. You can write this function , then apply on a noisy image and then see both noisy and denoised together

Morphological Filtering

- We know that all filters need two inputs: image and kernel, which decides the nature of the operation.
- In Morphological filters are filter operations are based on image shape
- Morphological operations are very useful for segmentation.

Erosion

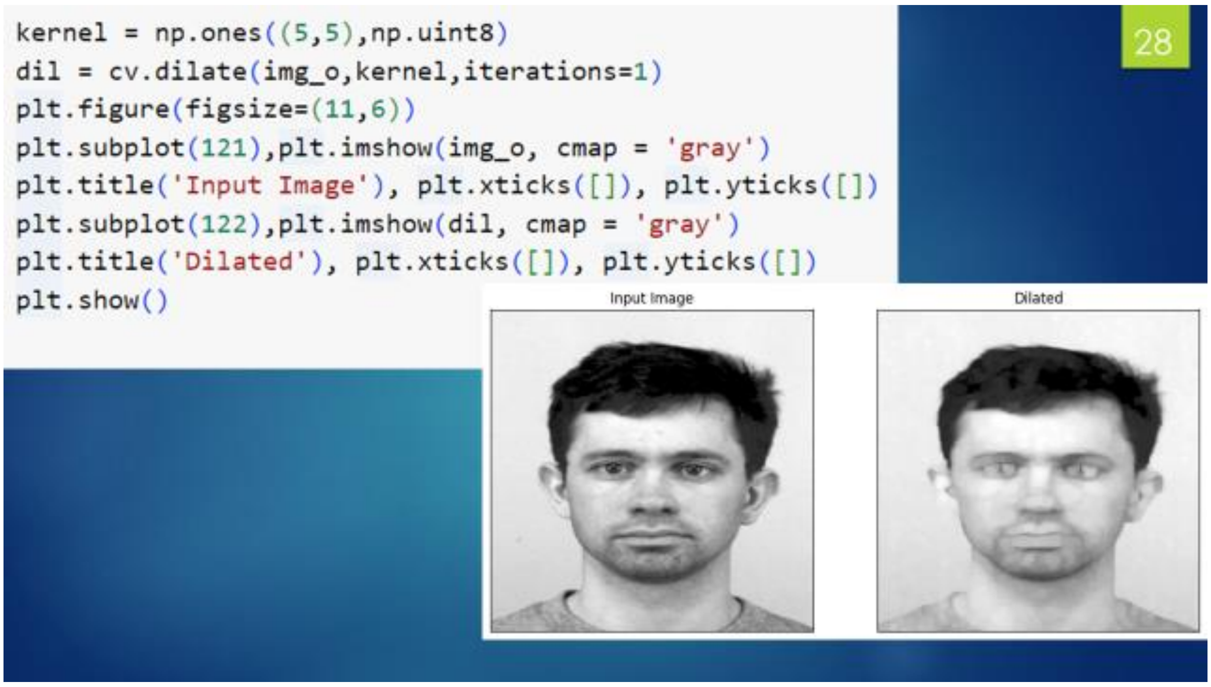
- It is just like soil erosion; it erodes the boundary, it warns away the boundaries of foreground objects, i.e., tries to keep the foreground white.
- So the operation follows as the kernel slides over the image, and a pixel of the image is considered one only if all the pixels under the kernel are 1; otherwise, it is eroded.
- 

26

```
kernel = np.ones((5,5),np.uint8)
erosion = cv.erode(img_o,kernel,iterations=1)
plt.figure(figsize=(11,6))
plt.subplot(121),plt.imshow(img_o, cmap = 'gray')
plt.title('Input Image'), plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(erosion, cmap = 'gray')
plt.title('Eroded'), plt.xticks([]), plt.yticks([])
plt.show()
```

Dilation

- It is just the opposite of erosion;
- here, the pixel is considered as one if at least one pixel under the kernel is one,
- so in this case, it increases the white region in the image. Dilation is also helpful in joining the broken part of an object.



```
kernel = np.ones((5,5),np.uint8)
dil = cv.dilate(img_o,kernel,iterations=1)
plt.figure(figsize=(11,6))
plt.subplot(121),plt.imshow(img_o, cmap = 'gray')
plt.title('Input Image'), plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(dil, cmap = 'gray')
plt.title('Dilated'), plt.xticks([]), plt.yticks([])
plt.show()
```

Conclusion

- Advance level filters, advance Gaussian filters known as bilateral filter, minimum filter, maximum filter, conservative filter, CCC algorithm for filtering.
- Kuwahara filter, Accelerated graph-based spectral polynomial filters, Nonlinear iterative filters, Chebyshev and conjugate gradient filters etc
- Filters for edge detection will be discussed in upcoming session
- I hope this session would have give you a good quick start of the edge preserving filters and morphological filters and you will explore more related filters based on this understanding
- Thank You all . Happy Coding.
- Meet you again in next Tutorial