Image Processing Lab

2022-2023

Group #8

Experiment #5

Final Report

Pair No. 43

ID1: 301613501

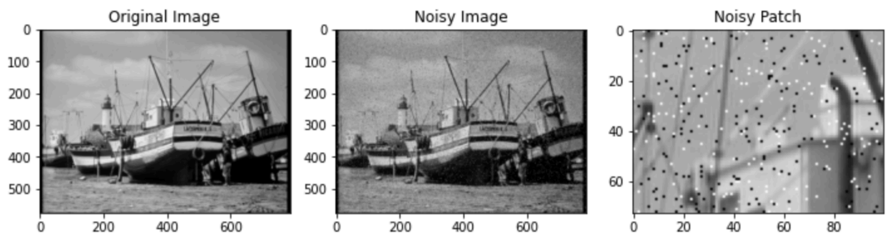
ID2: 208560086

1. Done – Introduction was read.
2. 3-level brute force implementation for salt and pepper noise with parameter p.

Code

```
def brute_saltpaper(img,p):
 w,h = img.shape
 # create the base for base for the noisy image
 noised_img = img
 rand_img = np.random.rand(w,h)
 # create the salt and paper noise
 noised_img = np.where(rand_img < p/2, 0, noised_img)
 noised_img = np.where(rand_img > 1 − p/2, 255, noised_img)
 # return the noisy image
 return noised_img

noisy_image = brute_saltpaper(img, 0.05)
noisy_patch = noisy_image[200: 273, 300: 400]

fig, ax = plt.subplots(nrows = 1, ncols = 3, figsize = (10, 10))
ax = ax.ravel()

ax[0].imshow(img, cmap = 'gray')
ax[0].set_title('Original Image')
ax[1].imshow(noisy_image, cmap = 'gray')
ax[1].set_title('Noisy Image')
ax[2].imshow(noisy_patch, cmap = 'gray')
ax[2].set_title('Noisy Patch')

plt.tight_layout()
plt.show()
```

Results

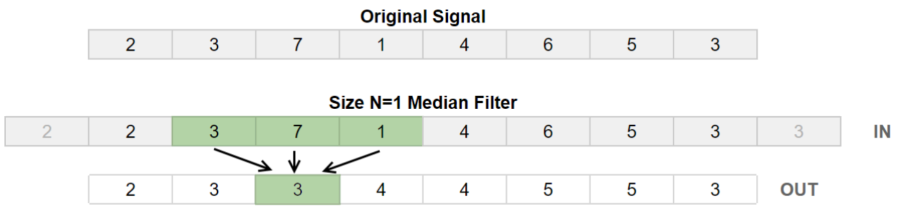


3. 1D and 2D median filter

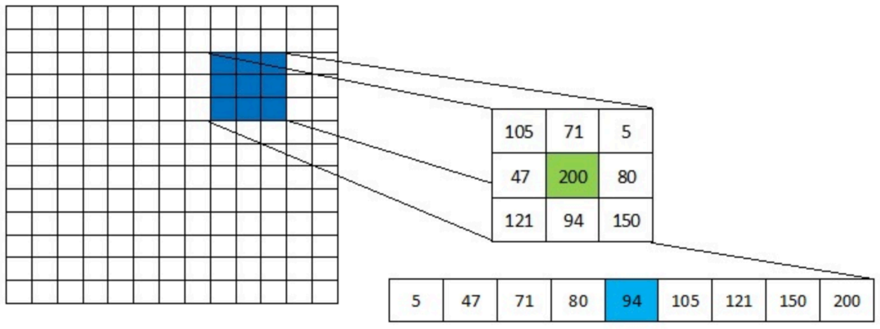
Median filter is used to remove noises from a signal. The idea of this filter is to scan all the signal's entries and replace the entry with the median of its neighbor values.

For example, in 1-D [Source] we can see illustration of median filter with size of N=1. That can also be represented mathematically by the following form that will help to understand better the size N=1 (it is 1 from each side of the entry).

$$g(i) = median[f(i-N), \dots, f(i-1) \cdot f(i), \dots, x(f+N)]$$



The following example of 2-D median filter how 3x3 median filter replaces a large (noisy) value (i.e., 200) with the median of its neighbors (i.e., 94) [Source].



Linearity of the median filter [Source]

Claim: median filter is a non-linear operation

$$median(x_m + y_m) \neq median(x_m) + median(y_m)$$

Proof: We will prove this claim by contradiction on 1D signals.

Let $x_m = \{1, 1, 3\}\ and\ y_m = \{1, 2, 0\}$

So, $median(x_m) = 1$ & $median(y_m) = 1$

But, $median(x_m + y_m) = 3$

We see that $3 \neq 2$ ■

Shift invariant of the median filter [Source]

Claim: Median filter is a shift invariant operation

Proof: For a 1D signal, shift invariance of a filter F implies the following:

If $y_1[n] = F(x1[n])$, then $y_2[n] = F\{x_1[n-k])\} = y_1[n-k]$

The 1D median filter is $y[n] = median\{x[i], i \in w\}$, where w is the specified neighbourhood.

The following example illustrates the shift invariance (for all the signals, the sample at the origin is in bold, and zero padding is assumed).

For the input, $x1[n] = [\mathbf{4}, 2,3,1,5,9]$ and $w = 3$, the output is, $y1[n] = [\mathbf{2}, 3,2,3,5,5]$

Now, consider a new input which is a shifted version of the previous input
$x2[n] = x1[n-2] = [\mathbf{0}, 0,4,2,3,1,5,9]$

The output is $y2[n] = [\mathbf{0}, 0,2,3,2,3,5,5]$ which is nothing but $y1[n-2]$ ■

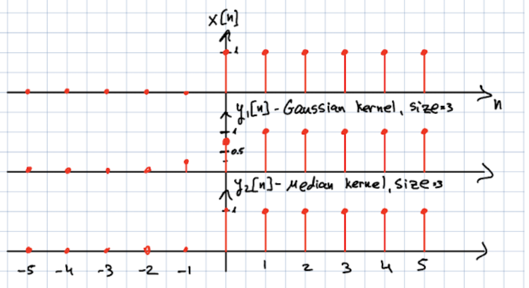
4. Handwritten solution for applying the given filter kernels on the step signal.

Input $x[n] = Step[n] = \begin{cases} 1 & n \geq 0 \\ 0 & n < 0 \end{cases}$

$h_1 = [0.25 \quad 0.5 \quad 0.25]$

$h_2 =$ median Siz=3

$y[n] = (x * h)(n) = \sum_{m=-\infty}^{\infty} x[n-m]\, h[n]$



5. **scipy.signal.medfilt2d** is a 2-dimensional median filter function in the Python scientific computing library **scipy**. It is used to apply a median filter to a 2-dimensional array (e.g., an image).

   A brief overview of this function's parameters:

- **image**: The 2-dimensional array to be filtered. It should be a NumPy array of type **float** or **int**.
- **kernel_size**: The size of the kernel (i.e., the sliding window) to use for the median filter. It should be a tuple of odd integers, specifying the number of rows and columns in the kernel.
- **footprint**: (optional) An array of values that defines the shape of the kernel. If provided, it should be a NumPy array of **bool** values.
- **mode**: (optional) A string specifying the border mode to use when the kernel extends beyond the edges of the input array. Valid values are **'reflect'**, **'constant'**, **'nearest'**, **'mirror'**, and **'wrap'**.

   The **kernel_size** parameter determines the size of the kernel that is used to compute the median value for each element in the output array. The kernel is a sliding window that moves across the input array and calculates the median value for the elements within the window.

   The kernel size should be an odd integer tuple, with the first element specifying the number of rows in the kernel and the second element specifying the number of columns. For example, a kernel size of **(3, 3)** would be a 3x3 kernel, while a kernel size of **(5, 5)** would be a 5x5 kernel.

   A valid value for **kernel_size** would be a tuple of odd integers, such as **(3, 3)**, **(5, 5)**, **(7, 7)**, etc. The kernel size should be chosen based on the size and complexity of the features in the input array that you want to preserve or enhance. A larger kernel size may be more effective at smoothing out noise, but it may also blur finer details in the input array.

6. Wiener filter where σ_n^2 is the variance of the noise and a is a constant.

   Wiener filter is defined by:

$$H^{WI}(u,v) = \frac{H^*(u,v)}{|H(u,v)|^2 + \frac{S_{nn}(u,v)}{S_{ff}(u,v)}}$$

   In the introduction for this lab there was an assumption that the noise is white, which means it has a constant spectral density for all spatial frequencies, so:

$$S_{nn}(u,v) = S_n^2$$

In addition, there was one more assumption that the spectral density of the original signal is in inverse proportion to the square of the spatial frequency that gives us

$$s_{ff}(u,v) = \frac{1}{k} \cdot \frac{1}{u^2 + v^2}$$

So, we got that the Wiener filter can also be represented as

$$H^{WI}(u,v) = \frac{H^*(u,v)}{|H(u,v)|^2 + kS_n^2(u^2 + v^2)}$$

Now, we need to find the relation between k and the a in the given expression.

$$H^{WI}(u,v) = \frac{H^*(u,v)}{|H(u,v)|^2 + a\sigma_n^2(u^2 + v^2)}$$

We can see from both of the expressions that

$$kS_n^2 = a\sigma_n^2 \rightarrow \frac{k}{a} = \frac{s_n^2}{\sigma_n^2}$$

Since we assumed that the noise signal is a white noise with variance σ_n^2, we can calculate the spectral density of S_{nn} and compare it to the given S_n^2 parameter.

$$\mathcal{F}\{S_{nn}(u,v)\} = \frac{1}{N}\sum_{m=0}^{N-1}\sum_{n=0}^{N-1} \sigma_n^2 \delta(n,m) e^{-j\frac{2\pi}{N}(nu+mv)} = \frac{\sigma_n^2}{N} \triangleq S_n^2$$

Now we will use the proportion that we showed earlier,

$$\frac{k}{a} = \frac{s_n^2}{\sigma_n^2} = \frac{\sigma_n^2}{\sigma_n^2 N} = \frac{1}{N}$$

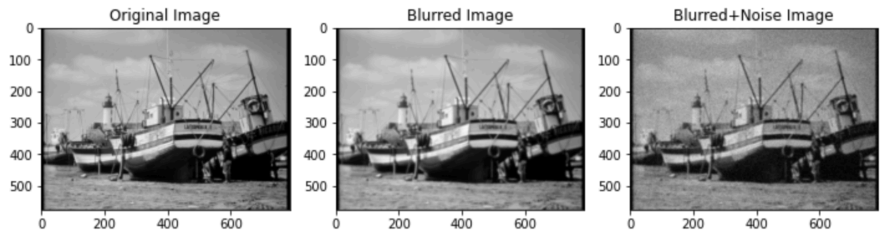
Therefore, by the relation $a = N \cdot k$ we can express the following term of the Wiener filter.

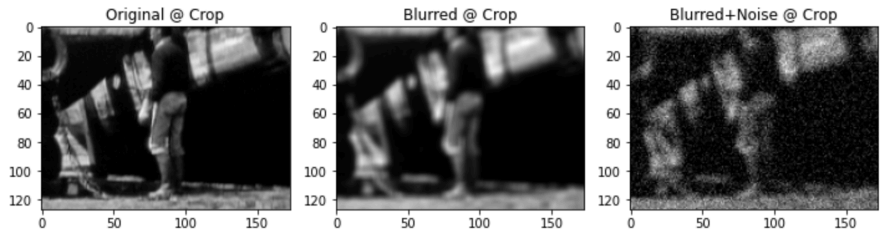
$$H^{WI}(u,v) = \frac{H^*(u,v)}{|H(u,v)|^2 + a\sigma_n^2(u^2 + v^2)} \quad \blacksquare$$

7. We loaded the image and blurred it by an average image of size 5x5. After that we added a gaussian noise. For all those operations we used the required functions.

<u>Results</u>

```
Image shape:(576, 787), Image data type:float64
```





Code

```
from skimage.util import random_noise
from skimage.util.dtype import img_as_float64

img64f = color.rgb2gray(io.imread('BoatsColor.jpg'))
img64f = img_as_float64(img64f)
print ('Image shape: {}, Image data type: {}\n'.format(img64f.shape, img64f.dtype))

k_avg_filter = np.ones((5,5))/ 25

img64f_blurred = cv2.filter2D(img64f, −1, k_avg_filter)
img64f_noisy = random_noise(img64f, mode = 'gaussian', var = 0.01)
img64f_blurred_noisy = random_noise(img64f_blurred, mode = 'gaussian', var = 0.01)

fig, ax = plt.subplots(nrows = 2, ncols = 3, figsize = (10, 10))
ax = ax.ravel()

img64f = img_as_ubyte(img64f)
img64f_blurred = img_as_ubyte(img64f_blurred)
img64f_noisy = img_as_ubyte(img64f_noisy)
img64f_blurred_noisy = img_as_ubyte(img64f_blurred_noisy)

ax[0].imshow(img64f, cmap = 'gray')
ax[0].set_title('Original Image')
ax[1].imshow(img64f_blurred, cmap = 'gray')
ax[1].set_title('Blurred Image')
ax[2].imshow(img64f_blurred_noisy, cmap = 'gray')
ax[2].set_title('Blurred + Noise Image')
ax[3].imshow(img64f[373: 500, 500: 673], cmap = 'gray')
ax[3].set_title('Original @ Crop')
ax[4].imshow(img64f_blurred[373: 500, 500: 673], cmap = 'gray')
ax[4].set_title('Blurred @ Crop')
ax[5].imshow(img64f_blurred_noisy[373: 500, 500: 673], cmap = 'gray')
ax[5].set_title('Blurred + Noise @ Crop')

plt.tight_layout()
plt.show()
```