An Analysis of Image Denoising Techniques

Benjamin A. Schifman
Department of Electrical and Computer Engineering
University of Arizona
Tucson, AZ 85719
bschifman@email.arizona.edu

Abstract

Digital denoising techniques are used to filter out unwanted noise in a signal. In images, noisy signals are present in the form of non coherent Salt & Pepper noise and Gaussian noise to coherent noise introduced inherently from the imager or from signal processing algorithms. This paper examines some of the common methods for removing unwanted noise, along with implementing more adept filtering techniques in the form wavelet filtering.

1 Introduction

This paper explores noise filtering techniques implemented in Python and an available Python image processing library $OpenCV$. The filters are implemented on images with random Gaussian noise and Salt & Pepper noise, and their output Peak Signal to Noise Ratios are compared. The two $python$ files are detailed in section 6. The standard filters that were implemented using the $OpenCV$ library were the: *blur, gaussian blur, median* and *bilateral* filters and the filter windows were varied in order to generate the optimal resulting filter. The Haar Wavelet Transform was implemented by hand, and the Daubechies 4 wavelet was implemented using the available $PyWavelets$ library after an attempt by hand of the algorithm was unsuccessful.

2 Methods/Approach

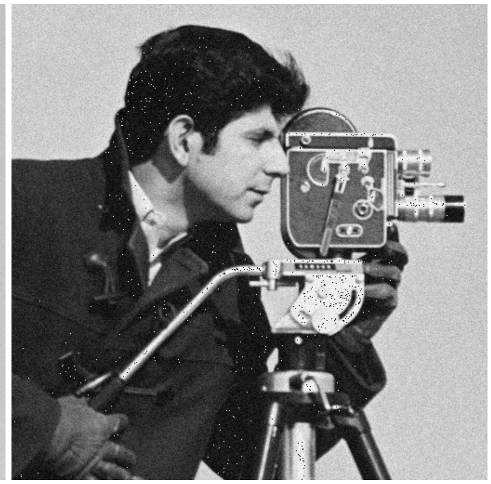
2.1 Noise Generation

Two images were generated with different noise distributions for the purpose of analyzing the efficacy of the different applied filtering techniques. The first noisy image was generated with a normal Gaussian distribution that had been scaled by a factor of 10. The noise was scaled in order to be more visually evident in the image along with increasing the noise power in the image. Gaussian noise is generally a common form of noise that principally arises in images during acquisition and is caused by a number of factors, a few being poor illumination, high circuitry temperature, and electronic interference. The second noisy image was generated through adding a 0.4% Salt & Pepper (S&P) distribution. The S&P noise added was equally distributed "Salt" white pixels, and "Pepper" black pixels. S&P noise potentially occurs in images were intermittent and non-reliable image communication systems are present as they can elicit sharp and sudden disturbances in the image signal. The following Figure 1 depicts the original image along with the noise induced images.

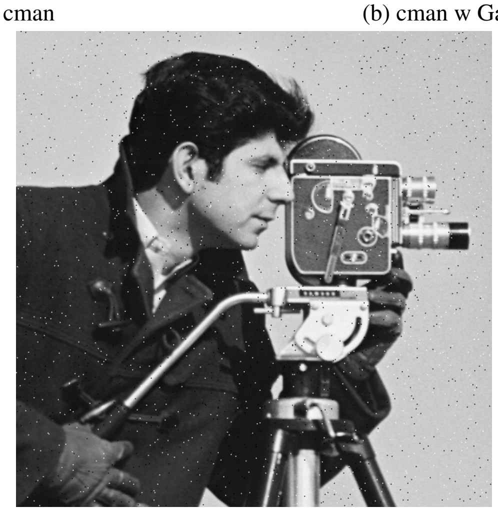
2.2 Peak Signal to Noise Ratio

In order to analyze the utility of the aforementioned filtering techniques the Peak Signal to Noise Ratios (PSNR) for each filtered image was calculated. The PSNR of an image is the maximum power

(a) cman



(b) cman w Gaussian Noise



(c) cman w S&P Noise

Figure 1: Input Images

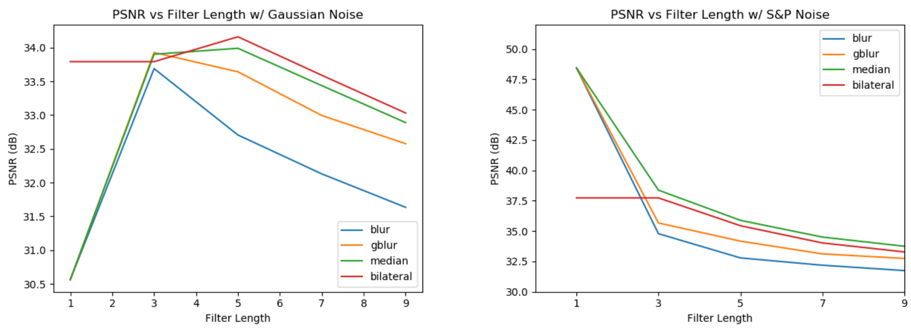
of an image and the power of the image noise. The following Equation (1) details the calculations for PSNR output in decibels (dB) which is the unit that will be continued throughout this paper.

$$PSNR = 10 * log_{10}\Big(\frac{MAX^2}{MSE}\Big) \tag{1}$$

Where MAX is the maximum grayscale pixel value for the image which in this case is an unsigned 8-bit image with a maximum value of 255, and the MSE is the Mean Squared Error between the filtered output image and the noisy image.

2.3 Standard Filters

The standard filters described in the following sections were implemented using the available python $OpenCV$ library and can be found in the $standardFilters$ function in the $dataSetup.py$ file. The correct filter lengths were chosen through the process of iterating over odd filter lengths and then observing the corresponding PSNR values from the filters and this can be seen in the following Figure 2 and in the $dataSetup.py$, $detFilterLength$ funciton. The resultant filter outputs can be found in the the Results section in Figures 7, 8, 9, 10



(a) Filters w Gaussian Noisy Image.

(b) Filters w S&P Noisy Image.

Figure 2: PSNR vs. Filter Lengths

2.3.1 Blur Filtering

The Blur filter was implemented by convolving the the image with the normalized box window. $OpenCV$, however, implemented all of the convolution process under the hood, so only a specific filter window length was required. In the example of a 3x3 normalized block filter the kernel would look like the following Equation (2)

$$K = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{2}$$

2.3.2 Gaussian Blur Filtering

The Gaussian Blur filter was implemented by convolving the the image with a Gaussian kernel, with the standard deviation of the Gaussian distribution calculated by the length of the filter. In the example of a 3x3 Gaussian filter the kernel would look like the following Equation (3)

$$K = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{3}$$

2.3.3 Median Filtering

The Median filter was implemented by taking the pixels of the image that were under the kernel filter size and then replacing the central pixel element with the median value.

2.3.4 Bilateral Fitlering

The Bilateral Filter is a non-linear, edge preserving filter, and noise reducing smoothing filter. It replaces the intensity of the central pixel value with a weighted average of the other pixels in the filter window

2.4 Haar Wavelet Transform

The Haar Wavelet Transform (HWT), proposed by the Hungarian mathematician Alfréd Haar is a computationally efficient method for analyzing the local aspects of an image. A key advantage to the use and implementation of the HWT is in it simplicity to compute, along with it being easier to understand than most other wavelet transforms. A great benefit to using the HWT is that it is effective in signal and image compression and the algorithm is memory efficient in that all of the calculations can be done in place, however, the software depicted below uses temporary vectors for ease of following along. The motivation behind the use of the HWT and wavelet transforms in general is that wavelets are a means of expressing a function in a certain basis, but in contrast to Fourier analysis, where the basis is fixed, wavelets provide a general framework with varying orthogonal bases. This is beneficial due to the fact that in Fourier Analysis, and specifically the Discrete Fourier Transform (DFT), the DFT is limited to a fixed frequency content over time in representation of a trigonometric function, and in an image their is often contrast in image data where the characteristics can be vastly different in varying parts of the image. This can provide differenet resolution at different parts of the time-frequency plane [1]. The HWT can be expressed in terms of matrix operations such that the Forward HWT is expressed in equation (4):

$$y_n = W_n v_n \tag{4}$$

where v_n is the input vector to be transformed, W_n is the HWT matrix, and y_n is the transformed output vector. This calculation can be even more easily illustrated in the form of the following expanded graphic in Figure 3 that details the Haar transform of an input vector of length 8, its corresponding Haar Matrix W_8, and its output vector which is simply the mean or trend of two sequential pixel elements for the first half of the vector, and then the second half of the values are a running difference or fluctuation of two sequential pixel elements. The Haar Matrix is a set of odd rectangular pulse pairs

$$\tilde{W}_8 \mathbf{v} = \begin{bmatrix} 1/2 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/2 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1/2 & 1/2 \\ -1/2 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1/2 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1/2 & 1/2 \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \\ v_8 \end{bmatrix} = \begin{bmatrix} (v_1+v_2)/2 \\ (v_3+v_4)/2 \\ (v_5+v_6)/2 \\ (v_7+v_8)/2 \\ (v_2-v_1)/2 \\ (v_4-v_3)/2 \\ (v_6-v_5)/2 \\ (v_8-v_7)/2 \end{bmatrix} = \mathbf{y}$$

Figure 3: Haar Transform

The algorithm for this vector-wise HWT has been implemented by hand in the software and can be found in the $OneD\_HWT$ function in the $DWT.py$ file. On a 2D image the HWT is first calculated on the rows of the image, and then the HWT is calculated on the columns of the image. This 2D implementation can be found in the $TwoD\_HWT$ in the $DWT.py$ file. This 2D function also accounts for multiple iterations of the HWT in that each following iteration will reduce the image into smaller sub-wavelets. The definition of the HWT is simply defined for discrete signals that are of size N = 2^n, so for ease of calculations the input image was rescaled to a 512x512 size. The Inverse HWT can be calculated by the following Equation (5) with the same variables:

$$x_n = H^T y_n \tag{5}$$

The single iteration forward HWT of the input image "cman" can be found in the following Figure 4 where each quadrant represents a frequency direction. The 1st quadrant being the Horizontal orientation sub-image, the 2nd being the Low resolution sub-image where the low frequency pixel data values are, the 3rd quadrant corresponds to the Vertical orientation sub-image, and the final 4th quadrant is the Diagonal orientation sub-image [2]. Due to the fact that the figure is grayscale the higher frequency lateral components of the image are more difficult to see than the low frequency, Low resolution sub-image.

Figure 4: Cman Forward HWT.

2.5 Daubechies Wavelet Transform

The Daubechies Wavelet Transforms, discovered by mathematician Ingrid Daubechies, are a family of orthogonal bases that are conceptually similar to the HWT in that the mathematical computations are running averages and a differences via scalar products [1]. Being that the HWT and Daubechies Wavelet Transform have similar characteristics the HWT is also referred to as the 'DB1' transform. This project solely implements the 'DB4' wavelet algorithm, which utilizes four wavelet and scaling function coefficients. This Daubechies wavelet type has a balanced frequency response but a non-linear phase response. The coefficients in the DB4 wavelet are as follows in equation (6):

$$c_0 = \frac{1+\sqrt{3}}{4\sqrt{2}}, c_1 = \frac{3+\sqrt{3}}{4\sqrt{2}}, c_2 = \frac{3-\sqrt{3}}{4\sqrt{2}}, c_3 = \frac{1-\sqrt{3}}{4\sqrt{2}} \tag{6}$$

2.6 Thresholding

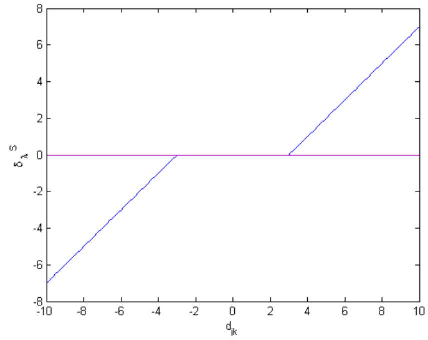
Pixel thresholding is often a simple but efficient non-linear denoising approach in the application of a wavelet transform. The thresholding is performed on the wavelet transformed image, and then the image is inverse wavelet transformed to yield a filtered output image. To determine the best threshold value to set, the $detThreshold$ function from $dataSetup.py$ was used. This function first iterates through thresholding values, then thresholds the Wavelet Transformed image, then inverse Wavelet Transforms the thresholded image, and finally calculates the Peak Signal to Noise ratio of the filtered image. The step size of $0.5 * std$ where std is the standard deviation of the Gaussian Noise was chosen to allow for a high enough resolution to see how the Threshold effects the PSNR, where the threshold is chosen based off of the one that results in the largest Peak Signal to Noise Ratio. Figure 6 depicts this PSNR vs. Threshold graph. The following Equations (7) and (8) detail the Hard and Soft Thresholding calculations respectively, with Figure 5 depicting the thresholds.

Hard Thresholding

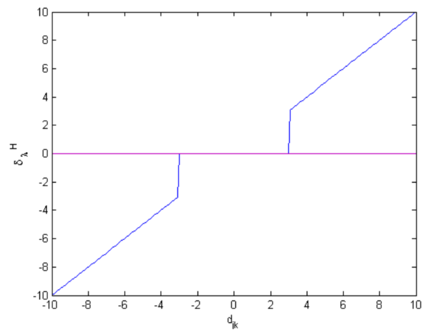
$$D^H(d|T) = \begin{cases} 0, & \text{if } |d| \leq T \\ d, & \text{if } |d| > T \end{cases} \tag{7}$$

Soft Thresholding

$$D^S(d|T) = \begin{cases} 0, & \text{if } |d| \leq T \\ d - T, & \text{if } d > T \\ d + T, & \text{if } d < -T \end{cases} \tag{8}$$



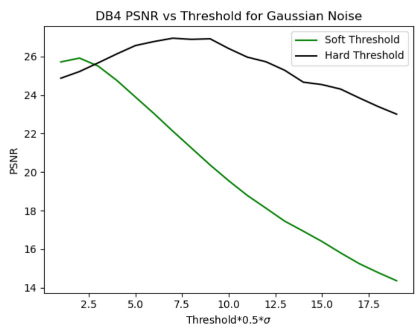
(a) Hard Threshold.



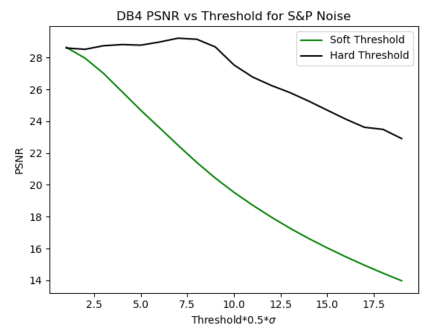
(b) Soft Threshold.

Figure 5: Thresholding Methods

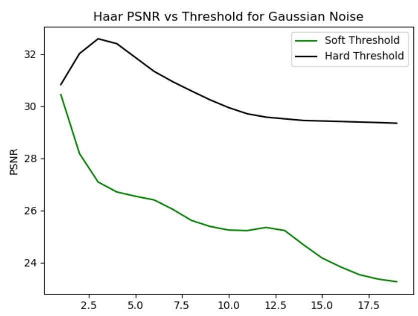
The following Figure 6 depicts the PSNR of the filtered wavelet transforms vs. a given threshold value, and the threshold value was chosen that resulted in the greatest PSNR value.



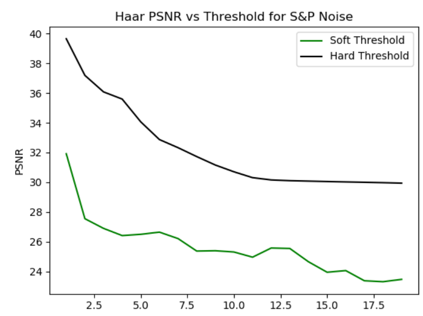
(a) DB4 w Gaussian Noise



(b) DB4 w S&P Noise



(c) HWT w Gaussian Noise



(d) HWT w S&P Noise.

Figure 6: PSNR vs. Thresholding Value

3 Results

The following Table 1 depicts the PSNR results over each filter on their corresponding Gaussian and S&P noisy images.

Filter	PSNR w/ Gaussian Noise	PSNR w/ SP Noise
Blur	33.757	48.465
Gaussian Blur	33.995	48.465
Median	34.066	48.465
Bilateral	34.205	37.723
HWT Soft Threshold	30.456	31.906
HWT Hard Threshold	32.59	39.446
DB4 Soft Threshold	26.179	25.674
DB4 Hard Threshold	27.219	26.506

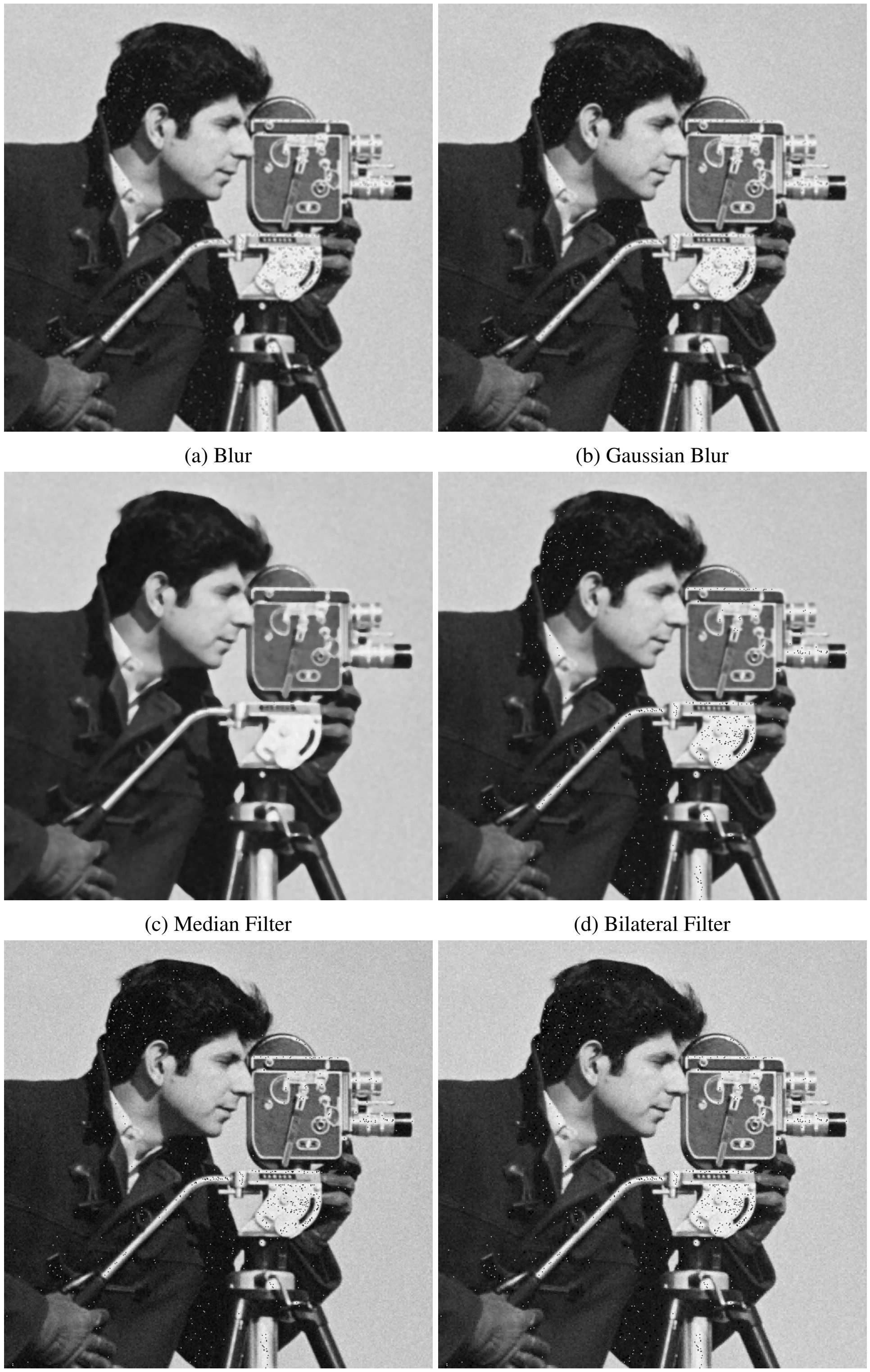
Table 1: PSNR Filter Results

3.1 Gaussian Noise Results

From observing the PSNR w/ Gaussian Noise we can see that the Bilateral filter appears to yield the highest PSNR with it implementing a filter of length 5. At first this appears strange due to Bilaterally Filtered image in Figure 7 not appearing to have the most aesthetically pleasing look to it, however, when reviewed, the PSNR simply returns the "Peak" signal to noise ratio compared to that of the original non-noisy image, and this is not a strong indicator of human aesthetics. This being said, the effects of all of the filters, aside from the DB4 filters, on the Gaussian noise produce a similar PSNR, and this is due to the fact that the Gaussian noise corrupts the original signals PSNR large enough that in order for a filter to yield a greater PSNR it must have a filter length greater than 1 in order to smooth out some of the Gaussian noise. In the case of the Soft and Hard HWT Thresholded images the hard threshold visually and numerically retains the edges in the image, and thus, yields a higher PSNR. The noise in the DB4 transformed images seems to visually appear slightly more reduced than in the HWT images, but the PSNR is significantly lower. This is due to the fact that in the DB4 transform the coefficients, seen previously in equation (6), are scaled significantly larger and significantly smaller than in the HWT and therefore it numerically effects the images pixel values in a greater capacity which results in a higher pixel error in the calculation of the MSE for the PSNR.

3.2 S&P Noise Results

The PSNR results from the S&P Noise appears to validate the claims made in the previous section on the Gaussian Noise results which is that the intrinsic PSNR of the noisy images vastly differ from the Gaussian noisy image to that of the S&P noisy image. The S&P PSNR values from Table 1 illustrate that the original PSNR of the image is the maximum PSNR without any additional filtering methods. The filter lengths implemented in the $Blur$, $Gaussian\ Blur$, and $Median$ filters are of length 1, which computationally wise does nothing to the output image. It is not until the S&P percentage or the original S&P noisy image becomes larger, upwards of 10%, that the filter lengths begin to change for the standard filters in order to lower the PSNR value. The HWT results in a higher PSNR because it is essentially applying a smoothing filter to pairwise pixel values and therefore reducing some S&P noise, however, it is also pairwise differentiating values which results in a difference in the signal power from the original image. Finally, the DB4 filter appears to have little to no effect on the visuals of the image, and it significantly decreases the outputted PSNR value. The reason for this decrease in value is similar to that in the Gaussian Noise case in that since the original S&P noisy image has such few corrupted pixels that inducing a scaling coefficient from the DB4 filter and then thresholding the image will result and a higher deviated output image than the original.



(a) Blur (b) Gaussian Blur

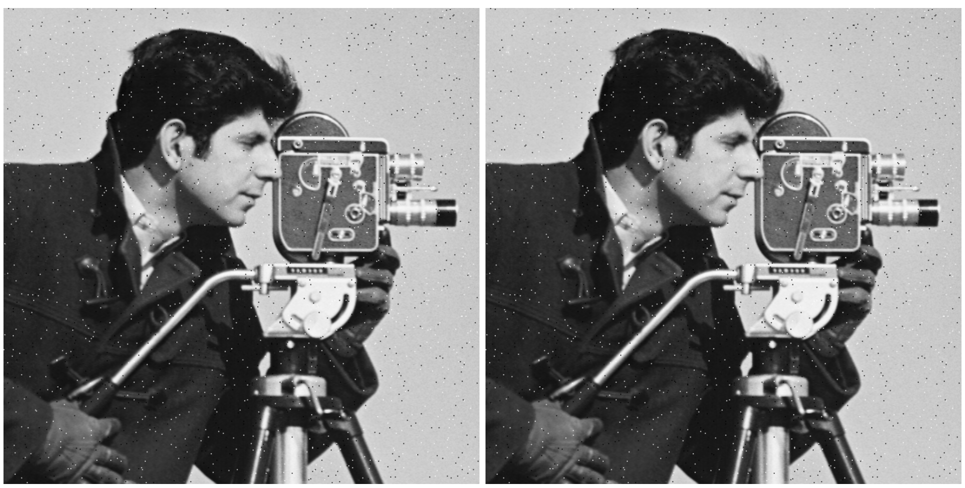
(c) Median Filter (d) Bilateral Filter

(e) Soft Threshold HWT (f) Hard Threshold HWT

Figure 7: Filter Outputs on Gaussian Noise

(a) Soft Threshold DB4T (b) Hard Threshold DB4T

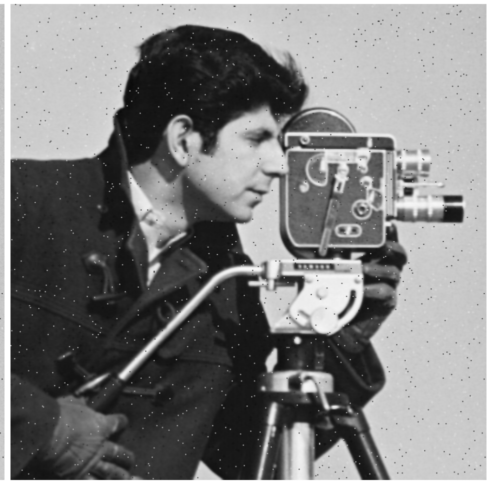
Figure 8: Filter Outputs on Gaussian Noise cont.

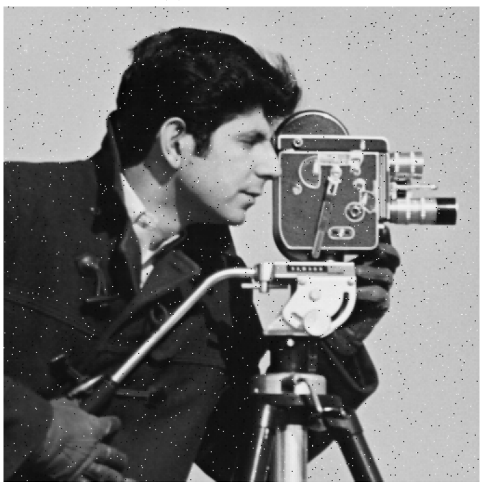


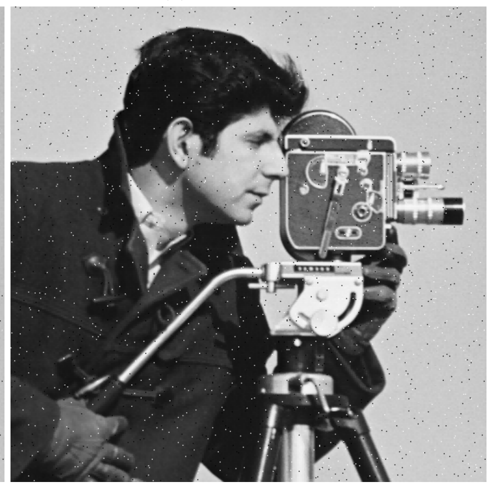
(a) Blur (b) Gaussian Blur

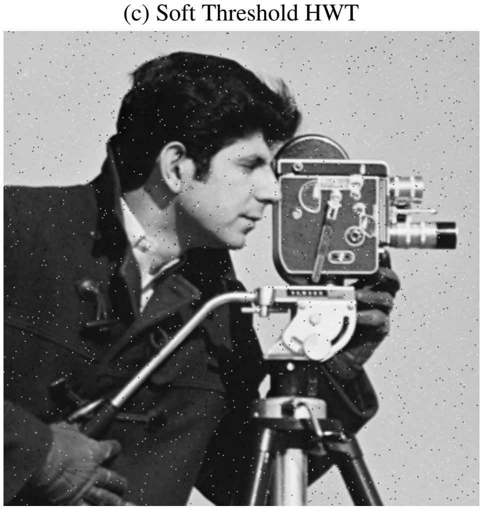
Figure 9: Filter Outputs on S&P Noise


(a) Median Filter


(b) Bilateral Filter


(c) Soft Threshold HWT


(d) Hard Threshold HWT


(e) Soft Threshold DB4T

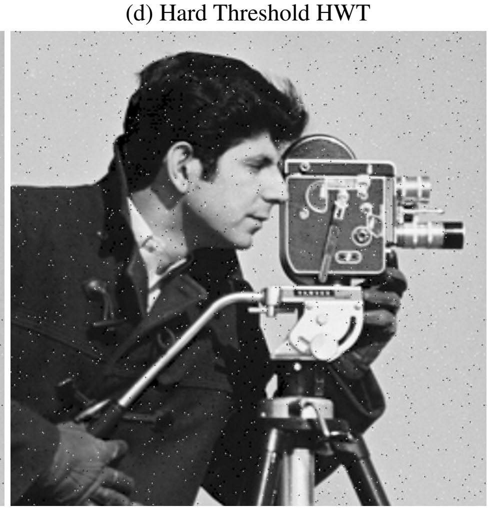

(f) Hard Threshold DB4T

Figure 10: Filter Outputs on S&P Noise cont.

4 Conclusion

In the present work, multiple filters are evaluated from observations of their corresponding filtered output image's PSNR values. The standard library $Blur$, $Gaussian\ Blur$, $Median$, and $Bilateral$ filters have been specifically tuned in filter length size in order to produce the highest outputted PSNR value. Along with these filters, the HWT and the DB4 filters have been tuned via a thresholding parameter in order to yield a larger PSNR output. From this paper one can conclude that the PSNR value has little impact on the visual "goodness" of an image, but rather, it is a method of comparing two similar images and the noise distribution. In review of this project a more in depth analysis on different wavelet techniques would be beneficial to see how changing wavelet taps/coefficients effects outputted images. Also, a different method other than the PSNR to compare the outputted image to the original would be desired. Potentially looking at the Structural Similarity Index of the images, which is a method of measuring image quality similarity between two images could provide more insight into a more appropriate approach of evaluation.

5 References

References

[1] Piotr Porwik and Agnieszka Lisowska. The haar-wavelet transform in digital image processing: its status and achievements. *Machine graphics and vision*, 13(1/2):79–98, 2004.

[2] Madhumita Sengupta and JK Mandal. An authentication technique in frequency domain through daubechies transformation (atfdd). *International Journal of Advanced Research in Computer Science*, 3(4), 2012.

6 Software Lisiting

6.1 dataSetup

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
import pandas as pd
import csv
import pywt
import time
import DWT as dwt
#%%
img = cv2.imread('data/messi5.jpg', 0)
cman = cv2.imread("data/cman.png", 0)
cman = cv2.resize(cman, (512, 512))
cv2.imwrite('data/cman_512_512.png',cman)
N_rows, N_cols = cman.shape
#%%Noise
#Gaussian
def genNoisy():
    mu = 0
    std = 10
    cman_gnoise = np.copy(cman)
    gauss_noise = np.uint8(np.random.normal(loc=mu, scale=std,size=(
        N_rows, N_cols)))
    std_gauss   = np.std(gauss_noise)
    cman_gnoise = cman + gauss_noise
#    cman_gnoise = np.uint8(np.clip(cman_gnoise, 0, 255))
    cman_gnoise = np.uint8(cman_gnoise)
    #Salt and Pepper
    s_vs_p = 0.5
    amount = 0.008
    cman_spnoise = np.copy(cman)
    num_salt_pepper = np.ceil(amount * cman.size * s_vs_p)
    s_coords = [np.random.randint(0, i - 1, int(num_salt_pepper)) for i
        in cman.shape]
    p_coords = [np.random.randint(0, i - 1, int(num_salt_pepper)) for i
        in cman.shape]
    cman_spnoise[s_coords] = 255
    cman_spnoise[p_coords] = 0
    return(cman_gnoise, cman_spnoise, std)


#%%
def pltFilters(g_out, sp_out):
    for filter in g_out:
        str_title = filter + ' Gaussian'
        plt.figure()
        plt.title(str_title)
        plt.imshow(g_out[filter], cmap='gray')

    for filter in sp_out:
        str_title = filter + ' S&P'
        plt.figure()
        plt.title(str_title)
        plt.imshow(sp_out[filter], cmap='gray')

#%%
def detFilterLength(img_gnoise, img_spnoise, img):

    g_out = {'blur' : np.zeros(0), 'gblur' : np.zeros(0),
             'median' : np.zeros(0), 'bilateral' : np.zeros(0)}

```

```
    sp_out = {'blur' : np.zeros(0), 'gblur' : np.zeros(0),
              'median' : np.zeros(0), 'bilateral' : np.zeros(0)}

    filter_length_temp = np.ones((len(g_out), 2), dtype=int)
    filter_length_best = np.ones((len(g_out), 2), dtype=int)
    psnr_best = np.zeros((len(g_out), 2))

    def detFilterPSNR(g_out, sp_out, img):
        psnr_curr = np.zeros((len(g_out),2))
        i = 0
        for filter in g_out:
            mse = mean_squared_error(img, g_out[filter])
            psnr_curr[i,0] = 10*np.log10((g_out[filter].max()**2)/mse)
            i += 1

        i = 0
        for filter in sp_out:
            mse = mean_squared_error(img, sp_out[filter])
            psnr_curr[i,1] = 10*np.log10((sp_out[filter].max()**2)/mse)
            i += 1

        return(psnr_curr)

    i = 0
    psnr_hist = []
    while(i < 5):
        g_out['blur'], g_out['gblur'], g_out['median'], g_out['bilateral'] = standardFilters(img_gnoise, filter_length_temp, 0)
        sp_out['blur'], sp_out['gblur'], sp_out['median'], sp_out['bilateral'] = standardFilters(img_spnoise, filter_length_temp, 1)
        psnr_curr = detFilterPSNR(g_out, sp_out, img)
        psnr_hist.append(psnr_curr)
        for j in range(len(filter_length_temp)):
            if(psnr_curr[j,0] > psnr_best[j,0]):
                psnr_best[j,0] = psnr_curr[j,0]
                filter_length_best[j,0] = filter_length_temp[j,0]

            if(psnr_curr[j,1] > psnr_best[j,1]):
                psnr_best[j,1] = psnr_curr[j,1]
                filter_length_best[j,1] = filter_length_temp[j,1]

        filter_length_temp = np.add(filter_length_temp, 2)
        i += 1

    return(filter_length_best, psnr_best, psnr_hist)
#%%
#Peak Signal to Noise Ratio
def peakSNR(g_out, sp_out, img):
    psnr_output = pd.DataFrame(pd.np.zeros((len(g_out), 2)))
    i = 0
    for filter in g_out:
        mse = mean_squared_error(img, g_out[filter])
        psnr_output.iloc[i,0] = np.round(10*np.log10((g_out[filter].max()**2)/mse), 3)
        i += 1

    i = 0
    for filter in sp_out:
        mse = mean_squared_error(img, sp_out[filter])
        psnr_output.iloc[i,1] = np.round(10*np.log10((sp_out[filter].max()**2)/mse), 3)
        psnr_output.rename({i: str(filter)}, axis='index')
        i += 1

```

```python
    print(psnr_output)
    psnr_output.to_csv('psnr.csv', header=False)


#%%
def detThreshold(HWT_g, HWT_sp, DB4T_g, DB4T_sp, std, img):
    N = 20
    psnr = np.zeros((N,8))
    T = np.zeros(8)

    def detPeakSNR(iHWTg_soft, iHWTg_hard, iHWTsp_soft, iHWTsp_hard,
        iDB4Tg_soft, iDB4Tg_hard, iDB4Tsp_soft, iDB4Tsp_hard,
                    img, psnr, i):
        #HAAR WAVELETS
        mse = mean_squared_error(img, iHWTg_soft)
        psnr[i,0] = 10*np.log10((iHWTg_soft.max()**2)/mse)

        mse = mean_squared_error(img, iHWTg_hard)
        psnr[i,1] = 10*np.log10((iHWTg_hard.max()**2)/mse)

        mse = mean_squared_error(img, iHWTsp_soft)
        psnr[i,2] = 10*np.log10((iHWTsp_soft.max()**2)/mse)

        mse = mean_squared_error(img, iHWTsp_hard)
        psnr[i,3] = 10*np.log10((iHWTsp_hard.max()**2)/mse)

# =============================================================================
        #DB4 WAVELETS
        mse = mean_squared_error(img, iDB4Tg_soft)
        psnr[i,4] = 10*np.log10((iDB4Tg_soft.max()**2)/mse)

        mse = mean_squared_error(img, iDB4Tg_hard)
        psnr[i,5] = 10*np.log10((iDB4Tg_hard.max()**2)/mse)

        mse = mean_squared_error(img, iDB4Tsp_soft)
        psnr[i,6] = 10*np.log10((iDB4Tsp_soft.max()**2)/mse)

        mse = mean_squared_error(img, iDB4Tsp_hard)
        psnr[i,7] = 10*np.log10((iDB4Tsp_hard.max()**2)/mse)

        return (psnr)

    i = 1
    while(i < N):
        #HAAR WAVELETS
        HWT_g_hard = pywt.threshold(HWT_g, 0.5*i*std, 'hard')
        HWT_g_soft = pywt.threshold(HWT_g, 0.5*i*std, 'soft')

        iHWTg_hard = dwt.TwoD_IHWT(HWT_g_hard, 1)
        iHWTg_soft = dwt.TwoD_IHWT(HWT_g_soft, 1)

        HWT_sp_hard = pywt.threshold(HWT_sp, 0.5*i*std, 'hard')
        HWT_sp_soft = pywt.threshold(HWT_sp, 0.5*i*std, 'soft')

        iHWTsp_hard = dwt.TwoD_IHWT(HWT_sp_hard, 1)
        iHWTsp_soft = dwt.TwoD_IHWT(HWT_sp_soft, 1)
# =============================================================================
        #DB4 WAVELETS
        temp1 = np.array(pywt.threshold(DB4T_g[0], 0.5*i*std, 'soft'))
        temp2 = np.array(pywt.threshold(DB4T_g[1], 0.5*i*std, 'soft'))

```

```python
        temp3 = np.array(pywt.threshold(DB4T_sp[0], 0.5*i*std, 'soft'))
        temp4 = np.array(pywt.threshold(DB4T_sp[1], 0.5*i*std, 'soft'))

        iDB4Tg_soft = pywt.idwt2((temp1,(temp2)),'db4')
        iDB4Tsp_soft = pywt.idwt2((temp3,(temp4)),'db4')

        temp1 = np.array(pywt.threshold(DB4T_g[0], 0.5*i*std, 'hard'))
        temp2 = np.array(pywt.threshold(DB4T_g[1], 0.5*i*std, 'hard'))

        temp3 = np.array(pywt.threshold(DB4T_sp[0], 0.5*i*std, 'hard'))
        temp4 = np.array(pywt.threshold(DB4T_sp[1], 0.5*i*std, 'hard'))

        iDB4Tg_hard = pywt.idwt2((temp1,(temp2)),'db4')
        iDB4Tsp_hard = pywt.idwt2((temp3,(temp4)),'db4')

        psnr = detPeakSNR(iHWTg_soft, iHWTg_hard, iHWTsp_soft, iHWTsp_hard, iDB4Tg_soft,
                          iDB4Tg_hard, iDB4Tsp_soft, iDB4Tsp_hard, img, psnr, i)
        i += 1

    psnr[0,:] = np.nan
    for i in range(psnr.shape[1]):
        T[i] = np.nanargmax(psnr[:,i])

    return(psnr, T)
#%%
def pltDetThreshold(psnr):
    plt.figure()
    plt.title('Haar PSNR vs Threshold for Gaussian Noise')
    plt.ylabel('PSNR')
    plt.xlabel('Threshold*0.5*$\sigma$')
    plt.plot(psnr[:,0], color='green', label='Soft Threshold')
    plt.plot(psnr[:,1], color='black', label='Hard Threshold')
    plt.legend()

    plt.figure()
    plt.title('Haar PSNR vs Threshold for S&P Noise')
    plt.ylabel('PSNR')
    plt.xlabel('Threshold*0.5*$\sigma$')
    plt.plot(psnr[:,2], color='green', label='Soft Threshold')
    plt.plot(psnr[:,3], color='black', label='Hard Threshold')
    plt.legend()

    plt.figure()
    plt.title('DB4 PSNR vs Threshold for Gaussian Noise')
    plt.ylabel('PSNR')
    plt.xlabel('Threshold*0.5*$\sigma$')
    plt.plot(psnr[:,4], color='green', label='Soft Threshold')
    plt.plot(psnr[:,5], color='black', label='Hard Threshold')
    plt.legend()

    plt.figure()
    plt.title('DB4 PSNR vs Threshold for S&P Noise')
    plt.ylabel('PSNR')
    plt.xlabel('Threshold*0.5*$\sigma$')
    plt.plot(psnr[:,6], color='green', label='Soft Threshold')
    plt.plot(psnr[:,7], color='black', label='Hard Threshold')
    plt.legend()

#%%
def pltPSNRFilters(psnr_hist):
    blur_psnr_g      = np.zeros(len(psnr_hist))
    gblur_psnr_g     = np.zeros(len(psnr_hist))
    median_psnr_g    = np.zeros(len(psnr_hist))
```

```
    bilateral_psnr_g  = np.zeros(len(psnr_hist))
    blur_psnr_sp      = np.zeros(len(psnr_hist))
    gblur_psnr_sp     = np.zeros(len(psnr_hist))
    median_psnr_sp    = np.zeros(len(psnr_hist))
    bilateral_psnr_sp = np.zeros(len(psnr_hist))
    xaxis             = np.zeros(len(psnr_hist))
    k = 1
    i = 0
    while(i < len(psnr_hist)):
        j = 0
        blur_psnr_g[i]       = psnr_hist[i][0,j]
        gblur_psnr_g[i]      = psnr_hist[i][1,j]
        median_psnr_g[i]     = psnr_hist[i][2,j]
        bilateral_psnr_g[i]  = psnr_hist[i][3,j]
        j += 1
        blur_psnr_sp[i]      = psnr_hist[i][0,j]
        gblur_psnr_sp[i]     = psnr_hist[i][1,j]
        median_psnr_sp[i]    = psnr_hist[i][2,j]
        bilateral_psnr_sp[i] = psnr_hist[i][3,j]
        xaxis[i]             = k
        k += 2
        i += 1


    plt.figure()
    plt.title('PSNR vs Filter Length w/ Gaussian Noise')
    plt.xlabel('Filter Length')
    plt.ylabel('PSNR (dB)')
    plt.plot(range(1,11,2), blur_psnr_g,      label='blur'     )
    plt.plot(range(1,11,2), gblur_psnr_g,     label='gblur'    )
    plt.plot(range(1,11,2), median_psnr_g,    label='median'   )
    plt.plot(range(1,11,2), bilateral_psnr_g, label='bilateral')
    plt.legend()

    plt.figure()
    plt.title('PSNR vs Filter Length w/ S&P Noise')
    plt.xlabel('Filter Length')
    plt.xlim(0,5)
    plt.xticks(xaxis)
    plt.ylim(30, 52)
    plt.ylabel('PSNR (dB)')
    plt.plot(range(1,11,2), blur_psnr_sp,      label='blur'     )
    plt.plot(range(1,11,2), gblur_psnr_sp,     label='gblur'    )
    plt.plot(range(1,11,2), median_psnr_sp,    label='median'   )
    plt.plot(range(1,11,2), bilateral_psnr_sp ,label='bilateral')
    plt.legend()

#%%Filtering
def standardFilters(img, f_length, j):
    blur = cv2.blur(img,(f_length[0,j],f_length[0,j]))
    gblur = cv2.GaussianBlur(img,(f_length[1,j],f_length[1,j]),0)
    median = cv2.medianBlur(img,f_length[2,j])
    bilateral = cv2.bilateralFilter(img,f_length[3,j],75,75)
    return(blur, gblur, median, bilateral)
#%%Save Images
def outputImages(g_out, sp_out, cman_gnoise, cman_spnoise):
    cv2.imwrite('data/cman_gnoise.png', cman_gnoise)
    cv2.imwrite('data/cman_spnoise.png', cman_spnoise)
    for filter in g_out:
        file_path = 'data/' + filter + '_g.png'
        cv2.imwrite(file_path, g_out[filter])
    for filter in sp_out:
        file_path = 'data/' + filter + '_sp.png'
        cv2.imwrite(file_path, sp_out[filter])

```

```python
#%%MAIN
# ================================================================================
s_time = time.clock()
cman_gnoise, cman_spnoise, std = genNoisy()
iterations = 1
g_out = {'blur' : np.zeros(0), 'gblur' : np.zeros(0),
            'median' : np.zeros(0), 'bilateral' : np.zeros(0), 'IHWT_soft' : np.zeros(0),
            'IHWT_hard' : np.zeros(0), 'IDB4T_soft' : np.zeros(0), 'IDB4T_hard' : np.zeros(0)}

sp_out = {'blur' : np.zeros(0), 'gblur' : np.zeros(0),
            'median' : np.zeros(0), 'bilateral' : np.zeros(0), 'IHWT_soft' : np.zeros(0),
            'IHWT_hard' : np.zeros(0), 'IDB4T_soft' : np.zeros(0), 'IDB4T_hard' : np.zeros(0)}

HWT_g = dwt.TwoD_HWT(cman_gnoise, iterations)
HWT_sp = dwt.TwoD_HWT(cman_spnoise, iterations)
DB4T_g = pywt.dwt2(cman_gnoise, 'db4')
DB4T_sp = pywt.dwt2(cman_spnoise, 'db4')

#This section need only be run once per new image to attain the appropriate Threshold value
# ================================================================================
# psnr, T = detThreshold(HWT_g, HWT_sp, DB4T_g, DB4T_sp, std, cman)
# pltDetThreshold(psnr)
# ================================================================================

f_length, psnr_best, psnr_hist = detFilterLength(cman_gnoise, cman_spnoise, cman)
#pltPSNRFilters(psnr_hist)

g_out['blur'], g_out['gblur'], g_out['median'], g_out['bilateral'] = standardFilters(cman_gnoise, f_length, 0)
sp_out['blur'], sp_out['gblur'], sp_out['median'], sp_out['bilateral'] = standardFilters(cman_spnoise, f_length, 1)

#Gaussian Noise Thresholding and Inverese Haar Transform
HWT_soft_g = pywt.threshold(HWT_g, T[0]*0.5*std, 'soft')
HWT_hard_g = pywt.threshold(HWT_g, T[1]*0.5*std, 'hard')
g_out['IHWT_soft']= dwt.TwoD_IHWT(HWT_soft_g, iterations)
g_out['IHWT_hard'] = dwt.TwoD_IHWT(HWT_hard_g, iterations)



#Salt & Pepper Noise, Thresholding, and Inverese Haar Transform
HWT_soft_sp = pywt.threshold(HWT_sp, T[2]*0.5*std, 'soft')
HWT_hard_sp = pywt.threshold(HWT_sp, T[3]*0.5*std, 'hard')
sp_out['IHWT_soft']= dwt.TwoD_IHWT(HWT_soft_sp, iterations)
sp_out['IHWT_hard'] = dwt.TwoD_IHWT(HWT_hard_sp, iterations)

#Gaussian Noise and S&P Thresholding for Inverese DB4 Transform
temp1 = np.array(pywt.threshold(DB4T_g[0], T[4]*0.5*std, 'soft'))
temp2 = np.array(pywt.threshold(DB4T_g[1], T[4]*0.5*std, 'soft'))

temp3 = np.array(pywt.threshold(DB4T_sp[0], T[6]*0.5*std, 'soft'))
temp4 = np.array(pywt.threshold(DB4T_sp[1], T[6]*0.5*std, 'soft'))
```

```
g_out['IDB4T_soft'] = pywt.idwt2((temp1,(temp2)),'db4')
sp_out['IDB4T_soft'] = pywt.idwt2((temp3,(temp4)),'db4')

temp1 = np.array(pywt.threshold(DB4T_g[0], T[5]*0.5*std, 'hard'))
temp2 = np.array(pywt.threshold(DB4T_g[1], T[5]*0.5*std, 'hard'))

temp3 = np.array(pywt.threshold(DB4T_sp[0], T[7]*0.5*std, 'hard'))
temp4 = np.array(pywt.threshold(DB4T_sp[1], T[7]*0.5*std, 'hard'))

g_out['IDB4T_hard'] = pywt.idwt2((temp1,(temp2)),'db4')
sp_out['IDB4T_hard'] = pywt.idwt2((temp3,(temp4)),'db4')

peakSNR(g_out, sp_out, cman)
#pltFilters(g_out, sp_out)
outputImages(g_out, sp_out, cman_gnoise, cman_spnoise)

e_time = time.clock()
t_time = e_time - s_time
print('\n', 'Total_Time: ', round(t_time,2))
# ===========================================================================


#%%
```

6.2 DWT

```
import numpy as np
#%%Haar Transform
# ================================================================================
def OneD_HWT(x):
     w = np.array([0.5, -0.5])
     s = np.array([0.5, 0.5])
#     w = np.array([np.sqrt(2)/2, -np.sqrt(2)/2])
#     s = np.array([np.sqrt(2)/2, np.sqrt(2)/2])
     temp_vector = np.float64((np.zeros(len(x))))
     h = np.int(len(temp_vector)/2)
     i = 0
     while(i < h):
         k = 2*i
         temp_vector[i] = x[k]*s[0] + x[k+1]*s[1]
         temp_vector[i+h] = x[k]*w[0] + x[k+1]*w[1]
         i += 1
     return(temp_vector)
 #2D Haar Transform
def TwoD_HWT(x, iterations):
     temp = np.float64(np.copy(x))
     N_rows, N_cols = temp.shape

     k = 0
     while(k < iterations):
         lev = 2**k
         lev_Rows = N_rows/lev
         lev_Cols = N_cols/lev
         row = np.zeros(np.int(lev_Cols))
         i = 0
         while(i < lev_Rows):
             row = temp[i,:]
             temp[i,:] = OneD_HWT(row)
             i += 1

         col = np.zeros(np.int(lev_Rows))
         j = 0
         while(j < lev_Cols):
             col = temp[:,j]
             temp[:,j] = OneD_HWT(col)
             j += 1

         k += 1

 #     temp = np.clip(temp, 0, 255)
     return(temp)
# ================================================================================

#%%Inverse Haar Transform
# ================================================================================
def OneD_IHWT(x):
     w = np.array([0.5, -0.5])
     s = np.array([0.5, 0.5])
#     w = np.array([np.sqrt(2)/2, -np.sqrt(2)/2])
#     s = np.array([np.sqrt(2)/2, np.sqrt(2)/2])
     temp_vector = np.float64(np.copy(x))
     h = np.int(len(temp_vector)/2)
     i = 0
```

```python
    while(i < h):
        k = 2*i
        temp_vector[k] = (x[i]*s[0] + x[i+h]*w[0])/w[0]
        temp_vector[k+1] = (x[i]*s[1] + x[i+h]*w[1])/s[0]
        i += 1
    return(temp_vector)

#2D Inverse Haar Transform
def TwoD_IHWT(x, iterations):
    temp = np.float64(np.copy(x))
    N_rows, N_cols = temp.shape
    k = iterations - 1


    while(k >= 0):
        lev = 2**k
        lev_Cols = N_cols/lev
        lev_Rows = N_rows/lev
        col = np.zeros(np.int(lev_Rows))

        j = 0
        while(j < lev_Cols):
            col = temp[:,j]
            temp[:,j] = OneD_IHWT(col)
            j += 1

        row = np.zeros(np.int(lev_Cols))
        i = 0
        while(i < lev_Rows):
            row = temp[i,:]
            temp[i,:] = OneD_IHWT(row)
            i += 1

        k -= 1
    temp = np.clip(temp, 0, 255)
    return(np.uint8(temp))
# ============================================================================

# ============================================================================
def OneD_DB4(x, n):
    h = np.zeros(4)
    g = np.zeros(len(h))
    h[0] = (1+np.sqrt(3))/(4*np.sqrt(2))
    h[1] = (3+np.sqrt(3))/(4*np.sqrt(2))
    h[2] = (3-np.sqrt(3))/(4*np.sqrt(2))
    h[3] = (1-np.sqrt(3))/(4*np.sqrt(2))
    g[0] = h[3]
    g[1] = -h[2]
    g[2] = h[1]
    g[3] = -h[0]

    temp_vector = np.float64((np.zeros(len(x))))
    half = np.int(len(temp_vector)/2)
    i = 0
    j = 0
    while(j < n-3):
        temp_vector[i] = x[j]*h[0] + x[j+1]*h[1] + x[j+2]*h[2] + x[j+3]*h[3]
        temp_vector[i+half] = x[j]*g[0] + x[j+1]*g[1] + x[j+2]*g[2] + x[j+3]*g[3]
        j += 2
```

```
        i += 1

    temp_vector[i] = x[n-2]*h[0] + x[n-1]*h[1] + x[0]*h[2] + x[1]*h[3]
    temp_vector[i+half] = x[n-2]*g[0] + x[n-1]*g[1] + x[0]*g[2] + x[1]*g[3]
    return(temp_vector)
#2D Daubechies Transform
def TwoD_DB4(x, iterations):
    temp = np.float64(np.copy(x))
    N_rows, N_cols = temp.shape

    k = 0
    while(k < iterations):
        lev = 2**k
        lev_Rows = N_rows/lev
        lev_Cols = N_cols/lev
        row = np.zeros(np.int(lev_Cols))
        i = 0
        while(i < lev_Rows):
            n = len(x)
            while(n >= 4):
                row = temp[i,:]
                temp[i,:] = OneD_DB4(row, n)
                n = np.int(n/2)
            i += 1

        col = np.zeros(np.int(lev_Rows))
        j = 0
        while(j < lev_Cols):
            n = len(x)
            while(n >= 4):
                col = temp[:,j]
                temp[:,j] = OneD_DB4(col, n)
                n = np.int(n/2)

            j += 1

        k += 1

#      temp = np.clip(temp, 0, 255)
    return(temp)
#==============================================================================
    #%%Inverse Daubechies Transform
#==============================================================================
def OneD_IDB4(x, n):
    h = np.zeros(4)
    g = np.zeros(len(h))
    h[0] = (1+np.sqrt(3))/(4*np.sqrt(2))
    h[1] = (3+np.sqrt(3))/(4*np.sqrt(2))
    h[2] = (3-np.sqrt(3))/(4*np.sqrt(2))
    h[3] = (1-np.sqrt(3))/(4*np.sqrt(2))
    g[0] = h[3]
    g[1] = -h[2]
    g[2] = h[1]
    g[3] = -h[0]

    temp_vector = np.float64(np.copy(x))
    half = np.int(len(temp_vector)/2)

    temp_vector[0] = x[half-1]/h[2] + x[n-1]/g[2] + x[0]/h[0] + x[half]/h[3]
```

```
    temp_vector[1] = x[half-1]/h[3] + x[n-1]/g[3] + x[0]/h[1] + x[half]/g[1]
    i = 0
    j = 2
    while(i < half-1):
        temp_vector[j] = x[i]/h[2] + x[i+half]/g[2] + x[i+1]/h[0] + x[i+half+1]/h[3]
        j += 1
        temp_vector[j] = x[i]/h[3] + x[i+half]/g[3] + x[i+1]/h[1] + x[i+half+1]/g[1]
        j += 1
        i += 1
    return(temp_vector)

#2D Inverse Daubechies Transform
def TwoD_IDB4(x, iterations):
    temp = np.float64(np.copy(x))
    N_rows, N_cols = temp.shape
    k = iterations - 1


    while(k >= 0):
        lev = 2**k
        lev_Cols = N_cols/lev
        lev_Rows = N_rows/lev
        col = np.zeros(np.int(lev_Rows))

        j = 0
        while(j < lev_Cols):
            n = 4
            while(n <= len(x)):
                col = temp[:,j]
                temp[:,j] = OneD_IDB4(col, n)
                n = np.int(n*2)
            j += 1

        row = np.zeros(np.int(lev_Cols))
        i = 0
        while(i < lev_Rows):
            n = 4
            while(n <= len(x)):
                row = temp[i,:]
                temp[i,:] = OneD_IDB4(row, n)
                n = np.int(n*2)
            i += 1

        k -= 1
    temp = np.clip(temp, 0, 255)
    return(np.uint8(temp))
#=============================================================================
```