Exercise 1

Successfully implemented the ISTA algorithm for solving the LASSO problem using Python.

The program takes the dictionary matrix **H** and signal vector **y** as input, as well as the parameter λ, and returns the solution x. It also automatically determines the parameter α.

The function also uses error ratio concept for the number of iterations so that the function converges to give us a solution.

```
import numpy as np
def ista(y,H,lamda,alpha):

    x = np.zeros((64,1))
    T = lamda/(2*alpha)
    error_old = 1
    error_ratio = 10
    k=1
    Hx = np.zeros((64,1))
    while( error_ratio > 10^-16):
        Hx = np.dot(H,x)
        #x = wthresh(x + ((H.transpose()*(y - Hx))/alpha),'s', T)
        X = x + np.dot(np.transpose(H)/alpha,(y - Hx)/alpha)
        res = (abs(X) - T)
        res = (res + abs(res))/2
        x  = np.sign(X)*res
        error_new = np.sum(np.abs(Hx[:] - y[:]))
        #error_new = np.sum(np.abs(Hx - y)
        error_ratio = (error_old - error_new)/error_old
        error_old = error_new
        k=k+1
    #end
    return x
```

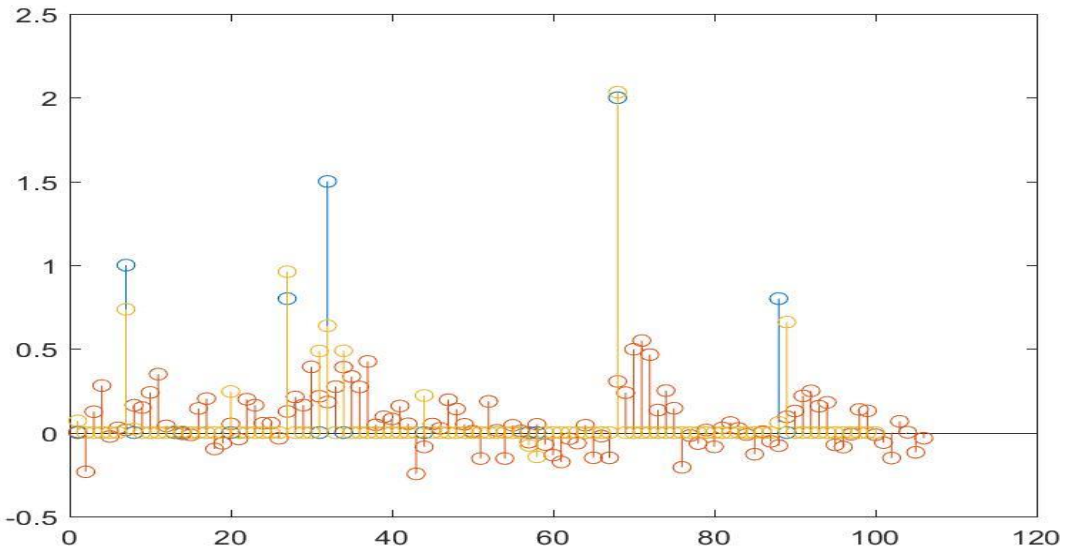
Exercise 2

Successfully implemented our own function DCT_basis_gen(N) which takes dimension N as input and returns 1D DCT basis vectors.

```
def DCT_basis_gen(N):
    h = np.zeros((N, N))
    temp=np.zeros((N))

    for k  in range (0,N):
        if k == 0:
            a = math.sqrt(1.0/N)
        else:
            a = math.sqrt(2.0/N)

        for n in range (0,N):
            temp[n] = a * (math.cos((((2*n)+1)*(k)*(math.pi))/(2*N)))

        #end
        h[0:N,k] = temp[0:N]
        #end
    return h
```

We tested the functionality using a 1D sparse array and adding noise later on

and we got back almost all of the signal back



Exercise 3

Using the results from exercise 1 and 2 and implementing the algorithm in a block wise manner we could denoise the noisy images.

Results:

Original Image:

1) **Noise = 0.01*255 and Lambda=1**

2) **Noise = 0.01*255 and Lambda=10**

3) **Noise = 0.01*255 and Lambda=10**

4) **Noise = 0.1*255 and Lambda=10**

Exercise 4

In this case the ISTA algorithm works directly on images(2D) instead of 1D vectors.

The Haar, Daubechies 8/8 wavelet are used separately as the forward and reverse transforms in the inbuilt python functions. Three level decomposition was used for both wavelets.

Element was operation (thresholding) was used in this case for smoothing out the noise.

```
while(error_ratio >0.05 ):

    coeffs = pywt.wavedecn(d,type,mode='symmetric',level=3,axes=None)
    cA= coeffs[0]

    Level3 = coeffs[1]
    V3=Level3['ad']
    H3=Level3['da']
    D3=Level3['dd']

    Level2 = coeffs[2]
    V2=Level2['ad']
    H2=Level2['da']
    D2=Level2['dd']

    Level1 = coeffs[3]
    V1=Level1['ad']
    H1=Level1['da']
    D1=Level1['dd']

    #do processing that is thresholding

    T= lamda/2

    V3=pywt.threshold(V3, T, mode='soft', substitute=0)
    H3=pywt.threshold(H3, T, mode='soft', substitute=0)
    D3=pywt.threshold(D3, T, mode='soft', substitute=0)

    V2=pywt.threshold(V2, T, mode='soft', substitute=0)
    H2=pywt.threshold(H2, T, mode='soft', substitute=0)
    D2=pywt.threshold(D2, T, mode='soft', substitute=0)

    V1=pywt.threshold(V1, T, mode='soft', substitute=0)
    H1=pywt.threshold(H1, T, mode='soft', substitute=0)
    D1=pywt.threshold(D1, T, mode='soft', substitute=0)

    new_coeffs = [cA,{'ad':V3,'da':H3,'dd':D3},{'ad':V2,'da':H2,'dd':D2},{'ad':V1,'da':H1,'dd':D1}]
    denoised_img = pywt.waverecn(new_coeffs,type,mode='symmetric',axes=None)

    d= denoised_img
```

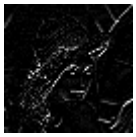
Results:

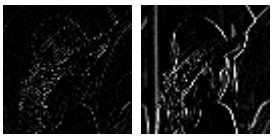
Original Image:

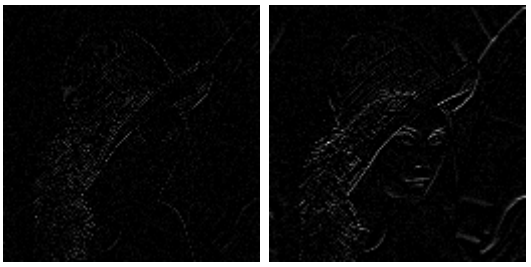
Noisy Image 0.01:

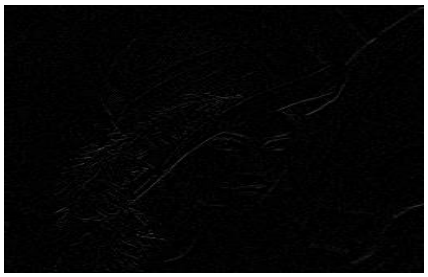
Noisy Image 0.01:

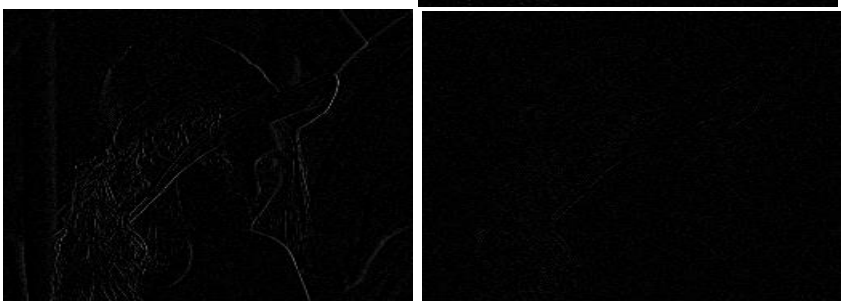
Noisy Image Wavelets:



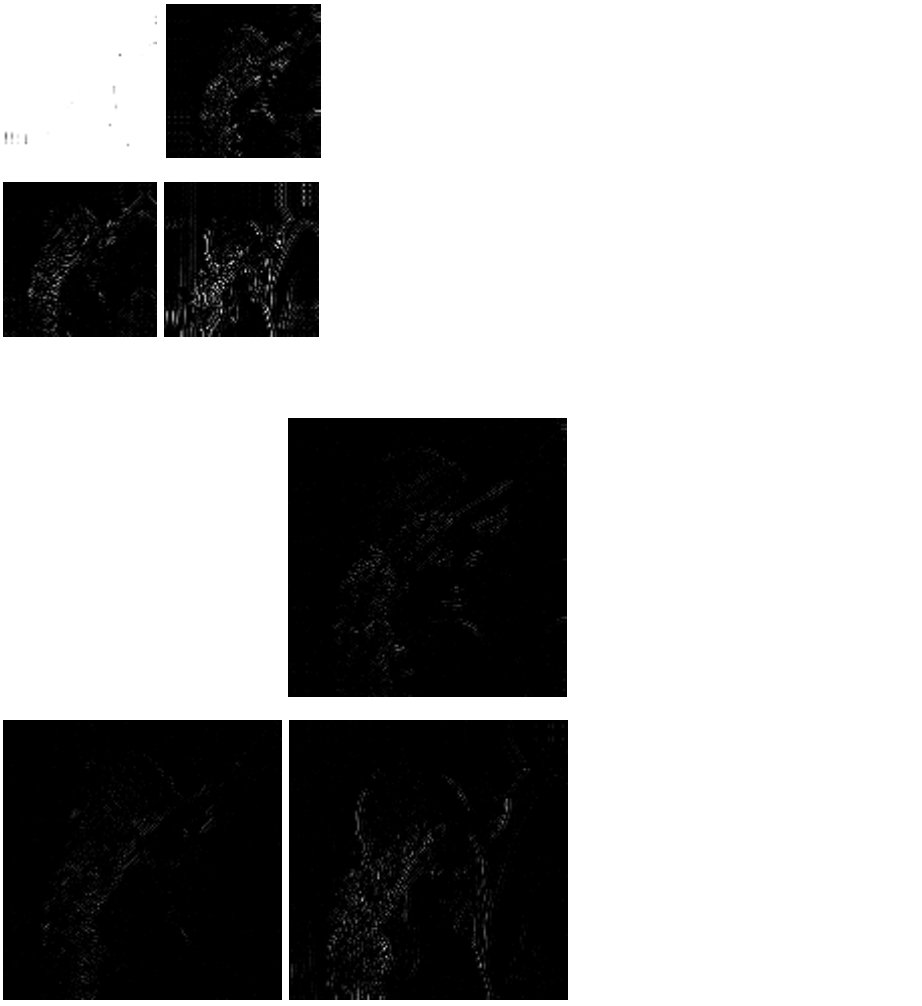




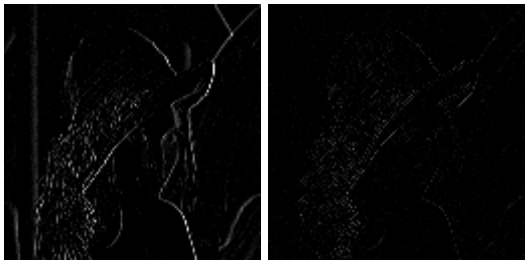


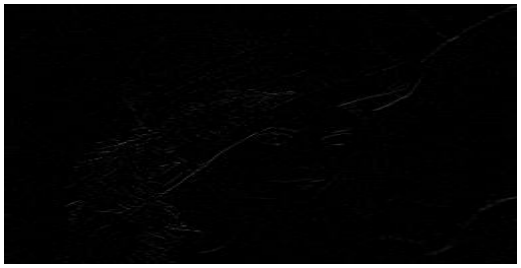


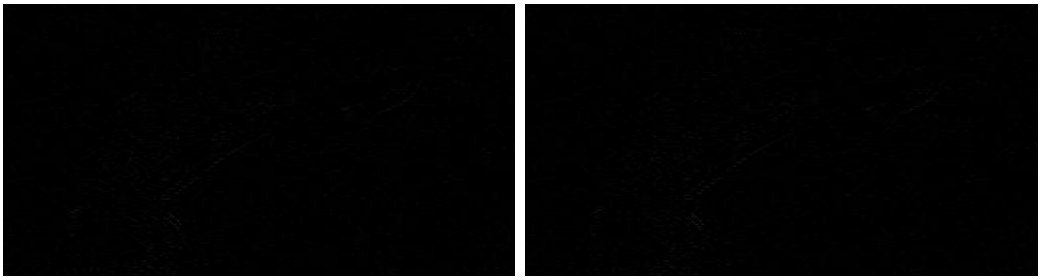
Denoised Image Wavelets: Daubechies



Denoised Image Wavelets: Haar







Final Denoised Image: Daubechies

Final Denoised Image: Daubechies

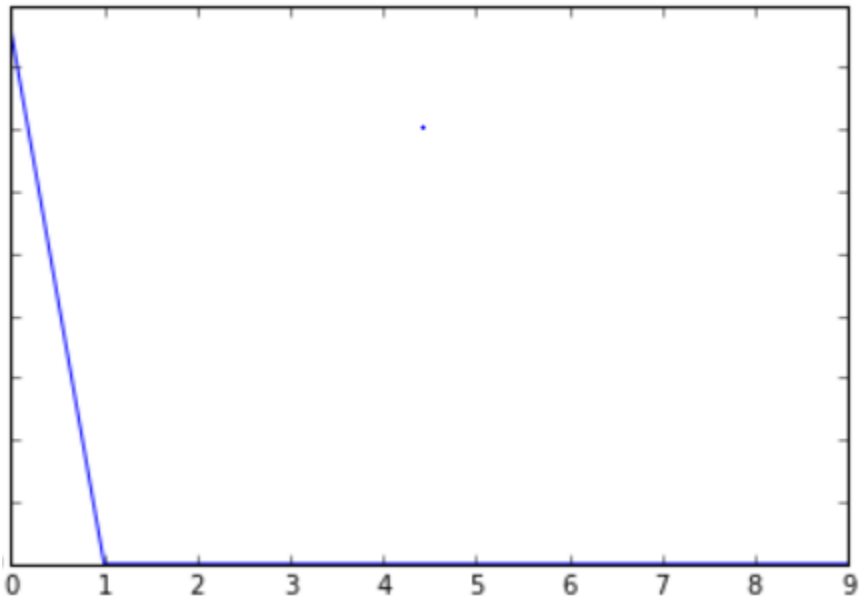
Pros and Cons of the different wavelet filters:

1) While experimenting with the filters we found that the Haar does not smooth out the images as good as the Daubechies filter. In Haar, the image tends to be pixelated.

2) In case of Daubechies there is an extra element added which made the matrices uneven and hence complicated to calculate and use, whereas Haar exactly divides the image in equal proportions and hence easy to use and calculate.

3) The number of iterations and the threshold play a role of how much affect the wavelet (both wavelets) play on the noisy image.

Plot of Error vs. iteration numbers



Note:

When we use a orthonormal transform basis as the dictionary then $\alpha = 1$

That is, the solution stays the same no matter how many iterations you go, and essentially it does soft thresholding on the coefficients. The error curve would not decrease as you go on more iterations. So, using the convergence criterion, you will stop after one iteration. The error curve would not decrease as you go on more iterations.