Guan, Hui Hua

In [1]:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import pywt
% matplotlib inline
```

Q1: ISTA using Python

In [2]:

```
def soft(x, T):
     return np.maximum(x-T,0)+np.minimum(x+T,0)
```

In [3]:

```
def ista(y, H, lam):

    k = 0
    old_err = 1
    error_ratio=1

    #HT = np.transpose(H)
    #HTH = np.dot(np.transpose(H), H)
    alpha = np.real(np.max(np.linalg.eig(np.dot(np.transpose(H), H))[0]))


    x = np.dot((0 * H), y)

    T = float(lam) / (2 * alpha)

    while error_ratio>1e-7:
        # calculate the 'cost' error
        new_err = np.linalg.norm(y - np.dot(H,x), 2) + lam * np.linalg.norm(
x,1)
        x = soft(x + (1/alpha)*np.dot(np.transpose(H), y-np.dot(H,x)) , T)

        error_ratio = (old_err - new_err)/old_err

        k+=1
        old_err = new_err

    return x
```

Q2: DCT transform and ISTA

In [4]:

```
def DCT_basis_gen(N=16):

    h=[[0 for i in range(N)] for j in range(N)]
```

```
    h = [[0 for i in range(N)] for j in range(N)]

    a0 = np.sqrt(float(1)/N)
    a_ = np.sqrt(float(2)/N)

    for k in range(N):
        for n in range(N):
            if k ==0:

                h[n][k]=a0*np.cos(((2*n+1)*k*np.pi)/(2*N))
            else:
                h[n][k]=a_*np.cos(((2*n+1)*k*np.pi)/(2*N))

    h = np.array(h)
    return h
```

Below is a test of the DCT transform and ISTA functions with a sparse vector. 1000 values of lambda will be tested; the optimal lambda will be returned.

For a sigma level of 0.01, it is observed that a larger lambda (of order 0.01) will introduce more reconstruction error.

Below a comparision of the sparse vector and the reconstructed vector can be seen. The error is almost 0 because both signals overlay each other.

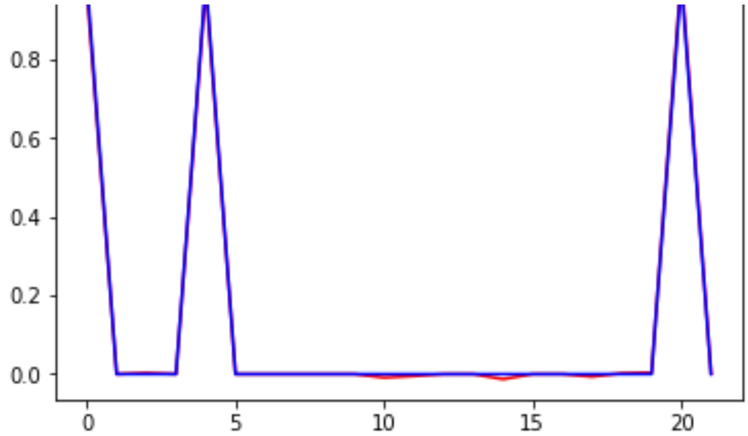
In [5]:

```
x = np.array([1, 0 , 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,0, 0, 0, 0, 0, 0, 0, 0, 1, 0
])

N= len(x)
sigma = 0.01
w = sigma*np.random.normal(0, 1,N)

lambdas = np.linspace(0.00001, 0.5, 1000)

H = DCT_basis_gen(N)
y = H.dot(x) + w

err=[]

for lam in lambdas:
    x_recover = ista(y, H, lam)
    err.append(np.abs(x_recover-x))
best_lam = lambdas[np.argmin(err)]
print("Optimal lambda = " + str(best_lam))


# Reconstruction
x_best= ista(y,H, best_lam)
plt.plot(x_best, 'r')
plt.plot(x, 'b')
```

```
Optimal lambda = 0.0175271671672
```

Out[5]:

```
[<matplotlib.lines.Line2D at 0x10e2209e8>]
```



After sweeping through 1000 values for λ, it is apparent that a larger λ (~order of 10s) will create larger reconstruction error. We find that the optimal lambda is in the order of ~0.01.

Q3: Wavelets and ISTA

In [6]:

```
def forward_transform(y,wav_type):
    # This function is only for 3-level decomp
    Hty = pywt.wavedec2(y,wav_type, level=3)
    # make into a list data structure instead of tuple
    # tuple is immutable but we want to be able to change the tuple
    HTy = []
    for i in Hty:
        HTy.append(list(i))
    return HTy
```

In [7]:

```
def inverse_transform(x, wav_type):
    # use auxillary list to format into [cAn,(cH3, cV3, cD3),(cH2, cV2, cD2
),(cH1, cV1, cD1)]
    # that is the data structure the pywt function returns

    # This function is only for 3-level rec
    aux = []
    aux.append(x[0])
    for i in range(1, 4):
        aux.append(tuple(x[i]))
    Hx = pywt.waverec2(aux, wav_type)
    #return Hx
    # This function is only for 3-level rec
    #aux = []
    #aux.append(x[0])
    #for i in x:
    #    aux.append(tuple(i))
    #Hx = pywt.waverec2(aux, wav_type)
    return Hx
```

In [28]:

```
def ista_img(y, lam, wav_type):
```

```
    # initialisations #

    alpha = 1
    x = forward_transform(y,wav_type)  # dec to get noisy wavelet 
coefficeints
    T = lam / (2 * alpha)
    error_ratio =1
    k = 0
    old_err = 1
    errs=[]

    ##

    while error_ratio>1e-7:

        Hx = inverse_transform(x, wav_type) #rec on noisy wavelet coeff

        new_err = np.linalg.norm(y - Hx, 2) # err on noisy img and recoverec
img with noisy wavlets
        errs.append(new_err)


        HTy = forward_transform(y-Hx,wav_type) # # dec to get noisy wavelet 
coefficeints

        for i in range(0, 4):
            for j in range(0, 3):
                x[i][j] = soft(HTy[i][j],T)

        # update #
        error_ratio = (old_err - new_err)/old_err
        k+=1
        old_err = new_err

    # we want to return errs and k for the error plot
    return x, errs, k, Hx
```

In [9]:

```
# Used to display the resulting plots nicely
# Adapted from stackoverflow.com/questions/36006136/how-to-display-images-i
n-a-row-with-ipython-display
def grid_display(list_of_images, list_of_titles=[], no_of_columns=3, 
figsize=(20,20)):

    fig = plt.figure(figsize=figsize)
    column = 0
    for i in range(len(list_of_images)):
        column += 1
        #  check for end of column and create a new figure
        if column == no_of_columns+1:
            fig = plt.figure(figsize=figsize)
            column = 1
        fig.add_subplot(1, no_of_columns, column)
        if '_' in list_of_titles[i]:
            plt.imshow(list_of_images[i])
        else:
            plt.imshow(list_of_images[i], cmap='gray')
        plt.axis('off')
```

```
        if len(list_of_titles) >= len(list_of_images):
            plt.title(list_of_titles[i])
```

In [29]:

```
# Test
img = cv2.imread('lena512gray.png', 0)
img=np.array(img)

lam=50

# Add noise
noisy_img = img + 0.1*255* np.random.normal(0, 1, (512,512))

x_haar, errs_haar, k_haar, Hx_haar = ista_img(noisy_img, lam, 'haar')
denoised_img_haar_50 = inverse_transform(x_haar, 'haar')

x_db8, errs_db8, k_db8, Hx_db8 = ista_img(noisy_img, lam, 'db8')
denoised_img_db8_50 = inverse_transform(x_db8, 'db8')

# Display
grid_display([img, noisy_img,denoised_img_haar,img, noisy_img,denoised_img_db8 ], list_of_titles=['Original', 'Noisy', 'Denoised Haar','Original', 'Noisy', 'Denoised Db8'], no_of_columns=3, figsize=(20,20))
```

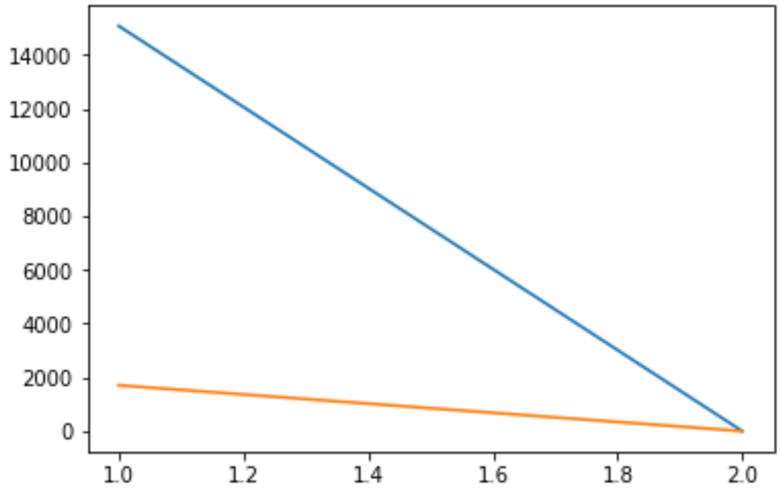
Below is a plot of the iteration errors for a tuned lambda=50

In [11]:

```
plt.plot(np.linspace(1,2, 2),errs_haar[::-1])
plt.plot(np.linspace(1,2, 2),errs_db8[::-1])
```

Out[11]:

```
[<matplotlib.lines.Line2D at 0x117116eb8>]
```



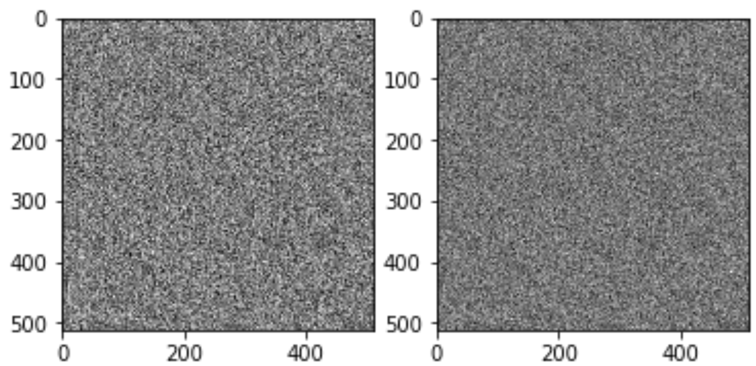
Below displays a visual of the noise removed from the noisy image

In [12]:

```
plt.subplot(121)
plt.imshow(noisy_img-denoised_img_haar, cmap='gray')
plt.subplot(122)
plt.imshow(noisy_img-denoised_img_db8, cmap='gray')
```

Out[12]:

```
<matplotlib.image.AxesImage at 0x118e9e0b8>
```



The below show_transform function will display the wavelet transform images from the coefficients.

In [13]:

```
def show_transform(cAn, cH3, cV3, cD3, cH2, cV2, cD2, cH1, cV1, cD1):
    a=cAn.shape[0]
    b=cH3.shape[0]
    c=cH2.shape[0]
    d=cH1.shape[0]
    total = a+b+c+d
    newnew = np.zeros((total,total))

    # 4
    newnew[0:a,0:a]=cAn
    # 3
    newnew[a:a+b,0:b]=cV3
    newnew[a:a+b, b:2*b]=cD3
    newnew[0:b, a:a+b]=cH3
```

```
    #2
    newnew[a+b:a+b+c, 0:c]=cV2
    newnew[a+b:a+b+c,c:2*c ]=cD2
    newnew[0:c, a+b:a+b+c]=cH2

    #1
    newnew[a+b+c:total, 0:d]=cV1
    newnew[a+b+c:total, d:2*d]=cD1
    newnew[0:d, a+b+c:total]=cH1


    return newnew
```

Wavelet Transform Images of the Denoised Images and Noisy Image

In [14]:

```
# Wavelet transform of the final denoised image (db8)
[cAn_,(cH3_, cV3_, cD3_),(cH2_, cV2_, cD2_),(cH1_, cV1_, cD1_)]=pywt.wavede
c2(denoised_img_db8,'db8', level=3)

wt_db8=show_transform(cAn_, cH3_, cV3_, cD3_, cH2_, cV2_, cD2_, cH1_, cV1_,
cD1_)
#plt.imshow(wt_db8, cmap='gray')
#plt.show()
```

In [15]:

```
# Wavelet transform of the final denoised image (haar)
[cAn,(cH3, cV3, cD3),(cH2, cV2, cD2),(cH1, cV1, cD1)]=pywt.wavedec2(denoised
_img_haar,'haar', level=3)

wt_haar=show_transform(cAn, cH3, cV3, cD3, cH2, cV2, cD2, cH1, cV1, cD1)
#plt.imshow(wt_haar, cmap='gray')
#plt.show()
```

In [16]:

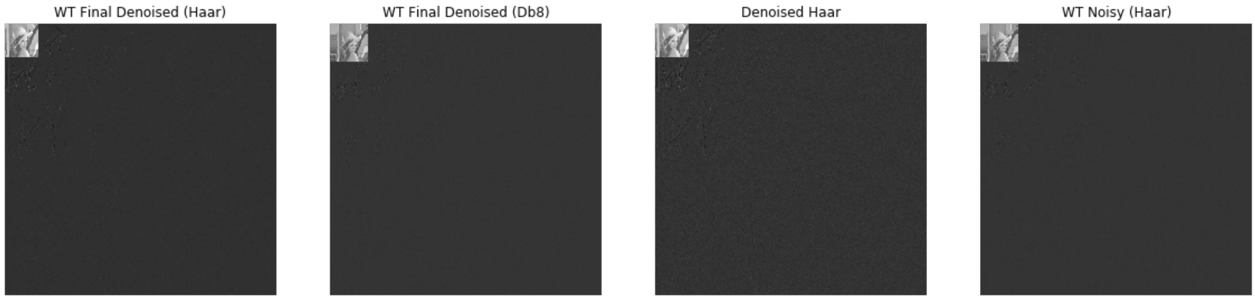
```
# Wavelet transform of the noisy image (db8)
[cAn,(cH3, cV3, cD3),(cH2, cV2, cD2),(cH1, cV1, cD1)]=pywt.wavedec2(noisy_im
g,'db8', level=3)

wt_noisy_db8=show_transform(cAn, cH3, cV3, cD3, cH2, cV2, cD2, cH1, cV1, cD1
)
#plt.imshow(wt_noisy_db8, cmap='gray')
#plt.show()
```

In [17]:

```
# Wavelet transform of the noisy image (haar)
[cAn,(cH3, cV3, cD3),(cH2, cV2, cD2),(cH1, cV1, cD1)]=pywt.wavedec2(noisy_im
g,'haar', level=3)

wt_noisy_haar=show_transform(cAn, cH3, cV3, cD3, cH2, cV2, cD2, cH1, cV1, cD
1)
#plt.imshow(wt_noisy_haar, cmap='gray')
#plt.show()
```

In [18]:

```
grid_display([wt_haar, wt_db8,wt_noisy_haar,wt_db8], list_of_titles=['WT Final Denoised (Haar)', 'WT Final Denoised (Db8)', 'Denoised Haar','WT Noisy (Haar)', 'WT Noisy (Db8)'], no_of_columns=4, figsize=(20,20))
```



Below are the result images from λ=[0.1, 1, 50, 10000]. It can be concluded that a small lambda like 0.1 and 1 will yield a denoised image with much of the noise left. If we increase λ to 50, we get close to the optimal reconstructed image. Once we pass a threshold of about λ=100, the resultant images experience pixelization (seen in the Haar) and distortion (seen in the Db8).

In [19]:

```
lam=0.1

noisy_img = img + 0.1*255* np.random.normal(0, 1, (512,512))

x_haar, errs_haar, k_haar, Hx_haar = ista_img(noisy_img, lam, 'haar')
denoised_img_haar_01 = inverse_transform(x_haar, 'haar')

x_db8, errs_db8, k_db8, Hx_db8 = ista_img(noisy_img, lam, 'db8')
denoised_img_db8_01 = inverse_transform(x_db8, 'db8')

# Display
#grid_display([img, noisy_img,denoised_img_haar,img, noisy_img,denoised_img_db8 ], list_of_titles=['Original', 'Noisy', 'Denoised Haar','Original', 'Noisy', 'Denoised Db8'], no_of_columns=3, figsize=(20,20))
```

In [20]:

```
lam=1

# Add noise
noisy_img = img + 0.1*255* np.random.normal(0, 1, (512,512))

x_haar, errs_haar, k_haar, Hx_haar = ista_img(noisy_img, lam, 'haar')
denoised_img_haar_1 = inverse_transform(x_haar, 'haar')

x_db8, errs_db8, k_db8, Hx_db8 = ista_img(noisy_img, lam, 'db8')
denoised_img_db8_1 = inverse_transform(x_db8, 'db8')

# Display
#grid_display([img, noisy_img,denoised_img_haar,img, noisy_img,denoised_img_db8 ], list_of_titles=['Original', 'Noisy', 'Denoised Haar','Original', 'Noisy', 'Denoised Db8'], no_of_columns=3, figsize=(20,20))
```

In [21]:

```
lam=100

# Add noise
noisy_img = img + 0.1*255* np.random.normal(0, 1, (512,512))

x_haar, errs_haar, k_haar, Hx_haar = ista_img(noisy_img, lam, 'haar')
denoised_img_haar_100 = inverse_transform(x_haar, 'haar')

x_db8, errs_db8, k_db8, Hx_db8 = ista_img(noisy_img, lam, 'db8')
denoised_img_db8_100 = inverse_transform(x_db8, 'db8')

# Display
#grid_display([img, noisy_img,denoised_img_haar,img,
noisy_img,denoised_img_db8 ], list_of_titles=['Original', 'Noisy', 'Denoise
d Haar','Original', 'Noisy', 'Denoised Db8'], no_of_columns=3, figsize=
(20,20))
```

In [22]:

```
lam=100000

# Add noise
noisy_img = img + 0.1*255* np.random.normal(0, 1, (512,512))

x_haar, errs_haar, k_haar, Hx_haar = ista_img(noisy_img, lam, 'haar')
denoised_img_haar_da = inverse_transform(x_haar, 'haar')

x_db8, errs_db8, k_db8, Hx_db8 = ista_img(noisy_img, lam, 'db8')
denoised_img_db8_da = inverse_transform(x_db8, 'db8')

# Display
#grid_display([img, noisy_img,denoised_img_haar,img,
noisy_img,denoised_img_db8 ], list_of_titles=['Original', 'Noisy', 'Denoise
d Haar','Original', 'Noisy', 'Denoised Db8'], no_of_columns=3, figsize=
(20,20))
```

In [25]:

```
grid_display([denoised_img_haar_01,denoised_img_haar_1,denoised_img_haar_50
, denoised_img_haar_100,denoised_img_haar_da, denoised_img_db8_01, denoised
_img_db8_1, denoised_img_db8_50, denoised_img_db8_100, denoised_img_db8_da]
, list_of_titles=['Haar (lam=0.1)', 'Haar (lam=1)','Haar (lam=50)','Haar (l
am=100)', 'Haar (lam=100000)', 'Db8 (lam=0.1)', 'Db8 (lam=1)', 'Db8 (lam=50
)','Db8 (lam=100)', 'Db8 (lam=100000)'], no_of_columns=5, figsize=(20,20))
```

There are pros and cons of using different wavelet filters. The Haar can be especially useful if we just to compress an image without minding the details. Without making the details a priority, we can generate a sparse image, thus we can save space. However, we can see that the image will be very obviously pixelized (see above examples). Note that it is also easier to implement.

The Daubechie 8/8 filter will, too, generate sparse representation of a given image. It is a generalization of the Haar filter, but implementation is harder. A pro is such a filter will return a more fine detailed image than the Haar filter given the same lambda level. But, we can see that the denoised image will suffer from distortion when the lambda is too high (see above examples).