HW3_12

June 7, 2022

1 EE798 HW3

1.1 Kutay Ugurlu

1.1.1 To achieve the notebook and the MATLAB scripts, click here.

[ ]:

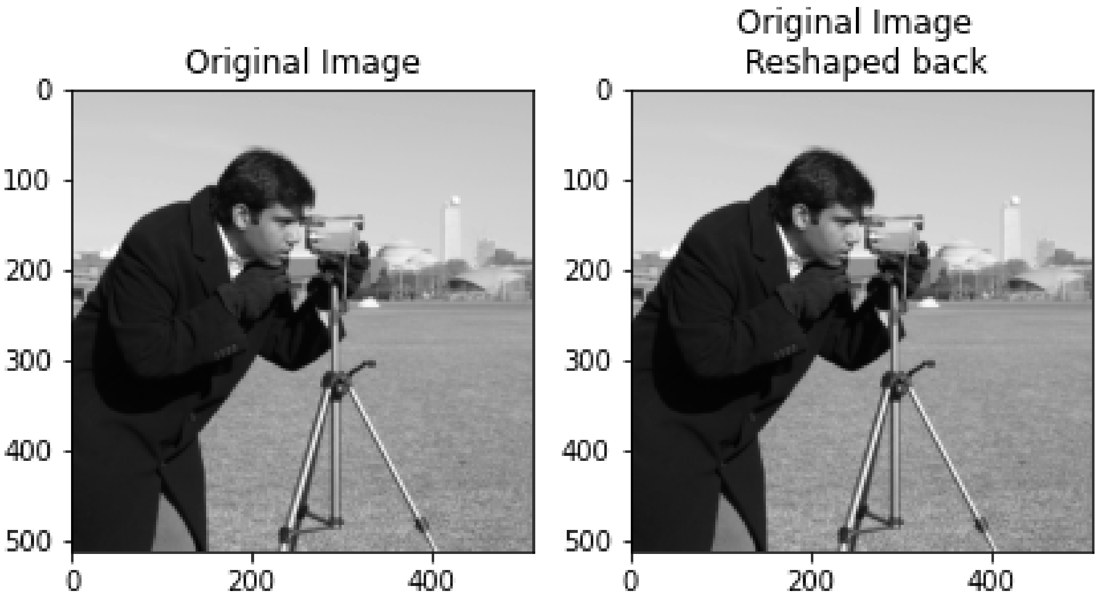
```
from scipy.linalg import toeplitz, dft, inv, eigvals, eig, kron, eigh
from scipy.signal import convolve2d, convolve
from sklearn.preprocessing import normalize
import numpy as np
from numpy.fft import fft, ifft, fft2, ifft2
import matplotlib.pyplot as plt
from skimage.data import camera
```

2 Q1 Matrix Representation of 2D Convolution

2.1 Q1 a

[ ]:

```
I = camera()
I_flatten = I.flatten(order="F")
I_back = np.reshape(I,(I.shape))
plt.subplot(1,2,1)
plt.imshow((I),cmap="gray")
plt.title("Original Image")
plt.subplot(1,2,2)
plt.imshow((I_back),cmap="gray")
plt.title("Original Image \n Reshaped back")
plt.tight_layout()
```



2.1.1 The vectors stacked in matrix H are flattened version of the convolved vectors of basis pixels. Since flattened basis vectors are lexicographically ordered, we observed shifted versions of block of the output vectors in the matrix, hence in a Toeplitz form.

2.1.2 This matrix may be too huge to store for a huge image size.

3 Q1 b

3.0.1 Given that H is of size (P_1, P_2) and image of size (N_1, N_2), FFT points of

$$(N_1 + P_1 - 1, N_2 + P_2 - 1) = (L_1, L_2)$$

should be used to have linear convolution.

3.0.2 If FFT points (N_1, N_2) are higher than the image sizes, the images are padded zeros until they have FFT points size, as in the case of linear convolution calculation. On the other hand, if we were to use fewer points FFT, then the images are rolled onto themselves in the following manner:

$$\hat{x}(m, n) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} x(m + iN_1, n + jN_2)$$

[ ]:

```
def cconv2(x:np.ndarray, y:np.ndarray,N1:int,N2:int):
    sizes = (N1,N2)
    return ifft2(np.multiply(fft2(x,sizes),fft2(y,sizes)))
```

3.0.3 Tests

[ ]:

```python
for _ in range(200):
    N1 = np.random.randint(1,5)
    N2 = np.random.randint(1,10)
    N3 = np.random.randint(1,20)
    x = np.random.randint(0,100,(N1,N1))
    y = np.random.randint(0,100,(N2,N3))
    cconv_size_1 = N1+N2-1
    cconv_size_2 = N1+N3-1
    res = convolve2d(x,y,mode="full")
    res2 = np.round(cconv2(y,x,cconv_size_1,cconv_size_2))
    assert np.sum(np.isclose(res,res2)) == res.size
print("All tests are success!")
```

```
All tests are success!
```

4 Q1 c

[ ]:

```python
def cconvmtx2(h:np.ndarray, image_shape, N1, N2):
    L1,L2 = image_shape
    H = np.empty((N1*N2,0))
    image_size = L1*L2
    for i in range(image_size):
        row = np.mod(i,L1)
        col = i//L1
        basis_vec = np.zeros((L1,L2))
        basis_vec[row,col] = 1
        basis_vec_output = cconv2(h,basis_vec,N1,N2)
        transformed_basis_vec = basis_vec_output.flatten(order="F")
        H = np.column_stack((H, transformed_basis_vec))
    return H
```

4.1 Verification

[ ]:

```python
for trial in range(100):

    P1 =  np.random.randint(1,3)
    P2 =  np.random.randint(1,3)
    L1 = np.random.randint(P1,10)
    L2 = np.random.randint(P2,10)
    N1 = np.random.randint(L1,20)
    N2 = np.random.randint(L2,20)


    h = np.random.randint(1,5,(P1,P2))
    test_image = np.random.randint(1,100,(L1,L2))
    res = cconv2(h,test_image, N1, N2)
    H = cconvmtx2(h, test_image.shape, N1, N2)
    res2 = np.reshape(H.dot(test_image.flatten(order="F")),(N1,N2),order="F")
    assert np.sum(np.isclose(res,res2)) == res.size

print("All tests are success!")
```

```
All tests are success!
```

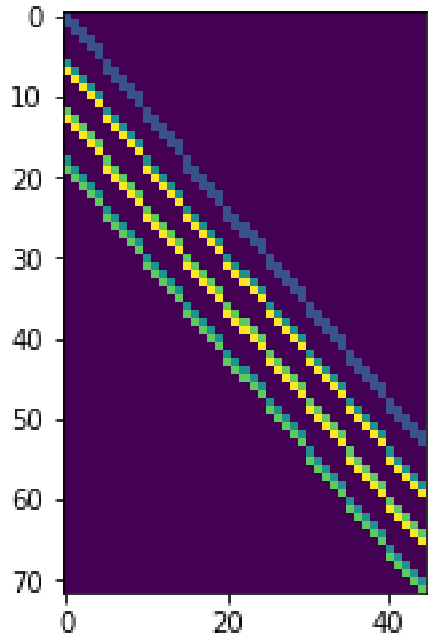
## A linear convolution example ↓

[ ]:

```
P1 =  np.random.randint(1,6)
P2 =  np.random.randint(1,6)
L1 = np.random.randint(P1,10)
L2 = np.random.randint(P2,10)
N1 = L1 + P1 - 1
N2 = L2 + P2 - 1


h = np.random.randint(1,5,(P1,P2))
test_image = np.random.randint(1,100,(L1,L2))
res = cconv2(h,test_image, N1, N2)
H = cconvmtx2(h, test_image.shape, N1, N2)
plt.imshow(np.real(H))
```

[ ]:

```
<matplotlib.image.AxesImage at 0x1ba32a202b0>
```



5 Q1 d

[ ]:

```
def dftmtx2(N1:int, N2:int):
    F = np.empty((N1*N2,0))
    for i in range(N1*N2):
        row = np.mod(i,N1)
        col = i//N1
        basis_vec = np.zeros((N1,N2))
        basis_vec[row,col] = 1
        basis_vec_output = fft2(basis_vec,(N1,N2))
        transformed_basis_vec = basis_vec_output.flatten(order="F")
        F = np.column_stack((F, transformed_basis_vec))
    return F
```

5.1 Verification

5.1.1 $(2DFFT(I))_{flat} == FI_{flat}$

[ ]:

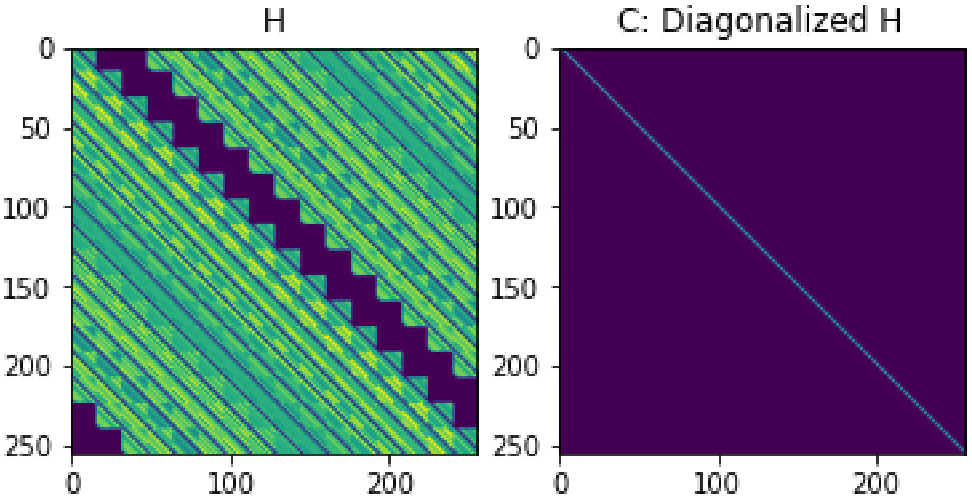
```
for trial in range(50):

    N1 = np.random.randint(1,20)
    N2 = np.random.randint(1,20)


    test_image = np.random.randint(1,100,(N1,N2))
    res = fft2(test_image, (N1, N2))
    F = dftmtx2(N1, N2)
    res2 = np.reshape(F.dot(test_image.flatten(order="F")),(N1,N2),order="F")
    F2 =  dftmtx2(N1, N1)
    assert np.sum(np.isclose(res,res2)) == res.size and np.sum(np.
 ↪isclose(inv(F2),F2.conjugate().T/N1**2)) == F2.size

print("All tests are success!")
```

```
All tests are success!
```

5.1.2 Since FFT matrix entries are symmetrical with respect to variables $n1$, $k1$ and $n2$, $k2$ the matrix is symmetric, *i.e.* $ F = F^T$. Inverse FFT matrix F^{-1} have just conjugated entries scaled with $\frac{1}{N_1N_2}$ of F. Hence $F^{-1} = \frac{F^H}{N_1N_2}$

[ ]:

```
for _ in range(100):

    N = np.random.randint(2,20,(1,1)).item()
    P = np.random.randint(1,N,(1,1)).item()
    h = np.random.randint(40,100,(P,P))
    Image = np.random.randint(1,30,(N,N))
    H = cconvmtx2(h,Image.shape,N,N)
    h = np.pad(h,((0,N-P),(0,N-P)))
    F = dftmtx2(N,N)
    C = F.dot(H).dot(F.conjugate().T)/N**2
    diagC = np.diag(C)
    diagC_check = fft2(h,(N,N)).flatten(order="F")
    assert np.sum(np.isclose(diagC,diagC_check)) == diagC_check.size # Check␣
 ↪whether diagonalized matrix stores the flattened FFT2 coefficients
    x = Image.flatten(order="F")
    y = H.dot(x)
    Y = F.dot(y)
    H = F.dot(h.flatten(order="F"))
    X = F.dot(x)
    assert np.sum(np.isclose(Y,H*X)) == Y.size # Check whether the given␣
 ↪relation is true

H = cconvmtx2(h,Image.shape,N,N)
F = dftmtx2(N,N)
C = F.dot(H).dot(F.conjugate().T)/N**2
plt.subplot(1,2,1)
plt.imshow(np.real(H))
plt.title("H")
plt.subplot(1,2,2)
plt.imshow(np.abs(normalize(np.abs(C),norm="max")))
plt.title("C: Diagonalized H")
plt.show()
```



5.2 $\tilde{\mathbf{C}}$ is a diagonal matrix storing the flattened version of 2D FFT coefficients of the filter.

5.3

$$diag(\tilde{\mathbf{C}}) = 2DFFT(H)_{flat}$$

5.4 In addition, $\tilde{\mathbf{C}} = \frac{FHF^H}{N_1 N_2}$ corresponds to the eigen decomposition of matrix H, hence diagonal matrix $\tilde{\mathbf{C}}$ corresponds to the matrix storing the eigen values. Therefore, eigenvalues of H are the 2-dimensional FFT coefficients of the filter H.

5.5 Let b_i's be the basis vectors of $(N1, N2)$ grid and (:) operator be the lexicographic ordering operator, F_N is the N dimensional FFT operator.

2D FFT matrix F_2 can be considered as follows:

$$F_2 = [F_2 b_1(:)\ F_2 b_2(:)\ \ldots\ F_2 b_N(:)]$$

Circular convolution matrix H has columns as follows:

$$[{F_2}^{-1} diag(F_2 h(:)) F_2 b_1(:)\ {F_2}^{-1} diag(F_2 h(:)) F_2 b_2(:)\ \ldots {F_2}^{-1} diag(F_2 h(:)) F_2 b_N(:)]$$

Then $\tilde{\mathbf{C}} = \frac{F_2 H {F_2}^H}{N_1 N_2}$ can be rewritten as:

$$[\underbrace{F_2 {F_2}^{-1}}_{I} diag(\underbrace{F_2 h(:)}_{H}) F_2 b_1(:) {F_2}^{-1}\ \ \underbrace{F_2 {F_2}^{-1}}_{I} diag(\underbrace{F_2 h(:)}_{H}) F_2 b_2(:) {F_2}^{-1} \ldots]$$

The terms can be collected as follows:

$$diag(H)\ \underbrace{\underbrace{[F_2 b_1(:) F_2 b_2(:) F_2 b_3(:) \ldots]}_{F_2}\, {F_2}^{-1}}_{I}$$

Hence

$$\tilde{\mathbf{C}} = diag(H)$$

IF the problem was not shift invariant, we could not utilize convolution matrix to define the relationship between the input and output of the system. Hence, utilization of DFT via circular convolution to conduct linear convolution would become impossible.

6 Q2 a

$$\delta(t - s_1 \cos\theta - s_2 \sin\theta) = \begin{cases} 1 & t = s_1 \cos\theta + s_2 \sin\theta \\ 0 & otherwise \end{cases}$$

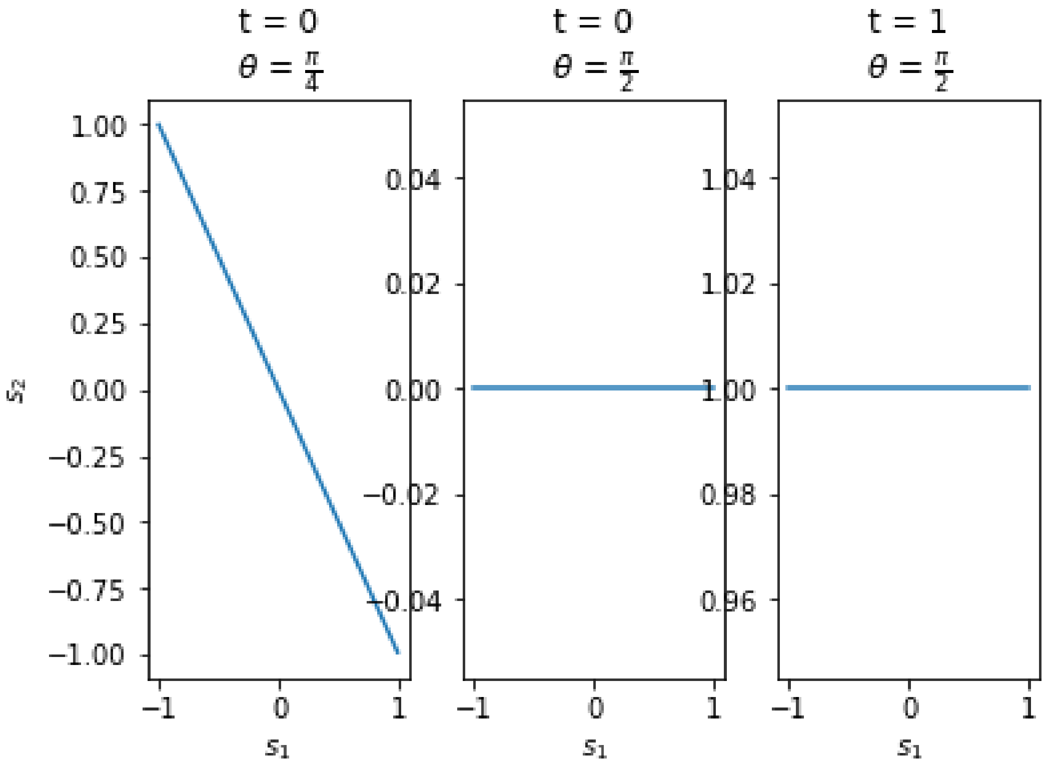
6.1 In s_1,s_2 space equation $t = s_1 \cos\theta + s_2 \sin\theta$ corresponds to the line in the form

$$s_2 = m s_1 + b$$

where $m = -\cot\theta$ and $b = \frac{t}{\sin\theta}$

[ ]:

```
s1 = np.arange(-1,1,0.01)
i = 1
thetas = [np.pi/4, np.pi/2]
theta_names = [r'$\frac{\pi}{4}$',r'$\frac{\pi}{2}$']
others = [0,1,1]
for t in [0,0,1]:
    theta = thetas[others[i-1]]
    s2 = -1/np.tan(theta) * s1 + t/np.sin(theta)
    plt.subplot(1,3,i)
    plt.plot(s1,np.round(s2,4))
    plt.title("t = " + str(t) + "\n" + r'$\theta$ = ' +␣
 ↪theta_names[others[i-1]])
    plt.xlabel(r'$s_1$')
    if i == 1:
        plt.ylabel(r'$s_2$')
    i += 1
```



6.2 As can be seen in the above figure, θ changes the angle which the line makes with the axes and t changes the shift in the s_2 direction.

6.3 $y(\theta, t)$ is the integration of the object along a line specified by the projection variables. The reason behind the line integral is that the impulse function given here zeroes out the whole function except for the line specified by the arguments of itself.

7 Q2 b

7.1 Projection operation is defined as follows:

$$y(\theta, t) = \iint\limits_{-\infty}^{\infty} \delta(t - s_1 \cos\theta - s_2 \sin\theta) x(s_1, s_2) ds_1 ds_2$$

7.2 Using the given note that The adjoint of a general integral equation over an infinite domain is given by the same integral equation integrated with respect to the output variables. adjoint of the projection operation can be defined as follows:

$$\begin{aligned} x(s_1, s_2) &= \int\limits_{\theta=0}^{\pi} \int\limits_{t=-\infty}^{\infty} \delta(t - s_1 \cos\theta - s_2 \sin\theta) y(\theta, t) d\theta dt \\ &= \int\limits_{\theta=0}^{\pi} y(\theta, s_1 \cos\theta + s_2 \sin\theta) d\theta dt \end{aligned}$$

## Physically this equation describes a procedure where projection matrix is integrated over a line described by s_1, s_2 and θ and the value of $x(s_1, s_2)$ is computed. This projection is known as back-projection.

[ ]:

```python
from skimage.transform import radon, resize
from skimage.data import camera, shepp_logan_phantom
```

8 Q2 c

8.1

$$y(\theta, t) = \iint\limits_{-\infty}^{\infty} \delta(t - s_1 \cos\theta - s_2 \sin\theta) x(s_1, s_2) ds_1 ds_2$$

8.1.1 The matrix T that conducts discrete random transform operation is composed of elements t_{ij}'s representing the line integral of i^{th} ray along the j^{th} basis vector, ***i.e.***, j^{th} pixel. In the given discretization scheme, the integral boundaries regarding $s1$ and s_2 directions can be found as follows:

$$\begin{aligned} s_1 &\in [\tfrac{N}{2} - i \bmod N, \tfrac{N}{2} - i \bmod N - 1] \\ s_2 &\in [\tfrac{-N}{2} + \lfloor j/N \rfloor, \tfrac{-N}{2} + \lfloor j/N \rfloor + 1] \end{aligned}$$

8.2 In more analytical notation,

$$b_j(s_1,s_2) = \Pi(s_1 - (\frac{N}{2} - 0.5 - i \bmod N), s_2 - (\frac{-N}{2} + 0.5 + \lfloor j/N \rfloor))$$

where Π represent the rectangular function.

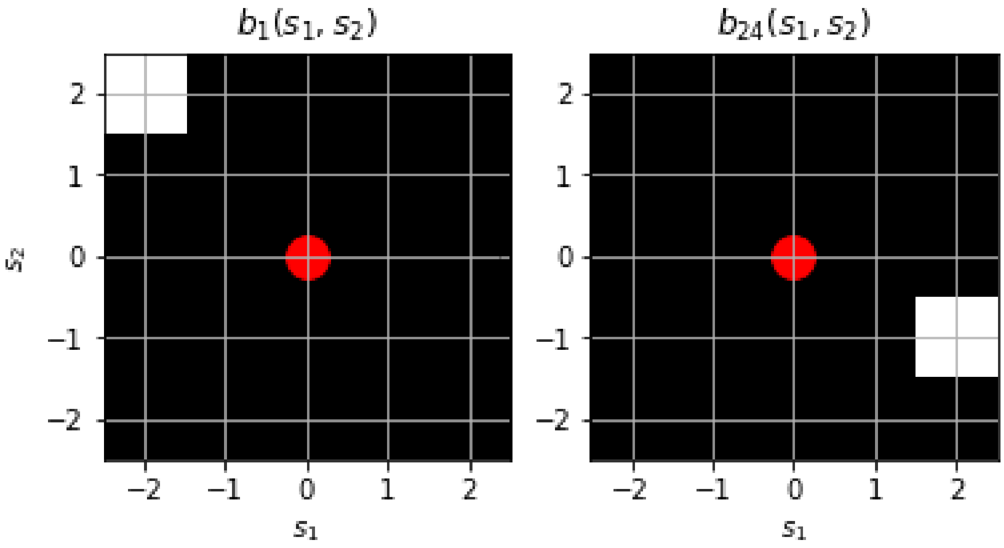
8.3 Hence, entries of T can be written as

$$T_{ij} = y(\theta,t) = \int\limits_{s_1=\frac{N}{2}-i \bmod N}^{\frac{N}{2}-i \bmod N-1} \int\limits_{s_2=\frac{-N}{2}+\lfloor j/N \rfloor}^{\frac{-N}{2}+\lfloor j/N \rfloor+1} \delta(t - s_1\cos\theta - s_2\sin\theta)\underbrace{x(s_1,s_2)}_{=1} ds_1 ds_2$$

8.4 Soem example basis functions can be found below.

```
[ ]: A = np.zeros((5,5))
     A[0,0] = 1
     plt.subplot(1,2,1)
     plt.imshow(A,extent=[-A.shape[1]/2., A.shape[1]/2., -A.shape[0]/2., A.shape[0]/
      ↪2. ],cmap="gray")
     plt.grid()
     plt.scatter(0,0,c="red",s=250)
     plt.title(r'$b_1(s_1,s_2)$')
     plt.xlabel(r'$s_1$')
     plt.ylabel(r'$s_2$')
     plt.subplot(1,2,2)
     A[0,0] = 0
     A[3,4] = 1
     plt.title(r'$b_{24}(s_1,s_2)$')
     plt.xlabel(r'$s_1$')
     plt.xlabel(r'$s_2$')
     plt.imshow(A,extent=[-A.shape[1]/2., A.shape[1]/2., -A.shape[0]/2., A.shape[0]/
      ↪2. ],cmap="gray")
     plt.grid()
     plt.scatter(0,0,c="red",s=250)
     plt.xlabel(r'$s_1$')
     plt.ylabel(r'$s_2$')
```

```
[ ]: Text(0, 0.5, '$s_2$')
```



9 Q2 d

[ ]:

```
def projmtx(N, thetas):

    T = np.empty((int(np.ceil(N*np.sqrt(2))*len(thetas)),0))
    for i in range(N**2):
        row = np.mod(i,N)
        col = i//N
        basis_vec = np.zeros((N,N))
        basis_vec[row,col] = 1
        basis_vec_output = radon(basis_vec,thetas,circle=False)
        transformed_basis_vec = basis_vec_output.flatten(order="F")
        T = np.column_stack((T,transformed_basis_vec.flatten(order="F")))
    return T
```

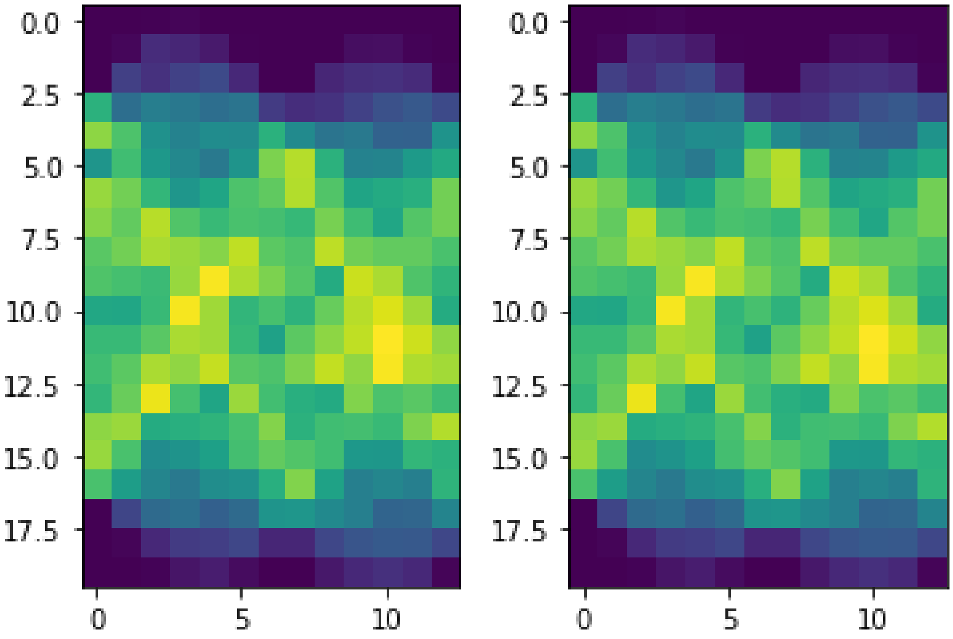
[ ]:

```
for _ in range(100):
    N = np.random.randint(4,20,(1,1)).item()
    a = np.random.randint(1,100,(N,N))
    thetas = np.arange(0,180,N)
    T = projmtx(N,thetas)
    res = T.dot(a.flatten(order="F"))
    res = np.reshape(res,(res.shape[0],1))
    res2 = radon(a,thetas,circle=False, preserve_range=True)
    res = np.reshape(res,(res2.shape),order="F")
    assert np.sum(np.isclose(res,res2)) == res.size

print("[INFO] All tests are success!")
```

```
plt.subplot(1,2,1)
plt.imshow(res)
plt.subplot(1,2,2)
plt.imshow(res2)
```

```
[INFO] All tests are success!
```

```
[ ]: <matplotlib.image.AxesImage at 0x1ba334e3e20>
```

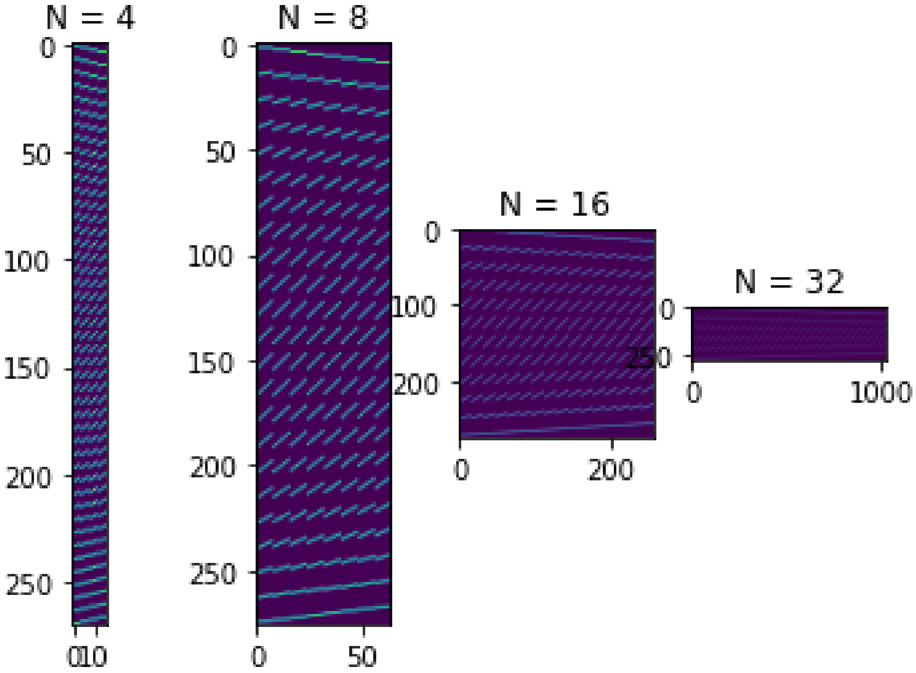


10 Q2 e

```
[ ]: i = 1
     for N in [1,4,8,16,32]:
         if not N == 1:
             X = resize(shepp_logan_phantom() ,(N,N))
         else:
             X = np.random.randint(1,100,(N,N))
         thetas = np.arange(0,180,N)
         T = projmtx(N,thetas)
         plt.subplot(1,4,i)
         plt.imshow(T)
         plt.title("N = " + str(N))
         if not N == 1:
             i += 1
         res = T.dot(X.flatten(order="F"))
```

```
    res = np.reshape(res,(res.shape[0],1))
    res2 = radon(X,thetas,circle=False, preserve_range=True)
    res = np.reshape(res,(res2.shape),order="F")
    assert np.sum(np.isclose(res,res2)) == res.size
print("[INFO] All tests are success!")
```

```
[INFO] All tests are success!
```



10.1 Shepp-Logan and other random phantoms have the same projection via projection matrix and built in radon.

10.2 The resultant matrices are not Toeplitz matrices, hence the transform is not shift invariant.

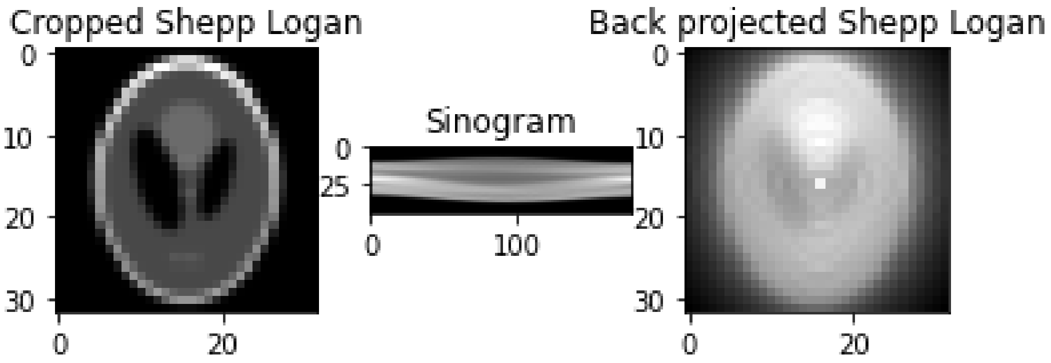
11 Q2 f

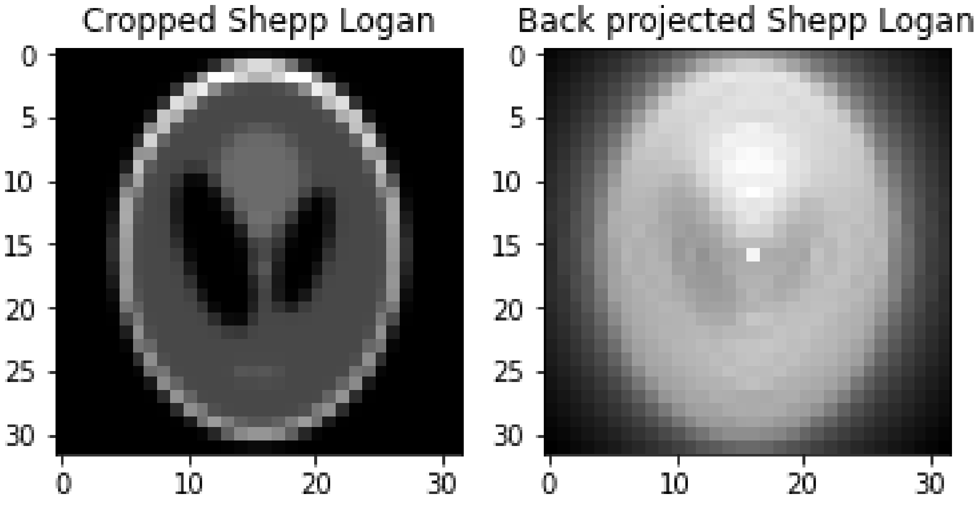
11.1 Let A be the matrix representation of a linear system operator. Then the adjoint operator A^* is defined as A^T.

[ ]:
```
X = resize(shepp_logan_phantom() ,(N,N))
thetas = np.arange(0,180,1)
T = projmtx(N,thetas)
projections = radon(X,theta=thetas,circle=False,preserve_range=True)
back_projected_X = np.reshape(T.T.dot(projections.flatten(order="F")),X.
  ↪shape,order="F")
```

```
plt.figure()
plt.subplot(1,3,1)
plt.imshow(X,cmap="gray")
plt.title("Cropped Shepp Logan")
plt.subplot(1,3,2)
plt.imshow(projections,cmap="gray")
plt.title("Sinogram")
plt.subplot(1,3,3)
plt.imshow(back_projected_X,cmap="gray")
plt.title("Back projected Shepp Logan")

plt.figure()
plt.subplot(1,2,1)
plt.imshow(X,cmap="gray")
plt.title("Cropped Shepp Logan")
plt.subplot(1,2,2)
plt.imshow(back_projected_X,cmap="gray")
plt.title("Back projected Shepp Logan")
```

```
[ ]: Text(0.5, 1.0, 'Back projected Shepp Logan')
```





11.2 The remaining questions are implemented in MATLAB.

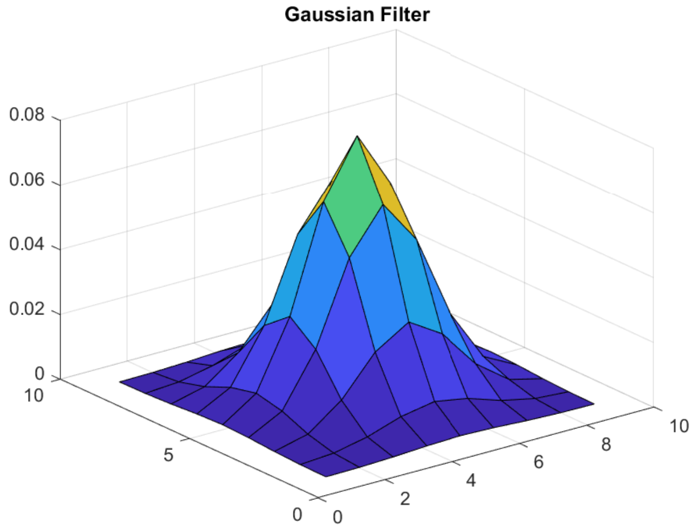
Blurring Filter

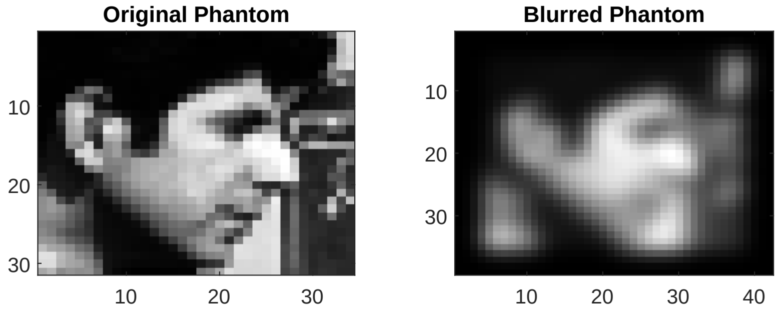
Setup

```
close all;
clc;
```

Part a

```
I = imread('cameraman.tif');
X = im2double(I);
X = X(50:80,105:138);
F = gauss2d(9,9,0,0,2,2);
figure
surf(F)
title('Gaussian Filter')
C = convmtx2(F,size(X,1),size(X,2));
X_vec = X(:);
y = C*X_vec;
f1 = figure;
ax = gca;
subplot(2,2,1,ax)
imagesc(X), colormap gray
title('Original Phantom')
subplot(2,2,2)
Y_blurred = reshape(y,size(F)+size(X)-1);
imagesc(Y_blurred)
title('Blurred Phantom')
```



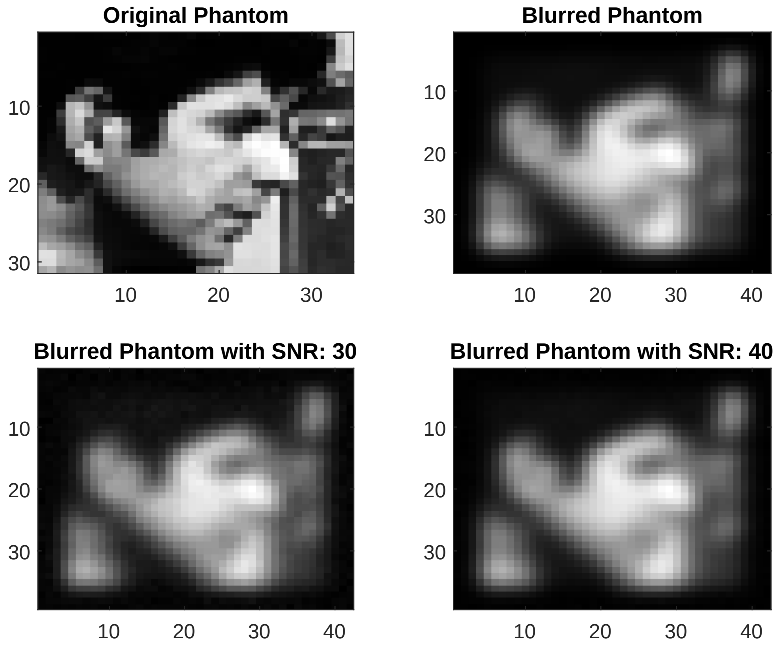


Original Phantom

Blurred Phantom

Part b

```
for ratio = [3 4]
    variance = var(y)/10^ratio; % SNR adjustment
    sigma_n = sqrt(variance);
    noise = sigma_n * randn(size(y)); % N~(0,sigma_n^2)
    y_n = y + noise;
    y_img = reshape(y_n,size(F)+size(X)-1);
    subplot(2,2,ratio)
    imagesc(y_img)
    title(['Blurred Phantom with SNR: ',num2str(10*ratio)])
end
```



Original Phantom

Blurred Phantom

Blurred Phantom with SNR: 30

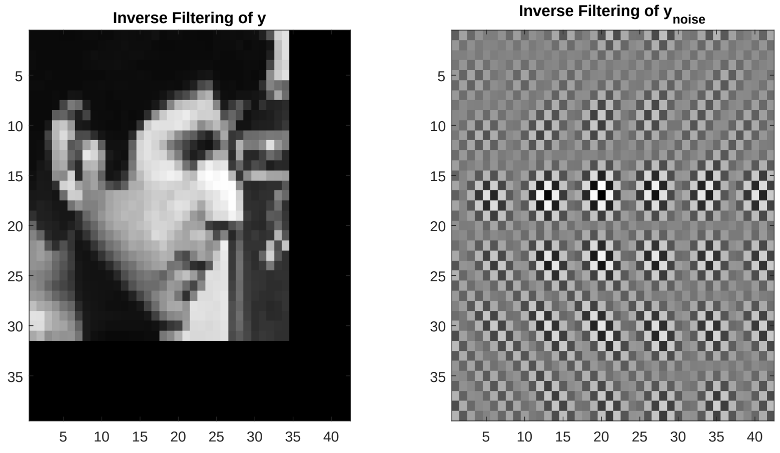
Blurred Phantom with SNR: 40

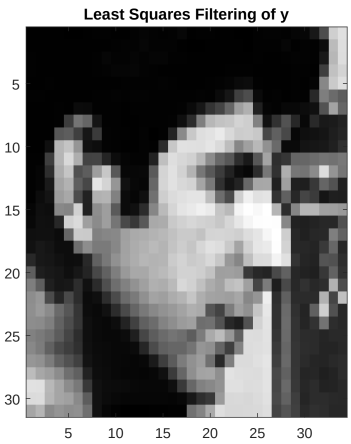
Comments

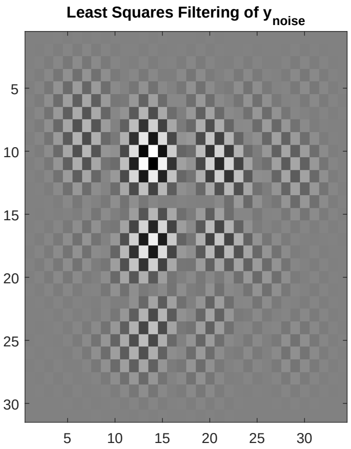
The effect of the noise is not observable to human eye under the effect of blurring, *i.e.* it is difficult to distinguish the noise from the blurred image.

Part c: Inverse and LS Filtering in Frequency domain

```
figure
size_Y = size(Y_blurred);
X_inv = ifft2(fft2(Y_blurred,size_Y(1), ...
    size_Y(2))./fft2(F,size_Y(1),size_Y(2)),size_Y(1),size_Y(2));
X_inv_noisy = ifft2(fft2(y_img)./fft2(F, ...
    size_Y(1),size_Y(2)),size_Y(1),size_Y(2));
subplot(1,2,1)
imagesc(X_inv), colormap gray
title('Inverse Filtering of y')
subplot(1,2,2)
imagesc(X_inv_noisy)
title('Inverse Filtering of y_{noise}')

figure
C = full(C); % The LS solution outputs wrong results if C remains sparse!
X_inv = reshape(inv(C'*C)*C'*y,size(X));
X_inv_noisy = reshape(inv(C'*C)*C'*y_n,size(X));
subplot(1,2,1)
imagesc(X_inv), colormap gray
title('Least Squares Filtering of y')
subplot(1,2,2)
imagesc(X_inv_noisy)
title('Least Squares Filtering of y_{noise}')
```







Comments

$$\begin{aligned} \tilde{y} &= y + n && (1) \\ A &= U\Sigma V^H && (2) \\ A^T &= V\Sigma U^H && (3) \\ x_{\hat{LS}} &= (A^T A)^{-1} A^T && (4) \end{aligned}$$

Inserting Eqn. 2 and 3 in 4, results in following expression:

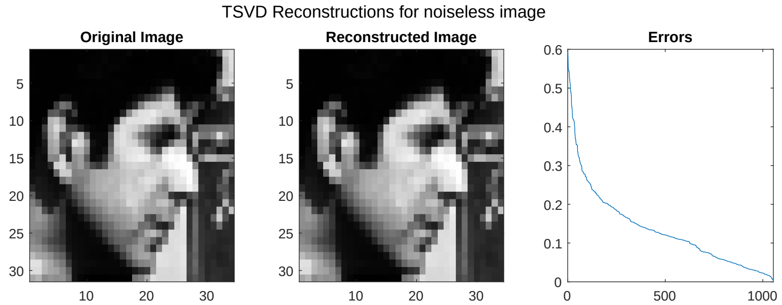
$$x_{\hat{LS}} = \underbrace{V\Sigma^{-1}U^H}_{A^\dagger}(y + n) \tag{5}$$

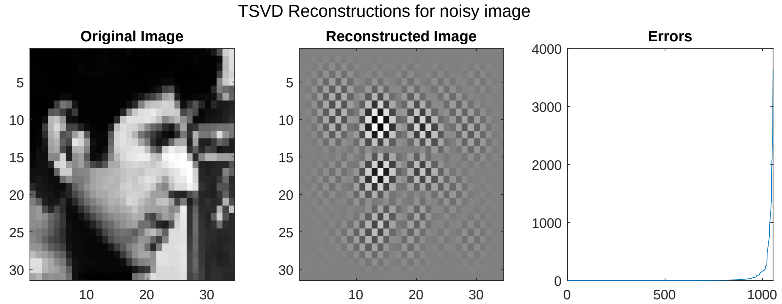
The least squares solution amplifies the noise with pseudo-inverse of A (in SVD sense) hence results in the checkerboard pattern in the reconstruction as an implication of amplified high frequency noise. Since A is an ill-conditioned matrix and hence has really small singular values, noise is amplified with $\frac{1}{\sigma}$ which is a huge amplification factor.

Part d

y = U S VH x; x = V 1/S UH y;

```
[U,S,V] = svd(C);
singular_values = diag(S);
X_back_vec = zeros(size(X_vec));
Errors = zeros(size(singular_values));
for i = 1:length(singular_values)
    s = singular_values(i);
    v = V(:,i);
    u = U(:,i);
    X_back_vec = X_back_vec + 1/s * v * u' * y;
    Errors(i) = relative_error(X_vec, X_back_vec);
end
X_back = reshape(X_back_vec,size(X));
f = figure;
f.Position = [100 100 900 300];
colormap gray
subplot(1,3,1)
imagesc(X)
title('Original Image')
subplot(1,3,2)
imagesc(X_back)
title('Reconstructed Image')
subplot(1,3,3)
plot(Errors)
title('Errors')
sgtitle('TSVD Reconstructions for noiseless image')

[U,S,V] = svd(C);
singular_values = diag(S);
X_back_vec = zeros(size(X_vec));
Errors = zeros(size(singular_values));
for i = 1:length(singular_values)
    s = singular_values(i);
    v = V(:,i);
    u = U(:,i);
    X_back_vec = X_back_vec + 1/s * v * u' * y_n;
    Errors(i) = relative_error(X_vec, X_back_vec);
end
Full_Errors = Errors;
X_back = reshape(X_back_vec,size(X));
f = figure;
f.Position = [100 100 900 300];
colormap gray
subplot(1,3,1)
imagesc(X)
title('Original Image')
subplot(1,3,2)
imagesc(X_back)
title('Reconstructed Image')
subplot(1,3,3)
plot(Errors)
title('Errors')
sgtitle('TSVD Reconstructions for noisy image')
```





Comments

For this question a distance error metric is defined:

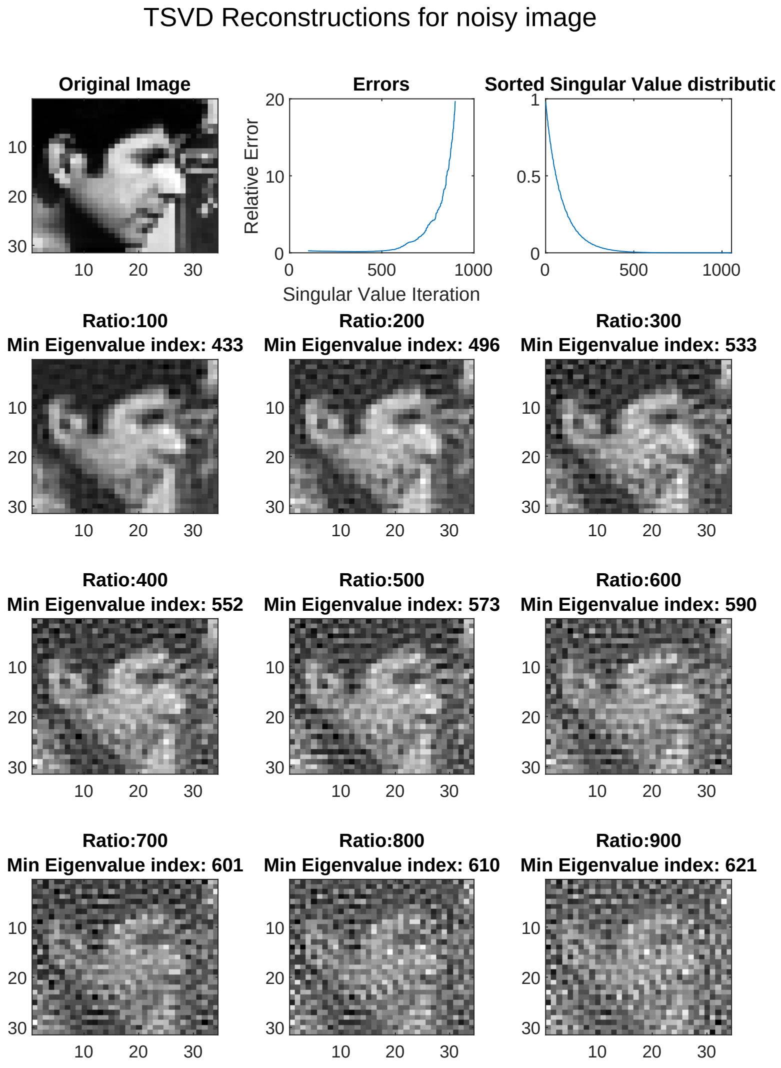
$$RelativeError = \frac{||\hat{x} - x||}{||x||} \tag{6}$$

In the figure above, we observe that error is growing after the 900th singular value. Hence, to find a suitable threshold, this interval is scanned.

Find threshold for noisy reconstruction

A ratio of $\frac{1}{200}$ in singular value amplitude provided a satisfactory amount of reconstruction.

```
f = figure; colormap gray;
f.Position = [100 100 600 800];
plot_idx = 4;
for ratio = 100:100:900
    max_sing_value = max(singular_values);
    threshold = max_sing_value / ratio;
    [minimum, index] = min(abs(singular_values-threshold));

    X_back_vec = zeros(size(X_vec));
    Errors = zeros(index,1);
    for i = 1:index
        s = singular_values(i);
        v = V(:,i);
        u = U(:,i);
        X_back_vec = X_back_vec + 1/s * v * u' * y_n;
        Errors(i) = relative_error(X_vec, X_back_vec);
    end
    X_back = reshape(X_back_vec,size(X));
    subplot(4,3,plot_idx)
    imagesc(X_back)
    title({['Ratio:',num2str(ratio)] ...
        ,['Min Eigenvalue index: ',num2str(index)]})
    plot_idx = plot_idx + 1;
end

colormap gray
subplot(4,3,1)
imagesc(X)
title('Original Image')

subplot(4,3,2)
plot(100:900,Full_Errors(100:900))
title('Errors')
xlabel('Singular Value Iteration')
ylabel('Relative Error')

subplot(4,3,3)
plot(singular_values)
title('Sorted Singular Value distribution')
sgtitle('TSVD Reconstructions for noisy image')
```



Comments

Experiments showed that applying a threshold around the 1% of the maximum singular value resulted in moderate success when compared to the other reconstruction thresholds. The corresponding singular value index is found to be 433. This can also be seen in the error plot given in the top of the above figure, where error gets drastically higher after the inclusion of 500 singular values.

Conjugate Gradient

a

Assuming the vector-vector product has computational cost of N operations, the total cost of CG for N iterations and direct inversion is calculated as follows: For one iteration of Conjugate Gradient:

- $\alpha_k : 2N^2 + N \approx 2N^2$
- $x_{k+1} : \ \approx 1$
- $r_{k+1} : N^2$
- $\beta_k : 2N$
- $p_{k+1} : \ \approx 1$

Therefore, total cost per iteration can be approximated as $3N^2 + 2N$. For N_{iter} iterations, the overall cost is $(3N^2 + 2N)N_{iter}$. On the other hand, for direct inversion, requiring the computation of $(C^T C + \lambda D^T D)^{-1}$. The computation of the forward matrix costs $2N^2$ operations. Inversion of this matrix costs approximately N^3 operations, resulting in an overall cost of $N^3 + 2N^2$. Assuming $N >> 3$, then CG requires $3N^2 N_{iter}$ and direct inversion requires N^3 operations. Hence, for small N_{iter}, CG requires less computation.

b

Comments

As stated, the main intensive task in the Conjugate gradient method is the repetitive forward operators. Tikhonov regularized solution can be found by assigning $A = (C^T C + \lambda D^T D)$. However, this may not be useful in memory-limited systems. Hence forward projections of A should be replaced with another operation requiring less memory. This can be done as follows:

$$\begin{aligned} Az &= (C^T C + \lambda D^T D)z && (7) \\ &= C^T(Cz) + \lambda D^T(Dz) && (8) \end{aligned}$$

This type of formulation allows us to replace Az with forward operators Cz, $C^T z$, Dz and $D^T z$ for some arbitrary vector z. For some particular C and D, the quantities Cz and $C^T z$ in Eqn. 8 can be computed using fast-forward routines, without forming the huge matrices that operate on flattened vector representations of the images. Keeping in mind that the operator C represents convolution operation, hence it can be computed via FFTs. C^T is the adjoint operator of convolution and corresponds to the convolution of input with flipped or inverted filter $H(-x, -y)$. In the case of matrix C representing Radon transform, the quantities Cz and $C^T z$ can be computed by utilizing the fast-forward routines `radon.m` and `iradon.m` in MATLAB. This result is already derived in Homework 2. Similarly, discrete derivative operation can be computed by convolving the image with the first order derivative kernels, without the need of calculation the matrices R_1 and R_2. Hence, one only needs to store the discrete derivative operator and using the separability feature of the 2D discrete derivative operator, reduces memory requirements drastically. The derivative operator D is operated on the input vector x using the given stacked decomposition as follows:

$$Dx = \begin{bmatrix} R_1 x & R_2 x \end{bmatrix} \tag{9}$$

$$D^T z = \begin{bmatrix} {R_1}^T {R_2}^T \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = {R_1}^T z_1 + {R_2}^T z_2 \tag{10}$$

$$D^T D x = {R_1}^T R_1 x + {R_2}^T R_2 x \tag{11}$$

where ${R_N}^z$ corresponds to convolution with first order derivative kernel in dimension N and ${R_N}^T z$ is convolution with flipped derivative kernel. Using these fast-forward operations, the following scripts are written to perform Tikhonov regularization via Conjugate Gradient.

Forward Convolution A

```
function res = forward_A(H,lambda,D1,D2,p)
H_flipped = flip(flip(H,1),2);
D1_flipped = flip(D1);
D2_flipped = flip(D2);
Cp = conv2(p,H,'same');
CtranposeCp = conv2(Cp,H_flipped,'same');
R1 = conv2(conv2(p,D1,'same'),D1_flipped,'same');
R2 = conv2(conv2(p,D2,'same'),D2_flipped,'same');
res = CtranposeCp + lambda*(R1+R2);
res = res(:);
end
```

Forward Radon A

```
function res = forward_radon(lambda,D1,D2,p)
D1_flipped = flip(D1);
D2_flipped = flip(D2);
theta_step = 180/size(p,1);
thetas = 0:theta_step:180-theta_step;
Cp = radon(p,thetas);
CtranposeCp = iradon(Cp,thetas);
CtranposeCp = CtranposeCp(2:end-1,2:end-1);
R1 = conv2(conv2(p,D1,'same'),D1_flipped,'same');
R2 = conv2(conv2(p,D2,'same'),D2_flipped,'same');
```

```
res = CtranposeCp + lambda*(R1+R2);
res = res(:);
end
```

CG Convolution Tikhonov

```
function estimate = cgconvtik(H, D1, D2, Y, x_0, lambda, n_iter)
size_Y = size(Y);
Y = Y(:);
Ax_0 = forward_A(H,lambda,D1,D2,x_0);
x_0 = x_0(:);
r = Y - Ax_0;
p = r;
for k = 1:n_iter
    Ap = forward_A(H,lambda,D1,D2,reshape(p,size_Y));
    alpha = (r'*r)/(p'*Ap);
    x_0 = x_0 + alpha*p;
    r_new = r - alpha*Ap;
    beta = (r_new'*r_new)/(r'*r);
    p = r_new + beta*p;
    r = r_new;
end
estimate = x_0;
end
```

CG Radon Tikhonov

```
function estimate = cgradontik(D1, D2, Y, x_0, lambda, n_iter)
size_Y = size(Y);
Y = Y(:);
Ax_0 = forward_radon(lambda,D1,D2,x_0);
x_0 = x_0(:);
r = Y - Ax_0;
p = r;
for k = 1:n_iter
    Ap = forward_radon(lambda,D1,D2,reshape(p,size_Y));
    alpha = (r'*r)/(p'*Ap);
    x_0 = x_0 + alpha*p;
    r_new = r - alpha*Ap;
    beta = (r_new'*r_new)/(r'*r);
    p = r_new + beta*p;
    r = r_new;
end
estimate = x_0;
end
```

Q4 d

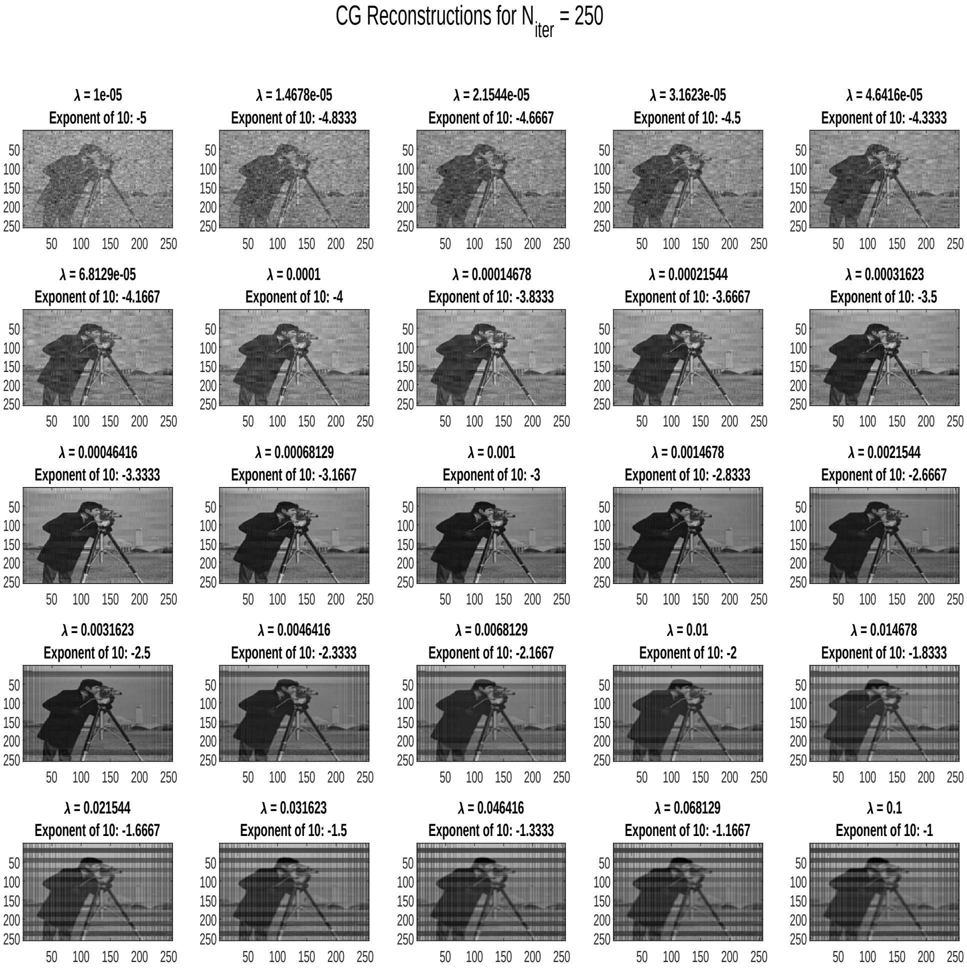
```
close all; clc; clear;
X = mat2gray(double(imread('cameraman.tif')));
H = gauss2d(15,15,0,0,2,2);
D1 = [-1 0 1];
D2 = D1';
H_flipped = flip(flip(H,1),2);
Y = conv2(X,H,'same');
sigma_s = var(Y(:));
sigma_n = sigma_s * 1e-3;
noise = sigma_n * randn(size(Y));
Y = Y + noise;
b = conv2(Y,H_flipped,'same');
n_iter = 250;
x_0 = randn(size(X));
i = 1;
f1 = figure('units','normalized','outerposition',[0.0823 0.1111 0.7063 0.7963]);
exponents = linspace(-5,-1,25);
for lambda = power(10,exponents)
    estimate = cgconvtik(H, D1, D2, b, x_0, lambda, n_iter);
    X_back = reshape(estimate,size(X));
    subplot(5,5,i)
    imagesc(X_back), colormap gray
    title({['\lambda = ',num2str(lambda)],['Exponent of 10: ',num2str(exponents(i))]})
    i = i+1;
end
sgtitle(['CG Reconstructions for N_{iter} = ',num2str(n_iter)])


X = phantom('Modified Shepp-Logan',256);
D1 = [-1 0 1];
D2 = D1';

theta_step = 180/(size(X,1));
thetas = 0:179;
Y = radon(X);
sigma_s = var(Y(:));
sigma_n = sigma_s * 1e-3;
noise = sigma_n * randn(size(Y));
Y = Y + noise;
b = iradon(Y,thetas);
b = b(2:end-1,2:end-1);
```

```
n_iter = 250;
x_0 = randn(size(X));
i = 1;
f2 = figure('units','normalized','outerposition',[0.1964 0.2250 0.4896 0.6019]);
exponents = linspace(-1,2,20);
for lambda = power(10,exponents)
    tic
    estimate = cgradontik(D1, D2, b, x_0, lambda, n_iter);
    X_back = reshape(estimate,size(X));
    subplot(4,5,i)
    imagesc(X_back), colormap gray
    title({['\lambda = ',num2str(lambda)],['Exponent of 10: ',num2str(exponents(i))]})
    i = i+1;
    toc
end
sgtitle(['CG Reconstructions for N_{iter} = ',num2str(n_iter)])
```

Comments

$$||y - Cx||_2^2 + \lambda ||Dx||_2^2 \qquad (12)$$

The Tikhonov Regularization uses the cost function given in Eqn. 12. The regularization parameter λ serves as a weighting parameter between prior and data-model-mismatch (effect of noise) that penalizes the quantity expressed by Dx. In our scenarios, this regularization enforces the reconstruction to be smooth. In figures Figures 1 and 2, the trade-off effect is observable when λ is swept through the given intervals. We see the effect of noise for small λ values where noise is amplified and a smoothness (low-pass effect) introduced by highly weighted prior in the larger λ cases.


CG Reconstructions for N_{iter} = 250

λ = 1e-05 Exponent of 10: -5

λ = 1.4678e-05 Exponent of 10: -4.8333

λ = 2.1544e-05 Exponent of 10: -4.6667

λ = 3.1623e-05 Exponent of 10: -4.5

λ = 4.6416e-05 Exponent of 10: -4.3333

λ = 6.8129e-05 Exponent of 10: -4.1667

λ = 0.0001 Exponent of 10: -4

λ = 0.00014678 Exponent of 10: -3.8333

λ = 0.00021544 Exponent of 10: -3.6667

λ = 0.00031623 Exponent of 10: -3.5

λ = 0.00046416 Exponent of 10: -3.3333

λ = 0.00068129 Exponent of 10: -3.1667

λ = 0.001 Exponent of 10: -3

λ = 0.0014678 Exponent of 10: -2.8333

λ = 0.0021544 Exponent of 10: -2.6667

λ = 0.0031623 Exponent of 10: -2.5

λ = 0.0046416 Exponent of 10: -2.3333

λ = 0.0068129 Exponent of 10: -2.1667

λ = 0.01 Exponent of 10: -2

λ = 0.014678 Exponent of 10: -1.8333

λ = 0.021544 Exponent of 10: -1.6667

λ = 0.031623 Exponent of 10: -1.5

λ = 0.046416 Exponent of 10: -1.3333

λ = 0.068129 Exponent of 10: -1.1667

λ = 0.1 Exponent of 10: -1

Figure 1: Cameraman Reconstructions

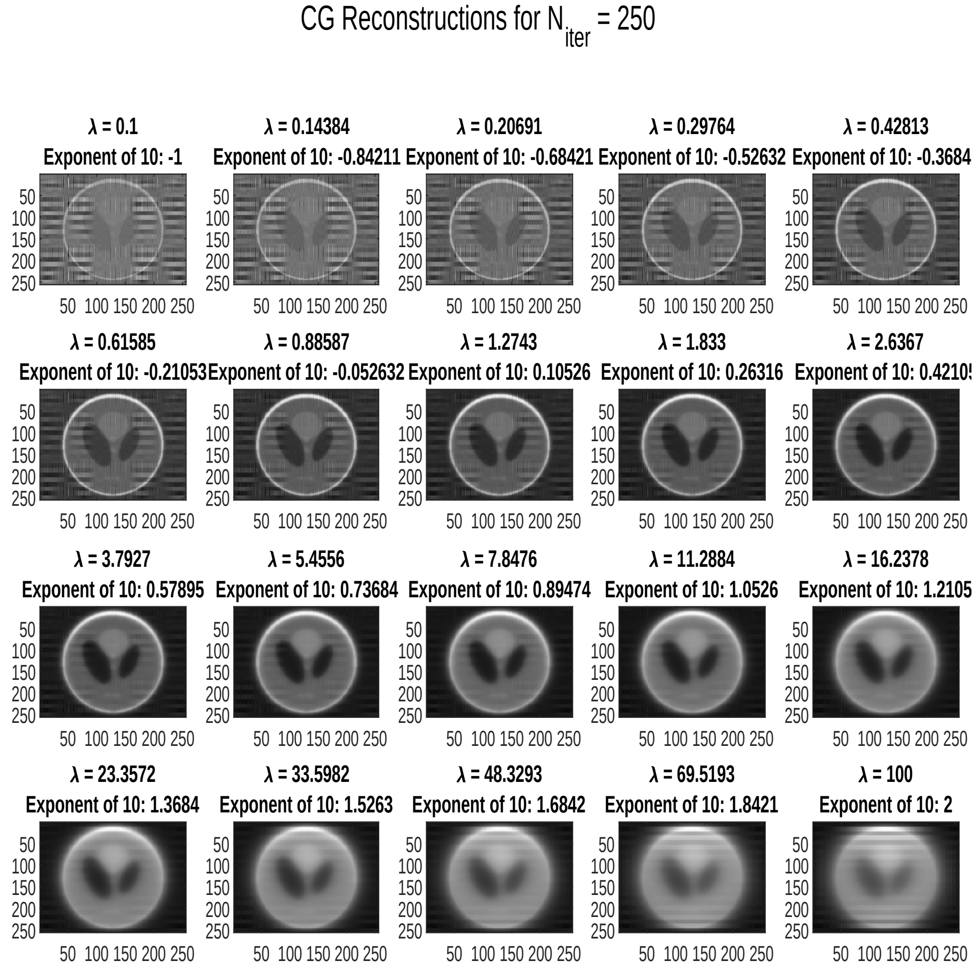


Figure 2: Shepp-Logan Reconstructions