Assignment 3

Neeraj Kumar (2020BSZ8607) bsz208607@iitd.ac.in
Krishna Chaitanya Reddy Tamataam(2018MT60785) mt6180785@maths.iitd.ac.in
Chayan Kumar Paul(2020EEZ7528) eez207528@iitd.ac.in
IIT Delhi

28th, October 2020

1 Question 1

1.1 Huffman Coding

1.1.1 Matlab Code :

```
% read the image
Image = imread('test2.png');

% calculate the frequency of each pixel
[frequency,pixelValue] = imhist(Image());

% sum all the frequencies
tf = sum(frequency) ;

% calculate the frequency of each pixel
probability = frequency ./ tf ;

% create a dictionary
dict = huffmandict(pixelValue,probability);

% get the image pixels in 1D array
imageOneD = Image(:) ;

% encoding
testVal = imageOneD ;
encodedVal = huffmanenco(testVal,dict);

% decoding
decodedVal = huffmandeco(encodedVal,dict);

% display the length
kb = 8 * 1024 ;
disp('Size of original image:')
disp(numel(de2bi(testVal))/kb) ;
disp('Size of the encoded image:')
disp(numel(encodedVal)/kb) ;
% disp(numel(de2bi(decodedVal))/kb) ;

% get the original image from 1D Array
[rows, columns, numberOfColorChannels] = size(Image);
% oi = reshape(testVal,[rows, columns, numberOfColorChannels]) ;
% imwrite(oi,'decoedg.png');
```

```
% get the decoded image from 1D Array
decodedVal = uint8(decodedVal);
ci = reshape(decodedVal,[rows, columns, numberOfColorChannels]) ;
imwrite(ci,'decoded.png');
```

1.1.2 Results

Original and the compressed image

1.1.3 Discussion

We can see the lossless compression of Huffman Coding. The size is reduced just a bit when doing the operation as a whole. The size comparison is given below :

1.2 Arithmetic Coding

Arithmetic coding is a form of entropy encoding used in lossless data compression. Arithmetic coding differs from other forms of entropy encoding, such as Huffman coding, in that rather than separating the input into component symbols and replacing each with a code, arithmetic coding encodes the entire message into a single number, a fraction n where $[0.0 \leq n < 1.0)$

1.2.1 Matlab Code

```
import encoding as e
import decoding as d
import collections
import cv2
import numpy


def encode(block_size ,imagePath="test.jpg",encodedFile="encoded"
,probabilityFile="probability.npy", float_type='float64'):
```

```
    img = cv2.imread(imagePath, cv2.IMREAD_GRAYSCALE).flatten()
    img = numpy.append(img, [0] * (block_size - len(img) % block_size))
    codes = numpy.array([])
    probability = (collections.Counter(img))
    print("Encoding Started")
    prob = [None]*256
    for i in range(0,256):
        prob[i] = probability[i]

    if float_type != 'float16' and float_type != 'float32' and float_type != '
        float64':
        float_type = 'float64'

    prob = numpy.asarray(prob)
    prob = numpy.true_divide(prob, len(img))

    for i in range(0, len(img), block_size):
        x = e.encoding(prob, img[i:(block_size + i)])
        if not(0<=x<=1):
            print("encoding error")
        codes = numpy.append(codes, [x])
    numpy.save('original.npy', img)
    print("Encoding Done:")
    numpy.save(encodedFile, codes.astype(float_type)) # save
    print(" ->" + encodedFile + " is created")
    numpy.save(probabilityFile, prob) # save
    print(" ->" + probabilityFile + " is created\n")

    return prob


def decode(block_size, rows, columns, probability, encodedFile=
"encoded", resultImage="result.jpg"):
    codes = numpy.load(encodedFile)
    probability = numpy.load(probability)
    img = numpy.array([])
    print("Decoding Started")
    for i in range(0, len(codes)):
        x = d.decoding(probability, codes[i], block_size)
        img = numpy.append(img, x)
    img = img[:rows*columns]
    print("Decoding Done:")
    img = img.reshape(rows, columns)
    cv2.imwrite(resultImage, img)
    print(" ->" + resultImage + ' is created\n')


import utility


def main():

    ################################################ ENCODING
        ################################################################

    mode = int(input("please choose mode:\n1.Encode then Decode the same files\
        n2.Encode only\n3.Decode only\n"))
    block_size = int(input("Block Size = (4 is recommended) "))

    if mode != 3:
        imagePath = input("image name: (e.g. test.png) ")
```

```
        # name of the encoded file that will be created
        encodedFile = input("output encoded file name: (e.g. encoded.npy) ")
        # name of the file to save the probability 1D numpy array in it
        probabiltyFile = input("output probability file name: (e.g. probability.
            npy) ")
        float_type = "float type: (float64 is recommended) "
    if mode == 2:
        utility.encode(block_size, imagePath, encodedFile,probabiltyFile,
            float_type)
        return
    ################################################ DECODING
       ######################################################################
    # rows = 183
    # columns = 275
    columns = int(input("number of columns (WIDTH) of the original image: (e.g.
        256) "))
    rows = int(input("number of rows (HEIGHT) of the original image: (e.g. 256)
        "))
    resultImage = input("output image name: (e.g. result.png) ")
    # name of the encoded file to be decoded
    if mode == 3:
        encodedFile = input("encoded file name: ( e.g. encoded.npy) ")
        # name of the file having the probabilty 1D numpy array
        probabiltyFile = input("probability file name: ( e.g. probability.npy)
            ")
    elif mode == 1:
        utility.encode(block_size, imagePath, encodedFile, probabiltyFile,
            float_type)
    utility.decode(block_size, rows, columns, probabiltyFile, encodedFile,
        resultImage)


main()


def get_range(probability, element, lower_bound=0, upper_bound=1):

    # elements = list(probability.keys())
    number = len(probability)
    LR = [lower_bound]
    UR = []
    # range of symbol = lower limit : lower limit + (upper bound - lower bound)
        * P[symbol]
    # lower ranges: LR = [ lower bound,...., LR[i-1] + d(P[i-1]) ]
    d = upper_bound - lower_bound
    for i in range(1, number):
        x = LR[i-1] + d*(probability[i-1])
        LR.append(x)
        UR.append(x)
    UR.append(upper_bound)

    return [LR[element], UR[element]]


def encoding(probability, sequence):
    x = get_range(probability, sequence[0])

    for i in range(1, len(sequence)):
        x = get_range(probability, sequence[i], x[0], x[1])
    return (x[0] + x[1]) / 2
```

```
import numpy


def get_range_decoding(probability, tag, lower_bound=0, upper_bound=1):

    # elements = list(probability.keys())
    number = len(probability)
    LR = [lower_bound]
    UR = []
    d = upper_bound - lower_bound
    for i in range(1, number):
        x = LR[i-1] + d*(probability[i-1])
        LR.append(x)
        UR.append(x)
        if LR[i-1] <= tag <= UR[i-1]:
            return i-1, [LR[i-1], UR[i-1]]
    UR.append(upper_bound)
    if not LR[number - 1] <= tag <= UR[number - 1]:
        print("DECODING ERROR DETECTED")
    return number-1, [LR[number-1], UR[number-1]]


def decoding(probability, tag, number):
    (element, x) = get_range_decoding(probability, tag)
    for i in range(1, number):
        (y, x) = get_range_decoding(probability, tag, x[0], x[1])
        element = numpy.append(element,y)
    return element
```

1.2.2 Results

Fig Original and Compressed Image

1.2.3 Discussion

We can see the 15 Kb image is compressed to 7 Kb. So the compression is much better now. In contrast to this the Huffman coding always produces rounding errors, because its code length is restricted to multiples of a bit. This deviation from the theoretical optimum is much higher in comparison to the arithmetic coding's inaccuracies.

The efficiency of an arithmetic code is always better or at least identical to a Huffman code.

1.3 Golomb Coding

Golomb coding is a lossless data compression method invented by Solomon W. Golomb. Symbols following a geometric distribution will have a Golomb code as an optimal prefix code, making Golomb coding highly suitable for situations in which the occurrence of small values in the input stream is significantly more likely than large values.

1.3.1 Matlab Code

```
P=imread('lena.png');
I=double(P);

[size_x,size_y]=size(I);
I_dec=zeros(size_x,size_y); %decoded image will be stored in this matrix
m=randi(max(max(I(:,:))),size_x,size_y); % generate a random set of coding
    parameters

% Encoding/Decoding for image
for i=1:1:size_x
    for j=1:1:size_y
        x = golomb_enco(I(i,j),m(i,j));
        I_dec(i,j) = golomb_deco(x, m(i,j));
    end
end
imwrite(uint8(I_dec),'Decoded.png')
% Test to verify correct decoding
if isequal(I_dec,I)
    fprintf('Decoding successful');
end

function code = golomb_enco(n, m)
q = floor(n/m);                     %compute the integer part of the quotient
r=rem(n,m);                         %compute the reaminder of n/m

q1=ones(1,q);
q_code=[q1 0];                      %unary code of quotient q
[f,e]=log2(m);                      %f,e used to check if m is a power of 2

if f==0.5 && e == 1                 %special case of m=1
    code=q_code;
else if f==0.5                      %check whether m is a power of 2
        r_code=de2bi(r,log2(m),'left-msb'); %log2(m)-bit binary code of r
    else
        a=ceil(log2(m));
        b=floor(log2(m));

        if r < (2^a - m)
            r_code=de2bi(r,b,'left-msb'); %b-bit binary representation of r
        else
            r=r+(2^a - m);
            r_code=de2bi(r,a,'left-msb'); %a-bit binary representation of r
        end

    end
        code=[q_code r_code];               %golomb code of the input
end


function n_dec = golomb_deco (codeword, m)

code_len = length(codeword);

% Count the number of 1's followed by the first 0
q = 0;
for i = 1: code_len
    if codeword(i) == 1
        q = q + 1; %count the number of 1s
    else
```

```
        ptr = i;   % first 0
        break;
    end
end

if (m == 1)
    n_dec = q;   % special case for m = 1
else
    A = ceil(log2(m));
    B = floor(log2(m));

    % decoding the remainder
    bcode = codeword((ptr+1): (ptr + B));
    r = bi2de(bcode,'left-msb');
    if r < (2^A - m)
        ptr = ptr + B;
    else
        % r is the A-bit represtation of (r + (2^A - m))
        bcode = codeword((ptr+1): (ptr + A));
        r = bi2de(bcode,'left-msb') - (2^A - m);
        ptr = ptr + A;
    end
    n_dec = q * m + r; %computing the symbol from the decoded quotient and
       remainder
end

if ~isequal(ptr, code_len)
    error('Error: More than one codeword detected!');
end
```

1.3.2 Results

Fig Original and Compressed Image

1.3.3 Discussion

We see the compression is better than Huffman coding as well as Arithmetic Coding for the image. Though the entropy remains the same for all the cases. The size of the original image is 64 Kb and the compressed image has a size of 8Kb.

```
New to MATLAB? See resources for Getting Started.

Decoding successfulSize of original image:
    64

Size of the encoded image:
     8

fx >>
```

2 Question 2

2.1 DCT coding

2.1.1 MATLAB code

```
input_image = imread('lenna.jpeg');
P = double(rgb2gray(input_image));
Q = dct(P,[],1);
R = dct(Q,[],2);

X = R(:);

[~,ind] = sort(abs(X),'descend');



coeffs = 1000;
compression_ratio = (numel(R)/coeffs);

R(abs(R) < abs(X(ind(coeffs)))) = 0;


energy = norm(X(ind(1:coeffs)))/norm(X);

fprintf('%d of %d and %f and %f coefficients are sufficient\n',coeffs,numel(R),
    compression_ratio, energy)

S = idct(R,[],2);
T = idct(S,[],1);

T  = uint8(T);
figure, imshow(T); title('IDCT');

#plot

xlabel = [87.38,131.072, 262.14,524.28,2621.44] # 3000, 2000, 1000, 500, 100
ylabel = [0.995,0.993, 0.99, 0.986,0.969]

plt.plot(ylabel,xlabel,)
plt.ylabel('compression ratio')
plt.xlabel('Energy ratio - compressed_image/Input image')
for i_x, i_y in zip(ylabel, xlabel):
    plt.text(i_x, i_y, '({}, {})'.format(i_x, i_y))
```

```
plt.show()
```

2.1.2 Results

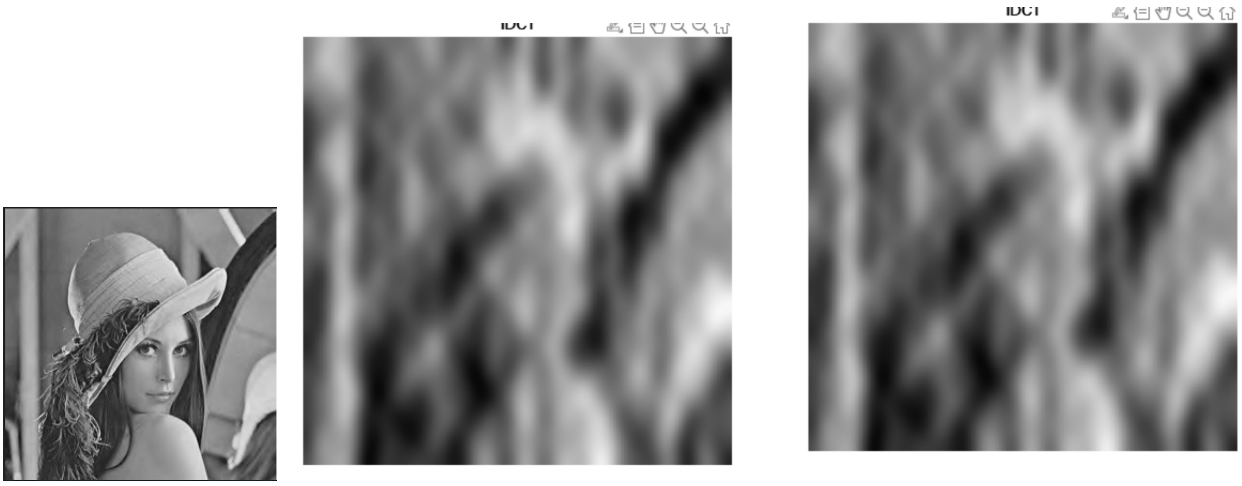

Figure 1: **Left:Original lenna image , Centre : Inverse DCT with 100 coefficients ,Right : Inverse DCT with 500 coefficient**

Figure 2: **Left : Inverse DCT with 1000 coefficients , Centre : Inverse DCT with 2000 coefficients ,Right : Inverse DCT with 5000 coefficient**

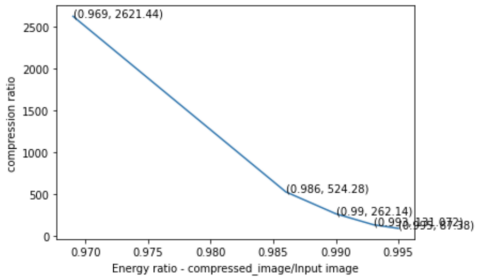

Figure 3: **DCT plot of compression with percentage oof energy retained after compression**

2.2 DFT coding

2.2.1 Python code

```
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
import numpy as np
```

```
from glob import glob
from os import mkdir, path
from numpy import linalg as LA


def IFFT(arr):
    ''' Recursive Inverse Fast Fourier Transformation implementation
    '
    '   Arguments: arr - 1d numpy array
    '   Return: 1d numpy array
    '''
    return FFT(arr, inverse=True) / arr.shape[0]


def FFT(arr, inverse=False):
    ''' Recursive Fast Fourier Transformation implementation
    '''
    sign = 1 if inverse else -1

    if arr.shape[0] == 1:
        # Return the array of lenght 1
        return arr
    else:
        # Recursively run FFT for even and odd arr's elements
        even, odd = FFT(arr[0::2], inverse=inverse), FFT(arr[1::2],
                                                         inverse=inverse)

        omega = sign * 2j * np.pi / arr.shape[0]

        # Calculate the range of values
        values = np.exp(omega * np.arange(arr.shape[0] // 2)) * odd

        return np.concatenate([even + values, even - values])


def compress(img):

    # Do 2D FFT
    img = np.apply_along_axis(FFT, 1, img)
    img = np.apply_along_axis(FFT, 0, img)

    orign_norm = LA.norm(img)

    print(np.mean(img))

    # Drop some values
#     img[img < np.mean(img) - 0 * np.std(img)] = 0

    img[img < 40000] = 0
    temp = (img.shape[0]*img.shape[1])
    print(np.count_nonzero(img),temp)

    later_norm = LA.norm(img)

    print(later_norm/orign_norm, temp/np.count_nonzero(img))



#     print(img)

    # Do 2D IFFT
```

```
    img = np.apply_along_axis(IFFT, 1, img)
    img = np.apply_along_axis(IFFT, 0, img)

    # Return real values of img
    return np.real(img)


imgs = mpimg.imread('lena.jpeg')
compres= compress(imgs)



#plot


xlist = [1.97, 2.19, 2.46,12.39,27.42 ,257.76,677.374] # 10, 200, 400, 4000,
    8000,40000, 80000
ylist= [0.96, 0.960, 0.96 , 0.959, 0.958,0.955 ,0.953]

plt.plot(ylist,xlist)
plt.ylabel('compression ratio')
plt.xlabel('Energy ratio - compressed_image/Input image')
for i_x, i_y in zip(ylist, xlist):
    plt.text(i_x, i_y, '({}, {})'.format(i_x, i_y))

plt.show()
```

2.2.2 Results

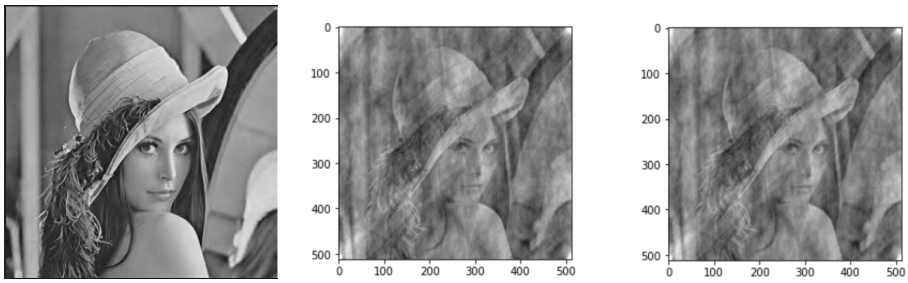

Figure 4: **Left : Original Lenna image, Centre : Inverse DFT with coefficients value less than 10 ,Right : Inverse DFT with coefficient value less than 400**

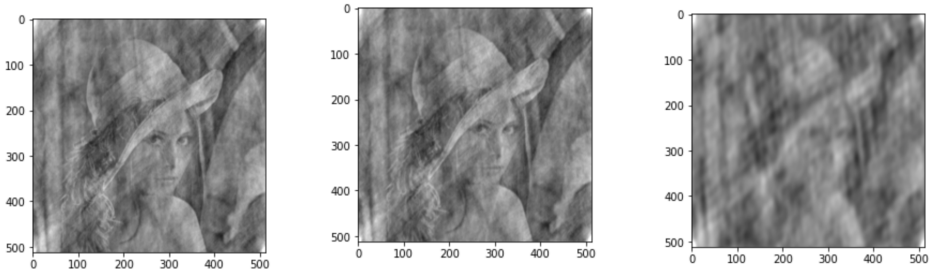

Figure 5: **Left : Inverse DFT with coefficients value less than 4000, Centre : Inverse DFT with coefficients value less than 8000,Right : Inverse DFT with coefficient value less than 80000**

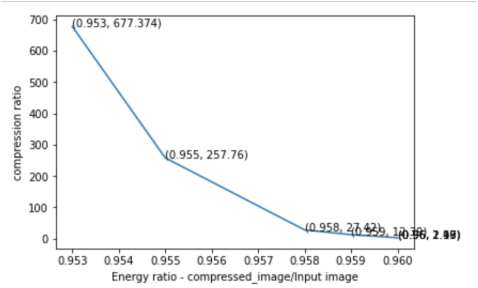

Figure 6: **DFT plot of compression with percentage of energy retained after compression**

2.2.3 Observation

- DCT has better energy compaction than DFT
- the larger the number of coefficients that get wiped out in DCT as compared to DFT
- the greater bit savings for the same loss in case of DCT.

3 Question 3

3.1 Predictive Coding

3.1.1 MATLAB code

```
I = [0  0  1  5  6;
     0  0  1  5  6;
     2  2  4  7  8;
     3  3  7  4  2;
     6  6  5  1  0];

I = uint8(I);
figure;
imshow(I)
figure;
imhist(I)
dim = size(I);

B = padarray(double(I),[1  1],'replicate');

start = [2,2] + [-1  ,  -1];
last = start + dim -1 ;
f1 = B(start(1):last(1),start(2):last(2));

start = [2,2] + [-1  ,  0];
last = start + dim -1 ;
f2 = B(start(1):last(1),start(2):last(2));

start = [2,2] + [-1  ,  1];
last = start + dim -1 ;
f3 = B(start(1):last(1),start(2):last(2));

start = [2,2] + [0  ,  -1];
last = start + dim -1 ;
f4 = B(start(1):last(1),start(2):last(2));

start = [2,2] + [0  ,  0];
last = start + dim -1 ;
f5 = B(start(1):last(1),start(2):last(2));
```

```
start = [2,2] + [0  , 1];
last = start + dim -1 ;
f6 = B(start(1):last(1),start(2):last(2));

start = [2,2] + [1  , -1];
last = start + dim -1 ;
f7 = B(start(1):last(1),start(2):last(2));

start = [2,2] + [1  , 0];
last = start + dim -1 ;
f8 = B(start(1):last(1),start(2):last(2));

start = [2,2] + [1  , 1];
last = start + dim -1 ;
f9 = B(start(1):last(1),start(2):last(2));

s1 = reshape(f1, [dim(1)*dim(2),1]);
s2 = reshape(f2, [dim(1)*dim(2),1]);
s3 = reshape(f3,[dim(1)*dim(2),1]);
s4 = reshape(f4, [dim(1)*dim(2),1]);
s5 = reshape(f5, [dim(1)*dim(2),1]);
s6 = reshape(f6, [dim(1)*dim(2),1]);
s7 = reshape(f7, [dim(1)*dim(2),1]);
s8 = reshape(f8, [dim(1)*dim(2),1]);
s9 = reshape(f9, [dim(1)*dim(2),1]);

m1 = mean(s1);
m2 = mean(s2);
m3 = mean(s3);
m4 = mean(s4);
m5 = mean(s5);
m6 = mean(s6);
m7 = mean(s7);
m8 = mean(s8);
m9 = mean(s9);


% for 2 w1*f(x-1,y) + w2* f(x,y-1)
M1 = cov([s2 s4]) + [m2;m4]*[m2 m4];
B = cov([s5  s2]);
C = cov([s5  s4]);
b = [B(1,2) + m5*m2 ; C(1,2) + m5 * m4];

param = inv(M1)*b;
restored = param(1,1)*f2 + param(2,1)*f4;
error_before = mean(mean((I - uint8(restored)).^2))
% restored = uint8(restored);
% figure
% imhist(restored);

error = double(I) - restored ;

avg = (mean(error(error>=0))  - mean(error(error < 0)))/2 ;
avg;


[m,n] = size(I);
for i = 1 : m
    for j = 1:n
        if(error(i,j) >= avg)
            error(i,j) = avg;
```

```
        elseif (error(i,j) <= -avg)
                error(i,j) = -avg;
        else
            error(i,j) = 0;
        end
    end
end

error;

final_restored = uint8(restored + error);
final_error = mean(mean((I- final_restored).^2))
figure;
imshow(final_restored);
figure;
imhist(final_restored);
```

```
M1 = cov([s2 s4 s6 s8]) + [m2;m4;m6;m8]*[m2 m4 m6 m8];
B = cov([s5  s2]);
C = cov([s5  s4]);
D = cov([s5  s6]);
E = cov([s5  s8]);
b = [B(1,2) + m5*m2 ; C(1,2) + m5*m4 ; D(1,2) + m5*m6 ; E(1,2)+ m5*m8];

param = inv(M1)*b;
restored = param(1,1)*f2 + param(2,1)*f4 + param(3,1)*f6 + param(4,1)*f8;
error_before = mean(mean((I - uint8(restored)).^2))
% restored = uint8(restored);
% figure
% imhist(restored);

error = double(I) - restored ;

avg = (mean(error(error>=0))  - mean(error(error < 0)))/2 ;
avg;


[m,n] = size(I);
for i = 1 : m
    for j = 1:n
        if(error(i,j) >= avg)
            error(i,j) = avg;
        elseif (error(i,j) <= -avg)
                error(i,j) = -avg;
        else
            error(i,j) = 0;
        end
    end
end

error;

final_restored = uint8(restored + error);
final_error = mean(mean((I- final_restored).^2))
figure;
```

```
imshow(final_restored);
figure;
imhist(final_restored);
```

```
k1 = cov([s1 s2 s3 s4 s6 s7 s8 s9]) + [m1;m2;m3;m4;m6;m7;m8;m9]*[m1 m2 m3 m4 m6
    m7 m8 m9];
k2 = cov([s5  s1]);
k3 = cov([s5  s2]);
k4 = cov([s5  s3]);
k5 = cov([s5  s4]);
k6 = cov([s5  s6]);
k7 = cov([s5  s7]);
k8 = cov([s5  s8]);
k9 = cov([s5  s9]);
b = [k2(1,2) + m5*m1 ; k3(1,2)+ m5*m2 ; k4(1,2)+ m5*m3 ; k5(1,2)+ m5*m4;k6(1,2)+
     m5*m6 ; k7(1,2)+ m5*m7 ; k8(1,2)+ m5*m8 ; k9(1,2)+ m5*m9];

param = inv(k1)*b;
restored = param(1,1)*f1 + param(2,1)*f2 + param(3,1)*f3 + param(4,1)*f4+param
    (5,1)*f6 + param(6,1)*f7 + param(7,1)*f8 + param(8,1)*f9;
error_before =mean(mean((I - uint8(restored)).^2))
% restored = uint8(restored);
% figure
% imhist(restored);

error = double(I) - restored ;

avg = (mean(error(error>=0))  - mean(error(error < 0)))/2 ;
avg;


[m,n] = size(I);
for i = 1 : m
    for j = 1:n
        if(error(i,j) >= avg)
            error(i,j) = avg;
        elseif (error(i,j) <= -avg)
                error(i,j) = -avg;
        else
            error(i,j) = 0;
        end
    end
end

error;

final_restored = uint8(restored + error);
final_error = mean(mean((I- final_restored).^2))
figure;
imshow(final_restored);
figure;
imhist(final_restored);
```

3.1.2 Procedure

We have to find the estimator of the image which minimizes the mean square error between image and the estimator.The estimator is assumed to be a linear combination of the image neighbour and we should estimate the coefficients of each neighbours. further more the error is between the image and the estimator is quantified by the calculating the avg of the +ve error terms and the average of the -ve error terms and taking the avg of these two. if error is ¿= avg error then it is approximated to avg error. if error lies between -avg to +avg error then error is 0. if error ¡= -avg then it is approximated as - avg.The Final estimator is the uint8 of the (estimator + quantified error). Given below is a derivation for finding out the optimal parameter for the mse error.

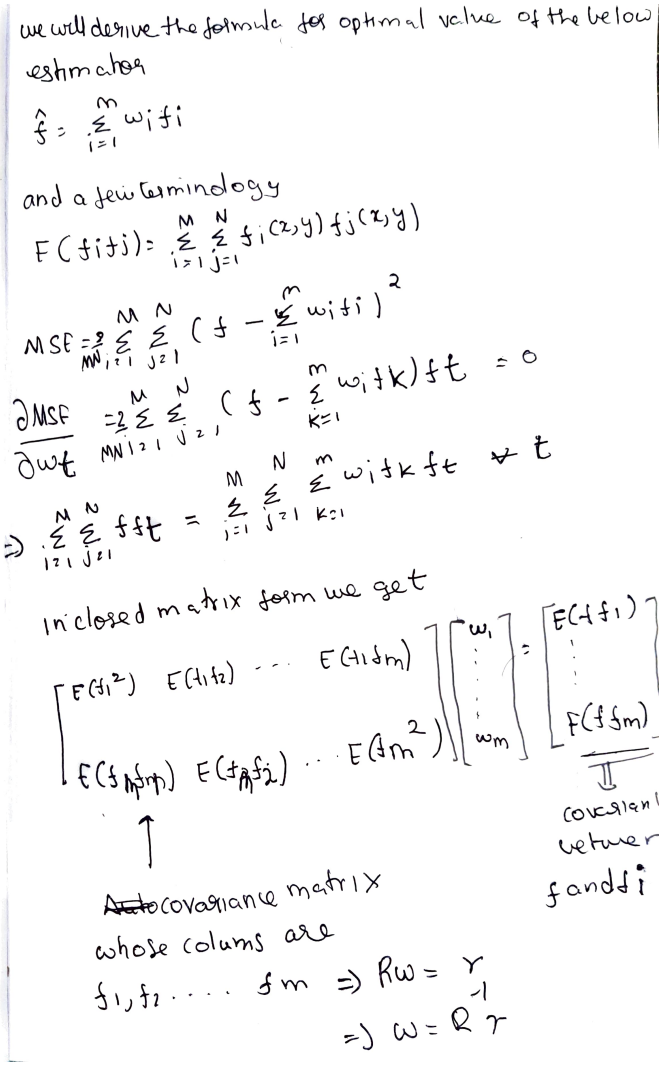

Figure 7: **Estimating the optimal predictor using MSE derivation**

Initially we replicate pad the Image and find out the values of f(x-1,y) , f(x,y-1) and so on and then apply the above process by using 2 neighbours , 4 neighbours and 8 neighbours.

3.1.3 Results

0	0	1	5	6
0	0	1	5	6
2	2	4	7	8
3	3	7	4	2
6	6	5	1	0

0	0	0	4	5
0	0	0	4	5
1	1	3	6	7
3	3	5	5	4
5	5	6	3	1

0	0	1	4	6
0	1	2	5	6
2	3	5	7	8
4	4	5	5	3
6	5	5	2	1

0	0	1	5	6
0	0	1	5	6
2	2	5	7	8
3	4	5	4	2
6	6	5	1	0

Figure 8: **from left to right: original matrix, 2 neighbour estimate, 4 neighbour estimate, 8 neighbour estimate**

3.1.4 Results

Figure 9: **from left to right: original image, 2 neighbour estimate, 4 neighbour estimate, 8 neighbour estimate**

3.1.5 Observation

Table 1: MSE error for different diff estimates

Image	**2-nei**	**4-nei**	**8-nei**
matrix	0.6800	0.2400	0.1600
cameraman	9.5665	4.7772	4.0351