ELL786 REPORT

ASSIGNMENT 1

SARANSH AGARWAL (2019MT60763)
HARSH SHARMA (2019MT60628)

PART-1

This is a programming assignment which requires you to encode and decode binary message bits using repetition codes and arithmetic codes.

Input: A text file of size 1 KB of your choice.

Batman is a superhero who appears in American comic books published by DC Comics. The character was created by artist Bob Kane and writer Bill Finger, and debuted in the 27th issue of the comic book Detective Comics on March 30, 1939. In the DC Universe continuity, Batman is the alias of Bruce Wayne, a wealthy American playboy, philanthropist, and industrialist who resides in Gotham City. Batman origin story features him swearing vengeance against criminals after witnessing the murder of his parents Thomas and Martha, a vendetta tempered with the ideal of justice. He trains himself physically and intellectually, crafts a bat-inspired persona, and monitors the Gotham streets at night. Kane, Finger, and other creators accompanied Batman with supporting characters, including his sidekick Robin, allies Alfred Pennyworth, James Gordon, and Cat woman, and foes such as the Penguin, the Riddler, Two-Face, and his archenemy, the Joker. Kane conceived Batman in early 1939 to capitalize on the popularity of DC Superman.

Figure 1: Following Text file is used for Encoding and Decoding

Computing Environment: MATLAB / Python without using built-in libraries for arithmetic code/repetition code

Experiment 1:

1)Read the input text-file and convert it to a binary string, say of length M bits.

```
#reading the input text file
txt_file = open("sample.txt", "r")
#data is the input string
data = txt_file.read()
txt_file.close()
```

Figure 2: Wrapper Code

```
#text_to_binary is function which converts the string into binary string
def text_to_binary(s):

    n = len(s)
    s1 = ""

    for i in range(n):
        s2 = ""

        #to get ascii value of character
        val = ord(s[i])

        #converting the ascii value into 7 bits binary string
        while(val>0):
            if(val%2==0):
                s2+='0'
            else:
                s2+='1'
            val = val//2

        #if length of binary string is less than 7 then appending zeros
        c=len(s2)
        if(c<7):
            for i in range(7-c):
                s2+='0'
        s2 = s2[::-1]

        #appending the binary string of each character
        s1+=s2
    return s1
```

Figure 3: Wrapper Code

2) Generate a random binary error pattern of length M with hamming weight d such that the non-zero entries are uniformly distributed across M bits.

```
#generate_random_binary_string function generates a binary string of length n with hamming weight d
#such that non-zero entries are distributed uniformly
def generate_random_binary_string(n,d):
    s = ""

    for i in range(n):
        s+='0'

    #random.sample generate list of uniformly distributed d random numbers in range 0-n
    randomlist = random.sample(range(n), d)

    for i in range(d):
        c = randomlist[i]
        #using slicing of string
        s = s[:c]+'1'+s[c+1:]
    return s
```

Figure 4: Wrapper Code

3) XOR the above error pattern with the message bits to obtain a new sequence denoted by y.

```
#xor_binary_strings function takes xor of two binary strings s1 and s2
def xor_binary_strings(s1,s2):

    n = len(s1)
    s = ""

    for i in range(n):

        #if both bits are same then xor will be 0
        if(s1[i]==s2[i]):
            s+='0'
        #if both bits are different then xor will be 0
        else:
            s+='1'
    return s
```

Figure 5: Wrapper Code

4) Using y, retrieve the text-file without any error detection/ correction coding.

```
#binarystring_textfile function converts the binary string into text string
def binarystring_textfile(s):

    s1=""
    n = len(s)
    i=0

    while(i<n):

        #converting the chuncks of 7 bits into character using ascii
        s1=s1+chr(int(s[i:i+7], 2))
        i=i+7
    return s1
```

Figure 6: Wrapper Code

5) In the decoded file, compute the percentage of modified characters with respect to the input file.

```
#compare function calculates percentage of modified characters with
#respect to the input file
def compare(s1,s2):

    n = len(s1)
    c=0
    for i in range(n):
        if(s1[i]!=s2[i]):
            c+=1
    print("Number of Different characters are",c)
    print("Percentage of Modified characters is",(c/n)*100)
```

Figure 7: Wrapper Code

6) Repeat the above experiment by varying the value of d = {10, 100, 200, 500, 5000}.

VARIATION OF % OF MODIFIED CHARACTERS WITH THE VALUE OF D

Total No. Of Characters	D value	No. Of Different Characters	Percentage Of Modified Characters
1024	10	10	0.97%
1024	100	98	9.57%
1024	200	180	17.57%
1024	500	400	39.06%
1024	5000	1023	99.90%

Figure 8: Variation of % of Modified Characters with the value of D

CONCLUSION

From the Figure 8 we can observe that as the value of d increase the percentage of modified characters also increases. We can observe that when value of d is 10 only 10 characters are modified about 0.97% with respect to original text. So, we can conclude that output text after decoding is similar to input text in this case. While if we take value of d as 5000 almost all the characters are different about 99.90% with respect to original text. Since the value of D lies between 10 to 5000 so the percentage varies from 9.57% to 39.06% with respect to original text

Experiment 2

1) Read the input text-file and convert it into binary string and divide it into several chunks such that each chunk is of size k.

```
# divide input string s into parts of size k
# last part is of size <= k
def divide_str(s, k):
    parts = []
    i = 0
    n = len(s)
    while i < n:
        parts.append(s[i:i + k])
        i += k
    return parts
```

Figure 9: Wrapper Code

2) Encode each chunk by using (i) Huffman, (ii) (4-bit per symbol) Extended Huffman, and repetition codes and (iii) Arithmetic codes. After encoding each chunk, the encoded chunks are appended and let the total number of bits generated from the entire text-file be M′.

```
import heapq
from functools import reduce

class Letter:
    # symbol : symbol of an alphabet
    # freq : statistical freq of the symbol
    def __init__(self, symbol, freq):
        self.symbol = symbol
        self.freq = freq
        self.left = None
        self.right = None

    def __lt__(self, other):
        return self.freq < other.freq

    def __str__(self):
        return self.symbol

    def get_left(self):
        return self.left

    def set_left(self, letter):
        self.left = letter

    def get_right(self):
        return self.right

    def set_right(self, letter):
        self.right = letter

    def is_leaf(self):
        return self.left is None and self.right is None
```

```
def huffman(alphabet, prob_model):
    letter_list = []
    n = len(alphabet)
    for i in range(n):
        letter_list.append(Letter(alphabet[i], prob_model[i]))

    heapq.heapify(letter_list)
    while len(letter_list) != 1:
        l1 = heapq.heappop(letter_list)
        l2 = heapq.heappop(letter_list)

        l3 = Letter(l1.symbol + l2.symbol, l1.freq + l2.freq)
        l3.set_left(l1)
        l3.set_right(l2)

        heapq.heappush(letter_list, l3)

    # len(letter_list) == 1
    return letter_list[0]

# generating codewords
# alphabet - source alphabet set
# coding_tree - source huffman coding_tree
def compute_codebook(alphabet, coding_tree):
    letter_codewords = {}
    for s in alphabet:
        letter_codewords[reduce(lambda x, y: x+y, s, "")] = ""
    aux_generate_codewords("", coding_tree, letter_codewords)
    return letter_codewords

def aux_generate_codewords(prefix, coding_tree, letter_codeword_dict):
    if coding_tree.is_leaf():
        letter_codeword_dict[reduce(lambda x, y: x+y, coding_tree.symbol, "")] = prefix

    else:
        aux_generate_codewords(prefix + '0', coding_tree.get_left(), letter_codeword_dict)
        aux_generate_codewords(prefix + '1', coding_tree.get_right(), letter_codeword_dict)
```

Figure 10: Wrapper Code

```
# encode a sequence using extended huffman coding
# k - size of each symbol
# s - sequence
# codebook - {symbol : codeword} dictionary
def extended_huffman_encode(k, s, codebook):
    output = ""
    i = 0
    n = len(s)
    while i < n:
        output += codebook[s[i:i+k]]
        i += k
    return output


# encode a sequence using huffman coding
def huffman_encode(s, codebook):
    return extended_huffman_encode(1, s, codebook)
```

Figure 11: Wrapper Code

```python
# rep_encode function takes string and integer as an argument and encodes it
# s is binary string to encode
# r is the value that is agreed commonly between transmitter and receiver
# r must preferably an odd number

def rep_encode(s, r):

    output = ""
    for c in s:
        output += c*r

    return output
```

Figure 12: Wrapper Code

```python
import math

# finds the predecessor of 'symbol a' in alphabet set
def pred(a, alphabet):
    n = len(alphabet)
    for i in range(n):
        if alphabet[i] == a:
            return i - 1

# computes cdf for each symbol in the alphabet set
# alphabet - alphabet set
# prob_model - probability of each symbol in the alphabet
def compute_cdf(alphabet, prob_model):
    cdf_dict = dict()
    f = 0
    n = len(alphabet)
    for i in range(n):
        f += prob_model[i]
        cdf_dict[alphabet[i]] = f
    return cdf_dict

# alphabet - alphabet_set
# prob_model - probability of each symbol in the alphabet
def generate_tag(x, alphabet, prob_model):
    # x is a sequence of symbols (to encode)
    # cdf_dict is a dictionary with keys as symbols and values as cdf
    cdf_dict = compute_cdf(alphabet, prob_model)
    n = len(x)
    _l = 0
    _u = 1
    for i in range(n):
        j = pred(x[i], alphabet)
        l = _l + (_u - _l) * cdf_dict[alphabet[j]] if j >= 0 else _l
        u = _l + (_u - _l) * cdf_dict[x[i]]

        _l = l
        _u = u
    return (_l + _u) / 2
```

Figure 13: Wrapper Code

```python
# returns probability of a sequence
# s - sequence for which to compute probability for
# alphabet - alphabet set
# prob_model - probability of each symbol in the alphabet
def prob_seq(s, alphabet, prob_model):
    prob = 1.0
    for c in s:
        prob *= prob_model[alphabet.index(c)]
    return prob

# encoding tag into binary up to "length" many bits
# 0<=tag<1
def encode_tag(tag, length):
    code = ""
    i = 0
    while tag != 0.0 and i < length:
        tag *= 2
        code += str(math.floor(tag))
        tag = tag - math.floor(tag)
        i += 1
    if len(code) < length:
        code += "0"*(length - len(code))
    return code

# generates binary code for a sequence x with given alphabet set and probability model
def generate_binary_code(x, alphabet, prob_model):
    # tag value and probability of the sequence
    prob = prob_seq(x, alphabet, prob_model)
    tag = generate_tag(x, alphabet, prob_model)
    length = math.ceil(math.log(1 / prob, 2)) + 1
    return encode_tag(tag, length)
```

```python
# decipher the tag value
# k: length of the sequence to decipher
# cdf_dict: dictionary of the (letter, cdf) pair
def decipher_tag(tag, k, cdf_dict):
    _l = 0
    _u = 1
    output = ""
    for i in range(k):
        t = (tag - _l)/(_u - _l)
        _f = 0    # value of f in prev iteration
        _s = ""   # value of s in prev iteration
        for s, f in cdf_dict.items():
            if _f <= t < f:
                output += s
                l = _l + (_u - _l) * cdf_dict[_s] if _s != "" else _l
                u = _l + (_u - _l) * cdf_dict[s]

                _l = l
                _u = u
                break
            else:
                _f = f
                _s = s
    return output
```

Figure 14: Wrapper Code

3) Generate a random binary error pattern of length M′ with hamming weight d such that the non-zero entries are uniformly distributed across M′ bits.

```
#generate_random_binary_string function generates a binary string of length n with hamming weight d
#such that non-zero entries are distributed uniformly
def generate_random_binary_string(n,d):
    s = ""

    for i in range(n):
        s+='0'

    #random.sample generate list of uniformly distributed d random numbers in range 0-n
    randomlist = random.sample(range(n), d)

    for i in range(d):
        c = randomlist[i]
        #using slicing of string
        s = s[:c]+'1'+s[c+1:]
    return s
```

Figure 15: Wrapper Code

4) XOR the above error pattern with the message bits to obtain a new sequence denoted by y.

```
#xor_binary_strings function takes xor of two binary strings s1 and s2
def xor_binary_strings(s1,s2):

    n = len(s1)
    s = ""

    for i in range(n):

        #if both bits are same then xor will be 0
        if(s1[i]==s2[i]):
            s+='0'
        #if both bits are different then xor will be 0
        else:
            s+='1'
    return s
```

Figure 16: Wrapper Code

5) Using y, retrieve the text-file by decoding.

```
# decode a binary string using huffman decompression
# s - string to decode
# coding_tree - coding_tree for the source
def huffman_decode(s, coding_tree):
    # s is a binary string
    # coding tree must not be empty
    letter_node = coding_tree
    output = ""
    for c in s:
        if c == '0':
            letter_node = letter_node.get_left()
        else:
            letter_node = letter_node.get_right()
        if letter_node.is_leaf():
            output = output + reduce(lambda x, y: x+y, letter_node.__str__(), "")
            letter_node = coding_tree  # back to the root

    if letter_node != coding_tree:
        return output    # message is not decodable with given alphabet set
    return output
```

Figure 17: Wrapper Code

```
# rep_decode function takes string and integer as an argument and decodes it
# s is binary string to encode
# r is the value that is agreed commonly between transmitter and receiver
# decoding using majority rule

def rep_decode(s, r):

    i = 0
    n = len(s)

    output = ""
    errors_corrected = 0
    while i < n:
        count0 = s.count('0', i, i+r)
        if count0 > r//2:
            output += '0'
        #else append 1
        else:

            output += '1'
        i += r
        errors_corrected += 1 if count0 != r else 0
    return output, errors_corrected
```

Figure 18: Wrapper Code

```
# decode a binary sequence s using arithmetic decoding given k is original sequence length
def arith_decode(s, k, alphabet, prob_model):   # k is length of original string in the message
    # s is a binary string
    tag = 0.0
    i = 1
    for c in s:
        if c == '1':
            tag += math.pow(2, -i)
        i += 1
    return decipher_tag(tag, k, compute_cdf(alphabet, prob_model))
```

Figure 19: Wrapper Code

6) Compute the number of errors you could detect, and the number of errors you could correct.

Since, error correcting code used in this assignment is repetition code, before we send our message, each bit in the compressed transmitted message is repeated r (an odd number) number of times.

At the receiver side, if a block of size r in the received message has more than r/2 bits of one kind (say x) then we decode that block as x (majority rule)

If a block of size r has all 1's or all 0's then we say no error detected/corrected

If a block of size r has some combination of 1's and 0's then we say an error has been detected, we correct that error by decoding that block using majority rule

7) In the decoded file, compute the percentage of modified characters with respect to the input file.

```
# compare differences in two string of same sizes
def compare(s1, s2):
    n = len(s1)
    m = len(s2)
    c = 0
    for i in range(n):
        if i < m and s1[i] != s2[i]:
            c += 1
        elif i == m:
            c += n-m
            return c
    return c
```

Figure 20: Wrapper Code

8) Repeat the above experiment by varying the value of d = {10, 100, 200, 500, 5000}.

9) Compare the results between the three coding techniques and experiment-1 without any such technique.

VARIATION OF % OF MODIFIED CHARACTERS WITH THE VALUE OF D FOR EXP1

Total No. Of Characters	D value	No. Of Different Characters	Percentage Of Modified Characters
1024	10	10	0.97%
1024	100	98	9.57%
1024	200	180	17.57%
1024	500	400	39.06%
1024	5000	1023	99.90%

Figure 21: Variation of % of Modified Characters with the value of D

VARIATION OF % OF MODIFIED CHARACTERS WITH THE VALUE OF D FOR EXPERMINENT 2

Total No. Of Characters	D value	Percentage Of Modified Characters(Huffman)	Percentage Of Modified Characters(Extended Huffman)	Percentage Of Modified Characters(Arithmetic)
1024	10	0.0%	0.0%	0.0%
1024	100	0.0%	0.0%	0.0%
1024	200	0.0%	0.0%	0.0%
1024	500	0.20%	0.0%	0.19%
1024	5000	14.94%	89.84%	21.19%

Figure 22: Variation of % of Modified Characters with the value of D (k=16 and r=5)

CONLCUSION

We can see the very high error rate for extended Huffman coding because of symbol losses i.e., when message is placed into fixed length blocks and transmitted through a communication channel with bit inversion errors, the resulting code sequence on the receiver side may not be even decodable with the given codebook of extended Huffman and thus leading to symbol losses

From above two tables clearly repetition coding helps reduce number of errors when passing the message through this noisy channel and combined with compression the later beats the former as lesser number of bits would be needed to transfer via the channel improving the overall performance of communication

Define, compression ratio of an encoding scheme as follows

$$\text{Compression Ratio(encoding scheme)} = \frac{\text{\#bits after compression}}{\text{\#bits before compression}}$$

Compression Ratio for each compression coding scheme

k value	Compression Ratio (Huffman)	Compression Ratio (Extended Huffman)	Compression Ratio (Arithmetic)
8	1.00	0.902	1.075
16	1.00	0.958	1.041
32	1.00	0.992	1.024
48	1.00	0.998	1.019
52	1.00	0.997	1.014

Figure 23: Variation of compression ratio with different values of k

CONLCUSION

1. The variant of arithmetic coding and decoding done in class would work only if floats had unlimited precision, but in python it only works for short messages of the order of 54 encoded bits because of the fact that floats in python have 16 significant digits, which amounts to $16 * \frac{log(10)}{log(2)} \approx 53.15$ bits, thus k must not be more than or equal to 54.
2. As we observe from above table that compression of binary strings via arithmetic coding leads to a compression ratio of more than 1.
3. Compression ratio of Huffman coding is 1 as it doesn't really compress a binary string.

VARIATION OF NUMBER OF ERRORS CORRECTED WITH THE VALUE OF D

D value	Number Of Errors Corrected (Huffman)	Number Of Errors Corrected (Extended Huffman)	Number Of Errors Corrected (Arithmetic)
10	3674	3556	3733
100	3717	3592	3784
200	3761	3647	3822
500	3910	3784	3984
5000	5504	5348	5674

Figure 24: Variation of Number of error corrected with the value of D(k=16 and r=5)

PART-2

This is a programming assignment which requires you to apply Discrete Wavelet Transform on an image and see how much compression you can achieve without a prominent loss.

Input:

1) A Gray-scale png image (512x512 matrix) of your choice.

2) Levels of decomposition

3) Threshold/s for different sub-bands

Experiment:

1) Read the input image-file and convert it to a matrix.

```
#Reading the input image-file of 512x512 and converting it into a matrix
img=image.imread('sample.gif')
print(img)
```

Figure 25: Wrapper Code

2) Take the 2-D DWT of the image.

```
coeffs = pywt.wavedec2(img, 'db1',level=1)
LL,(LH,HL,HH)= coeffs
```

Figure 26: Wrapper Code

3) Having found the DWT coefficients of the image, compute the average energy of each sub-band. Now, we are going to "prune" some of the coefficients in each sub-band. Keep those with the highest energy, by selecting as sub-band specific threshold and discard the rest. Experiment with different values for the threshold and see how the reconstructed image looks after taking the 2-D inverse DWT.

```
#calculating the averge energy of subband
def average_energy_subband(matrix):
    s=0
    m,n = matrix.shape
    for i in range(m):
        for j in range(m):
           s+=((matrix[i][j])*(matrix[i][j]))
    return np.sqrt(s/(m*m))

#pruning the matrix
def pruning(matrix,thresolds):
    m,n = matrix.shape
    for i in range(m):
        for j in range(m):
            if(matrix[i][j]<thresolds):
                matrix[i][j]=0
    return matrix

#setting matrix to zero
def set_to_zero(matrix):
    m,n = matrix.shape
    for i in range(m):
        for j in range(m):
            matrix[i][j]=0
    return matrix


matrix = pywt.waverec2(coeffs, 'db1')
data = im.fromarray(matrix)
data.save('dummy_pic1.gif')
data.show()
```

Figure 27: Wrapper Code

4) Next, vary the number of coefficients retained and see how many coefficients you can discard until the image degradation is perceptually significant.

```
#restoring top k elements of a matrix
def restore(matrix,k):
    m,n = matrix.shape
    p = np.zeros((m*m,3))
    for i in range(m):
        for j in range(m):
            p[j+i*m][0] = matrix[i][j]
            p[j+i*m][1] = i
            p[j+i*m][2] = j
    p[p[:, 0].argsort()]
    for i in range(k,m*m):
        matrix[int(p[i][1])][int(p[i][2])] = 0
    return matrix
```

Figure 28: Wrapper Code

VARYING THRESOLDS FOR LEVEL OF DECOMPOSITION = 1

AVERAGE ENERGY OF SUBBANDS: LL = 194.95, LH = 12.58, HL =15.89, HH = 7.09

We set initial thresholds as the average energy of sub bands and then varies value of thresholds until there is no significant image degradation.

Figure 29: Reconstructed Image

Figure 30: Original Image

First, we set thresholds of bands as LL = 194, LH = 12, HL = 15, HH = 7 and we can observe significant image degradation so we now decrease threshold values.

Figure 31: Reconstructed Image

Figure 32: Original Image

Next, we take thresholds of bands as LL = 194, LH = 12, HL = 15, HH = 3.5 and we can observe significant image degradation.

CONCLUSION

From above two figures we can conclude that changing threshold of sub band HH has not much affected the image so we can set HH matrix to be zero for our further analysis

Figure 33: Reconstructed Image

Figure 34: Original Image

Next, we take thresholds of bands as LL = 194, LH = 12, HL = 7.5, HH = 3.5 and we can observe significant image degradation.

Figure 35: Reconstructed Image

Figure 36: Original Image

Next, we take thresholds of bands as LL = 194, LH = 6, HL = 7.5, HH = 3.5 and we can observe significant image degradation.

CONCLUSION

From the above two figures we can conclude that changing threshold of sub band HL, LH, HH has not much affected the image so we can set LH, HL, HH matrix to be zero for our further analysis.

Figure 37: Reconstructed Image

Figure 38: Original Image

Now, we vary threshold of sub band LL only and set other sub bands to zero as they don't affect much. Here we have taken threshold of LL = 97 and there is significant image degradation.

Figure 39: Reconstructed Image

Figure 40: Original Image

Now, we have taken threshold of LL = 50 and there is not much image degradation.

Figure 41: Reconstructed Image

Figure 42: Original Image

Now, we have taken threshold of LL = 40 and there is not much image degradation.

Figure 43: Reconstructed Image

Figure 44: Original Image

Now, we have taken threshold of LL = 30 and there is not much image degradation.

Figure 45: Reconstructed Image

Figure 46: Original Image

Now, we have taken threshold of LL = 10 and there is not much image degradation.

VARIATION OF SIZE OF COMPRESSED IMAGE AND THRESHOLD OF LL SUBBAND SETTING OTHER SUB BAND TO ZERO MATRIX

THRESHOLD	SIZE OF ORIGINAL IMAGE	SIZE OF COMPRESSED IMAGE
97	292KB	117KB
50	292KB	140KB
40	292KB	145KB
30	292KB	151KB
10	292KB	164KB

Figure 47': size of compressed image variation with the threshold of LL sub band

CONCLUSION

We varied the threshold of sub-bands until there is no significant image degradation for sub-bands LL, LH, HL, HH respectively. We start by taking the initial threshold as the average energy of sub-bands. In Figure 29,31,33,35 respectively we varied thresholds for sub-band HH, LH, HL and we observe not much difference in the image. So next we varied the threshold of sub-band LL by completely pruning sub-bands HL, LH, HH. In figure 39,41,43,45 respectively we observe no significant image degradation. Figure 47', shows the variation of the threshold of sub-band LL and the size of compressed image. To conclude, by setting the threshold of LL = 50 and completely pruning HL, LH, HH we get the compressed image of size 140 Kb.

VARYING THRESOLDS FOR LEVEL OF DECOMPOSITION = 2

AVERAGE ENERGY OF SUBBANDS: LL = 392.09, LH = 31.4, HL =40.64, HH = 18.74,

LH1 = 12.58, HL1 = 15.89, HH1 = 7.09

We set initial thresholds as the average energy of sub bands and then varies value of thresholds until there is no significant image degradation.

Figure 47: Reconstructed Image

Figure 48: Original Image

First, we set thresholds of bands as LL = 392, LH = 31, HL = 40, HH = 18, LH1=12, HL1=16, HH1=7 and we can observe significant image degradation so we now decrease threshold values.

Figure 49: Reconstructed Image

Figure 50: Original Image

Next, we half the thresholds of bands and set as LL = 196, LH = 15.5, HL = 20, HH = 9, LH1=6, HL1=8, HH1=3.5 and we can observe significant image degradation so we further have to decrease threshold values.

Figure 51: Reconstructed Image

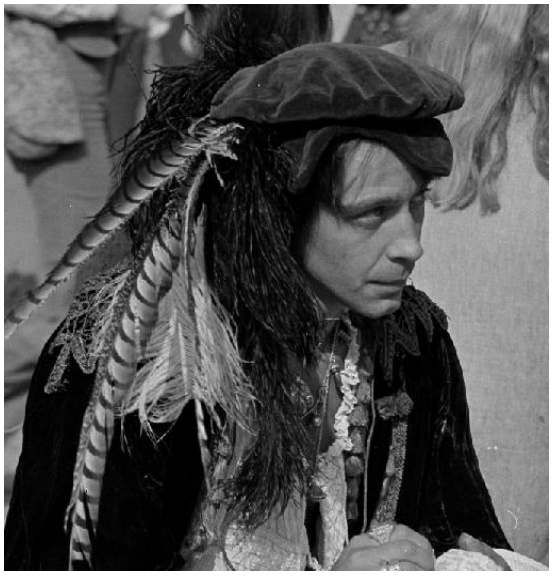

Figure 52: Original Image

Next, we half the thresholds of bands and set as LL = 98, LH = 8, HL = 10, HH = 4.5, LH1=3, HL1=4, HH1=1.75 and we can observe significant image degradation so we further have to decrease threshold values.

Figure 53: Reconstructed Image

Figure 54: Original Image

Next, we half the thresholds of bands and set as LL = 49, LH = 4, HL = 5, HH = 2, LH1=1.5, HL1=2, HH1=0.87 and we can observe significant image degradation so we further have to decrease threshold values.

Figure 55: Reconstructed Image

Figure 56: Original Image

Next, we half the thresholds of bands and set as LL = 25, LH = 2, HL = 2.5, HH = 1, LH1=0.75, HL1=1, HH1=0.4 and we can observe significant image degradation so we further have to decrease threshold values.

Figure 57: Reconstructed Image

Figure 58: Original Image

Next, we half the thresholds of bands and set as LL = 13, LH = 1, HL = 1.25, HH = 0.5, LH1=0.37, HL1=0.5, HH1=0.2 and we can observe significant image degradation so we further have to decrease threshold values.

Figure 59: Reconstructed Image

Figure 60: Original Image

Next, we half the thresholds of bands and set as LL = 6, LH = 0.5, HL = 0.625, HH = 0.25, LH1=0.2, HL1=0.25, HH1=0.1 and we can observe significant image degradation so we further have to decrease threshold values.

Figure 61: Reconstructed Image

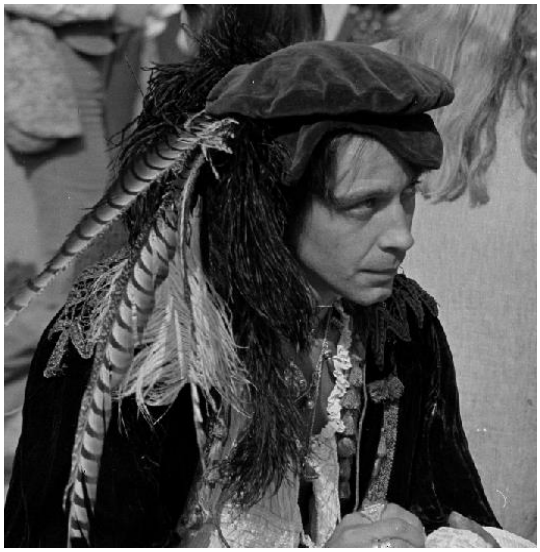

Figure 62: Original Image

Next, we can see that average energy of some sub bands are very low such as HH, LH1, HL1, HH1 sub bands have very low average energy so we can try pruning these bands completely and vary threshold of other bands. In the above figure I have pruned sub bands HH, LH1, HL1, HH1 completely and set threshold of sub band LL = 200 and we can observe significant image degradation so we further have to decrease threshold values.

Figure 63: Reconstructed Image

Figure 64: Original Image

Next, we can see that average energy of some sub bands are very low such as HH, LH1, HL1, HH1 sub bands have very low average energy so we can try pruning these bands completely and vary threshold of other bands. In the above figure I have pruned sub bands HH, LH1, HL1, HH1 completely and set threshold of sub band LL = 100 and we can observe significant image degradation so we further have to decrease threshold values.

Figure 65: Reconstructed Image

Figure 66: Original Image

Next, we can see that average energy of some sub bands are very low such as HH, LH1, HL1, HH1 sub bands have very low average energy so we can try pruning these bands completely and vary threshold of other bands. In the above figure I have pruned sub bands HH, LH1, HL1, HH1 completely and set threshold of sub band LL = 50 and we can observe not much image degradation.

Figure 67: Reconstructed Image

Figure 68: Original Image

Next, we can see that average energy of some sub bands are very low such as HH, LH1, HL1, HH1 sub bands have very low average energy so we can try pruning these bands completely and vary threshold of other bands. In the above figure I have pruned sub bands HH, LH1, HL1, HH1 completely and set threshold of sub band LL = 25 and we can observe not much image degradation.

Figure 69: Reconstructed Image

Figure 70: Original Image

Next, we can see that average energy of some sub bands are very low such as HH, LH1, HL1, HH1 sub bands have very low average energy so we can try pruning these bands completely and vary threshold of other bands. In the above figure I have pruned sub bands HH, LH1, HL1, HH1 completely and set threshold of sub band LL = 10 and we can observe not much image degradation.

CONCLUSION

We varied the threshold of sub-bands until there is no significant image degradation for sub-bands LL, LH, HL, HH, LH1, HL1, HH1 respectively. We start by taking the initial threshold as the average energy of sub-bands. In figure 47,49,51,53,55,57,59 respectively we varied the threshold for sub-band LL, LH, HL, HH, LH1, HL1, HH1 and we observe significant image degradation. In figure 61,63 respectively we varied the threshold of sub-band LL and completely pruned sub-band HH, LH1, HL1, HH1 we observe significant image degradation so we further decrease the threshold values. In figure 65,67,69 respectively we further decrease the threshold value for sub-band LL and observe no significant image degradation. Further size of compressed image 65,67,69 is 166kb,169kb,173kb respectively.

VARYING THRESOLDS FOR LEVEL OF DECOMPOSITION = 3

AVERAGE ENERGY OF SUBBANDS: LL = 798, LH = 76, HL =98, HH = 46, LH1 = 31, HL1 = 41, HH1 = 18, LH2 = 12.58, HL2 = 15.89, HH2 = 7.09

We set initial thresholds as the average energy of sub bands and then varies value of thresholds until there is no significant image degradation.

Figure 71: Reconstructed Image

Figure 72: Original Image

First, we set thresholds of bands as LL = 798, LH = 76, HL = 98, HH = 46, LH1=31, HL1=41, HH1=18, LH2=12, HL2=16, HH2=7 and we can observe significant image degradation so we now decrease threshold values.

Figure 73: Reconstructed Image

Figure 74: Original Image

First, we set thresholds of bands as LL = 400, LH = 38, HL = 49, HH = 23, LH1=15, HL1=20, HH1=9, LH2=6, HL2=8, HH2=3.5 and we can observe significant image degradation so we now decrease threshold values.

Figure 75: Reconstructed Image

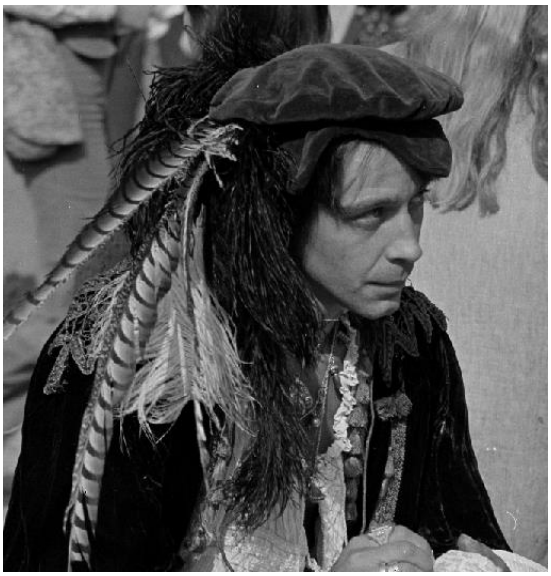

Figure 76: Original Image

First, we set thresholds of bands as LL = 200, LH = 19, HL = 25, HH = 12, LH1=8, HL1=10, HH1=4.5, LH2=3, HL2=4, HH2=3.5 and we can observe significant image degradation so we now decrease threshold values.

Figure 77: Reconstructed Image

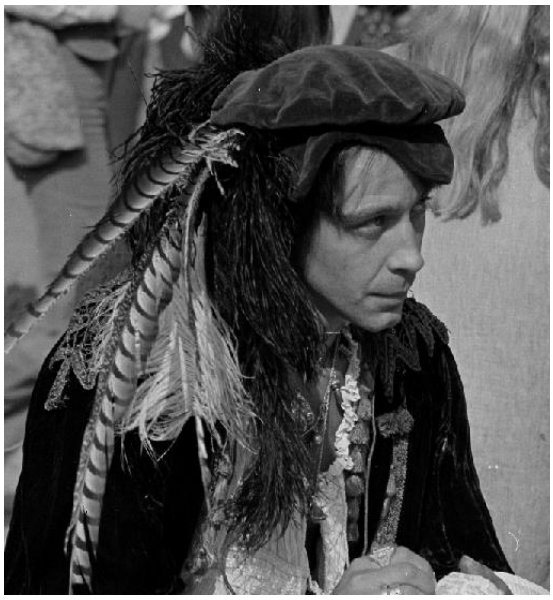

Figure 78: Original Image

First, we set thresholds of bands as LL = 100, LH = 10, HL = 12.5, HH = 6, LH1=4, HL1=5, HH1=2.25, LH2=1.5, HL2=2, HH2=2 and we can observe significant image degradation so we now decrease threshold values.

.

Figure 79: Reconstructed Image

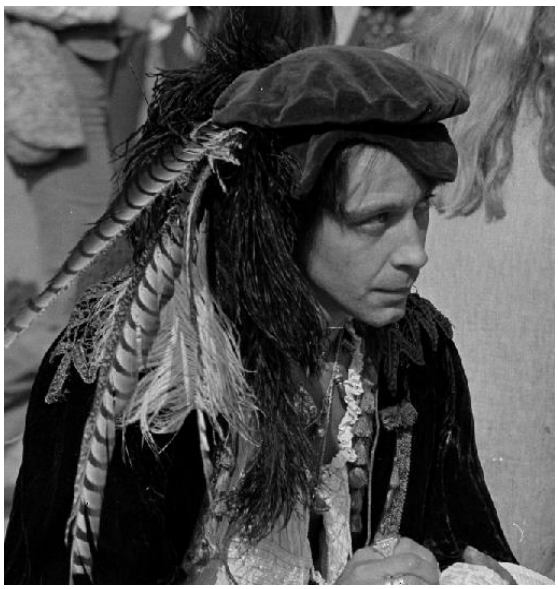

Figure 80: Original Image

First, we set thresholds of bands as LL = 50, LH = 5, HL = 6, HH = 3, LH1=2, HL1=2.5, HH1=1.25, LH2=0.75, HL2=1, HH2=1 and we can observe significant image degradation so we now decrease threshold values.

Figure 81: Reconstructed Image

Figure 82: Original Image

First, we set thresholds of bands as LL = 25, LH = 2.5, HL = 3, HH = 1.5, LH1=1, HL1=1.25, HH1=0.62, LH2=0.37, HL2=0.5, HH2=0.5 and we can observe significant image degradation so we now decrease threshold values.

Figure 83: Reconstructed Image

Figure 84: Original Image

Next, we can see that average energy of some sub bands are very low such as HH1, LH2, HL2, HH2 sub bands have very low average energy so we can try pruning these bands completely and vary threshold of other bands. In the above figure I have pruned sub bands HH1, LH2, HL2, HH2 completely and set threshold of sub band LL = 50 and we can observe not much image degradation.

Figure 85: Reconstructed Image

Figure 86: Original Image

Next, we can see that average energy of some sub bands are very low such as HH1, LH2, HL2, HH2 sub bands have very low average energy so we can try pruning these bands completely and vary threshold of other bands. In the above figure I have pruned sub bands HH1, LH2, HL2, HH2 completely and set threshold of sub band LL = 30 and we can observe not much image degradation.

Figure 87: Reconstructed Image

Figure 88: Original Image

Next, we can see that average energy of some sub bands are very low such as HH1, LH2, HL2, HH2 sub bands have very low average energy so we can try pruning these bands completely and vary threshold of other bands. In the above figure I have pruned sub bands HH1, LH2, HL2, HH2 completely and set threshold of sub band LL = 100 and we can observe not much image degradation.

CONCLUSION

We varied the threshold of sub-bands until there is no significant image degradation for sub-bands LL, LH, HL, HH, LH1, HL1, HH1, LH2, HL2, HH2 respectively. We start by taking the initial threshold as the average energy of sub-bands. In figure 71,73,75,77,79,81 respectively we varied threshold for sub-band LL, LH, HL, HH, LH1, HL1, HH1, LH2, HL2, HH2 and we observe significant image degradation. In figure 83,85,87 respectively we varied the threshold of sub-band LL and completely pruned sub band HH1, LH2, HL2, HH2 we observe no significant image degradation. Further size of compressed image 83,85,87 is 171kb,170kb,168kb respectively.

VARYING NUMBER OF COFFICENTS IN SUBBANDS KEEPING LEVEL OF DECOMPOSITION = 1

We have taken input image of 512x512 pixels. By taking level of decomposition = 1 we will get four sub bands each of size 256x256 pixels. (256*265 = 65536 coefficients in each sub band). Now we will vary the number of coefficients retained in each sub band until the image degradation is perceptually significant.

SUB BANDS = LL, LH, HL, HH

Figure 89: Reconstructed Image

Figure 90: Original Image

First, we retained 20000 coefficients of sub band HH out of 65536 coefficients and we can observe that there is no significant image degradation.

Figure 91: Reconstructed Image

Figure 92: Original Image

Next, we retained 20000 coefficients of sub band HL out of 65536 coefficients and prune all the coefficients of sub band HH and we can observe that there is no significant image degradation.

Figure 93: Reconstructed Image

Figure 94: Original Image

Next, we retained 10000 coefficients of sub band HL out of 65536 coefficients and prune all the coefficients of sub band HH and we can observe that there is no significant image degradation.

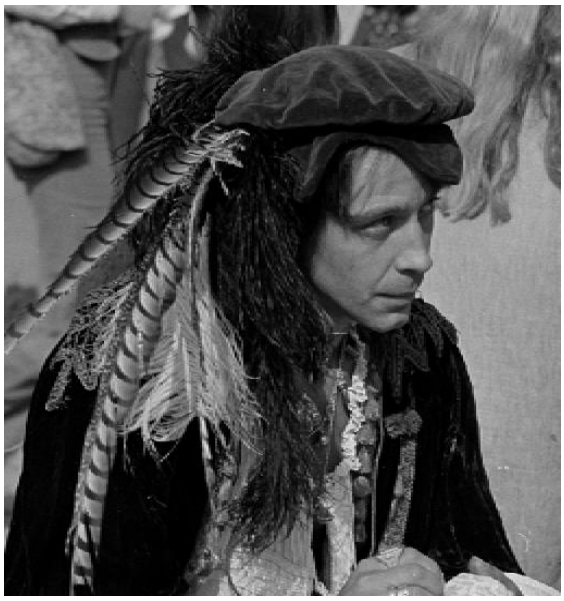

Figure 95: Reconstructed Image

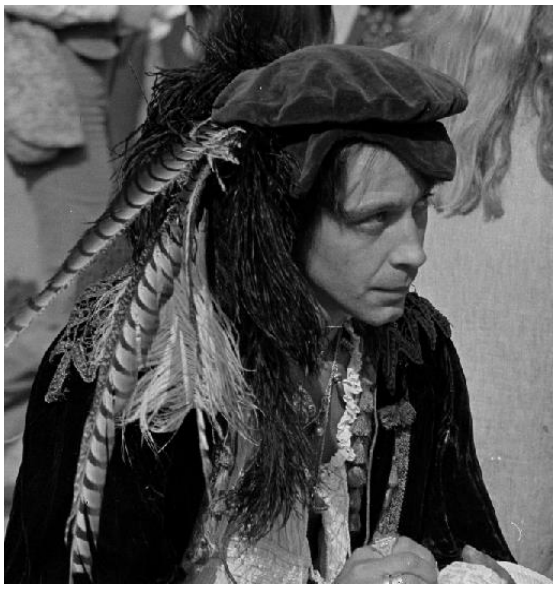

Figure 96: Original Image

Next, we prune all the coefficients of sub band HH, HL and we can observe that there is no significant image degradation.

Figure 97: Reconstructed Image

Figure 98: Original Image

Next, we retained 40000 coefficients of sub band LH out of 65536 coefficients and prune all the coefficients of sub band HH, HL and we can observe that there is no significant image degradation.

Figure 99: Reconstructed Image

Figure 100: Original Image

Next, we retained 20000 coefficients of sub band LH out of 65536 coefficients and prune all the coefficients of sub band HH, HL and we can observe that there is no significant image degradation.

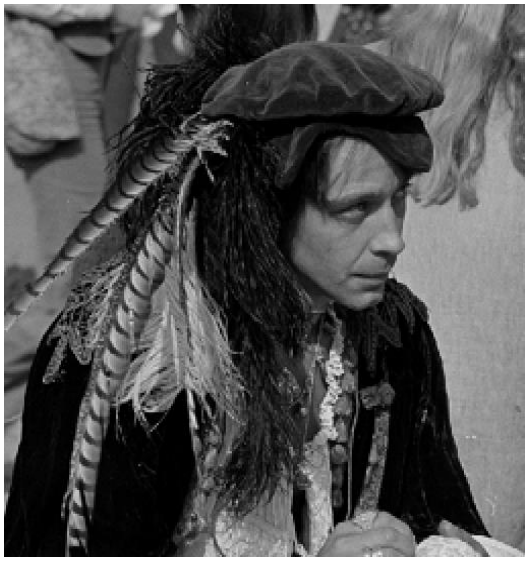

Figure 101: Reconstructed Image

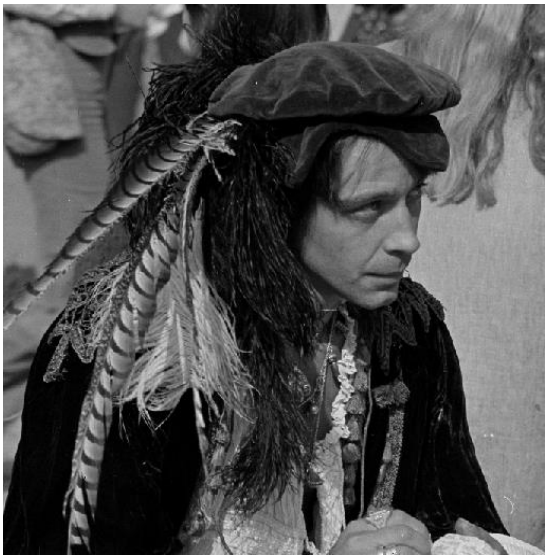

Figure 102: Original Image

Next, we prune all the coefficients of sub band HH, HL, LH and we can observe that there is no significant image degradation.

Figure 103: Reconstructed Image

Figure 104: Original Image

Next, we retained 40000 coefficients of sub band LL out of 65536 coefficients and prune all the coefficients of sub band HH, HL, LH and we can observe that there is significant image degradation.

Figure 105: Reconstructed Image

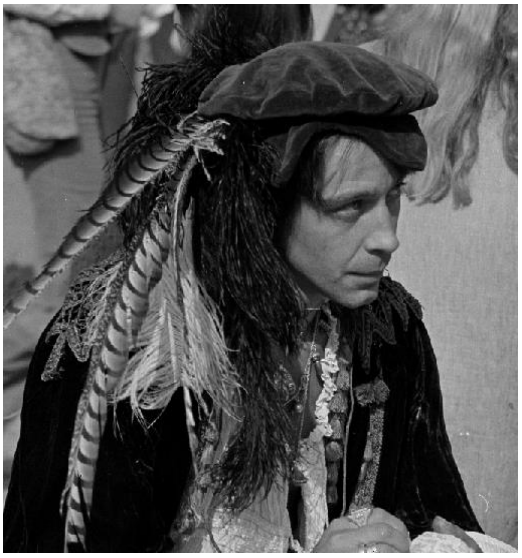

Figure 106: Original Image

Next, we retained 50000 coefficients of sub band LL out of 65536 coefficients and prune all the coefficients of sub band HH, HL, LH and we can observe that there is significant image degradation.

Figure 107: Reconstructed Image

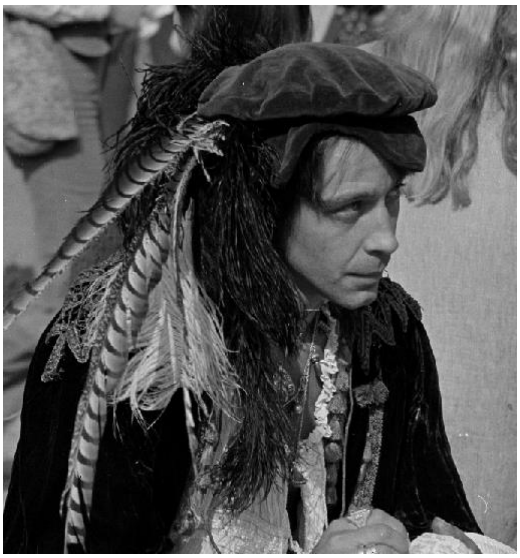

Figure 108: Original Image

Next, we retained 60000 coefficients of sub band LL out of 65536 coefficients and prune all the coefficients of sub band HH, HL, LH and we can observe that there is significant image degradation.

Figure 109: Reconstructed Image

Figure 110: Original Image

Next, we retained 62000 coefficients of sub band LL out of 65536 coefficients and prune all the coefficients of sub band HH, HL, LH and we can observe that there is significant image degradation.

Figure 111: Reconstructed Image

Figure 112: Original Image

Next, we retained 63000 coefficients of sub band LL out of 65536 coefficients and prune all the coefficients of sub band HH, HL, LH and we can observe that there is significant image degradation.

Figure 113: Reconstructed Image

Figure 114: Original Image

Next, we retained 64000 coefficients of sub band LL out of 65536 coefficients and prune all the coefficients of sub band HH, HL, LH and we can observe that there is significant image degradation.

Figure 115: Reconstructed Image

Figure 116: Original Image

Next, we retained 65000 coefficients of sub band LL out of 65536 coefficients and prune all the coefficients of sub band HH, HL, LH and we can observe that there is no significant image degradation.

VARIATION OF SIZE OF COMPRESSED IMAGE AND NUMBER OF COFF. RETAINED IN THE LL SUBBAND SETTING OTHER SUB BAND TO ZERO MATRIX

NO. OF COFF. RETAINED	SIZE OF ORIGINAL IMAGE	SIZE OF COMPRESSED IMAGE
40000	292KB	109KB
50000	292KB	135KB
60000	292KB	158KB
62000	292KB	163KB
63000	292KB	165KB
64000	292KB	167KB
65000	292KB	170KB

Figure 117: size of compressed image variation with the threshold of LL sub band

CONCLUSION

We varied a number of coefficients in the sub-bands HH, HL in figures 89,91,93 respectively and we observe no significant image degradation. So, we are pruning matrix HH, HL completely in figure 95 and we observe no significant image degradation. Next, we varied the number of coefficients in sub-band LH in figures 97,99 and we observe no significant image degradation.in figure 101. We tried to prune sub band LH completely and we observe no significant image degradation. So now we varied the number of coefficients in sub-band LL by pruning sub-band LH, HL, HH completely. We can observe how the image quality in figure 103,105,107,109,111,113,115 respectively varies with the number of coefficients retained in sub-band LL. Figure 117 shows the variation of the number of coefficients retained in sub-band LL and the size of the compressed image. We can conclude that by retaining 65000 coefficients in sub-band LL we get no significant image degradation and size of image compresses from 292 Kb to 170 Kb.