Information Theory Project

Rishabh Tripathi - 180030036

S U Swakath - 180020036

Huffman Coding and JPEG

In this project we first implement the Huffman Coding technique on a symbol sequence.

Further we implement different methods to compress an image, finally to arrive at the JPEG encoding technique.

Huffman Coding

- Here we load a **bitmap image (BMP)** into a matrix.
- We observe the dimensions of the loaded matrix and display the RGB components of the image.
- To demonstrate Huffman Coding we take the red component of the image and flatten the matrix. It has 256 unique intensity levels occuring with different probabilities. We use this to perform Huffman Coding on the flattened sequence.

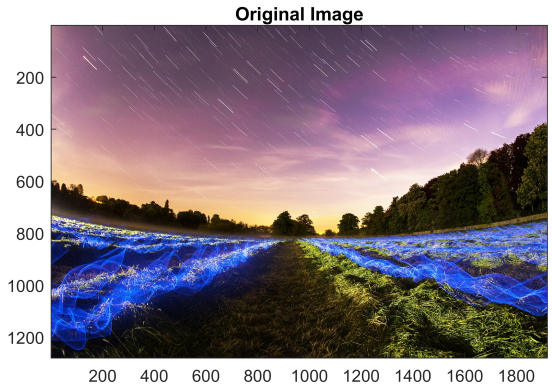
```
% Loading the BMP file
image_bmp = imread('bitmap_sample.bmp');
```

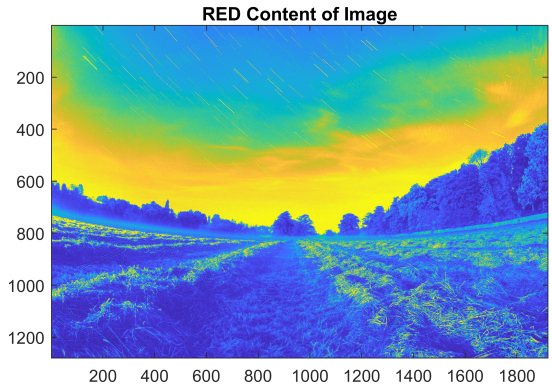
```
% Display the uncoded image
imshow(image_bmp)
```

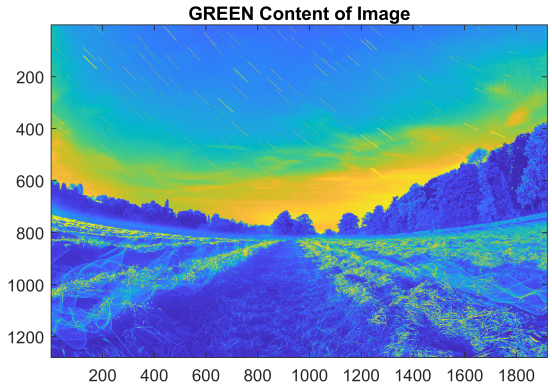
```
% Display the size of the image matrix
size(image_bmp)
```

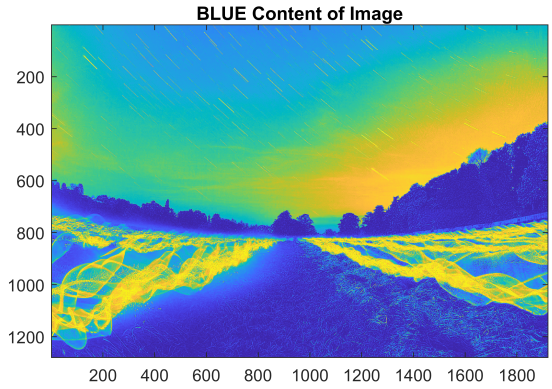
```
ans = 1×3
        1280        1920           3
```

```
% RGB Representation of Image
figure(1)
subplot(2,2,1);
image(image_bmp);
title('Original Image');
% RED content of the image
subplot(2,2,2);
image(image_bmp(:,:,1));
title('RED Content of Image');
% GREEN content of the image
subplot(2,2,3);
image(image_bmp(:,:,2));
title('GREEN Content of Image');
% BLUE content of the image
subplot(2,2,4);
image(image_bmp(:,:,3));
title('BLUE Content of Image');
```







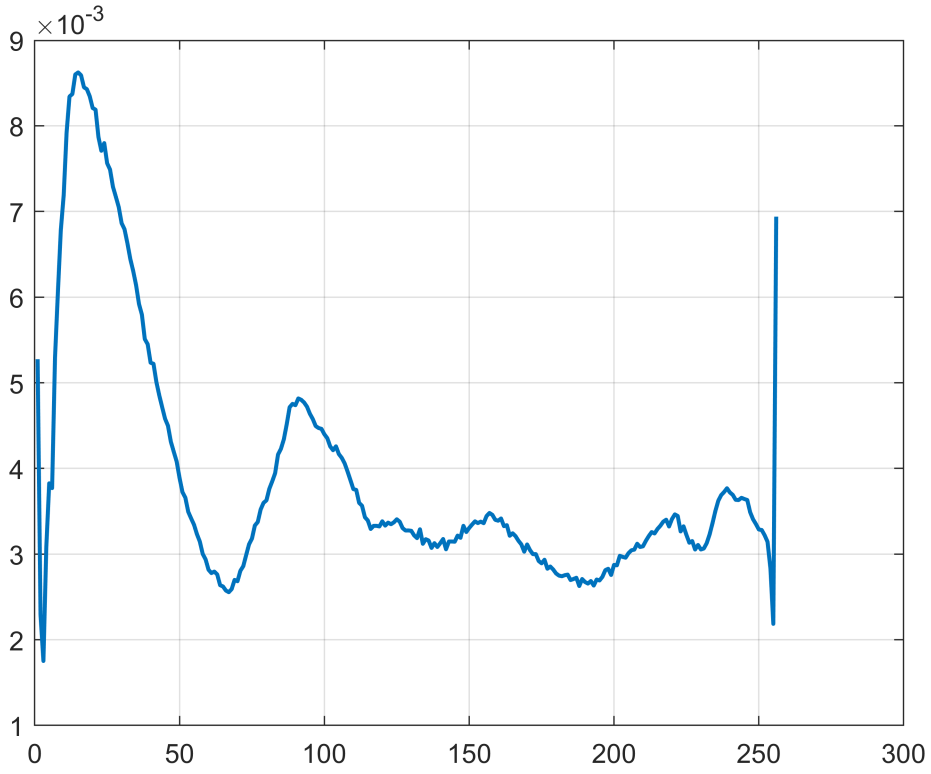



```
% Demonstration of Huffman Coding

red_content = image_bmp(:,:,1);
% The size of the RED content matrix
whos("red_content")
```

```
  Name                Size                Bytes  Class    Attributes

  red_content      1280x1920              2457600  uint8
```

```
red_content_flat = red_content(:);
intensity_levels = 0:1:255;

% Obtain the frequency of each possible intensity level between 0 to 255
freq_count = histcounts(red_content_flat, [intensity_levels 256]);

% Compute the probability of occurrence of each intensity level
prob_levels = freq_count / numel(red_content_flat);

% The probabilty distribution is showed as follows
figure(2)
plot(prob_levels, 'LineWidth',1.5)
```

```
grid on
```




```
% Perform Huffman encoding
%huff_dict = huffmandict(intensity_levels, prob_levels);
huff_code = huffman(prob_levels);
huff_dict = toCell(huff_code)
```

huff_dict = 256×2 cell

	1	2
1	0	[0,1,1,0...
2	1	[0,1,0,1...
3	2	[0,0,1,0...
4	3	[1,0,0,0...
5	4	[0,0,0,1...
6	5	[0,0,0,1...
7	6	[0,1,1,0...
8	7	[0,1,1,1...
9	8	[1,1,0,1...
10	9	[1,1,1,1...
11	10	[0,0,1,0...
12	11	[0,0,1,1...

	1	2
13	12	[0,0,1,1...
14	13	[0,1,0,0...
15	14	[0,1,0,0...
16	15	[0,1,0,0...
17	16	[0,0,1,1...
18	17	[0,0,1,1...
19	18	[0,0,1,1...
20	19	[0,0,1,0...
21	20	[0,0,1,0...
22	21	[0,0,1,0...
23	22	[0,0,0,1...
24	23	[0,0,0,1...
25	24	[0,0,0,1...
26	25	[0,0,0,1...
27	26	[0,0,0,0...
28	27	[1,1,1,1...
29	28	[1,1,1,1...
30	29	[1,1,1,0...
31	30	[1,1,1,0...
32	31	[1,1,0,0...
33	32	[1,0,1,0...
34	33	[1,0,0,1...
35	34	[1,0,0,0...
36	35	[0,1,1,1...
37	36	[0,1,1,1...
38	37	[0,1,1,0...
39	38	[0,1,1,0...
40	39	[0,1,0,1...
41	40	[0,1,0,1...
42	41	[0,1,0,1...
43	42	[0,1,0,1...
44	43	[0,1,0,1...
45	44	[0,1,0,1...

	1	2
46	45	[0,1,0,0...
47	46	[0,1,0,0...
48	47	[0,0,1,1...
49	48	[0,0,1,0...
50	49	[0,0,1,0...
51	50	[0,0,0,0...
52	51	[0,0,0,0...
53	52	[1,1,1,1...
54	53	[1,1,1,0...
55	54	[1,1,0,0...
56	55	[1,0,1,1...
57	56	[1,0,0,1...
58	57	[0,1,1,1...
59	58	[0,1,1,1...
60	59	[0,1,1,1...
61	60	[0,1,1,0...
62	61	[0,1,1,1...
63	62	[0,1,1,0...
64	63	[0,1,1,0...
65	64	[0,1,1,0...
66	65	[0,1,0,1...
67	66	[0,1,0,1...
68	67	[0,1,0,1...
69	68	[0,1,1,0...
70	69	[0,1,1,0...
71	70	[0,1,1,1...
72	71	[0,1,1,1...
73	72	[0,1,1,1...
74	73	[1,0,0,1...
75	74	[1,0,1,0...
76	75	[1,1,0,0...
77	76	[1,1,0,1...
78	77	[1,1,1,1...

	1	2
79	78	[0,0,0,0...
80	79	[0,0,0,0...
81	80	[0,0,0,1...
82	81	[0,0,0,1...
83	82	[0,0,1,0...
84	83	[0,0,1,0...
85	84	[0,0,1,1...
86	85	[0,1,0,0...
87	86	[0,1,0,0...
88	87	[0,1,0,1...
89	88	[0,1,0,1...
90	89	[0,1,0,1...
91	90	[0,1,0,1...
92	91	[0,1,0,1...
93	92	[0,1,0,1...
94	93	[0,1,0,1...
95	94	[0,1,0,1...
96	95	[0,1,0,0...
97	96	[0,1,0,0...
98	97	[0,1,0,0...
99	98	[0,1,0,0...
100	99	[0,1,0,0...

⋮

```
red_encoded = huffmanenco(red_content_flat, huff_dict);

% Decoding and error check
red_decoded = huffmandeco(red_encoded, huff_dict);

isequal(red_decoded, red_content_flat)
```

```
ans = logical
   1
```

Comparison

Size of the original sequence with every number stored as **uint8:**

```
size(red_content_flat)
```

```
ans = 1×2
     2457600           1
```

Memory size of each of the above number is **8 bits.** Therefore in total we have **2457600 bytes = 2.3438 MBs** of data.

Size of the encoded sequence with every number stored as a **bit:**

```
size(red_encoded)
```

```
ans = 1×2
    19519478           1
```

Memory size of each of the above number is **1 bit.** Therefore in total we have **19519478 bits = 2.3269 MBs** of data.

As demonstrated above, this compression is **lossless** and is much more effective if the probabilty distribution in the data is more skewed.

JPEG Encoding

- Now we focus on encoding the loaded bitmap image according to the JPEG standards.
- Following are the steps for compressing an image according to JPEG standards:
- Divide the image into square matrices of **8x8** each.
- Perform DCT of the obtained 8x8 matrix and quantize the values obtained.
- JPEG is a lossy compression as we round off values in order to quantize them.

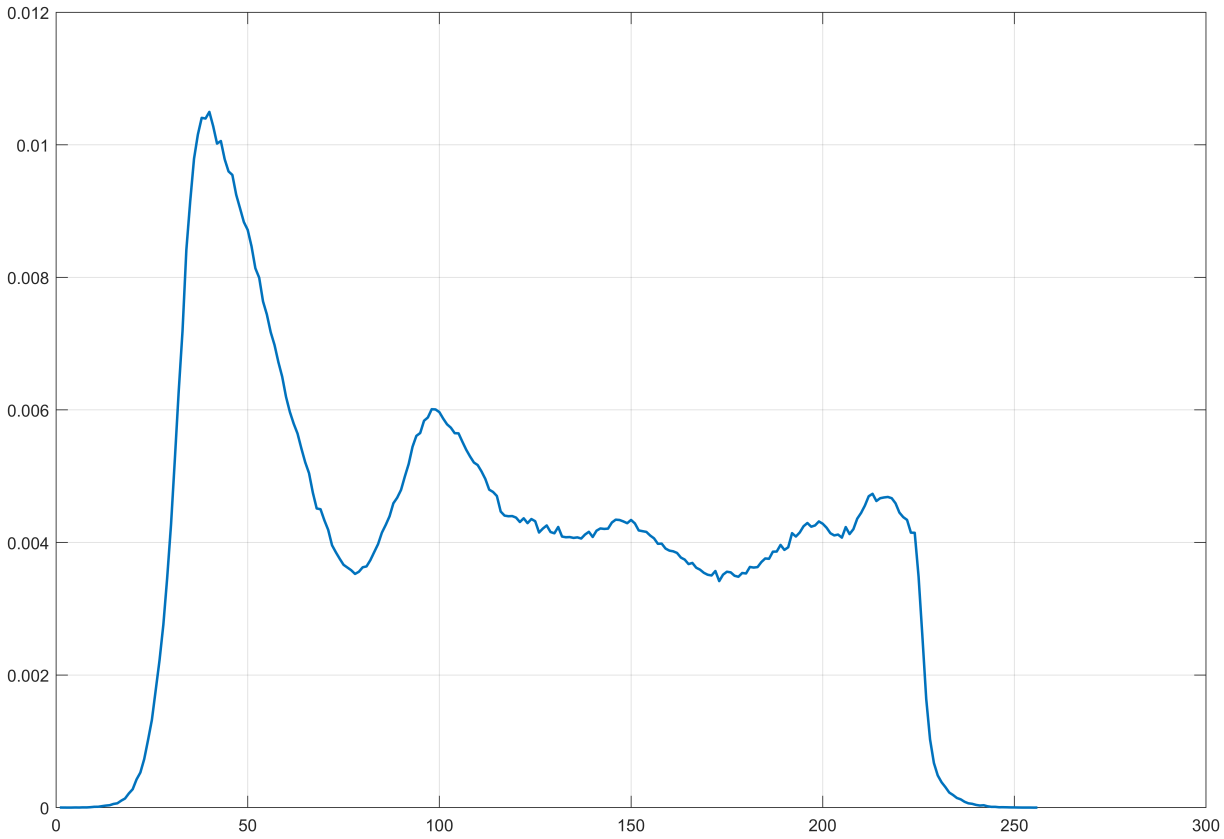
```
%I = imread('bitmap_sample.bmp');
I = red_content;
I1=I;

[row,coln]=size(I);      %storing size of image

rem1 = rem(row,8);
row=row-rem1;

rem2 = rem(coln,8);
coln= coln-rem2;

I = imresize(I,[row coln]);
I= double(I);

% Subtracting each image pixel value by 128
%I = I - (128*ones(row, coln));
I = I - 128;
quality = input('What quality of compression you require - ');

% Quality Matrix Formulation
Q50 = [ 16 11 10 16 24 40 51 61;
     12 12 14 19 26 58 60 55;
```

```
     14 13 16 24 40 57 69 56;
     14 17 22 29 51 87 80 62;
     18 22 37 56 68 109 103 77;
     24 35 55 64 81 104 113 92;
     49 64 78 87 103 121 120 101;
     72 92 95 98 112 100 103 99];

if quality > 50
    QX = round(Q50.*(ones(8)*((100-quality)/50)));
    QX = uint8(QX);
elseif quality < 50
    QX = round(Q50.*(ones(8)*(50/quality)));
    QX = uint8(QX);
elseif quality == 50
    QX = Q50;
end


% Formulation of forward DCT Matrix and inverse DCT matrix
DCT_matrix8 = dct(eye(8));
iDCT_matrix8 = DCT_matrix8';   %inv(DCT_matrix8);

% Jpeg Compression
dct_restored = zeros(row,coln);
QX = double(QX);

% Jpeg Encoding

% Forward Discret Cosine Transform
for i1=[1:8:row]
    for i2=[1:8:coln]
        zBLOCK=I(i1:i1+7,i2:i2+7);
        win1=DCT_matrix8*zBLOCK*iDCT_matrix8;
        dct_domain(i1:i1+7,i2:i2+7)=win1;
    end
end

% Quantization of the DCT coefficients
for i1=[1:8:row]
    for i2=[1:8:coln]
        win1 = dct_domain(i1:i1+7,i2:i2+7);
        win2=round(win1./QX);
        dct_quantized(i1:i1+7,i2:i2+7)=win2;
    end
end

compresed_img=dct_quantized;

% Jpeg Decoding

% Dequantization of DCT Coefficients
for i1=[1:8:row]
    for i2=[1:8:coln]
        win2 = dct_quantized(i1:i1+7,i2:i2+7);
```

```
        win3 = win2.*QX;
        dct_dequantized(i1:i1+7,i2:i2+7) = win3;
    end
end


% Inverse DISCRETE COSINE TRANSFORM
for i1=[1:8:row]
    for i2=[1:8:coln]
        win3 = dct_dequantized(i1:i1+7,i2:i2+7);
        win4=iDCT_matrix8*win3*DCT_matrix8;
        dct_restored(i1:i1+7,i2:i2+7)=win4;
    end
end
I2=dct_restored;

% Conversion of Image Matrix to Intensity image
K=mat2gray(I2);
%Display of Results
%figure(1);imshow(I1);title('original image');
figure(2);imshow(compresed_img);title('Dct Points');
```


```
figure(3);imshow(K);title('Compressed Image');
```


```
K_uint8 = uint8(255 * K);
K_flat = K_uint8(:);
% Obtain the frequency of each possible intensity level between 0 to 255
freq_count = histcounts(K_uint8, [intensity_levels 256]);

% Compute the probability of occurrence of each intensity level
prob_K = freq_count / numel(K_uint8);

% The probabilty distribution is showed as follows
plot(prob_K, 'LineWidth',1.5)
grid on
```




```
huff_code_k = huffman(prob_K);
huff_dict_k = toCell(huff_code_k)
```

huff_dict_k = 256×2 cell

	1	2
1	0	1×19 double
2	1	1×20 double
3	2	1×22 double
4	3	1×21 double
5	4	1×18 double
6	5	1×19 double
7	6	1×18 double
8	7	1×18 double
9	8	1×17 double
10	9	1×16 double
11	10	1×16 double
12	11	1×15 double

	1	2
13	12	1×15 double
14	13	1×15 double
15	14	1×14 double
16	15	1×14 double
17	16	1×13 double
18	17	1×13 double
19	18	1×12 double
20	19	1×12 double
21	20	1×11 double
22	21	1×11 double
23	22	[1,0,1,1...
24	23	[1,1,1,0...
25	24	[1,0,1,1...
26	25	[1,0,1,1...
27	26	[0,1,0,1...
28	27	[1,0,0,0...
29	28	[1,0,1,1...
30	29	[0,0,1,0...
31	30	[1,0,0,0...
32	31	[1,0,1,1...
33	32	[1,1,0,1...
34	33	[0,0,0,1...
35	34	[0,1,0,1...
36	35	[0,1,1,0...
37	36	[0,1,1,1...
38	37	[0,1,1,1...
39	38	[0,1,1,1...
40	39	[1,0,0,0...
41	40	[0,1,1,1...
42	41	[0,1,1,0...
43	42	[0,1,1,1...
44	43	[0,1,1,0...
45	44	[0,1,1,0...

	1	2
46	45	[0,1,1,0...
47	46	[0,1,0,1...
48	47	[0,1,0,0...
49	48	[0,1,0,0...
50	49	[0,0,1,1...
51	50	[0,0,1,0...
52	51	[1,1,1,1...
53	52	[1,1,1,1...
54	53	[1,1,1,0...
55	54	[1,1,1,0...
56	55	[1,1,0,1...
57	56	[1,1,0,0...
58	57	[1,0,1,1...
59	58	[1,0,1,1...
60	59	[1,0,1,1...
61	60	[1,0,1,0...
62	61	[1,0,1,0...
63	62	[1,0,0,1...
64	63	[1,0,0,0...
65	64	[1,0,0,0...
66	65	[0,1,1,1...
67	66	[0,1,1,0...
68	67	[0,1,0,0...
69	68	[0,1,0,0...
70	69	[0,0,1,1...
71	70	[0,0,0,1...
72	71	[1,1,1,1...
73	72	[1,1,1,0...
74	73	[1,1,1,0...
75	74	[1,1,0,1...
76	75	[1,1,0,1...
77	76	[1,1,0,0...
78	77	[1,1,0,0...

	1	2
79	78	[1,1,0,0...
80	79	[1,1,0,1...
81	80	[1,1,0,1...
82	81	[1,1,0,1...
83	82	[1,1,1,0...
84	83	[1,1,1,1...
85	84	[0,0,0,1...
86	85	[0,0,1,0...
87	86	[0,1,0,0...
88	87	[0,1,0,1...
89	88	[0,1,0,1...
90	89	[0,1,1,0...
91	90	[0,1,1,1...
92	91	[0,1,1,1...
93	92	[1,0,0,1...
94	93	[1,0,0,1...
95	94	[1,0,0,1...
96	95	[1,0,1,0...
97	96	[1,0,1,0...
98	97	[1,0,1,1...
99	98	[1,0,1,0...
100	99	[1,0,1,0...

⋮

```
red_encoded_k = huffmanenco(K_flat, huff_dict_k);

% Decoding and error check
red_decoded_k = huffmandeco(red_encoded_k, huff_dict_k);

isequal(red_decoded_k, K_flat)
```

```
ans = logical
   1
```

Comparison

Size of the original sequence with every number stored as **uint8:**

```
size(red_content_flat)
```

```
ans = 1×2
     2457600           1
```

Memory size of each of the above number is **8 bits.** Therefore in total we have **2457600 bytes = 2.3438 MBs** of data.

Size of the encoded sequence with every number stored as a **bit:**

```
size(red_encoded_k)
```

```
ans = 1×2
    18790460           1
```

Memory size of each of the above number is **1 bit.** Therefore in total we have **18790460 bits = 2.24 MBs** of data.

Functions

```
function [code,compression]=huffman(p)
    p=p(:)/sum(p);    %normalises probabilities
    c=huff5(p);     %calling function for creating huffmann tree
    code=char(getcodes(c,length(p)));   %generate huffmann code
    compression=ceil(log(length(p))/log(2))/ (sum(code' ~= ' ')*p); %compression rate
end

% Create Huffmann tree
% Input : p : probability (or number of occurences) of each alphabet symbol
% Output: c :huffmann tree
function c=huff5(p)
    % Simulates a tree structure using a nested cell structure
    c = cell(length(p),1);    % Generate cell structure
    for i=1:length(p)       % fill cell structure with 1,2,3...n
       c{i}=i;        %(n=number of symbols in alphabet)
    end
    % disp(c)
    while size(c)-2        % Repeat till only two branches
        [p,i]=sort(p);    % Sort to ascending probabilities
        c=c(i);            % Reorder tree.
        c{2}={c{1},c{2}};c(1)=[];    % join branch 1 to 2 and prune 1
        p(2)=p(1)+p(2);p(1)=[];        % Merge Probabilities
    end
end

% Generate Code
% Input: a is the nested cell structure created by huffcode5
% Output: n is the number of symbols
function y= getcodes(a,n)
    global y
    y=cell(n,1);
    getcodes2(a,[]) %pull out codes
end
```

```
% Using Recursion to pull out codes
function getcodes2(a,dum)
global y
if isa(a,'cell')
         getcodes2(a{1},[dum 0]);
         getcodes2(a{2},[dum 1]);
else
   y{a}=setstr(48+dum);
end
end


function huff_dict = toCell(codes)
huff_dict = cell(256,2);

for i = 0:255
    curCode = codes(i+1,:);
    codeArr = codeArray(curCode);
    huff_dict(i+1,1) = {[i]};
    huff_dict(i+1,2) = {codeArr};
end
end


function arr = codeArray(code)
    len = numel(code);
    arr = [];
    for i = 1:len
        cur = code(i);
        if cur == '0' ||cur == '1'
            arr = [arr str2num(cur)];
        end
    end
end
```