Advanced Multimedia Applications

Assignment - 2011

Group Members:

Y6356585
Y6356306
Y6167785

Contents

1 Introduction

The following design is a video compression and decompression system based on JPEG/MPEG-2 compression standard(without motion estimation). The test image is first subjected to Discrete Cosine transform in two dimension for spectrum analysis. The resulting image is quantised to remove unwanted high frequency components this is basically the first step towards image compression. The resulting AC-Components are subjected to block wise(8x8) zigzag scaned thus sorting the high energy components in a decreasing order with high energy at the top of each block. The DC component of every block is DPCM coded.

The zigzag encoded AC coefficients and the differential coded DC coefficients are then subjected to RunLength Coding and Huffman coding to produce the final compressed image, this is the second compression stage.

For image reconstruction, the same procedure is employed but in the reverse direction. The compressed image is subjected to Huffman de-coding, RunLength de-coding, Reverse zigzag, DPCM decoding, Inverse quantisation and inverse DCT to reconstruct the original image.

Sections 2-6 focus on each sub-component of theentire system. Section 7 of the report contains the overall results obtained by performing the encoding/decoding on 6 different images. Testing is perfoemed on smooth low frequency images as well as edgy/sharp high frequency images. The results are critically analysed by using graphs.

The Matlab source code for all test scripts and functions used in the sub-systems can be found in the appendix section.

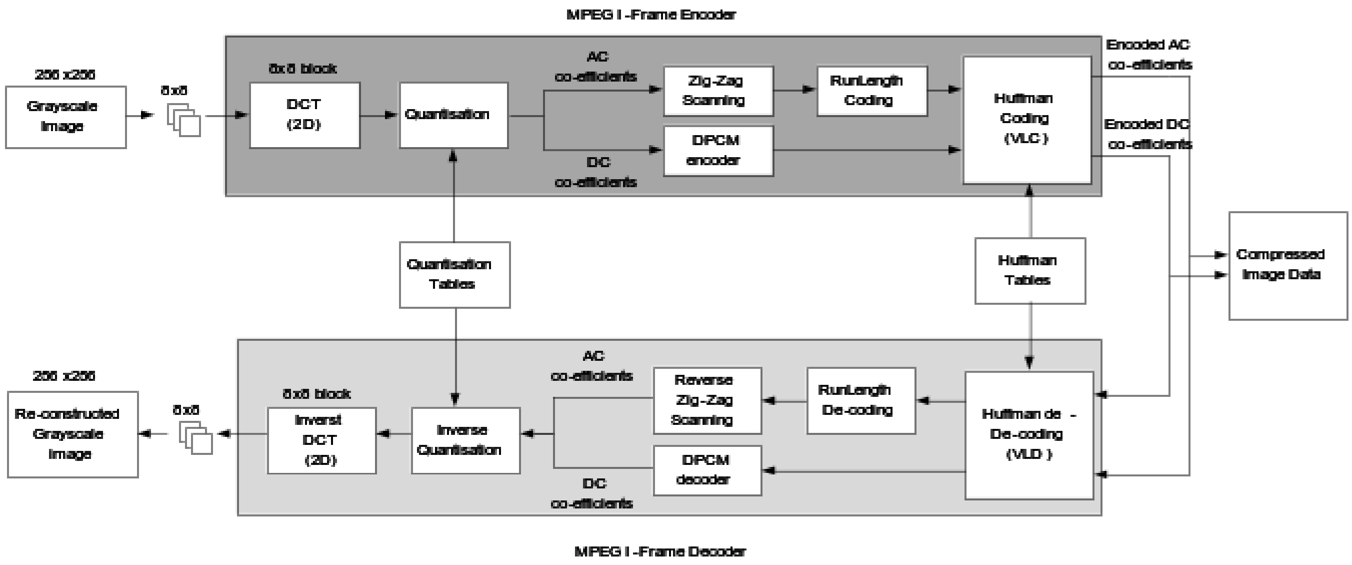


Figure 1.1 - High Level Block Diagram (MPEG/JPEG Encoder – Decoder)

- Each sub component of the system will be developed and tested together. E.g. DCT and IDCT implemented and tested together on sample data to verify if the reverse process outputs the original data.

- As each of the six sub-components are finished being developed, they will be integrated to the rest of the components to be tested as a whole system.
- From a development point – it was simpler to implement the Run-Length coding and Huffman coding as one entity. The code attached in the appendix will contain RLC and Huffman coding implemented in a single m-file.

2 DCT and IDCT

- Discrete Cosine Transform is used to convert an image from spatial domain to frequency domain.
- However before tranformation, the inputs need to be between the range -128 to 127 for DCT to be effective. So first all the dct inputs were reduced by 128.

(All Matlab source code can be found in the Appendix section)

2.1 1D-DCT/IDCT

2.1.1 Implementation (Simple-DCT/IDCT)

DCT:

Equations taken from [1] :

$$G(0)=\frac{1}{\sqrt{N}}\sum_{n=0}^{N-1} g(n)$$

$$G(k)=\sqrt{\frac{2}{N}}\sum_{n=0}^{N-1} g(n)Cos\left[\frac{(2n+1)k\pi}{2N}\right]$$

Where G(k) is the output of the 1-D DCT function and g(n) is the image pixels.

- N = length of the input array.
- Matlab does not support zero index arrays, hence the ‘n’ and ‘k’ counters were started from 1.
- G(1) coefficient calculated seperately – summation of the g(n) terms.
- Two loops were used.
  - **Inner Loop**: [g(n) * cos(((2n+1)k*pi)/2N)] term, n ▭1 to N
  - **Outer Loop**: G(k), k ▭ 2 to N

IDCT:

Equations taken from [1]:

$$g(x)=\sqrt{\frac{2}{N}}\sum_{u=0}^{N-1} C(u)G(u)Cos\left[\frac{(2x+1)u\pi}{2N}\right]$$

$$C(u)=\frac{1}{\sqrt{2}} \text{ for } u=0 \text{ else } C(u)=1$$

Where g(x) is the output if the 1D-IDCT function and G(u) is the DCT-coefficients.

- Similar to DCT, two loops. g(x): x ▭1 to N, G(u): u ▭ 1 to N
- C(u) vector formed before the loop.

2.1.2 Implementation (Fast-DCT/IDCT)

- The fast DCT uses fewer number of additions and multiply operations to perform a 1D-DCT (8 samples), this is useful because DCT is a very computationally expensive process

Fast DCT/IDCT was performed using the butterfly graph given in Appendix

- First the respective equations for each stage of the butterfly graph was written down on paper and then transfered to Matlab.
- The Matlab symbolic toolbox was used to derive the equations for the FAST-Inverse DCT.
- The cosine and sine terms were calculated upto 15 fractional places, and hardcoded into the matlab script. This was done to improve on the speed.
- A scalling factor of 2 needed to be used at the output to match the simple DCT results.

The table below shows the equations used for the Fast-DCT:

Input	Stage-1	Stage-2	Stage-3	Stage-4 (output)
p1	s1 = p1 + p8	t6 = (s6 * -cos(pi/4)) + (s7*cos(pi/4))	r1 = s1 + s4	c1 = (r1*cos(pi/4)) + (r2*cos(pi/4))
p2	s2 = p2 + p7	t7 = (s6 * cos(pi/4)) + (s7*cos(pi/4))	r2 = s2 - s3	c5 = (r1*cos(pi/4)) + (r2*-cos(pi/4))
p3	s3 = p3 + p6		r3 = s2 - s3	c3 = (r3*sin(pi/8)) + (r4*cos(pi/8))
p4	s4 = p4 + p5		r4 = s1 - s4	c7 = (r3*-sin(3pi/8)) + (r4*cos(3pi/8))
p5	s5 = p4 - p5		r5 = s5 + t6	c2 = (r5*sin(pi/16)) + (r8*cos(pi/16))
p6	s6 = p3 - p6		r6 = s5 - t6	c4 = (r6*-sin(3pi/16)) + (r7*cos(3pi/16))
p7	s7 = p2 - p7		r7 = s8 - t7	c6 = (r6*sin(5pi/16)) + (r7*cos(5pi/16))
p8	s8 = p1 p8		r8 = s8 + t7	c8 = (r5*-sin(7pi/16)) + (r8*cos(7pi/16))

The table below shows the equations used for the Fast-IDCT:

Input	Stage-1	Stage-2	Stage-3	Stage-4 (output)
c1	r1 = (2^(1/2)*(c1 + c5))/2	t5 = (r5/2) + (r6/2)	s1 = (r1/2) + (r4/2)	p1 = (s1/2) + (t8/2)
c2	r2 = (2^(1/2)*(c1 - c5))/2	t6 = (r5/2) - (r6/2)	s2 = (r2/2) + (r3/2)	p2 = (s2/2) + (s7/2)

c3	r3 = (c3*(2 - 2^(1/2))^(1/2))/2 - (c7*(2^(1/2) + 2)^(1/2))/2	t7 = (r8/2) - (r7/2)	s3 = (r2/2) + (r3/2)	p3 = (s3/2) + (s6/2)
c4	r4 = (c7*(2 - 2^(1/2))^(1/2))/2 + (c3*(2^(1/2) + 2)^(1/2))/2	t8 = (r7/2) + (r8/2)	s4 = (r1/2) + (r4/2)	p4 = (s4/2) + (t5/2)
c5	r5 = (c2*cos(7pi/16) - c8*cos(pi/16))/(cos(pi/16)*sin(7pi/16) + cos(7pi/16)*sin(pi/16))		s6 = -(2^(1/2)*(t6 - t7))/2	p5 = (s4/2) - (t5/2)
c6	r6 = -(c4*cos(5pi/16) - c6*cos(3pi/16))/(cos(3pi/16)*sin(5pi/16) + cos(5pi/16)*sin(3pi/16))		s7 = (2^(1/2)*(t6 + t7))/2	p6 = (s3/2) - (s6/2)
c7	r7 = (c4*sin(5pi/16) + c6*sin(3pi/16))/(cos(3pi/16)*sin(5pi/16) + cos(5pi/16)*sin(3pi/16))			p7 = (s2/2) - (s7/2)
c8	r8 = (c2*sin(7pi/16) + c8*sin(pi/16))/(cos(pi/16)*sin(7pi/16) + cos(7pi/16)*sin(pi/16))			p8 = (s1/2) - (t8/2)

2.1.3 Testing and results

DCT Output Testing

Data was entered to the DCT function and the co-efficients were checked by entering the co-efficients back to the IDCT function and checked if the IDCT output = original data. Data = random 8 signed integers between (-128 to 127)

Eg: data =[-88   121   113   3   66   -13   -50   77]

DCT-simple - **dct_1()**

```
dct_coeffs =
 [ 8.0964e+001,  1.9031e+001, -4.2504e+001, -1.2957e+002, -3.9952e+001 ,-
1.4359e+002, -1.9111e+000,
  1.9681e+001]
```

DCT-fast – **fdct_1()**

```
dct_coeffs =
[ 8.0964e+001,  1.9031e+001, -4.2504e+001, -1.2957e+002, -3.9952e+001,
-1.4359e+002, -1.9111e+000,
  1.9681e+001]
```

Inverse-DCT – **idct_1()**

```
idata =[ -88   121   113     3    66   -13   -50    77]
```

The DCT function output was also compared against the Matlab built-in ‘dct’ function.

The Fast DCT output = Simple DCT output.

Speed Calculation

Using Matlab ‘tic’ and ‘toc’ to measure how long the functions took to execute. Fast-dct vs simple-dct speed comparisons were made. (please refer Appendix for source code)

Test details:

- Random Data □ DCT □ IDCT □ Original Data.
- 8 data samples.
- 10000 tests run , each time a new set of 8 random data samples generated. Difference measured at each test (simple_method_time _elapsed – fast_method_time_elapsed).

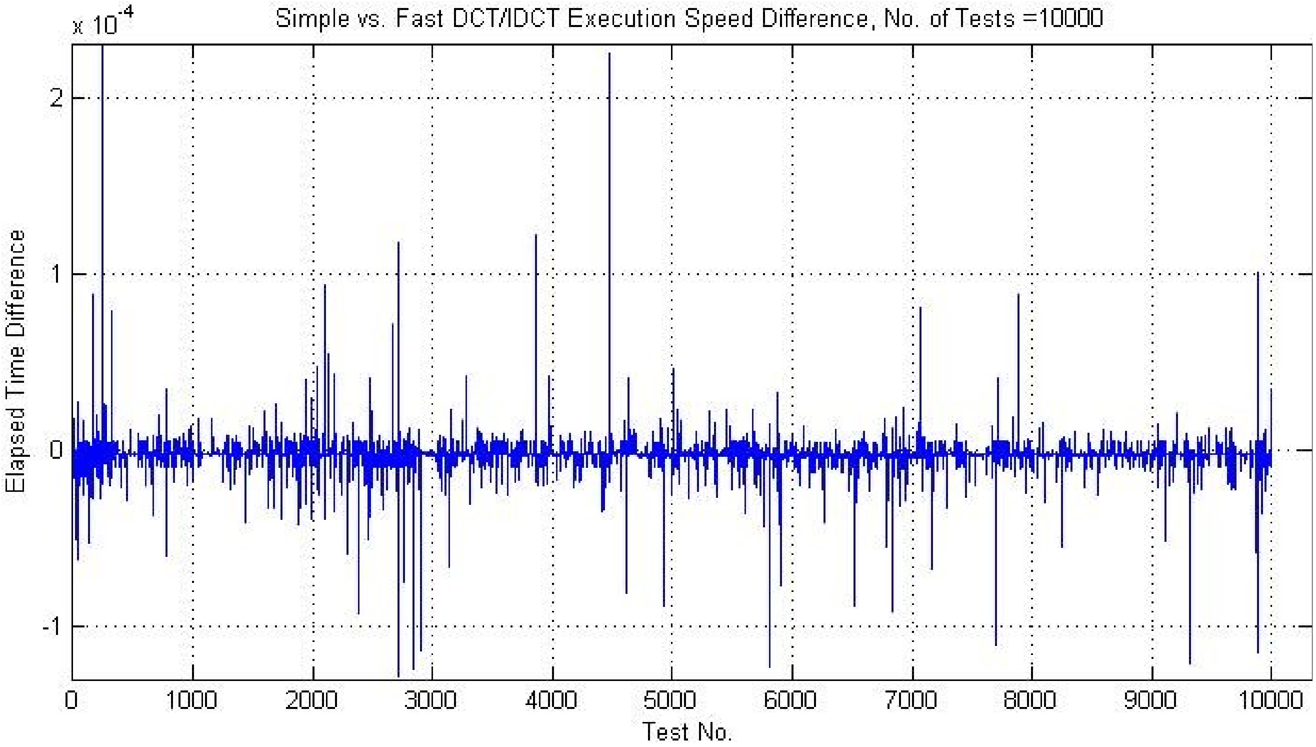


Figure 2.2 – Execution speed difference between simple vs. Fast DCT/IDCT, for 10000 Tests.

Figure 2.1 – shows the execution time difference between simple DCT/IDCT vs. Fast DCT/IDCT. The differences were obtained for 10000 tests and plotted.

Mean execution speed difference between simple vs. Fast DCT/IDCT =

-2.1511e-005 seconds (10 tests)

1.2479e-005 seconds (100 tests)

7.9007e-006 seconds (1000 tests)

4.3022e-006 seconds (10000 tests)

Thereby proving overall the Fast DCT gives slight improvement over simple-dct. Maybe speed difference would be larger for larger sample sizes (eg: 32, 64 DCT coefficients).

2.2 2D-DCT/IDCT

2D-DCT is basically performing the frequency transform for a 2-Dimentional Matrix (rows x cols).

2.2.1 Implementation - 2D Block Transform (8x8 block)

- First perform the above 1D-DCT on the rows of the 2D-matrix, and then perform the same 1D-DCT on the result of the first transform – but column-wise.

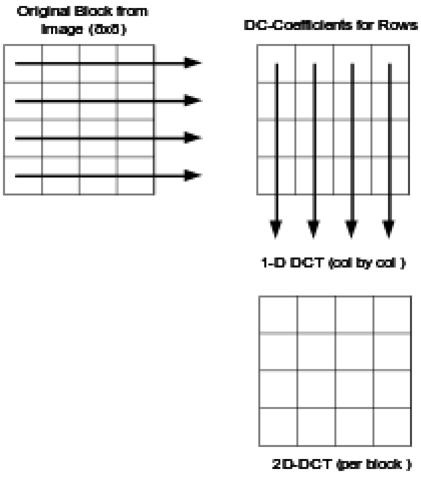


Figure 2.3 - 2D-DCT transform being performed using row-col wise 1D-DCT

- The 8x8 2D-DCT function will have an input parameter called ‘dct_type’ which will be 1 or 0, and thereby allowing the 2D-DCT to be performed using the fast/simple 1D-DCT.
    - 1 = 'fast dct'
    - 0 = ‘simple dct’
- The 2D-IDCT function is implemented exactly the same way as the 2D-DCT.

2.2.2 Implementation – Splitting the Image into 8x8 sub-blocks

The matlab function **mat2cell** was used to split the entire image into an array of adjacent sub-matrices – and the final resulting data structure in matlab is known as a ‘cell’.

```
% split the input larger matrix into smaller 8x8 blocks
split_img = mat2cell(img, b*ones(m/b,1), b*ones(1,n/b));
```

where b = 8 (block size), m = width of image(eg:512) and n = height of image (eg: 512)

b*ones(m/b,1) = will result in an array of 64 length with 8 as the value for each index, this informs the mat2cell() function, to divide the input matrix into 64 blocks, of size 8.

After dividing the image into equal sub blocks (8x8) the Compression process (i.e DCT->Quantisation->ZigZag scanning -> RLC->Huffman etc.) was performed on each sub block, and stored in the matlab workspace for decoding.

2.2.3 Implementation – Simple 2D Transform (using equations)

2D-DCT:

Equations taken from [1]:

$$F(u,v)=\frac{1}{4}C(u)C(v)\sum_{i=0}^{7}\sum_{j=0}^{7}f(i,j)Cos\left[(2i+1)u\frac{\pi}{16}\right]Cos\left[(2j+1)v\frac{\pi}{16}\right]$$

$$C(u)=\frac{1}{\sqrt{2}}\text{ for }u=0\text{ else }C(u)=1$$

$$C(v)=\frac{1}{\sqrt{2}}\text{ for }v=0\text{ else }C(v)=1$$

- The basic 2D-DCT transform was implmented using 4 loops. 2 inner loops and 2 outer loops nested within each other.

2D-IDCT:

Equations taken from [4]:

$$2\pi\frac{i}{2N}(u+0.5)$$

$$2\pi\frac{j}{2N}(v+0.5)$$

$$f(u,v)=\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}c(i,j)F(i,j)\cos ¿¿\cos ¿¿$$

2.2.4 Testing and results

(All scripts written for testing can be found in the Appendix)

Test Inputs:

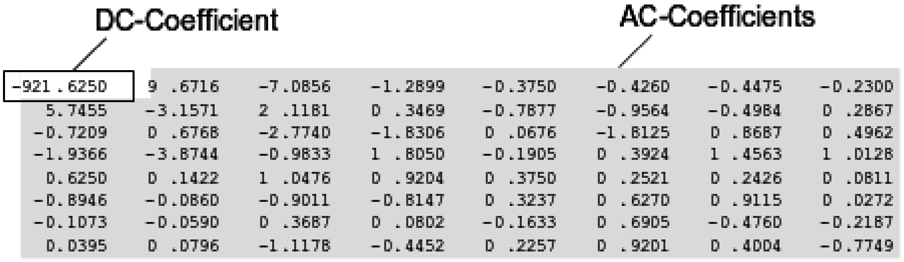
The 2D (8x8 block) DCT was tested against several 8x8 block matrices taken from the **'cameraman.bmp'** image.

Input Matrix - block (15,15) of cameraman.bmp:

```
 -116  -115  -113  -114  -113  -114  -116  -116
 -115  -112  -113  -114  -113  -115  -116  -116
 -114  -112  -113  -112  -115  -116  -116  -117
 -113  -113  -114  -114  -115  -116  -116  -117
 -114  -115  -115  -114  -115  -115  -116  -118
 -116  -116  -115  -115  -115  -116  -117  -119
```

```
  -117  -116  -113  -113  -115  -117  -120  -120
  -115  -113  -113  -113  -115  -118  -120  -120
```

Output of **dct_2_8x8()** – 2D block DCT function:



```
-921.6250   9.6716  -7.0856  -1.2899  -0.3750  -0.4260  -0.4475  -0.2300
   5.7455  -3.1571   2.1181   0.3469  -0.7877  -0.9564  -0.4984   0.2867
  -0.7209   0.6768  -2.7740  -1.8306   0.0676  -1.8125   0.8687   0.4962
  -1.9366  -3.8744  -0.9833   1.8050  -0.1905   0.3924   1.4563   1.0128
   0.6250   0.1422   1.0476   0.9204   0.3750   0.2521   0.2426   0.0811
  -0.8946  -0.0860  -0.9011  -0.8147   0.3237   0.6270   0.9115   0.0272
  -0.1073  -0.0590   0.3687   0.0802  -0.1633   0.6905  -0.4760  -0.2187
   0.0395   0.0796  -1.1178  -0.4452   0.2257   0.9201   0.4004  -0.7749
```

Output from the 2D-DCT function **dct_2_simple()** – which uses the basic transform equations:

```
 -921.6250    9.6716   -7.0856   -1.2899   -0.3750   -0.4260   -0.4475   -0.2300
    5.7455   -3.1571    2.1181    0.3469   -0.7877   -0.9564   -0.4984    0.2867
   -0.7209    0.6768   -2.7740   -1.8306    0.0676   -1.8125    0.8687    0.4962
   -1.9366   -3.8744   -0.9833    1.8050   -0.1905    0.3924    1.4563    1.0128
    0.6250    0.1422    1.0476    0.9204    0.3750    0.2521    0.2426    0.0811
   -0.8946   -0.0860   -0.9011   -0.8147    0.3237    0.6270    0.9115    0.0272
   -0.1073   -0.0590    0.3687    0.0802   -0.1633    0.6905   -0.4760   -0.2187
    0.0395    0.0796   -1.1178   -0.4452    0.2257    0.9201    0.4004   -0.7749
```

The above results confirm that the dct_2_8x8() – 2D-block DCT function is working as expected.

Input Block (8x8)	**Dct_2_8x8() Output**

Block (25,25) of cameraman.bmp
-1 1 2 -2 0 8 0 -1
3 -1 0 -4 2 1 -3 -6
-6 -9 0 -5 -4 -12 -15 -13
0 2 12 15 7 8 7 -3
8 3 9 10 2 10 1 1
2 -3 -2 -1 -9 -3 2 1
-9 9 5 2 3 5 3 -5
-1 11 11 21 14 7 -3 0
	10.8750 8.9020 -17.0173 -1.8070 -5.6250 2.0547 3.6663 -3.0339
	-17.1467 -0.2090 6.4305 6.9031 0.8828 0.8586 6.1674 5.9445
	1.4643 -0.9024 -2.9383 1.3091 -1.8894 -10.0181 -3.4848 -1.5019
	1.3836 -7.7338 15.6347 4.7052 -5.1555 2.8297 2.2365 -2.9871
	28.8750 -0.6087 -9.9761 -0.8467 0.8750 1.7943 2.3734 -5.6904
	-2.7352 -4.8184 -3.9899 -2.8093 -8.1524 -0.5839 -3.7161 1.0821
	-15.0835 1.6912 4.0152 -4.2335 6.8711 2.0481 2.1883 0.0230
	-7.9551 3.8054 -1.3783 -0.1745 -0.9315 -4.7915 2.9855 0.0877

Block (10,10) of cameraman.bmp
-114 -114 -114 -109 -111 -111 -110 -109
-113 -112 -112 -110 -109 -110 -110 -110
-113 -112 -112 -112 -111 -111 -112 -110
-114 -113 -112 -111 -112 -111 -112 -112
-115 -114 -112 -112 -113 -113 -112 -112
-116 -115 -114 -113 -113 -114 -112 -112
-113 -114 -113 -113 -111 -113 -113 -112
-115 -115 -114 -114 -114 -114 -113 -114
	-899.3750 -6.6084 -2.8161 -1.6761 1.3750 0.2331 -0.5924 -0.1675
	7.7710 -2.0411 -0.7692 0.0906 0.9197 -0.2489 -0.4555 -1.8090
	0.1305 -0.9966 0.2652 1.0615 1.4138 0.3978 -0.7866 -0.0721
	-0.8285 -2.0110 -0.2720 -1.0245 1.0979 1.1048 -0.9660 -0.1314
	-1.8750 -0.1651 -0.3943 -0.6822 -0.6250 1.6960 0.0280 -1.0924
	0.7221 0.2349 0.1635 0.4887 1.0342 -0.2987 0.5134 -0.4504
	-2.6247 -0.9531 0.9634 -0.8014 -0.1797 0.2165 -0.2652 -0.4941
	0.4643 0.5749 0.8057 -0.3118 0.4378 -0.5129 0.6159 0.3643

Block (15,25) of cameraman.bmp
20 -25 -37 -34 -6 -32 -39 -42
22 -11 -26 -11 18 -3 -25 -25
38 40 36 39 47 44 38 39
48 56 56 56 58 58 56 57
49 53 56 59 58 55 55 58
48 62 59 58 58 57 60 60
47 59 60 59 56 59 60 60
47 60 58 59 58 59 58 56
	291.2500 8.6902 -8.3013 19.3195 12.2500 -10.1950 -3.0558 4.4918
	-203.0548 28.8257 6.1936 40.6194 28.8565 -2.5425 4.2287 11.1977
	-128.5426 21.6671 5.3891 23.3219 13.6669 -0.2621 0.6213 6.6342
	-53.3218 11.5688 7.4801 5.3204 7.5936 2.5074 -0.1440 5.6733
	-4.2500 4.2557 3.5370 -2.0690 1.2500 3.5012 0.6997 2.8239
	21.9245 -2.6672 5.5060 -8.9705 -7.3996 2.3230 0.0188 0.8011
	20.2311 -2.1683 3.6213 -7.3918 -6.2022 0.1087 -2.3891 0.4426
	7.6800 -2.7803 0.9965 -4.6897 -0.5088 2.6133 1.2146 0.0309

All tests were verified by performing 2D inverse DCT and comparing against the basic transform equations (function: **idct_2_simple()**).

The following is the image of 2D-DCT performed on the entire lena512.bmp image. This took over **1-2 minutes**, due to the computationaly intensive nature of the 2D-DCT transform. (output of the **dct_2_8x8()** function – block size = size of the whole image = 512).

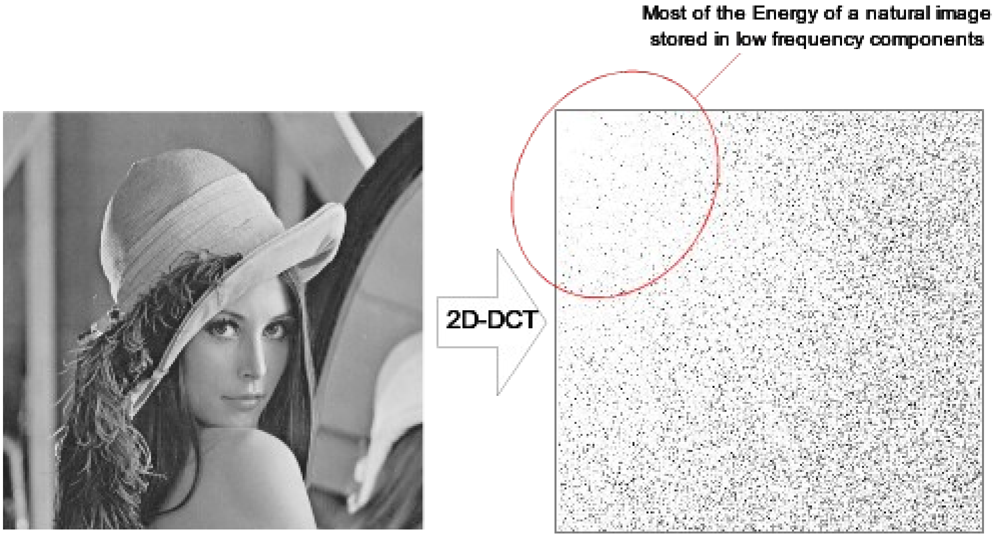

Most of the Energy of a natural image stored in low frequency components
2D-DCT

Figure 2.4 - Row/Column DCT performed on Whole image

However when the 2D-basic transform(via equations) was performed on the entire image, the speed difference was very large. The basic transform took over 20 minutes to compute the 2D-DCT of theabove image(Lena512.bmp).

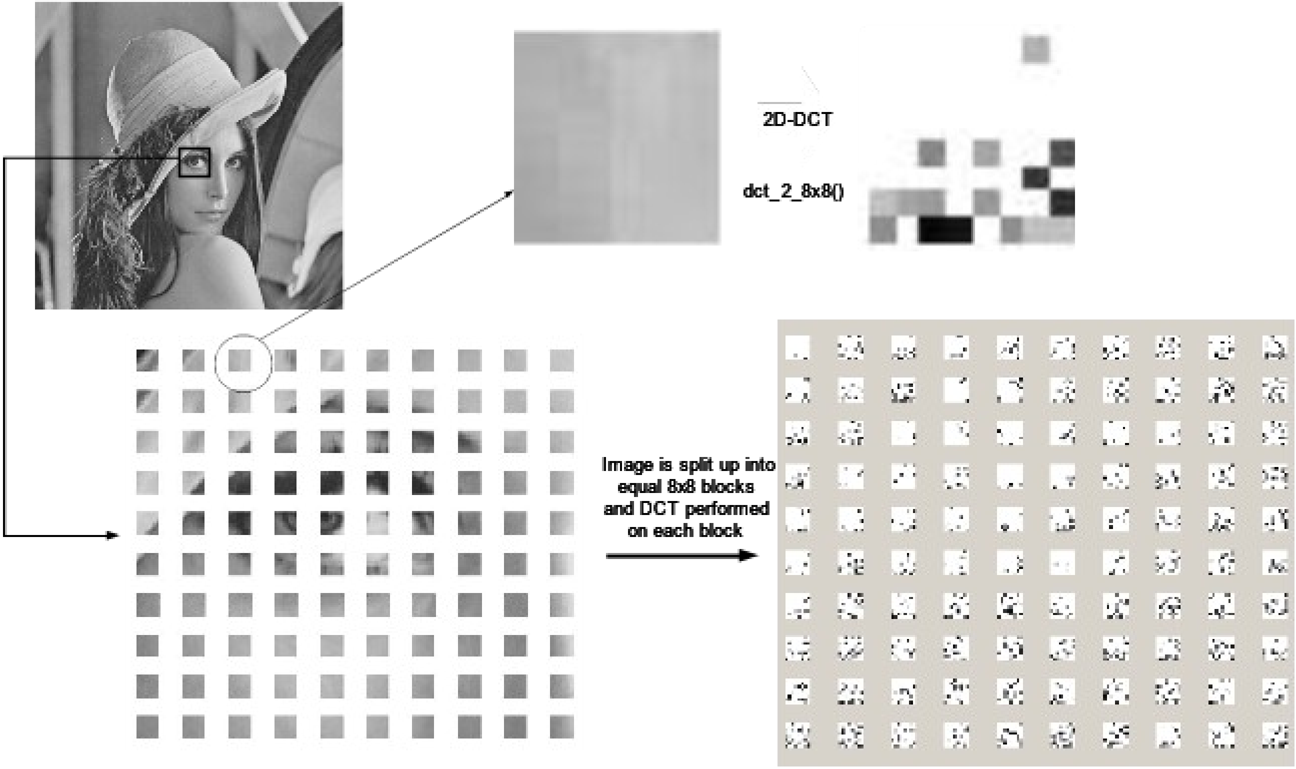

2D-DCT
dct_2_8x8()
Image is split up into equal 8x8 blocks and DCT performed on each block

Figure 2.5 - Image sub-divided and DCT performed on each sub-block

From the above illustrations, it is clear that the DCT helps to seperate the high energy components from the low energy components(high-energy=white, low-energy=black), and this feature is taken into use in the other steps of the MPEG/JPEG compression.

Speed Comparison Test

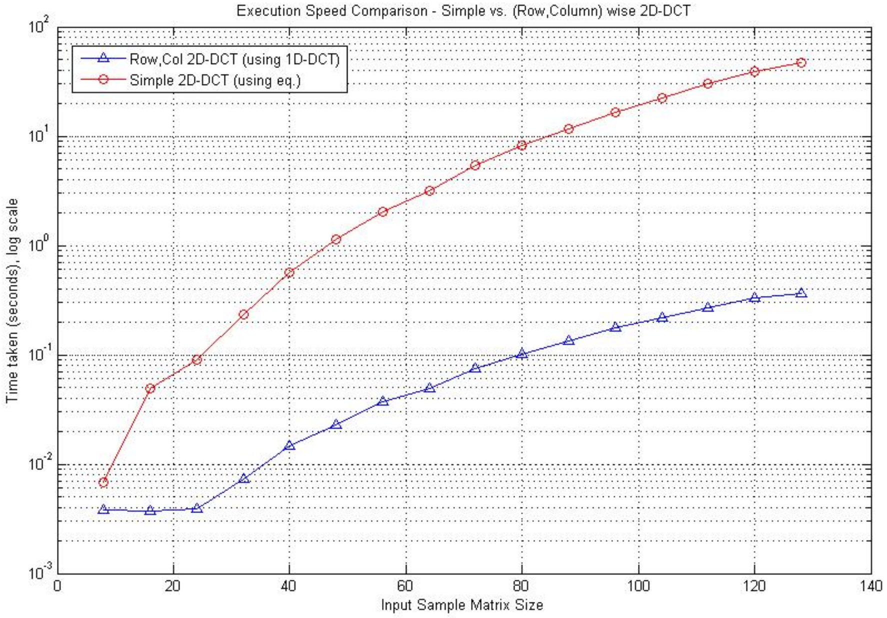


Figure 2.6 - Execution Speed Comparison - Simple vs. (Row-Col) wise 2D-DCT calculation

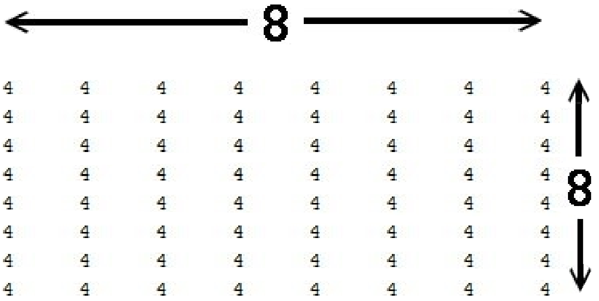
From Figure 2.3 it is clear that the Simple 2D-DCT function (implemented using the basic transform equations) is extremely slow for large matrix sizes.

3 Quantisation and Inverse Quantisation

3.1 Implementation notes

3.1.1 Linear quantisation/inverse-quantisation

Linear quantisation and inverse quantisation is done using a 8x8 Q matrix with element values equal to the required step size of the as shown alongside. In the example shown, step size of 4 is selected.



$$\begin{matrix} 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \end{matrix}$$

The Q matrix thus received is divided element wise from the input matrix and cut-off procedure is performed to get the quantised matrix.

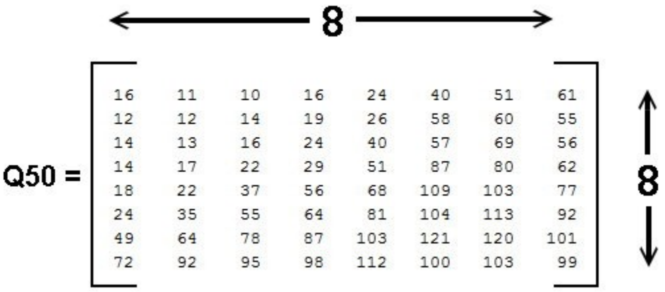
Cut-off is the process by which all values below the cut-off threshold are truncated to zero.

For inverse quantisation, the procedure is the reverse of quantisation. The Q matrix is multiplied element wise to the quantised matrix (input matrix) to get the original matrix.

3.1.2 Non-linear quantisation/inverse-quantisation

Non-linear quantisation performed here is based around Q50 matrix.

The level of non-linear quantisation can be varied from 1 to 100. For any level, the corresponding Q matrix has to be calculated from Q50 matrix.



$$Q50 = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

For quantisation and inverse-quantisation at level 50 Q50 matrix can be directly used.

For levels above 50, the Q matrix can be calculate by the formulae **Q = Q50 * (100 – level)/50**

For levels below 50, the Q matrix can be calculate by the formulae **Q = Q50 * (50/level)**

The resulting Q matrix should be rounded and clipped between values 1 to 255.

After receiving the final Q matrix, for quantisation, Q matrix should be divided element wise from the input matrix and rounded.

For inverse quantisation, Q matrix has to be multiplied to input matrix element wise.

3.2 Testing and results

Along with the images results of tests performed with varying parameters, for getting a mathematical estimation of performance, several other calculations were made.

Mean Square Error (MSE)

$$MSE=\frac{1}{N_{rows}M_{cols}}\sum_{\text{¿}}^{\overset{\text{¿}}{i=1}} j=1\,\text{¿}\,M\,\text{¿¿¿}\square\left(x_{ij}-\hat{x}_{ij}\right)^2$$

x_{ij} = Original value, $\hat{x}_{ij}$ = Reconstructed value

Signal to Noise Ratio (SNR)

$$SNR=10\log_{10}\left(\frac{\sigma^2}{MSE}\right)dB$$

σ^2 = Variance

PeakSNR

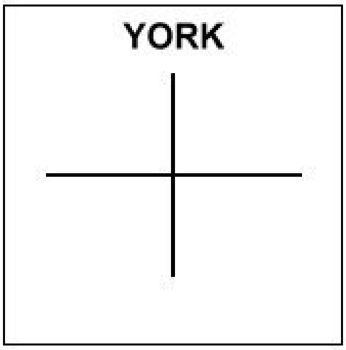
$$PeakSNR=10\log_{10}\left(\frac{{x_{max}}^2}{MSE}\right)$$

$x_{max}^{\square}$ = typically 255

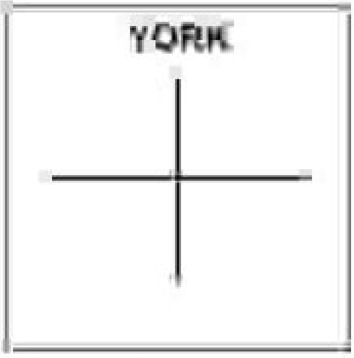
Zeros are the number of zeros present in the image after quantisation which can be used to approximately determine compression. For plotting, Zero% is calculated which depicts how many zeros are present per 100 pixels.

3.2.1 Linear quantisation

Quantisation error is an important issue to be taken under consideration as higher levels of error can compromise the quality of image tremendously. To depict how quantisation error looks like, a binary image was linearly quantised with cut off at 8 and step size 32.

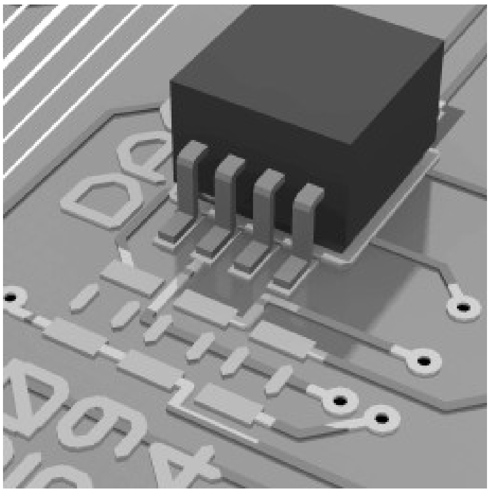


Original image

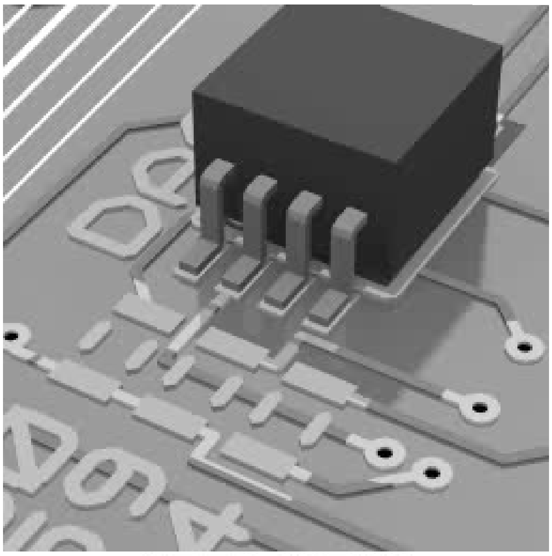


Liniarly quantised
Cutoff : 8 Step size 32

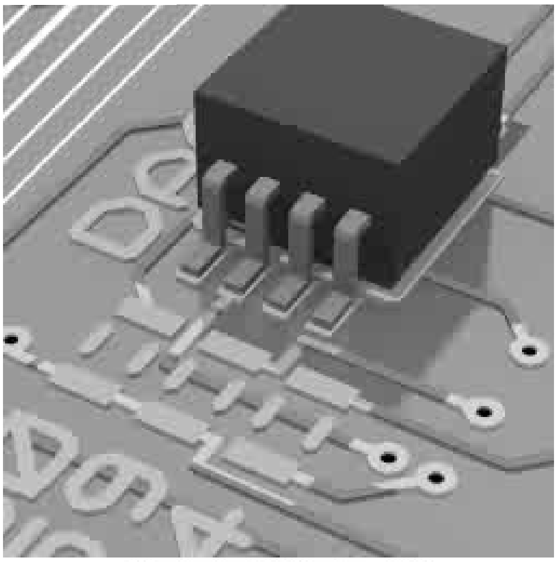
3.2.1.1Varying step size with cut off set at 2



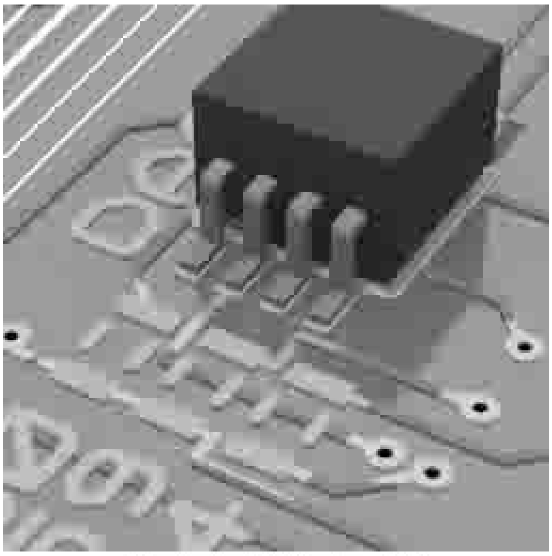
Original



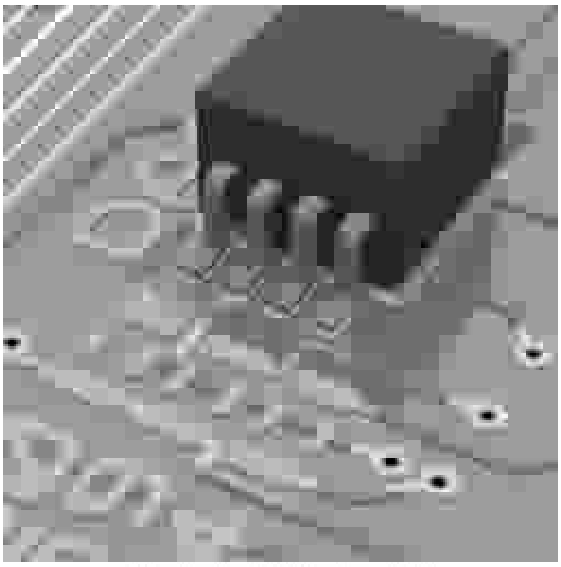
STEP SIZE 8



STEP SIZE 16



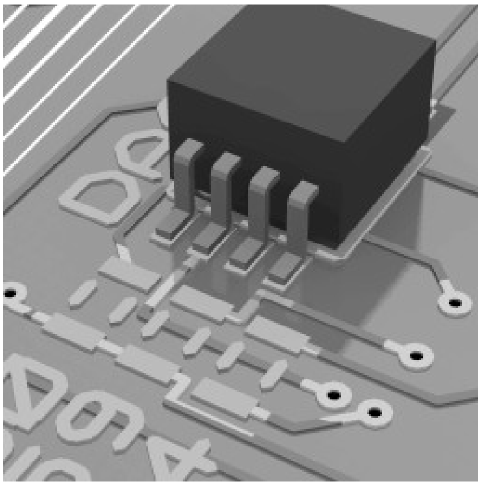
STEP SIZE 32



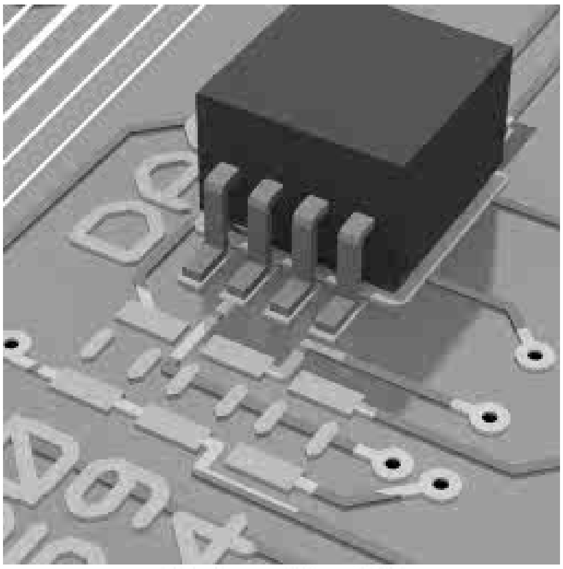
STEP SIZE 64

It can be seen that as the step size increases, the quality of image reproduced drops.

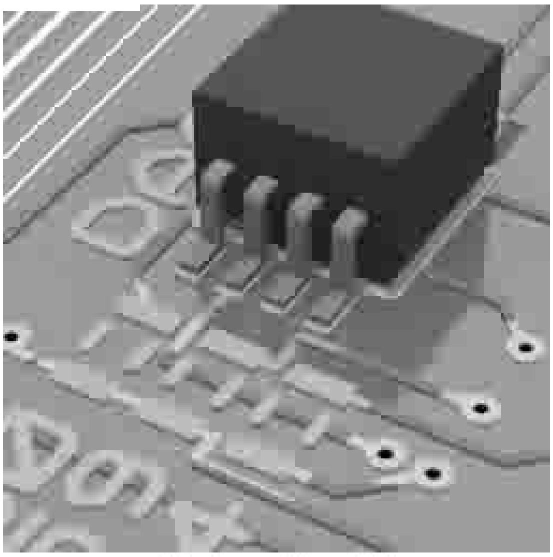
3.2.1.2Varying cut off with step size fixed at 32



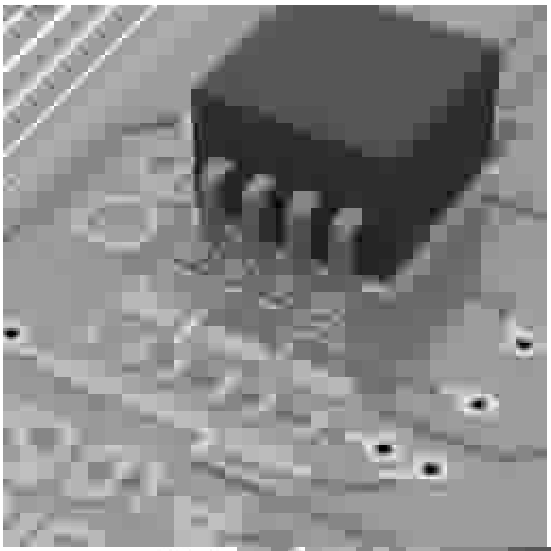
Original



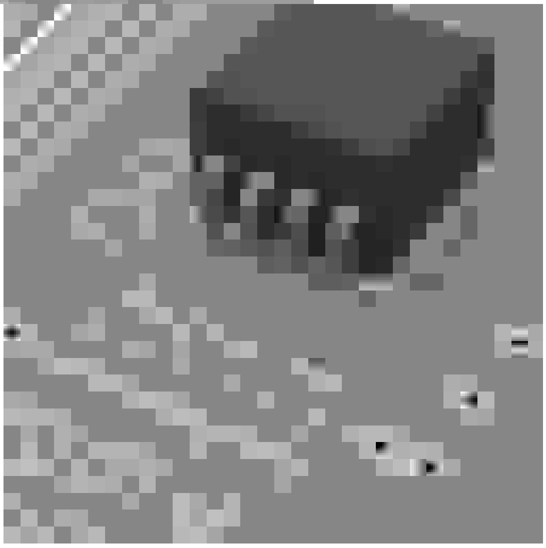
Cut off at 0



Cut off at 2



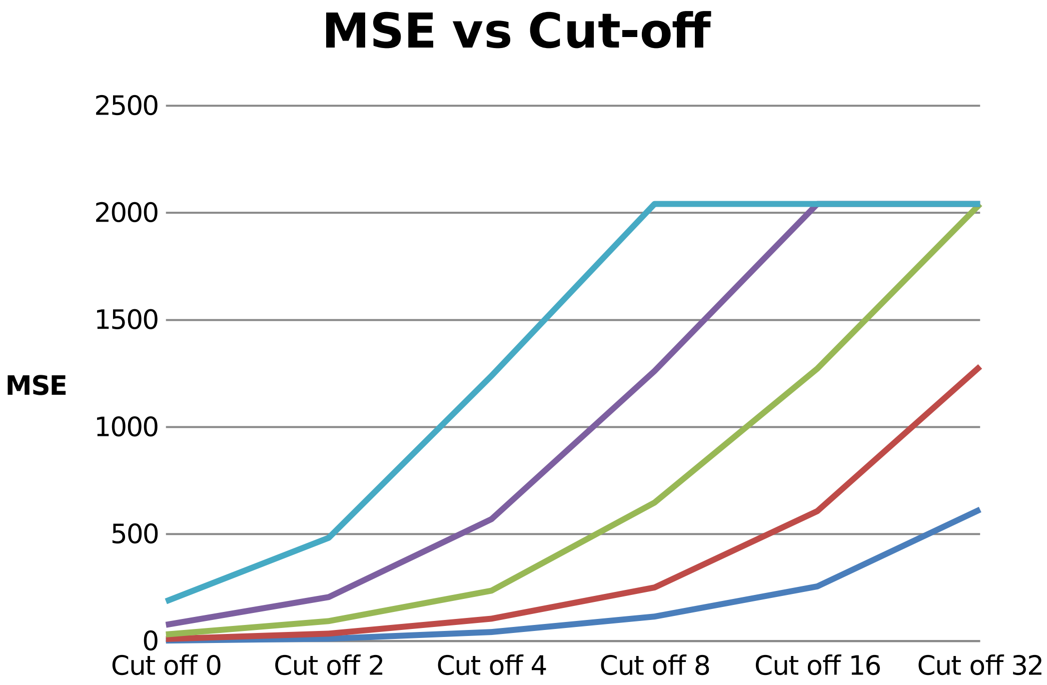
Cut



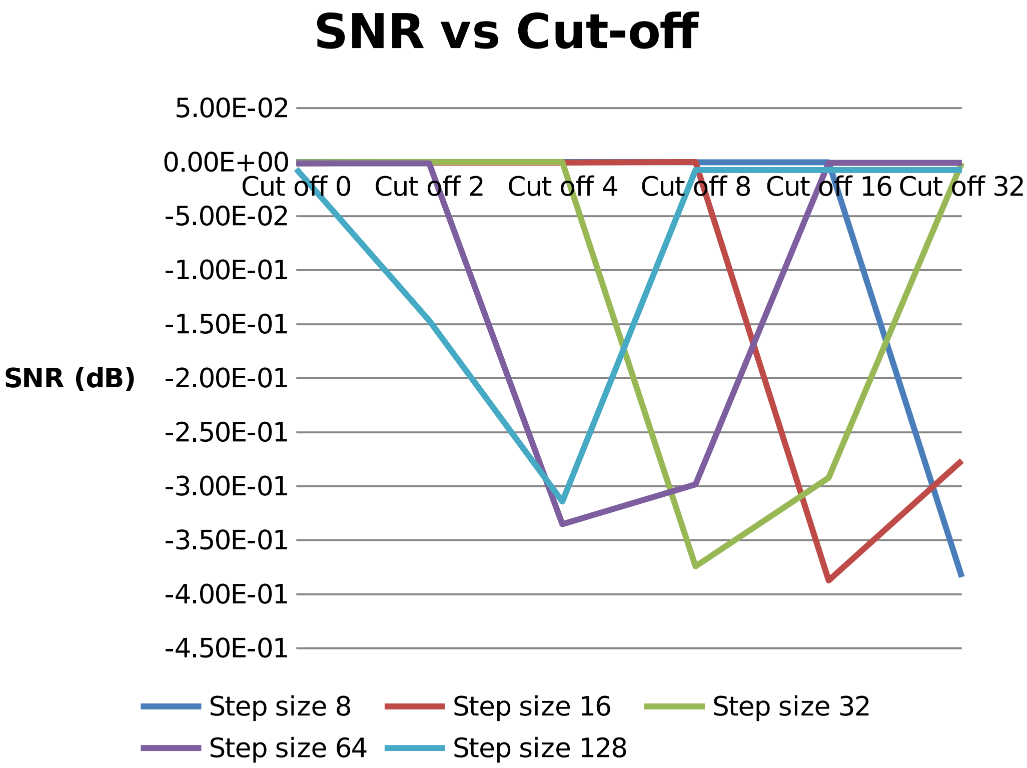
Cut off at 8

Increase in cut off level also degrades the image but it seems to have a drastic effect severely degrading the quality when compared to step width variations.

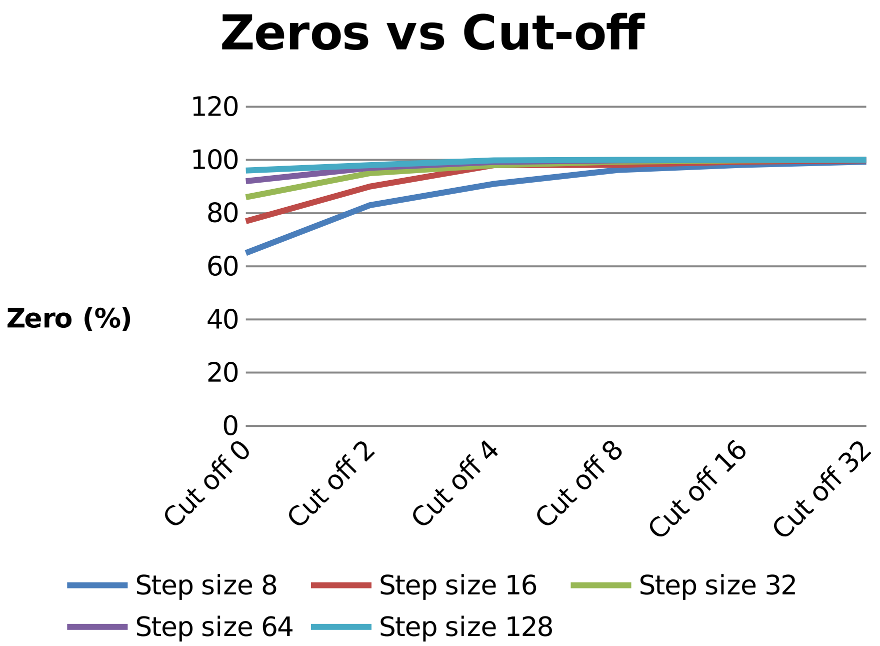
3.2.1.3Result Analysis



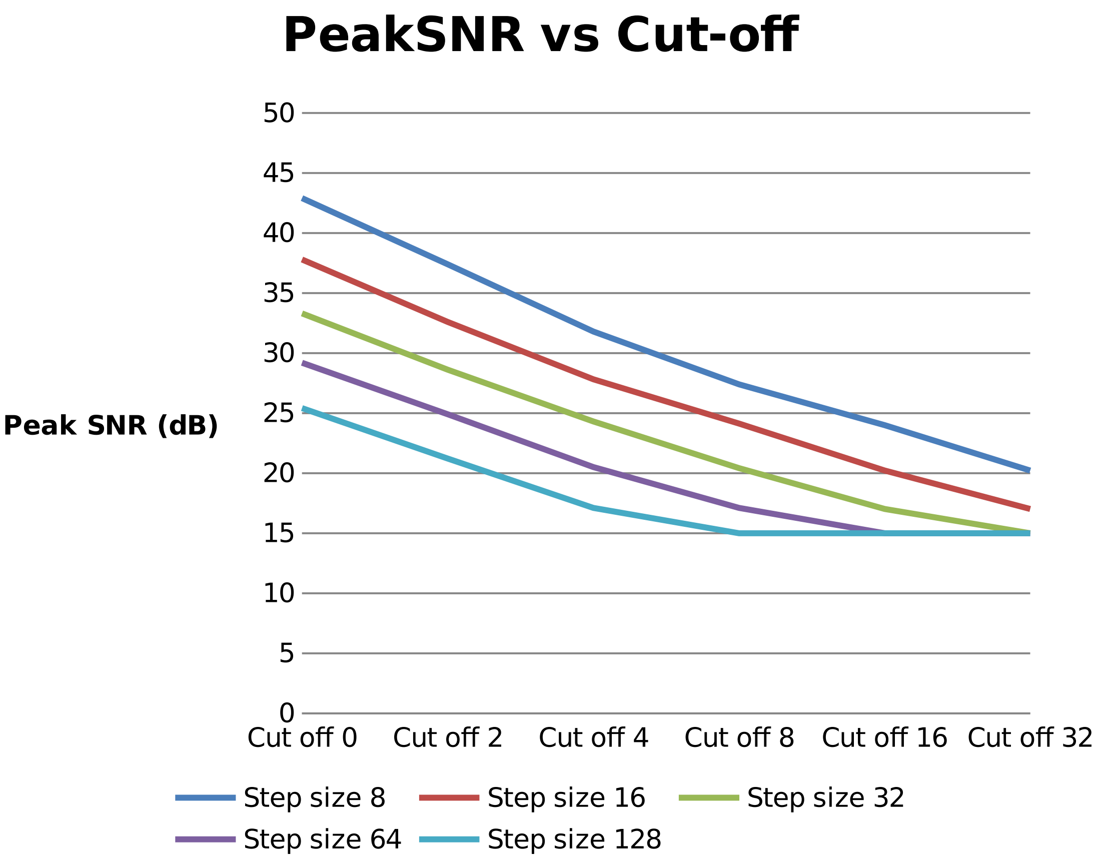
From the MSE vs Cut-off graph, it can be seen that as the value of cut-off increases, MSE inverses. It can be observed that MSE reaches saturation at 2042 after which it stays constant no matter the step size.



From the SNR vs Cut-off plot, it can be observed that the SNR drops considerably at a certain cut off value which is distinct for each step size. The intensity of drop seems to be increasing with increase in cut off value.



The number of zeros can be used for estimation of compression approximately. The above plot points out that the compression increases with increase in step size and also cut off.

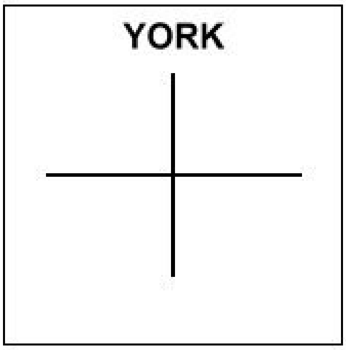


From PeakSNR vs Cut-off graph, it is quite evident how increase in cut off level degrades Peak SNR. As expected, the level is lower in higher step sizes compared to lower ones.

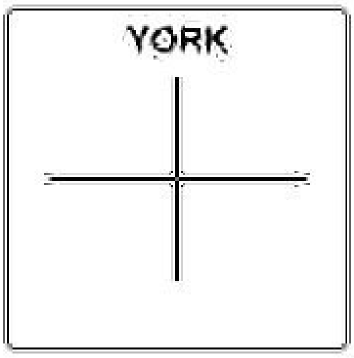
It can also be observed that at Peak SNR value gets saturated at value 15 after which it doesn't drop and holds steady.

3.2.2 Non-Linear quantisation

Quantisation error is an important issue to be taken under consideration as higher levels of error can compromise the quality of image tremendously. To depict how quantisation error looks like, a binary image was linearly quantised at level 2 and shown below.

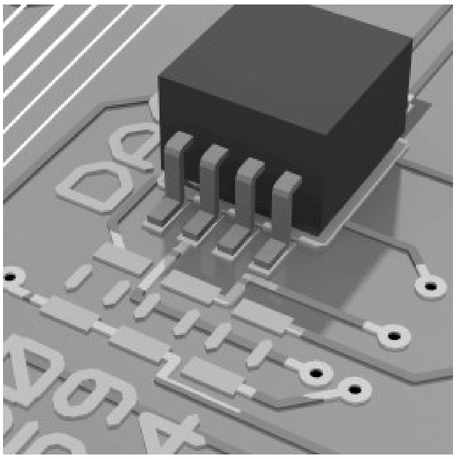


Original image

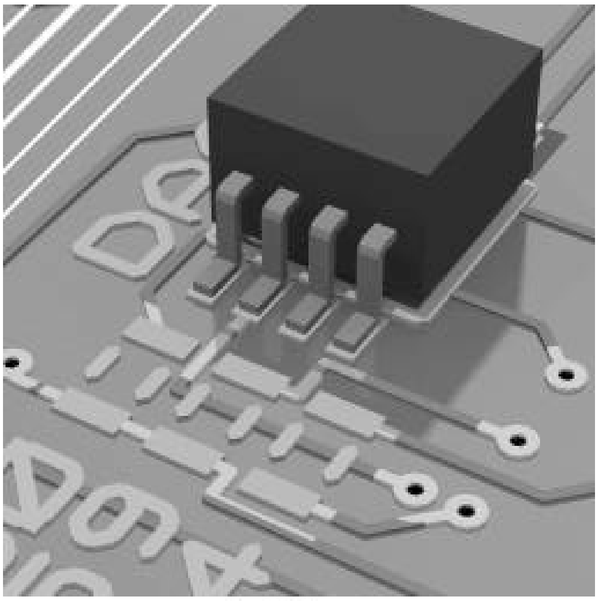


Non-Liniary Quantised at 2

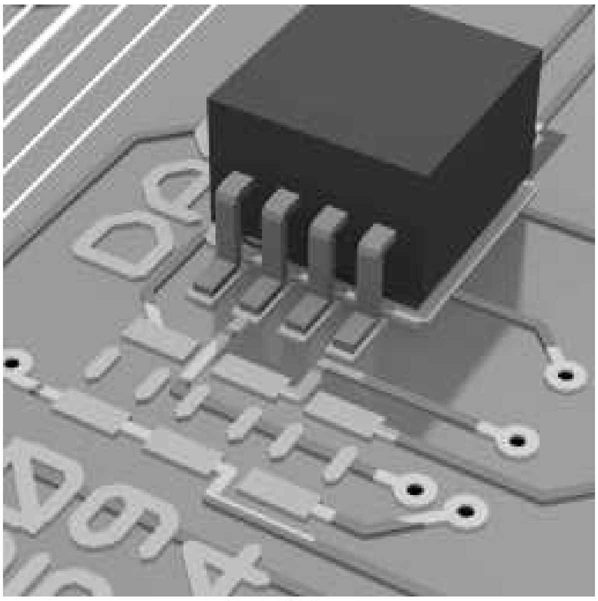
3.2.2.1With different quality ratios



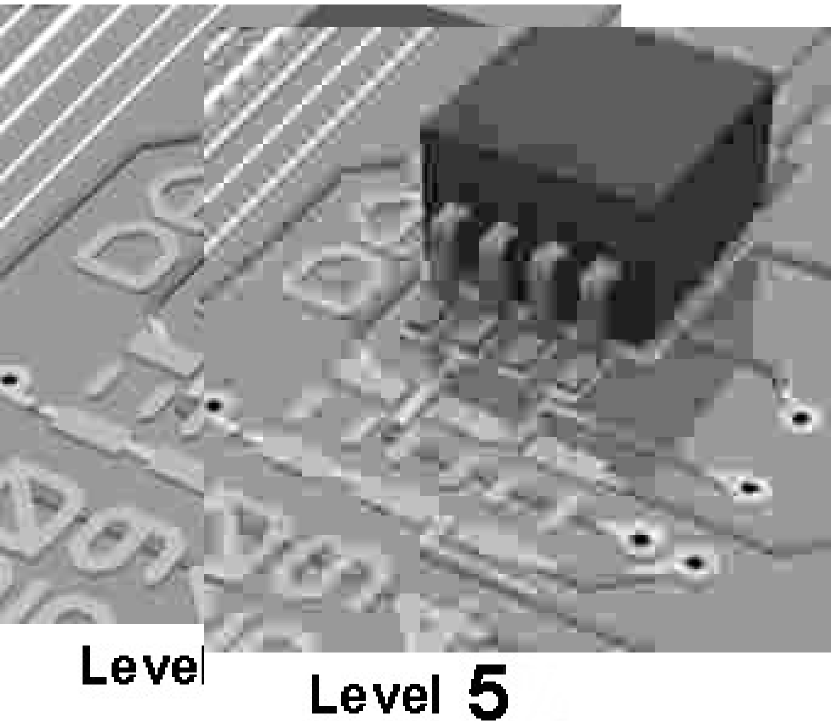
Original



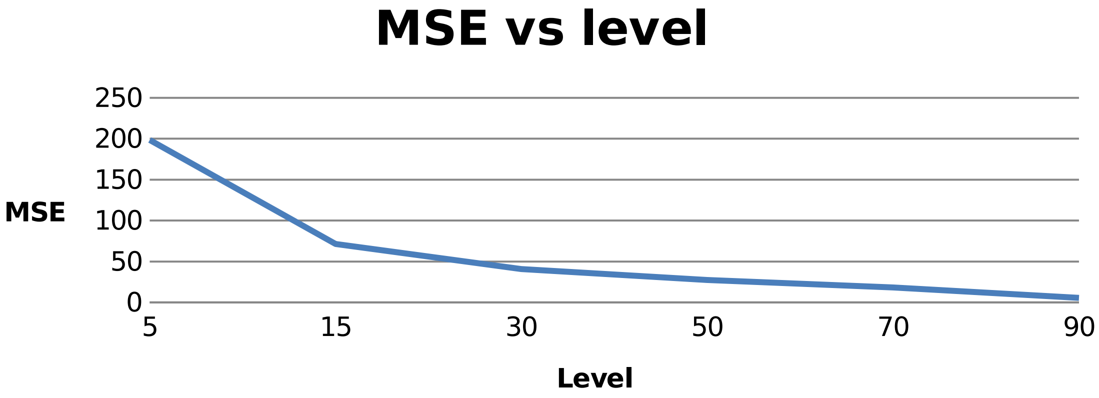
Level 70



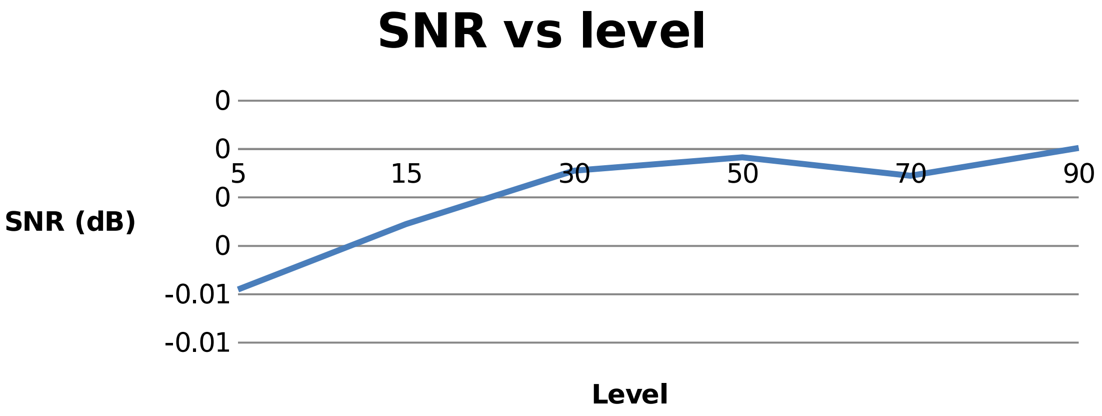
Level 50



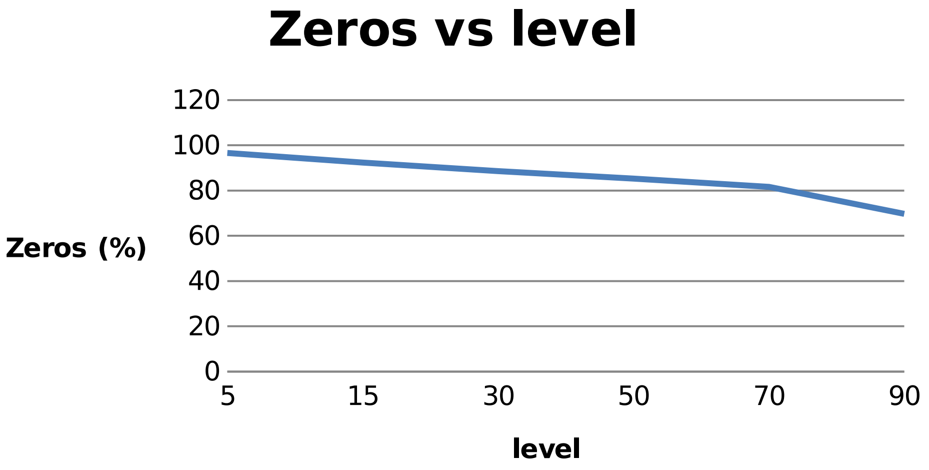
3.2.2.2Result analysis



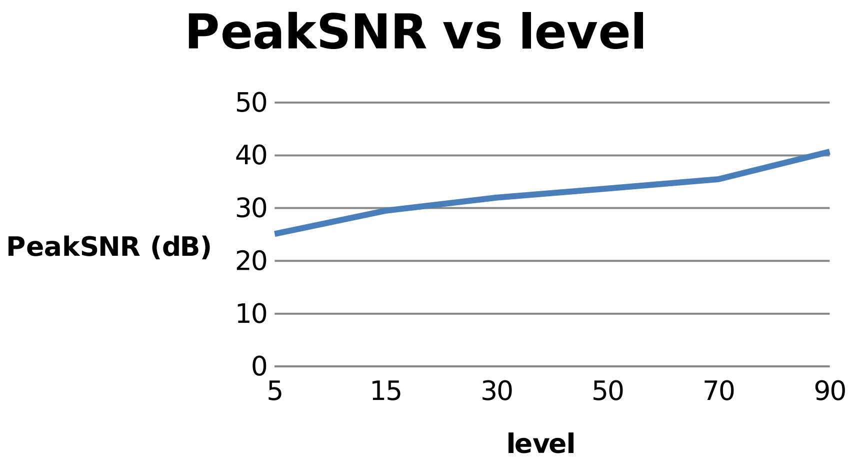
From the MSE vs level graph, it can be seen that MSE drops phenomenally with increase in quality of image. So, if the image is compressed to high levels, the overall quality seems to degrade and hence MSE increases.



SNR vs level plot just validates the fact that as image quality increases, SNR increases. It is seen that at 70 there is a slight drop in SNR but the exact reason is unknown.

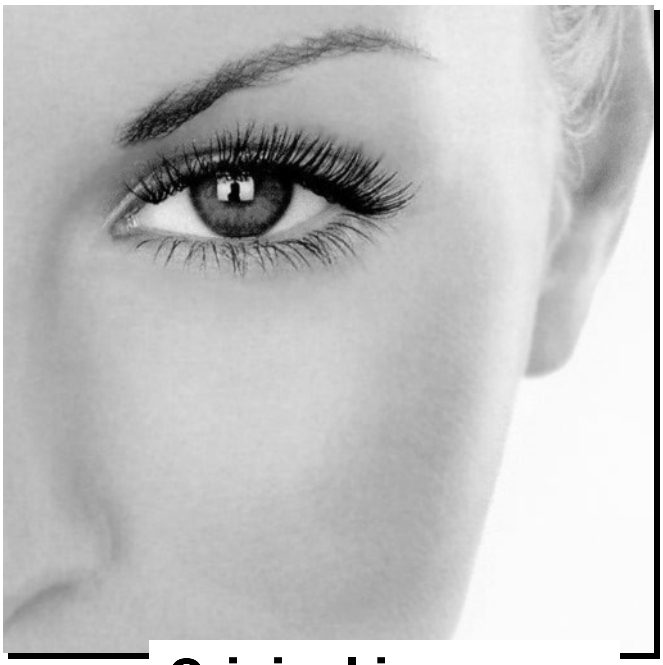


It is an obvious fact that the image quality would be higher for less compressed images. An approximation of this can be observed at Zeros vs level graph shown alongside. It can be seen that at level 90, the compression is less and at level 5 quality, the compression is seen very high.



From the PeakSNR vs level graph, it is quite evident how PeakSNR gets degraded due to high compression.

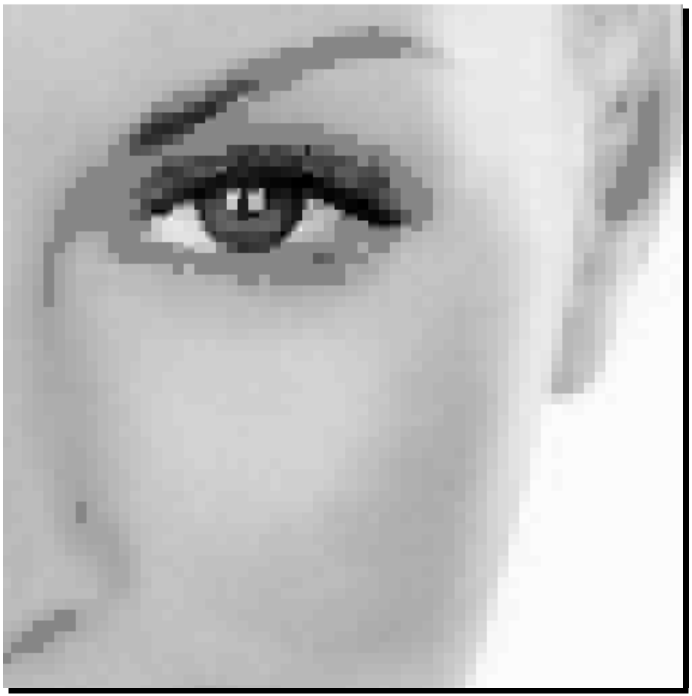
3.2.3 Linear vs nonlinear quantiser

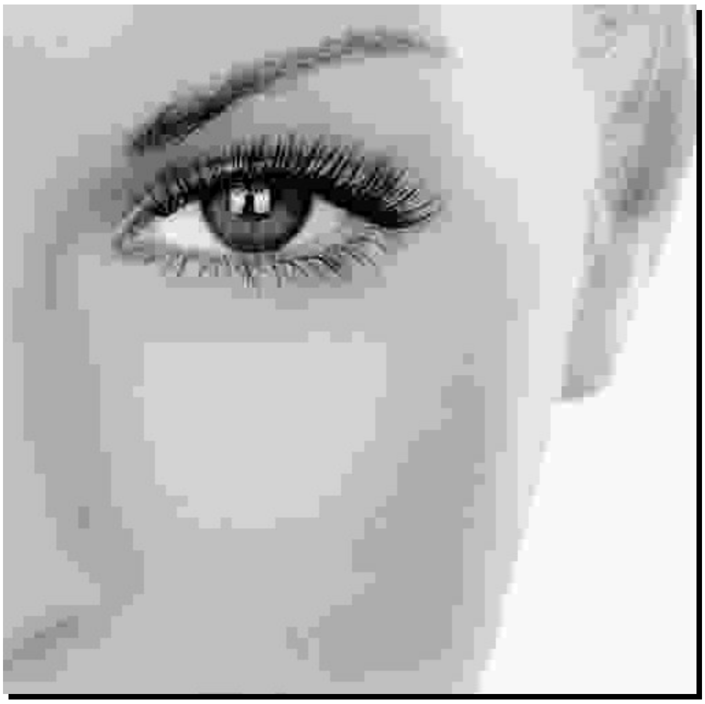


Original image

Linear quantised image

quantised





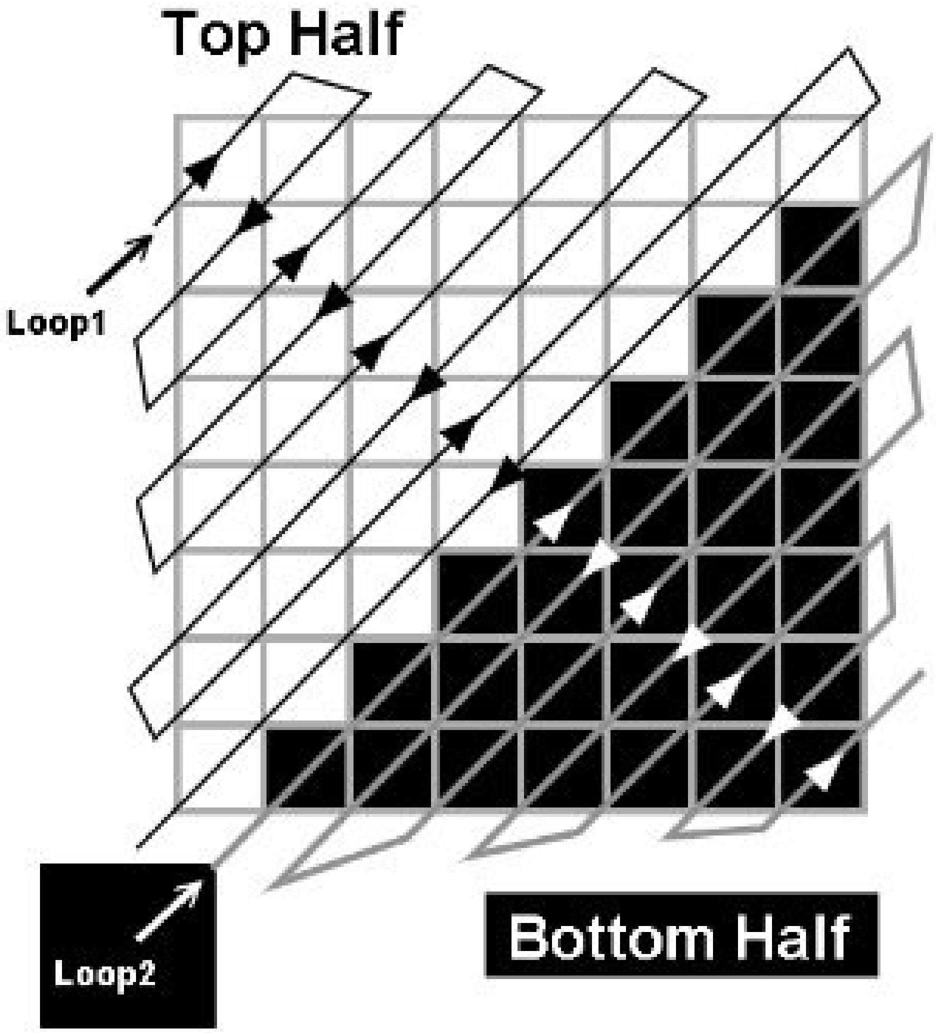
From the above test images, the basic difference between linear and non-linear quantisation is evident. As in linear quantisation the step sizes are equal, the whole part of the image gets degraded. When we analyse the linear quantised image, we can see that the part of eye(darker areas) is degraded equally as the other parts are.

For non-linear quantisation the step size varies, and hence we can see parts of the eye(darker area) preserved better than in the linear quantised image though it comes at the cost of other areas which gets degraded more(lighter areas).

4 Zig-Zag encoder and Zig-Zag decoder

4.1 Implementation notes

4.1.1 Encoder



The encoder code treats the whole matrix as two, the top half and bottom.

As shown in the figure, matrix scanning starts from the top using an upward-loop. When it reaches the edge signified by y = 1 , the code enters loop ‘downward-loop’ by which the matrix is scanned downwards diagonally. The left edge of the matrix is identified by checking for x = 1 and when it’s reached, the code enters the upward loop again and the process continues until top half is complexly finished.

The same logic is utilised for the lower section with just the difference that the right edge and bottom edge is signified by x > width and y> width respectively.

4.1.2 Decoder

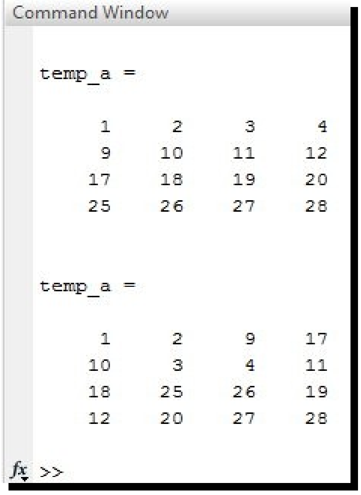
The decoder basically backtracks from the functionality of the encoder.

Firstly the decoder creates a temporary matrix of same dimensions of the input matrix. The temporary matrix would be filled with incrementing values which is shown as the first matrix in the figure alongside.

Then to identify how each element gets displaced by zigzag scanning, a zigzag encoding is performed as shown by the second matrix in the figure.

Then by scanning the new temporary matrix(zigzag encoded one) the actual position of each element is found out.

For example, (as seen in the snapshot) when the element with value 9 in first row is scanned, we can infer that the 3rd element of the input matrix originally belong to 9th place. So the 3rd element of the input matrix is picked and placed as the 9th element of the output matrix. When the next element in the temporary matrix '17' is scanned, it means that the 4th element of the input matrix belongs to 17th position and hence the 4th element is picked from input matrix and placed at 17th position of the output matrix.



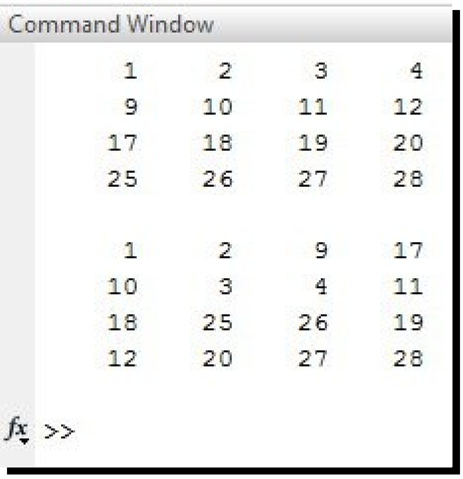
```
temp_a =

     1     2     3     4
     9    10    11    12
    17    18    19    20
    25    26    27    28


temp_a =

     1     2     9    17
    10     3     4    11
    18    25    26    19
    12    20    27    28

fx >>
```

4.2 Testing and results

4.2.1 Encoder

The encoder was tested with a simple code which creates a 4x4 matrix with incrementing values. Then the encode function is applied to the matrix.

In the snapshot of the command window provided alongside, it can be seen that the module works as expected.



```
     1     2     3     4
     9    10    11    12
    17    18    19    20
    25    26    27    28

     1     2     9    17
    10     3     4    11
    18    25    26    19
    12    20    27    28

fx >>
```

4.2.2 Decoder (Combined test for encoder and decoder)

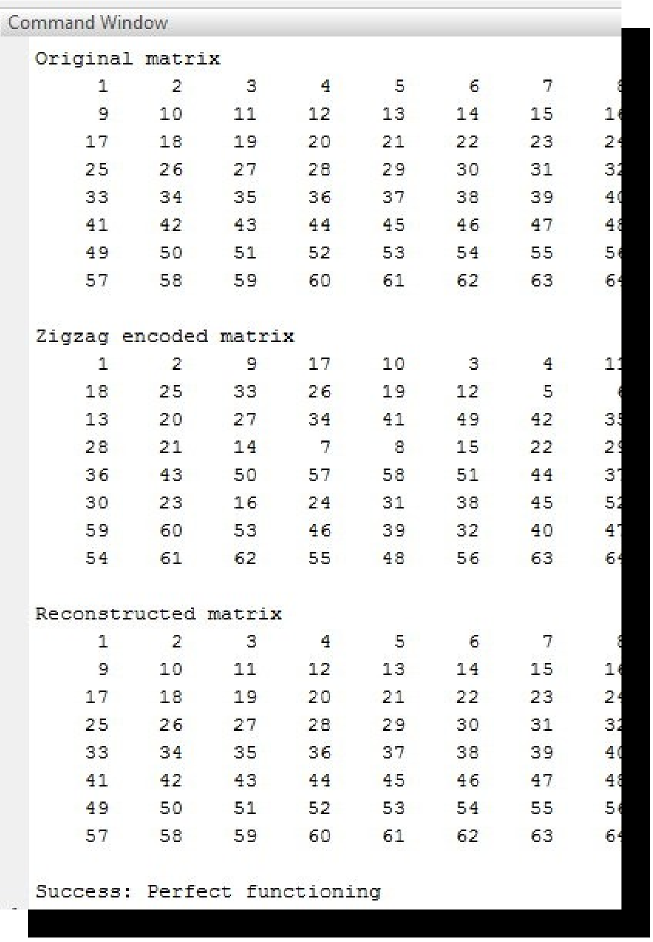
For testing the decoder, first a matrix of 8x8 is made with incrementing values as seen in the snapshot alongside.

Then it is encoded using the zigzag encoder and the resulting output is also displayed.

Then the result is passed through zigzag decoder module and the result is obtained.

Using ‘isequal’ function in matlab, the final result is compared to the initial matrix and found to be the same.

The accuracy can also be verified visually by looking for perfectly incrementing values throughout the final matrix.



```
Original matrix
     1     2     3     4     5     6     7     8
     9    10    11    12    13    14    15    16
    17    18    19    20    21    22    23    24
    25    26    27    28    29    30    31    32
    33    34    35    36    37    38    39    40
    41    42    43    44    45    46    47    48
    49    50    51    52    53    54    55    56
    57    58    59    60    61    62    63    64

Zigzag encoded matrix
     1     2     9    17    10     3     4    11
    18    25    33    26    19    12     5     6
    13    20    27    34    41    49    42    35
    28    21    14     7     8    15    22    29
    36    43    50    57    58    51    44    37
    30    23    16    24    31    38    45    52
    59    60    53    46    39    32    40    47
    54    61    62    55    48    56    63    64

Reconstructed matrix
     1     2     3     4     5     6     7     8
     9    10    11    12    13    14    15    16
    17    18    19    20    21    22    23    24
    25    26    27    28    29    30    31    32
    33    34    35    36    37    38    39    40
    41    42    43    44    45    46    47    48
    49    50    51    52    53    54    55    56
    57    58    59    60    61    62    63    64

Success: Perfect functioning
```

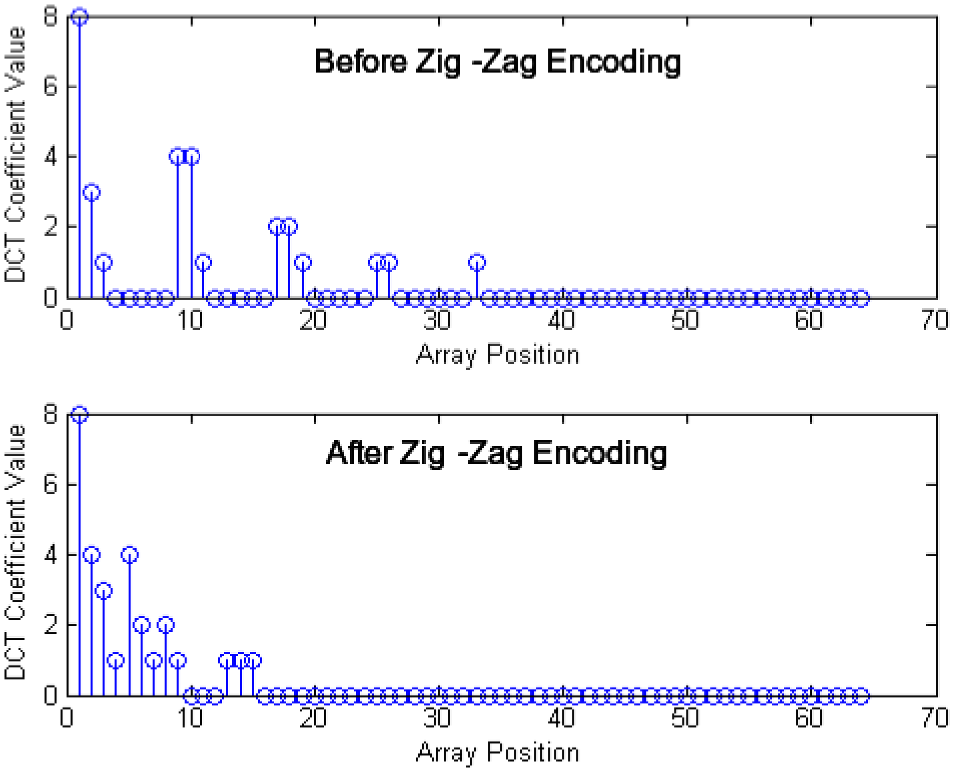


Figure 4.7 - Stem plot - Before vs.After ZigZag Encoding

Figure 4.1 shows the result of a zig-zag encoding test run. It is clear that the zig zag coding attempts to 'sort' the array, so that the high amplitude coefficients are grouped together.

5 Differential Pulse Code Modulation (DPCM)

- Just as the AC-components were zigzag scanned, the DC-components need to be Differential coded seperate to the AC-coefficients.
- This seperate treatment from the AC coefficients is to exploit the correlatin between DC values of **adjacent blocks** and to encode them more efficiently.[2]
- The DC coefficients typically contain the most amount of energy of the image/block.

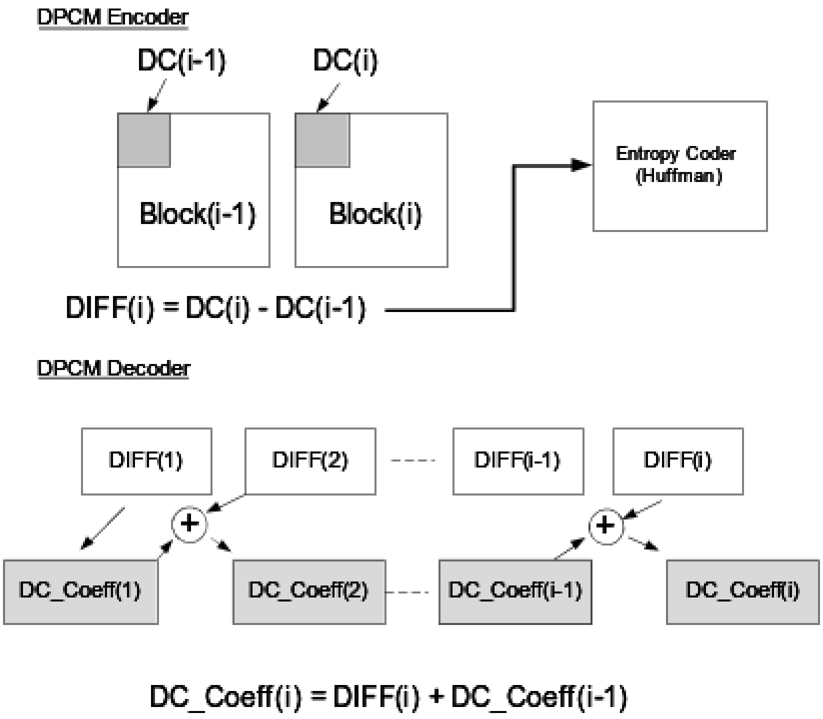


Figure 5.8 - Differentially encoding the DC-coefficients, (adapted from [2])

- The matlab functions **dc_dpcm_enc()** and **dc_dpcm_dec()** were written to perform DPCM encoding/decoding respectively
- The first block is differentially encoded against zero.
- The resulting differential array is then fed into the Entropy Coder (Huffman).
- The DPCM decoder exactly reverses the process of the encoder.

5.1 Testing and Results

The following is a test run of the dpcm encoder/decoder. The results are

```
dc =    10    8    8    9    9   10   10   11   13   11    5   -5  -12  -11  -11  -11

>> diffs = dc_dpcm_enc(dc)
diffs =    10   -2    0    1    0    1    0    1    2   -2   -6  -10   -7    1    0    0
```

```
>> dc_reconst = dc_dpcm_dec(diffs)
dc_reconst =    10    8    8    9    9   10    10    11    13    11     5    -5   -12   -11   -11   -11
```

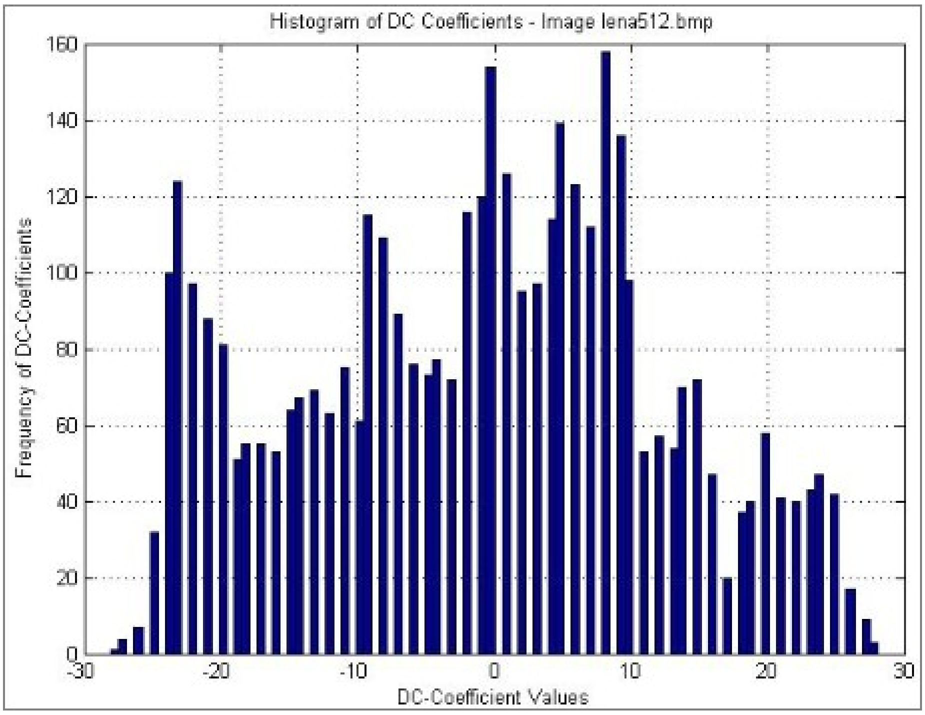


Figure 5.9 - Histogram of DC Coefficients, before DPCM encoding

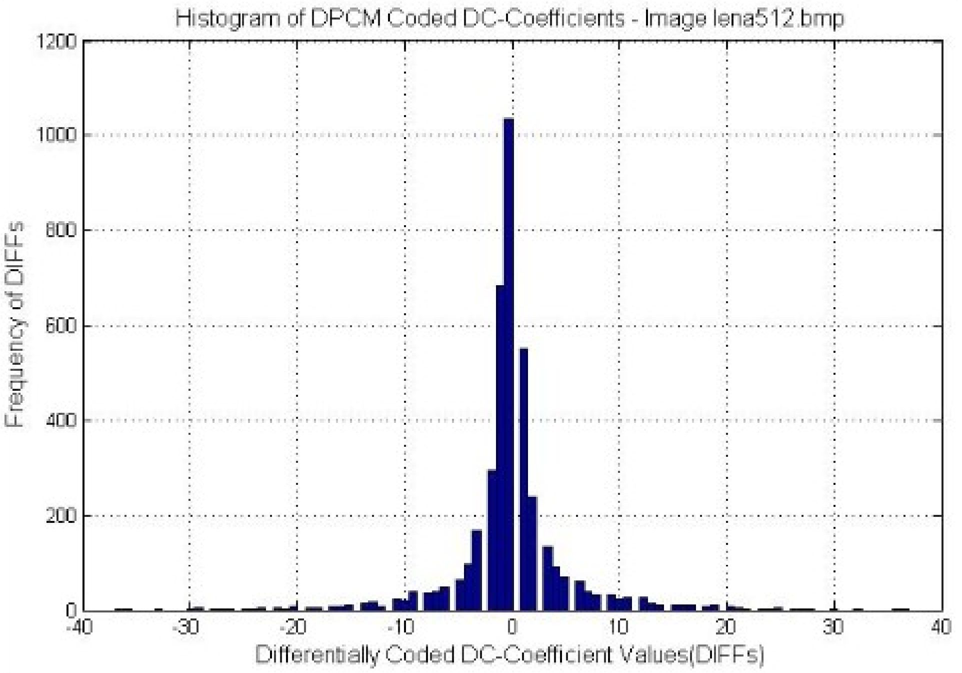


Figure 5.10 - Histogram of Differentially encoded DC-Coefficients (DIFFs)

- It is clear from Figures 5.2 and 5.3 that after DPCM encoding, the differential values will enable more efficient entropy coding.
- After DPCM coding the histogram will peak around zero .
- The entropy of the difference signal- DIFF is much smaller than that of the original discrete signal treated as independent samples.

-

-

6 Run-length and Huffman Encoder/Decoder

The Huffman Encoder/Decoder section is consisted of two parts: DC Huffman Encoder/Decoder and AC Huffman Encoder/Decoder.

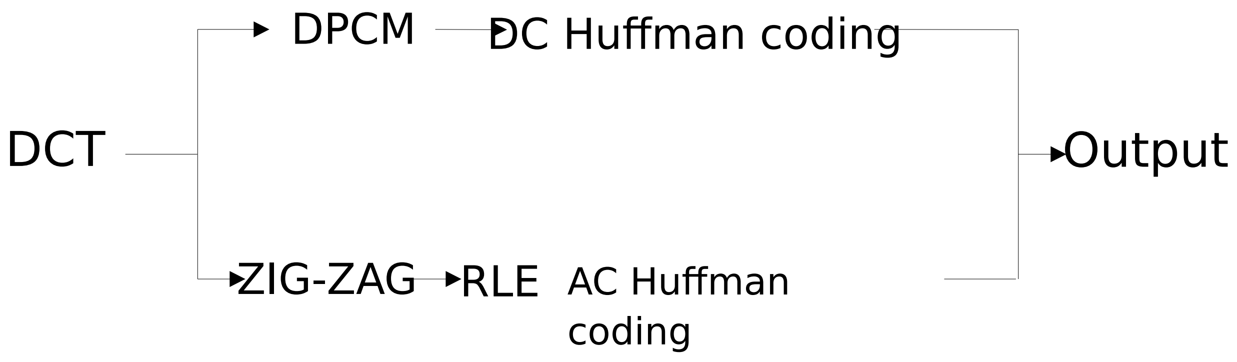


Figure6.1- Flowchart of Huffman Encoding

For each block, the output should have 3 parts which are DC component, AC component and an EOB symbol. After decoding process, the sequence should be exact as the output of Zig-Zag scanning.

6.1 DC Huffman

6.1.1 DC Huffman Encoding

The input of DC Huffman Encoding is the DC components sequence after DPCM. The first step is to judge whether the current DC component is 0: if it is then add it directly to the encoding sequence, if it isn't then first get the category which represent how many bits the number takes in the base code. And get the recognize code through the DC-Look-up table (see Appendix). The final code in output would be in '0' and '1' form, the first several bit should match the recognize code in look-up table, after that is the exact value.

Test example:

The input DC sequence is DC = [12 34 22 6 76 0 77 5 0 0 354]

```
>> dc_huffman(DC)

ans =

101110011101000101101011010011011110100110000111101001101100101000011111
10101100010
```

6.1.2 DC Huffman Decoding

For the DC Huffman Decoding part, the basic is to find the recognize code and get category number first and calculate the original value out. Then do the cycle from the next bit to find next recognize code.

The DC Decoding-Look-Up table can be found in Appendix.

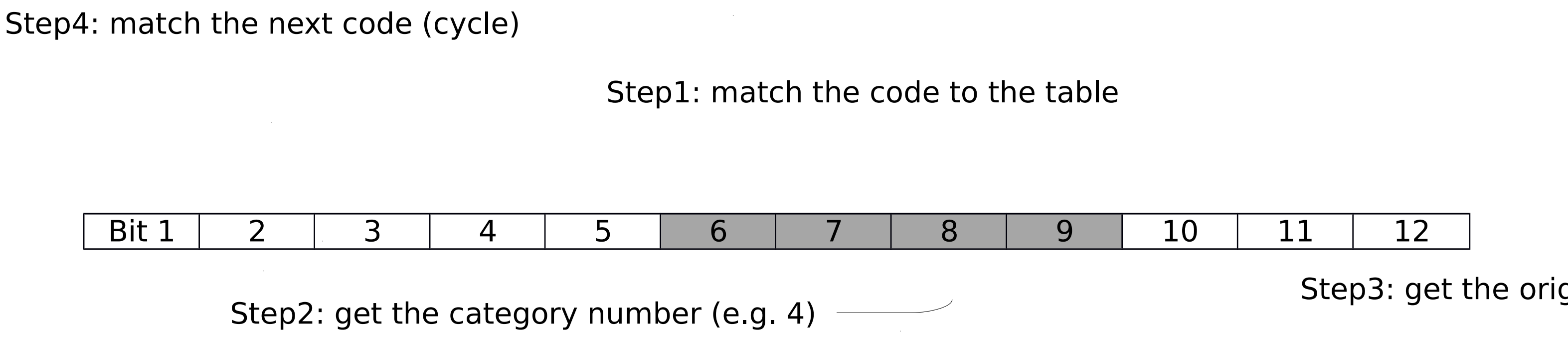

Step4: match the next code (cycle)
Step1: match the code to the table
Bit 1
2
3
4
5
6
7
8
9
10
11
12
Step3: get the orig
Step2: get the category number (e.g. 4)

Figure 6.2 – DC decoding algorithm

Take the sequence generated in encoding part to do the test:

```
>>
huffman_dc_decoding('101110011101000101101011010011011110100110000111101001
1011001010000111110101100010')

ans =

  12    34    22     6    76     0    77     5     0     0   354
```

From which we can see the decoding function (see Appendix) finished correctly.

6.2 AC Huffman

Different from the DC part, every block have 63 continuous AC components, and before doing the Huffman Encoding a Run-Length Coding is needed.

6.2.1 Run-Length Coding

Run –Length Encoding is a very simple form of data compression in which runs of data are stored as a single data value and count, rather than as the original. In this lab assignment, we would record the runs of number ‘0’. For example if there are N continuous ‘0’ appear in the sequence followed with a non-zero number M, the form would be like (N, M). A special case is when the maximum of N is 15, so when there are 17 zero, the form is (15, 0), (1, M) and in ‘34 zero’ case it is (15, 0), (15, 0), (2, M).

Turn it into the coding, the flow chart would be:

al DC value

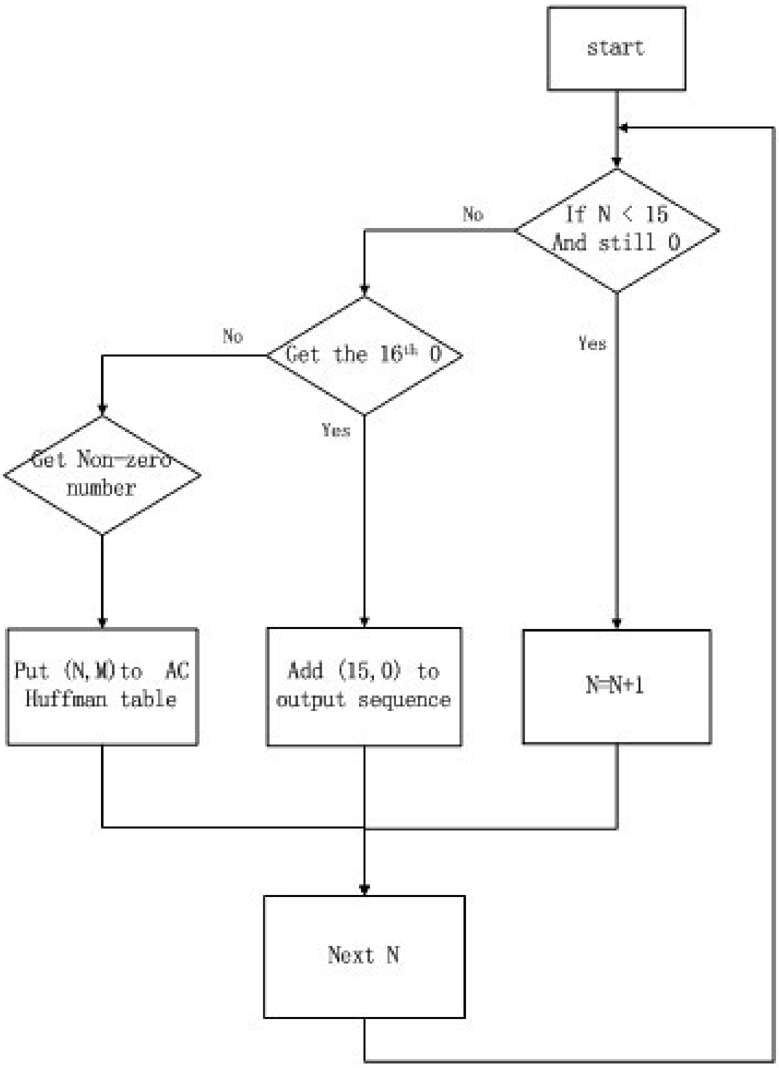


Figure 6.3 - the flow chart of Run Length Coding

The vector used in test is test: [23 -17 1 -2 0 0 -14 5 3 -1 -1 2 -2 -3 -11 27 19 2

-1 2 0 -1 1 0 0 1 4 5 0 -2 -4 0 0 -1 1 0 0 -1 0 0 1 -2 1 1 0

-1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

The result is (from the second number is the AC part):

```
>> ac_huffman(test)
```

Part1:
pair =   0   -17 --------------------- -17
pair =   0   1   --------------------- -1
pair =   0   -2 --------------------- -2
pair =   2   -14---------------------   0   0   -14
....

We put all the result in a table and do the comparison with the original sequence:

Table 6-1: RLC output

Vector	-17	1	-2	0	0	-14
pair	(0,-17)	(0,1)	(0,-2)	(2,-14)		

Vector	5	3	-1	-1	2	-2
pair	(0,5)	(0,3)	(0,-1)	(0,-1)	(0,2)	(0,-2)

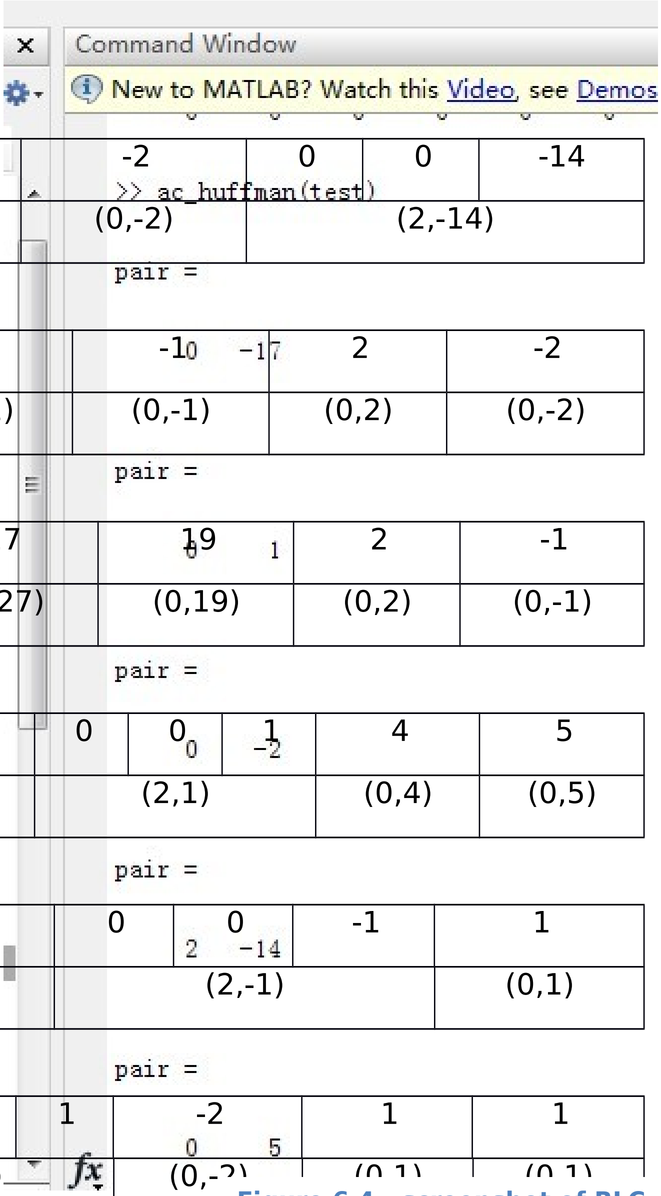
Vector	-3	-11	27	19	2	-1
pair	(0,-3)	(0,-11)	(0,27)	(0,19)	(0,2)	(0,-1)

Vector	2	0	-1	1	0	0	1	4	5
pair	(0,2)	(1,-1)		(0,1)	(2,1)			(0,4)	(0,5)

Vector	0	-2	-4	0	0	-1	1
pair	(1,-2)		(0,-4)	(2,-1)			(0,1)

Vector	0	0	-1	0	0	1	-2	1	1
pair	(2,-1)			(2,1)			(0,-2)	(0,1)	(0,1)

Vector	0	-1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
pair	(1,-1)		(1,1)		(15,0)															



```
>> ac_huffman(test)

pair =

     0   -17

pair =

     0     1

pair =

     0    -2

pair =

     2   -14

pair =

     0     5
```

Figure 6.4 - screenshot of RLC output

Take a piece of sequence ‘0, 0, -14’ as example, the original length is 8*3=24 bits, after the RLC the code would be ‘11111011000011010’ and only takes 17 bits. But when we increase the number of ‘0’ to 10, the length would be reduced from 90 to 24. It means with the more continuous 0s, the effect of compression is more obvious after RLC.

6.2.2 AC Huffman Encoding

The basic steps of AC Huffman Encoding are the same as DC Huffman Encoding with its own Look-up table. The only difference is in AC Huffman Encoding, we need to put the runs of zeros as well.

Still with the ‘test’ vector, the output of AC Huffman Encoding is:

```
>> ac_huffman(test)

Part2:

ans =

1101001110001010111111111110001001000110010101110000000110010101001011010
011010110111101010011011000001101100000111011110010010010111100101100011
1101100011101101101110101001001110001100111111111111111111111010

>>
```

6.2.3 AC Huffman Decoding

For the decoding aspect, the output of the AC decode Look-up table would also in pairs, the first value is the runs of number’0’ and the second is the category of the following non-zero number. So the only step needed to add on the DC decoding process is to return the ‘0’s to the sequence according to the runs. And one more special situation: once get the symbol of EOB which means filling the rest of the sequence with ‘0’s.

Form the binary sequence of encoding result we can do the inverse transformation:

```
>>
huffman_ac_decoding('1101001110001010111111111110001001000110010101110000000
110010101001011010011010110111101010011011000001101100000111011110010010
010111100101100011110110001110110110111010100100111000110011111111111111
11111010')

ans =

  Columns 1 through 17
   -17     1    -2     0     0   -14     5     3    -1    -1     2    -2    -3   -11    27    19     2
  Columns 18 through 34
    -1     2     0    -1     1     0     0     1     4     5     0    -2    -4     0     0    -1     1
  Columns 35 through 51
     0     0    -1     0     0     1    -2     1     1     0    -1     0     1     0     0     0     0
  Columns 52 through 63
     0     0     0     0     0     0     0     0     0     0     0     0
```

Which is exactly the AC part of ‘test’ vector.

6.3 Efficiency evaluation

We use a picture to test the effect of Huffman Encoding/Decoding without quantisation (stepsize=1,cutoff=0)

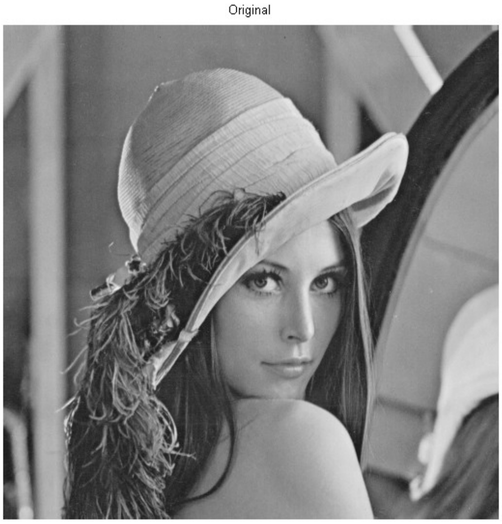


Figure 6.5 - Lena512.bmp (original)

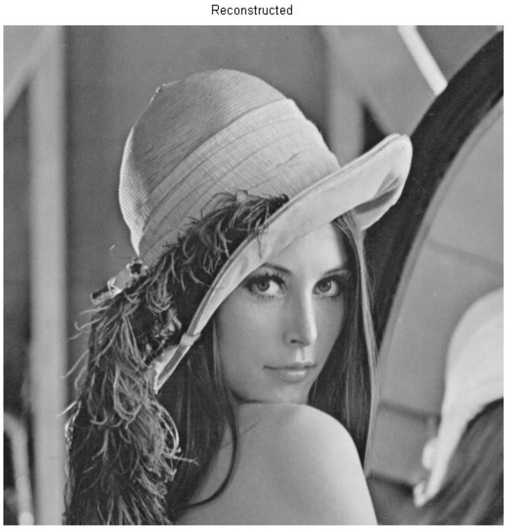


Figure 6.6 - Lena512.bmp (after Huffman Coding)

The image size was 2097152 bits and after Huffman coding it reduced to 1294191 (AC-codewords bit length + DC codewords bitlength)

The bpp(bits per pixel) is 4.9369 and the compression ratio is 38.2882 by Huffman Coding. Hence even without quantisation, the huffman encoding plays a significant role in compression(no loss of data)

7 Overall evaluation of results

Several tests were performed on the MPEG Encoder/Decoder system. The Main objective was to observe what kind of compression was able to achive at different quantisation levels.

The tests were performed on different images – high frequency images and low frequency images. The Test images can be seen in Appendix-J

Lena512.bmp (512x512) – a smooth area around the shoulder of the person, but sharp edges near her hair.

Cameraman256.pgm (256x256) – Smooth area on the sky, but buildings are high freq.

Barb256.pgm (256x256) – Overall the image has many high frequency areas – table cloth, shawl, trouser etc.

Xilinx.bmp (512x512) – certain areas are smooth (near the logo), but the chop pins are sharp.

Child.bmp (512x512) – hair is high frequency, face is lower frequency.

Smooth.bmp (512x512) – overall the image has many low frequency areas.

Figure 7.1 shows the tests that were performed. Both Non-linear and Linear quantization was explored.

Original File Size = width* height * (8-bits)

Compressed Image Size = Huffman Coder output (AC-coded bit length + DC Coded bit length) (size of huffman tables not taken into account)

Compression Ratio = 100 - ((Compressed_Size/Original_Size)*100)

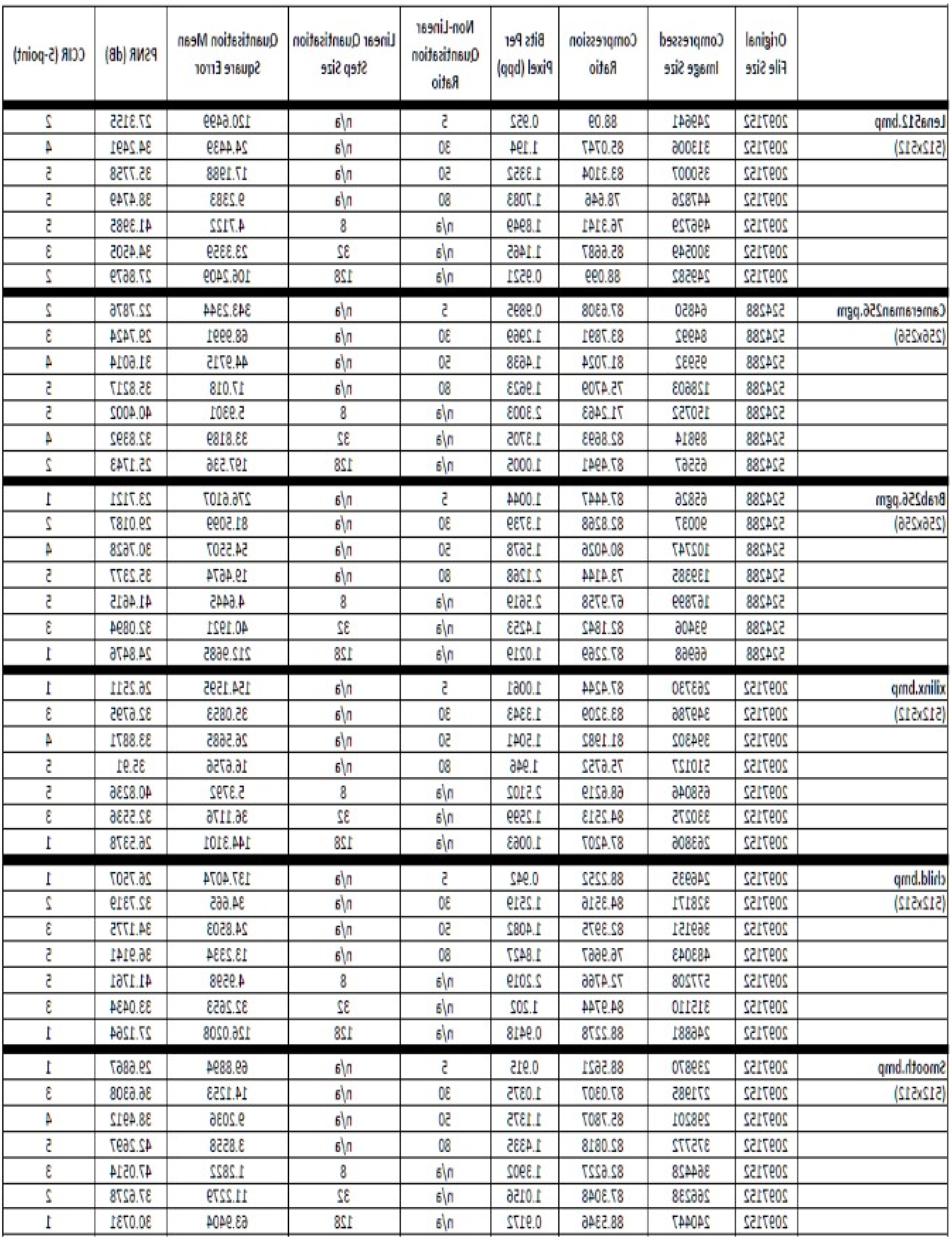
Bits Per Pixel = (Compressed Image Size)/(width*height)

Non-Linear Quantisation Ratio = The Q-factor which gets multiplied by the Q50 Quantisation Matrix

Linear-Quantisation Step Size = The step size of the linear quantizer (number of divisions)

Quantisation Mean Suqare Error = (2D Summation of the Quantisation Error^2)/(width*height)

PSNR = Peak Signal to Noise Ratio = 10*log(255^2)/MSE



	Original File Size	Compressed Image Size	Compression Ratio	Bits Per Pixel (bpp)	Non-Linear Quantisation Ratio	Linear Quantisation Step Size	Quantisation Mean Square Error	PSNR (dB)	CCIR (5-point)
Lena512.bmp	2097152	249641	88.09	0.952	5	n/a	120.6499	27.3155	2
(512x512)	2097152	313006	85.0747	1.194	30	n/a	24.4439	34.2491	4
	2097152	350007	83.3104	1.3352	50	n/a	17.1988	35.7758	5
	2097152	447826	78.646	1.7083	80	n/a	9.2383	38.4749	5
	2097152	496729	76.3141	1.8949	n/a	8	4.7122	41.3985	5
	2097152	300549	85.6687	1.1465	n/a	32	23.3359	34.4505	3
	2097152	249582	88.099	0.9521	n/a	128	106.2409	27.8679	2
Cameraman256.pgm	524288	64850	87.6308	0.9895	5	n/a	343.2344	22.7876	2
(256x256)	524288	84992	83.7891	1.2969	30	n/a	68.9991	29.7424	3
	524288	95932	81.7024	1.4638	50	n/a	44.9715	31.6014	4
	524288	128603	75.4709	1.9623	80	n/a	17.018	35.8217	5
	524288	150752	71.2463	2.3003	n/a	8	5.9301	40.4002	5
	524288	89814	82.8693	1.3705	n/a	32	33.8189	32.8392	4
	524288	65567	87.4941	1.0005	n/a	128	197.536	25.1743	2
Brab256.pgm	524288	65826	87.4447	1.0044	5	n/a	276.6107	23.7121	1
(256x256)	524288	90037	82.8268	1.3739	30	n/a	81.5099	29.0187	2
	524288	102747	80.4026	1.5678	50	n/a	54.5507	30.7628	4
	524288	139385	73.4144	2.1268	80	n/a	19.4674	35.2377	5
	524288	167899	67.9758	2.5619	n/a	8	4.6445	41.4615	5
	524288	93406	82.1842	1.4253	n/a	32	40.1921	32.0894	3
	524288	66968	87.2269	1.0219	n/a	128	212.9685	24.8476	1
xilinx.bmp	2097152	263730	87.4244	1.0061	5	n/a	154.1595	26.2511	1
(512x512)	2097152	349786	83.3209	1.3343	30	n/a	35.0853	32.6795	3
	2097152	394302	81.1982	1.5041	50	n/a	26.5685	33.8871	4
	2097152	510127	75.6752	1.946	80	n/a	16.6756	35.91	5
	2097152	658046	68.6219	2.5102	n/a	8	5.3792	40.8236	5
	2097152	330275	84.2513	1.2599	n/a	32	36.1176	32.5536	3
	2097152	263806	87.4207	1.0063	n/a	128	144.3101	26.5378	1
child.bmp	2097152	246935	88.2252	0.942	5	n/a	137.4074	26.7507	1
(512x512)	2097152	328171	84.3516	1.2519	30	n/a	34.665	32.7319	2
	2097152	369151	82.3975	1.4082	50	n/a	24.8503	34.1775	3
	2097152	483043	76.9667	1.8427	80	n/a	13.2334	36.9141	5
	2097152	577208	72.4766	2.2019	n/a	8	4.9598	41.1761	5
	2097152	315110	84.9744	1.202	n/a	32	32.2653	33.0434	3
	2097152	246881	88.2278	0.9418	n/a	128	126.0208	27.1264	1
Smooth.bmp	2097152	239870	88.5621	0.915	5	n/a	69.8894	29.6867	1
(512x512)	2097152	271985	87.0307	1.0375	30	n/a	14.1253	36.6308	3
	2097152	298201	85.7807	1.1375	50	n/a	9.2036	38.4912	4
	2097152	375772	82.0818	1.4335	80	n/a	3.8558	42.2697	5
	2097152	364428	82.6227	1.3902	n/a	8	1.2822	47.0514	3
	2097152	266238	87.3048	1.0156	n/a	32	11.2279	37.6278	2
	2097152	240447	88.5346	0.9172	n/a	128	63.9404	30.0731	1

Figure 7.11 - Test Sheet for JPEG/MPEG Encoding-Decoding - 6 images

Below is the key to the different graphs:

Code	Quantisation type
NL-5	Non-Linear quantisation with level 5
NL-30	Non-Linear quantisation with level 30
NL-50	Non-Linear quantisation with level 50
NL-80	Non-Linear quantisation with level 80
L-8	Linear quantisation with step size 8
L-32	Linear quantisation with step size 32
L-128	Linear quantisation with step size 128

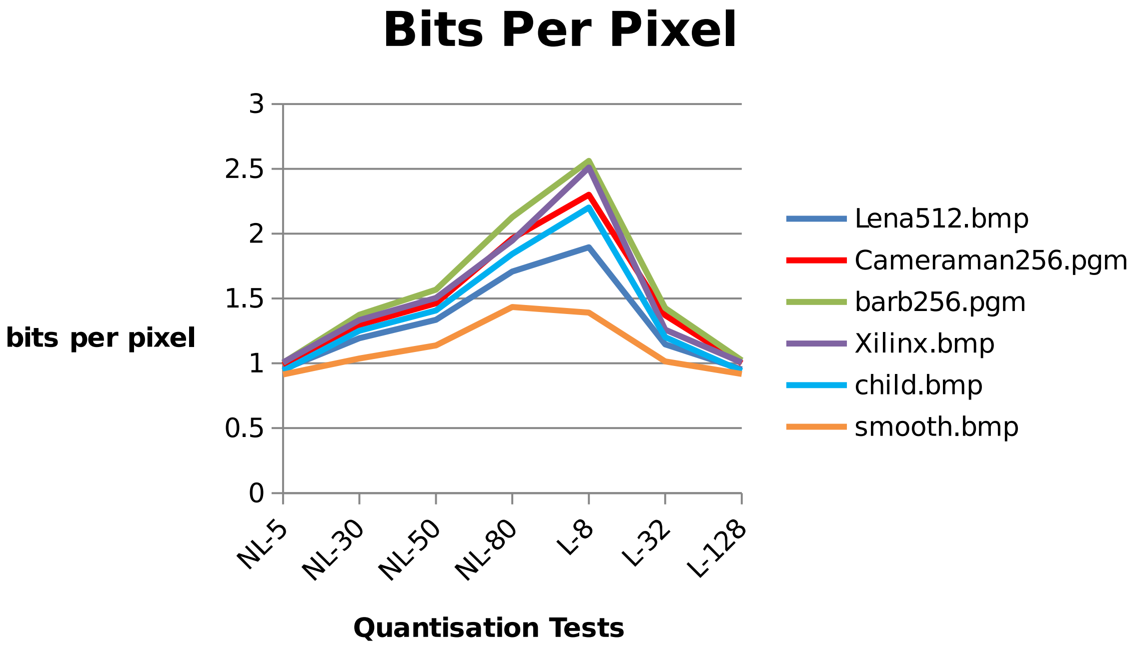


Figure 7.12 - The line graph of bpp (bits per pixel)

Analysis: The index of bpp represent the number of bits used to represent the color of a single pixel in a bitmapped image. The lower the index gets the better result it means in the compression. The figure indicated that in all condition the ‘smooth.bmp’ had a better performance and ‘barb256.pgm’ did not well. The reason is in the later picture there are lots of abrupt tonal transitions in a small space like the plaid dress, while in the picture of ‘smooth.bmp’ the tone remains relatively constant throughout a large area.

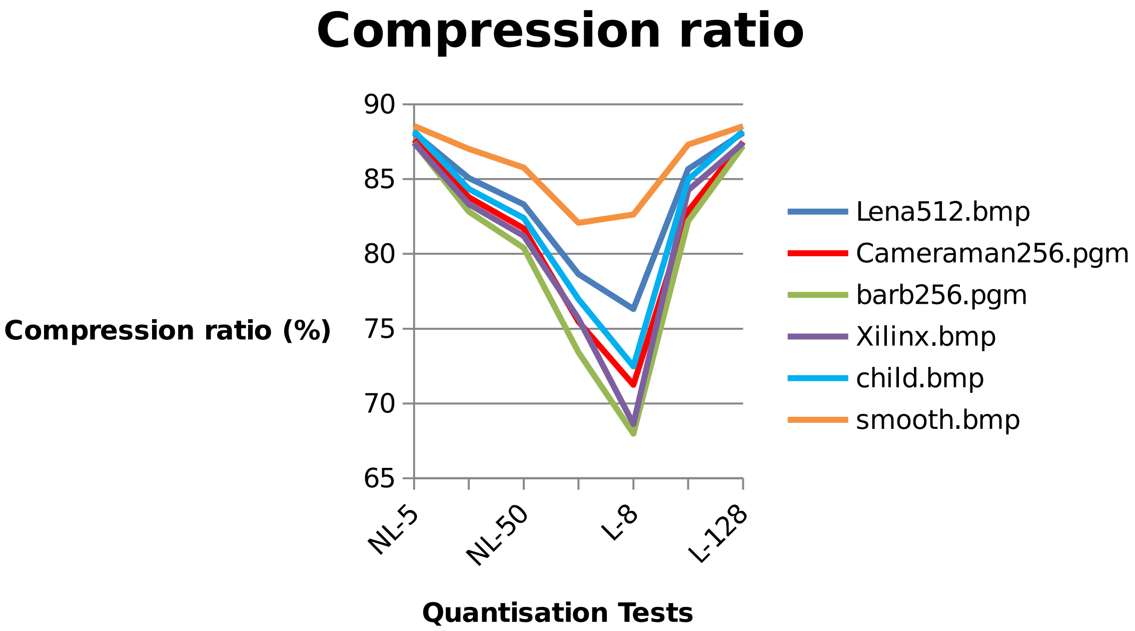


Figure 7.13 – The line graph of Compression ratio

Analysis: Obviously, the higher compression ratio means more space been saved, but also imply the probability of more information lost. The result of this item is similar as the former bpp one. And the conclusion can be drawn that the complex picture needs more space while doing compression. And there are two ways to increase the compression ratio: first one is to have a low quality ratio and the second is to increase the stepsize.

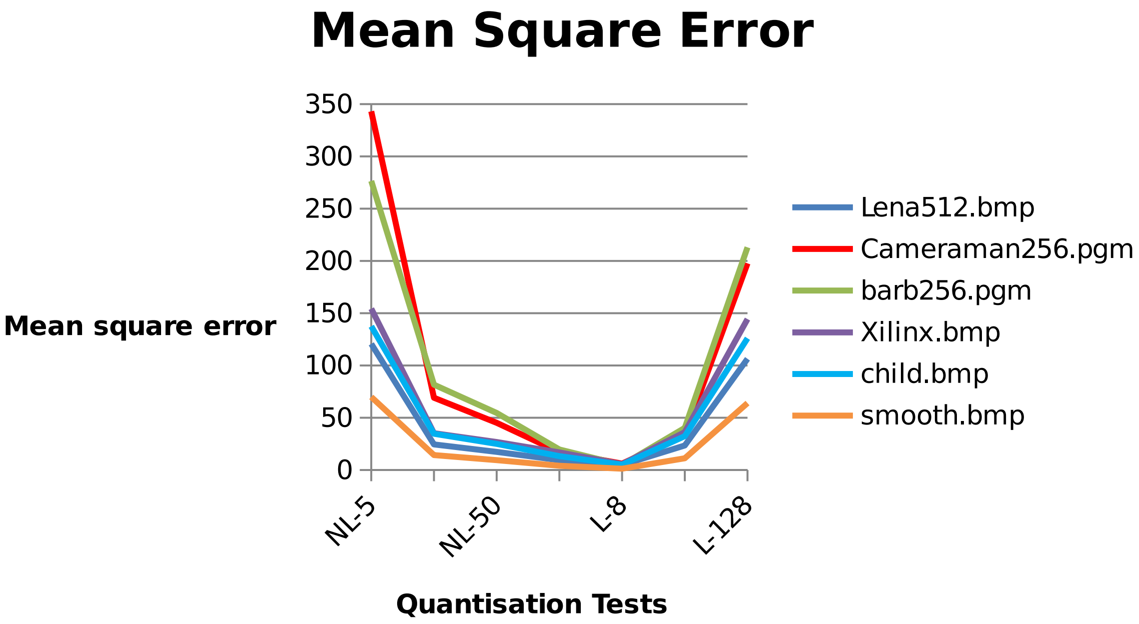


Figure 7.14 – The line graph of Mean Square Error

Analysis: With the higher compression ratio, the numbers of mean square error are increasing. Also it is depends on the complex level(frequency/shard edges) of the original picture. The simple picture (like smooth.bmp) has the lowest error number among six even in the highest compression ratio.Cameraman256.pgm and barb256.pgm seems to be the worse performing two images.

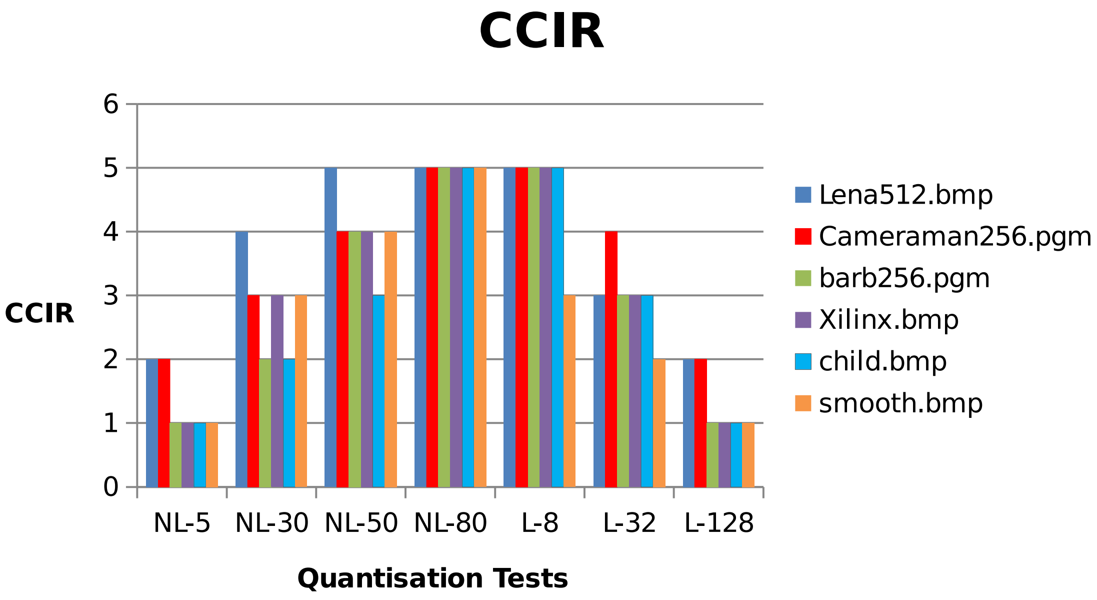


Figure 7.15 – General CCIR

This figure shows a general impression of the six pictures after different compression process. The value in Y-axis represents the idea of the quality after processing: ‘5’ is excellent (imperceptible), ‘4’ is good (perceptible, but not annoying), ‘3’ is fair (slightly annoying), ‘2’ is poor (annoying) and 1 is bad (very annoying). From the perspective of quality of the picture after compression, the one with less compression ratio always have better performance. But since the arm of the process itself is to reduce the space of pictures, we must take a consideration and make a balance when doing the compression. From observing the figure we can see that for the simple and smooth picture, even in a high compression ratio, it would still remain a relative good quality. So the complexity of the picture might be a consideration when we choose the method of doing the compression.

8 Conclusion

Goals achieved.

This report introduced the different sub-sections of the JPEG/MPEG-2video compression standard. It explained in detail how the different sub-sections can be implemented(algorithm), simulations were performed and an analysis of the results were made.

The Primary goals of the project was to gain better understanding of the MPEG-2 Compression standards (for still images – iframes). Considerable research was done to understand each sub-section of the entire system. Certain limitations and advanatges of the system was also identified through the process.

The second and third goals were to simulate the encoder/decoder system using Matlab, and then to implement the actual system in hardware (FPGA). The algorithms for each sub-section was successfully developed in Matlab and each component was tested thoroughly. However the final goal of implementing the system in hardware was not achieved due to time constraints.

Limitations

Several limitations in the algorithm are evident – the major one being – the speed at which the compression is performed. The DCT and Huffman coding process takes a considerable amount of time. The DCT uses many multiplications, additions and many loops. Perhaps if a 32 wide Fast DCT was able to perform, then the speed could have been increased further. Huffman coding and decoding – could have been implemented using a binary tree – which is proved to be faster than using lookup tables and reverse-lookup tables. The data in the tables can be stored in a binary tree, and traversed recursively.

Issues with Implementing on Hardware

One major implementation hurdle would be to implement the DCT and Huffman Encoder in hardware – as both components will take up large amount of resources – in terms of processing speed (CPU cycles) and memory (to store the tables). However the FAST-DCT method seems a good alternative to using basic transforms. One other design stratergy would be to pipeline the different sub-components, to achieve a good throughput.

Summary of results

The results obtained from the tests show that this compression standard works well for images with many low-frequency areas (eg: smooth.bmp), but does not perform well for ‘busy/complex’ images. The quantisation plays a large role in compression, and can be adjusted depending on the application (eg: web viewing, print media etc). However it was impressive to realise that even with no-quantisation, the huffman coder was able to reduce the size of the image by 38%. So it is very clear that both quantisation and huffman coding are directly involved in this compression system. There are several parameters of the compression system that can be adjusted depending on the application – therebye the system can be used as an efficient compression technology.

9 References

[1] J.Dell – “Advanced Multimedia Applications”, Lecture notes, University of York, 2011.

[2] M.Ghanbari – “Standard Codecs: Image Compression to Advanced Video Coding”, IEE Telecommunications Series 49, 2003

[3] R.C Gonzalez, R.E Woods - "Digital Image Processing”, Prentice Hall(2008)

[4] A. Tarczynski, “Broadcast Media Systems – image Compression, Introduction to JPEG”, Lecture notes, University of Westminster, 2007.

[5] H. Kumar, A. Pundir "Image compression Using Discrete Cosine Transform Implimenting Matlab (Project report)",NIT H.P.,India,2008

[6] Sanjeev Mehta, http://www.coderanch.com/t/485470/java/java/zigzag-traverse-matrix, accessed 25/06/2011

10 Appendix

Contents of the Apppendix:

10.1 Appendix-A

10.1.1 Top Level JPEG/MPEG Encoder-Decoder System

```
%=================================================================================
% Advanced Multimedia Applications
% Title      : JPEG/MPEG Encoding-Decoding - Top Level Test Script
%              (MPEG_Test.m)
% Description: Performs JPEG Encoding and Decoding. The parameter section
%              will contain various input parameters to the different sub
%              sections
%              Sub-sections:
%              --------------
%               Encoder:
%                   - DCT (2D)
%                   - Quantisation
%                   - ZigZag Scanning (AC-coeffs), DPCM (DC-coeffs)
%                   - RLC and VLC (Huffman)
%
%               Decoder:
%                   - Huffman Decoding
%                   - Reverse ZigZag(AC-coeffs), DPCM-Decoding(DC-coeffs)
%                   - Inverse-Quantisation
%                   - IDCT (2D)
%=================================================================================

clc;clear;close all;

%=====================================================================
%       T E S T  -  S C R I P T   P A R A M E T E R S
%=====================================================================

% -- image params (uncomment/comment as required) --
file_ext = 'bmp';  % currently supports : bmp, pgm

%filename = 'lena512.bmp';          %[Test-image-1]
%filename  = 'cameraman256.pgm';    %[Test-image-2]
%filename = 'barb256.pgm';          %[Test-image-3]
filename  = 'xilinx.bmp';          %[Test-image-4]
%filename  = 'child.bmp';           %[Test-image-5]
%filename  = 'smooth.bmp';          %[Test-image-6]

% -- DCT params --
speed   = 1;     % 0-simple dct, 1-fast dct

% -- Non-Linear Quantisation Params --
q_type  = 'L';  % NL = Non-Linear, L=Linear
q_ratio = 30;   % quantisation quality factor

% -- Linear Quantisation Params --
step_size = 32; % step-size
cut_off   = 0;  % cut-off threshold


%=====================================================================
%       INITIALISATION
%=====================================================================

% ---- Load the Image File ----

if strcmp(file_ext,'pgm')
    [img,rows,cols,maxlum] = loadPGM(filename);
    %imshow(img);
    colormap(gray(255));
    image(img); title('Original')
elseif strcmp(file_ext,'bmp')
    img = imread(filename);
    imshow(img); title('Original')
else
    error('Unknown Filetype');
end

% change the range of the pixel values
img = double(img)-128;
[m,n] = size(img);
b = 8;  %block size

num_blocks = (m*n)/(b*b);       % total number of sub-blocks
num_hor_blks = m/b;             % number of horizontal subblocks
num_ver_blks = n/b;             % number of vertical sub blocks

enc_achuff_blk_bitlen=0;
enc_dc_coeffs_array = zeros(num_hor_blks,num_ver_blks);


%======================================================================
%       M P E G/J P E G     E N C O D E R
%======================================================================

% split the input larger matrix into smaller 8x8 blocks
split_img = mat2cell(img, b*ones(m/b,1), b*ones(1,n/b));

% perform DCT->Quantisation->Inverse-Quantisation->IDCT for each block
for i = 1:num_ver_blks
    for j = 1:num_hor_blks

        % --------------- 2D-DCT --------------------------------------
        % perform DCT - Encoder side
        % (output is a array of cells)
        enc_dct_block = dct_2_8x8(split_img{i,j},b,speed)';

        test_enc_dct_block{i,j} = enc_dct_block;

        % --------------- Quantisation -------------------------------
        % select quantisation method
        if strcmp(q_type, 'NL')
            enc_q_block =  NL_Quantizer(enc_dct_block,q_ratio,0);
        elseif strcmp(q_type, 'L')
            enc_q_block = L_Quantizer(enc_dct_block,0,step_size,cut_off);
        else
            error('unknown q_type');
        end

        test_enc_q_blocks{i,j} = enc_q_block;

        % --------------- Zig-Zag Scanning (AC Coeffs) ----------------
        enc_zz_block = zigzag_e(enc_q_block);
        enc_zz_block = reshape(enc_zz_block',1,b*b);

        test_enc_zz_block{i,j} = enc_zz_block;

        % save DC-Coefficient (for DPCM coding later)
        enc_dc_coeffs_array(i,j) = enc_zz_block(1);

        % --------------- RLC and Huffman Coding (AC Coeffs) ----------
        [blockACbit_seq,blockbit_len, zero_nonzero_pair]=ac_huffman(enc_zz_block);

        % save bit string
        enc_achuff_bits{i,j} = blockACbit_seq;
        % accumulate block-AC-bit-length
        enc_achuff_blk_bitlen = enc_achuff_blk_bitlen+blockbit_len;
    end
end

% --------------- DPCM Encoding (DC Coeffs) -------------------
enc_dc_coeffs_array = reshape(enc_dc_coeffs_array,1,num_blocks);

% DPCM encode the extracted coeffs
enc_dpcm_diffs = dc_dpcm_enc(enc_dc_coeffs_array);

% --------------- RLC and Huffman Coding (DC Coeffs) ----------
[blockDCbit_seq,blockbit_len]=dc_huffman(enc_dpcm_diffs);
% save the encoded codewords
enc_dchuff_bits = blockDCbit_seq;
% save the dc codeword bits length
enc_dchuff_blk_bitlen = blockbit_len;


%=====================================================================
%       M P E G/J P E G   D E C O D E R
%=====================================================================

% --------------- RLC and Huffman Decoding (DC Coeffs) ---------------
dec_dpcm_diffs = huffman_dc_decoding(enc_dchuff_bits);

% --------------- DPCM Decoding (DC Coeffs) --------------------------
dec_dc_coeffs_array = dc_dpcm_dec(dec_dpcm_diffs);
dec_dc_coeffs_array = reshape(dec_dc_coeffs_array,num_hor_blks,num_ver_blks);


for i = 1:num_ver_blks
    for j = 1:num_hor_blks

    % --------------- RLC and Huffman Decoding (AC Coeffs) ---------------
    % prepend dc coeff at the front
    dec_huff_ac_coeffs_blk{i,j}  = [dec_dc_coeffs_array(i,j) ...
                                huffman_ac_decoding(enc_achuff_bits{i,j})];
    dec_huff_ac_coeffs_blk{i,j}  = reshape(dec_huff_ac_coeffs_blk{i,j},b,b)';

    % --------------- Reverse Zig-Zag Scanning (AC Coeffs) ---------------
    dec_zz_blocks{i,j} = zigzag_dx(dec_huff_ac_coeffs_blk{i,j});

    % --------------- Inverse - Quantisation----------------------
    % select inverse-quantisation method
    if strcmp(q_type, 'NL')
        dec_q_block =  NL_Quantizer(dec_zz_blocks{i,j},q_ratio,1);
    elseif strcmp(q_type, 'L')
        dec_q_block = L_Quantizer(dec_zz_blocks{i,j},1,step_size,cut_off);
    else
        error('unknown q_type');
    end

    % --------------- 2D - Inverse - DCT -------------------------
    dec_idct_block{i,j} = idct_2_8x8(dec_q_block,b, speed)';
    reconstructed_cell{i,j} = dec_idct_block{i,j};


    clc;display(['Finished Decoding block=',num2str(i),',',num2str(j)])

    end
end

% convert the array of cells back to a matrix
reconstructed_img = cell2mat(reconstructed_cell);

quant_error = img - reconstructed_img;
```

```
% add back the 128, and convert to uint8 (to be displayed)
reconstructed_img = reconstructed_img + 128;
reconstructed_img = uint8(reconstructed_img);

% display reconstructed picture
figure;imshow(reconstructed_img); title('Reconstructed')


%=====================================================================
%       A N A L Y S I S
%=====================================================================

img_size_before_compression = (m*n)*8    % 8-bits per pixel
img_size_after_compression = enc_achuff_blk_bitlen+enc_dchuff_blk_bitlen

bpp = img_size_after_compression/(m*n)

compression_ratio = 100 - ((img_size_after_compression/...
                            img_size_before_compression) * 100)


% Q-error calcs
MSE = sum(sum((quant_error.* quant_error)))/(m*n); % calculating MSE
SNR = 10*log10(var(reshape(quant_error',1,m*n))/MSE);% calculating SNR
PeakSNR = 10*log10((255^2)/MSE);% calculating peak SNR
disp(strcat('MSE =',num2str(MSE)));
disp(strcat('SNR =',num2str(SNR)));
disp(strcat('PeakSNR =',num2str(PeakSNR)));

%close all
```

10.1.2 Load a PGM File

```
%=================================================================================
% Advanced Multimedia Applications
% Title       : Load PGM Image (loadPGM.m)
% Description: Returns a matrix containing the image loaded from the PGM format
%              file filename.  Handles ASCII (P2) and binary (P5) PGM file formats.
%
%              If the filename has no extension, and open fails, a '.pgm' will
%              be appended.
%
%              Copyright (c) Peter Corke, 1999  Machine Vision Toolbox for Matlab
%              Adapted from - Peter Corke 1994
% Input      : filename (image path)
% Output     : I       - data samples from image
%             rows    - number of rows
%             cols    - number of columns
%             maxlum - maximum luminance value
%=================================================================================
function [I,rows,cols,maxlum] = loadPGM(filename)

    % define whitespaces
    white = [' ' 9 10 13];  % space, tab, lf, cr
    white = char(white);

    fid = fopen(filename, 'r');

    % check if file exists
    if fid < 0,
        fid = fopen([filename '.pgm'], 'r');
    end

    if fid < 0,
        error('Couldn''t open file');
    end

    % -- start the header extraction --

    % <magic number>
    magic = fread(fid, 2, 'char');  % read in magic number (assume 2 chars)
    while 1
        c = fread(fid,1,'char');
        if c == '#',
            fgetl(fid); % ignore comment line
        elseif ~any(c == white)
            fseek(fid, -1, 'cof');  % go back one char
            break;
        end
    end

    % <number of columns>
    cols = fscanf(fid, '%d', 1);    % read in columns (integer)
    while 1
        c = fread(fid,1,'char');
        if c == '#',
            fgetl(fid);  % ignore comment line
        elseif ~any(c == white)
            fseek(fid, -1, 'cof');  % go back one char
            break;
        end
    end

    % <number of rows>
    rows = fscanf(fid, '%d', 1);    % read in rows (integer)
    while 1
        c = fread(fid,1,'char');
        if c == '#',
            fgetl(fid);  % ignore comment line
        elseif ~any(c == white)
            fseek(fid, -1, 'cof');  % go back one char
            break;
        end
    end

    % <max luminance value>
    maxlum = fscanf(fid, '%d', 1);  % read in max grey (integer)
    while 1
        c = fread(fid,1,'char');
        if c == '#',
            fgetl(fid);  % ignore comment line
        elseif ~any(c == white)
            fseek(fid, -1, 'cof');  % go back one char
            break;
        end
    end


    % check magic number to see which pgm version this is
    % throw error if different format
    if magic(1) == 'P',
        if magic(2) == '2',
            %disp(['ASCII PGM file ' num2str(rows) ' x ' num2str(cols)])
            I = fscanf(fid, '%d', [cols rows])';
        elseif magic(2) == '5',
            %disp(['Binary PGM file ' num2str(rows) ' x ' num2str(cols)])
            if maxlum == 1,
                fmt = 'unint1';
            elseif maxlum == 15,
                fmt = 'uint4';
            elseif maxlum == 255,
                fmt = 'uint8';
            elseif maxlum == 2^32-1,
                fmt = 'uint32';
            end
            I = fread(fid, [cols rows], fmt)';
        else
            disp('Not a PGM file');
        end
    end

    fclose(fid);    % close file
```

10.2 Appendix-B

10.2.1 1D-DCT (simple)

```
% ================================================================================
% Advanced Multimedia Applications
% Title       : 1D-DCT (dct_1.m)
% Description : Performs 1-Dimentional Discrete Cosine Transform using basic
% Input       : image samples (array)
% Output      : DCT-Coefficients
% ================================================================================
function dct1_result = dct_1(data)

    N = length(data);   % get length of the intput sample

    gn = data;
    GK = zeros(1,N);    % 1x8 matrix needed to store result

    sum_term = 0;

    % calculate the first G(k) term
    GK(1) = (1/sqrt(N))*sum(data);

    % outer loop
    for k = 2:N

        % inner loop
        for n = 1:N
            % the summation term
            sum_term = sum_term + (gn(n) * cos((((2*(n-1))+1)*(k-1)*pi)/(2*N)));
        end

        % final scaling of the dct output
        GK(k) = sqrt(2/N)*sum_term;

        sum_term = 0; % reset sum term

    end

    % produce the output
    dct1_result = GK;
```

10.2.2 1D-IDCT(simple)

```
% ================================================================================
% Advanced Multimedia Applications
% Title       : Inverse-1D-DCT (idct_1.m)
% Description : 1D- Discrete Cosine Transform.
% Input       : dct coefficients (array)
% Output      : inverse-dct results (array)
% ================================================================================
function idct1_result = idct_1(coeffs)

    % get array length
    N = length(coeffs);

    % preallocate
    gx = zeros(1,N);

    % first term C(u)
    Cu = [(1/sqrt(2)) ones(1,N-1)];

    % reset sum term
    sum_term = 0;

    % outer loop
    for x = 1:N

        % inner loop
        for u = 1:N
            sum_term = sum_term + (Cu(u)*coeffs(u)*cos((((2*(x-1))+1)*(u-1)*pi)/...
                                                                           (2*N)));
        end

        % scale factor for each sum term
        gx(x) = sqrt(2/N) * sum_term;

        sum_term = 0; % reset sum term

    end

    % output
    idct1_result = gx;
```

10.2.3 1D-FDCT(fast)

```
% ===================================================================================
% Advanced Multimedia Applications
% Title       : Fast-1D-DCT (fdct_1.m)
% Description : Fast 1D-DCT - using buttefly graph given in
%               "Standard Coecs - by Ghanbari"
% Input       : data samples (array)
% Output      : DCT-coefficients (array)
% ===================================================================================
function fdct_result = fdct_1(p)

    % verify if length is 8, can't do for any other size
    if(length(p)~=8)
        error('input data samples is not = 8');
    end

    % pre-calculating the sine and cosine values (for speed)
    cos_pi_4        =  0.707106781186548;
    neg_cos_pi_4    = -0.707106781186548;
    sin_pi_8        =  0.382683432365090;
    cos_pi_8        =  0.923879532511287;
    neg_sin_3pi_8   = -0.923879532511287;
    cos_3pi_8       =  0.382683432365090;
    sin_pi_16       =  0.195090322016128;
    cos_pi_16       =  0.980785280403230;
    cos_3pi_16      =  0.831469612302545;
    neg_sin_3pi_16  = -0.555570233019602;
    sin_5pi_16      =  0.831469612302545;
    cos_5pi_16      =  0.555570233019602;
    neg_sin_7pi_16  = -0.980785280403230;
    cos_7pi_16      =  0.195090322016128;

    % preallocation
    s = [0 0 0 0 0 0 0 0];
    t = [0 0 0 0 0 0 0 0];
    r = [0  0  0  0  0  0  0  0];
    c = [0  0  0  0  0  0  0  0];

    % first stage
    s(1) = p(1) + p(8);
    s(2) = p(2) + p(7);
    s(3) = p(3) + p(6);
    s(4) = p(4) + p(5);
    s(5) = p(4) - p(5);
    s(6) = p(3) - p(6);
    s(7) = p(2) - p(7);
    s(8) = p(1) - p(8);

    % second stage
    t(6) = (s(6)*(neg_cos_pi_4)) + (s(7)*cos_pi_4);
    t(7) = (s(6)*cos_pi_4)       + (s(7)*cos_pi_4);

    % third stage
    r(1) = s(1) + s(4);
    r(2) = s(2) + s(3);
    r(3) = s(2) - s(3);
    r(4) = s(1) - s(4);
    r(5) = s(5) + t(6);
    r(6) = s(5) - t(6);
    r(7) = s(8) - t(7);
    r(8) = s(8) + t(7);

    % fourth stage
    c(1) = (r(1)*cos_pi_4)      + (r(2)*cos_pi_4);
    c(5) = (r(1)*cos_pi_4)      + (r(2)*neg_cos_pi_4);
    c(3) = (r(3)*sin_pi_8)      + (r(4)*cos_pi_8);
    c(7) = (r(3)*neg_sin_3pi_8) + (r(4)*cos_3pi_8);
    c(2) = (r(5)*sin_pi_16)     + (r(8)*cos_pi_16);
    c(4) = (r(6)*neg_sin_3pi_16)+ (r(7)*cos_3pi_16);
    c(6) = (r(6)*sin_5pi_16)    + (r(7)*cos_5pi_16);
    c(8) = (r(5)*neg_sin_7pi_16)+ (r(8)*cos_7pi_16);

    % scaling performed (to match the simple dct)
    fdct_result = c/2;
```

10.2.4 1D-FIDCT(fast)

```
% ===================================================================================
% Advanced Multimedia Applications
% Title       : Fast-1D-IDCT (fidct_1.m)
% Description : Fast 1D-IDCT(Inverse) - using buttefly graph given in
%               "Standard Coecs - by Ghanbari"
% Input       : dct coefficients (array)
% Output      : inverse-dct results (array)
% ===================================================================================
function fidct_result = fidct_1(c)

    % verify if length is 8, can't do for any other size
    if(length(c)~=8)
        error('input data samples is not = 8');
    end

    % pre-calculating the sine and cosine values (for speed?)
    sin_pi_16   = 0.195090322016128;
    cos_pi_16   = 0.980785280403230;
    cos_3pi_16  = 0.831469612302545;
    sin_3pi_16  = 0.555570233019602;
    sin_5pi_16  = 0.831469612302545;
    cos_5pi_16  = 0.555570233019602;
    sin_7pi_16  = 0.980785280403230;
    cos_7pi_16  = 0.195090322016128;

    % working backwards from the FDCT stages (rescale)
    c = c*2;

    % first stage
    % equations pre calculated using symbol maths in matlab
    r(1) = (2^(1/2)*(c(1) + c(5)))/2;
    r(2) = (2^(1/2)*(c(1) - c(5)))/2;
    r(3) = (c(3)*(2 - 2^(1/2))^(1/2))/2 - (c(7)*(2^(1/2) + 2)^(1/2))/2;
    r(4) = (c(7)*(2 - 2^(1/2))^(1/2))/2 + (c(3)*(2^(1/2) + 2)^(1/2))/2;
    r(5) = (c(2)*cos_7pi_16 - c(8)*cos_pi_16)/(cos_pi_16*sin_7pi_16 +...
                                                cos_7pi_16*sin_pi_16);
    r(6) = -(c(4)*cos_5pi_16 - c(6)*cos_3pi_16)/(cos_3pi_16*sin_5pi_16 +...
                                                cos_5pi_16*sin_3pi_16);
    r(7) = (c(4)*sin_5pi_16 + c(6)*sin_3pi_16)/(cos_3pi_16*sin_5pi_16 +...
                                                cos_5pi_16*sin_3pi_16);
    r(8) = (c(2)*sin_7pi_16 + c(8)*sin_pi_16)/(cos_pi_16*sin_7pi_16 +...
                                                cos_7pi_16*sin_pi_16);

    % second stage    (skipped t1-t4)
    t(5) = r(5)/2 + r(6)/2;
    t(6) = r(5)/2 - r(6)/2;
    t(7) = r(8)/2 - r(7)/2;
    t(8) = r(7)/2 + r(8)/2;


    % third stage
    s(1) = r(1)/2 + r(4)/2;
    s(2) = r(2)/2 + r(3)/2;
    s(3) = r(2)/2 - r(3)/2;
    s(4) = r(1)/2 - r(4)/2;
    s(6) = -(2^(1/2)*(t(6) - t(7)))/2;
    s(7) = (2^(1/2)*(t(6) + t(7)))/2;

    % fourth stage
    p(1) = s(1)/2 + t(8)/2;
    p(2) = s(2)/2 + s(7)/2;
    p(3) = s(3)/2 + s(6)/2;
    p(4) = s(4)/2 + t(5)/2;
    p(5) = s(4)/2 - t(5)/2;
    p(6) = s(3)/2 - s(6)/2;
    p(7) = s(2)/2 - s(7)/2;
    p(8) = s(1)/2 - t(8)/2;

    % round the result to the nearest integer
    fidct_result = round(p);
```

10.2.5 2D-DCT – 8x8 Block

```
% ================================================================================
% Advanced Multimedia Applications
% Title           : dct_2_result (dct_2_8x8.m)
% Description     : 2D-DCT, using 1D-DCT, can toggle
%                   between fast/simple DCT
% Input           : mat2d - input matrix (8x8),
%                    b - block size
%                    dct_type - fast(1)/simple(0) DCT
% Output           : DCT-Coefficients (Matrix 8x8)
% ===========================================================================
function dct_2_result = dct_2_8x8(mat2d,b, dct_type)

    % check if the supplied block size matches the matrix size
    if(b ~= length(mat2d))
        error('Input Matrix size does not match block size');
    end

    % check dct type - fast/simple, else throw an error
    if(dct_type == 1 || dct_type == 0)
        if(dct_type == 1)
            % check if block size is 8, else throw an error
            if(b ~= 8)
                error('Fast DCT can only be supported for 8x8 blocks');
            end
        end
    else
        error('Unknown DCT Type, select 1(fast), 0(slow)');
    end

    % preallocate for speed
    dct_block = zeros(b,b);
    dct_block2 = zeros(b,b);

    % first do rows
    for j = 1:b
        % decide on fast/slow dct
        if(dct_type == 1)
            dct_block(j,:) = fdct_1(double(mat2d(j,:))); % rows (fast)
        else
            dct_block(j,:) = dct_1(double(mat2d(j,:)));  % rows (simple)
        end
    end

    % the result of the dct on rows
    dct_block_rows = dct_block;

    % then do columns on the transform of the rows
    for i = 1:b
         % decide on fast/slow dct
         if(dct_type == 1)
            dct_block2(:,i) = fdct_1(double(dct_block_rows(:,i))); % cols  (fast)
         else
            dct_block2(:,i) = dct_1(double(dct_block_rows(:,i))); % cols (simple)
         end
    end

    dct_2_result = dct_block2';
```

10.2.6 2D-IDCT-8x8 Block

```
%===========================================================================
% Advanced Multimedia Applications
% Title       : 2D-IDCT (8x8 block), using 1D-IDCT (idct_2_8x8.m)
% Description : Performs 2D-Inverse DCT using basic transform equations.
% Input       : mat2d - input matrix is a bxb matrix
%               b - block size
%               idct_type = 1('fast idct'), 0('simple idct')
% Output      : original samples (matrix)
%=================================================================================
function idct_2_result = idct_2_8x8(mat2d,b, idct_type)

    % check if the supplied block size matches the matrix size
    if(b ~= length(mat2d))
        error('Input Matrix size does not match block size');
    end

    % check dct type - fast/simple
    if(idct_type == 1 || idct_type == 0)
        if(idct_type == 1)
            if(b ~= 8)
                error('Fast IDCT can only be supported for 8x8 blocks');
            end
        end
    else
        error('Unknown IDCT Type, select 1(fast), 0(slow)');
    end

    % preallocate for speed
    idct_block = zeros(b,b);
    idct_block2 = zeros(b,b);

    % first do cols
    for i = 1:b
        if(idct_type == 1)
            idct_block(:,i) = fidct_1(double(mat2d(:,i)));  % cols (fast)
        else
            idct_block(:,i) = idct_1(double(mat2d(:,i)));  % cols (simple)
        end
    end

    idct_block_rows = idct_block;

    % then do columns on the transform of the rows
    for j = 1:b
         if(idct_type == 1)
            idct_block2(j,:) = fidct_1(double(idct_block_rows(j,:))); % rows  (fast)
         else
            idct_block2(j,:) = idct_1(double(idct_block_rows(j,:))); % rows (simple)
         end
    end

    % invert the output
    idct_2_result = idct_block2';
```

10.2.7 2D-DCT (Whole Image – rows, columns method)

```
% ===============================================================================
% Advanced Multimedia Applications
% Title       : 2D-DCT  (dct_2.m)
% Description : 2D-DCT on a NxN matrix (N should be divisible by 8)
%               returns - a cell array, of DCT blocks(8x8 block size)
% Input       : image samples (matrix), speed (1=fast, 0=simple)
% Output      : DCT-Coefficients (cell array)
% ===============================================================================
function dct_2_result = dct_2(mat,speed)
```

```
% split into bxb blocks
[m,n] = size(mat);
b=8;

% check if the image size is a multiple of block size
if(mod((m*n),(b*b)) ~= 0)
    error('image size not a multiple of block size - 8x8');
end

% split the input larger matrix into smaller 8x8 blocks
split_img = mat2cell(mat, b*ones(m/b,1), b*ones(1,n/b));

% loopiing through the rows
for r = 1:length(split_img) %rows
    % loopiing through the cols
    for c = 1:length(split_img) %cols

        % get each block
        block = split_img{r,c};

        % perform 2d-block dct on each block
        dct_block = dct_2_8x8(block,b,speed);  % 0-simple dct, 1-fast dct

        % save dct result
        dct_cell_array(r,c) = {dct_block};

    end
end

% point to output
dct_2_result = dct_cell_array;
```

10.2.8 2D-IDCT (Whole Image – rows, columns method)

```
%==================================================================================
% Advanced Multimedia Applications
% Title       : 2D-IDCT on a NxN matrix (idct_2.m)
% Description : Performs 2D-Inverse DCT, for a whole image, using block wise
%               2D-DCT  (N should be divisible by 8)
% Input       : mat - input matrix is a bxb matrix
%               speed - use fast/slow(simple) dct
% Output      : a cell array, of DCT blocks(8x8 block size)
%==================================================================================


function idct_2_result = idct_2(mat,speed)

% split into bxb blocks
[m,n] = size(mat);
b=8;

if(mod((m*n),(b*b)) ~= 0)
    error('mat size not a multiple of block size - 8x8');
end

% split the input larger matrix into smaller 8x8 blocks
split_mat = mat2cell(mat, b*ones(m/b,1), b*ones(1,n/b));
```

```
for r = 1:length(split_mat) %rows
    for c = 1:length(split_mat) %cols

        block = split_mat{r,c};

        idct_block = idct_2_8x8(block,b,speed);  % 0-simple dct, 1-fast dct

        idct_cell_array(r,c) = {idct_block};

    end
end

idct_2_result = idct_cell_array;
```

10.2.9 2D-DCT (Simple – using basic transform equations)

```
% =========================================================================================
% Advanced Multimedia Applications
% Title         : 1D-DCT (simple) (dct_2_simple.m)
% Description   : 2D - DCT (Simple), using basic transform equations
% Input         : image samples (NxN matrix)
% Output        : DC-coefficients (NxN matrix)
% ================================================================================
function dct2_result = dct_2_simple(data)

    N = length(data);   % we assume that height=width
    xn = data;
    Xij = zeros(N,N);   % NxN matrix needed to store result

    % preallocation
    sum_k=zeros(1,N);
    sum_l=0;
    inner = zeros(1,N);

    % first outer loop for the rows
    for i=1:N
        % second outer loop for the rows
        for j=1:N
            % first inner loop for the outer summation
            for k = 1:N
                % second inner loop for the cos term summation
                for l = 1:N
                    inner(l) = xn(k,l) * cos(((2*pi)*((i-1)/(2*N))*((k-1)+0.5)))...
                                        * cos(((2*pi)*((j-1)/(2*N))*((l-1)+0.5)));
                end

                % get sum of inner loop terms
                sum_l = sum(inner);
                % summation of the k terms
                sum_k(k) = sum_l;
                % reset the summation (for the next run of the loop)
                sum_l = 0;
            end

            % summation for the first inner loop
            sum_kk = sum(sum_k);

            % calculating and adding the cj term
            if(i==1)
                ci = 1/sqrt(2);
            else
```

```
                ci = 1;
            end;
            if (j==1)
                cj = 1/sqrt(2);
            else
                cj = 1;
            end;

            Xij(i,j) = (2/N)* ci * cj * sum_kk;

            % reset the summations (for the next run of the loop)
            sum_k =0;
            sum_kk = 0;
        end

        % output status
        %clc;display(['Finished Encoding =',num2str(i),',',num2str(j)])
    end

    % send output
    dct2_result = Xij;
```

10.2.10 2D-IDCT (Simple – using basic transform equations)

```
%==================================================================================
% Advanced Multimedia Applications
% Title       : Simple 2D-IDCT (idct_2_simple.m)
% Description : Performs 2D-Inverse DCT using basic transform equations.
% Input       : dct-coefficients(matrix)
% Output      : original samples (matrix)
%==================================================================================
function dct2_result = idct_2_simple(coeffs)

    N = length(coeffs);   % we assume that height=width

    % initialisation
    XN = coeffs;
    Xkl = zeros(N,N); % NxN matrix needed to store result
    sum_i=0;
    sum_j=0;

    % first outer loop
    for k=1:N
        %second outer loop
        for l=1:N

            % first inner loop
            for i = 1:N
                % second inner loop
                for j = 1:N

                    % calculating and adding the cj term
                    if(i==1)
                        ci = 1/sqrt(2);
                    else
                        ci = 1;
                    end;

                    if (j==1)
                        cj = 1/sqrt(2);
                    else
                        cj = 1;
                    end;

                    % inner main cosine calculation
                    inner =  ci * cj * XN(i,j) * cos(((2*pi)*((i-1)/(2*N))*...
                            ((k-1)+0.5))) * cos(((2*pi)*((j-1)/(2*N))*((l-1)+0.5)));

                    sum_j = sum_j + inner;

                end
                % get summarion for second inner loop
                sum_i = sum_i + sum_j;
                sum_j = 0;
            end
            % get result for first inner loop
            Xkl(k,l) = (2/N) * sum_i;
            sum_i = 0;

        end
    end

    dct2_result = Xkl;
```

10.3 Appendix-C

10.3.1 Linear Quantizer/Inverse-Quantizer

```
%==================================================================================
% Advanced Multimedia Applications
% Title       : Linear Quantizer/inv-Quantizer for matrixs (L_Quantizer.m)
% Description :  For *qunatisattion*
%                ------------------------------
%                Set rev = 0 to set to quantsation mode
%                idata is the input matrix of 8 x 8
%
%                For *inv-qunatisattion*
%                ------------------------------
%                Set rev = 1 to set to inv-quantsation mode
%                idata is the input matrix of 8 x 8 and
%                odata is the non linaraly quantsied output
%
% Input      : idata        (input data - matrix)
%              rev          (rev=1 - inverse-quantisation)
%              step_size    (quantisation step size)
%              cut_off      (cut-off for the quantisation)
% Output     : odata -  non linaraly quantsied output / original data samples
%              rate - number of zeros indirectly specifiying the data loss
%==================================================================================
function [odata,rate] = L_Quantizer(idata,rev,step_size,cut_off)

[xlen,ylen] = size(idata);      % getting size of matrix
if (xlen ~= 8) || (ylen ~= 8)   % checking for 8x8 matrix
    disp('  ');
    disp('Quantzer ERROR : 8x8 matrix needed as input.');
    disp('  ');
    return;
end;

      Q  = ones(8,8)*step_size;                       % Specifiying Q matrix

  if (rev == 0)   % for qunatizing
    odata = round(idata./Q);                  % Quantising matrix
    odata(odata < cut_off & odata > (cut_off *-1)) = 0;         % engaging cut_off
    rate = sum(sum(odata == 0));         % counting zeros(data loss)
  else
    odata = idata.*Q;
    rate = 0;
  end;
```

10.3.2 Non-Linear Quantizer-Inverse Quantizer

```
%==================================================================================
% Advanced Multimedia Applications
% Title         : Non linear Quantizer/Inv-Quantzer (NL_Quantizer.m)
% Description   : usage =>      [Qunat_data,rate] = NL_Quantizer(idata,ratio,rev)

%               For qunatisattion
%               ------------------------------
%               Set rev = 0 to set to quantsation mode
%               idata is the input matrix of 8 x 8 and ratio is the quality level
%               of quantsed matrix. The higher the quality, lesser the comprssion.
%               odata is the non linaraly quantsied output and rate give the number
%               of zeros indirectly specifiying the data loss
%                ----------------------

%                For inv-qunatisattion
%                -----------------------------
%                Set rev = 1 to set to inv-quantsation mode
%                idata is the input matrix of 8 x 8 and ratio is the quality level
%                of quantiser used previously odata is the non linaraly quantsied
%                output
%                ----------------------
% Input          : idata (input matrix)
%                : ratio (Quality ratio)
%                : rev   (1=inv-quantisation, 0=quantisation)
%
% Output         : odata (output data - quantised/inverse quantised samples)
%                  rate - number of zeros (after quantisation)
%==================================================================================
function [odata,rate] = NL_Quantizer(idata,ratio,rev)

    if (ratio < 1) || (ratio > 99)      % checking for ratio range
        disp('  ');
        disp('Quantzer ERROR : Please follow ratio range =>  0 < ratio < 100');
        disp('  ');
        return;
    end;
    [xlen,ylen] = size(idata);          % getting size of matrix
    if (xlen ~= 8) || (ylen ~= 8)       % checking for 8x8 matrix
        disp('  ');
        disp('Quantzer ERROR : 8x8 matrix needed as input.');
        disp('  ');
        return;
    end;


    odata = zeros(8 , 8);                       % initialsizing output matrix

    Q50 = [ 16  11  10  16  24  40  51  61;     % Q50 matrix
            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];

    % for calculating Q marix
    if     (ratio < 50) Q = round(Q50 * (50/ratio));        % for Q < 50
    elseif (ratio > 52) Q = round(Q50 * (100 - ratio)/50);  % for Q > 50
    else                Q = Q50;  end;                      % for Q = 50


    if (rev == 0)    % for qunatizing
        Q(Q >=256) = 255;                       % thresholding Q matrix
        odata = round(idata./Q);                % quantizing idata
        rate = sum(sum(odata == 0));       % counting zeros(data loss)
    else             % for inv-qunatizing
        odata = idata.*Q;
        rate = 0;
    end;
```

10.4 Appendix-D

10.4.1 ZigZag Scanner

```
%=================================================================================
% Advanced Multimedia Applications
% Title       : ZigZag Encoder (zigzag_e.m)
% Description : Zig-zag encoder for matrixs
%               usage =>     optdata = zigzag_e(inpdata);
%               The function scans the input data(inpdata) in zig-zag format and
%               returns it as a matrix of same dimention
%
% Input       : (matrix)
% Outout      : Inverse-Zigzag (matrix)
%=================================================================================
function [zig_data] = zigzag_e(idata)
%%
%% Zig-zag encoder for matrixs
%%
%%
%% usage =>     optdata = zigzag_e(inpdata);
%%
%% The function scans the input data(inpdata) in zig-zag format and returns
%% it as a matrix of same dimention
%%

[xlen,ylen] = size(idata);      % getting size of matrix
if xlen ~= ylen                 % checking for square matrix
    disp('  ');
    disp('Zig-Zag encoder ERROR : Square matrix needed as input.');
    disp('  ');
    return;
end;

count = 1; x = 1; y = 1;        % initalising counter variables
zig_data = zeros(xlen , ylen);  % initialising output matrix
zig_data(1,1) = idata(1,1);

% Top half ---------------
while (1)
    y=y+1;

    count = count + 1;          % data logger
    zig_data(fix(((count-1)/xlen)+1),count-((fix((count-1)/xlen)*xlen))) =...
                                                                        idata(x,y);

    while(y~=1) % to loop till reaching the top edge(for moving diagonally up)
        x=x+1;
        y=y-1;

        count = count + 1;      % data logger
        zig_data(fix(((count-1)/xlen)+1),count-((fix((count-1)/xlen)*xlen))) =...
                                                                        idata(x,y);

    end
    x=x+1;
    if(x>xlen)  % to detect the end of top half
        x=x-1;
        break;
    end;

    count = count + 1;          % data logger
    zig_data(fix(((count-1)/xlen)+1),count-((fix((count-1)/xlen)*xlen))) =...
                                                                        idata(x,y);
    while(x~=1)  % to loop till reaching the left edge(for moving diagonally down)
        x=x-1;
        y=y+1;

        count = count + 1;      % data logger
        zig_data(fix(((count-1)/xlen)+1),count-((fix((count-1)/xlen)*xlen))) =...
                                                                        idata(x,y);


    end;
end;

% bottom half -----------
while(1)
    y=y+1;

    count = count + 1;          % data logger
    zig_data(fix(((count-1)/xlen)+1),count-((fix((count-1)/xlen)*xlen))) =...
                                                                        idata(x,y);
     % to loop till reaching the bottom edge(for moving diagonally down)
    while(y~=ylen)
        y=y+1;
        x=x-1;

        count = count + 1;      % data logger
        zig_data(fix(((count-1)/xlen)+1),count-((fix((count-1)/xlen)*xlen))) =...
                                                                        idata(x,y);


    end;
    x=x+1;
    if(x>xlen)  % to detect end of bottom half
        x=x-1;
        break;
    end;

    count = count + 1;          % data logger
    zig_data(fix(((count-1)/xlen)+1),count-((fix((count-1)/xlen)*xlen))) =...
                                                                        idata(x,y);

    while(x~=xlen)  % to loop till reaching the right edge(for moving diagonally up)
        x=x+1;
        y=y-1;

        count = count + 1;      % data logger
        zig_data(fix(((count-1)/xlen)+1),count-((fix((count-1)/xlen)*xlen))) =...
                                                                        idata(x,y);


    end;
end;
```

10.4.2 Reverse-ZigZag Scanner

```
%=================================================================================
% Advanced Multimedia Applications
% Title       : ZigZag Encoder (zigzag_e.m)
% Description : Zig-zag decoder for matrixs
%               usage =>     optdata = zigzag_dx(inpdata);
%               The function scans the zig-zag input data(inpdata) and returns
%               it as a normal matrix form.
% Input       : matrix
% Output      : sorted data (matrix)
%=================================================================================
function [out_data] = zigzag_dx(idata)

[xlen,ylen] = size(idata);      % getting size of matrix
if xlen ~= ylen                 % checking for square matrix
    disp('  ');
    disp('Zig-Zag decoder ERROR : Square matrix needed as input.');
    disp('  ');
    return;
end;

out_data = zeros(xlen , ylen);  % initialising output matrix
temp_a = zeros(xlen , ylen);    % initialising temporary matrix

for i = 1 : xlen
    for j = 1 : ylen % setting index numbers for temporary matrix
        temp_a(i,j) = (((i-1) *xlen )+j);
    end
end
temp_b = zigzag_e(temp_a); % zigzag-ing temporary matrix

% scanning tempporary matrix for calculating output data positions
for i = 1 : xlen
    for j = 1 : ylen
        out_data(fix(((temp_b(i,j)-1)/xlen)+1),temp_b(i,j)-((fix((temp_b(i,j)-1)...
                                                  /xlen)*xlen))) = idata(i,j);
    end
end
```

10.4.3 DPCM – Encoder

```
% ==================================================================================
% Advanced Multimedia Applications
% Title         : DPCM Encoder (dc_dpcm_enc.m)
% Description   : Differential coding of DC-Components
% Input         : dc_coeffs_from_each_block (array)
% Output        : Diff (differentially encoded outputs, array)
% ==================================================================================
function diff = dc_dpcm_enc(dc_coeffs_from_each_block)

    % first value is encoded against zero
    dc_vals = [0 dc_coeffs_from_each_block];

    i = 2:length(dc_vals);
    diff = dc_vals(i)-dc_vals(i-1);

end
```

10.4.4 DPCM – Decoder

```
% ==================================================================================
% Advanced Multimedia Applications
% Title         : DPCM Decoder (dc_dpcm_dec.m)
% Description   : Differential decoding of Diffs.
% Input         : Differential codes (array)
% Output        : Diff (differentially decoded output, array)
% ==================================================================================
function dc_coeffs = dc_dpcm_dec(diffs)

    % first coeff is the same as first diff
    dc_coeffs = zeros(1,length(diffs));
    dc_coeffs(1) = diffs(1);   % first dc value is the same as first diff
    % loop through the differentials
    for i=2:length(diffs)
        dc_coeffs(i) = diffs(i)+dc_coeffs(i-1);
    end

end
```

10.5 Appendix-E

10.5.1 Huffman Encoder (AC – Coefficients)

```
% ================================================================================
% Advanced Multimedia Applications
% Title       : AC Huffman Encoding function (ac_huffman.m)
% Description : input would be the sequence after Zig-Zag
%               zerolen is the number of continuing 0, amplitude is the value
%               of non-0 number after that
%               do Huffman after every AC RLC, the result is bit_seq
%               blockACbit_seq is the endocing result of whole block
%
%               AC Codeword - [symbol-1, symbol-2]
%               symbol-1 = (run-length, size)
%               symbol-2 = (amplitude)
%
% Input       : AC-components (after zigzag coding)
% Output      : blockACbit_seq    (encoded bit sequence)
%               blockbit_len      (length of bit sequence)
%               zero_nonzero_pair (RLC pairs)
% ================================================================================
function[blockACbit_seq,blockbit_len,zero_nonzero_pair]=ac_huffman(after_z)

    % initialisation
    e = after_z;
    eob_seq=dec2bin(10,4); % eob=1010(end symbol)
    blockACbit_seq=[];
    blockbit_seq=[];
    zerolen=0;
    zeronumber=0;

    zero_nonzero_pair = {};
    n=1;


    if numel(e)==1           %% the situation of all AC factors are 0
        blockACbit_seq=[];
        blockbit_len=length(blockbit_seq);
    else
        for i=2:length(e) %the AC part is from the second number

            if ( e(i)==0 & zeronumber<15)
                zeronumber=zeronumber+1;


            elseif ((e(i)==0) & (zeronumber==15)) %(15, 0) = 16 zeros
                zeronumber=0;
                blockACbit_seq=[blockACbit_seq,'11111111111111111'];

                % save RLC pair
                zero_nonzero_pair{n} = [15 0];
                n=n+1;

             elseif(e(i))  %if value non-equal to zero
```

```
                      % save RLC pair
                      zero_nonzero_pair{n} = [zeronumber e(i)];
                      n=n+1;

                      zerolen=zeronumber;
                      amplitude=ac_category(e(i));
                      zeronumber=0;

                      bit_seq=tableAC2(zerolen,amplitude);

                      % get symbol-2
                      symbol_2 = amp_to_sym2(e(i),amplitude);

                      % append symbol-1 and symbol-2 to the overall bit
                      % string.
                      blockACbit_seq=[blockACbit_seq,bit_seq,symbol_2];
             end
        end
    end

           blockACbit_seq = [blockACbit_seq eob_seq];
           blockbit_len=length(blockACbit_seq);

           % blockACbit_seq is the sequence for the AC part
           % blockbit_len is the length
           % the sequence of whole block would be:
           % blockbit_seq=[blockDCbit_seq,blockACbit_seq,eob_seq];
end
```

10.5.2 Huffman Encoder (DC – Coefficients)

```
% ================================================================================
% Advanced Multimedia Applications
% Title         : DC huffman encoding (dc_huffman.m)
% Description   : Huffman Coding of DC-Coefficients (using Huffman tables)
% Input         : DC components (array)
% Output        : binary sequence(string), length of sequence
% ================================================================================
function [blockDCbit_seq,blockbit_len]=dc_huffman(dc_seq)
    blockDCbit_seq=[];
    blockbit_seq=[];
    e = dc_seq;
    len = length(dc_seq);
    for i =1:len;
     amplitude=dc_category(e(i)); % get category
     bit_seq=tableDC(amplitude);  % get basecode

         if(amplitude == 0) % if the number is 0
             blockDCbit_seq=[blockDCbit_seq,bit_seq];
             blockbit_len=length(blockDCbit_seq);
         else
            symbol_2 = amp_to_sym2(e(i),amplitude);
            blockDCbit_seq=[blockDCbit_seq,bit_seq,symbol_2];
            blockbit_len=length(blockDCbit_seq);
         end


    end
    %blockDCbit_seq is the sequence for the DC part
    %blockbit_len is the length
```

10.5.3 Huffman Decoder (AC – Coefficients)

```
% ================================================================================
% Advanced Multimedia Applications
% Title       : huffman AC decoding function (huffman_ac_decoding.m)
% Description : this function is to do the Inverse huffman coding the
%               input should be the whole sequence including the output
%               AC components and EOB, the idea is to match the sequence
%               with the code in decode table, get the category of the value
%
% Input       :  binary sequence (string)
% Output      :  AC-coefficients (array)
% ================================================================================
function [ac_dehuff]=huffman_ac_decoding(blockbit_seq)
ac_dehuff=[];point=1;
len=length(blockbit_seq);

for i = point:len  %'point' is a variable to mark the start bit of each
                   %code, 'len' is the length of the code put into the
                   %decode table
    temp_ac = blockbit_seq(point:i);
    group_ac = detableAC2(temp_ac);

    if group_ac(1,1)==0 && group_ac(1,2)==0; % EOB reached.
       group_ac;

       ac_dehuff = [ac_dehuff zeros(1,63-length(ac_dehuff))];

       break

    elseif group_ac(1,2) >= 0 && group_ac(1,2) <= 10;%get correct output
       % while doing the loop to match the decoding table, for other
       % situations the group_ac(1,2)would be 11
        group_ac;

       for zero = 1: group_ac(1,1)
          ac_dehuff=[ac_dehuff, 0];
       end

        %ac_dehuff = [ac_dehuff zeros(1,group_ac(1,1));

        if group_ac(1,2)==0;
            ac_dehuff=[ac_dehuff, 0];
        else
         r_ac = blockbit_seq(i+1:i+group_ac(1,2));
          if r_ac(1)=='0';   %% if it's a negative number
            for j=1:length(r_ac)
                 if r_ac(j)=='0';
                     r_ac(j)='1';
                 else
                     r_ac(j)='0';
                 end
            end
            r_new=-bin2dec(r_ac);
           else r_new=bin2dec(r_ac);
          end
          r_new;
          ac_dehuff=[ac_dehuff,r_new];
        end
     point=i+group_ac(1,2)+1;  %once decoded the current code, move to the
                               %next number
     i=point;
     ac_dehuff;
    end
end %finish AC part
ac_dehuff;
```

10.5.4 Huffman Decoder (DC – Coefficients)

```
% ==================================================================================
% Advanced Multimedia Applications
% Title       : huffman DC decoding function (huffman_dc_decoding.m)
% Description : this function is to do the Inverse huffman coding for dc
%               part the input should be the whole sequence including the
%               output of DC, the idea is to match the sequence with the
%               code in decode table, get the category of the value then cut
%               the relative number,change them into decimal
%
% Input       : binary sequence (string)
% Output      : decoded dc-coefficients(array)
% ==================================================================================
function [dc_dehuff]=huffman_dc_decoding(blockbit_seq)
dc_dehuff=[];point=1;
len=length(blockbit_seq);
r_new=0;
for i = point:len  %'point' is a variable to mark the start bit of each
                   %code, 'len' is the length of the code put into the
                   %decode table
    temp_dc = blockbit_seq(point:i);
    category_dc = detableDC(temp_dc);

    if(category_dc == 0) % to distinguish between 0 and -1
        r_new = 0;
        dc_dehuff=[dc_dehuff,r_new];
        point = i + category_dc+1;
        i = point;

    %get correct output
    elseif category_dc > 0 && category_dc <= 11;
       % while doing the loop to match the decoding table, for other
       % situations the category_dc would be 12
          category_dc;
        r_dc = blockbit_seq(i+1:i+category_dc);
        r_new = sym2_to_amp(r_dc,category_dc);
          dc_dehuff=[dc_dehuff,r_new];

          point = i + category_dc+1;
          i = point;
    end

    r_new;


end   %finish DC part
```

```
%dc_dehuff
```

10.5.5 Huffman Tables (AC/DC, Encoding-Decoding)

```
% ================================================================================
% Advanced Multimedia Applications
% Title       : AC-Category Table (ac_category.m)
% Description : The Category (CAT) of the baseline encoder (AC Values)
%                 Tables taken from [R.C Gonzalez, R.E Woods - "Digital Image
%                 Processing, Prentice Hall(2008)]
% input  : AC-Coefficient amplitude
% output : Amplitude(category)
% ================================================================================
function amplen = ac_category(x)

    R=abs(x);   % only check for (+)ve ranges

    if R==1;                             amplen=1;
    elseif(R >=   2  & R <=    3);   amplen = 2;
    elseif(R >=   4  & R <=    7);   amplen = 3;
    elseif(R >=   8  & R <=   15);   amplen = 4;
    elseif(R >=  16  & R <=   31);   amplen = 5;
    elseif(R >=  32  & R <=   63);   amplen = 6;
    elseif(R >=  64  & R <=  127);   amplen = 7;
    elseif(R >= 128  & R <=  255);   amplen = 8;
    elseif(R >= 256  & R <=  511);   amplen = 9;
    elseif(R >= 512  & R <= 1023);   amplen = 10;
    elseif(R >= 1024 & R <= 2047);   amplen = 11;
    end
```

```
% ================================================================================
% Advanced Multimedia Applications
% Title       : DC-Category Table (dc_category.m)
% Description : The Category (CAT) of the baseline encoder (DC Values)
%                 Tables taken from [R.C Gonzalez, R.E Woods - "Digital Image
%                 Processing, Prentice Hall(2008)]
% Input       : Non-zero coefficient value (DC-Coeff)
% Output      : Amplitude
% ================================================================================
function amplen = dc_category(x)

    val=abs(x); % only check for (+)ve ranges

    if (val==0);                          amplen=0;
    elseif(val==1);                       amplen=1;
    elseif(val >=    2 & val  <=    3);amplen= 2;
    elseif(val >=    4 & val  <=    7);amplen= 3;
    elseif(val >=    8 & val  <=   15);amplen= 4;
    elseif(val >=   16 & val  <=   31);amplen= 5;
    elseif(val >=   32 & val  <=   63);amplen= 6;
    elseif(val >=   64 & val  <=  127);amplen= 7;
    elseif(val >=  128 & val  <=  255);amplen= 8;
    elseif(val >=  256 & val  <=  511);amplen= 9;
    elseif(val >=  512 & val  <= 1023);amplen=10;
    elseif(val >= 1024 & val  <= 2047);amplen=11;
    end
```

```
%=================================================================================
% Advanced Multimedia Applications
% Title         : AC-Coefficient Huffman Table (tableAC2.m)
% Description   : AC Huffman coefficients of Luminance (table used for Encoding)
%                 Tables taken from [R.C Gonzalez, R.E Woods - "Digital Image
%                 Processing, Prentice Hall(2008)]
% Input         : RUN (integer), CAT (integer)
% Output        : Codeword
%=================================================================================
function basecode=tableAC2(run,category)

    %%%%%%%%%zero%%%%%%%%%%

    if run==0 & category==0
        basecode='1010';
    elseif run==0 & category==1
        basecode='00';
    elseif run==0 & category==2
        basecode='01';
    elseif run==0 & category==3
        basecode='100';
    elseif run==0 & category==4
        basecode='1011';
    elseif run==0 & category==5
        basecode='11010';
    elseif run==0 & category==6
        basecode='111000';
    elseif run==0 & category==7
        basecode='1111000';
    elseif run==0 & category==8
        basecode='1111110110';
    elseif run==0 & category==9
        basecode='1111111110000010';
    elseif run==0 & category==10
        basecode='1111111110000011';

    %%%%%%%ones%%%%%%%%%%
    elseif run==1 & category==1
        basecode='1100';
    elseif run==1 & category==2
        basecode='111001';
    elseif run==1 & category==3
        basecode='1111001';
    elseif run==1 & category==4
        basecode='111110110';
    elseif run==1 & category==5
        basecode='1111111010';
    elseif run==1 & category==6
        basecode='1111111110000100'; %%%
    elseif run==1 & category==7
        basecode='1111111110000101';
    elseif run==1 & category==8
        basecode='1111111110000110';
    elseif run==1 & category==9
        basecode='1111111110000111';
    elseif run==1 & category==10
        basecode='1111111110001000';

    %%%%%%%twos%%%%%%%%%%
    elseif run==2 & category==1
        basecode='11011';
    elseif run==2 & category==2
        basecode='11111000';
    elseif run==2 & category==3
        basecode='1111110111';
    elseif run==2 & category==4
        basecode='1111111110001001';
    elseif run==2 & category==5
        basecode='1111111110001010';
    elseif run==2 & category==6
        basecode='1111111110001011';
    elseif run==2 & category==7
        basecode='1111111110001100';
    elseif run==2 & category==8
        basecode='1111111110001101';
    elseif run==2 & category==9
        basecode='1111111110001110';
    elseif run==2 & category==10
        basecode='1111111110001111';

    %%%%%%%threes%%%%%%%%%%
    elseif run==3 & category==1
        basecode='111010';
    elseif run==3 & category==2
        basecode='111110111';
    elseif run==3 & category==3
        basecode='11111110111';
    elseif run==3 & category==4
        basecode='1111111110010000';
    elseif run==3 & category==5
        basecode='1111111110010001';
    elseif run==3 & category==6
        basecode='1111111110010010';
    elseif run==3 & category==7
        basecode='1111111110010011';
    elseif run==3 & category==8
        basecode='1111111110010100';
    elseif run==3 & category==9
        basecode='1111111110010101';
    elseif run==3 & category==10
        basecode='1111111110010110';

    %%%%%%%fours%%%%%%%%%%
    elseif run==4 & category==1
        basecode='111011';
    elseif run==4 & category==2
        basecode='1111111000';
    elseif run==4 & category==3
        basecode='1111111110010111';
    elseif run==4 & category==4
        basecode='1111111110011000';
    elseif run==4 & category==5
        basecode='1111111110011001';
    elseif run==4 & category==6
        basecode='1111111110011010'; %%%
    elseif run==4 & category==7
        basecode='1111111110011011';
    elseif run==4 & category==8
        basecode='1111111110011100'; %%%%
    elseif run==4 & category==9
        basecode='1111111110011101';
    elseif run==4 & category==10
        basecode='1111111110011110';

    %%%%%%%fives%%%%%%%%%%
    elseif run==5 & category==1
        basecode='1111010';
    elseif run==5 & category==2
        basecode='1111111001';
    elseif run==5 & category==3
        basecode='1111111110011111';
    elseif run==5 & category==4
        basecode='1111111110100000';
    elseif run==5 & category==5
        basecode='1111111110100001';
    elseif run==5 & category==6
        basecode='1111111110100010';
    elseif run==5 & category==7
```

```
        basecode='1111111110100011';
    elseif run==5 & category==8
        basecode='1111111110100100';
    elseif run==5 & category==9
        basecode='1111111110100101';
    elseif run==5 & category==10
        basecode='1111111110100110';

    %%%%%%%sixes%%%%%%%%%%
    elseif run==6 & category==1
        basecode='1111011';
    elseif run==6 & category==2
        basecode='11111111000';
    elseif run==6 & category==3
        basecode='1111111110100111';
    elseif run==6 & category==4
        basecode='1111111110101000';
    elseif run==6 & category==5
        basecode='1111111110101001';
    elseif run==6 & category==6
        basecode='1111111110101010';
    elseif run==6 & category==7
        basecode='1111111110101011';
    elseif run==6 & category==8
        basecode='1111111110101100';
    elseif run==6 & category==9
        basecode='1111111110101101';
    elseif run==6 & category==10
        basecode='1111111110101110';

    %%%%%%%sevens%%%%%%%%%%
    elseif run==7 & category==1
        basecode='11111001';
    elseif run==7 & category==2
        basecode='11111111001';
    elseif run==7 & category==3
        basecode='1111111110101111';
    elseif run==7 & category==4
        basecode='1111111110110000';
    elseif run==7 & category==5
        basecode='1111111110110001';
    elseif run==7 & category==6
        basecode='1111111110110010';
    elseif run==7 & category==7
        basecode='1111111110110011';
    elseif run==7 & category==8
        basecode='1111111110110100';
    elseif run==7 & category==9
        basecode='1111111110110101';%%%
    elseif run==7 & category==10
        basecode='1111111110110110';

    %%%%%%%eights%%%%%%%%%%
    elseif run==8 & category==1
        basecode='11111010';
    elseif run==8 & category==2
        basecode='111111111000000';
    elseif run==8 & category==3
        basecode='1111111101110111';
    elseif run==8 & category==4
        basecode='1111111110111000';
    elseif run==8 & category==5
        basecode='1111111110111001';
    elseif run==8 & category==6
        basecode='1111111110111010';
    elseif run==8 & category==7
        basecode='1111111110111011';
    elseif run==8 & category==8
        basecode='1111111110111100';
    elseif run==8 & category==9
        basecode='1111111110111101';
    elseif run==8 & category==10
        basecode='1111111110111110'; %%%

    %%%%%%%nines%%%%%%%%%%
    elseif run==9 & category==1
        basecode='111111000';
    elseif run==9 & category==2
        basecode='1111111110111111';
    elseif run==9 & category==3
        basecode='1111111111000000';
    elseif run==9 & category==4
        basecode='1111111111000001';
    elseif run==9 & category==5
        basecode='1111111111000010';
    elseif run==9 & category==6
        basecode='1111111111000011';
    elseif run==9 & category==7
        basecode='1111111111000100';
    elseif run==9 & category==8
        basecode='1111111111000101';
    elseif run==9 & category==9
        basecode='1111111111000110';
    elseif run==9 & category==10
        basecode='1111111111000111';

    %%%%%%%tens%%%%%%%%%%
    elseif run==10 & category==1
        basecode='111111001';
    elseif run==10 & category==2
        basecode='1111111111001000';
    elseif run==10 & category==3
        basecode='1111111111001001';
    elseif run==10 & category==4
        basecode='1111111111001010';
    elseif run==10 & category==5
        basecode='1111111111001011';
    elseif run==10 & category==6
        basecode='1111111111001100';
    elseif run==10 & category==7
        basecode='1111111111001101';
    elseif run==10 & category==8
        basecode='1111111111001110';
    elseif run==10 & category==9
        basecode='1111111111001111';
    elseif run==10 & category==10
        basecode='1111111111010000';

    %%%%%%%elevens%%%%%%%%%%
    elseif run==11 & category==1
        basecode='111111010';
    elseif run==11 & category==2
        basecode='1111111111010001';
    elseif run==11 & category==3
        basecode='1111111111010010';
    elseif run==11 & category==4
        basecode='1111111111010011';
    elseif run==11 & category==5
        basecode='1111111111010100';
    elseif run==11 & category==6
        basecode='1111111111010101';
    elseif run==11 & category==7
        basecode='1111111111010110';
    elseif run==11 & category==8
        basecode='1111111111010111';
    elseif run==11 & category==9
        basecode='1111111111011000';
    elseif run==11 & category==10
```

```
        basecode='1111111111011001';

    %%%%%%%twelevs%%%%%%%%%%
    elseif run==12 & category==1
        basecode='11111111111111110'; %%%%
    elseif run==12 & category==2
        basecode='1111111111011010';
    elseif run==12 & category==3
        basecode='1111111111011011';
    elseif run==12 & category==4
        basecode='1111111111011100';
    elseif run==12 & category==5
        basecode='1111111111011101';
    elseif run==12 & category==6
        basecode='1111111111011110'; %%%
    elseif run==12 & category==7
        basecode='1111111111011111';
    elseif run==12 & category==8
        basecode='1111111111100000'; %%%
    elseif run==12 & category==9
        basecode='1111111111100001';
    elseif run==12 & category==10
        basecode='1111111111100010';

    %%%%%%%13s%%%%%%%%%%
    elseif run==13 & category==1
        basecode='11111111010';
    elseif run==13 & category==2
        basecode='1111111111100011';
    elseif run==13 & category==3
        basecode='1111111111100100';
    elseif run==13 & category==4
        basecode='1111111111100101';
    elseif run==13 & category==5
        basecode='1111111111100110';
    elseif run==13 & category==6
        basecode='1111111111100111';
    elseif run==13 & category==7
        basecode='1111111111101000';
    elseif run==13 & category==8
        basecode='1111111111101001';
    elseif run==13 & category==9
        basecode='1111111111101010';
    elseif run==13 & category==10
        basecode='1111111111101011';

    %%%%%%%14s%%%%%%%%%%
    elseif run==14 & category==1
        basecode='111111110110';
    elseif run==14 & category==2
        basecode='1111111111101100';
    elseif run==14 & category==3
        basecode='1111111111101101';
    elseif run==14 & category==4
        basecode='1111111111101110';
    elseif run==14 & category==5
        basecode='1111111111101111';
    elseif run==14 & category==6
        basecode='1111111111110000'; %%%
    elseif run==14 & category==7
        basecode='1111111111110001'; %%%%
    elseif run==14 & category==8
        basecode='1111111111110010';
    elseif run==14 & category==9
        basecode='1111111111110011';
    elseif run==14 & category==10
        basecode='1111111111110100'; %%%
    %%%%%%%15s%%%%%%%%%%
    elseif run==15 & category==0
        basecode='11111111111111111'; %%%
    elseif run==15 & category==1
        basecode='1111111111110101';
    elseif run==15 & category==2
        basecode='1111111111110110';
    elseif run==15 & category==3
        basecode='1111111111110111';
    elseif run==15 & category==4
        basecode='1111111111111000';
    elseif run==15 & category==5
        basecode='1111111111111001';
    elseif run==15 & category==6
        basecode='1111111111111010';
    elseif run==15 & category==7
        basecode='1111111111111011';
    elseif run==15 & category==8
        basecode='1111111111111100'; %%%
    elseif run==15 & category==9
        basecode='1111111111111101'; %%%
    elseif run==15 & category==10
        basecode='1111111111111110';
    end
```

```
%=================================================================================
% Advanced Multimedia Applications
% Title       : DC-Coefficients Huffman Table (tableDC.m)
% Description : DC Huffman Co-efficients of Luminance (used for Encoding)
%               Tables taken from [R.C Gonzalez, R.E Woods - "Digital Image
%               Processing, Prentice Hall(2008)]
% input       : Category
% output      : Base Codeword
%=================================================================================
function basecode=tableDC(category)

    % large conditional statement containing the category to basecode mapping
    if category==0
        basecode='00';
    elseif category==1
        basecode='010';
    elseif category==2
        basecode='011';
    elseif category==3
        basecode='100';
    elseif category==4
        basecode='101';
    elseif category==5
        basecode='110';
    elseif category==6
        basecode='1110';
    elseif category==7
        basecode='11110';
    elseif category==8
        basecode='111110';
    elseif category==9
        basecode='1111110';
    elseif category==10
        basecode='11111110';
    elseif category==11
        basecode='111111110';
    end
```

```
% ================================================================================
% Advanced Multimedia Applications
% Title       : AC-Huffman Decode tables (detableAC2.m)
% Description : AC Huffman coefficients of Luminance (table used for Decoding)
%                 Tables taken from [R.C Gonzalez, R.E Woods - "Digital Image
%                 Processing, Prentice Hall(2008)]
% input       : Codeword
% output      : [Runlength (length of zeros), Category]
% ================================================================================
function O=detableAC2(basecode)
    run=0;
    category=0;
    basecode= num2str(basecode);

    %%%%%%%%%zero%%%%%%%%%%
    if strcmp(basecode,'1010')==1
        run=0;  category=0;
    elseif strcmp(basecode,'00')==1
        run=0;    category=1;
    elseif strcmp(basecode,'01')==1
        run=0;  category=2;
    elseif strcmp(basecode,'100')==1
        run=0;  category=3;
    elseif strcmp(basecode,'1011')==1
        run=0;  category=4;
    elseif strcmp(basecode,'11010')==1
        run=0;  category=5;
    elseif strcmp(basecode,'111000')==1
        run=0;  category=6;
    elseif strcmp(basecode,'1111000')==1
        run=0;  category=7;
    elseif strcmp(basecode,'1111110110')==1
        run=0;  category=8;
    elseif strcmp(basecode,'1111111110000010')==1
        run=0;  category=9;
    elseif strcmp(basecode,'1111111110000011')==1
        run=0;  category=10;

    %%%%%%%ones%%%%%%%%%%
    elseif strcmp(basecode,'1100')==1
        run=1;  category=1;
    elseif strcmp(basecode,'111001')==1
        run=1;  category=2;
    elseif strcmp(basecode,'1111001')==1
        run=1;  category=3;
    elseif strcmp(basecode,'111110110')==1
        run=1;  category=4;
    elseif strcmp(basecode,'1111111010')==1
        run=1;  category=5;
    elseif strcmp(basecode,'1111111110000100')==1
        run=1;  category=6;
    elseif strcmp(basecode,'1111111110000101')==1
        run=1;  category=7;
    elseif strcmp(basecode,'1111111110000110')==1
        run=1;  category=8;
    elseif strcmp(basecode,'1111111110000111')==1
        run=1;  category=9;
    elseif strcmp(basecode,'1111111110001000')==1
        run=1;  category=10;

    %%%%%%%twos%%%%%%%%%%
    elseif strcmp(basecode,'11011')==1
        run=2;  category=1;
    elseif strcmp(basecode,'11111000')==1
        run=2;  category=2;
    elseif strcmp(basecode,'1111110111')==1
        run=2;  category=3;
    elseif strcmp(basecode,'1111111110001001')==1
        run=2;  category=4;
```

```
elseif strcmp(basecode,'1111111110001010')==1
    run=2;  category=5;
elseif strcmp(basecode,'1111111110001011')==1
    run=2;  category=6;
elseif strcmp(basecode,'1111111110001100')==1
    run=2;  category=7;
elseif strcmp(basecode,'1111111110001101')==1
    run=2;  category=8;
elseif strcmp(basecode,'1111111110001110')==1
    run=2;  category=9;
elseif strcmp(basecode,'1111111110001111')==1
    run=2;  category=10;

%%%%%%%threes%%%%%%%%%%
elseif strcmp(basecode,'111010')==1
    run=3;  category=1;
elseif strcmp(basecode,'111110111')==1
    run=3;  category=2;
elseif strcmp(basecode,'11111110111')==1
    run=3;  category=3;
elseif strcmp(basecode,'1111111110010000')==1
    run=3;  category=4;
elseif strcmp(basecode,'1111111110010001')==1
    run=3;  category=5;
elseif strcmp(basecode,'1111111110010010')==1
    run=3;  category=6;
elseif strcmp(basecode,'1111111110010011')==1
    run=3;  category=7;
elseif strcmp(basecode,'1111111110010100')==1
    run=3;  category=8;
elseif strcmp(basecode,'1111111110010101')==1
    run=3;  category=9;
elseif strcmp(basecode,'1111111110010110')==1
    run=3;  category=10;

%%%%%%%fours%%%%%%%%%%
elseif strcmp(basecode,'111011')==1
    run=4;  category=1;
elseif strcmp(basecode,'1111111000')==1
    run=4;  category=2;
elseif strcmp(basecode,'1111111110010111')==1
    run=4;  category=3;
elseif strcmp(basecode,'1111111110011000')==1
    run=4;  category=4;
elseif strcmp(basecode,'1111111110011001')==1
    run=4;  category=5;
elseif strcmp(basecode,'1111111110011010')==1 %%%
    run=4;  category=6;
elseif strcmp(basecode,'1111111110011011')==1
    run=4;  category=7;
elseif strcmp(basecode,'1111111110011100')==1 %%%
    run=4;  category=8;
elseif strcmp(basecode,'1111111110011101')==1
    run=4;  category=9;
elseif strcmp(basecode,'1111111110011110')==1
    run=4;  category=10;

%%%%%%%fives%%%%%%%%%%
elseif strcmp(basecode,'1111010')==1
    run=5;  category=1;
elseif strcmp(basecode,'1111111001')==1
    run=5;  category=2;
elseif strcmp(basecode,'1111111110011111')==1
    run=5; category=3;
elseif strcmp(basecode,'1111111110100000')==1
    run=5;  category=4;
elseif strcmp(basecode,'1111111110100001')==1
    run=5;  category=5;
elseif strcmp(basecode,'1111111110100010')==1
```

```
        run=5;  category=6;
    elseif strcmp(basecode,'1111111110100011')==1
        run=5;  category=7;
    elseif strcmp(basecode,'1111111110100100')==1
        run=5;  category=8;
    elseif strcmp(basecode,'1111111110100101')==1
        run=5;  category=9;
    elseif strcmp(basecode,'1111111110100110')==1
        run=5;  category=10;

    %%%%%%%sixes%%%%%%%%%%
    elseif strcmp(basecode,'1111011')==1
        run=6;  category=1;
    elseif strcmp(basecode,'11111111000')==1
        run=6;  category=2;
    elseif strcmp(basecode,'1111111110100111')==1
        run=6;  category=3;
    elseif strcmp(basecode,'1111111110101000')==1
        run=6;  category=4;
    elseif strcmp(basecode,'1111111110101001')==1
        run=6;  category=5;
    elseif strcmp(basecode,'1111111110101010')==1
        run=6;  category=6;
    elseif strcmp(basecode,'1111111110101011')==1
        run=6;  category=7;
    elseif strcmp(basecode,'1111111110101100')==1
        run=6;  category=8;
    elseif strcmp(basecode,'1111111110101101')==1
        run=6;  category=9;
    elseif strcmp(basecode,'1111111110101110')==1
        run=6;  category=10;

    %%%%%%%sevens%%%%%%%%%%
    elseif strcmp(basecode,'11111001')==1
        run=7;  category=1;
    elseif strcmp(basecode,'11111111001')==1
        run=7;  category=2;
    elseif strcmp(basecode,'1111111110101111')==1
        run=7;  category=3;
    elseif strcmp(basecode,'1111111110110000')==1
        run=7;  category=4;
    elseif strcmp(basecode,'1111111110110001')==1
        run=7;  category=5;
    elseif strcmp(basecode,'1111111110110010')==1
        run=7;  category=6;
    elseif strcmp(basecode,'1111111110110011')==1
        run=7;  category=7;
    elseif strcmp(basecode,'1111111110110100')==1
        run=7;  category=8;
    elseif strcmp(basecode,'1111111110110101')==1 %%%%
        run=7;  category=9;
    elseif strcmp(basecode,'1111111110110110')==1
        run=7;  category=10;

    %%%%%%%eights%%%%%%%%%%
    elseif strcmp(basecode,'11111010')==1
        run=8;  category=1;
    elseif strcmp(basecode,'111111111000000')==1
        run=8;  category=2;
    elseif strcmp(basecode,'1111111101110111')==1
        run=8;  category=3;
    elseif strcmp(basecode,'1111111110111000')==1
        run=8;  category=4;
    elseif strcmp(basecode,'1111111110111001')==1
        run=8;  category=5;
    elseif strcmp(basecode,'1111111110111010')==1
        run=8;  category=6;
    elseif strcmp(basecode,'1111111110111011')==1
        run=8;  category=7;
    elseif strcmp(basecode,'1111111110111100')==1
        run=8;  category=8;
    elseif strcmp(basecode,'1111111110111101')==1
        run=8;  category=9;
    elseif strcmp(basecode,'1111111110111110')==1 %%%
        run=8;  category=10;

    %%%%%%%nines%%%%%%%%%%
    elseif strcmp(basecode,'111111000')==1
        run=9;  category=1;
    elseif strcmp(basecode,'1111111110111111')==1
        run=9;  category=2;
    elseif strcmp(basecode,'1111111111000000')==1
        run=9;  category=3;
    elseif strcmp(basecode,'1111111111000001')==1
        run=9;  category=4;
    elseif strcmp(basecode,'1111111111000010')==1
        run=9;  category=5;
    elseif strcmp(basecode,'1111111111000011')==1
        run=9;  category=6;
    elseif strcmp(basecode,'1111111111000100')==1
        run=9;  category=7;
    elseif strcmp(basecode,'1111111111000101')==1
        run=9;  category=8;
    elseif strcmp(basecode,'1111111111000110')==1
        run=9;  category=9;
    elseif strcmp(basecode,'1111111111000111')==1
        run=9;  category=10;

    %%%%%%%tens%%%%%%%%%%
    elseif strcmp(basecode,'111111001')==1
        run=10;  category=1;
    elseif strcmp(basecode,'1111111111001000')==1
        run=10;  category=2;
    elseif strcmp(basecode,'1111111111001001')==1
        run=10;  category=3;
    elseif strcmp(basecode,'1111111111001010')==1
        run=10;  category=4;
    elseif strcmp(basecode,'1111111111001011')==1
        run=10;  category=5;
    elseif strcmp(basecode,'1111111111001100')==1
        run=10;  category=6;
    elseif strcmp(basecode,'1111111111001101')==1
        run=10;  category=7;
    elseif strcmp(basecode,'1111111111001110')==1
        run=10;  category=8;
    elseif strcmp(basecode,'1111111111001111')==1
        run=10;  category=9;
    elseif strcmp(basecode,'1111111111010000')==1
        run=10;  category=10;

    %%%%%%%elevens%%%%%%%%%%
    elseif strcmp(basecode,'111111010')==1
        run=11;  category=1;
    elseif strcmp(basecode,'1111111111010001')==1
        run=11;  category=2;
    elseif strcmp(basecode,'1111111111010010')==1
        run=11;  category=3;
    elseif strcmp(basecode,'1111111111010011')==1
        run=11;  category=4;
    elseif strcmp(basecode,'1111111111010100')==1
        run=11;  category=5;
    elseif strcmp(basecode,'1111111111010101')==1
        run=11;  category=6;
    elseif strcmp(basecode,'1111111111010110')==1
        run=11;  category=7;
    elseif strcmp(basecode,'1111111111010111')==1
        run=11;  category=8;
    elseif strcmp(basecode,'1111111111011000')==1
```

```
        run=11;  category=9;
    elseif strcmp(basecode,'1111111111011001')==1
        run=11;  category=10;

    %%%%%%%twelevs%%%%%%%%%%
    elseif strcmp(basecode,'11111111111111110')==1 %%%
        run=12;  category=1;
    elseif strcmp(basecode,'1111111111011010')==1
        run=12;  category=2;
    elseif strcmp(basecode,'1111111111011011')==1
        run=12;  category=3;
    elseif strcmp(basecode,'1111111111011100')==1
        run=12;  category=4;
    elseif strcmp(basecode,'1111111111011101')==1
        run=12;  category=5;
    elseif strcmp(basecode,'111111111101110')==1
        run=12;  category=6;
    elseif strcmp(basecode,'1111111111011111')==1
        run=12;  category=7;
    elseif strcmp(basecode,'1111111111100000')==1 %%%
        run=12;  category=8;
    elseif strcmp(basecode,'1111111111100001')==1
        run=12;  category=9;
    elseif strcmp(basecode,'1111111111100010')==1
        run=12;  category=10;

    %%%%%%%13s%%%%%%%%%%
    elseif strcmp(basecode,'11111111010')==1
        run=13;  category=1;
    elseif strcmp(basecode,'1111111111100011')==1
        run=13; category=2;
    elseif strcmp(basecode,'1111111111100100')==1
        run=13;  category=3;
    elseif strcmp(basecode,'1111111111100101')==1
        run=13;  category=4;
    elseif strcmp(basecode,'1111111111100110')==1
        run=13;  category=5;
    elseif strcmp(basecode,'1111111111100111')==1
        run=13;  category=6;
    elseif strcmp(basecode,'1111111111101000')==1
        run=13;  category=7;
    elseif strcmp(basecode,'1111111111101001')==1
        run=13;  category=8;
    elseif strcmp(basecode,'1111111111101010')==1
        run=13;  category=9;
    elseif strcmp(basecode,'1111111111101011')==1
        run=13;  category=10;

    %%%%%%%14s%%%%%%%%%%
    elseif strcmp(basecode,'111111110110')==1
        run=14;  category=1;
    elseif strcmp(basecode,'1111111111101100')==1
        run=14;  category=2;
    elseif strcmp(basecode,'1111111111101101')==1
        run=14;  category=3;
    elseif strcmp(basecode,'1111111111101110')==1
        run=14;  category=4;
    elseif strcmp(basecode,'1111111111101111')==1
        run=14;  category=5;
    elseif strcmp(basecode,'1111111111110000')==1 %%%
        run=14;  category=6;
    elseif strcmp(basecode,'1111111111110001')==1 %%%
        run=14;  category=7;
    elseif strcmp(basecode,'1111111111110010')==1
        run=14;  category=8;
    elseif strcmp(basecode,'1111111111110011')==1
        run=14;  category=9;
    elseif strcmp(basecode,'1111111111110100')==1 %%%
        run=14; category=10;
    %%%%%%%15s%%%%%%%%%%
    elseif strcmp(basecode,'11111111111111111')==1%%%
        run=15;  category=0;
    elseif strcmp(basecode,'1111111111110101')==1
        run=15;  category=1;
    elseif strcmp(basecode,'1111111111110110')==1
        run=15;  category=2;
    elseif strcmp(basecode,'1111111111110111')==1
        run=15;  category=3;
    elseif strcmp(basecode,'1111111111111000')==1
        run=15;  category=4;
    elseif strcmp(basecode,'1111111111111001')==1
        run=15;  category=5;
    elseif strcmp(basecode,'1111111111111010')==1
        run=15;  category=6;
    elseif strcmp(basecode,'1111111111111011')==1 %%%
        run=15;  category=7;
    elseif strcmp(basecode,'1111111111111100')==1 %%%
        run=15;  category=8;
    elseif strcmp(basecode,'1111111111111101')==1 %%%
        run=15;  category=9;
    elseif strcmp(basecode,'1111111111111110')==1
        run=15;  category=10;
    else
        category=11; %if can not match with the input, return to 11
    end


    O=zeros(1,2);
    O(1,1)=run;
    O(1,2)=category;

end
```

```
% ==================================================================================
% Advanced Multimedia Applications
% Title       : Huffman Decoding Table for DC-coefficients (detableDC.m)
% Description : DC Huffman Co-efficients of Luminance (used for Decoding)
%                 Tables taken from [R.C Gonzalez, R.E Woods - "Digital Image
%                 Processing, Prentice Hall(2008)]
% Input       : Base Codeword
% Output      : Category
% ==================================================================================
function category=detableDC(basecode)

    % convert number to string
    basecode= num2str(basecode);

    % large conditional statement
    if strcmp(basecode,'00')==1
        category=0;
    elseif strcmp(basecode,'010')==1
        category=1;
    elseif strcmp(basecode,'011')==1
        category=2;
    elseif strcmp(basecode,'100')==1
        category=3;
    elseif strcmp(basecode,'101')==1
        category=4;
    elseif strcmp(basecode,'110')==1
        category=5;
    elseif strcmp(basecode,'1110')==1
        category=6;
    elseif strcmp(basecode,'11110')==1
        category=7;
    elseif strcmp(basecode,'111110')==1
```

```
        category=8;
    elseif strcmp(basecode,'1111110')==1
        category=9;
    elseif strcmp(basecode,'11111110')==1
        category=10;
    elseif strcmp(basecode,'111111110')==1
        category=11;
    else
        category=12; %if can not match with the input, return 12
    end
```

```
%================================================================================
% Advanced Multimedia Applications
% Title          : Symbol-2 to Amplitude Converter(sym2_to_amp.m)
% Description    : Converts an AC/DC Symbol-2 to signed integer format
% Input          : symbol-2, category
% Output         : Codeword
%================================================================================
function amp = sym2_to_amp(sym2, cat)

        % check for positive values
        if(sym2(1) == '1')
            amp = bin2dec(sym2);

        % check for negative values
        elseif(sym2(1) == '0')

            % add a '1' at the front before converting to binary
            ones_comp_str = ['1' sym2];
            dec = bin2dec(ones_comp_str);
            bin = dec2bin(bitxor(dec,2^16-1)); % 16bit binary xor(compliment)

            % use 'cat' to get the lower number of bits
            bin_low_bits = bin(end-(cat-1):end);

            % get corresponding negative value of the decimal
            amp = bin2dec(bin_low_bits) *-1;

        % throw error
        else
            error('unknown sym2');
        end
end
```

```
% ================================================================================
% Advanced Multimedia Applications
% Title       : Amplitude-to-Symbol2 Conversion (amp_to_sym2.m)
% Description : Converts an Amplitude value to a Symbol-2 binary code
% Input       : Amplitude, Category
% Output      : Symbol-2
% ================================================================================
function sym2 = amp_to_sym2(amp,cat)

    if(amp == 0) % special case for zero
        sym2 = '0';
    elseif (amp > 0)
        sym2 = dec2bin(amp,cat);
    elseif (amp < 0)

        % need to typecast and get 2's compliment first
        twos_comp = dec2bin(bitxor(uint16(abs(amp)),2^16-1));
        sym2 = twos_comp(end-(cat-1):end);

    else
```

```
        amp
        error('unknown amp');
    end

end
```

```
% ================================================================================
% Advanced Multimedia Applications
% Title       : Extract DC Coefficients (extract_dc_coeffs.m)
% Description : Take out the DC coefficients(1,1) from a given cell
% Input       : cell (combination of matrices)
% Output      : Array of DC-coeffs of each sub-matrix
% ================================================================================
function all_dc_coeffs = extract_dc_coeffs(cell)

    % invert
    cell = cell';

    % run a loop to get the (1,1) value from each sub-matrix
    for  i = 1:(length(cell)*length(cell))
        all_dc_coeffs(i) = cell{i}(1,1);
    end
end
```

10.6 Appendix-F

10.6.1 1D-DCT Test Script

```
%==================================================================================
% Advanced Multimedia Applications
% Title       : Test Script for Simple and Fast DCT (1D) (Test_1D_DCT.m)
% Description : Testing using a random 8 sample array.
%               Checking time taken by each function
%==================================================================================
clear;clc;

% -------- simple 1D-DCT --------

data = [-88   121   113     3    66   -13   -50    77]% original input data

datasize = 8; % number of data items
%data = round(-128 + (128+128).*rand(datasize,1))'

tic  % start timer
dct_coeffs = dct_1(data)             % perform 1D-DCT
idata = round(idct_1(dct_coeffs))    % perform 1D-IDCT
toc  % stop timer

% check if original data is the same as reconstructed data
if(any(data == round(idata)))
    fprintf('1D-DCT successful\n');
else
    fprintf('1D-DCT unsuccessful\n');
end

% -------- simple 1D-FDCT --------

tic  % start timer
fdct_coeffs = fdct_1(data);      % perform 1D-FDCT
idata = fidct_1(fdct_coeffs);    % perform 1D-IDCT
toc  % stop timer

% check if original data is the same as reconstructed data
if(any(data == round(idata)))
    fprintf('1D-FDCT successful\n');
else
    fprintf('1D-FDCT unsuccessful\n');
end
```

10.6.2 2D-DCT – Block DCT Analysis Test Script

```
%==================================================================================
% Advanced Multimedia Applications
% Title         : 2D-DCT Image analysis Test Script (Test_2D_DCT.m)
% Description   : Test Script for 2D_DCT Image Analysis. Analysis is done
%                 block by block for the whole image.
%==================================================================================

clc;clear;close all;

%======================================================================
%       T E S T   - S C R I P T   P A R A M E T E R S
%======================================================================
img = imread('lena512.bmp');
imshow(img);

% img = imread('Cameraman.bmp');
% imshow(img);

img = double(img)-128;
[m,n] = size(img);
b = 8;

num_blocks = (m*n)/(b*b);
num_hor_blks = m/b;
num_ver_blks = n/b;

speed   = 0;     % 0-simple dct, 1-fast dct

%=====================================================================
%       INITIALISATION
%=====================================================================

% split the input larger matrix into smaller 8x8 blocks
split_img = mat2cell(img, b*ones(m/b,1), b*ones(1,n/b));

% subplot index
sp=1;


%=====================================================================
%       DISCRETE COSINE TRANSFORM
%=====================================================================
tstart = tic;

% perform DCT --> IDCT for each block
for i = 1:num_ver_blks
    for j = 1:num_hor_blks

        % --------------- 2D-DCT -------------------------------------
        % perform DCT - Encoder side
        % (output is a array of cells)
        enc_dct_block = dct_2_8x8(split_img{i,j},b,speed)';
        test_enc_dct_block{i,j} = enc_dct_block;


%=====================================================================
%       INVERSE - DISCRETE COSINE TRANSFORM
%=====================================================================

        % --------------- 2D - Inverse - DCT -------------------------
        dec_idct_block{i,j} = idct_2_8x8(enc_dct_block,b, speed)';
        reconstructed_cell{i,j} = dec_idct_block{i,j};

        display(['Finished Decoding block=',num2str(i),',',num2str(j)])

    end
end

% get time taken for system
telapsed = toc(tstart)

% convert the array of cells back to a matrix
reconstructed_img = cell2mat(reconstructed_cell);

% add back the 128, and convert to uint8 (to be displayed)
reconstructed_img = reconstructed_img + 128;
reconstructed_img = uint8(reconstructed_img);

% display reconstructed picture
```

```
figure;imshow(reconstructed_img);

%imshow(abs(split_img{1,1}));

%======================================================================
%       OUTPUT TEST RESULTS
%======================================================================

% ---  work on block :(i,j) ---
i=30;j=32;

% original block matrix
img_block = split_img{i,j}

% output of 2D-DCT
dct_2_8x8 = test_enc_dct_block{i,j}

% print out basic transform output
dct_2_simple = dct_2_simple(split_img{i,j})

% output of 2D-IDCT
idct_2_8x8 = round(dec_idct_block{i,j})

% are they equal ??
img_block == idct_2_8x8

% ---  analyse a combination of blocks :(i,j) ---

% subplot the dct blocks (somewhere around the middle of the image)
for i = 30:39
    for j = 30:39

        subplot(10,10,sp);
        imshow(abs(test_enc_dct_block{i,j}));
        sp=sp+1;

        display(['Finished plotting block=',num2str(i),',',num2str(j)])

    end
end

figure;sp=1;
% subplot the original blocks
for i = 30:39
    for j = 30:39

        subplot(10,10,sp);
        imshow(uint8(split_img{i,j}+128));
        sp=sp+1;

        display(['Finished plotting block=',num2str(i),',',num2str(j)])

    end
end
```

10.6.3 2D-DCT – Speed Analysis (Simple vs. Row,Col Method)

```
%==================================================================================
% Advanced Multimedia Applications
% Title         : 2D-DCT Speed Test (Test_2DDCT_Speed.m)
% Description   : Test Script for 2D_DCT - to measure speed between the
%                 row and column wise 2D-DCT and the 2D-DCT using basic
%                 transform equations.
%=============================================================================

clc;clear;close all;
%=====================================================================
%       T E S T  -  S C R I P T   P A R A M E T E R S
%=====================================================================

max_datasize = 128; % maximum datasize to check
speed = 0;        %(using simple, 1D-DCT equation on rows and colums)


%=====================================================================
%       T E S T  -  M A I N   L O O P
%=====================================================================

i=1;
for datasize=8:8:max_datasize

    % the data - a random matrix
    data = round(-128 + (128+128).*rand(datasize,datasize))';

    % perform the 2D-DCT - using the 2D-DCT basic transform equation
    tstart = tic;
    dct2_s  = dct_2_simple(data);
    telapsed_dct2_s(i) = toc(tstart);

    % perform the 2D-DCT - using 1D-DCT transform equations
    % rows first then columns.
    tstart = tic;
    dct2_rc = dct_2_8x8(data,datasize,speed)';
    telapsed_dct2_rc(i) = toc(tstart);

    i=i+1;

    display(['Finished Test, datasize=',num2str(datasize)])

end

% display speed results on graph
% plot in logarithmic scale
datasize=[8:8:max_datasize];
semilogy(datasize,telapsed_dct2_rc,'b');
hold on;
semilogy(datasize,telapsed_dct2_s,'r');
grid on;
```

10.6.4 1D-DCT Speed Analysis Test Script

```
%=============================================================================
% Advanced Multimedia Applications
% Title       : Test Script for Simple and Fast DCT (1D) (Test_DCT_Speed.m)
% Description : Test Script to Measure speed of FAST-DCT vs. SIMPLE-DCT (1D)
%               Measuring speed over several test cases(getting average)
%=============================================================================
clc;clear;close all;

% ----- parameters -----
datasize = 8; % number of data items
sample_size = 10000; % number of speed measurements taken

for i=1:sample_size

    % generate random 8 numbers between -128 and 127
    data = round(-128 + (128+128).*rand(datasize,1))';

    % --- run simple-dct/idct ---
    tstart = tic;  % start timer
    dct_coeffs = dct_1(data);           % perform 1D-DCT
    idata = round(idct_1(dct_coeffs));  % perform 1D-IDCT
    simple_telapsed(i) = toc(tstart);   % stop timer

    % if original data is different to reconstructed data then inform
    if(idata ~= data)
        data
        idata
        error('simple-dct-operation failed');
    end

    % --- run fast-dct/idct ---
    tstart = tic;  % start timer
    dct_coeffs = fdct_1(data);       % perform 1D-DCT
    idata = fidct_1(dct_coeffs);     % perform 1D-IDCT
    fast_telapsed(i) = toc(tstart);  % stop timer

    % if original data is different to reconstructed data then inform
    if(idata ~= data)
        data
        idata
        error('fast-dct-operation failed');
    end

end

% calculate average of results
mean_time_difference = mean(simple_telapsed-fast_telapsed)

% plot results
plot(simple_telapsed-fast_telapsed,'b');
title('Simple vs. Fast DCT/IDCT Execution Speed Difference, No. of Tests =10000');
ylabel('Elapsed Time Difference');xlabel('Test No.');
grid on;
```

10.6.5 Quantisation Test Script (to obtain Error statistics)

```
%==================================================================================
% Advanced Multimedia Applications
% Title       : Test Script for Quantiser/Inverse Quantiser (Test_Quantiser.m)
% Description : Test Script to verify the output of the quantiser functions, and
%               to obtain error statistics after quantisation.
%==================================================================================

clc;clear

%======================================================================
%       TEST - SCRIPT PARAMETERS
%======================================================================

img = imread('york.bmp');
imshow(img);


img = double(img)-128;
[m,n] = size(img);
b = 8;
zer_count =0;
speed  = 1;     % 0-simple dct, 1-fast dct
```

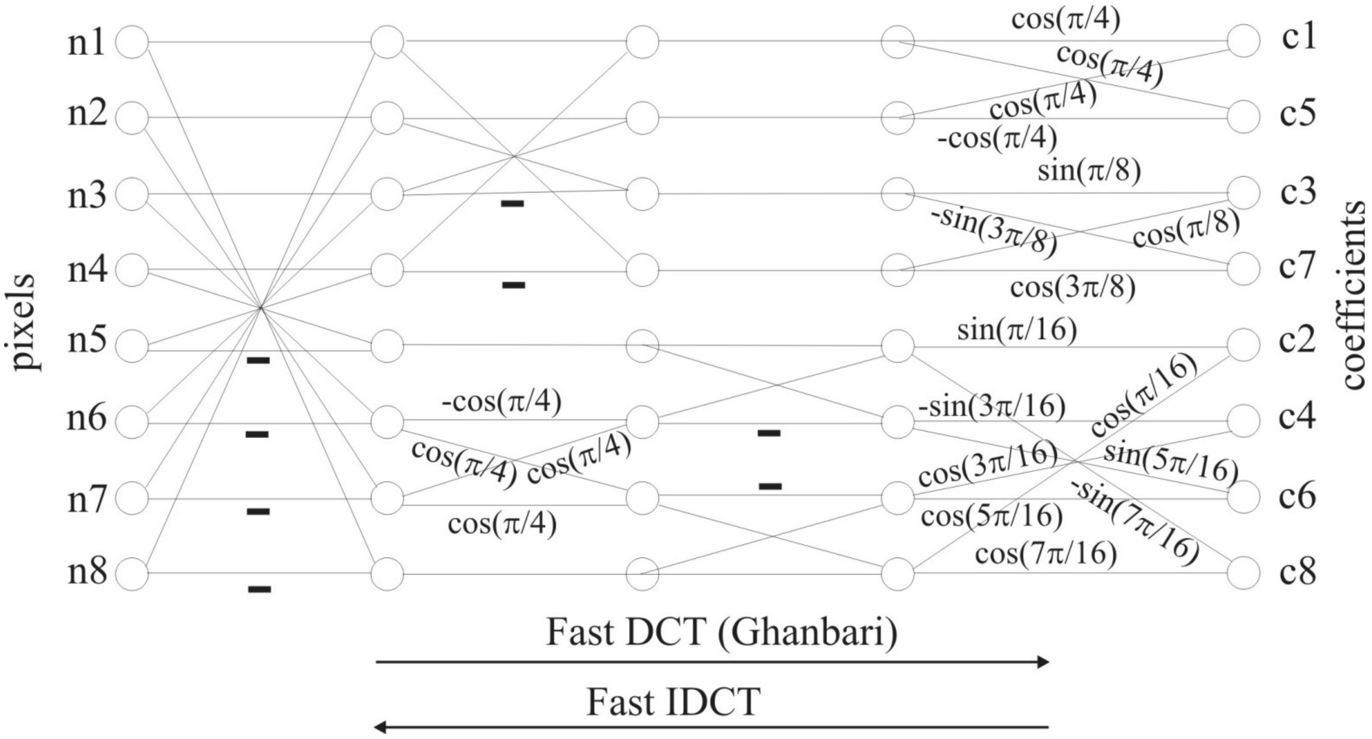
```
q_type = 'L';  % NL = Non-Linear, L=Linear

q_ratio = 90; % Non-Linear quantisation ratio
L_step_size = 32; % Linear quantisation step size
L_cut_off =8; % Linear quantisation cut off level
%=====================================================================
%       E N C O D E R
%=====================================================================

% split the input larger matrix into smaller 8x8 blocks
split_img = mat2cell(img, b*ones(m/b,1), b*ones(1,n/b));

% perform DCT->Quantisation->Inverse-Quantisation->IDCT for each block
for i = 1:length(split_img)
    for j = 1:length(split_img)

        % --------------- 2D-DCT ---------------------------------------------
        % perform DCT - Encoder side
        % output is a array of cells
        enc_dct_block = dct_2_8x8(split_img{i,j},b,speed);

        % --------------- Quantisation ---------------------------------------
        if (q_type == 'NL')
            [enc_q_block,zer] =  NL_Quantizer(enc_dct_block,q_ratio,0);
            zer_count = zer_count+zer;
        elseif(q_type == 'L')
            [enc_q_block,zer] =  L_Quantizer(enc_dct_block,0,L_step_size,L_cut_off);
            zer_count = zer_count+zer;
        else
            error('unknown q_type');
        end


    %=====================================================================
    %       D E C O D E R
    %=====================================================================

        % --------------- Inverse - Quantisation------------------------------
        if (q_type == 'NL')
            dec_q_block =  NL_Quantizer(enc_q_block,q_ratio,1);
        elseif(q_type == 'L')
            dec_q_block = L_Quantizer(enc_q_block,1,L_step_size,L_cut_off);
        else
            error('unknown q_type');
        end

    % --------------- 2D - Inverse - DCT ---------------------------------

        dec_idct_block = idct_2_8x8(dec_q_block,b, speed);
        reconstructed_cell{i,j} = dec_idct_block;

    end
end
reconstructed_img = cell2mat(reconstructed_cell);     % output is a matrix
quant_error = img - reconstructed_img; % cheching error

% add back the 128, and convert to uint8
reconstructed_img = reconstructed_img + 128;
reconstructed_img = uint8(reconstructed_img);

% display picture
%-----------------------
figure;imshow(reconstructed_img);


 MSE = sum(sum((quant_error.* quant_error)))/(m*n); % calculating MSE
 SNR = 10*log10(var(reshape(quant_error',1,m*n))/MSE);% calculating SNR
 PeakSNR = 10*log10((255^2)/MSE);% calculating peak SNR
 disp(strcat('MSE =',num2str(MSE)));
 disp(strcat('SNR =',num2str(SNR)));
 disp(strcat('PeakSNR =',num2str(PeakSNR)));
 disp(strcat('Zero% =',num2str((zer_count/(256*256))*100)));
% imwrite(reconstructed_img,'C:\Users\xxx\Desktop\nl\test.bmp');
```

10.6.6 ZigZag Test Script

```
% ================================================================================
% Advanced Multimedia Applications
% Title       : Test app for Zigzag encoder and decoder (zigzagtest.m)
%
% Description : The application makes an input 8x8 matrix with incrimenting values.
%               Then the test matrix is zigzag encoded and the result decoded using
%               zigzag decoder. Then, all three matrixs are displayed and the test
%               matrix and recustructed matrix are checked for errors.
% ================================================================================

test_m = ones(8,8); % initialsising input matrix
for i = 1 : 8
    for j = 1 : 8
        test_m(i,j) = ((i-1)*8)+j; % setting incrimenting values to matrix
    end;
end;

temp_m1 = zigzag_e(test_m); % zigzag encoding test matrix
temp_m2 = zigzag_d(temp_m1);% decoding original matrix from zigzag encoded matrix

% display matrix contents
disp('Original matrix');
disp(test_m);
disp('');
disp('Zigzag encoded matrix');
disp(temp_m1);
disp('');
disp('Reconstructed matrix');
disp(temp_m2);

% checking whether test matrix and recunstructed matrix are same
if (isequal(temp_m2,test_m))
    disp('Success: Perfect functioning');
else
    disp('Error');
end;
```

10.7 Appendix-G

Butterfly graph used for Fast DCT/IDCT (from Ghanbari)



10.8 Appendix-H

Luminance – Quantisation Table (Q50)extracted from [2].

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

10.9 Appendix-I

AC-Tables – (RUN, CAT)=CODEWORD, Adapted from [3] Gonzales DIP.

```
(0,0)='1010' (EOB)
(0,1)='00'
(0,2)='01'
(0,3)='100'
(0,4)='1011'
(0,5)='11010'
(0,6)='111000'
(0,7)='1111000'
(0,8)='1111110110'
(0,9)='1111111110000010'
(0,10)='1111111110000011'


(1,1)='1100'
(1,2)='111001'
(1,3)='1111001'
(1,4)='111110110'
(1,5)='1111111010'
(1,6)='1111111110000100'
(1,7)='1111111110000101'
(1,8)='1111111110000110'
(1,9)='1111111110000111'
(1,10)='1111111110001000'


(2,1)='11011'
(2,2)='11111000'
(2,3)='1111110111'
(2,4)='1111111110001001'
(2,5)='1111111110001010'
(2,6)='1111111110001011'
(2,7)='1111111110001100'
(2,8)='1111111110001101'
(2,9)='1111111110001110'
(2,10)='1111111110001111'

(3,1)='111010'
(3,2)='111110111'
(3,3)='11111110111'
(3,4)='1111111110010000'
(3,5)='1111111110010001'
(3,6)='1111111110010010'
(3,7)='1111111110010011'
(3,8)='1111111110010100'
(3,9)='1111111110010101'
(3,10)='1111111110010110'


(4,1)='111011'
(4,2)='1111111000'
(4,3)='1111111110010111'
(4,4)='1111111110011000'
(4,5)='1111111110011001'
(4,6)='1111111110011010'
(4,7)='1111111110011011'
(4,8)='1111111110011100'
(4,9)='1111111110011101'
(4,10)='1111111110011110'
```

```
(5,1)='1111010'
(5,2)='1111111001'
(5,3)='1111111110011111'
(5,4)='1111111110100000'
(5,5)='1111111110100001'
(5,6)='1111111110100010'
(5,7)='1111111110100011'
(5,8)='1111111110100100'
(5,9)='1111111110100101'
(5,10)='1111111110100110'

(6,1)='1111011'
(6,2)='11111111000'
(6,3)='1111111110100111'
(6,4)='1111111110101000'
(6,5)='1111111110101001'
(6,6)='1111111110101010'
(6,7)='1111111110101011'
(6,8)='1111111110101100'
(6,9)='1111111110101101'
(6,10)='1111111110101110'

(7,1)='11111001'
(7,2)='11111111001'
(7,3)='1111111110101111'
(7,4)='1111111110110000'
(7,5)='1111111110110001'
(7,6)='1111111110110010'
(7,7)='1111111110110011'
(7,8)='1111111110110100'
(7,9)='1111111110110101'
(7,10)='1111111110110110'

(8,1)='11111010'
(8,2)='111111111000000'
(8,3)='1111111101110111'
(8,4)='1111111110111000'
(8,5)='1111111110111001'
(8,6)='1111111110111010'
(8,7)='1111111110111011'
(8,8)='1111111110111100'
(8,9)='1111111110111101'
(8,10)='1111111110111110'

(9,1)='111111000'
(9,2)='1111111110111111'
(9,3)='1111111111000000'
(9,4)='1111111111000001'
(9,5)='1111111111000010'
(9,6)='1111111111000011'
(9,7)='1111111111000100'
(9,8)='1111111111000101'
(9,9)='1111111111000110'
(9,10)='1111111111000111'

(10,1)='111111001'
(10,2)='1111111111001000'
(10,3)='1111111111001001'
(10,4)='1111111111001010'
```

```
(10,5)='1111111111001011'
(10,6)='1111111111001100'
(10,7)='1111111111001101'
(10,8)='1111111111001110'
(10,9)='1111111111001111'
(10,10)='1111111111010000'

(11,1)='111111010'
(11,2)='1111111111010001'
(11,3)='1111111111010010'
(11,4)='1111111111010011'
(11,5)='1111111111010100'
(11,6)='1111111111010101'
(11,7)='1111111111010110'
(11,8)='1111111111010111'
(11,9)='1111111111011000'
(11,10)='1111111111011001'

(12,1)='11111111111111110'
(12,2)='1111111111011010'
(12,3)='1111111111011011'
(12,4)='1111111111011100'
(12,5)='1111111111011101'
(12,6)='1111111111011110'
(12,7)='1111111111011111'
(12,8)='1111111111100000'
(12,9)='1111111111100001'
(12,10)='1111111111100010'

(13,1)='11111111010'
(13,2)='1111111111100011'
(13,3)='1111111111100100'
(13,4)='1111111111100101'
(13,5)='1111111111100110'
(13,6)='1111111111100111'
(13,7)='1111111111101000'
(13,8)='1111111111101001'
(13,9)='1111111111101010'
(13,10)='1111111111101011'

(14,1)='111111110110'
(14,2)='1111111111101100'
(14,3)='1111111111101101'
(14,4)='1111111111101110'
(14,5)='1111111111101111'
(14,6)='1111111111110000'
(14,7)='1111111111110001'
(14,8)='1111111111110010'
(14,9)='1111111111110011'
(14,10)='1111111111110100'

(15,0)='11111111111111111'
(15,1)='1111111111110101'
(15,2)='1111111111110110'
(15,3)='1111111111110111'
(15,4)='1111111111111000'
(15,5)='1111111111111001'
(15,6)='1111111111111010'
(15,7)='1111111111111011'
```

```
(15,8)='1111111111111100'         (15,9)='1111111111111101'         (15,10)='1111111111111110'
```

DC-Tables (CAT = CODEWORD)

```
0='00'
1='010'
2='011'
3='100'
4='101'
5='110'
6='1110'
7='11110'
8='111110'
9='1111110'
10='11111110'
11='111111110'
```

Category of the baseline Encoder (Both for AC & DC)

```
0 = --
1 = -1,1
2 = -3, -2, 2, 3
3 = -7…-4, 4…7
4 = -15…-8,8…15
5 = -31…-16,16…31
6 = -63…-32,32…63
7 = -127…-64,64…127
8 = -255…-128,128…255
9 = -511…-256,256…511
10= -1023…-512,512…1023
11= -2047…-1024,1024…2047
```

10.10 Appendix – J

Original Images used for Test and Analysis:

lena512.bmp (512x512, 8-bit)

cameraman 256.pgm (256x256, 8-bit)

barb 256.pgm (512x512, 8-bit)

xilinx .bmp (512x512, 8-bit)

child .bmp (512x512, 8-bit)

smooth .bmp (512x512, 8-bit)