CS3IA16 – Image Analysis

Coursework Assignment 2 – Image Compression

Abstract

This report details the implementation, methodology, theory and results of implementing an image compression and decompression algorithm using MATLAB based on the JPEG compression scheme. The compression algorithm includes implementing down-sampling of the image, utilising the Discrete Cosine Transform (DCT) function, applying quantization to the relevant channels for luminance and chrominance, applying a zig-zag function to re-order the structure of the matrix elements and then utilising Huffman encoding on the re-structured matrix. The decompression algorithm includes implementing the inverse of the compression algorithm, decoding the Huffman encoded vector, re-constructing a matrix using the inverse of the zig-zag function, computing the inverse quantization and then applying the inverse DCT function. The results of the separate channels are concatenated together to form the final image and the compression ratio is calculated.

Introduction

The images used in this project in order to conduct image compression can be viewed in the two figures below:

Figure 1.
eng_HI.png

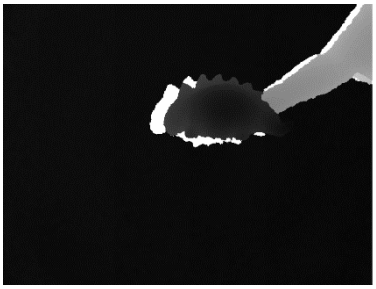

Figure 2.
kinect-dinosaur.png

The image shown in figure 1, named ‘eng_HI.png’ details a high quality RGB image with a size of 30.7MB. The second image detailed in figure highlighted a low-quality grayscale image with a size of 58KB. The two images were selected in order to ensure that the algorithm created would be able to both compress images in grayscale and RGB images.

Development

A typical compression scheme follows the concept displayed below in part a in figure 3.

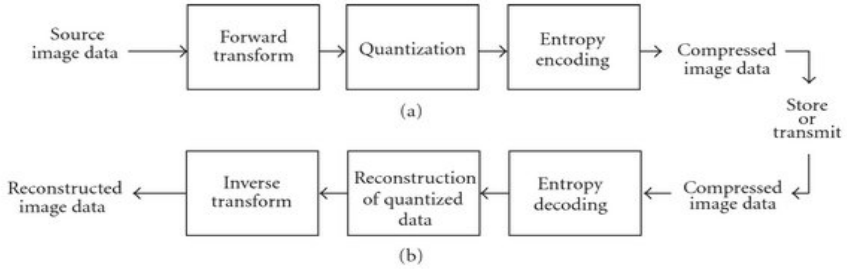

Figure 3.
Compression/Decompression Cycle

This concept details that input image data is firstly put into the compression algorithm and a forward transform is performed on the image data. In this case, the Discrete Cosine Transform (DCT) is used as a forward transform. Before applying this function to the image, the number of dimensions of the image must be considered. If the image is in grayscale, the image will only have one dimension, however if the image is in colour then it will have 3 dimensions. Bitmap images use R-G-B planes to represent colour images, however given that the human eye has different sensitivity to colour and brightness, R-G-B can be transformed to YCbCr space. Y corresponds to the luminance factor of an image, Cb corresponds to the blue chrominance and Cr corresponds to the red chrominance. By down-sampling the chrominance parts of the image, the amount of data in the image can be significantly reduced without losing the important data in which the eye is sensitive to. After down-sampling, the DCT function is then applied to the image data in 8x8 blocks of pixels. DCT works by separating the image data into parts of differing frequencies[1]. This is given by the following equation:

$$DCT(i,j) = \frac{1}{\sqrt{2N}} C(i)\, C(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} pixel(x,y) \cos\left[\frac{(2x+1)i\pi}{2N}\right] \cos\left[\frac{(2y+1)j\pi}{2N}\right]$$

$$C(x) = \frac{1}{\sqrt{2}} \text{ if } x \text{ is } 0, \text{ else } 1 \text{ if } x > 0$$

In order to be able to apply the DCT function to an image in 8x8 blocks, two functions from MATLAB are used. The first of these used is 'dctmtx(n)'[2]. This function returns an n by n DCT matrix, which can be used to perform a 2-D DCT on an image. An 8x8 DCT matrix, which is used in this compression algorithm can be seen by the image given in figure 4.

$$T = \begin{bmatrix} .3536 & .3536 & .3536 & .3536 & .3536 & .3536 & .3536 & .3536 \\ .4904 & .4157 & .2778 & .0975 & -.0975 & -.2778 & -.4157 & -.4904 \\ .4619 & .1913 & -.1913 & -.4619 & -.4619 & -.1913 & .1913 & .4619 \\ .4157 & -.0975 & -.4904 & -.2778 & .2778 & .4904 & .0975 & -.4157 \\ .3536 & -.3536 & -.3536 & .3536 & .3536 & -.3536 & -.3536 & .3536 \\ .2778 & -.4904 & .0975 & .4157 & -.4157 & -.0975 & .4904 & -.2778 \\ .1913 & -.4619 & .4619 & -.1913 & -.1913 & .4619 & -.4619 & .1913 \\ .0975 & -.2778 & .4157 & -.4904 & .4904 & -.4157 & .2778 & -.0975 \end{bmatrix}$$

Figure 4.
8x8 DCT Matrix

Another function provided by MATLAB is 'blockproc()'[3]. This function provides distinct block processing for an image, for a given size and function. Therefore by assigning a DCT function to multiply the DCT matrix by an 8x8 block of a given image, the transform can be processed across the entire image in blocks, converting this image into the frequency domain.

Once the image is converted into the frequency domain, the second step in the compression cycle will have been reached. During this step, the image must be quantized. Quantization works by dividing the

[1] Cabeen, K., Gent, P., Image Compression and the Discrete Cosine Transform (2008) [Online]. Available: https://www.math.cuhk.edu.hk/~lmlui/dct.pdf [Accessed 10 December 2018].

[2] MathWorks, Discrete Cosine Transform Matrix (2018) [Online]. Available: https://uk.mathworks.com/help/images/ref/dctmtx.html [Accessed 12 December 2018].

[3] MathWorks, Distinct Block Processing For Images (2018) [Online]. Available: https://uk.mathworks.com/help/images/ref/blockproc.html?s_tid=doc_ta [Accessed 12 December 2018].

values of each 8x8 block of the image by a relevant quantization matrix and rounding to the nearest integer. This is the main lossy operation of the process as high frequency components are removed from the data. As the insignificant high frequency data is discarded from the image, this compresses the image[4]. Given that the image has been previously separated into its Y, Cb and Cr components for down-sampling, different quantization matrices can be applied to the luminance and chrominance components[5]. The quantization matrix given for luminance and chrominance as shown in figures 5 and 6 below.

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

Figure 5.
Luminance Quantization Table

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Figure 6.
Chrominance Quantization Table

The matrices given above can be both scaled up or down in order to alter the quality factor (QF) of the image. By increasing the QF, significantly more data is preserved and therefore the size of the compressed then decompressed image is much higher. Conversely, if the QF is reduced, less data will be preserved, yet the size of the final image will smaller. Through conducting analysis regarding the difference in image quality and image size created once applying the compression/decompression technique, the ideal value for the QF can be found, which preserves the most image quality whilst also reducing the file size the greatest.

Having applied quantization to the matrices in 8x8 blocks, the entropy of the channels must be found. The initial step is to separate each channel into their relevant DC and AC coefficients, which are re-ordered from an 8x8 matrix into a 1x64 vector. This is achieved by iterating over each 8x8 block in a zigzag pattern , therefore grouping all of the zero-valued coefficients. The DC coefficient is directly correlated to the mean of the 8x8 block and the AC coefficients are the values of the cosine basis functions. In order to zigzag across the matrix, the order of the array traversal is defined and the function executes in a fashion similar to the image displayed below in figure 7.

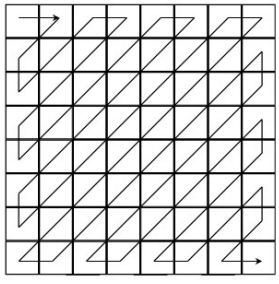

Figure 7.
Zigzag Ordering

[4] Roberts, E., Coefficient Quantization (2006) [Online]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossy/jpeg/coeff.htm [Accessed 13th December 2018]

[5] Girod, B., Image and Video Compression (2012) [Online]. Available: https://web.stanford.edu/class/ee398a/handouts/lectures/08-JPEG.pdf [Accessed 13th December 2018]

Once the matrix has been transformed into a vector, the information can be encoded using Huffman encoding in order to further reduce the size of the compressed image. Huffman encoding works to compress data by utilising fewer bits to encode more frequently occurring characters in the vector, therefore ensuring that not all characters use 8 bits. The variable length codes assigned to input characters are Prefix Codes. This ensures that a code assigned to one character is not the prefix of a code assigned to any other character. Considering this principle, there is no ambiguity when decoding the bitstream created through the encoding algorithm. In order to implement this transform, the function mat2huff() was utilised from Digital Image Processing Using MATLAB[6]. This function Huffman encodes a matrix, by using symbol probabilities in unit-width histogram bins between in input vector's minimum and maximum values. By applying this function to each component channel of Y, Cb and Cr, the image will be encoded and therefore compressed even further for use in transmission. Once this has been completed, the encoded data is stored and the compression cycle will be complete.

In order to decompress the compressed image, the inverse of all previous operations must be completed. This follows the order presented in part b in figure 3. Initially, this involves using the huff2mat() function also provided from Digital Image Processing Using MATLAB. Once each channel has been decoded, the information is transformed into a matrix and the inverse zig-zag operation is applied. Each channel is then multiplied back by the quantisation matrices for luminance and chrominance, depending on the relevant channel. Following de-quantization, the inverse DCT function is applied to each channel in order to convert the image back from the frequency domain. Finally, if the image being compressed was an RGB image, it must be converted back from YCbCr space to RGB before the image is output to the user.

Results

The compression ratio can be calculated by dividing the uncompressed image size by the size of the compressed image[7]. Given the implementation achieved, a compression ratio of 0.029 was found on the image 'eng_HI.png' when using a quality factor of 5. However, this does have a noticeable decrease in quality from the initial input image. Given that as much quality should be preserved as possible, analysis to find ideal quality factor must be performed.

Quality Factor	Compression Ratio	Size of Final Image	Image
5	0.028938	791KB	

[6] Gonzalez, R. C., Woods, R. E. & Eddins, S. L., Digital Image Processing Using MATLAB, Prentice-Hall, (2004)

[7] Broadcast Engineering, Pixel grids, bit rate and compression ratio (2007) [Online]. Available: https://web.archive.org/web/20131010224651/http://broadcastengineering.com/storage-amp-networking/pixel-grids-bit-rate-and-compression-ratio [Accessed 13th December 2018]

Quality Factor	Compression Ratio	Size of Final Image	Image
10	0.037344	1087KB	
15	0.043937	1229KB	
20	0.050126	1395KB	
30	0.061864	1595KB	
40	0.07089	1742KB	
50	0.07089	1742KB	

Quality Factor	Compression Ratio	Size of Final Image	Image
100	0.07089	1742KB	

Conclusion

Given the results of altering the quality factor, it can be seen that the ideal quality factor for this compression algorithm is a value of 15. With a quality factor set below 15, there is still noticeable image quality loss in the final image, which is most clearly present in the image with a quality factor of 5. Overall a successful lossy compression algorithm was able to be implemented which allows the user to determine a quality factor, which in turn affects the quality of the compressed image. In order to improve upon the algorithm further, it would be useful to create an original Huffman encoding and decoding scheme rather than implementing a function from the work Digital Image Processing Using MATLAB. Furthermore in order to better understand the ideal quality factor to use for the image, a second algorithm could have been created in order to find the mean square error of the output image compared to input image for varying values of the quality factor. This would have enabled the user to use the compression algorithm to get the highest quality image possible at the lowest size. This would however take significantly more time to compute compared to simply processing the algorithm once with a given quality factor.

Appendix

References

1 - Cabeen, K., Gent, P., Image Compression and the Discrete Cosine Transform (2008) [Online]. Available: https://www.math.cuhk.edu.hk/~lmlui/dct.pdf [Accessed 10 December 2018].

2 - MathWorks, Discrete Cosine Transform Matrix (2018) [Online]. Available: https://uk.mathworks.com/help/images/ref/dctmtx.html [Accessed 12 December 2018].

3 - MathWorks, Distinct Block Processing For Images (2018) [Online]. Available: https://uk.mathworks.com/help/images/ref/blockproc.html?s_tid=doc_ta [Accessed 12 December 2018].

4 - Roberts, E., Coefficient Quantization (2006) [Online]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossy/jpeg/coeff.htm [Accessed 13th December 2018]

5 - Girod, B., Image and Video Compression (2012) [Online]. Available: https://web.stanford.edu/class/ee398a/handouts/lectures/08-JPEG.pdf [Accessed 13th December 2018]

6 - Gonzalez, R. C., Woods, R. E. & Eddins, S. L., Digital Image Processing Using MATLAB, Prentice-Hall, (2004)

7 - Broadcast Engineering, Pixel grids, bit rate and compression ratio (2007) [Online]. Available: https://web.archive.org/web/20131010224651/http://broadcastengineering.com/storage-amp-networking/pixel-grids-bit-rate-and-compression-ratio [Accessed 13th December 2018]

Code

ImageCompressionCoursework.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%Image Compression/Decompression Algorithm%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear;
input_image = imread('eng_HI.png'); %Read in the image
dim1 = size(input_image,1); % image width
dim2 = size(input_image,2); % image height
dim3 = size(input_image,3); % number of channels
[xm, xn] = size(input_image);  % Get input size as array

% Up-sample chroma functions
upsample_filter_1d = [1 3 3 1] / 4;
upsample_filter = upsample_filter_1d' * upsample_filter_1d;

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

%Define Luminance Quantization matrix
qY = [  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];

%Define Chrominance Quantization matrix
qC = [  17 18 24 47 99 99 99 99;
    18 21 26 66 99 99 99 99;
    24 26 56 99 99 99 99 99;
    47 66 99 99 99 99 99 99;
    99 99 99 99 99 99 99 99;
    99 99 99 99 99 99 99 99;
    99 99 99 99 99 99 99 99;
    99 99 99 99 99 99 99 99];

T = dctmtx(8); % DCT matrix
scale = 255;

% Block processing functions
dct = @(block_struct) T * block_struct.data * T';
invdct = @(block_struct) T' * block_struct.data * T;
quantY = @(block_struct) round( block_struct.data./qY);
dequantY = @(block_struct) block_struct.data.*qY;
quantC = @(block_struct) round( block_struct.data./qC);
dequantC = @(block_struct) block_struct.data.*qC;
```

```
%Define the quality factor and apply to matrices
qf = 15;
if qf < 50
    q_scale = floor(5000/qf);
else
    q_scale = 200 - 2 * qf;
end
qY = round(qY * q_scale);
qC = round(qC * q_scale);


%Ensure that RGB and Grayscale images are considered differently
if (dim3 > 1)
    ycbcrmap = rgb2ycbcr(im2double(input_image)); % mapping, need double for DCT
    % Down-sample and decimate chroma
    cb = conv2(ycbcrmap(:, :, 2), [1 1; 1 1]) ./ 4.0;
    cr = conv2(ycbcrmap(:, :, 3), [1 1; 1 1]) ./ 4.0;
    cb = cb(2 : 2 : size(cb, 1), 2 : 2 : size(cb, 2));
    cr = cr(2 : 2 : size(cr, 1), 2 : 2 : size(cr, 2));

    %Upsample Chroma
    cb = conv2(upsample_filter, upsample(upsample(padarray(cb, [1 1],
'replicate'), 2)', 2)');
    cb = cb(4 : size(cb, 1) - 4, 4 : size(cb, 2) - 4);
    % Upsample Chroma
    cr = conv2(upsample_filter, upsample(upsample(padarray(cr, [1 1],
'replicate'), 2)', 2)');
    cr = cr(4 : size(cr, 1) - 4, 4 : size(cr, 2) - 4);
    y = ycbcrmap(:,:,1);
    ycbcrmap_new(:,:,2) = cb;
    ycbcrmap_new(:,:,3) = cr;
    ycbcrmap_new(:,:,1) = y;

else
    ycbcrmap_new = im2double(input_image); %Don't use rg2ycbcr on grayscale
images
end

%Define output image array of zeros
output_image = zeros(size(input_image), 'double');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%% Compression %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

disp('Starting compression cycle');

for ch=1:dim3
    channel = ycbcrmap_new(:,:,ch); % Get current channel

    if (ch == 1)
        disp('Starting Y channel compression');
    elseif (ch == 2)
        disp('Starting Cb channel compression');
    elseif (ch == 3)
```

```
        disp('Starting Cr channel compression');
    end
    % Apply DCT Function
    channel_dct = blockproc(channel, [8 8], dct, 'PadPartialBlocks',
true).*scale;

    % Apply quantization
    if (ch == 1)
        channel_qy = blockproc(channel_dct,[8 8], quantY);  % Quantization for
luminance
    elseif (ch == 2)
        channel_qcb = blockproc(channel_dct,[8 8], quantC);  % Quantization for
chrominance
    elseif (ch == 3)
        channel_qcr = blockproc(channel_dct,[8 8], quantC);  % Quantization for
chrominance
    end

    %Apply Zig-zag re-ordering of elements to Y Channel
    if (ch == 1)
        zig_y = im2col(channel_qy, [8 8], 'distinct');  % Break 8x8 blocks into
columns
        xb = size(zig_y, 2);                    % Get number of blocks
        zig_y = zig_y(order, :);                    % Reorder column elements
        eob = max(zig_y(:)) + 1;                % Create end-of-block symbol
        r = zeros(numel(zig_y) + size(zig_y, 2), 1);
        count = 0;
        for j = 1:xb                          % Process 1 block (col) at a time
            i = max(find(zig_y(:, j)));          % Find last non-zero element
            if isempty(i)                     % No nonzero block values
                i = 0;
            end

            p = count + 1;
            q = p + i;
            r(p:q) = [zig_y(1:i, j); eob];      % Truncate trailing 0's, add
EOB,
            count = count + i + 1;           % and add to output vector
        end

        r((count + 1):end) = [];            % Delete unusued portion of r
        yhuffman   = mat2huff(r);
        [ym, yn]    = size(channel_qy);

        % Zigzag CB Channel
    elseif (ch == 2)
        zig_cb = im2col(channel_qcb, [8 8], 'distinct');  % Break 8x8 blocks
into columns
        xb = size(zig_cb, 2);                    % Get number of blocks
        zig_cb = zig_cb(order, :);                    % Reorder column elements
        eob = max(zig_cb(:)) + 1;                % Create end-of-block symbol
        r = zeros(numel(zig_cb) + size(zig_cb, 2), 1);
        count = 0;
        for j = 1:xb                          % Process 1 block (col) at a time
            i = max(find(zig_cb(:, j)));          % Find last non-zero element
```

```
            if isempty(i)                    % No nonzero block values
                i = 0;
            end

            p = count + 1;
            q = p + i;
            r(p:q) = [zig_cb(1:i, j); eob];      % Truncate trailing 0's, add
EOB,
            count = count + i + 1;           % and add to output vector
        end

        r((count + 1):end) = [];            % Delete unusued portion of r
        cbhuffman   = mat2huff(r);
        [cbm, cbn]   = size(channel_qcb);

        %Apply zigzag to CR channel
    elseif (ch == 3)
        zig_cr = im2col(channel_qcb, [8 8], 'distinct');  % Break 8x8 blocks
into columns
        xb = size(zig_cr, 2);                    % Get number of blocks
        zig_cr = zig_cr(order, :);                    % Reorder column elements
        eob = max(zig_cr(:)) + 1;                % Create end-of-block symbol
        r = zeros(numel(zig_cr) + size(zig_cr, 2), 1);
        count = 0;
        for j = 1:xb                            % Process 1 block (col) at a time
            i = max(find(zig_cr(:, j)));          % Find last non-zero element
            if isempty(i)                     % No nonzero block values
                i = 0;
            end

            p = count + 1;
            q = p + i;
            r(p:q) = [zig_cr(1:i, j); eob];      % Truncate trailing 0's, add
EOB,
            count = count + i + 1;           % and add to output vector
        end

        r((count + 1):end) = [];            % Delete unusued portion of r
        crhuffman   = mat2huff(r);
        [crm, crn]   = size(channel_qcr);
    end
end
disp('Compression cycle complete');
if (dim3 > 1)
compressed_vector = cat(3, yhuffman, cbhuffman, crhuffman);
else
    compressed_vector = yhuffman;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%% Decompression %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
disp('Starting decompression cycle');

for ch=1:dim3
```

```
    rev = order;                            % Compute inverse ordering
    for k = 1:length(order)
        rev(k) = find(order == k);
    end

    if (ch == 1)
        x = huff2mat(yhuffman);                % Huffman decode.
        eob = max(x(:));                       % Get end-of-block symbol
        z = zeros(64, xb);   k = 1;            % Form block columns by copying
        for j = 1:xb                           % successive values from x into
            for i = 1:64                        % columns of z, while changing
                if x(k) == eob                   % to the next column whenever
                    k = k + 1;   break;           % an EOB symbol is found.
                else
                    z(i, j) = x(k);
                    k = k + 1;
                end
            end
        end
        z = z(rev, :);                                  % Restore order

        % DPCM on DC coefficients
        DCcoeff = cumsum(z(1,:));
        z(1,:) = DCcoeff;

        channel_qy_decoded = col2im(z, [8 8], [ym, yn], 'distinct');     % Form
matrix blocks

    elseif (ch == 2)
        x = huff2mat(cbhuffman);               % Huffman decode.
        eob = max(x(:));                       % Get end-of-block symbol
        z = zeros(64, xb);   k = 1;            % Form block columns by copying
        for j = 1:xb                           % successive values from x into
            for i = 1:64                        % columns of z, while changing
                if x(k) == eob                   % to the next column whenever
                    k = k + 1;   break;           % an EOB symbol is found.
                else
                    z(i, j) = x(k);
                    k = k + 1;
                end
            end
        end
        z = z(rev, :);                                  % Restore order

        % DPCM on DC coefficients
        DCcoeff = cumsum(z(1,:));
        z(1,:) = DCcoeff;

        channel_qcb_decoded = col2im(z, [8 8], [cbm, cbn], 'distinct');     %
Form matrix blocks

    elseif (ch == 3)
        x = huff2mat(crhuffman);               % Huffman decode.
        eob = max(x(:));                       % Get end-of-block symbol
```

```
        z = zeros(64, xb);    k = 1;            % Form block columns by copying
        for j = 1:xb                            % successive values from x into
            for i = 1:64                         % columns of z, while changing
                if x(k) == eob                    % to the next column whenever
                    k = k + 1;    break;           % an EOB symbol is found.
                else
                    z(i, j) = x(k);
                    k = k + 1;
                end
            end
        end
        z = z(rev, :);                                  % Restore order

        % DPCM on DC coefficients
        DCcoeff = cumsum(z(1,:));
        z(1,:) = DCcoeff;

        channel_qcr_decoded = col2im(z, [8 8], [crm, crn], 'distinct');      %
Form matrix blocks

    end

    % Apply dequantization depending on channel
    if (ch == 1)
        channel_new(:,:,1) = blockproc(channel_qy,[8 8], dequantY);
    elseif (ch == 2)
        channel_new(:,:,2) = blockproc(channel_qcb,[8 8], dequantC);
    elseif (ch == 3)
        channel_new(:,:,3) = blockproc(channel_qcr,[8 8], dequantC);
    end

    % Apply Inverse DCT Function depending on channel
    %output_data = blockproc(channel_new, [n n], invdct)./scale;
    if (ch == 1)
        output_data = blockproc(channel_new(:,:,1), [8 8], invdct) ./scale;
    elseif (ch == 2)
        output_data = blockproc(channel_new(:,:,2), [8 8], invdct) ./scale;
    elseif (ch == 3)
        output_data = blockproc(channel_new(:,:,3), [8 8], invdct) ./scale;
    end

    if (ch == 1)
        disp('Y channel decompression complete');
        % Set output image
        output_image(:,:,1) = output_data(1:dim1,1:dim2);
    elseif (ch == 2)
        disp('Cb channel decompression complete');
        % Set output image
        output_image(:,:,2) = output_data(1:dim1,1:dim2);
    elseif (ch == 3)
        disp('Cr channel decompression complete');
        % Set output image
        output_image(:,:,3) = output_data(1:dim1,1:dim2);
    end
end
```

```
% Compression ratio
% total number of bits in final file, divided by number of bits in original file
% Size of vectors
if (dim3 > 1)
all_vector_size = [yhuffman.size, crhuffman.size, cbhuffman.size];
num_size_y = all_vector_size(1:1);
num_size_cb = all_vector_size(1,3);
num_size_cr = all_vector_size(1,5);
sized_up = num_size_y + num_size_cb + num_size_cr;
sized_up_double = double(sized_up);
else
    all_vector_size = [yhuffman.size];
    num_size_y = all_vector_size(1:1);
    sized_up_double = double(num_size_y);
end

initial_size=dim1*dim2*dim3;

compRatio = sized_up_double/initial_size; %Compute ratio

%Output compression ratio
ratio_disp = ['Compression ratio: ', num2str(compRatio)];
input_disp = ['Input image size (bytes): ', num2str(initial_size)];
output_disp = ['Compressed image size (bytes): ', num2str(sized_up_double)];

disp('Decompression cycle complete');
disp('All cycles complete');
disp(input_disp);
disp(output_disp);
disp(ratio_disp);

if (dim3 > 1)
    output_image = im2uint8(ycbcr2rgb(output_image)); % Convert back to rgb
uint8
else
    output_image = im2uint8(output_image); % back to rgb uint8
end

subplot(1,2,1), imshow(input_image) % show initial image
subplot(1,2,2), imshow(output_image) % show resulting image
imwrite(output_image, 'compressed_image.jpeg');
```

The following code used was taken from Digital Image Processing Using Matlab by R. C. Gonzalez, R. E. Woods & S. L. Eddins (2004).

Huffman.m

```
function CODE = huffman(p)
%HUFFMAN Builds a variable-length Huffman code for a symbol source.
%   CODE = HUFFMAN(P) returns a Huffman code as binary strings in
%   cell array CODE for input symbol probability vector P. Each word
%   in CODE corresponds to a symbol whose probability is at the
%   corresponding index of P.
%
```

```
%   Based on huffman5 by Sean Danaher, University of Northumbria,
%   Newcastle UK. Available at the MATLAB Central File Exchange:
%   Category General DSP in Signal Processing and Communications.

%   Copyright 2002-2004 R. C. Gonzalez, R. E. Woods, & S. L. Eddins
%   Digital Image Processing Using MATLAB, Prentice-Hall, 2004
%   $Revision: 1.5 $  $Date: 2003/10/26 18:37:16 $

% Check the input arguments for reasonableness.
error(nargchk(1, 1, nargin));
if (ndims(p) ~= 2) | (min(size(p)) > 1) | ~isreal(p) | ~isnumeric(p)
   error('P must be a real numeric vector.');
end

% Global variable surviving all recursions of function 'makecode'
global CODE
CODE = cell(length(p), 1);  % Init the global cell array

if length(p) > 1            % When more than one symbol ...
   p = p / sum(p);          % Normalize the input probabilities
   s = reduce(p);           % Do Huffman source symbol reductions
   makecode(s, []);         % Recursively generate the code
else
   CODE = {'1'};            % Else, trivial one symbol case!
end;

%---------------------------------------------------------------------%
function s = reduce(p);
% Create a Huffman source reduction tree in a MATLAB cell structure
% by performing source symbol reductions until there are only two
% reduced symbols remaining

s = cell(length(p), 1);

% Generate a starting tree with symbol nodes 1, 2, 3, ... to
% reference the symbol probabilities.
for i = 1:length(p)
   s{i} = i;
end

while numel(s) > 2
   [p, i] = sort(p);     % Sort the symbol probabilities
   p(2) = p(1) + p(2);   % Merge the 2 lowest probabilities
   p(1) = [];            % and prune the lowest one

   s = s(i);             % Reorder tree for new probabilities
   s{2} = {s{1}, s{2}}; % and merge & prune its nodes
   s(1) = [];            % to match the probabilities
end

%---------------------------------------------------------------------%
function makecode(sc, codeword)
% Scan the nodes of a Huffman source reduction tree recursively to
% generate the indicated variable length code words.
```

```
% Global variable surviving all recursive calls
global CODE

if isa(sc, 'cell')                    % For cell array nodes,
   makecode(sc{1}, [codeword 0]);     % add a 0 if the 1st element
   makecode(sc{2}, [codeword 1]);     % or a 1 if the 2nd
else                                  % For leaf (numeric) nodes,
   CODE{sc} = char('0' + codeword);   % create a char code string
end
```

mat2huff.m

```
function y = mat2huff(x)
%MAT2HUFF Huffman encodes a matrix.
%   Y = MAT2HUFF(X) Huffman encodes matrix X using symbol
%   probabilities in unit-width histogram bins between X's minimum
%   and maximum values. The encoded data is returned as a structure
%   Y:
%      Y.code   The Huffman-encoded values of X, stored in
%               a uint16 vector.  The other fields of Y contain
%               additional decoding information, including:
%      Y.min    The minimum value of X plus 32768
%      Y.size   The size of X
%      Y.hist   The histogram of X
%
%   If X is logical, uint8, uint16, uint32, int8, int16, or double,
%   with integer values, it can be input directly to MAT2HUFF. The
%   minimum value of X must be representable as an int16.
%
%   If X is double with non-integer values---for example, an image
%   with values between 0 and 1---first scale X to an appropriate
%   integer range before the call. For example, use Y =
%   MAT2HUFF(255*X) for 256 gray level encoding.
%
%   NOTE: The number of Huffman code words is round(max(X(:))) -
%   round(min(X(:))) + 1.  You may need to scale input X to generate
%   codes of reasonable length.  The maximum row or column dimension
%   of X is 65535.
%
%   See also HUFF2MAT.

%   Copyright 2002-2004 R. C. Gonzalez, R. E. Woods, & S. L. Eddins
%   Digital Image Processing Using MATLAB, Prentice-Hall, 2004
%   $Revision: 1.5 $  $Date: 2003/11/21 15:21:12 $

if ndims(x) ~= 2 | ~isreal(x) | (~isnumeric(x) & ~islogical(x))
   error('X must be a 2-D real numeric or logical matrix.');
end

% Store the size of input x.
y.size = uint32(size(x));

% Find the range of x values and store its minimum value biased
% by +32768 as a UINT16.
x = round(double(x));
```

```
xmin = min(x(:));
xmax = max(x(:));
pmin = double(int16(xmin));
pmin = uint16(pmin + 32768);    y.min = pmin;

% Compute the input histogram between xmin and xmax with unit
% width bins, scale to UINT16, and store.
x = x(:)';
h = histc(x, xmin:xmax);
if max(h) > 65535
    h = 65535 * h / max(h);
end
h = uint16(h);   y.hist = h;

% Code the input matrix and store the result.
map = huffman(double(h));           % Make Huffman code map
hx = map(x(:) - xmin + 1);          % Map image
hx = char(hx)';                     % Convert to char array
hx = hx(:)';
hx(hx == ' ') = [];                 % Remove blanks
ysize = ceil(length(hx) / 16);      % Compute encoded size
hx16 = repmat('0', 1, ysize * 16); % Pre-allocate modulo-16 vector
hx16(1:length(hx)) = hx;            % Make hx modulo-16 in length
hx16 = reshape(hx16, 16, ysize);    % Reshape to 16-character words
hx16 = hx16' - '0';                 % Convert binary string to decimal
twos = pow2(15:-1:0);
y.code = uint16(sum(hx16 .* twos(ones(ysize, 1), :), 2))';
```

huff2mat.m

```
function x = huff2mat(y)
%HUFF2MAT Decodes a Huffman encoded matrix.
%   X = HUFF2MAT(Y) decodes a Huffman encoded structure Y with uint16
%   fields:
%      Y.min    Minimum value of X plus 32768
%      Y.size   Size of X
%      Y.hist   Histogram of X
%      Y.code   Huffman code
%
%   The output X is of class double.
%
%   See also MAT2HUFF.

%   Copyright 2002-2004 R. C. Gonzalez, R. E. Woods, & S. L. Eddins
%   Digital Image Processing Using MATLAB, Prentice-Hall, 2004
%   $Revision: 1.5 $  $Date: 2003/11/21 13:17:50 $

if ~isstruct(y) | ~isfield(y, 'min') | ~isfield(y, 'size') | ...
       ~isfield(y, 'hist') | ~isfield(y, 'code')
   error('The input must be a structure as returned by MAT2HUFF.');
end

sz = double(y.size);
m = sz(1);
n = sz(2);
```

```
xmin = double(y.min) - 32768;            % Get X minimum
map = huffman(double(y.hist));           % Get Huffman code (cell)

% Create a binary search table for the Huffman decoding process.
% 'code' contains source symbol strings corresponding to 'link'
% nodes, while 'link' contains the addresses (+) to node pairs for
% node symbol strings plus '0' and '1' or addresses (-) to decoded
% Huffman codewords in 'map'. Array 'left' is a list of nodes yet to
% be processed for 'link' entries.

code = cellstr(char('', '0', '1'));      % Set starting conditions as
link = [2; 0; 0];   left = [2 3];        % 3 nodes w/2 unprocessed
found = 0;   tofind = length(map);       % Tracking variables

while length(left) & (found < tofind)
   look = find(strcmp(map, code{left(1)}));    % Is string in map?
   if look                               % Yes
      link(left(1)) = -look;             % Point to Huffman map
      left = left(2:end);                % Delete current node
      found = found + 1;                 % Increment codes found

   else                                  % No, add 2 nodes & pointers
      len = length(code);                % Put pointers in node
      link(left(1)) = len + 1;

      link = [link; 0; 0];               % Add unprocessed nodes
      code{end + 1} = strcat(code{left(1)}, '0');
      code{end + 1} = strcat(code{left(1)}, '1');

      left = left(2:end);                % Remove processed node
      left = [left len + 1 len + 2];     % Add 2 unprocessed nodes
   end
end

x = unravel(y.code', link, m * n);       % Decode using C 'unravel'
x = x + xmin - 1;                        % X minimum offset adjust
x = reshape(x, m, n);                    % Make vector an array
```

unravel.c – This code must be first compiled in matlab using Mex and a relevant C/C++ compiler to create a .mex* file.

```
#include "mex.h"
void unravel(uint16_T *hx, double *link, double *x, double xsz, int hxsz)
{
    int i = 15, j = 0, k = 0;
    int n = 0;
    while(xsz - k)
    {
        if (*(link + n)>0)
        {
            if ((*(hx + j)>>i) & 0x0001)
            {
                n = *(link + n);
            }
            else n = *(link + n) - 1;
            if (i) i--; else {j++;i = 15;}
```

```
            if (j > hxsz)
                mexErrMsgTxt("out of code bits ???");
        }
        else
        {
            *(x + k++) = -*(link + n);
            n = 0;
        }
    }
    if (k == xsz - 1)
        \
            *(x + k++) = -*(link + n);
}
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    double *link, *x, xsz;
    uint16_T *hx;
    int hxsz;
    /*check for reasonableness*/
    if(nrhs != 3)
        mexErrMsgTxt("3 inputs required.");
    else if (nlhs > 1)
        mexErrMsgTxt("Too many output arguments");
    /*is the last input a scalar*/
        if (!mxIsDouble(prhs[2]) || mxIsComplex(prhs[2]) || mxGetN(prhs[2]) *
mxGetM(prhs[2]) != 1)
    {
        mexErrMsgTxt("input XSIZE must be a scalar.");
    }
    hx   = (uint16_T *) mxGetData(prhs[0]);
    link = (double *) mxGetData(prhs[1]);
    xsz = mxGetScalar(prhs[2]);     /* returns DOUBLE */
    /* Get the number of elemnts in hx */
    hxsz = mxGetM(prhs[0]);
    /* Create 'xsz' x 1 output matrix */
    plhs[0] = mxCreateDoubleMatrix(xsz, 1, mxREAL);
    /* Get C pointer to a copy of the output matrix */
    x = (double *) mxGetData(plhs[0]);
    /* Call the C subroutine */
    unravel(hx, link, x, xsz, hxsz);
}
```