An introduction to JPEG compression using MATLAB

Arno Swart

30 October, 2003

1 Introduction

This document describes the popular JPEG still image coding format. The aim is to compress images while maintaining acceptable image quality. This is achieved by dividing the image in blocks of 8×8 pixels and applying a discrete cosine transform (DCT) on the partitioned image. The resulting coefficients are quantised, less significant coefficients are set to zero. After quantisation two encoding steps are made, zero run length encoding (RLE) followed by an entropy coding. Part of the JPEG encoding is *lossy* (information may get lost. Therefore, the reconstruction may not be exact), part is *lossless* (no loss of information).

JPEG allows for some flexibility in the different stages, not every option is explored. Also the focus is on the DCT and quantisation. The entropy coding will only be briefly discussed and the file structure definitions will not be considered.

There is a MATLAB implementation of (a subset of) JPEG that can be used alongside this document. It can be down-loaded from the course's internet pages, follow the link to Wavelet & FFT resources. Try understanding the different functions in the source code while reading the document. Some questions will ask you to modify the code slightly.

2 Tiling and the DCT

To simplify matters we will assume that the width and height of the image are a multiple of 8 and that the image is either gray scale or RGB (red, green, blue). Each pixel is assigned a value between 0 and 255, or a triplet of RGB values.

In the case of a color RGB picture a pointwise transform is made to the YUV (luminance, blue chrominance, red chrominance) color space. This space is in some sense more decorrelated than the RGB space and will allow for better

quantisation later. The transform is given by

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.1687 & -0.3313 & 0.5 \\ 0.5 & -0.4187 & -0.0813 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} 0 \\ 0.5 \\ 0.5 \end{pmatrix}, \tag{1}$$

and the inverse transform is

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1.402 \\ 1 & -0.34414 & -0.71414 \\ 1 & 1.772 & 0 \end{pmatrix} \begin{pmatrix} Y \\ U - 0.5 \\ V - 0.5 \end{pmatrix}. \tag{2}$$

The next step is dividing each component of the image in blocks of 8×8 pixels. This is needed since the DCT has no locality information whatsoever. There is *only* frequency information available. We continue by shifting every block by -128, this makes the values symmetric around zero with average close to zero.

The DCT of a signal $f(x, y)$ and the inverse DCT (IDCT) of coefficients $F(u, v)$ are given by

$$\begin{aligned} F(u,v) &= \tfrac{1}{4} C(u)C(v) \textstyle\sum_{x=0}^{7} \sum_{y=0}^{7} f(x,y) \cos(\frac{(2x+1)u\pi}{16}) \cos(\frac{(2y+1)v\pi}{16}), \\ f(x,y) &= \tfrac{1}{4} \textstyle\sum_{u=0}^{7} \sum_{v=0}^{7} C(u)C(v)F(u,v) \cos(\frac{(2x+1)u\pi}{16}) \cos(\frac{(2y+1)v\pi}{16}), \end{aligned} \tag{3}$$

where

$$C(z) = \begin{cases} 1/\sqrt{2} & \text{if } z = 0, \\ 1 & \text{otherwise.} \end{cases} \tag{4}$$

Question 0 *What is the relation between YUV and f?*

Question 1 *The DCT is 'half' of the Fourier transform. Why not use the full Fourier representation instead of the cosine basis functions? It is because the DCT can approximate linear functions with less coefficients. To see this take the array* $8, 16, 24, 32, 40, 48, 56, 64$. *Do a forward Fourier transform, set the last 4 bits to zero (this is what will happen in the quantisation step explained later) and do a inverse Fourier transform. Perform the same procedure for the DCT. What do you notice? Can you explain the results?*

Question 2 *The DCT is given as a two dimensional transform. For implementation purposes it is more efficient to write it as a sequence of two one-dimensional transforms. Derive and implement this.*

Question 3 *The DCT only uses 64 basis functions. One can precompute these and reuse them in the algorithm at the expense of a little extra memory. A function that returns the basis functions is provided with the source. Build this in to the main code.*

The blocks are processed row by row, from left to right.

3 Quantisation and Zigzag scan

So far there has been no compression. We will now proceed by introducing zeros in the arrays by means of quantisation. Every coefficient c_{ij} is divided by an integer q_{ij} and rounded to the nearest integer. The importance of the coefficients is dependent on the human visual system, the eye is much more sensitive to low frequencies. Measurements led to standard quantisation tables, listed below. Note that the tables for luminance and chrominance are different (for gray scale images only the luminance tables are used). The tables are [1]

$$Q_{LUM} = \frac{1}{s}\begin{pmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{pmatrix} \tag{5}$$

$$Q_{CHR} = \frac{1}{s}\begin{pmatrix} 17 & 18 & 24 & 47 & 99 & 99 & 99 & 99 \\ 18 & 21 & 26 & 66 & 99 & 99 & 99 & 99 \\ 24 & 26 & 56 & 99 & 99 & 99 & 99 & 99 \\ 47 & 66 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \end{pmatrix} \tag{6}$$

(In the actual computations, q_{ij} is an integer which value is obtained by rounding to the nearest integer and taking the maximum with 1. Why maximum with 1?) The variable s in the above formulas stands for scale. The higher s, the fewer coefficients will be set to zero. This is also known as the *quality level*.

Question 4 *Another more primitive quantisation technique is thresholding. Every coefficient is divided by the same constant value (all q_{ij} are equal). Implement this, can you find a noticeable difference?*

The next ingredient is the so-called *zigzag scan*. Every block is converted to a vector by gathering elements in zigzag fashion. See figure (3) for the zigzag scheme.

This method has the advantage that low-frequency coeffients are placed first, while high frequency components come last in the array. This is likely to introduce long substrings of zeros at the end of the array.

Question 5 *What does 'low-frequency' and 'high-frequency' look like in an image. Can you see the logic in discarding (quantising to zero) the high-frequency components?*

[1] The JPEG standard also allows for inclusion of custom tables in the file header

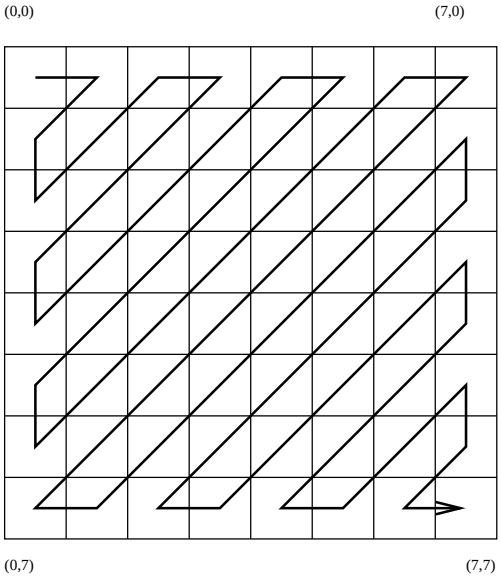

Figure 1: The zigzag scan order.

Question 6 *In what sense is the YUV color space more decorrolated than the RGB space (cf. the 2nd paragrapfh of Section 2).*

The first element of the resulting array is called the DC component. The other 63 entries are the AC components. They are treated separately in the entropy coding process.

Question 7 *What is the interpretation of the DC coefficient? (In the signal processing community DC stands for 'direct current' while 'AC' stands for 'alternating current'.)*

4 RLE and entropy coding

This section describes the final step in the compression. We will start with the DC component, which is simplest. We expect the DC components of a cluster of neighbouring blocks to differ only slightly from each other. For this reason we use *differential coding*, that is, we visit the DC coefficients of the blocks in row-by-row, left to right fashion and record the difference with the previous DC's (except for the first which remains unchanged). Then, on the array of differences, we proceed with Huffman coding; for a brief description of Huffman coding, see the next section. Huffman coding is an entropy coder.

For the remaining 63 AC coefficients we use zero RLE. This means we first store the number of zeros preceding a nonzero and then the nonzero. For example the values 0250001 become the pairs $(1, 2), (0, 5), (3, 1)$. The value $(0, 0)$ is

a special value signaling that the remaining coefficients are all zero. Since our zigzag scheme clusters the zeros at the end of the array we probably have good compression here. Now we come at a point where there is a difference between the JPEG standard and our implementation. We use 8 bits per value in the pair and do Huffman coding on the bit stream. The official JPEG standard works differently, but will probably not have significantly more compression.

The standard proposes to store no number greater than 15 in the first component of the pair. For example 17 zeros and a one would become $(15, 0), (2, 1)$. Now only four bits (a *nibble*) are needed for this number. For the second number we store the number of bits that the coefficient needs. This also needs a maximum of 4 bits and we put all bits in a bit stream and apply Huffman coding. After decoding we know the number of bits per number so we can reconstruct our zero length and bit length values. For the actual coefficients we also make one bit stream, only taking the number of bits we need. This stream is also Huffman compressed. When decoding we know this number of bits, it came from our first stream, so we can also retrieve these coefficients.

Question 8 *What part of the JPEG encoding is lossy and what part is lossless?*

How much compression is achieved with the lossless part of JPEG? When (for what value of s) do we have lossless compression?

5 Huffman coding (optional)

Huffman coding is the optimal lossless scheme for compressing a bit stream. It works by fixing an n and dividing the input bit-stream in sequences of length n. For simplicity of explanation, assign symbols, say $A, B, C, \_, \ldots$, to all sequences of zeros and ones (bit-codes) of length n. Now, a bit-stream that we want to encode might look like $ABAAB\_AACB$. Next, the Huffman encoder calculates how often each symbol occurs. Then, the symbols are assigned new bit-codes, but now not necessarily of length n. The more often a symbol occurs (as the A in our example), the smaller the length of the new bit-code will be. The output of the Huffman encoder is a bit stream formed by the new bit-coded symbols. For decoding, we need to know where the bit-code for one symbol stops and a new one begins. This is solved by enforcing the *unique prefix* condition: the bit-code of no symbol is the prefix of the bit-code of another symbol. For instance, the four symbols A, B, C and $\_$ might be coded as in Table 1: A might be coded as 0 in the output bit-stream, while it might represent 00 if $n = 2$ (assuming there are only four symbols A, B, C, and $\_$) in the input bit-steam. The number of bits to represent the string $ABAAB\_AACB$ of 10 symbols reduces from $10 \cdot 2 = 20$ bits to $5 \cdot 1 + 3 \cdot 2 + 1 \cdot 3 + 1 \cdot 3 = 17$ bits. Note that the reduction would be better if A would occur more often than in the example. JPEG allows for coding according to the number of occurences or according to predefined tables included in the header of the jpg-picture. There are standard tables compiled from counting the number of occurences in a large number of images.

original code	symbol	ouput code
00	A	0
01	B	11
10	C	100
11	$\llcorner$	101

original bit-stream:
0001000011100001001
symbolized: $ABAAB\llcorner AACB$
huffman coded bit-stream:
01100111010010011

Table 1: If the original input bit-stream is diveded in sequences of length 2 then the table gives the coding for the four symbols as obtained with the Huffman coding for, for instance, the string $ABAAB\llcorner AACB$. Note that reading the huffman coded bit-stream from left to right leads to exact reconstriction of the original bit-stream.

6 Experiments

Try out for yourself how well JPEG compresses images. The source code comes with a few images in MATLAB `.mat` *format. Try to make graphs of the scaling factor versus the number of bits per pixel. Also try to make visual estimations of the quality. A more quantitative measure would be the mean squared error of the reconstructed image with respect to the original. Later on you can compare the results with the performance of the more advanced JPEG-2000 format (it uses wavelets for compression).*