USING *DERIVE* TO EXPLORE THE MATHEMATICS OF THE JPEG IMAGE COMPRESSION ALGORITHM

Mark Michael, Ph.D., Associate Professor of Computer Science
King's College (Pennsylvania), 133 N. River St., Wilkes-Barre, PA 18711 USA
mmichael@acm.org

Many of the photographic images displayed on Web pages and saved on CDs have been compressed via an algorithm developed by the Joint Photographic Experts Group (JPEG). The algorithm makes use of several mathematical techniques (some of which exploit properties of human vision perception), but the heart of algorithm is the Discrete Cosine Transform (DCT), a variant of the classic Fourier Transform. This transform is more sophisticated than most compression techniques. Derive can be used to examine in detail the intermediate values that result during the algorithm's most crucial steps. By experimentation, one can gain insight into how the DCT organizes data according to its importance and how the following quantization step can be used to selectively prune the less important data.

Introduction

It is well-known that most of the still, photographic images we view on Web pages or save from digital cameras are files saved with an extension of .jpg. It is common to refer to these images as being saved in "JPEG format." It is not widely known that this is a misnomer. The Joint Photographic Experts Group did define an image compression algorithm — the JPEG algorithm — but did not specify an actual format for storing images. What is usually referred to as the "JPEG format" is actually the *JFIF format (JPEG File Interchange Format)*, defined by C-Cube Microsystems. Also supporting the JPEG compression is the TIFF format (Tag Image File Format) developed by a group of companies including Aldus and Microsoft.

Though we will follow common practice in doing so, it is also inaccurate to speak of "the JPEG algorithm." There is actually a toolkit of different algorithms. When employing "the" algorithm, one can invoke alternative modes of operation, skip optional steps, and specify a multitude of parameters.

Although the newest version of the JPEG algorithm dates to 2000, we explore the well-established version codified by the ISO (International Standards Organization) as ISO DIS 10918-1. (See [4].) We will make passing references to the variety of JPEG compression options available, but our focus will be on the JPEG *baseline process*, the minimal set of features common to all JPEG-aware applications.

Even this basic form of the algorithm provides numerous options, each of which has the potential to influence the compression ratio and the quality of the compressed image, that is to say, the image as it is perceived when eventually decompressed. Obviously, the smaller we make the file, the more quickly it can be transmitted over the Internet. There is no free lunch, of course: the more we compress, the more image information we lose. How much we choose to lose is influenced by how the image will be used.

Even the highest quality compressions result in remarkable savings in space (and, therefore, transmission time). Images with good quality generally enjoy a compression ratio of 20:1 or better! The JPEG algorithm achieves this vast improvement over simpler compression methods by heavily exploiting the fact that certain kinds of image imperfections are imperceptible to humans. Even at much higher compression ratios, images may still be understandable, though noticeably flawed. (See Figure 1(b), later.)

Colour Spaces and Human Visual Perception in a Nutshell

Before getting into the specifics of the JPEG algorithm, it is necessary to first survey the particular way in which humans *perceive* images. This has a tremendous influence on which types of image information can be most forgivingly sacrificed throughout the algorithm.

In one sense, *colour* is a specific wavelength in the visible range of the electromagnetic spectrum. In reality, we are surrounded by combinations of different wavelengths. The ultimate reality — and what we will refer to as a colour — is what we perceive. Unlike some members of the animal kingdom, our eyes have three different types of colour receptors. The *tri-stimulus response* in the brain could be identical for two different distributions of wavelengths striking the colour receptors.

It should come as no surprise, then, that three parameters are needed to specify a particular perceived colour. This was known by the time the Commission International de l'Éclairage (CIE) released its famous chromaticity diagram in 1931.

Nowadays, the most well-known set of parameters is RGB: the red, green, and blue components that correspond to tiny dots (essentially subpixels) on a display screen. Mostly commonly, the range 0-255 is used for the values of each RGB component. For example, the colour yellow can be specified as a mixture of full-blast red, full-blast green, and no blue by the hexadecimal notation `#FFFF00` in HTML code for Web pages. Within computer graphics applications, the range 0.0-1.0 allows for more elegant mathematical operations, say, for transparency. Other ranges have been used as well. Whatever, the scale of measurement, the RGB colour space can be visualized as a three-dimensional cube.

Another colour space seen in many popular image manipulation programs is HSL: hue, saturation, and luminance. One version has the following ranges: hue is a circular scale (red corresponds to 0=240, green to 80, blue to 160); saturation has a maximum of 240 for "pure" colours and a minimum of 0 for a grey tone; and luminance ranges from 0 for black to 240 for white. To see how differently colours are organized in this scheme, imagine the HSL colour space as a football: white and black are opposite tips, the pure colours form the equator, and the axis of symmetry is the grey tones. There are numerous other colour spaces using hue, but "brightness," "value," or "intensity" in place of luminance. We will not discuss the technical differences between luminance and these terms.

With "colours" like ultra-violet and infra-red being outside the range of frequencies humans can see, it should be expected that human receptivity drops off at the red and blue ends of the visible spectrum. In fact, human receptivity peaks at about 555 nm, a yellowish green. Of the three RGB components, green is the one that is most important. In fact, an electronic camera may have 50% green receptors and 25% of

the other two types. (A single ray of light can pass through and effect several layers in photographic film, but cannot strike two electronic receptors.)

Furthermore, humans are far more sensitive to changes in luminance than *chrominance* (colour information). With a moment's reflection, we realize that a black-and-white photograph contains no colour information, but the vast majority of what contributes to our understanding of the scene portrayed. This brings us to the YUV family of colour spaces. The Y represents luminance, but in a different sense than the L in the HSL colour space. We use the UV here to represents two (generic) chrominance coordinates or *chroma*. There are varying definitions of Y, and even for one such definition there may be different specific associated chroma.

Y can always be viewed as a function of the RGB components. The function used for the JPEG algorithm is the weighed average:

```
COMPUTE_Y(rgb) := (77·rgb↓1 + 150·rgb↓2 + 29·rgb↓3) / 256,
```

where `rgb` is a vector with the three obvious components in the usual order.

Observe that B makes the smallest contribution to this average. The relative importance of R, G, and B can be visually confirmed by generating swatches of "neighbouring" colours in the RGB colour space, changing only one component at a time; to achieve colours which are equally spaced in a perceptual sense, one needs $\Delta B > \Delta R > \Delta G$. By comparison, the old scheme by which Windows' Paint program picked a default palette of 256 colours for bitmaps used only 2 bits for B and 3 bits for each of G and R.

Two parameters are still needed to convey the chrominance information (even though, ironically, the combined information of the chroma is less important than the luminance). One specific pair of chroma appropriate for digital video are Cb and Cr (a.k.a. Y–B and Y–R, respectively, where B and R are RGB components). The *Derive* formulas for these chroma are:

```
COMPUTE_CB(rgb) := 128 + (-44·rgb↓1 - 87·rgb↓2 + 131·rgb↓3) / 256 and
COMPUTE_CR(rgb) := 128 + (131·rgb↓1 - 110·rgb↓2 - 21·rgb↓3) / 256.
```

Although the JPEG method allows any colour space to be used, RGB and HSL do not permit one to take full advantage of JPEG compression. There are two reasons for this. A YUV colour space allows separate downsampling of the chroma (see step 2 of the algorithm, below) and separate quantization of the chroma (see step 4 of the algorithm, below). This means we can make greater reductions in "redundant" chrominance information while preserving more of the crucial luminance information.

Data Compression in a Nutshell

To appreciate why the Joint Photographic Experts Group took the path it did, we need to take an additional detour to contrast JPEG compression with other methods of compressing data, which are typically based on simpler ideas. Three major categories of compression are run-length encoding (RLE), statistical methods, and dictionary methods.

RLE encoding simply indicates a sequence of repeated characters by signifying the number of repetitions followed by the character to be repeated. This method can be counter-productive if unicharacter sequences are short because of the overhead of storing the length of each sequence. Windows bitmaps support this type of compression, but switch back to normal encoding for short runs.

Statistical methods rely on a knowledge of the frequencies of different characters being sent; these frequencies may be computed for the current data being compressed or may be predicted from past experience. A common approach, used, for instance, in the well-known Huffman coding, is to employ variable-size rather than fixed-size codes. This means that the more frequently occurring characters are represented by shorter codes while longer codes represent the less popular characters. If done correctly, the net result is a smaller file than if each character is encoded by the same number of bits. However, if there are no significant differences among the frequencies of individual characters, statistical methods cannot decrease the file size.

First introduced by Jacob Ziv and Abraham Lempel, dictionary methods exploit repetitions in subsequences of characters; they are a natural fit with strings of natural, alphabetic languages, but work in broader settings. The frequencies of sequences of characters vary more than the frequencies of individual characters. But dictionary methods are not designed around the frequencies, though some "LZ" methods incorporate RLE or statistical methods. Rather, they store and reference sequences already seen.

To understand why these three categories of compression fall short for the type of image of concern here, it helps to examine an "original" photograph before any kind of compression has been applied. By this we mean something akin to a scan that has been saved directly as a standard TIFF or Windows bitmap. If a seemingly uniform blue sky is magnified by at least 400%, we can see that there is "blue noise." Therefore there are no long sequences of repeated pixels that can be exploited by RLE. There are so many different colours — hundreds or thousands — with similar probabilities that statistical methods make little improvement. There are few if any repeated sequences of pixels, so dictionary methods have nothing with which to work. The same is true for photographs with detailed features.

It is important to emphasize that the JPEG method is best suited to images in which there are gradual changes in colour as we move from one pixel to a neighbouring one. The JPEG algorithm has trouble handling large areas of a perfectly uniform colour. Because of the segmentation of the image into small, independently processed blocks (explained later), "blocking artifacts" may become evident at the boundaries between blocks. This is more and more the case as quality is decreased in favour of increased compression ratio. (See Figure 1, below.) CompuServe's GIF format and the open-source PNG format are better at compressing images with a small number of very distinct colours and sharp transitions between regions of different colours. For images with particular characteristics, there are vastly different approaches to compression that offer their own benefits. These include quad-trees, iterated function systems, and wavelets. See [5] for outlines of these and many more.

Goals of the JPEG Algorithm

We assume that an image is "originally" represented in a form that possesses separate information about each individual pixel; we will refer to this as the *original image*. (From what has been said about the

receptors in a digital camera, 2/3 of the RGB information for a specific pixel is missing; we take as "original" the image which has complete, albeit extrapolated, RGB data for each pixel.) In other words, we conceptualize the original representation as a collection of independent representations of pixels: the image contains within it data which changes if and only if one specific pixel changes; conversely, a change in one pixel changes only that data. The compression algorithm encodes the original image as a second representation, which can be considered a way of storing (and perhaps transmitting) an image while it is not being viewed. For the image to be of any use, there must be a third representation, typically on a computer screen. At that point, the screen is once again an assemblage of independent pixels. Thus, converting the second representation into the third involves a decoding algorithm that undoes (to the extent possible) what the encoding algorithm did. Ultimately, there is yet another representation in the human brain.

In judging an image compression algorithm, it is evident that the following should be considered:

1. How much has the image been compressed (compared to its original representation)?
2. How good is the decompressed image . . .
    a. on the screen? (This can be measured objectively in various ways.)
    b. in the judgment of a human? (Gauging human perception is more subjective.)
3. How fast are implementations of the compression and (perhaps more importantly) decompression algorithms?

There is another consideration as well — flexibility. In fact, the JPEG algorithm is really a (parameterized) family of algorithms. In designing the JPEG suite, the goals of the Joint Photographic Experts Group were:

1. high compression ratios;
2. good results with continuous-tone images;
3. a method simple enough for efficient coding and decoding on many platforms;
4. multiple modes; and
5. numerous parameters [5].

The last is important because of the inherent trade-off between the first two; developers can make precise decisions as to *how* they are willing to sacrifice quality for the sake of file size. The different modes will not be detailed here. Those of us who remember very slow-speed Internet connections, may have seen in action the *progressive* mode, which allows a blocky, low-resolution image to be displayed while a browser is downloading the full, high-resolution image.

What the JPEG algorithm does not specify is things such as colour space, pixel aspect ratio, and row-storage formats.

The JPEG Algorithm Step 1

The first step in the algorithm is not executed for greyscale images as it would make no sense. For colour images, the three components of the image (in whatever colour space) are handled separately. In light of previous comments, we will assume the application performing the compression has been wise enough to

convert the image from the original colour space to the YCbCr colour space. Henceforth, we refer to an individual matrix as if it represents the image, even though it represents only one component of the YCbCr information if the image is a colour image.

The value of a YUV-type colour space is that downsampling can be done to the chroma — it would never be done to the luminance, for it is too valuable. (It would also be inappropriate to downsample any RGB or HSL component as the important visual information is distributed among all three components [3].) This involves averaging pairs of adajecent pixels in the horizontal and, perhaps, vertical direction. This means that for the downsampled component (Cb or Cr), fewer pixels are saved. Sampling done in the horizontal direction only is designated 2h1v or 4:2:2; the latter notation means there is half as much data stored for each of Cb and Cr as for Y. Similarly, sampling done in both directions is denoted 2h2v or 4:1:1. Simple arithmetic shows the image size is reduced by 1/3 under the first sampling scheme and by 1/2 under the second scheme.

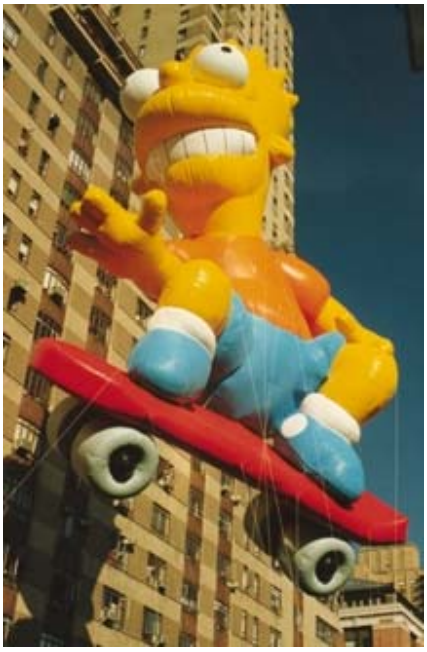
The JPEG Algorithm Step 2

The second step is to break the image into 8×8 pixel blocks. If the width or height is not a multiple of 8, the right-most and or bottom-most blocks are filled with repetitions of the "last" pixel in a row or column as needed. Although this padding is stored in the image file, it is never displayed.

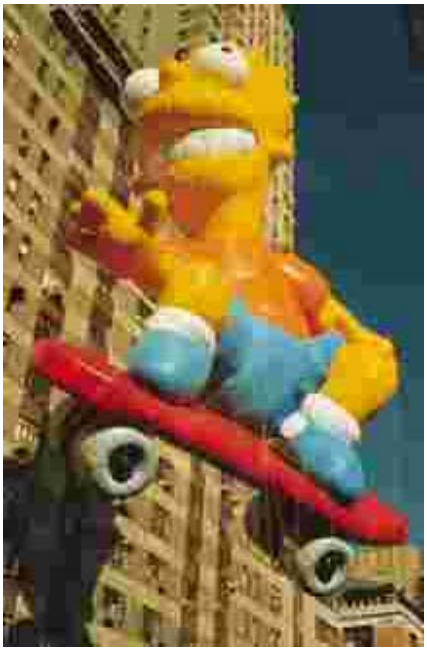
Although applying the following step in the algorithm to the entire image would produce better compression, the computation would require too many arithmetic computations to achieve the desired speed of encoding and decoding. Moreover, we will see that similarities between neighbouring pixels (for the type of images for which the JPEG method is designed) are crucial to JPEG's success in compressing an image. Since similarities do not, in general, extend very far, limiting one's view to smaller, 8×8 pixel blocks is reasonable.

Much as graph theory can be reduced to a study of connected graphs since two components of a graph have no interaction, each 8×8 block is independent of all others. Henceforth, we deal only with these individual 8×8 *data units* and refer to an individual matrix as if it represented the entire image. In fact, it represents only a tiny area of the image, and, for colour images, it only conveys the information for one of three components for that area.

These blocks become more evident as the compression ratio is increased. This can be seen in Figure 1, where the file sizes of the colour images were 70,259 bytes for the maximum quality version in (a) and only 3946 bytes for the much cruder version in (b).



(a)



(b)

Figure 1

The JPEG Algorithm Step 3

The third step is the heart of the JPEG algorithm. The idea is to use the Discrete Cosine Transform (DCT) to translate the data into a "spectrum" of "spatial frequencies." This is a simpler cousin of the Discrete Fourier Transform (DFT). The DCT lacks the DFT's terms with the sine function and the imaginary unit; hence, the DCT always produces a real number. Unlike the DFT, the DCT does not assume that the input values are periodic. This was the deciding feature in favour of the DCT, despite the fact that the DCT involves more arithmetic operations than the DFT. The DCT also has a better smoothing effect to minimize the contrast between neighbouring blocks.

Given an input matrix `p`, the following *Derive* code creates the element in one particular `row` and `col`umn of the discrete-cosine-transformed matrix:

```
DCT_ELT(p,row,col):=
    1/4·c↓[row+1]·c↓[col+1]·Σ(Σ(p↓[r_+1,c_+1]·COS((2·r_+ 1)·row·π/16)·
    COS((2·c_+1)·col·π/16),c_,0,7),r_,0,7)
```

In this definition, we assume that the coefficients `c↓1` to `c↓8` have been defined by the code:

```
c := [1/√2, 1, 1, 1, 1, 1, 1, 1]
```

Given an input matrix `p`, the following *Derive* code creates the entire transformed matrix by using the preceding auxiliary function 64 times.

```
DCT(p):=VECTOR(VECTOR(DCT_ELT(p,r__,c__),c__,0,7),r__,0,7)
```

The entry in Row 0, Column 0 is often referred to as the DC component, and the other 63 are called AC components. (This "direct current/alternating current" language harkens back to the electrical engineering connection to the DFT.) The most important of the 64 components, the DC is 8 times the average of the actual values of all 64 pixels in the data unit; no frequency information is involved. (With rounding to an integer, the elements of the transformed matrix have 11 bits of precision at this stage. [4, p. 36]) As we move toward the bottom right, the AC components measure higher and higher frequency information; intuitively, they capture increasingly finer detail.

The decoding process for this step, the Inverse DCT, can be achieved with the two lines of *Derive* code below. The input `t` is a discrete-cosine-transformed matrix. More precisely, `t` has also undergone the remaining steps and their [pseudo-]inverses. The first line creates a single element of the decoded matrix, and the second assembles all 64 elements.

```
IDCT_ELT(t,r_,c_):=1/4·∑(∑(c↓[row+1]·c↓[col+1]·t↓[row+1,col+ 1]·
    COS((2·r_+1)·row·π/16)·COS((2·c_+1)·col·π/16),col,0,7),row,0,7)

IDCT(t):=VECTOR(VECTOR(IDCT_ELT(p,r__,c__),c__,0,7),r__,0,7)
```

This IDCT decoding can, in theory, perfectly undo the DCT step in the encoding algorithm in a system, such as *Derive*, that can perform exact arithmetic. In practice, the internal round-off error means the decoded image is not an absolutely perfect match with the original image, even if the following, deliberately lossy step is skipped. However, a careful pixel-by-pixel census of RGB values in an original image and compressed-decompressed image show that the changes are extremely minute. For typical photographic images, the changes will be imperceptible. Remember, the human eye cannot really distinguish the 16,777,216 different colours obtainable by using one byte of information for each RGB component. (There are situations in which the most subtle changes become evident to human perception, e.g., when creating a left-to-right gradation from one colour to another using vertical bars of "all" intermediate shades.)

The JPEG Algorithm Step 4

The fourth step is where information is *deliberately* discarded so that the following step can achieve greater compression. The 8×8 matrix of data is divided component-wise by a 8×8 *quantization matrix*. The *Derive* code defining one such quantization matrix is shown below.

```
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]]
```

There are several comments that need to be made here. First, this particular quantization matrix, provided by the Joint Photographic Experts Group and quoted in [3] and [5], is intended for use with the Y component; matrices intended for use chroma would typically have more high values since the chroma

can be degraded more than Y while maintaining quality. Second, the particular numbers, which seem to be without rhyme or reason, were determined empirically; given the astronomical number of possible quantization matrices, it is plausible that better ones (improving compression and/or quality) will be found in the future. Third, a developer could specify any quantization matrix whatsoever: one such matrix merely represents 64 of the parameters that can be specified for use with the algorithm, i.e., stored in the file in the manner appropriate for the file format used. (Note that an application that encodes images a particular way and expects to decode only images so encoded would not need to store fixed parameters in a compressed image file.)

Because of this tremendous flexibility, there is no standard notation for "quality levels." Though it is common to use a 1-100 scale, any graphics manipulation program — Adobe Photoshop, Macromedia Fireworks, Corel Photo-Paint, to name just a few — can designate and implement different quality levels in any way the vendor chooses. One approach would be to have a very small set of quantization matrices and then scale them. However, each application could have its own set of quantization matrices. At http://www.kings.edu/mmichael/psu2/JPEGdemo.html is a comparison of different quality versions of the images shown in Figure 1, made in 1999 with a now-old version of Fireworks; the lowest quality version, which is worse than Figure 1(b), could not even be opened by contemporary versions of other graphics programs!

One important generalization that can be made is this: In general, one would choose smaller numbers toward the upper-left corner of the matrix and larger numbers toward the lower-right corner. This is because the importance of the image data decreases as we move from the upper-left to lower-right. Dividing by a lower number discards less information than does dividing by a higher. Since all the divisions are rounded to an integer for storage, what happens in practice is that many zeroes are created, especially for the lower quality levels.

At this juncture we should clarify the somewhat muddy issue of lossless JPEGs. Some authors claim JPEG compression is inherently loss. On the other hand, choosing a quality level labelled "100" or "100%" when saving a JPEG image from a typical image manipulation application gives one the impression that a lossless image is being created. At best, this choice would skip the quantization step or (equivalently) would use a quantization matrix of all ones. But, as mentioned before, this would not preclude round-off error resulting from performing the DCT followed by the IDCT. There actually is a completely separate lossless JPEG mode which uses a two-dimensional Differential Pulse Code Modulation (a form of Predictive Lossless Coding) rather than the DCT; its produces a much worse compression ratio than any mode using the DCT — only about 2:1 [3].

The straightforward *Derive* code to execute the quantization on a discrete-cosine-transformed matrix `t` using a quantization matrix `q` is:

```
QUANT(t,q):=
    VECTOR(VECTOR(FLOOR(t↓[r_+1,c_+1]/q↓[r_+1,c_+1]+0.5),c_,0,7),r_,0,7)
```

The even simpler "inverse" function, below, operates on a matrix `u` to which both the DCT and quantization have been applied. It cannot undo the rounding performed by `QUANT()`.

```
IQUANT(u,q):=VECTOR(VECTOR(u↓[r_+1,c_+1]·q↓[r_+1,c_+1],c_,0,7),r_,0,7)
```

The JPEG Algorithm Step 5

By themselves, Steps 3 and 4 do not reduce image size. (Step 3 actually converts 64 8-bit values to 64 11-bit values!) It is only in this next step that we appreciate the impact of the divisions performed propitiously (in the quantization step) thanks to the ordering (in the DCT step) of data based on its importance. What happens is, the entries in the array become more similar. There are many more repetitions. In more-severe quantizations, many zeroes result. The existence of repeated values means we can apply a more straightforward type of compression, referred to *entropy coding*. The JPEG baseline process specifies that Huffman coding be used. An allowable alternative is arithmetic coding. We will not elaborate on either of these here.

Regardless of how the data produced by Steps 1-4 is had been compressed and which form of entropy coding is used in this step, no additional loss of data occurs due to this step and its inverse.

How (Not) to Use Derive to Explore the JPEG Algorithm

There are several ways in which *Derive* is not useful in investigating JPEGs. The most obvious is that it is not going to display actual pictures that result from making choices of parameters. It is also not that easy to deal with a large array of data.

Where *Derive* excels is in letting us see (in a convenient display) what happens to a single numerical 8×8 matrix as it passes through Steps 3 and 4 and their respective inverses. Simply saving an image with several different quality levels does not provide any insight into what is secretly going on inside the graphics application. Although each graphics application will retain its own secrets, such as what quantization matrices and scaling factors are being used, we can get a better idea of how decisions at that level influence the outcome. There are several questions that can be addressed with suitable exercises.

(1) How does the DCT react to different types of data. For instance, if there is some kind of repetition, cyclical or otherwise, what shows up in the low- and high-frequency components of the transformed matrix? Some patterns in matrices are easy to generate in *Derive* by writing a line of code, rather than by entering all the entries manually. For example, the code

```
VECTOR(VECTOR(IF(MOD(i, 3) = 2, 128, 10), i, 1, 8), j, 1, 8)
```

simplifies to a matrix with a cyclical pattern of homogeneous columns (all 10s, all 128s, all 10s, repeat).

It should be mentioned that a useless idea is to generate a matrix of random data. First, random data with a uniform frequency distribution over the range of possible values should not compress significantly. More importantly, such a matrix in no way represents the typical situation in a photograph.

However, converting from a real photograph, even a tiny 8×8 pixel one, to a matrix in *Derive* is not a simple process. Examining each pixel in an application for its R (or G or B) values and manually entering them in a matrix would be painful. It is possible to write a program (say, in C) to extract the colour

components of pixels within a standard TIFF or Windows bitmap and save them in a text file which can be used as input by *Derive*.

An easier solution is to create fake "photo matrices" that exhibit different types of transitions. This can be done manually or with code. The preceding (10-128-10-repeat) example is the most extreme form of transition. More realistic would be more gradual changes among neighbouring pixels. A very simple example is:

```
VECTOR(VECTOR(i + j, i, 1, 8), j, 1, 8).
```

This has homogeneous diagonals that run from bottom-left to top-right; the uniform value increases linearly from top-left to bottom-right.

Clearly this can easily be made more interesting by applying a non-linear function to the sum or using some other function with a two-dimensional domain. Gradations across an entire block are not the only feature that should be investigated. Crucial in image perception are *edges*, relatively abrupt boundaries between regions of greater homogeneity. To undertake a thorough study, the collection of fake "photos" should incorporate edge features that not only have different "slopes" but also shapes (if one visualizes the graph of values across a one-dimensional cross-section through the edge). In particular, some samples should have values rise and fall, while others have "plateaus." The former might represent a telephone wire across the sky; the latter might represent the shades in a fold of clothing. (See [4, pp 34-5] for a canonical set of samples.)

Another exercise is to manually doctor a fake photo matrix to introduce an anomaly, representing, for instance, a dust particle in a scan, to see how the DCT reacts to this minute change.

(2) How does round-off error make a theoretically lossless DCT-IDCT encoding/decoding lossy in practice, even in the absence of any quantization? The obvious thing to do is start with a matrix, run it through the DCT and then the IDCT; the resulting matrix can be subtracted from the original matrix for a clearer view of how much damage is being done. The power of *Derive* can be used to automate this, for example, by applying this `p - IDCT(DCT(p))` process to a collection of matrices (which may themselves have been generated by code as mentioned before). Though not made explicit in the formula given earlier, here we must be sure that the DCT result has been rounded to an integer before the IDCT is applied.

Despite this rounding to an integer, the precision with which the operations are computed prior to that rounding is of interest. See [1] for a study involving binary (rather than decimal) precision. *Derive*'s ability to do "infinite-precision" arithmetic is an obvious asset in this context, but another feature is of particular note. It is the ability to program the rounding process with ease. The function `APPROX(_, _)` will, when simplified, approximate its first parameter to the number of decimal places indicated by its second parameter. The beauty of being able to write this code (rather than just click on some buttons to execute an approximation) is that we can vary the second parameter. Using the `VECTOR` construct, we can instantly compare different approximations of the same expression. For our purposes, the approximation function must be applied *within* the DCT and the IDCT. Experiments of this type can reveal

(approximately) how much precision would be necessary to achieve a lossless DCT-IDCT encoding/decoding, i.e., no discrepancies between the original and final integers in the matrix. Bearing in mind that the results will depend on the particular matrix chosen, the experiment can be automated to process a collection of matrices.

(3) What is the effect of different types of quantization? In one way, this is an area of experimentation which is nearly boundless: if each quantization coefficient ranges from 1-128, there are 128^{64} or on the order of 10^{134} different matrices! On the other hand, there is a chilling constraint. The ultimate criterion by which the suitability of a quantization matrix must be judged is human perception. Specifically, what is satisfactory for quantizing chroma may not be for quantizing luminance. *Derive* will not be able to help us in perceptual matters.

Nevertheless, experimentation with quantization can be revealing. It is not of interest to know how dividing by 3 differs from dividing by 2. What is of interest is the effect of quantization (and "dequantization") sandwiched between the DCT and IDCT.

The obvious candidates for quantization matrices are those whose values grow as we move from upper-left to lower-right (as in "addition" matrix coded above); the reasoning is that higher-frequency components can be more readily sacrificed. From the sole example we saw earlier, it is obvious that a regular pattern need not exist in a real-world quantization matrix. The three categories of matrices we can compare are (a) the real-world ones, (b) the ones that seem logical, and (c) those that seem "backwards" or counter-productive.

Conclusion

The JPEG family of image compression tools is complex. Furthermore, the DCT that is at the heart of the algorithm is conceptually more sophisticated than many approaches to data compression. More complications come from contradictions within the literature on JPEG issues, imprecise use of terminology, and software options that can be misleading.

The *Derive* computer algebra system is not well suited to dealing with many issues involved in this complicated area, the most important being those that pertain directly to human perception. However, it does provide tools that allow one to examine in detail two key steps in the algorithm. Experiments can be performed to gain an understanding of how separating low-frequency data from high-frequency data (Step 3) sets the stage for selectively discarding data (Step 4) with the goal of preserving the more important, low-frequency data. Though lacking the human touch, these cold, quantitative comparisons of numbers provide some *mathematical* insight into this important, ubiquitous compression technology and the transformation at its heart.

References

[1] José Bins, et al., "Precision vs. Error in JPEG Compression," available at: http://www.cs.colostate.edu/cameron/Publications/bin_spiec99.pdf.

[2] John Miano, Compressed Image File Formats, ACM Press/Addison-Wesley, 1999.

[3] James D. Murray and William van Ryper, Encyclopedia of Graphics File Formats, Second Edition, O'Reilly and Associates, 1996.

[4] William B. Pennebaker and Joan L. Mitchell, JPEG Still Image Compression Standard, Van Nostrand Reinhold, 1993.

[5] David Salomon, Data Compression, Springer-Verlag New York, 1998.