Lossy Image Compression for Continuous Tone Graphical Images

T. Vrind and K. Ramathreya
Advisor Dr. P. K. Mahanti
Department of Computer Science and Engineering
Birla Institute of Technology, Mesra, Ranchi, India

INTRODUCTION

Programs using complex graphics are showing up in virtually every area of computing applications; games, education, desktop publishing, and graphical design, just to mention a few. These programs have one factor in common. The images they use consume prodigious amounts of disk storage. In the IBM world, for example, the VGA display is probably the current lowest common denominator for high-quality color graphics. VGA displays 256 simultaneous colors selected from a palette of 262,144 colors. This lets the VGA display continuous tone images, such as color photographs, with a reasonable amount of fidelity. The problem with using images of photographic quality is the amount of storage required to use them in a program. For the previously mentioned VGA, a 256-color screen image has 200 rows of 320 pixels, each consuming a single byte of storage. This means that a single screen image consumes a minimum of 64K! It isn't hard to imagine applications that would require literally hundreds of these images to be accessed. The above discussion shows explicitly the very important use of image compression techniques to the different applications. Image compression on one hand saves valuable memory space for storage of continuous tone images, on the other hand it serves for transfer of still as well as video images between computers. In our project titled "SIMULATING IMAGE COMPRESSION" we wish to implement and discuss the use of lossy compression techniques (similar to the JPEG compression techniques and algorithms) to achieve very high levels of compression on continuous tone graphical images, such as digitized images of photographs. We planned to complete this project in two phases (2 semesters). In the first phase (i.e VIth Semester) we studied the compression techniques as given by JPEG and successfully implemented the DCT transformation, (using Fast Fourier Transformations), Inverse DCT Transformation and Quantization. The final part of Image compression i.e lossy compression of .GS files of the images and development of the interfaces for the compression of this image files was dealt with in the second phase of the our project .All the coding in the project has been done in the language `C'.

DESIGN

The design approach to this project is same as that of the JPEG algorithm. The image is taken in form of a black and white or a gray scale file. It is a normal text file with a ".gs" type extension name, consisting of a 320 by 200 matrix of pixel values .The various transformations and approximations are applied to the "*.gs" file and a smaller memory size requirement for the image is found without degrading the quality of the image.

REQUIREMENTS

Hardware & Software requirements:

- Processor (Pentium I or above).
- 50 MB of free-space (for image files and compressing programs).
- Window 95 with 256 colors.
- MS Word for Documentation.
- SVGA if available.
- Scanner if available for scanning of the image to be compressed.

JPEG COMPRESSION

JPEG (pronounced "jay-peg") is a standardized image compression mechanism. JPEG stands for Joint Photographic Experts Group, the original name of the committee that wrote the standard.

JPEG is designed for compressing either full-color or gray-scale images of natural, real-world scenes. It works well on photographs, naturalistic artwork, and similar material; not so well on lettering, simple cartoons, or line drawings. JPEG handles only still images, but there is a related standard called MPEG for motion pictures.

JPEG is "lossy," meaning that the decompressed image isn't quite the same as the one you started with. (There are lossless image compression algorithms, but JPEG achieves much greater compression than is possible with lossless methods.) JPEG is designed to exploit known limitations of the human eye, notably the fact that small color changes are perceived less accurately than small changes in brightness. Thus, JPEG is intended for compressing images that will be looked at by humans. If one plans to machine-analyze the images, the small errors introduced by JPEG may be a problem, even if they are invisible to the eye.

A useful property of JPEG is that the degree of lossiness can be varied by adjusting compression parameters. This means that the image maker can trade off file size against output image quality. You can make *extremely* small files if

you don't mind poor quality; this is useful for applications such as indexing image archives. Conversely, if you aren't happy with the output quality at the default compression setting, you can jack up the quality
until you are satisfied, and accept lesser compression.

Another important aspect of JPEG is that decoders can trade off decoding speed against image quality, by using fast but inaccurate approximations to the required calculations. Some viewers obtain remarkable speedups in this way. (Encoders can also trade accuracy for speed, but there's usually less reason to make such a sacrifice when writing a file.)

USE OF JPEG

There are two good reasons for using JPEG:

- To make image files smaller
- To store 24-bit-per-pixel color data instead of 8-bit-per-pixel data.

Making image files smaller is a win for transmitting files across networks and for archiving libraries of images. Being able to compress a 2 Mbyte full-color file down to, say, 100 Kbytes makes a big difference in disk space and transmission time! . Thus using JPEG is essentially a time/space tradeoff: one gives up some time in order to store or transmit an
image more cheaply. But it's worth noting that when network transmission is involved, the time savings from transferring a shorter file can be greater
than the time needed to decompress the file.

The second fundamental advantage of JPEG is that it stores full color information: 24 bits/pixel (16 million colors). GIF, the other image format widely used on the net, can only store 8 bits/pixel (256 or fewer colors). GIF is reasonably well matched to inexpensive computer displays --- most run-of-the-mill PCs can't display more than 256 distinct colors at once. But full-color hardware is getting cheaper all the time, and JPEG photos look *much* better than GIFs on such hardware. Within a couple of years, GIF will probably seem as obsolete as black-and-white MacPaint format does today. Furthermore, JPEG is far more useful than GIF for exchanging images among people with widely varying display hardware, because it avoids prejudging how many colors to use . Hence JPEG is considerably more appropriate than GIF for use as a Usenet and World Wide Web standard photo format.

A lot of people are scared off by the term "lossy compression". But when it comes to representing real-world scenes, *no* digital image format can retain all the information that impinges on the eyeball. By comparison with the real-world scene, JPEG loses far less information than GIF.
The real disadvantage of lossy compression is that if one repeatedly compresses and decompresses an image, one loses a little more quality each time. This is a serious objection for some applications but matters not at all for many others.

JPEG is *not* going to displace GIF entirely; for some types of images, GIF is superior in image quality, file size, or both. One of the first things to learn about JPEG is which kinds of images to apply it to.

Generally speaking, JPEG is superior to GIF for storing full-color or gray-scale images of "realistic" scenes; that means scanned photographs continuous-tone artwork, and similar material. Any smooth variation in color, such as occurs in highlighted or shaded areas, will be represented more faithfully and in less space by JPEG than by GIF.

GIF does significantly better on images with only a few distinct colors, such as line drawings and simple cartoons. Not only is GIF lossless for such images, but it often compresses them more than JPEG can. For example, large areas of pixels that are all *exactly* the same color are compressed very efficiently indeed by GIF. JPEG can't squeeze such data as much as GIF does without introducing visible defects. (One implication of this is that large single-color borders are quite cheap in GIF files, while they are best avoided in JPEG files.)

Computer-drawn images, such as ray-traced scenes, usually fall between photographs and cartoons in terms of complexity. The more complex and subtly rendered the image, the more likely that JPEG will do well on it. The same goes for semi-realistic artwork (fantasy drawings and such). But icons that use only a few colors are handled better by GIF.

JPEG has a hard time with very sharp edges: a row of pure-black pixels adjacent to a row of pure-white pixels, for example. Sharp edges tend to come out blurred unless you use a very high quality setting. Edges this sharp are rare in scanned photographs, but are fairly common in GIF files: consider borders, overlaid text, etc. The blurriness is particularly
objectionable with text that's only a few pixels high. Most recent JPEG software can deal with textual comments in a JPEG file, although older viewers may just ignore the
comments.

Plain black-and-white (two level) images should never be converted to JPEG; they violate all of the conditions given above. One needs at least about 16 gray levels before JPEG is useful

for gray-scale images. It should also be noted that GIF is lossless for gray-scale images of up to 256 levels, while JPEG is not.

If one has a large library of GIF images, one may want to save space by converting the GIFs to JPEG. This is trickier than it may seem --- even when the GIFs contain photographic images, they are actually very poor source material for JPEG, because the images have been color-reduced. Non-photographic images should generally be left in GIF form. Good-quality photographic GIFs can often be converted with no visible quality loss, but only if one knows what one is doing and one takes the time to work on each image individually. Otherwise it is likely to lose a lot of image quality or waste a lot of disk space ... quite possibly both.

How well does JPEG compress images

Very well indeed, when working with its intended type of image (photographsand suchlike). For full-color images, the uncompressed data is normally 24 bits/pixel. The best known lossless compression methods can compress such data about 2:1 on average. JPEG can typically achieve 10:1 to 20:1 compression without visible loss, bringing the effective storage requirement down to 1 to 2 bits/pixel. 30:1 to 50:1 compression is possible with small to moderate defects, while for very-low-quality purposes such as previews or archive indexes, 100:1 compression is quite feasible. An image compressed 100:1 with JPEG takes up the same space as a full-color one-tenth-scale thumbnail image, yet it retains much more detail than such a thumbnail.

For comparison, a GIF version of the same image would start out by sacrificing most of the color information to reduce the image to 256 colors (8 bits/pixel). This provides 3:1 compression. GIF has additional "LZW" compression built in, but LZW doesn't work very well on typical photographic data; at most one may get 5:1 compression overall, and it's not at all uncommon for LZW to be a net loss (i.e., less than 3:1 overall compression). LZW *does* work well on simpler images such as line drawings, which is why GIF handles that sort of image so well. When a JPEG file is made from full-color photographic data, using a quality setting just high enough to prevent visible loss, the JPEG will typically be a factor of four or five smaller than a GIF file made from the same data.

Gray-scale images do not compress by such large factors. Because the human eye is much more sensitive to brightness variations than to huge variations JPEG can compress hue data more heavily than brightness (gray-scale) data. A gray-scale JPEG file is generally only about 10%-25% smaller than a full-color JPEG file of

similar visual quality. But the uncompressed gray-scale data is only 8 bits/pixel, or one-third the size of the color data, so the calculated compression ratio is much lower. The threshold of visible loss is often around 5:1 compression for gray-scale images.

The exact threshold at which errors become visible depends on viewing conditions. The smaller an individual pixel, the harder it is to see an error; so errors are more visible on a computer screen (at 70 or so dots/inch) than on a high-quality color printout (300 or more dots/inch). Thus a higher-resolution image can tolerate more compression ... which is fortunate considering it's much bigger to start with. The compression ratios quoted above are typical for screen viewing. Also note that the threshold of visible error varies considerably across images.

"Quality" settings for JPEG

Most JPEG compressors lets one pick a file size vs. image quality tradeoff by selecting a quality setting. There seems to be widespread confusion about the meaning of these settings. "Quality 95" does NOT mean "keep 95% of theinformation", as some have claimed. The quality scale is purely arbitrary; it's not a percentage of anything.

In fact, quality scales aren't even standardized across JPEG programs. For example:

- Apple used to use a scale running from 0 to 4, not 0 to 100.
- Recent Apple software uses an 0-100 scale that has nothing to do with
- the IJG scale (their Q 50 is about the same as Q 80 on the IJG scale).
- Paint Shop Pro's scale is the exact opposite of the IJG scale, PSP
- setting N = IJG 100-N; thus lower numbers are higher quality in PSP.
- Adobe Photoshop doesn't use a numeric scale at all, it just gives "high"/"medium"/"low" choices.

Fortunately, this confusion doesn't prevent different implementations from exchanging JPEG files. But one does need to keep in mind that quality scales vary considerably from one JPEG-creating program to another, and that just saying "I saved this at Q 75" doesn't mean a thing if one does noy say which program he has used.

In most cases the user's goal is to pick the lowest quality setting, or smallest file size, that decompresses into an image indistinguishable from the original. This

setting will vary from one image to another and from one observer to another, but here are some rules of thumb.

For good-quality, full-color source images, the default qIuality setting is very often the best choice. This setting is about the lowest one can go without expecting to see defects in a typical image.

If the image was less than perfect quality to begin with, one might be able to drop down to lower quality without objectionable degradation. On the other hand, one might need to go to a *higher* quality setting to avoid further loss. This is often necessary if the image contains dithering or moire patterns

Except for experimental purposes, never go above about Q 95; using Q 100 will produce a file two or three times as large as Q 95, but of hardly any better quality. Q 100 is a mathematical limit rather than a useful setting. If one sees a file made with Q 100, it's a pretty sure sign that the maker didn't know what he/she was doing.

If one wants a very small file (say for preview or indexing purposes) and is prepared to tolerate large defects, a Q setting in the range of 5 to 10 is about right. Q 2 or so may be amusing as "op art". (It's worth mentioning that the current IJG software is not optimized for such low quality factors. Future versions may achieve better image quality for the same file size at low quality settings.)

If image contains sharp colored edges, one may notice slight fuzziness or jagginess around such edges no matter how high one make the quality setting. This can be suppressed, at a price in file size, by turning off chroma downsampling in the compressor. The IJG encoder regards downsampling as a separate option which you can turn on or off independently of the Q setting. Other JPEG implementations may or may not provide user control of

downsampling. Adobe Photoshop, for example, automatically switches off downsampling at its higher quality settings. On most photographic images, we recommend leaving downsampling on, because it saves a significant amount of space at little or no visual
penalty.

For images being used on the World Wide Web, it's often a good idea to give up a small amount of image quality in order to reduce download time. Quality settings around 50 are often perfectly acceptable on the Web. In fact, a user viewing such an image on a browser with a 256-color display is unlikely to be able to see any difference from a higher quality setting, because the browser's color quantization artifacts will swamp any imperfections in the JPEG image itself. It's also worth knowing that current progressive-JPEG-making programs use default progression

sequences that are tuned for quality settings around 50-75: much below 50, the early scans will look really bad, while much above 75, the later scans won't contribute anything noticeable to the picture.

Loss accumulatation with repeated compression/decompression

It would be nice if, having compressed an image with JPEG, you could decompress it, manipulate it (crop off a border, say), and recompress it without any further image degradation beyond what you lost initially. Unfortunately THIS IS NOT THE CASE. In general, recompressing an altered image loses more information. Hence it's important to minimize the number of generations of JPEG compression between initial and final versions of an image.

There are a few specialized operations that can be done on a JPEG file without decompressing it, and thus without incurring the generational loss that you'd normally get from loading and re-saving the image in a regular image editor. In particular it is possible to do 90-degree rotations and flips losslessly, if the image dimensions are a multiple of the file's block size (typically 16x16, 16x8, or 8x8 pixels for color JPEGs). This fact used to be just an academic curiosity, but it has assumed practical importance recently because many users of digital cameras would like to be able to rotate their images from landscape to portrait format without incurring loss --- and practically all digicams that produce JPEG files produce images of the right dimensions for these operations to work. So software that can do lossless JPEG transforms has started to pop up. But you do need special software; rotating the image in a regular image editor won't be lossless.

It turns out that if you decompress and recompress an image at the same quality setting first used, relatively little further degradation occurs. This means that you can make local modifications to a JPEG image without material degradation of other areas of the image. (The areas you change will still degrade, however.) Counterintuitively, this works better the
lower the quality setting. But you must use *exactly* the same setting, or all bets are off. Also, the decompressed image must be saved in a full-color format; if you do something like JPEG=>GIF=>JPEG, the color quantization step loses lots of information.

Unfortunately, cropping doesn't count as a local change! JPEG processes the image in small blocks, and cropping usually moves the block boundaries, so that the image looks completely different to JPEG. You can take advantage of the low-degradation behavior if you are careful to crop the top and left margins only by a multiple of the block size (typically 16

pixels), so that the remaining blocks start in the same places. (True lossless cropping is possible under the same restrictions about where to crop, but again this requires specialized software.)

The bottom line is that JPEG is a useful format for compact storage and transmission of images, but you don't want to use it as an intermediate format for sequences of image manipulation steps. Use a lossless 24-bit format (PNG, TIFF, PPM, etc) while working on the image, then JPEG it when you are ready to file it away or send it out on the net. If you expect to edit your image again in the future, keep a lossless master copy to work from.

Progressive JPEG

A simple or "baseline" JPEG file is stored as one top-to-bottom scan of the image. Progressive JPEG divides the file into a series of scans. The first scan shows the image at the equivalent of a very low quality setting, and therefore it takes very little space. Following scans gradually improve the quality. Each scan adds to the data already provided, so that the total storage requirement is roughly the same as for a baseline JPEG image of the same quality as the final scan. (Basically, progressive JPEG is just a rearrangement of the same data into a more complicated order.)

The advantage of progressive JPEG is that if an image is being viewed on-the-fly as it is transmitted, one can see an approximation to the whole image very quickly, with gradual improvement of quality as one waits longer; this is much nicer than a slow top-to-bottom display of the image. The disadvantage is that each scan takes about the same amount of computation to display as a whole baseline JPEG file would. So progressive JPEG only makes sense if one has a decoder that's fast compared to the communication link.(If the data arrives quickly, a progressive-JPEG decoder can adapt by skipping some display passes. Hence, those of you fortunate enough to have T1 or faster net links may not see any difference between progressive and regular JPEG; but on a modem-speed link, progressive JPEG is great.)

Lossless JPEG

There's a great deal of confusion on this subject, which is not surprising because there are several different compression methods all known as "JPEG". The commonly used method is "baseline JPEG" (or its variant "progressive JPEG"). The same ISO standard also defines a very different method called "lossless

JPEG". And if that's not confusing enough, a new lossless standard called "JPEG-LS" is about to hit the streets.

When I say "lossless", I mean mathematically lossless: a lossless compression algorithm is one that guarantees its decompressed output is bit-for-bit identical to the original input. This is a much stronger claim than "visually indistinguishable from the original". Baseline JPEG can reach visual indistinguishability for most photo-like images, but it can never be truly lossless.

Lossless JPEG is a completely different method that really is lossless. However, it doesn't compress nearly as well as baseline JPEG; it typically can compress full-color data by

around 2:1. And lossless JPEG works well only on continuous-tone images. It does not provide useful compression of palette-color images or low-bit-depth images.

Lossless JPEG has never been popular --- in fact, no common applications support it --- and it is now largely obsolete. (For example, the new PNG standard outcompresses lossless JPEG on most images.) Recognizing this, the ISO JPEG committee recently finished an all-new lossless compression standard called JPEG-LS (you may have also heard of it under the name LOCO). JPEG-LS gives better compression than original lossless JPEG, but still nowhere near what you can get with a lossy method. It's anybody's guess whether this new standard will achieve any popularity.

It's worth repeating that cranking a regular JPEG implementation up to its maximum quality setting *does not* get you lossless storage; even at the highest possible quality setting, baseline JPEG is lossy because it is subject to roundoff errors in various calculations. Roundoff errors alone are nearly always too small to be seen, but they will accumulate if you put the image through multiple cycles of compression . Many implementations won't even let you get to the maximum possible setting, because it's such an inefficient way to use regular JPEG. With the IJG JPEG software, for example, you have to not only select "quality 100" but also turn off chroma downsampling to minimize loss of information. The resulting
files are far larger and of only fractionally better quality than files generated at more reasonable settings. And they're still slightly lossy! If you really need lossless storage, don't try to approximate it with regular JPEG.

The JPEG lossy compression algorithm operates in three sucessive stages namely

- Discrete Cosine Transformation
- Coefficient Quantization
- Lossless Compression

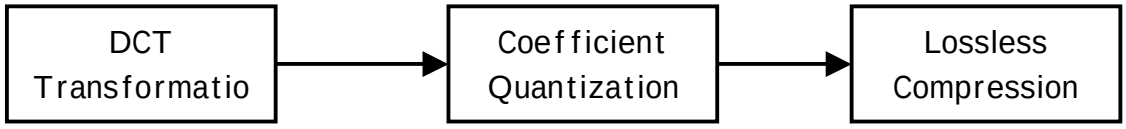

Figure 1. JPEG LOSSY COPRESSION

These three steps combine to form a powerful compressor, capable of compressing continuous tone images to less than 10 percent of their original size, while losing little, if any , of their original fidelity.

DISCRETE COSINE TRANSFORM

The key to the compression process is a mathematical transformation known as Discrete Coefficient Quantization (DCT). The DCT is a class of mathematical operations that includes the well known Fast Fourier Transform (FFT), as well as many others. The basic operation performed by these transforms is to take signal and to transform it from one type of representation to another.

This transform is done frequently when analyzing digital audio samples using the FFT. When we collect a set of sample points from an incoming audio signal, we end up with the representation of a signal in the time domain. That is, we have a collection of points that show what the voltage level was for the input signal at each point in time .The FFT transforms the set of sample points into a set of frequency values that describes exactly the same signal.

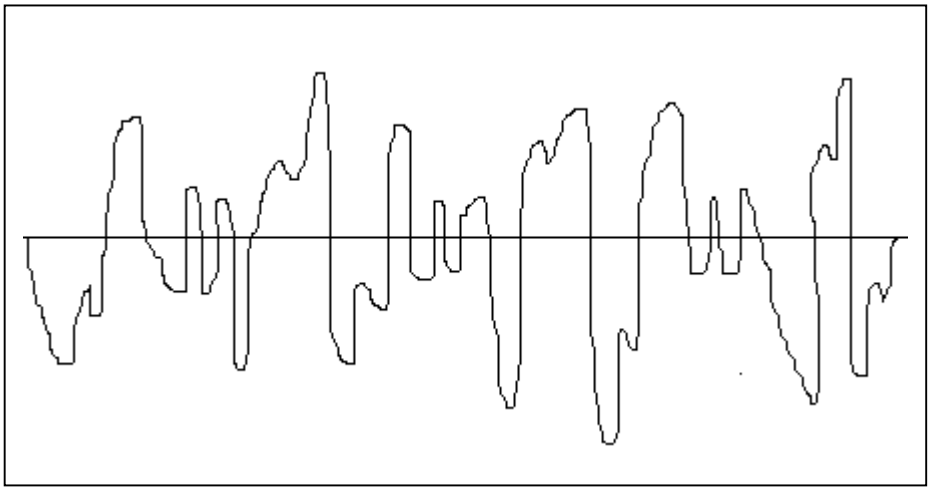

Figure 2 . ANALOG REPRESENTATION OF THREE SINE WAVES IN TIME DOMAIN

Figure 3 . DATA POINTS AFTER FFT PROCESSING

The X and Y-axes are the two dimensions of the screen. The amplitude of the "signal" in this case is simply the value of a pixel at a particular point on the screen. For the examples used in this chapter, that is simply an eight-bit value used to represent a gray-scale value. So a graphical image displayed on the screen can be thought of as a complex three-dimensional signal, with the value on the Z axis denoted by the color on the screen at a given point. This is the spatial representation of the signal.
The DCT can be used to convert spatial information into "frequency" or "spectral" information, with the X and Y-axes representing frequencies of the signal in two different dimensions. And like the FFT, there is an Inverse DCT (IDCT) function that can convert the spectral representation of the signal back to a spatial one.

DCT specifics

The actual formula for the two-dimensional DCT is shown in figure 7, with its partner, the IDCT, shown immediately below in figure 8. The DCT is performed on a N x N square matrix of pixel values, and it yields an N x N square matrix of frequency coefficients. The formula books somewhat intimidating at first glance, but it can be done with a relatively straightforward piece of code.
To write code to implement this function, it first becomes clear that simple table lookups can replace many terms of the equation. The two cosine terms that have to be multiplied together only need to be calculated once at the beginning for the program, and they can be stored for later use. Likewise, the C(x) terms that fall outside the summation loops can also be replaced with table lookups. Once that is done, code to compute the N-by-N portion of a display looks somewhat like that shown below:

```
for ( i = o ; i < n ; i++ )
  for ( j = 0 ; j < N ; j++ ) {


 temp = 0.0;
           for ( x = 0 ; x < N ; x++)
              for ( y = 0 ; y < N ; y++) {
                temp += Cosines [ x ] [ i ] *
                                Cosines [ y ] [ j ] *
                                 pixel [ x ] [ y ];
}
temp *= sqrt( 2 * N ) * Coefficients[ i ] [ j ];
DCT [ i ][ j ] = INT_ROUND( temp );
}
```

While this code fragment looks as though it may be somewhat interesting to a mathematician, why anyone would want to use it on a graphical image is not immediately obvious. After we transform the pixels to frequency coefficients, we still have just as many points as before. It doesn't seem as if that is a particularly good way to go about

compressing data. It would be much more impressive if the DCT took an N-by-N matrix of data and transformed it to an N/2 by N/2 matrix.

However, figure 6 provides a clue as to what the JPEG committee sees in this algorithm. Figure 6 shows that the spectral representation of the audio waveform takes all the information needed to describe the waveform and packs it into the three non-zero points on the graph. So in principle we could describe the 512 points that make up the input waveform with just three points of frequency data.

The DCT accomplishes something similar when it transform data. In the N-by-N matrix, all the elements in row 0 have a frequency component of zero in one direction of the signal. All the elements in column 0 have a frequency component of zero in the other direction. As the rows and columns move away from origin, the coefficients in the transformed DCT matrix begin to represent higher frequencies, with the highest frequencies found at position N-1 of the matrix.

This is significant because most graphical images on our computer screens are composed of low-frequency information. As it turns out, the components found in row and column 0 (the DC components) carry more useful information about the image than the higher-frequency components. As we move farther away from the DC components in the picture, we find that the coefficients not only tend to have lower values, but they become far less important for describing the picture.

So the DCT transformation identifies pieces of information in the signal that can be effectively "thrown away" without seriously compromising the quality of the image. It is hard to imagine how we would do this with a picture that hadn't been transformed. With the image still described in spatial terms, using pixels, a program would have a difficult time figuring out which pixels are important to the overall look of the picture and which aren't.

One of the first things that shows up when examining the DCT algorithm is that the calculation time required for each element in the DCT is heavily dependent on the size of the matrix. Since a doubly nested loop is used, the number of calculations is Order (N squared): as N goes up, the amount of time required to process each element in the DCT output array will go up dramatically.
One of the consequences of this is that it is virtually impossible to perform a DCT on an entire image. The amount of calculation needed to perform a DCT transformation on even a 256-by-256 gray-scale block is prohibitively large. To get around this, DCT implementations typically break the image down into smaller, more manageable blocks. The JPEG group selected an 8-by-8 block for the size of their DCT calculation.

While increasing the size of the DCT block would probably give better compression, it doesn't take long to reach a point of diminishing returns. Research shows that the connections between pixels tend to diminish quickly, such that pixels even fifteen or twenty positions away are of very little use predictors. This means that a DCT block of 64-by-64 might not compress much better than if we broke it down into four 16-by-16 blocks. And to make matters worse, the computation time would be much longer.

While there is probably a good argument for using 16-by-16 blocks as the basis for DCT computations, the JPEG committee elected to stick with 8-by-8. Much of this was motivated by a desire to allow for practical implementations that could be built using today's technology. This type of compression is referred to as "block coding."

Matrix Multiplication

The definition of the DCT shown above is a relatively straightforward, doubly nested loop. The inner element of the loop gets executed N*N times for every DCT element that is calculated. The inner line of the loop has two multiplication operations and a single addition operation.
A considerably more efficient form of the DCT can be calculated using matrix operations. To perform this operation, we first create an N-by-N matrix known as the Cosine Transform matrix, C. This matrix is defined by the equation shown in figure 9.
Once the Cosine Transform matrix has been built, we transpose it by rotating it around the main diagonal. This matrix is referred to in code as Ct, the Transposed Cosine Transform matrix. Building this matrix is done only once during program initialization. Both matrices can be built at the same time with a relatively short loop, shown below :

```
for ( j = 0 ; j < N ; j++ ) {
        C[ 0 ][ j ] = 1.0 / sqrt( N );
        Ct[ j ][ 0 ] = C[ 0 ][ j ];
}
for (i = 1 ; i < N ; i++ ) {
        for ( j = 0 ; j < N ; j++ ) {
           C[ i ][ j ] = sqrt( 2.0 / N )*
                              cos( ( 2 * j + 1 ) * i * pi
                                    / ( 2.0 * N ) ) ;
          Ct[ j [[ i ] = C[ i ][ j ];
        }
}
```

Once these two matrices have been built, we can take advantage of the alternate definition of the DCT function :

DCT = C * Pixels * Ct

In this particular equation, the `*' operator refers to matrix multiplication, not normal arithmetic multiplication. Each factor in the equation is an N-by-N matrix.

When multiplying two square matrices together, the arithmetic cost of each element of the output matrix will be N multiplication operations and N addition operations. Since we perform two matrix multiplications to create the DCT matrix, each element in the transformed DCT matrix was created at the cost of 2N multiplications and additions, a considerable improvement over the nested loop definition of the DCT used earlier.

```
/* MatrixMultiply( temp, input, Ct ); */
        for ( i = 0 ; i < N ; i++ ) {
          for ( j = 0 ; j < N ;  j++ )  {
                temp[ i ][ j ] = 0.0;
                for ( k = 0 ; k < N ; k++ )
                        temp[ i ][ j ] += (pixel[ i ][k ] ) * Ct{ k ]p j ];
        }
}

/* MatrixMultiply( output, C, temp  ); */
        for ( i = 0 ; i < N ; i++ ) {
          for ( j = 0 ; j < N ;  j++ )  {
                temp1 = 0.0;
                for ( k = 0 ; k < N ; k++ )
                        temp1 += C[ i ][ k ] *  temp[ k ][ j ];
                DCT[ i ][ j ] = temp1;
        }
}
```

A sample piece of code that implements the DCT via matrix arithmetic is shown above. We note that the code is essentially nothing more than a set of two triply nested loops. The first set of loops multiplies the transposed Cosine Transform Matrix by the input setof pixels, creating a temporary matrix. The temporary matrix is then multiplied by the Cosine Transform matrix, which results in the output, the DCT matrix.

Continued Improvements

The versions of the DCT presented here perform the same operations as those used in commercial implementations, but without several more optimization steps needed to produce JPEG compressors that operate in something approaching real time.

One improvement that can be made to the algorithm is to develop versions of the algorithm that only use integer arithmetic. To achieve the accuracy needed for faithful reproduction, the versions of the program tested in this project all stick with reliable floating point math. It is possible, however, to develop versions of the DCT that use scaled integer math, which is considerably faster on most platforms.

Since the DCT is related to the Discrete Fourier Transform, it shouldn't be surprising that many of the techniques used to speed up the family of Fourier Transforms can also be applied to the DCT. In fact, people all over the world are working full time on applying Digital Signal Processing techniques to the DCT. Every cycle shaved off the time taken to perform the transform can be worth a small fortune, so there is good incentive for these research efforts.

Output of the DCT

Figure below shows a representative input block from a gray-scale image. As can be seen, the input consists of an 8-by-8 matrix of pixel values which are somewhat randomly spread around the 140 to 175 range. These integer values are fed to the DCT algorithm, creating the output matrix shown below it.

Input Pixel matrix :

140	144	147	140	140	155	179	175
144	152	140	147	140	148	167	179
152	155	136	167	163	162	152	172
168	145	156	160	152	155	136	160
162	148	156	148	140	136	147	162
147	167	140	155	155	140	136	147
136	156	123	167	162	144	140	147
148	155	136	155	152	147	147	136

Output DCT matrix :

186	-18	15	-9	23	-9	-14	19
21	-34	26	-9	-11	11	14	7
-10	-24	-2	6	-18	3	-20	-1
-8	-5	14	-15	-8	3	-20	-1
-3	10	8	1	-11	18	18	15
4	-2	-18	8	8	-4	1	-7
9	1	-3	4	-1	-7	-1	-2
0	-8	-2	2	1	4	-6	0

The output matrix shows the spectral compression characteristic the DCT is supposed to create. The "DC coefficient" is at position 0,0 in the upper left-hand corner of the matrix. This value represents an average of the overall magnitude of the input matrix, since it represents the DC component in both the X and the Y axis. We note that the DC coefficient is almost an order of magnitude greater than any of the other values in the DCT matrix. In addition, there is a general trend in the DCT matrix, As the elements move farther and farther from the DC coefficient, they tend to become lower and lower in magnitude.

This means that by performing the DCT on the input data, we have concentrated the representation of the image in the upper left coefficients of the output matrix, with the lower right coefficients of the DCT matrix containing less useful information. The next section discusses how this can help compress data.

COEFFICIENT QUANTIZATION

Figure 1. Shows the JPEG compression process as a three-step procedure, the first step being a DCT transformation. DCT is a lossless transformation that doesn't actually perform compression. It prepares for the "lossy", or quantization, stage of the process.
The DCT output matrix takes more space to store than the original matrix of pixels. The input to the DCT function consists of eight-bit pixel values, but the values that come out can range from a low of -1,024 to a high of 1,023, occupying eleven bits. Something drastic needs to happen before the DCT matrix can take up less space.
The "drastic" action used to reduce the number of bits required for storage of the DCT matrix is referred to as "Quantization". Quantization is simply the process of reducing the number of bits needed to store an integer value by reducing the precision of the integer. Once a DCT image has been compressed, we can generally reduce the precision of the coefficient more and more as we move away from the DC coefficient at the origin. The farther away we are from 0,0, the less the element contributes to the graphical image, so the less we care about maintaining rigorous precision in its value.
The JPEG algorithm implements Quantization using a Quantization matrix. For every element position in the DCT matrix, a corresponding value in the quantization matrix gives a quantum value.

The quantum value indicates wht the step size is going to be for that element in the compressed rendition of the picture, with values ranging from one to 255.
The elements that matter most to the picture will be encoded with a small step size, size 1 offering the most precision. Values can become higher as we move away from the origin. The actual formula for quantization is quite simple :

Quantized Value (i, j) = DCT(i,j) / Quantum(i, j) Rounded to nearest integer

From the formula, it becomes clear that quantization values above twenty-five or perhaps fifty assure that virtually all higher-frequency components will be rounded down to zero. Only if the high-frequency coefficients get up to unusually large values will they be encoded as non-zero values.
During decoding the dequantization formula operates in reverse :

DCT(i, j) = Quantized Value(i, j) * Quantum(i, j)

Once again, from this we can see that when you use large quantum values, you run the risk of generating large errors in the DCT output during dequantization. Fortunately, errors generated in the high-frequency components during dequantization normally don't have a serious effect on picture quality.

Selecting Quantization Matrix

Carry an enormous number of schemes could be used to define values in the quantization matrix. At least two experimental approaches can test different quantization schemes. One measures the mathematical error found between an input and output image after it has been decompressed, trying to determine an acceptable level of error. A second approach tries to judge the effect of decompression on the human eye, which may not always correspond exactly with mathematical differences in error levels.

Since the quantization matrix can obviously be defined at runtime when compression takes place. JPEG allows for the use of any quantization matrix; however, the ISO has developed a standard set of quantization values supplied for use by implementers of JPEG code. These tables are based on extensive testing by members of the JPEG committee, and they provide a good baseline for established levels of compression.

One nice feature about selecting quantization matrices at runtime is that it is quite simple to"dial in" a picture quality value when compressing graphics using the JPEG algorithm. By choosing extraordinarily high step sizes for most DCT coefficients, we get excellent compression ratios and poor picture quality. By choosing cautiously low step sizes, compression ratios will begin to slip to not so impressive levels, but picture quality should be excellent. This allows for a great deal of flexibility for the user of JPEG code, choosing picture quality based on both imaging requirements and storage capacity.

The quantization tables used in the test code supplied with this program are created using a very simple algorithm. To determine the value of the quantum step sizes, the user iputs a single "quality factor" which should range from one to about twenty-five. Values larger than twenty-five would work, but picture quality has degraded far enough at quality level 25 to make going any farther an exercise in futility.

```
for ( i = 0 ; i < N ; i++ )
  for ( j = 0 ; j < N ; j++ )
      Quantum[ i ][ j ] = 1 + ( ( 1 + i + j ) * quality );
```

The quality level sets the difference between adjoining bands of the same quantization level. These bands are oriented on diagonal lines across the matrix, so quantization levels of the same value are all roughly the same distance from the origin. An example of what the quantization matrix looks like with a quality factor of two follows.

3	5	7	9	11	13	15	17
5	7	9	11	13	15	17	19
7	9	11	13	15	17	19	21
11	13	15	17	19	21	23	25
13	15	17	19	21	23	25	27
15	17	19	21	23	25	27	29
17	19	21	23	25	27	29	31

As a result of this configuration, the DCT coefficient at 7,7 would have to reach a value of sixteen to be encoded as a value other than zero. This sets the threshold for the value of an element before it is going to contribute any meaningful information to the picture. Any contribution under the value of this threshold is simply thrown out. This is the exact point in the algorithm where the "lossy" effect takes place. The first CT step is lossless except for minor mathematical precision loss. And the step following quantization is a lossless encoding step. So this is the only place where we get a chance to actually discard data.

DCT Matrix before Quantization :

92	3	-9	-7	3	-1	0	2
-39	-58	12	17	-2	2	4	2
-84	62	1	-18	3	4	-5	5
-52	-36	-10	14	-10	4	-2	0
-86	-40	49	-7	17	-6	-2	5
-62	65	-12	-2	3	-8	-2	5
-17	14	-36	17	-11	3	3	-1
-54	32	-9	-9	22	0	1	3

DCT Matrix after Dequantizaion :

90	0	-7	0	0	0	0	0
-35	-56	9	11	0	0	0	0
-84	54	0	-13	0	0	0	0
-45	-33	0	0	0	0	0	0
-77	-39	45	0	0	0	0	0
-52	60	0	0	0	0	0	0
-15	0	-19	0	0	0	0	0
-51	19	0	0	0	0	0	0

Figure above shows the effects of the quantization on a DCT matrix. The quantization/dequantization cycle has readily apparent effects. The high-frequency portions of the matrix have for the most part been truncated down to zero, eliminating their effect on the decompressed image. The coefficient in the matrix that are close to the DC coefficient may have been modified, but only by small amounts.
The interesting thing is that while we appear to be making wholesale changes to the saved image, quality factor 2 makes only minor changes that are barely noticeable. Yet the cleaning of so many of the coefficients allows the image to be compressed by 60 percent, even in the very simple compression program used in this project

Color quantization

Many people don't have full-color (24 bit per pixel) display hardware. Inexpensive display hardware stores 8 bits per pixel, so it can display at most 256 distinct colors at a time. To display a full-color image, the computer must choose an appropriate set of representative colors and map the image into these colors. This process is called "color quantization". (This is something of a misnomer; "color selection" or "color reduction" would be a better term. But we're stuck with the standard usage.)

Clearly, color quantization is a lossy process. It turns out that for most images, the details of the color quantization algorithm have *much* more impact on the final image quality than do any errors introduced by JPEG itself (except at the very lowest JPEG quality settings). Making a good color quantization method is a black art, and no single algorithm is best for all images.

Since JPEG is a full-color format, displaying a color JPEG image on 8-bit-or-less hardware requires color quantization. The speed and image quality of a JPEG viewer running on such hardware are largely determined by its quantization

algorithm. Depending on whether a quick-and-dirty or good-but-slow method is used, you'll see great variation in image quality
among viewers on 8-bit displays, much more than occurs on 24-bit displays.

On the other hand, a GIF image has already been quantized to 256 or fewer colors. (A GIF always has a specific number of colors in its palette, and the format doesn't allow more than 256 palette entries.) GIF has the advantage that the image maker precomputes the color quantization, so viewers don't have to; this is one of the things that make GIF viewers
faster than JPEG viewers. But this is also the *disadvantage* of GIF: one is stuck with the image maker's quantization. If the maker quantized to a different number of colors than what one can display, one will either waste display capability or else have to requantize to reduce the number of colors (which usually results in much poorer image quality than

quantizing once from a full-color image). Furthermore, if the maker didn't use a high-quality color quantization algorithm, you're out of luck --- the image is ruined.

For this reason, JPEG promises significantly better image quality than GIF for all users whose machines don't match the image maker's display hardware. JPEG's full color image can be quantized to precisely match the viewer's display hardware. Furthermore, one will be able to take advantage of future improvements in quantization algorithms, or purchase better display hardware, to get a better view of JPEG images one already has.

A closely related problem is seen in many current World Wide Web browsers: when running on an 8-bit display, they force all images into a pre-chosen palette. (They do this to avoid having to worry about how to allocate the limited number of available color slots among the various items on a Web page.) A GIF version of a photo usually degrades very badly in this
situation, because it's effectively being forced through a second quantization step. A JPEG photo won't look wonderful either, but it will look less bad than the GIF equivalent because it's been quantized only once.

A growing number of people have better-than-8-bit display hardware already: 15- or 16-bit/pixel "high color" displays are now quite common, and true 24-bit/pixel displays are no longer rare. For these people, GIF is already obsolete, as it cannot represent an image to the full capabilities of their display. JPEG images can drive these displays much more effectively.

In short, JPEG is an all-around better choice than GIF for representing photographic images in a machine-independent fashion.

It's sometimes thought that a JPEG converted from a GIF shouldn't require color quantization. That is false; even when you feed a 256-or-less-color GIF into JPEG, what comes out of the decompressor is not 256 colors, but thousands of colors. This happens because JPEG's lossiness affects each pixel a little differently, so two pixels that started with identical colors will usually come out with slightly different colors. Considering the whole image, each original color gets "smeared" into a cluster of nearby colors. Therefore quantization is always required to display a color JPEG on a colormapped display, regardless of the image source.

The same effect makes it nearly meaningless to talk about the number of colors used by a JPEG image. Even if you tried to count the number of distinct pixel values, different JPEG decoders would give you different results because of roundoff error differences. JPEGs can be classified as color or gray-scale, but number of colors just isn't a useful concept for JPEG, any more than it is for a real photograph.

SCOPE OF FUTURE WORK

The JPEG algorithm is applicable only to still images or pictures. This process can be extended to moving pictures or images. This algorithm can be used in case of MPEG. The algorithm will be same as that of JPEG with the subroutine being called for each frame of the moving picture. A still better approach will be to calculate the difference between the two frames and then to apply the JPEG subroutine to the difference of the two images. In the project a bitmap is being read and the values are written down in a text format or a non-formatted gray scale file on which the algorithm is applied.

JPEG is only for still images. Nonetheless, one will frequently see references to "motion JPEG" or "M-JPEG" for video. *There is no such standard*. Various vendors have applied JPEG to individual frames of a video sequence, and have called the result "M-JPEG". Unfortunately, in the absence of any recognized standard, they've each done it differently. The resulting files are usually not compatible across different vendors.

MPEG is the recognized standard for motion picture compression. It uses many of the same techniques as JPEG, but adds inter-frame compression to exploit the similarities that usually exist between successive frames. Because of this, MPEG typically compresses a video sequence by about a factor of three more than "M-JPEG" methods can for similar quality.
The disadvantages of MPEG are

- It requires far more computation to generate the compressed sequence (since detecting visual similarities is hard for a computer),
- It's difficult to edit an MPEG sequence on a frame-by-frame basis (since each frame is intimately tied to the ones around it). This latter problem has made "M-JPEG" methods rather popular for video editing products.

References

- THE DATA COMPRESSION BOOK MARK NELSON & JEAN-LOUP GAILLY
- PROGRAMMING IN ANSI C RITCHIE
- LET US C YASHWANT KANITKAR
- IMAGE PROCESSING A.GONSALEZ

1.