JPEG IMAGE COMPRESSION & EDITING IMPLEMENTED IN MATLAB

A Project Report

Submitted By

KUMAR GAURAV (REG.NO-1201210282)

SUJEET KUMAR GUPTA (REG.NO-1201210245)

OM PRAKASH SINHA (REG.NO-1201210305)

in partial fulfillment for the award of the degree of

B.Tech

IN

APPLIED ELECTRONICS AND INSTRUMENTATION

Under the esteemed guidance of

Mr. SUKANTA KUMAR TULO

Asst. PROFESSOR

APPLIED ELECTRONICS AND INSTRUMENTATION

GANDHI INSTITUTE OF ENGINEERING AND TECHNOLOGY

GUNUPUR, ODISHA- 765022

2012-2016

Gandhi Institute of

Engineering & Technology

GUNUPUR – 765 022, Dist: Rayagada (Odisha), India

*(Approved **by AICTE, Govt. of Orissa** and Affiliated **to Biju Patnaik University of Technology**)*

☎: 06857 – 250172(Office), 251156(Principal), 250232(Fax),

e-mail: gandhi_giet@yahoo.com visit us at www.giet.org

ISO 9001:2000

Certified Institute

APPLIED ELECTRONICS AND INSTRUMENTATION

CERTIFICATE

*This is to certify that the project entitled **"JPEG Image Compression & Editing Implemented in Matlab"** is the bonafide work carried out by **KUMAR GAURAV**, REG NO-**1201210282, SUJEET KUMAR GUPTA** REG NO-**1201210245, OM PRAKASH SINHA** REG NO-**1201210305** student of **B.Tech, GANDHI INSTITUTE OF ENGINEERING AND TECHNOLOGY** during the academic year 2012-2016 in partial fulfillment of the requirements for the award of the Degree of **B.Tech** in **APPLIED ELECTRONICS AND INSTRUMENTATION ENGINEERING**.*

Mr. SUKANT KUMAR TULO **Mr.SUBHRAJITPRADHAN**

Asst. PROFESSOR **HOD(AE&I)**

ACKNOWLEDGEMENT

Apart from the efforts of me, the success of this project depends largely on the encouragement and guidance of many others. I take this opportunity to my gratitude to the people who have been instrumental in The successful completion of this project. I would like to show my greatest appreciation to **Mr. Sukanta kumar tulo** from Applied Electronics and Instrumentation Department. I can't say thank you enough for their tremendous support and help. I feel motivated and encouraged every time. Without their Encouragement and Guidance this project would not have materialized. The guidance and support received from my friends who contributed and are contributing to this project, was vital for the success of the project. I am grateful for their constant support and help.

Finally, I must acknowledge with due respect the constant support and patience of our parents.

Signature of the Student:

Place:

Date:

DECLARATION

I hereby declare that the project entitled **"JPEG Image Compression & Editing Implemented in Matlab"** submitted for the B.Tech Degree is my original work and the project has not formed the basis for the award of any degree, associate ship, fellowship or any other similar titles.

Signature of the Student:

Place: Gunupur

Date:

ABSTRACT

In this project we have implemented the Baseline JPEG standard using MATLAB.We have done both the encoding and decoding of grayscale images in JPEG. With this project we have also shown the differences between the compression ratios and time spent in encoding the images with two different approaches viz-a-viz classic DCT and fast DCT. The project also shows the effect of coefficients on the image restored.

The steps in encoding starts with first dividing the original image in 8X8 blocks of sub-images. Then DCT is performed on these sub-images separately. And it is followed by dividing the resulted matrices by a Quantization Matrix. And the last step in algorithm is to make the data one-dimensional which is done by zigzag coding and compressed by Huffman coding, run level coding, or arithmetic coding.

The decoding process takes the reverse process of encoding. Firstly, the bit-stream received is converted back into two-dimensional matrices and multiplied back by Quantization Matrix. Then, the Inverse DCT is performed and the sub-images are joined together to restore the image.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

Multimedia images have become a vital and ubiquitous component of everyday life. The amount of information encoded in an image is quite large. Even with the advances in bandwidth and Storage capabilities, if images were not compressed many applications would be too costly. The following Research project attempts to answer the following questions: What are the basic principles of image compression? How do we measure how efficient a compression algorithm is? When is JPEG the best image compression algorithm? How does JPEG work? What are the alternatives to JPEG? Do they have any advantages or disadvantages? Finally, what is JPEG200?

1.1 What Is an Image?

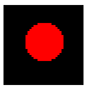
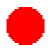
Basically, an image is a rectangular array of dots, called pixels. The size of the image is the number of pixels (width x height). Every pixel in an image is a certain color. When dealing with a black and white (where each pixel is either totally white, or totally black) image, the choices are limited since only a single bit is needed for each pixel. This type of image is good for line art, such as a cartoon in a newspaper. Another type of colorless image is a grayscale image. Grayscale images, often wrongly called "black and white" as well, use 8 bits per pixel, which is enough to represent every shade of gray that a human eye can distinguish. When dealing with color images, things get a little trickier. The number of bits per pixel is called the depth of the image (or bit plane). A bit plane of n bits can have 2^n colors. The human eye can distinguish about 2^{24} colors, although some claim that the number of colors the eye can distinguish is much higher. The most common color depths are 8, 16, and 24 (although 2-bit and 4-bit images are quite common, especially on older systems).

There are two basic ways to store color information in an image. The most direct way is to represent each pixel's color by giving an ordered triple of numbers, which is the combination of red, green, and blue that comprise that particular color. This is referred to as an RGB image. The second way to store information about color is to use a table to store the triples, and use a reference into the table for each pixel. This can markedly improve the storage requirements of an image.

1.2 Transparency

Transparency refers to the technique where certain pixels are layered on top of other pixels so that the bottom pixels will show through the top pixels. This is sometime useful in combining two images on top of each other. It is possible to use varying degrees of transparency, where the degree of transparency is known as an alpha value. In the context of the Web, this technique is often used to get an image to blend in well with the browser's background. Adding transparency can be as simple as choosing an unused color in the image to be the “special transparent” color, and wherever that color occurs, the program displaying the image knows to let the background show through.

Transparency Example:

 Non-transparent	 Transparent

1.3 File Formats

There are a large number of file formats (hundreds) used to represent an image, some more common than others. Among the most popular are:

- **GIF** (Graphics Interchange Format)
  The most common image format on the Web. Stores 1 to 8-bit color or grayscale images.
- **TIFF** (Tagged Image File Format)
  The standard image format found in most paint, imaging, and desktop publishing programs. Supports 1- to 24- bit images and several different compression schemes.
- **SGIImage**
  Silicon Graphics' native image file format. Stores data in 24-bit RGB color.

- **SunRaster**
  Sun's native image file format; produced by many programs that run on Sun workstations.
- **PICT**
  Macintosh's native image file format; produced by many programs that run on Macs. Stores up to 24-bit color.
- **BMP**(Microsoft Windows Bitmap)
  Main format supported by Microsoft Windows. Stores 1-, 4-, 8-, and 24-bit images.
- **XBM**(X Bitmap)
  A format for monochrome (1-bit) images common in the X Windows system.
- **JPEG File Interchange Format**
  Developed by the Joint Photographic Experts Group, sometimes simply called the JPEG file format. It can store up to 24-bits of color. Some Web browsers can display JPEG images inline (in particular, Netscape can), but this feature is not a part of the HTML standard.

  The following features are common to most bitmap files:

- Header: Found at the beginning of the file, and containing information such as the image's size, number of colors, the compression scheme used, etc.
- Color Table: If applicable, this is usually found in the header.
- Pixel Data: The actual data values in the image.
- Footer: Not all formats include a footer, which is used to signal the end of the data.

1.4 Bandwidth and Transmission

In our high stress, high productivity society, efficiency is key. Most people do not have the time or patience to wait for extended periods of time while an image is downloaded or retrieved. In fact, it has been shown that the average person will only wait 20 seconds for an image to appear on a web page. Given the fact that the average Internet user still has a 28k or 56k modem, it is essential to keep image sizes under control. Without some type of compression, most images would be too cumbersome and impractical for use. The following table is used to show the correlation between modem speeds and download time. Note that even high speed Internet users require over one second to download the image.

Modem Speed	**Throughput – How Much Data Per Second**	**Download Time For a 40k Image**
14.4k	1kB	40 seconds
28.8k	2kB	20 seconds
33.6k	3kB	13.5 seconds
56k	5kB	8 seconds
256k DSL	32kB	1.25 seconds
1.5M T1	197kB	0.2 seconds

Table1: Download Time Comparison

CHAPTER 2. AN INTRODUCTION TO IMAGE COMPRESSION

This is done by removing all redundant or unnecessary information. An uncompressed image requires an enormous amount of data to represent it. As an example, a standard 8.5" by 11" sheet of paper scanned at 100 dpi and restricted to black and white requires more than 100k bytes to represent. Another example is the 276-pixel by 110-pixel banner Image compression is the process of reducing the amount of data required to represent a digital image that appears at the top of Google.com. Uncompressed, it requires 728k of space. Image compression is thus essential for the efficient storage, retrieval and transmission of images. In general, there are two main categories of compression. Lossless compression involves the preservation of the image as is (with no information and thus no detail lost). Lossy compression on the other hand, allows less than perfect reproductions of the original image. The advantage being that, with a lossy algorithm, one can achieve higher levels of compression because less information is needed. Various amounts of data may be used to represent the same amount of information. Some representations may be less efficient than others, depending on the amount of redundancy eliminated from the data. When talking about images there are three main sources of redundant information:

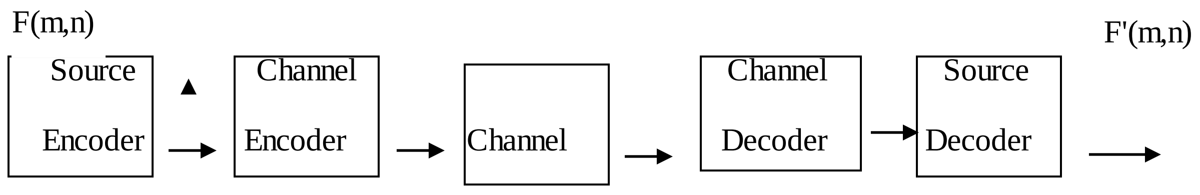
- Coding Redundancy- This refers to the binary code used to represent grey values.
- Inter pixel Redundancy- This refers to the correlation between adjacent pixels in an image.
- Psycho visual Redundancy - This refers to the unequal sensitivity of the human eye to different visual information.

In comparing how much compression one algorithm achieves verses another, many people talk about a compression ratio. A higher compression ratio indicates that one algorithm removes more redundancy then another (and thus is more efficient). If n_1 and n_2 are the number of bits in two datasets that represent the same image, the relative redundancy of the first dataset is defined as:

$R_d=1/C_R$	where C_R (the compression ratio) $=n_1/n_2$

The benefits of compression are immense. If an image is compressed at a ratio of 100:1, it may be transmitted in one hundredth of the time, or transmitted at the same speed through a channel of one-hundredth the bandwidth (ignoring the compression/decompression overhead). Since images have become so commonplace and so essential to the function of computers, it is hard to see how we would function without them.

2.1 The Image Compression Model



Although image compression models differ in the way they compress data, there are many general features that can be described which represent most image compression algorithms. The source encoder is used to remove redundancy in the input image. The channel encoder is used as overhead in order to combat channel noise. A common example of this would be the introduction of a parity bit. By introducing this overhead, a certain level of immunity is gained from noise that is inherent in any storage or transmission system. The channel in this model could be either a communication link or a storage/retrieval system. The job of the channel and source decoders is to basically undo the work of the source and channel encoders in order to restore the image to the user.

2.2 Fidelity Criterion

A measure is needed in order to measure the amount of data lost (if any) due to a compression scheme. This measure is called a fidelity criterion. There are two main categories of fidelity criterion: subjective and objective. Objective fidelity criterion, involve a quantitative approach to error criterion. Perhaps the most common example of this is the root mean square error. A very much related measure is the mean square signal to noise ratio. Although objective field criteria may be useful in analyzing the amount of error involved in a compression scheme, our eyes do not always see things as they are. Which is why the second category of fidelity criterion is important. Subjective field criteria are quality evaluations based on a human observer. These ratings are often averaged to come up with an evaluation of a compression scheme. There are absolute comparison scales, which are based solely on the decompressed image, and there are relative comparison scales that involve viewing the original and decompressed images side by side in comparison. Examples of both scales are provided, for interest.

Value	**Rating**	**Description**
1	Excelle nt	An image of extremely high quality. As good as desired.
2	Fine	An image of high quality, providing enjoyable viewing.
3	Passable	An image of acceptable quality.
4	Margina l	An image of poor quality; one wishes to improve it.
5	Inferior	A very poor image, but one can see it.
6	Unusabl e	An image so bad, one can't see it.

Table 2 : Absolute Comparizon Scale

VALUE	**-3**	**-2**	**-1**	**0**	**1**	**2**	**3**
Rating	Much Worse	Worse	Slightly Worse	Same	Slightly Better	Better	Much Better

Table 3: Relative Comparison Scale

An obvious problem that arises is that subjective fidelity criterion may vary from person to person. What one person sees a marginal, another may view as passable, etc.

2.3 Information Theory

In the 1940's Claude E. Shannon pioneered a field that is now the theoretical basis for most data compression techniques. Information theory is useful in answering questions such as what is the minimum amount of data needed to represent an image without loss of information. ? Or, theoretically what is the best compression possible?

The basic premise is that the generation of information may be viewed as a probabilistic process. The input (or source) is viewed to generate one of N possible symbols from the source alphabet set A={a ,b , c,…, z), {0, 1}, {0, 2, 4…, 280}, etc. in unit time. The source output can be denoted as a discrete random variable E, which is a symbol from the alphabet source along with a corresponding probability (z). When an algorithm scans the input for an occurrence of E, the result is a gain in information denoted by I(E), and quantified as:

$$I(E) = \log(1/ P(E))$$

This relation indicated that the amount of information attributed to an event is inversely related to the probability of that event. As an example, a certain event (P(E) = 1) leads to an I(E) = 0. This makes sense, since as we know that the event is certain, observing its occurrence adds nothing to our information. On the other hand, when a highly uncertain event occurs, a significant gain of information is the result.

An important concept called the entropy of a source (H(z)), is defined as the average amount of information gained by observing a particular source symbol. Basically, this allows an algorithm to quantize the randomness of a source. The amount of randomness is quite important because the more random a source is (the more unlikely it is to occur) the more information that is needed to represent it. It turns out that for a fixed number of source symbols, efficiency is maximized when all the symbols are equally likely. It is based on this principle that codewords are assigned to represent information. There are many different schemes of assigning codewords, the most common being the Huffman coding, run length encoding, and LZW.

2.4 Compression Summary

Image compression is achieved by removing (or reducing) redundant information. In order to effectively do this, patterns in the data must be identified and utilized. The theoretical basis for this is founded in Information theory, which assigns probabilities to the likelihood of the occurrence of each symbol in the input. Symbols with a high probability of occurring are represented with shorter bit strings (or codewords). Conversely, symbols with a low probability of occurring are represented with longer codewords. In this way, the average length of codewords is decreased, and redundancy is reduced. How efficient an algorithm can be, depends in part on how the probability of the symbols is distributed, with maximum efficiency occurring when the distribution is equal over all input symbols.

CHAPTER 3 A LOOK AT SOME JPEG ALTERNATIVES

Before examining the JPEG compression algorithm, the report will now proceed to examine some of the widely available alternatives. Each algorithm will be examined separately, with a comparison at the end. The best algorithms to study for our purposes are GIF, PNG, and TIFF.

3.1 GIF Compression

The GIF (Graphics Interchange Format) was created in 1987 by Compuserve. It was revised in 1989. GIF uses a compression algorithm called "LZW," written by Abraham Lempel, Jacob Ziv, and Terry Welch. Unisys patented the algorithm in 1985, and in 1995 the company made the controversial move of asking developers to pay for the previously free LZW license. This led to the creation of GIF alternatives such as PNG (which is discussed later). However, since GIF is one of the oldest image file formats on the Web, it is very much embedded into the landscape of the Internet, and it is here to stay for the near future. The LZW compression algorithm is an example of a lossless algorithm. The GIF format is well known to be good for graphics that contain text, computer-generated art, and/or large areas of solid color (a scenario that does not occur very often in photographs or other real life images). GIF's main limitation lies in the fact that it only supports a maximum of 256 colors. It has a running time of $O(m^2)$, where m is the number of colors between 2 and 256.

The first step in GIF compression is to "index" the image's color palette. This decreases the number of colors in your image to a maximum of 256 (8-bit color). The smaller the number of colors in the palette, the greater the efficiency of the algorithm. Many times, an image that is of high quality in 256 colors can be reproduced effectively with 128 or fewer colors.

LZW compression works best with images that have horizontal bands of solid color. So if you have eight pixels across a one-pixel row with the same color value (white, for example), the LZW compression algorithm would see that as "8W" rather than "WWWWWWWW," which saves file space.

Sometimes an indexed color image looks better after dithering, which is the process of mixing existing colors to approximate colors that were previously eliminated. However, dithering leads to an increased file size because it reduces the amount of horizontal repetition in an image.

Another factor that affects GIF file size is interlacing. If an image is interlaced, it will display itself all at once, incrementally bringing in the details (just like progressive JPEG), as opposed to the consecutive option, which will display itself row by row from top to bottom. Interlacing can increase file size slightly, but is beneficial to users who have slow connections because they get to see more of the image more quickly.

3.2 PNG Compression

The PNG (Portable Network Graphic) image format was created in 1995 by the PNG Development Group as an alternative to GIF (the use of GIF was protested after the Unisys decision to start charging for use of the LZW compression algorithm). The PNG (pronounced "ping") file format uses the LZ77 compression algorithm instead, which was created in 1977 by Lemper and Ziv (without Welch), and revised in 1978.

PNG is an open (free for developers) format that has a better average compression than GIF and a number of interesting features including alpha transparency (so you may use the same image on many different-colored backgrounds). It also supports 24-bit images, so you don't have to index the colors like GIF. PNG is a lossless algorithm, which is used under many of the same constraints as GIF. It has a running time of $O(m^2 \log m)$, where m is again the number of colors in the image.

Like all compression algorithms, LZ77 compression takes advantage of repeating data, replacing repetitions with references to previous occurrences. Since some images do not compress well with the LZ77 algorithm alone, PNG offers filtering options to rearrange pixel data before compression. These filters take advantage of the fact that neighboring pixels are often similar in value. Filtering does not compress data in any way; it just makes the data more suitable for compression.

As an example, of how PNG filters work, imagine an image that is 8 pixels wide with the following color values: 3, 13, 23, 33, 43, 53, 63, and 73. There is no redundant information here, since all the values are unique, so LZ77 compression won't work very well on this particular row of pixels. When the "Sub" filter is used to calculate the difference between the pixels (which is 10) then the data that is observed becomes: 3, 10, 10, 10, 10, 10, 10, 10 (or 3, 7*10). The LZ77 compression algorithm then takes advantage of the newly created redundancy as it stores the image.

Another filter is called the “Up” filter. It is similar to the Sub filter, but tries to find repetitions of data in vertical pixel rows, rather than horizontal pixel rows.

The Average filter replaces a pixel with the difference between it and the average of the pixel to the left and the pixel above it.

The Paeth (pronounced peyth) filter, created by Alan W. Paeth, works by replacing each pixel with the difference between it and a special function of the pixel to the left, the pixel above and the pixel to the upper left.

The Adaptive filter automatically applies the best filter(s) to the image. PNG allows different filters to be used for different horizontal rows of pixels in the same image. This is the safest bet, when choosing a filter in unknown circumstances.

PNG also has a no filter, or "None" option, which is useful when working with indexed color or bitmap mode images.

A final factor that may influence PNG file size is interlacing, which is identical to the interlacing described for GIF.

3.3 TIFF Compression

TIFF (Tagged Interchange File Format), developed in 1995, is a widely supported, highly versatile format for storing and sharing images. It is utilized in many fax applications and is widespread as a scanning output format.

The designers of the TIFF file format had three important goals in mind:

a. Extendibility. This is the ability to add new image types without affecting the functionality of previous types.

b. Portability. TIFF was designed to be independent of the hardware platform and the operating system on which it executes. TIFF makes very few demands upon its operating environment. TIFF should (and does) perform equally well in a wide variety of computing platforms such as PC, MAC, and UNIX.

c. Revisability. TIFF was designed not only to be an efficient medium for exchanging image information but also to be usable as a native internal data format for image editing applications.

The compression algorithms supported by TIFF are plentiful and include run length encoding, Huffman encoding and LZW. Indeed, TIFF is one of the most versatile compression formats. Depending on the compression used, this algorithm may be either lossy or lossless. Another effect is that its running time is variable depending on which compression algorithm is chosen.

Some limitations of TIFF are that there are no provisions for storing vector graphics, text annotation, etc (although such items could be easily constructed using TIFF extensions). Perhaps TIFF’s biggest downfall is caused by its flexibility. An example of this is that TIFF format permits both MSB ("Motorola") and LSB ("Intel") byte order data to be stored, with a header item indicating which order is used. Keeping track of what is being used when can get quite entertaining, but may lead to error prone code.

TIFF’s biggest advantage lies primarily in its highly flexible and platform-independent format, which is supported by numerous image-processing applications. Since it was designed by developers of printers, scanners, and monitors it has a very rich space of information elements for colorimetry calibration, gamut tables, etc. Such information is also very useful for remote sensing and multispectral applications. Another feature of TIFF that is also useful is the ability to decompose an image by tiles rather than scanlines.

CHAPTER 4. Comparison of Image Compression Techniques

Although various algorithms have been described so far, it is difficult to get a sense of how each one compares to the other in terms of quality, efficiency, and practicality. Creating the absolute smallest image requires that the user understand the differences between images and the differences between compression methods. Knowing when to apply what algorithm is essential. The following is a comparison of how each performs in a real world situation.

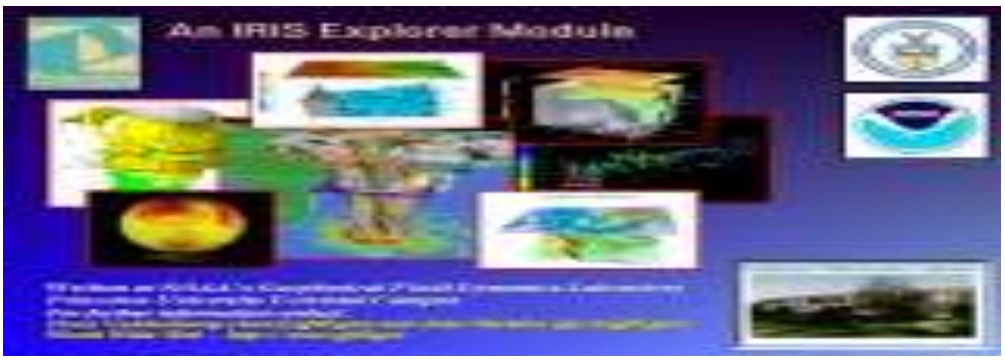


Figure 1: Example Image

The following screen shot was compressed and reproduced by all the three compression algorithms. The results are summarized in the following table.

	File size in bytes	
Raw 24-bit	9216	

	00	
GIF (LZW)	118937	
TIFF (LZW)	462124	
PNG (24-bit)	248269	
PNG (8-bit)	99584	

Table 4: Image Compression Comparison

In this case, the 8-bit PNG compression algorithm produced the file with the smallest size (and thus greater compression). Does this mean that PNG is always the best option for any screen shot? The answer is a resounding NO! Although there are no hard and fast rules for what is the best algorithm for what situation, there are some basic guidelines to follow. A summary of findings of this report may be found in the following table.

	TIFF	GIF	PNG
Bits/pixel (max. color depth)	24-bit	8-bit	48-bit
Transparency		●	●
Interlace method		●	●
Compression of the image	●	●	●
Photographs	●		●
Line art, drawings and images with large solid color areas	●	●	

Table 5: Summary of GIF, PNG, and TIFF

CHAPTER 5. THE JPEG ALGORITHM

The Joint Photographic Experts Group developed the JPEG algorithm in the late 1980's and early 1990's. They developed this new algorithm to address the problems of that era, specifically the fact that consumer-level computers had enough processing power to manipulate and display full color photographs. However, full color photographs required a tremendous amount of bandwidth when transferred over a network connection, and required just as much space to store a local copy of the image. Other compression techniques had major tradeoffs. They had either very low amounts of compression, or major data loss in the image. Thus, the JPEG algorithm was created to compress photographs with minimal data loss and high compression ratios.

Due to the nature of the compression algorithm, JPEG is excellent at compressing full-color (24-bit) photographs, or compressing grayscale photos that include many different shades of gray. The JPEG algorithm does not work well with web graphics, line art, scanned text, or other images with sharp transitions at the edges of objects. The reason this is so will become clear in the following sections. JPEG also features an adjustable compression ratio that lets a user determine the quality and size of the final image. Images may be highly compressed with lesser quality, or they may forego high compression, and instead be almost indistinguishable from the original.

JPEG compression and decompression consist of 4 distinct and independent phases. First, the image is divided into 8 x 8 pixel blocks. Next, a discrete cosine transform is applied to each block to convert the information from the spatial domain to the frequency domain. After that, the frequency information is quantized to remove unnecessary information. Finally, standard compression techniques compress the final bit stream. This report will analyze the compression of a grayscale image, and will then extend the analysis to decompression and to color images.

5.1 Phase One: Divide the Image

Attempting to compress an entire image would not yield optimal results. Therefore, JPEG divides the image into matrices of 8 x 8 pixel blocks. This allows the algorithm to take advantage of the fact that similar colors tend to appear together in small parts of an image. Blocks begin at the upper left part of the image, and are created going towards the lower right. If the image dimensions are not multiples of 8, extra pixels are added to the bottom and right part of the image to pad it to the next multiple of 8 so that we create only full blocks. The dummy values are easily removed during decompression. From this point on, each block of 64 pixels is processed separately from the others, except during a small part of the final compression step.

Phase one may optionally include a change in colorspace. Normally, 8 bits are used to represent one pixel. Each byte in a grayscale image may have the value of 0 (fully black) through 255 (fully white). Color images have 3 bytes per pixel, one for each component of red, green, and blue (RGB color). However, some operations are less complex if you convert these RGB values to a different color representation. Normally, JPEG will convert RGB color space to YCbCr color space. In YCbCr, Y is the luminance, which represents the intensity of the color. Cb and Cr are chrominance values, and they actually describe the color itself. YCbCr tends to compress more tightly than RGB, and any color space conversion can be done in linear time. The color space conversion may be done before we break the image into blocks; it is up to the implementation of the algorithm.

Finally, the algorithm subtracts 128 from each byte in the 64-byte block. This changes the scale of the byte values from 0…255 to –128…127. Thus, the average value over a large set of pixels will tend towards zero.

The following images show an example image, and that image divided into an 8 x 8 matrix of pixel blocks. The images are shown at double their original sizes, since blocks are only 8 pixels wide, which is extremely difficult to see. The image is 200 pixels by 220 pixels, which means that the image will be separated into 700 blocks, with some padding added to the bottom of the image. Also, remember that the division of an image is only a logical division, but in figure 1 lines are used to add clarity.

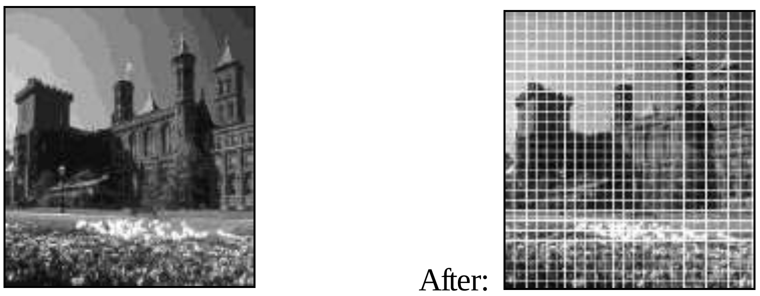
Before:  After:

Figure 2: Example of Image Division

5.2 Phase Two: Conversion to the Frequency Domain

At this point, it is possible to skip directly to the quantization step. However, we can greatly assist that stage by converting the pixel information from the spatial domain to the frequency domain. The conversion will make it easier for the quantization process to know which parts of the image are least important, and it will de-emphasize those areas in order to save space.

Currently, each value in the block represents the intensity of one pixel (remember, our example is a grayscale image). After converting the block to the frequency domain, each value will be the amplitude of a unique cosine function. The cosine functions each have different frequencies. We can represent the block by multiplying the functions with their corresponding amplitudes, then adding the results together. However, we keep the functions separate during JPEG compression so that we may remove the information that makes the smallest contribution to the image.

Human vision has a drop-off at higher frequencies, and de-emphasizing (or even removing completely) higher frequency data from an image will give an image that appears very different to a computer, but looks very close to the original to a human. The quantization stage uses this fact to remove high frequency information, which results in a smaller representation of the image.

There are many algorithms that convert spatial information to the frequency domain. The most obvious of which is the Fast Fourier Transform (FFT). However, due to the fact that image information does not contain any imaginary components, there is an algorithm that is even faster than an FFT. The Discrete Cosine Transform (DCT) is derived from the FFT, however it requires fewer multiplications than the FFT since it works only with real numbers. Also, the DCT produces fewer significant coefficients in its result, which leads to greater compression. Finally, the DCT is made to work on one-dimensional data. Image data is given in blocks of two-dimensions, but we may add another summing term to the DCT to make the equation two-dimensional. In other words, applying the one-dimensional DCT once in the x direction and once in the y direction will effectively give a two-dimensional discrete cosine transform.

The 2D discrete cosine transform equation is given in figure 2, where C(x) = 1/√2 if x is 0, and C(x) = 1 for all other cases. Also, f (x, y) is the 8-bit image value at coordinates (x, y), and F (u, v) is the new entry in the frequency matrix.

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

DCT Equation

We begin examining this formula by realizing that only constants come before the brackets. Next, we realize that only 16 different cosine terms will be needed for each different pair of (u, v) values, so we may compute these ahead of time and then multiply the correct pair of cosine terms to the spatial-domain value for that pixel. There will be 64 additions in the two summations, one per pixel. Finally, we multiply the sum by the 3 constants to get the final value in the frequency matrix. This continues for all (u, v) pairs in the frequency matrix. Since u and v may be any value from 0…7, the frequency domain matrix is just as large as the spatial domain matrix.

The frequency domain matrix contains values from -1024…1023. The upper-left entry, also known as the DC value, is the average of the entire block, and is the lowest frequency cosine coefficient. As you move right the coefficients represent cosine functions in the vertical direction that increase in frequency. Likewise, as you move down, the coefficients belong to increasing frequency cosine functions in the horizontal direction. The highest frequency values occur at the lower-right part of the matrix. The higher frequency values also have a natural tendency to be significantly smaller than the low frequency coefficients since they contribute much less to the image. Typically the entire lower-right half of the matrix is factored out after quantization. This essentially removes half of the data per block, which is one reason why JPEG is so efficient at compression.

Computing the DCT is the most time-consuming part of JPEG compression. Thus, it determines the worst-case running time of the algorithm. The running time of the algorithm is discussed in detail later. However, there are many different implementations of the discrete cosine transform. Finding the most efficient one for the programmer's situation is key. There are implementations that can replace all multiplications with shift instructions and additions. Doing so can give dramatic speedups, however it often approximates values, and thus leads to a lower quality output image. There are also debates on how accurately certain DCT algorithms compute the cosine coefficients, and whether or not the resulting values have adequate precision for their situations. So any programmer should use caution when choosing an algorithm for computing a DCT, and should be aware of every trade-off that the algorithm has.

5.3 Phase Three: Quantization

Having the data in the frequency domain allows the algorithm to discard the least significant parts of the image. The JPEG algorithm does this by dividing each cosine coefficient in the data matrix by some predetermined constant, and then rounding up or down to the closest integer value. The constant values that are used in the division may be arbitrary, although research has determined some very good typical values. However, since the algorithm may use any values it wishes, and since this is the step that introduces the most loss in the image, it is a good place to allow users to specify their desires for quality versus size.

Obviously, dividing by a high constant value can introduce more error in the rounding process, but high constant values have another effect. As the constant gets larger the result of the division approaches zero. This is especially true for the high frequency coefficients, since they tend to be the smallest values in the matrix. Thus, many of the frequency values become zero. Phase four takes advantage of this fact to further compress the data.

The algorithm uses the specified final image quality level to determine the constant values that are used to divide the frequencies. A constant of 1 signifies no loss. On the other hand, a constant of 255 is the maximum amount of loss for that coefficient. The constants are calculated according to the user's wishes and the heuristic values that are known to result in the best quality final images. The constants are then entered into another 8 x 8 matrix, called the quantization matrix. Each entry in the quantization matrix corresponds to exactly one entry in the frequency matrix. Correspondence is determined simply by coordinates, the entry at (3, 5) in the quantization matrix corresponds to entry (3, 5) in the frequency matrix.

A typical quantization matrix will be symmetrical about the diagonal, and will have lower values in the upper left and higher values in the lower right. Since any arbitrary values could be used during quantization, the entire quantization matrix is stored in the final JPEG file so that the decompression routine will know the values that were used to divide each coefficient.

Table 3 shows an example of a quantization matrix.

1	2	4	8	12	16	22	24
2	4	8	12	16	22	24	27
4	8	12	16	22	24	27	29
8	12	16	22	24	27	29	32
12	16	22	24	27	29	32	35
16	22	24	27	29	32	35	38
22	24	27	29	32	35	38	44
24	27	29	32	35	38	44	50

Table 3: Sample Quantization Matrix

The equation used to calculate the quantized frequency matrix is fairly simple. The algorithm takes a value from the frequency matrix (F) and divides it by its corresponding value in the quantization matrix (Q). This gives the final value for the location in the quantized frequency matrix ($F_{quantize}$). Figure 4 shows the quantization equation that is used for each block in the image.

$$F_{Quantize}(u,v) = \left(\frac{F(u,v)}{Q(u,v)} \right) + 0.5$$

Quantization Equation

By adding 0.5 to each value, we essentially round it off automatically when we truncate it, without performing any comparisons. Of course, any means of rounding will work.

5.4 Phase Four: Entropy Coding

After quantization, the algorithm is left with blocks of 64 values, many of which are zero. Of course, the best way to compress this type of data would be to collect all the zero values together, which is exactly what JPEG does. The algorithm uses a zigzag ordered encoding, which collects the high frequency quantized values into long strings of zeros.

To perform a zigzag encoding on a block, the algorithm starts at the DC value and begins winding its way down the matrix, as shown in figure 5. This converts an 8 x 8 table into a 1 x 64 vector.

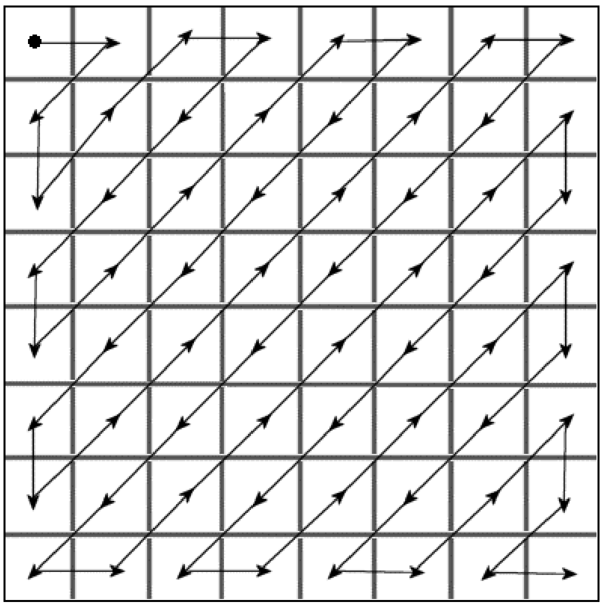


Figure 3: Zigzag Ordered Encoding

All of the values in each block are encoded in this zigzag order except for the DC value. For all of the other values, there are two tokens that are used to represent the values in the final file. The first token is a combination of {size, skip} values. The size value is the number of bits needed to represent the second token, while the skip value is the number of zeros that precede this token. The second token is simply the quantized frequency value, with no special encoding. At the end of each block, the algorithm places an end-of-block sentinel so that the decoder can tell where one block ends and the next begins.

The first token, with {size, skip} information, is encoded using Huffman coding. Huffman coding scans the data being written and assigns fewer bits to frequently occurring data, and more bits to infrequently occurring data. Thus, if a certain values of size and skip happen often, they may be represented with only a couple of bits each. There will then be a lookup table that converts the two bits to their entire value. JPEG allows the algorithm to use a standard Huffman table, and also allows for custom tables by providing a field in the file that will hold the Huffman table.

DC values use delta encoding, which means that each DC value is compared to the previous value, in zigzag order. Note that comparing DC values is done on a block by block basis, and does not consider any other data within a block. This is the only instance where blocks are not treated independently from each other. The difference between the current DC value and the previous value is all that is included in the file. When storing the DC values, JPEG includes a size field and then the actual DC delta value. So if the difference between two adjacent DC values is –4, JPEG will store the size 3, since -4 requires 3 bits. Then, the actual binary value 100 is stored. The size field for DC values is included in the Huffman coding for the other size values, so that JPEG can achieve even higher compression of the data.

5.5 Other JPEG Information

There are other facts about JPEG that are not covered in the compression of a grayscale image. The following sections describe other parts of the JPEG algorithm, such as decompression, progressive JPEG encoding, and the algorithm's running time.

5.5.1 Color Images

Color images are usually encoded in RGB colorspace, where each pixel has an 8-bit value for each of the three composite colors. Thus, a color image is three times as large as a grayscale image, and each of the components of a color image can be considered its own grayscale representation of that particular color.

In fact, JPEG treats a color image as 3 separate grayscale images, and compresses each component in the same way it compresses a grayscale image. However, most color JPEG files are not three times larger than a grayscale image, since there is usually one color component that does not occur as often as the others, in which case it will be highly compressed. Also, the Huffman coding steps will have the opportunity to compress more values, since there are more possible values to compress.

5.5.2 Decompression

Decompressing a JPEG image is basically the same as performing the compression steps in reverse, and in the opposite order. It begins by retrieving the Huffman tables from the image and decompressing the Huffman tokens in the image. Next, it decompresses the DCT values for each block, since they will be the first things needed to decompress a block. JPEG then decompresses the other 63 values in each block, filling in the appropriate number of zeros where appropriate. The last step in reversing phase four is decoding the zigzag order and recreate the 8 x 8 blocks that were originally used to compress the image.

To undo phase three, the quantization table is read from the JPEG file and each entry in every block is then multiplied by its corresponding quantization value.

Phase two was the discrete cosine transformation of the image, where we converted the data from the spatial domain to the frequency domain. Thus, we must do the opposite here, and convert frequency values back to spatial values. This is easily accomplished by an inverse discrete cosine transform. The IDCT takes each value in the spatial domain and examines the contributions that each of the 64 frequency values make to that pixel.

In many cases, decompressing a JPEG image must be done more quickly than compressing the original image. Typically, an image is compressed once, and viewed many times. Since the IDCT is the slowest part of the decompression, choosing an implementation for the IDCT function is very important. The same quality versus speed tradeoff that the DCT algorithm has applies here. Faster implementations incur some quality loss in the image, and it is up to the programmer to decide which implementation is appropriate for the particular situation. Figure 6 shows the equation for the inverse discrete cosine transform function.

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

Inverse DCT Equation

Finally, the algorithm undoes phase one. If the image uses a colorspace that is different from RGB, it is converted back during this step. Also, 128 is added to each pixel value to return the pixels to the unsigned range of 8-bit numbers. Next, any padding values that were added to the bottom or to the right of the image are removed. Finally, the blocks of 8 x 8 pixels are recombined to form the final image.

5.5.4 Compression Ratio

The compression ratio is equal to the size of the original image divided by the size of the compressed image. This ratio gives an indication of how much compression is achieved for a particular image. Most algorithms have a typical range of compression ratios that they can achieve over a variety of images. Because of this, it is usually more useful to look at an average compression ratio for a particular method.

The compression ratio typically affects the picture quality. Generally, the higher the compression ratio, the poorer the quality of the resulting image. The tradeoff between compression ratio and picture quality is an important one to consider when compressing images.

5.5.4 Sources of Loss in an Image

JPEG is a lossy algorithm. Compressing an image with this algorithm will almost guarantee that the decompressed version of the image will not match the original source image. Loss of information happens in phases two and three of the algorithm.

In phase two, the discrete cosine transformation introduces some error into the image, however this error is very slight. The error is due to imprecision in multiplication, rounding, and significant error is possible if the DCT implementation chosen by the programmer is designed to trade off quality for speed. Any errors introduced in this phase can affect any values in the image with equal probability. It does not limit its error to any particular section of the image.

Phase three, on the other hand, is designed to eliminate data that does not contribute much to the image. In fact, most of the loss in JPEG compression occurs during this phase. Quantization divides each frequency value by a constant, and rounds the result. Therefore, higher constants cause higher amounts of loss in the frequency matrix, since the rounding error will be higher. As stated before, the algorithm is designed in this way, since the higher constants are concentrated around the highest frequencies, and human vision is not very sensitive to those frequencies. Also, the quantization matrix is adjustable, so a user may adjust the amount of error introduced into the compressed image. Obviously, as the algorithm becomes less lossy, the image size increases. Applications that allow the creation of JPEG images usually allow a user to specify some value between 1 and 100, where 100 is the least lossy. By most standards, anything over 90 or 95 does not make the picture any better to the human eye, but it does increase the file size dramatically. Alternatively, very low values will create extremely small files, but the files will have a blocky effect. In fact, some graphics artists use JPEG at very low quality settings (under 5) to create stylized effects in their photos.

5.5.5 Progressive JPEG Images

A newer version of JPEG allows images to be encoded as progressive JPEG images. A progressive image, when downloaded, will show the major features of the image very quickly, and will then slowly become clearer as the rest of the image is received. Normally, an image is displayed at full clarity, and is shown from top to bottom as it is received and decoded. Progressive JPEG files are useful for slow connections, since a user can get a good idea what the picture will be well before it finishes downloading. Note that progressive JPEG is simply a rearrangement of data onto a more complicated order, and does not actually change any major aspects of the JPEG format. Also, a progressive JPEG file will be the same size as a standard JPEG file. Finally, displaying progressive JPEG images is more computationally intense than displaying a standard JPEG, since some extra processing is needed to make the image fade into view.

There are two main ways to implement a progressive JPEG. The first, and easiest, is to simply display the DC values as they are received. The DC values, being the average value of the 8 x 8 block, are used to represent the entire block. Thus, the progressive image will appear as a blocky image while the other values are received, but since the blocks are so small, a fairly adequate representation of the image will be shown using just the DC values. The alternative method is to begin by displaying just the DC information, as detailed above. But then, as the data is received, it will begin to add some higher frequency values into the image. This makes the image appear to gain sharpness until the final image is displayed. To implement this, JPEG first encodes the image so that certain lower frequencies will be received very quickly. The lower frequency values are displayed as they are received, and as more bits of each frequency value are received they are shifted into place and the image is updated.

5.5.6 Running Time

The running time of the JPEG algorithm is dependent on the implementation of the discrete cosine transformation step, since that step runs more slowly than any other step. In fact, all other steps run in linear time. Implementing the DCT equation directly will result in a running time that is $O(n^3)$ to process all image blocks. This is slower than using a FFT directly, which we avoided due to its use of imaginary components. However, by optimising the implementation of the DCT, one can easily achieve a running time that is $O(n^2 \log(n))$, or possibly better. Even faster algorithms for computing the DCT exist, but they sacrafice quality for speed. In some applications, such as embedded systems, this may be a valid trade-off.

CHAPTER 6. VARIANTS OF THE JPEG ALGORITHM

Quite a few algorithms are based on JPEG. They were created for more specific purposes than the more general JPEG algorithm. This section will discuss variations on JPEG. Also, since the output stream from the JPEG algorithm must be saved to disk, we discuss the most common JPEG file format.

6.1 JFIF (JPEG file interchange format)

JPEG is a compression algorithm, and does not define a specific file format for storing the final data values. In order for a program to function properly there has to be a compatible file format to store and retrieve the data. JFIF has emerged as the most popular JPEG file format. JFIF's ease of use and simple format that only transports pixels was quickly adopted by Internet browsers. JFIF is now the industry standard file format for JPEG images. Though there are better image file formats currently available and upcoming, it is questionable how successful these will be given how ingrained JFIF is in the marketplace.

JFIF image orientation is top-down. This means that the encoding proceeds from left to right and top to bottom. Spatial relationship of components such as the position of pixels is defined with respect to the highest resolution component. Components are sampled along rows and columns so a subsampled component position can be determined by the horizontal and vertical offset from the upper left corner with respect to the highest resolution component.

The horizontal and vertical offsets of the first sample in a subsampled component, Xoffset i [0,0] and Yoffset i [0,0], is defined to be Xoffset i [0,0] = (Nsamples ref / Nsamples i) / 2 - 0.5 Yoffset i [0,0] = (Nlines ref / Nlines i) / 2 - 0.5 where Nsamples ref is the number of samples per line in the largest component, Nsamples i is the number of samples per line in the ith component, Nlines ref is the number of lines in the largest component, Nlines i is the number of lines in the ith component.

As an example, consider a 3 component image that is comprised of components having the following dimensions:

Component 1: 256 samples, 288 lines

Component 2: 128 samples, 144 lines

Component 3: 64 samples, 96 lines

In a JFIF file, centers of the samples are positioned as illustrated below:

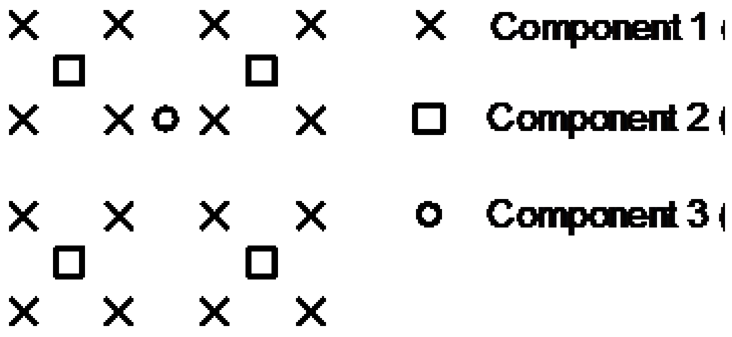


Figure 7: Example of JFIF Samples

6.2 JBIG Compression

JBIG stands for Joint Bi-level Image Experts Group. JBIG is a method for lossless compression of bi-level (two-color) image data. All bits in the images before and after compression and decompression will be exactly the same.

JBIG also supports both sequential and progressive encoding methods. Sequential encoding reads data from the top to bottom and from left to right of an image and encodes it as a single image. Progressive encoding allows a series of multiple-resolution versions of the same image data to be stored within a single JBIG data stream.

JBIG is platform-independent and implements easily over a wide variety of distributed environments. However, a disadvantage to JBIG that will probably cause it to fail is the twenty-four patented processes that keep JBIG from being freely distributed. The most prominent is the IBM arithmetic Q-coder, which is an option in JPEG, but is mandatory in JBIG.

JBIG encodes redundant image data by comparing a pixel in a scan line with a set of pixels already scanned by the encoder. These additional pixels are called a template, and they form a simple map of the pattern of pixels that surround the pixel that is being encoded. The values of these pixels are used to identify redundant patterns in the image data. These patterns are then compressed using an adaptive arithmetic compression coder.

JBIG is capable of compressing color or grayscale images up to 255 bits per pixel. This can be used as an alternative to lossless JPEG. JBIG has been found to produce better to equal compression results then lossless JPEG on data with pixels up to eight bits in depth.

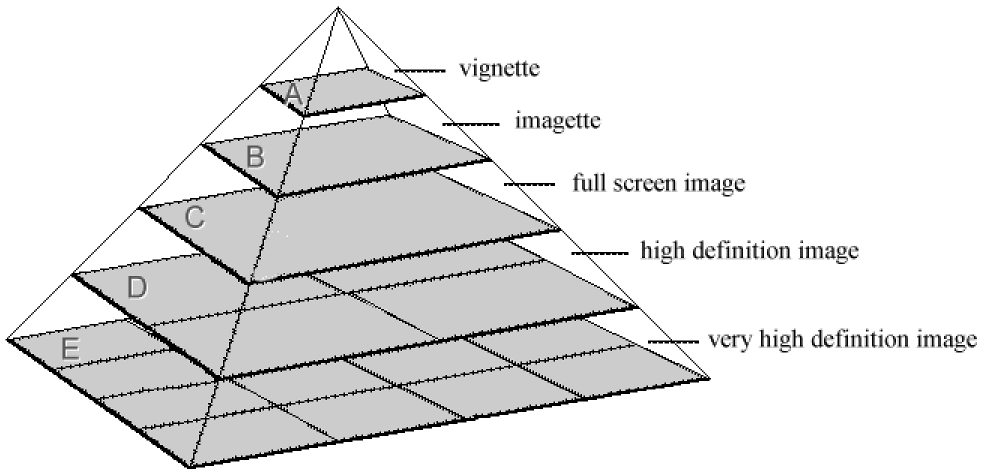
Progressive coding is a way to send an image gradually to a receiver instead of all at once. During sending the receiver can build the image from low to high detail. JBIG uses discrete steps of detail by successively doubling the resolution. For each combination of pixel values in a context, the probability distribution of black and white pixels can be different. In an all-white context, the probability of coding a white pixel will be much greater than that of coding a black pixel. The Q-coder assigns, just like a Huffman coder, more bits to less probable symbols, and thus achieves very good compression. However, the Q-coder can, unlike a Huffman coder, assign one output codebit to more than one input symbol, and thus is able to compress bi-level pixels without explicit clustering, as would be necessary using a Huffman coder.

6.3 JTIP (JPEG Tiled Image Pyramid)

JTIP cuts an image into a group of tiled images of different resolutions. The highest level of the pyramid is called the vignette which is 1/16 the original size and is primarily used for browsing. The next tile is called the imarets which is ¼ the original size and is primarily used for image comparison. The next tile is the full screen image which is the only full representation of the image. Below this tile would be the high and very high definition images. These tiles being 4 and 16 times greater then the full screen image contain extreme detail. This gives the ability to have locally increased resolution or increase the resolution of the whole image.

The primary problem with JTIP is how to adapt the size of the digital image to the screen definition or selected window. This is avoided when the first reduction ratio is a power of 2 times the size of the screen. Thus all tiles will be a power of 2 in relation to the screen. Tiling is used to divide an image into smaller sub images. This allows easier buffering in memory and quicker random access of the image.

JTIP typically uses internal tiling so each tile is encoded as part of the same JPEG data stream, as opposed to external tiling where each tile is a separately encoded JPEG data stream. The many advantages and disadvantages of internal versus external tiling will not be discussed here. Figure 8 shows a logical representation of the JTIP pyramid. As you go down the pyramid, the size of the image (graphically and storage-wise) increases.



<u>**Example of pyramidal tiled image**</u>

Figure 5: JTIP Tiling

CHAPTER 7. IMAGE HISTOGRAM

An **image histogram** is a type of histogram that acts as a graphical representation of the tonal distribution in a digital image. It plots the number of pixels for each tonal value. By looking at the histogram for a specific image a viewer will be able to judge the entire tonal distribution at a glance.

Image histograms are present on many modern digital cameras. Photographers can use them as an aid to show the distribution of tones captured, and whether image detail has been lost to blown-out highlights or blacked-out shadows.[2]

The horizontal axis of the graph represents the tonal variations, while the vertical axis represents the number of pixels in that particular tone.[1] The left side of the horizontal axis represents the black and dark areas, the middle represents medium grey and the right hand side represents light and pure white areas. The vertical axis represents the size of the area that is captured in each one of these zones. Thus, the histogram for a very bright image with few dark areas and/or shadows will have most of its data points on the right side and center of the graph. Conversely, the histogram for a very dark image will have the majority of its data points on the left side and center of the graph.

7.2 Application of histogram

- Histogram of an image represents the relative frequency of occurrence of various gray levels in the image
- Takes care the global appearance of an image
- Basic method for numerous spatial domain processing techniques
- Used effectively for image enhancement

7.3 Why Histogram Processing Techniques?

It is used for equalization of images. Most of time images are not in good quality and the frequency band width of that image not capture the whole time slot that's why the image is not clear so increase the band width of image and increase the quality of image we are used the histogram technique in the MATLAB.

7.3.1 Image enhancement using plateau histogram equalization algorithm

This self-adaptive contrast enhancement algorithm is based on plateau histogram equalization for infrared images. By analyzing the histogram of image, the threshold value is got self-adaptively. This new algorithm can enhance the contrast of targets in most infrared images greatly. The new algorithm has very small computational complexity while still produces high contrast output images, which makes it ideal to be implemented byFPGA (Field Programmable Gate Array) for real-time image process. This paper describes a simple and effective implementation of the proposed algorithm, including its threshold value calculation, by using pipeline and parallel computation architecture. The proposed algorithm is used to enhance the contrast of infrared images generated from an infrared focal plane array system and image contrast is improved significantly. Theoretical analysis and other experimental results also show that it is a very effective enhancement algorithm for most infrared images.

An infrared image is created from infrared radiation of objects and their backgrounds. Generally the temperature difference between target objects and their background is small, and the temperature of background is high, which result in the fact that most infrared images have highly bright back-ground and low contrast between background and targets. In order to recognize targets correctly from these images, good enhancement algorithms must be applied firstly. Gray stretch, histogram equalization are general image enhancement algorithms. And histogram equalization is a widely used enhancement algorithm, in which the contrast of an image is enhanced by adjusting gray levels according to its cumulative histogram. But histogram equalization algorithm is not applicable to many infrared images, because the algorithm often mainly enhances image background instead of targets. In an effort to overcome this problem, Virgil E. Vichers and Silverman, proposed two new histogram-based algorithms:

- plateau histogram equalization and
- histogram projection

Plateau histogram equalization has been proven to be more effective, which suppresses the enhancement of background by using a plateau threshold value. But the plateau threshold value is an empirical value in general which limits the algorithm's practical usage. A modification is made to plateau histogram equalization. By analyzing the histogram of infrared images, an estimated value of plateau threshold value is got self-adaptively. This modified algorithm is able to enhance the contrast of target objects in most infrared images more effectively than the original algorithm. It has very small computational complexity while still produces high contrast output images, which makes it ideal to be implemented by FPGA for real-time imaging applications. This describes an implementation of our proposed algorithm, including its plateau threshold value calculation by using pipeline and parallel computation architecture. The proposed algorithm has been used to enhance infrared images generated from an infrared focal plane array system and the contrast of image has been improved significantly.

7.4 The principles of self-adaptive plateau Histogram equalization

- **Plateau histogram equalization**

Plateau histogram equalization is a modification of histogram equalization, proposed by Virgil. Vichers and Silverman. An appropriate thresh-old value is selected firstly, which is represented as

$$P_T(k) = \begin{cases} P(k), & P(k) \leqslant T, \\ T, & P(k) > T, \end{cases} \tag{1}$$

where k represents the gray level of an image, $0 \leqslant k \leqslant 255$. Then an image enhancement is made by $P_T(k)$ as follows:

$$F_T(k) = \sum_{j=0}^{k} P_T(j), \quad 0 \leqslant k \leqslant 255, \tag{2}$$

$$D_T(k) = \left\lfloor \frac{255 \cdot F_T(k)}{F_T(255)} \right\rfloor, \tag{3}$$

7.5 Selection of self-adaptive plateau threshold Value

Selection of plateau threshold value is very important in the infrared image enhancement algorithm of plateau histogram equalization. It would have effect on the contrast enhancement of images. Appropriate plateau threshold value would greatly enhance the contrast of image. In addition, some plateau value would be appropriate to some infrared images, but not appropriate to others. As a result, the plateau threshold value would be selected self-adaptively according to different infrared images in the process of image enhancement.

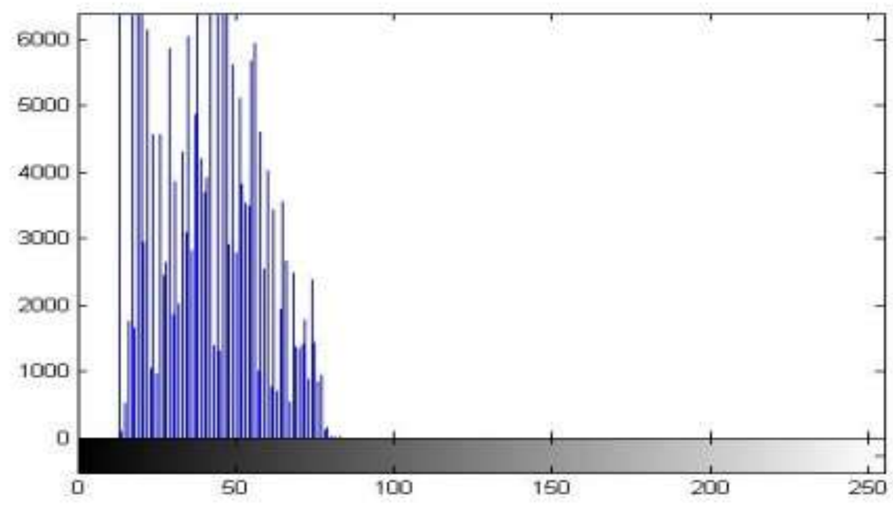
7.6 PROGRAM FOR ALGORITHM

```
Clear all

P=100;                    % set the pixel parameter from  1 to (n1*n2)

A=imread  (‘c:\image\256_256bmp’);

B=rgb2gray(a);

Figure  (1);

imshow(b);

b=double(b);

[n1,n2]  = size(b);                    % read the size of b

For j= 1:256

  k(i)=0;

  c(i)=0;

  d(i)=0;

  ds(i)=0;

end

for i = 1:n1

each

  For j=1:n2

        d=b(i,j);

        k(:,d+1)=k(:,d+1)+1;   % count the no of pixel level(intensity)

    end

end


figure(2);

plot(k);                                             %plot the histogram

xlabel(‘GRAYSCALE  VALUES’);
```

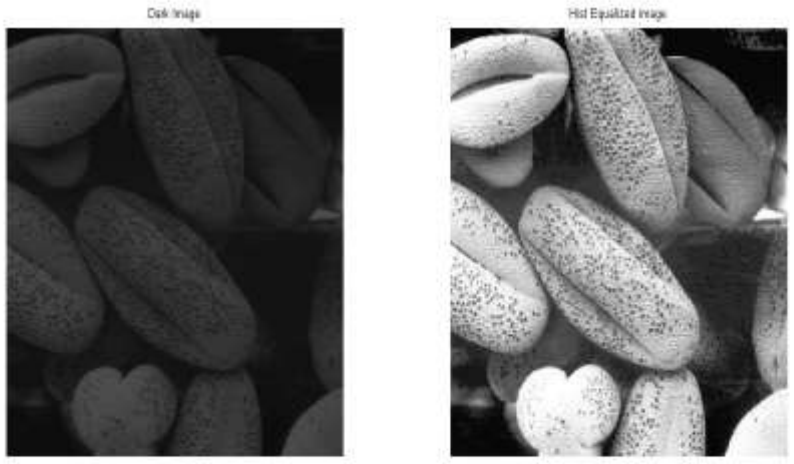
```
ylabel('NUMBER  OF PIXEL');


for i =1:256
   c(i)=main(k(i),P);
end
figure(4);
plot(c);
for i =1:256
    sum=0;
for j=1:i
     sum = sum+c(j);
  end
    d(i)=sum;
end


for i=1:256
    ds(i)=floor((256*d(i)/d(256));
  end
for i=1:n1
     for J=1:n2
            kk=b(i,j);
            c(i,j)=ds(:,k+1)
     end
end


figure(3);
```

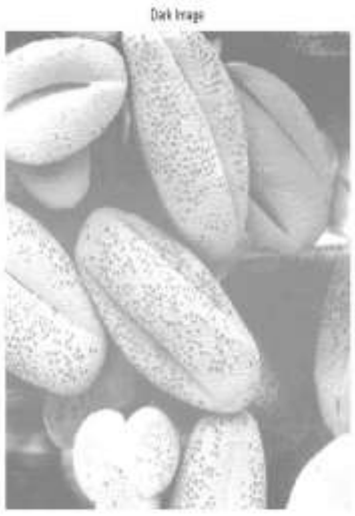
```
imshow(c,[])
```

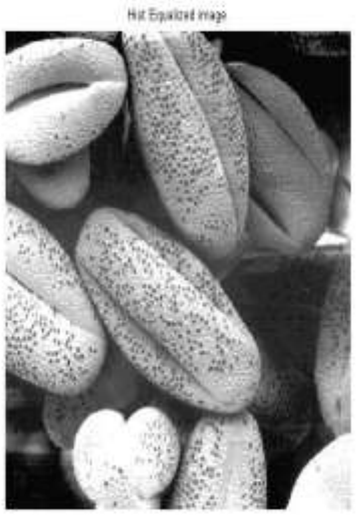
7.7 Histogram Equalization Example



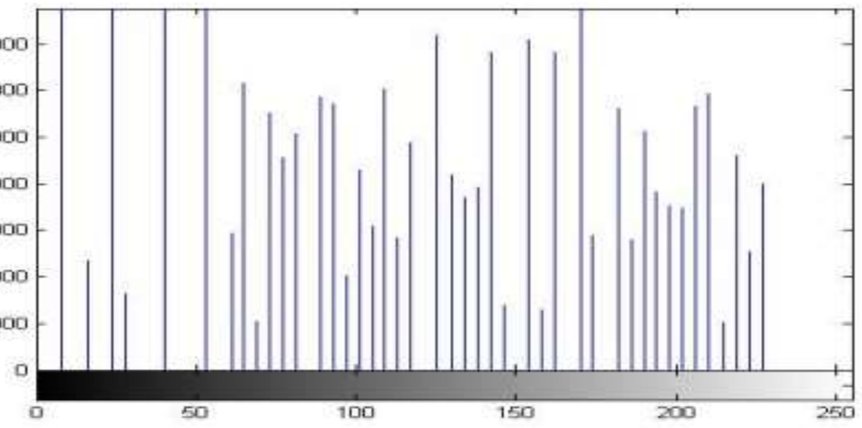
Before applying histogram







Before After



After the equalization

FIGURE 7. HISTOGRAM EQUILIZATION

CHAPTER 8. MATLAB CODE:

```
function jpeg

close all

%   plot_bases( base_size,resolution,plot_type )

% will plot the 64 wanted bases. I will use "zero-padding" for increased resolution

% NOTE THAT THESE ARE THE SAME BASES !

% for reference I plot the following 3 graphs:

% a) 3D plot with basic resolution (64 plots of 8x8 pixels) using "surf" function

% b) 3D plot with x20 resolution (64 plots of 160x160 pixels) using "mesh" function

% c) 2D plot with x10 resolution (64 plots of 80x80 pixels) using "mesh" function

% d) 2D plot with x10 resolution (64 plots of 80x80 pixels) using "imshow" function

% NOTE: matrix size of pictures (b),(c) and (d), can support higher frequency = higher
bases

%      but I am not asked to draw these (higher bases) in this section !

%      the zero padding is used ONLY for resolution increase !

% get all base pictures (3D surface figure)

plot_bases( 8,1,'surf3d' );

 % get all base pictures (3D surface figure), x20 resolution

plot_bases( 8,20,'mesh3d' );

 % get all base pictures (2D mesh figure), x10 resolution

plot_bases( 8,10,'mesh2d' );

 % get all base pictures (2D mesh figure), x10 resolution
```

```
plot_bases( 8,10,'gray2d' );

% for each picture {'0'..'9'} perform a 2 dimensional dct on 8x8 blocks.

% save the dct inside a cell of the size: 10 cells of 128x128 matrix

% show for each picture, it's dct 8x8 block transform.

 for idx = 0:9

     % load a picture

   switch idx

   case {0,1}, input_image_128x128 = im2double( imread( sprintf( '%d.tif',idx ),'tiff' )
);

   otherwise, input_image_128x128 = im2double( imread( sprintf( '%d.tif',idx),'jpeg' )
);

   end

   % perform DCT in 2 dimension over blocks of 8x8 in the given picture

   dct_8x8_image_of_128x128{idx+1}          =          image_8x8_block_dct(
input_image_128x128 );

 if (mod(idx,2)==0)

      figure;

   end

   subplot(2,2,mod(idx,2)*2+1);

   imshow(input_image_128x128);

   title( sprintf('image #%d',idx) );

   subplot(2,2,mod(idx,2)*2+2);

   imshow(dct_8x8_image_of_128x128{idx+1});

   title( sprintf('8x8 DCT of image #%d',idx) );

end

% do statistics on the cell array of the dct transforms

% create a matrix of 8x8 that will describe the value of each "dct-base"
% over the transform of the 10 given pictures. since some of the values are
% negative, and we are interested in the energy of the coefficients, we will
% add the abs()^2 values into the matrix.
% this is consistent with the definition of the "Parseval relation" in Fourier Coefficients
 % initialize the "average" matrix
mean_matrix_8x8 = zeros( 8,8 );
% loop over all the pictures
for idx = 1:10
     % in each picture loop over 8x8 elements (128x128 = 256 * 8x8 elements)
  for m = 0:15
    for n = 0:15
      mean_matrix_8x8 = mean_matrix_8x8 + ...
        abs( dct_8x8_image_of_128x128{idx}(m*8+[1:8],n*8+[1:8]) ).^2;
    end
  end
end
 % transpose the matrix since the order of the matrix is elements along the columns,
% while in the subplot function the order is of elements along the rows
mean_matrix_8x8_transposed = mean_matrix_8x8';
 % make the mean matrix (8x8) into a vector (64x1)
mean_vector = mean_matrix_8x8_transposed(:);
 % sort the vector (from small to big)
[sorted_mean_vector,original_indices] = sort( mean_vector );
 % reverse order (from big to small)
sorted_mean_vector = sorted_mean_vector(end:-1:1);
original_indices = original_indices(end:-1:1);
```

```
 % plot the corresponding  matrix  as asked in section  1.6

figure;

for idx = 1:64

   subplot(8,8,original_indices(idx));

   axis  off;

   h = text(0,0,sprintf('%4d',idx));

   set(h,'FontWeight','bold');

   text(0,0,sprintf('   \n_{%1.1fdb}',20*log10(sorted_mean_vector(idx))  ));

end

% add a title  to the figure

subplot(8,8,4);

h = title(  'Power of DCT coefficients  (section  1.6)' );

set( h,'FontWeight','bold'  );

% picture  8 is chosen

% In this  section  I will  calculate  the SNR of a compressed  image  againts

% the level  of compression.  the SNR calculation  is defined  in the header

% of the function:   <<calc_snr>>  which  is given  below.

%

% if we decide to take 10 coefficients  with  the most energy,  we will  add

% zeros to the other coefficients  and remain  with  a vector 64 elements  long

% (or a matrix  of 8x8)


% load the original  image

original_image  = im2double(  imread(  '8.tif','jpeg'  ) );


% I will  use this  matrix  to choose only  the wanted number  of coefficients
% the matrix is initialized to zeros -> don't choose any coefficient at all
coef_selection_matrix = zeros(8,8);

% compressed picture set (to show the degrading)
compressed_set = [1 3 5 10 15 20 30 40];

% this loop will choose each time, the "next-most-energetic" coefficient,
% to be added to the compressed image -> and thus to improove the SNR
for number_of_coefficient = 1:64

   % find the most energetic coefficient from the mean_matrix
   [y,x] = find(mean_matrix_8x8==max(max(mean_matrix_8x8)));

   % select if for the compressed image
   coef_selection_matrix(y,x) = 1;

   % replicate the selection matrix for all the parts of the dct transform
   % (remember that the DCT transform creates a set of 8x8 matrices, where
   % in each matrix I need to choose the coefficients defined by the
   % <<coef_selection_matrix>> matrix )
   selection_matrix = repmat( coef_selection_matrix,16,16 );

   % set it as zero in the mean_matrix, so that in the next loop, we will
   % choose the "next-most-energetic" coefficient
   mean_matrix_8x8(y,x) = 0;
```

```
    % choose the most energetic coefficients from the original image
    % (total of <<number_of_coefficient>> coefficients for this run in the loop)
    compressed_image = image_8x8_block_dct(original_image) .* selection_matrix;

    % restore the compressed image from the given set of coeficients
    restored_image = image_8x8_block_inv_dct( compressed_image );

    % calculate the snr of this image (based on the original image)
    SNR(number_of_coefficient) = calc_snr( original_image,restored_image );

    if ~isempty(find(number_of_coefficient==compressed_set))
        if (number_of_coefficient==1)
            figure;
            subplot(3,3,1);
            imshow( original_image );
            title( 'original image' );
        end
        subplot(3,3,find(number_of_coefficient==compressed_set)+1);
        imshow( restored_image );
        title( sprintf('restored image with %d coeffs',number_of_coefficient) );
    end
end

% plot the SNR graph
figure;
plot( [1:64],20*log10(SNR) );
```

```
xlabel( 'numer of coefficients taken for compression' );

ylabel( 'SNR [db] ( 20*log10(.) )' );

title( 'SNR graph for picture number 8, section 1.8' );

grid on;

% pdip_dct2 - implementation of a 2 Dimensional DCT

%

% assumption: input matrix is a square matrix !

% -----------------------------------------------------------------------------

function out = pdip_dct2( in )


% get input matrix size

N = size(in,1);


% build the matrix

n = 0:N-1;

for k = 0:N-1

   if (k>0)

      C(k+1,n+1) = cos(pi*(2*n+1)*k/2/N)/sqrt(N)*sqrt(2);

   else

      C(k+1,n+1) = cos(pi*(2*n+1)*k/2/N)/sqrt(N);

   end

end


out = C*in*(C');


% -----------------------------------------------------------------------------
% pdip_inv_dct2  - implementation  of an inverse  2 Dimensional  DCT
%
% assumption:  input  matrix  is a square  matrix  !
% --------------------------------------------------------------------------------
function  out = pdip_inv_dct2(  in )


% get  input  matrix  size
N = size(in,1);


% build  the  matrix
n = 0:N-1;
for  k = 0:N-1
   if  (k>0)
      C(k+1,n+1) = cos(pi*(2*n+1)*k/2/N)/sqrt(N)*sqrt(2);
   else
      C(k+1,n+1) = cos(pi*(2*n+1)*k/2/N)/sqrt(N);
   end
end


out = (C')*in*C;


% --------------------------------------------------------------------------------
% plot_bases  - use the inverse  DCT in  2 dimensions  to plot  the  base pictures
%
% Note: we can get resolution  be zero  pading  of the input  matrix  !!!
%       that  is  by calling:  in  = zeros(base_size*resolution)
%     where:  resolution  is an integer  > 1
%     So I will  use zero pading  for resolution  (same as in  the fourier  theory)
%     instead  of linear  interpolation.
% ---------------------------------------------------------------------------
function  plot_bases(  base_size,resolution,plot_type  )


figure;
for  k = 1:base_size
   for  l = 1:base_size
      in = zeros(base_size*resolution);
      in(k,l)  = 1;                        % "ask"  for the "base-harmonic  (k,l)"
      subplot(  base_size,base_size,(k-1)*base_size+l  );
      switch  lower(plot_type)
      case 'surf3d',  surf(  pdip_inv_dct2(   in  ) );
      case 'mesh3d',  mesh(  pdip_inv_dct2(   in  ) );
      case 'mesh2d',  mesh(  pdip_inv_dct2(   in  ) ); view(0,90);
      case 'gray2d',  imshow(  256*pdip_inv_dct2(   in ) );
      end
      axis  off;
   end
end


% add a title  to the  figure
subplot(base_size,base_size,round(base_size/2));
h = title(  'Bases of the DCT transform  (section  1.3)'  );
set( h,'FontWeight','bold'   );
```

```
% image_8x8_block_dct  - perform a block DCT for an image
% ----------------------------------------------------------------------------------
function  transform_image  = image_8x8_block_dct(  input_image  )


transform_image  = zeros(  size(  input_image,1  ),size(  input_image,2  ) );
for  m = 0:15
    for  n = 0:15
        transform_image(  m*8+[1:8],n*8+[1:8]  ) = ...
            pdip_dct2(  input_image(  m*8+[1:8],n*8+[1:8]  ) );
    end
end




% ----------------------------------------------------------------------------------
% image_8x8_block_inv_dct  - perform a block inverse  DCT for an image
% ----------------------------------------------------------------------------------
function  restored_image  = image_8x8_block_inv_dct(  transform_image  )


restored_image  = zeros(  size(  transform_image,1  ),size(  transform_image,2  ) );
for  m = 0:15
    for  n = 0:15
        restored_image(  m*8+[1:8],n*8+[1:8]  ) = ...
            pdip_inv_dct2(  transform_image(  m*8+[1:8],n*8+[1:8]  ) );
    end
end
```

```
% ------------------------------------------------------------------------------------
% calc_snr - calculates the snr of a figure being compressed
%
% assumption: SNR calculation is done in the following manner:
%           the deviation from the original image is considered
%           to be the noise therefore:
%
%                 noise = original_image - compressed_image
%
%           the SNR is defined as:
%
%                 SNR = energy_of_image/energy_of_noise
%
%           which yields:
%
%                 SNR = energy_of_image/((original_image-compressed_image)^2)
% ------------------------------------------------------------------------------------
function SNR = calc_snr( original_image,noisy_image )


original_image_energy = sum( original_image(:).^2 );
noise_energy = sum( (original_image(:)-noisy_image(:)).^2 );
SNR = original_image_energy/noise_energy;
```

------------------------------------*************************------------

CHAPTER 8. RESULTS AND DESCRIPTIONS

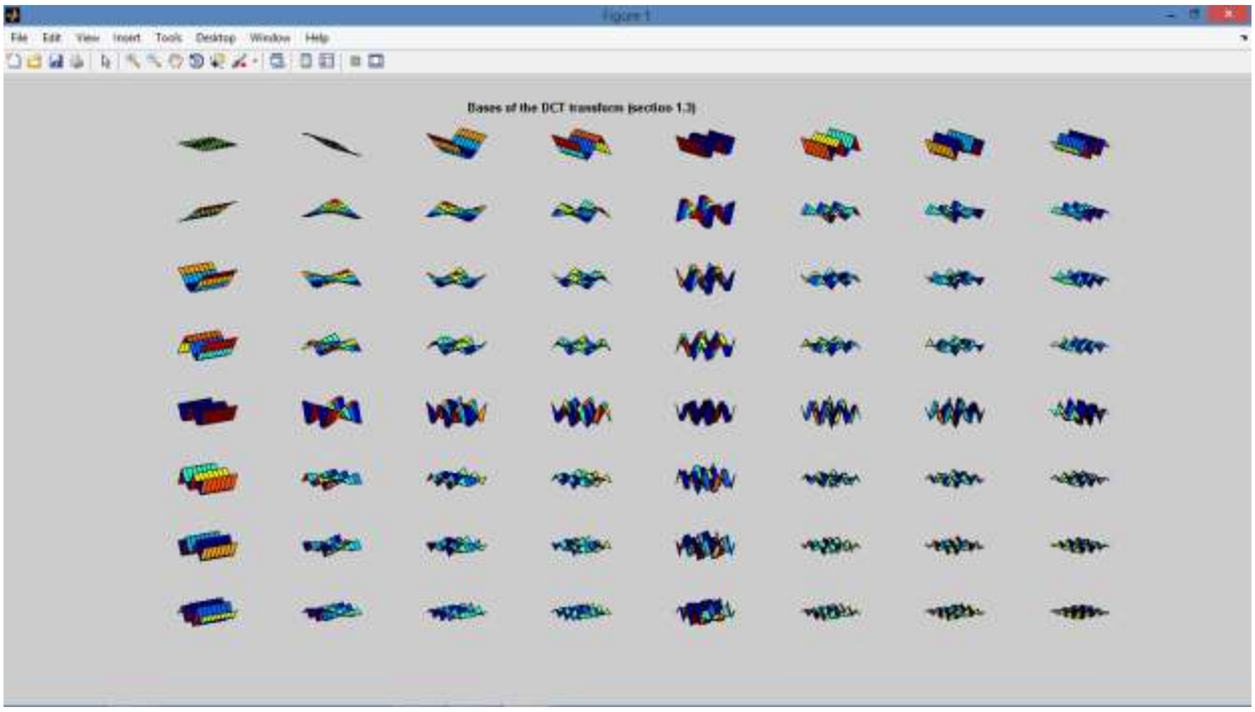


Figure 1
File Edit View Insert Tools Desktop Window Help
Bases of the DCT transform (section 1.3)

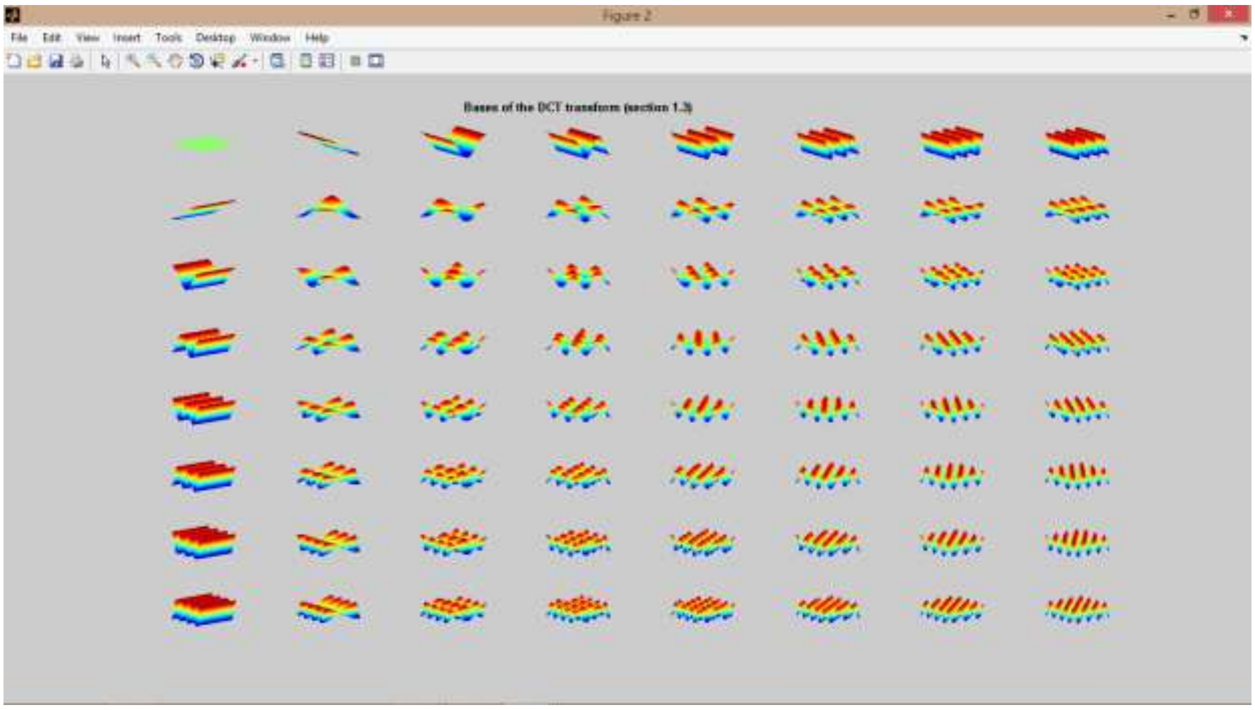


Figure 2
File Edit View Insert Tools Desktop Window Help
Bases of the DCT transform (section 1.3)

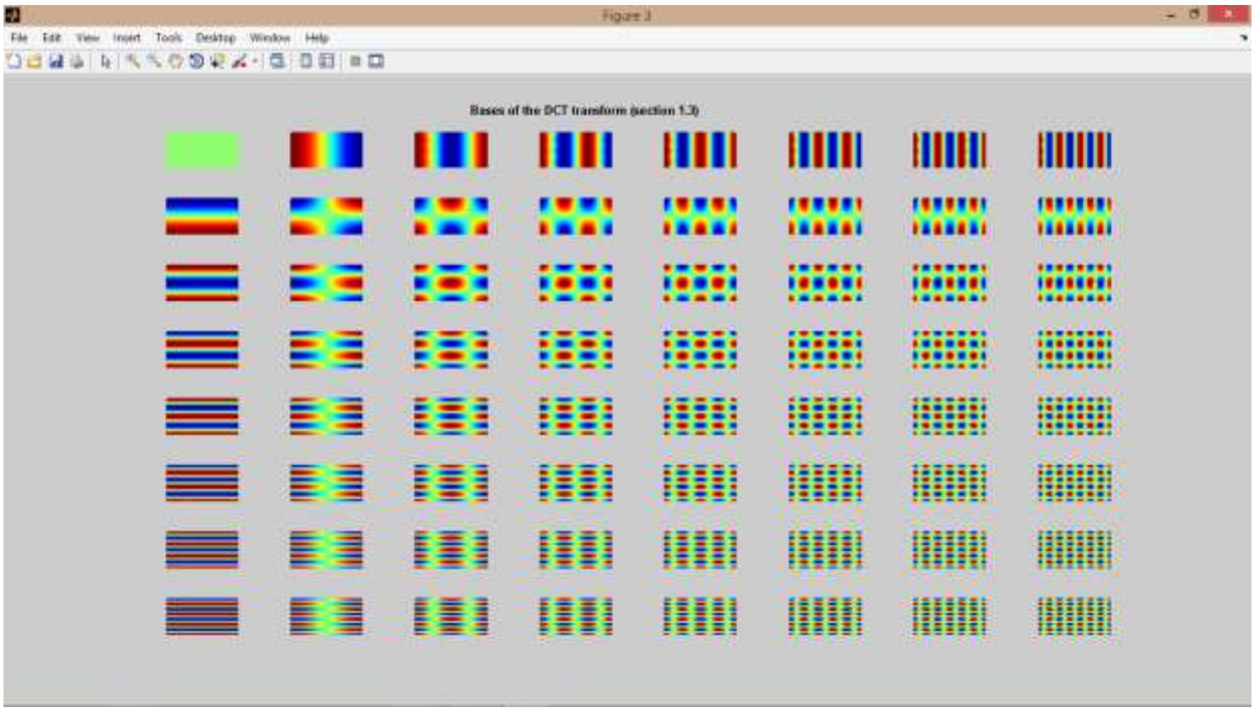


Figure 3
File Edit View Insert Tools Desktop Window Help
Bases of the DCT transform (section 1.3)

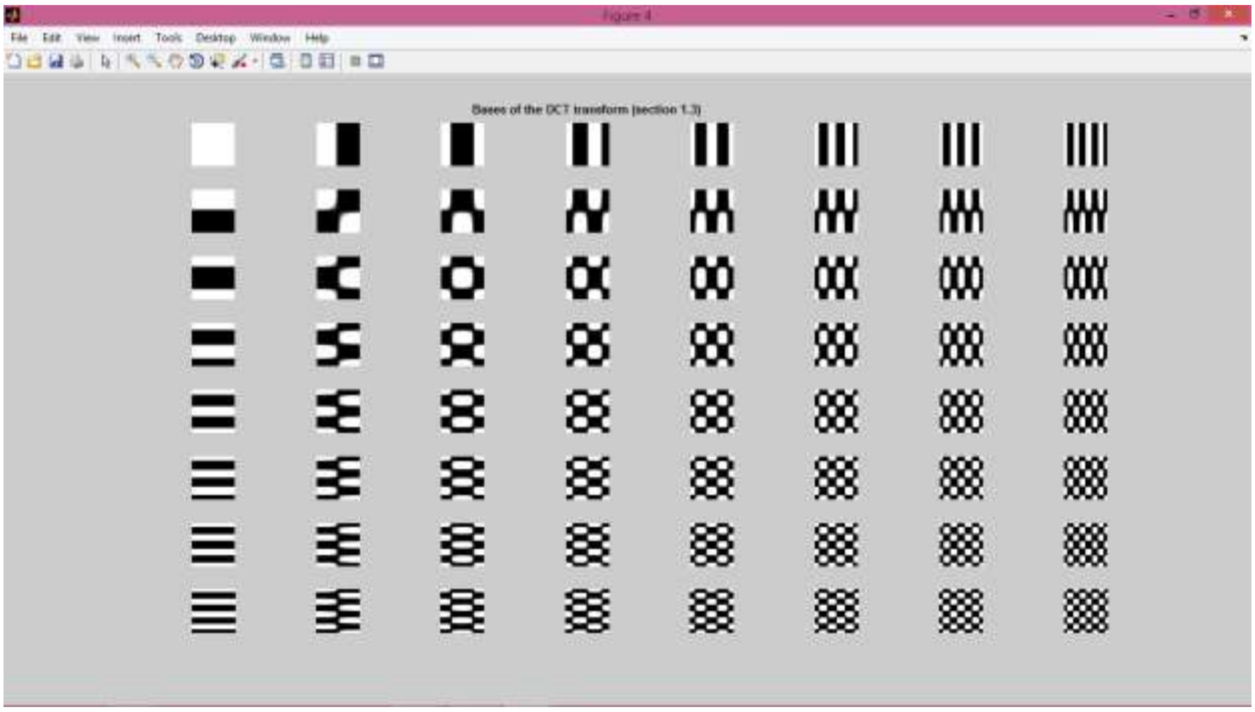


Figure 4
File Edit View Insert Tools Desktop Window Help
Bases of the DCT transform (section 1.3)

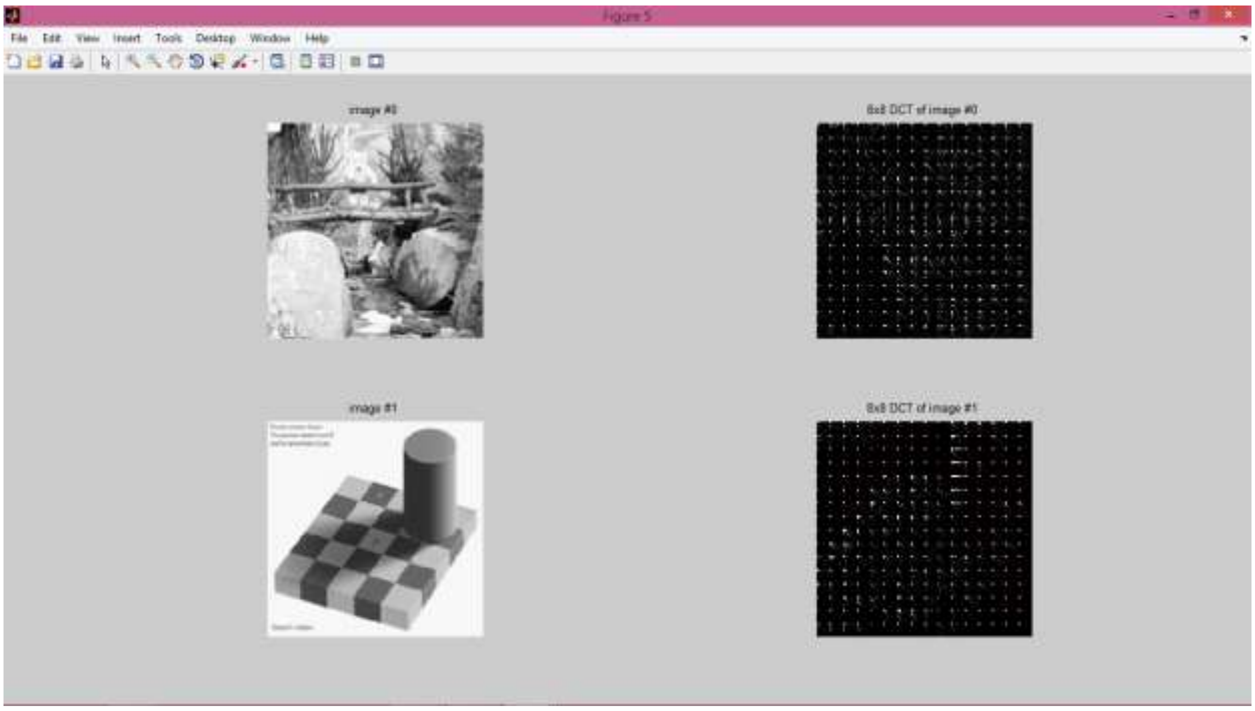


Figure 5
File Edit View Insert Tools Desktop Window Help
image #0
8x8 DCT of image #0
image #1
8x8 DCT of image #1

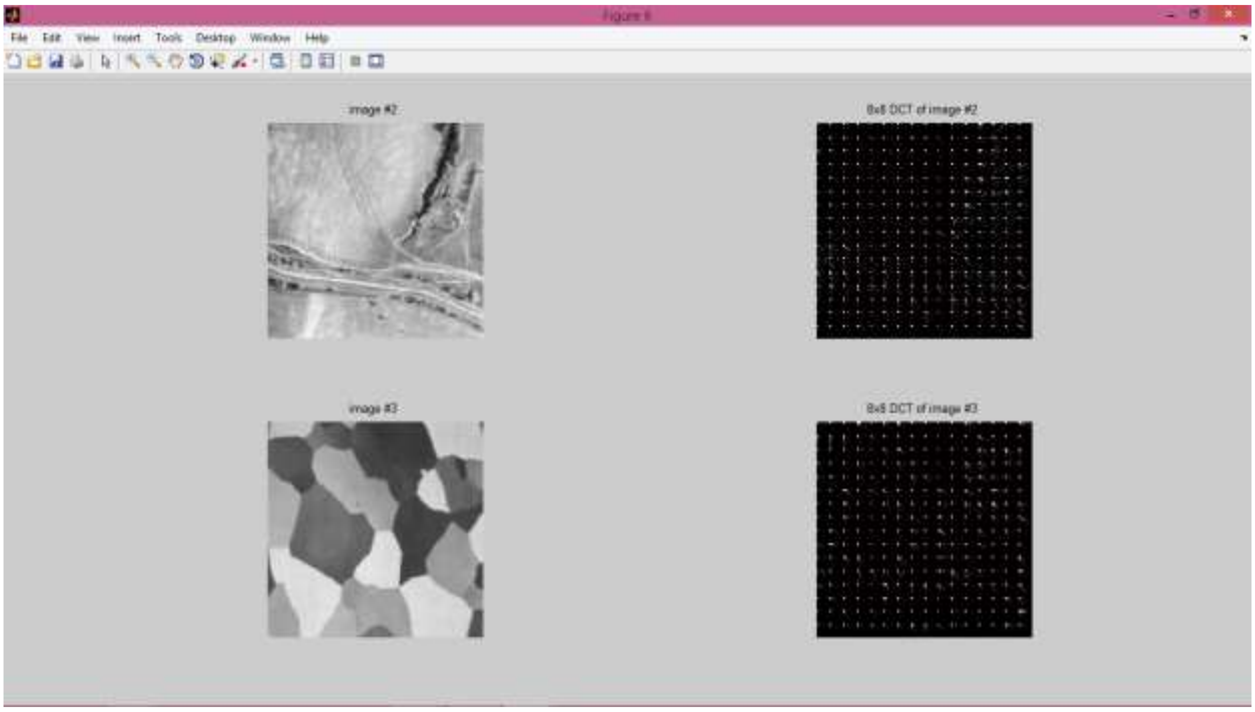


Figure 6
File Edit View Insert Tools Desktop Window Help
image #2
8x8 DCT of image #2
image #3
8x8 DCT of image #3

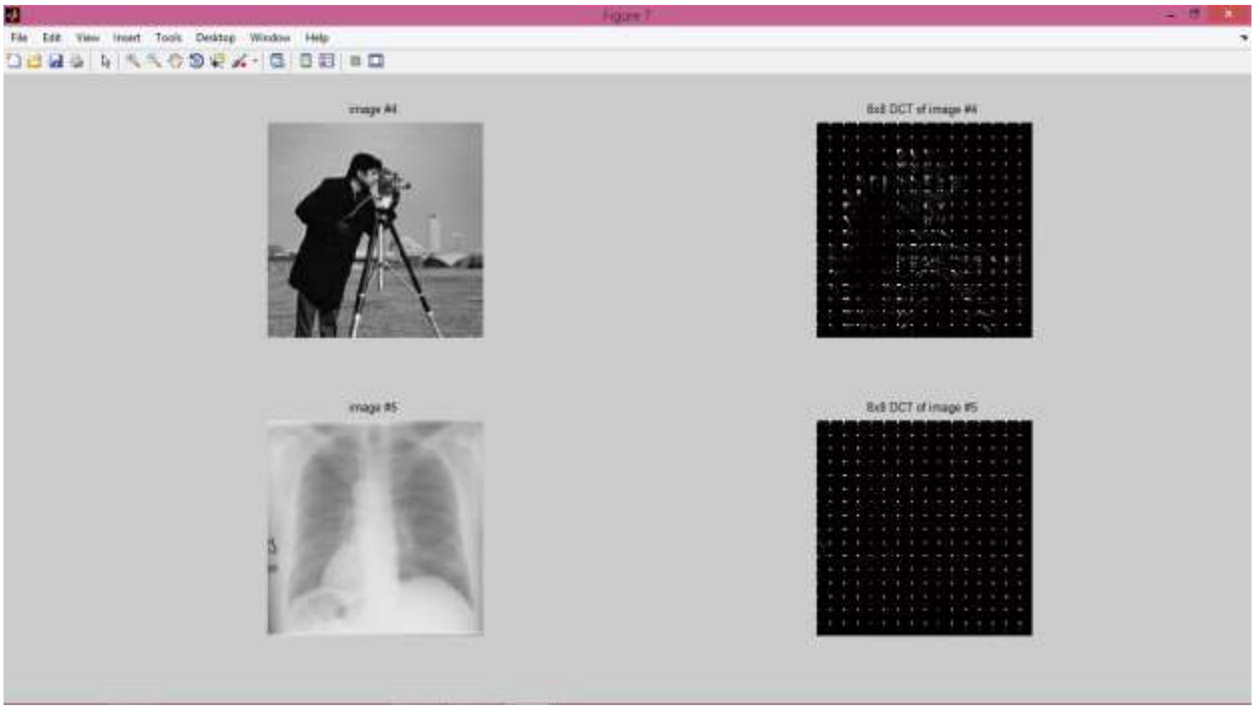


Figure 7
File Edit View Insert Tools Desktop Window Help
image #4
8x8 DCT of image #4
image #5
8x8 DCT of image #5

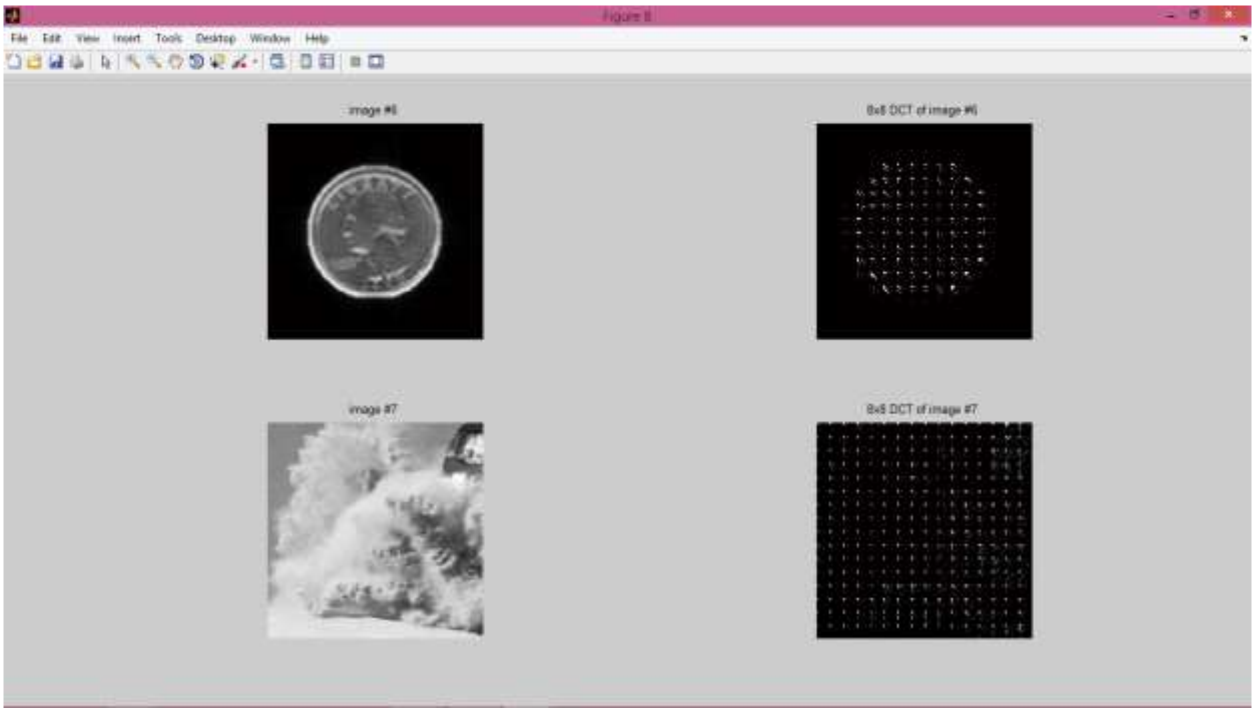


Figure 8
File Edit View Insert Tools Desktop Window Help
image #6
8x8 DCT of image #6
image #7
8x8 DCT of image #7

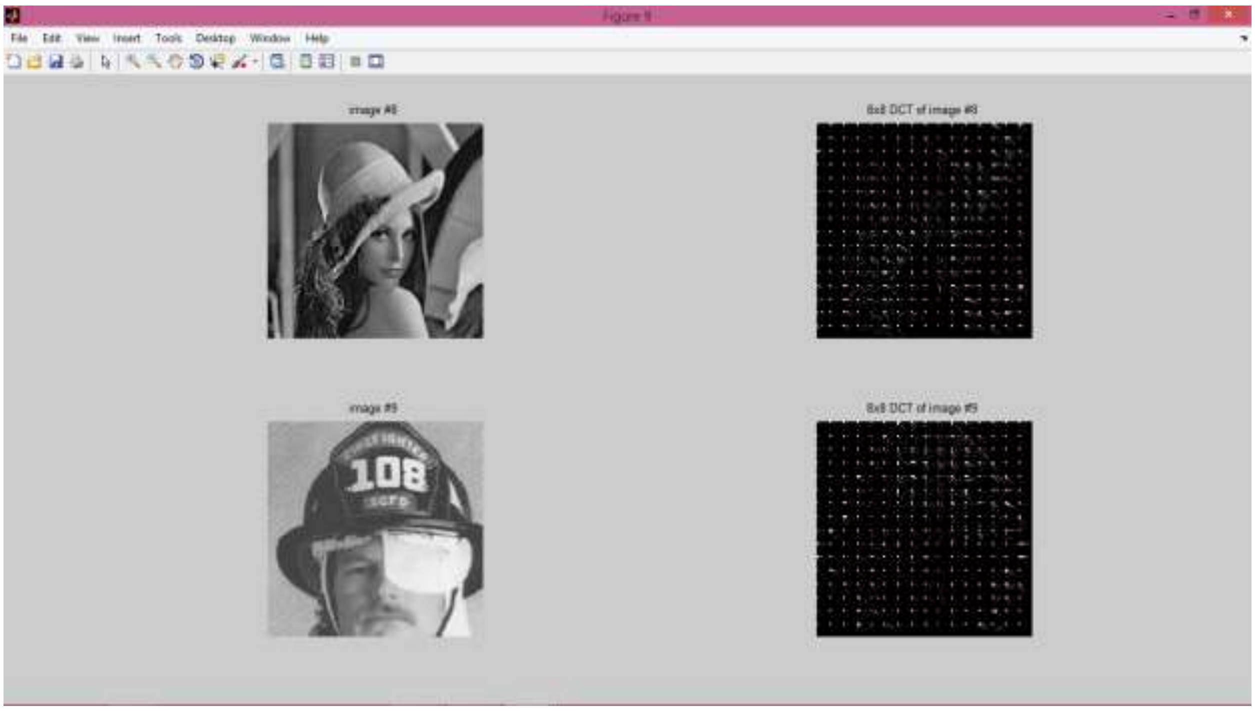
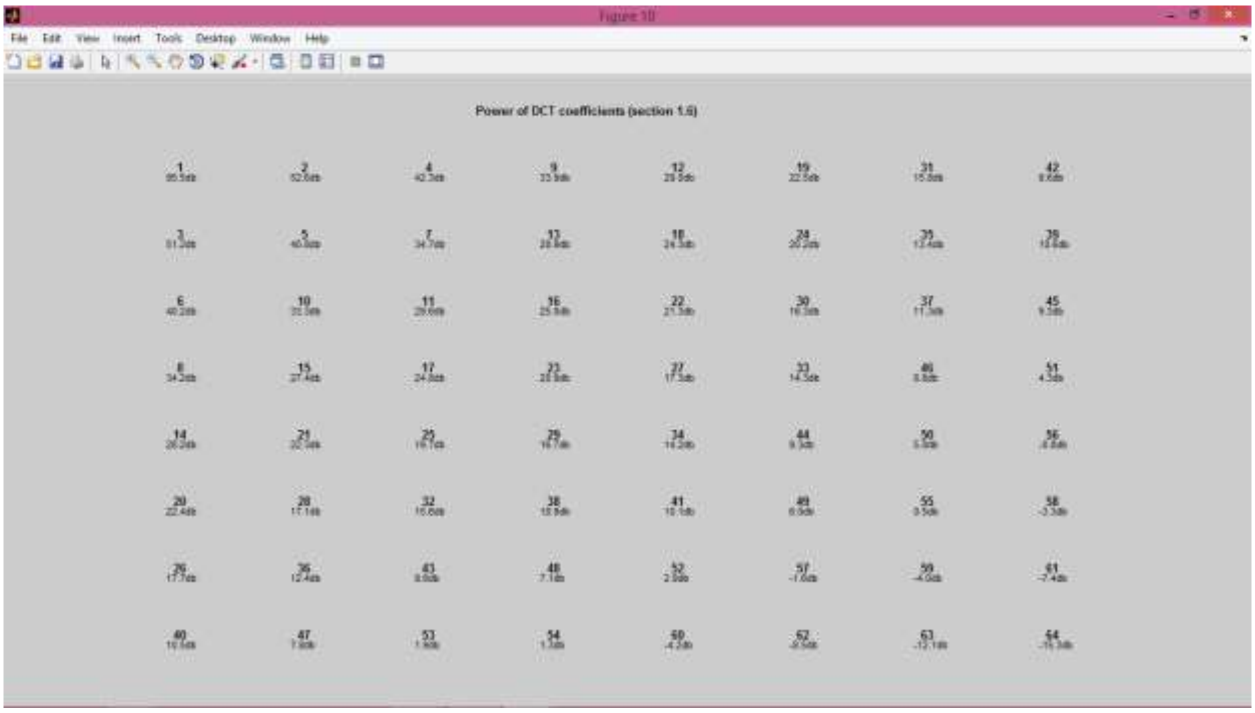


Figure 9
File Edit View Insert Tools Desktop Window Help
image #8
8x8 DCT of image #8
image #9
8x8 DCT of image #9



Power of DCT coefficients (section 1.6)

1 95.5db	2 52.6db	4 42.3db	9 33.9db	12 28.5db	19 22.5db	31 15.8db	42 9.6db
3 51.3db	5 40.8db	7 34.7db	13 28.6db	18 24.3db	24 20.2db	35 12.4db	39 10.6db
6 40.2db	10 33.3db	11 29.6db	16 25.9db	22 21.3db	30 16.3db	37 11.3db	45 9.3db
8 34.2db	15 27.4db	17 24.8db	23 20.9db	27 17.3db	33 14.5db	46 8.8db	51 4.3db
14 26.2db	21 22.3db	25 19.1db	29 16.7db	34 14.2db	44 9.3db	50 5.0db	56 -0.8db
20 22.4db	28 17.1db	32 15.6db	38 10.9db	41 10.1db	49 6.5db	55 3.5db	58 -2.3db
26 17.7db	36 12.4db	43 8.0db	48 7.1db	52 2.9db	57 -1.6db	59 -4.0db	61 -7.4db
40 10.5db	47 7.6db	53 1.9db	54 1.3db	60 -4.2db	62 -9.5db	63 -12.1db	64 -15.3db

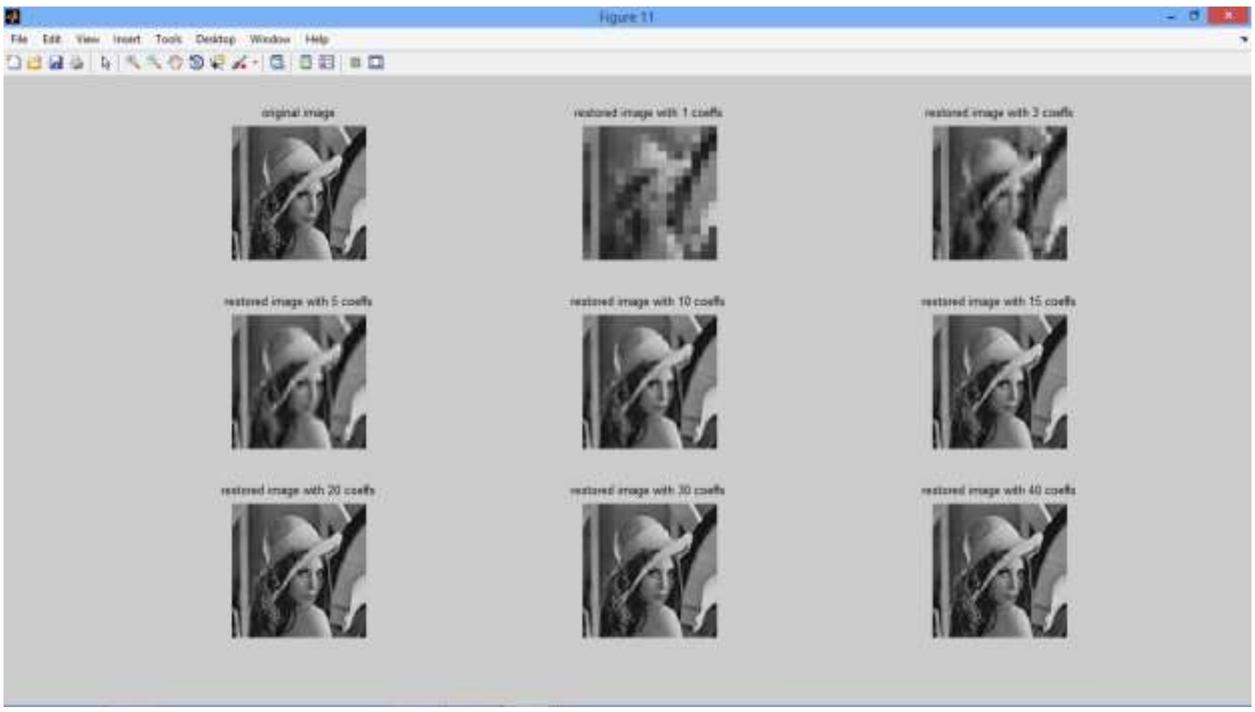
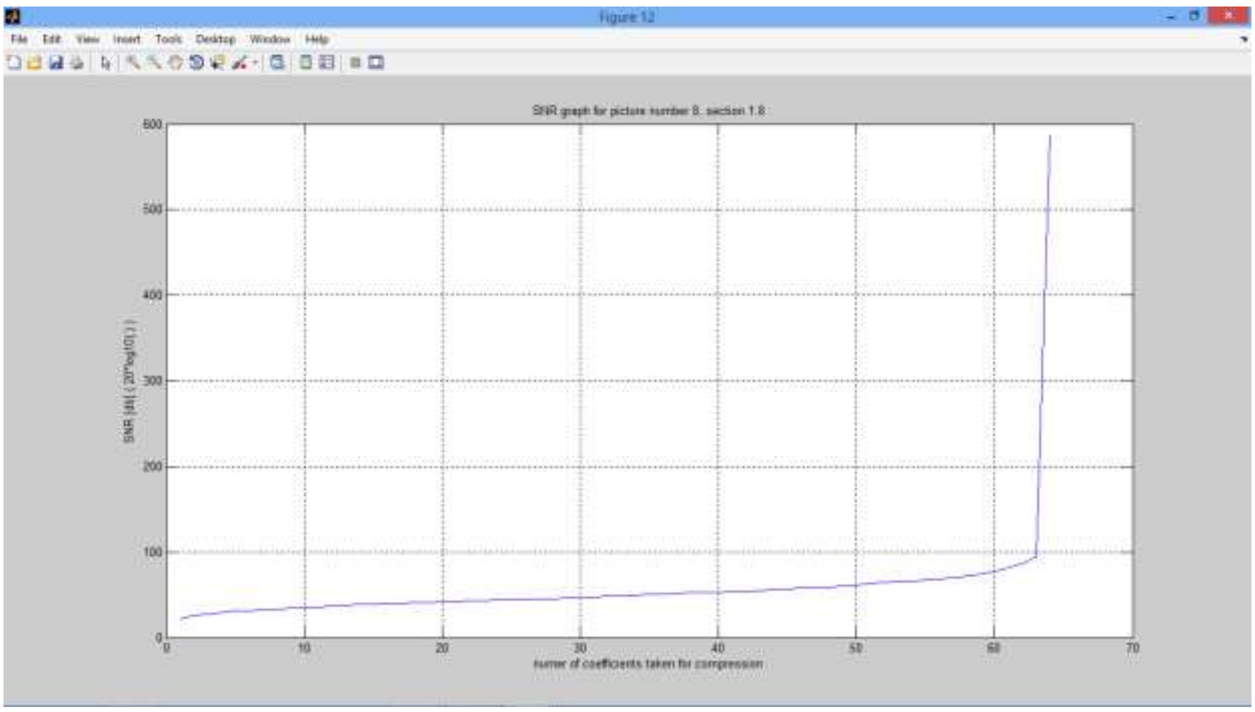


Figure 11
File Edit View Insert Tools Desktop Window Help
original image
restored image with 1 coeffs
restored image with 3 coeffs
restored image with 5 coeffs
restored image with 10 coeffs
restored image with 15 coeffs
restored image with 20 coeffs
restored image with 30 coeffs
restored image with 40 coeffs



CONCLUSION

The JPEG algorithm was created to compress photographic images, and it does this very well, with high compression ratios. It also allows a user to choose between high quality output images, or very small output images. The algorithm compresses images in 4 distinct phases, and does so in $O\left(n^2 \log(n)\right)$ time, or better. It also inspired many other algorithms that compress images and video, and do so in a fashion very similar to JPEG. Most of the variants of JPEG take the basic concepts of the JPEG algorithm and apply them to more specific problems.

Due to the immense number of JPEG images that exist, this algorithm will probably be in use for at least 10 more years. This is despite the fact that better algorithms for compressing images exist, and even better ones than those will be ready in the near future.

REFERENCES

[1] . www.mathworks.in

[2].