HIGH CAPACITY DATA HIDING SYSTEM USING

BPCS STEGANOGRAPHY

by

YESHWANTH SRINIVASAN, B.E.

A THESIS

IN

ELECTRICAL ENGINEERING

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

MASTER OF SCIENCE

IN

ELECTRICAL ENGINEERING

Approved

Dean of the Graduate School

December, 2003

ACKNOWLEDGEMENTS

First of all, I would like to thank Dr. Brian Nutter, without whose help I would never have finished my thesis in such a short duration. His enthusiasm and resourcefulness has helped me at every stage during my thesis, and otherwise, and to say the least, he has been more than a mere thesis advisor to me. Next, I would like to thank Dr. Sunanda Mitra for her support and encouragement. Her energy and commitment has seldom ceased to amaze me. I would also like to thank Dr. Tanja Karp for agreeing to be on my graduate committee and evaluating my work.

Finally, I would like to thank all my family and friends, especially my parents, for helping me get to where I am today.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS ii

LIST OF FIGURES v

LIST OF ABBREVIATIONS vi

CHAPTER

1. INTRODUCTION 1
   - 1.1. Steganography, Cryptography and Watermarking 1
   - 1.2. Steganography Model 2
   - 1.3. Hiding Data in Images–Image Steganography 3
   - 1.4. Applications of Image Steganography 3
2. BPCS STEGANOGRAPHY 5
   - 2.1. Complexity Measure 5
     - 2.1.1. Complexity measure based on the length of black and white border, α 5
     - 2.1.2. Complexity measure based on the number of connected areas, β.. 6
   - 2.2. Canonical Gray Coding System 7
   - 2.3. Resource Blocks and the Conjugation Operation 8
   - 2.4. File Headers and Other Overheads 12
   - 2.5. Encoding Procedure 12
   - 2.6. Decoding Procedure 14
   - 2.7. Calculation for the Total Overheads 15
   - 2.8. Variants and Suggestions for Improvement 16
3. NEW COMPLEXITY MEASURES 19
   - 3.1. Shortcomings of the α Measure 19
   - 3.2. The β Complexity Measure 20
   - 3.3. The γ Complexity Measure 23
   - 3.4. Threshold Values 26
   - 3.5. M-sequences 28

3.5.1. M-sequence Block Stream Conversion (MBSC) 30
3.6. File Headers and Overall Headers 32
3.7. Encoding Procedure 33
3.8. Decoding Procedure 34
3.9. Calculation for the Total Overheads 36
3.10. Suggestions for Improvement 37
4. RESULTS AND CONCLUSION.. 39
4.1. Original BPCS Scheme Using α Complexity Measure 39
4.2. New Complexity Measures with MBSC 43
4.3. Variation of PSNR with Amount of Data Embedded 45
4.4. Dependence of Embedding Capacity on the Base Image 45
4.5. Conclusions 46
4.6. Future Scope 47
REFERENCES 48
APPENDIX.. 50

LIST OF FIGURES

1.1 Steganography model 2

2.1 α & β values for some 8x8 blocks 7

2.2 Resource blocks corresponding to 2 different 8-character sequences 9

2.3 Example to illustrate the Conjugation operation 10

3.1 Two different blocks that are not actually complex 19

3.2 Some typical binary pixel sequences 20

3.3 Three 8x8 blocks with α & β values 22

3.4 Difference between adjacent rows for an 8x8 block 24

3.5 Blocks with various complexity values (α, β, γ) 25

3.6 Distribution of β and γ over 100,000 randomly generated 8x8 blocks 27

3.7 Fibonacci implementation of an m-stage Linear Feedback Shift Register ... 29

3.8 Format and example of the new block 37

4.1 Images showing the result of applying the BPCS scheme and other variants suggested in Chapter II 40

4.2 Graph comparing the number of blocks modified in each plane using fixed and adaptive thresholds 42

4.3 Data hiding using the β and γ measures 44

4.4 Plot of percentage of maximum data embedded with PSNR 45

4.5 Three test images and their maximum capacity 46

A.1 Main menu 50

A.2 Encoding module 51

A.3 Decoding module 51

LIST OF ABBREVIATIONS

2D	two-dimensional
PBC	Pure Binary Coding
CGC	Canonical Gray Coding
BPCS	Bit Plane Complexity Segmentation
LFSR	Linear Feedback Shift Register
MBSC	M-sequence Block Stream Conversion

CHAPTER I
INTRODUCTION

Steganography literally means covered writing and is the art of hiding secret messages within another seemingly innocuous message, or carrier [2]. The carrier could be any medium used to convey information, including wood or slate tablets, tiny photographs or word arrangements. With the advent of digital technology, the list of carriers has been made to include e-mails, audio and video messages, disk spaces and partitions and images. The following work describes a method of steganography for hiding large volumes of data using digital images as carriers.

1.1 Steganography, Cryptography and Watermarking

Steganography is commonly misinterpreted to be cryptography or watermarking. While they are related in many ways, there is a fundamental difference in the way they are defined and the problems to which they are applied. Steganography hides the very presence of secret data in the carrier, and when implemented in its pure form, a hacker can easily decipher and interpret the secret data once the presence of hidden data is detected. In cryptographic applications, the presence of secret data is not deliberately concealed, but the secret data is encrypted so that a hacker cannot easily decipher the secret data from its encrypted counterpart. In other words, while steganography makes the process of detecting the presence of secret information difficult (but allows easy decipherability), cryptography makes deciphering and interpreting the secret information difficult (but keeps its presence open).

Hiding information to protect text, music, movies, and art is usually called *watermarking*, a reference to the light image of the manufacturer's logo pressed into paper when the watermarked object was made [12]. A watermark usually specifies information about the creator of the document, and how and who could use the document. In the case where the watermark is kept invisible, watermarking can be

viewed akin to steganography [11]. In general, watermarks are more robust to malicious data processing than the secret data embedded using steganographic techniques.

1.2 Steganography Model

A model for hiding information in a carrier, using steganography, is shown in Figure 1.1[2].

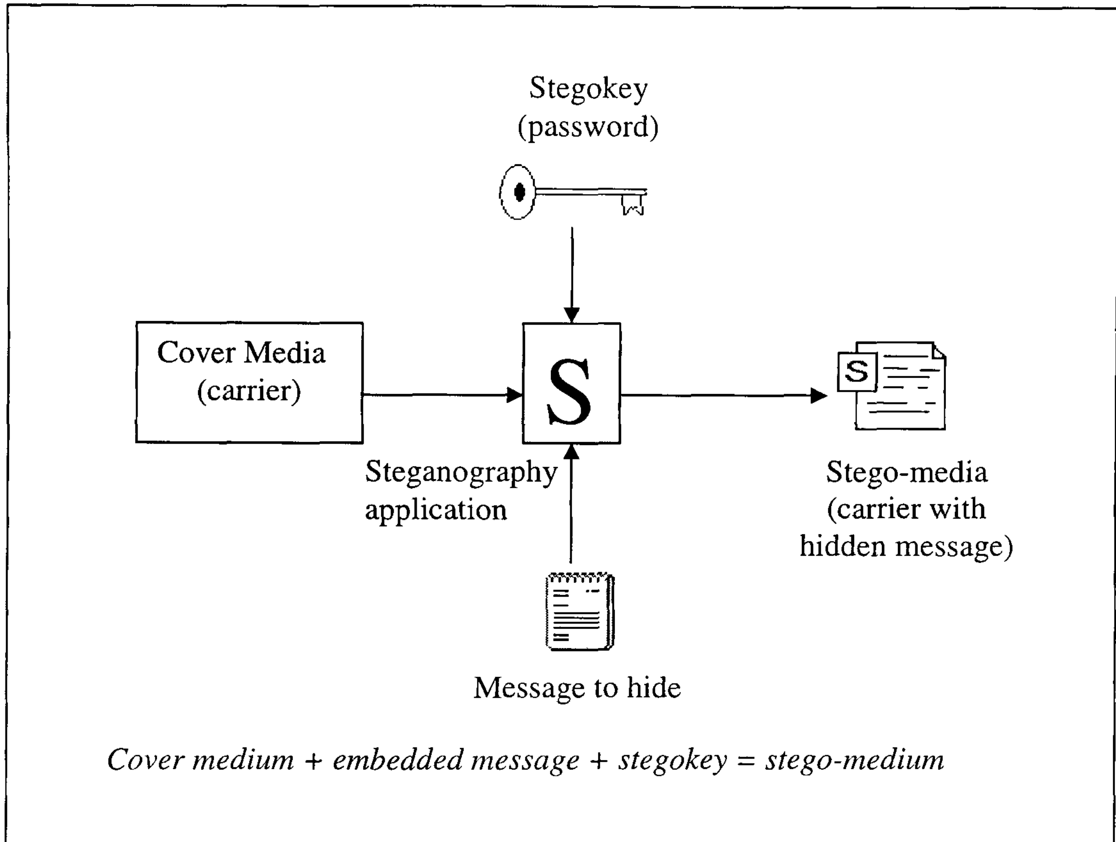


Figure 1.1 Steganography model

The *stegokey* is a password that may be used to encode the secret information to provide an additional level of security. The steganography application is an algorithm that that hides the secret message in the carrier, using the stegokey if necessary, and yields the *stego-media.*

1.3 Hiding Data in Images–Image Steganography

While there are numerous carriers available, the use of digital images as carriers is of particular interest. Even though audio and video files offer a much higher capacity to hide information, digital images are more easily disguised and can be exchanged on a much lower bandwidth. Image steganography techniques can be broadly classified into two categories–spatial domain techniques and transform domain techniques. Spatial domain techniques directly modify the image intensity values to embed the secret information. The most common spatial domain technique is the least significant bit (LSB) manipulation technique, where the LSB of the each intensity value is replaced with one bit of the secret data. Other spatial domain techniques include contrast adjustment, noise insertion etc. Transform domain techniques modify the transform coefficients of the image. The transform coefficients are obtained by applying transforms, such as the Fourier transform, discrete cosine transform or the wavelet transform, to the image [14, 15]. Since most images are compressed by manipulating transform domain coefficients, the transform domain techniques add a fair amount of robustness against the destruction of the secret data due to lossy image compression.

1.4 Applications of Image Steganography

Steganography finds tremendous scope in areas, where there is a need to protect the privacy of information or securely transmit covert information. Consider the case of a spy satellite in orbit. It could be easily made to appear as a regular weather satellite and if a high capacity image steganography system were available, the covert information the satellite gathers could easily be hidden in commonplace weather images. Steganography techniques can also be used to hide classified patient information in X-ray and scan images of the patient. This provides a secure method of associating patient records with their own X-rays and scans. Image steganography could also be used to embed secure information like customer name, account information and key presses in ATM camera feeds and numerous other legal

applications. Of course, it could also be used for various illegal applications like storing inappropriate material on shared computers and smuggling proprietary information from offices.

In this work, a spatial domain technique, called *bit plane complexity segmentation* (BPCS) steganography [1] that allows for hiding large chunks of data in images, is discussed and evaluated. Subtle and radical variations to the existing scheme are suggested and proven to provide a much higher capacity with a significantly improved PSNR.

The remaining chapters are organized as follows. Chapter II describes the BPCS scheme and some subtle variations to the original scheme to improve its performance. In Chapter III, the shortcomings of the complexity measure used in the traditional scheme are discussed and measures to overcome these shortcomings are presented. In the final chapter, the results obtained by using the techniques discussed in Chapters II and III are compared and conclusions drawn.

CHAPTER II

BPCS STEGANOGRAPHY

Bit Plane Complexity Segmentation (BPCS) was introduced in 1998 by Eiji Kawaguchi and Richard O. Eason [1] to overcome the shortcomings of the traditional Least Significant Bit (LSB) manipulation techniques [2]. While the LSB manipulation technique works very well for most gray scale and RGB color images, it is severely crippled by its limitation in capacity, which is restricted to about one-eighth the size of the base image. BPCS is based on the simple idea that the higher bit planes could also be used for embedding information provided they are hidden in seemingly "complex" regions.

2.1 Complexity Measure

The first step in BPCS Steganography is to find "complex" regions in the image where data can be hidden imperceptibly. There is no universal definition for the complexity of an image (or a region of an image). Kawaguchi and Niimi discuss two different complexity measures, one based on the length of the black-and-white border and another based on the number of connected areas that could be used to find the complex regions in an image [4].

2.1.1 Complexity measure based on the length of black and white border, α

This measure is defined on the 4-connected neighborhood of a pixel. The total length of the black-and-white border is defined as the sum of the color changes along the rows and columns in the image. For example, a single white pixel surrounded by 4 black pixels, i.e., having all its 4-connected neighbors as black pixels, will have a border length of 4 (2 color changes each along the rows and columns). Extrapolating this idea to a square binary image of size 2^N x 2^N, the minimum border length possible is 0, obtained for an all white or all black image, and the maximum border length possible is $2 * 2^N * (2^N - 1)$, for the black and white checker board pattern ($(2^N - 1)$ changes along each of the 2^N rows plus the same along the columns). The image

complexity measure, α, is then defined as the normalized value of the total length of the black and white border in the image, i.e.

$$\alpha = \frac{k}{2 \times 2^N \times (2^N - 1)} \ , \ 0 \le k \le (2x2^N x(2^N - 1)) \ , \qquad (2.1)$$

here k is the actual length of the black and white border in the image. It is evident that α lies in [0, 1].

2.1.2 Complexity measure based on the number of connected areas, β

This measure is again based on the 4-connected neighborhood. β is defined as

$$\beta = \frac{m}{2^N \times 2^N} \ , \qquad (2.2)$$

here m is the number of connected areas in the 2^N x 2^N square binary image. It is easily seen that β lies in [1/(2^N x 2^N), 1] with the maximum in the range obtained for the checker board pattern and the minimum obtained for the plain white or plain black image.

The assumption that the image is a square of size 2^N x 2^N severely cripples the applicability of these measures to all images, considering that images are not always perfectly square. To make these complexity measures more generic, they are applied to each exclusive 2^n x 2^n block, where n is typically between 2 and 4, of any MxN image. The only condition is that both M and N have to be divisible by 2^n. This limits higher values of n. Very small values for n (n = 1, 2) provide too much spatial localization for the complexity measures to be meaningful. In practice, n is fixed at 3 so that the complexity measure is applied to each exclusive 8x8 block of the image.

Figure 2.1 (a) and (b) show the α and β values for 2 typical 8x8 blocks. In practice is it found that, for 8x8 blocks, the α measure is more or less uniformly distributed in [0, 1] while the β measure tends to have a definite peak at β = 0.2 [4]. Hence the α measure is preferred for the BPCS application and is the only complexity measure discussed in the rest of this chapter. As already mentioned, the maximum values for α and β are obtained for the black and white checker board pattern shown in Figure 2.1 (c).

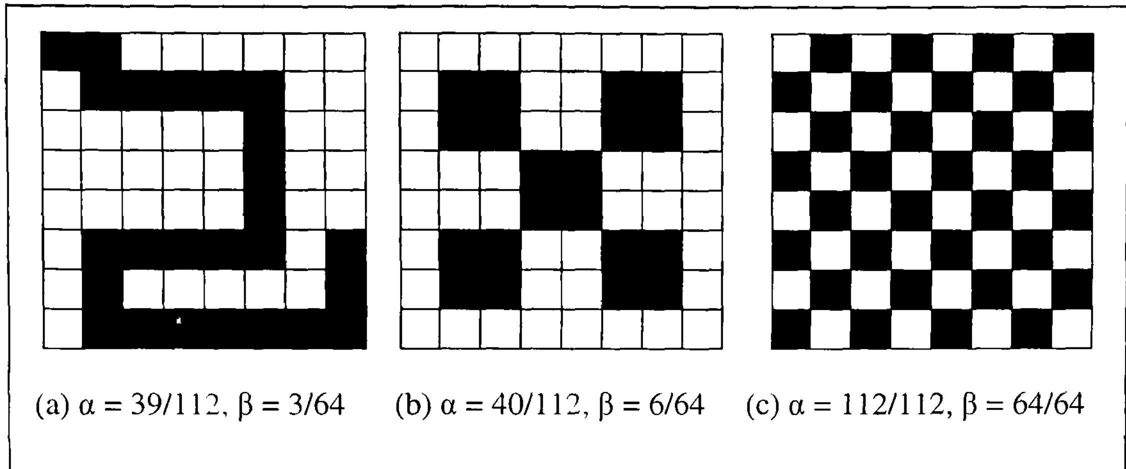


Figure 2.1 α and β values for some 8x8 blocks

2.2 Canonical Gray Coding System

The complexity measures defined in Section 2.1 are defined only for binary images. Each 8-bit grayscale image can be split into 8 binary planes, one plane for each of the 8 significant bits in the 8-bit binary representation of image intensity values. 24-bit color images are composed of three 8-bit planes, one each for Red, Green and Blue and can be split into 24 binary planes. The operation of splitting the image into its constituent binary planes is called Bit-Plane Slicing [5]. Bit-plane slicing can be done in the Pure-Binary Coding system (PBC) wherein the intensity values (for each plane in the case of RGB images) are represented as 8 bit binary numbers, but it suffers from a serious drawback. Consider an 8-bit image where a large portion of the image is composed of pixels whose intensity values alternate between 127 and 128. 127 is 01111111 in binary and 128 is 10000000. Thus all 8 corresponding bit-planes for the 2 pixels are different (coding theory people would say the 'distance' is 8). This idea of two numbers being very similar in value yet differing greatly in their binary representation, on a bit by bit basis, is called the "Hamming Cliff" [6]. In such a region, if these 2 gray levels are sufficiently randomly distributed, all the 8 planes, including the MSB plane, corresponding to these regions would appear complex and hence would be replaced by data to be hidden. After

embedding, 01111111 could easily become 11111111 and 10000000 could become 00000000 and what was an intensity difference of just 1 gray level and was rather unnoticeable, now becomes a difference of 256 and appears as an eccentric white pixel next to a black pixel or vice-versa.

This problem is easily alleviated by using the principle used in some electromechanical applications of digital systems where sensors are required to produce digital outputs that represent a mechanical position [3]. The coding system used is called the Canonical Gray Coding System (CGC), where successive decimal numbers differ in their representation by just one bit. It is a canonical system, as the binary system and the gray code system share a one-to-one correspondence. The 2 numbers in the above example, 127 and 128, would be represented in CGC as 01000000 and 11000000, respectively, and hence would not differ by more than 1 bit. Thus, the first step in BPCS Steganography is to convert the absolute intensity values (it is assumed that they lie in [0, 255]) into CGC by a 1-to-1, PBC-to-CGC mapping. This is followed by bit-plane decomposition on the CGC values, and the 8 binary images obtained are called the CGC images. The CGC images don't suffer from Hamming cliffs as regions that are rather smooth in the original image result in very few changes in the higher bit planes, and these regions are appropriately determined unsuitable for embedding data.

Since 24-bit RGB color images provide a very high capacity for data hiding applications, all the following sections are discussed with 24-bit RGB images as reference. BPCS Steganography can also be applied, effectively, to 8-bit grayscale images.

2.3 Resource Blocks and the Conjugation Operation

Once the 24-bit image (base image) has been split into its 24 constituent bit-planes, and the complexity, α, of each exclusive 8x8 block in each of the 24 bit-planes has been found, the complexity of each block is compared with a threshold, α_0. If $\alpha > \alpha_0$, the block is deemed complex enough to be replaced by data blocks. The

standard value used for α_0 is about 0.3. The data chunks that replace the complex blocks in the bit-plane image are called Data Blocks or Resource Blocks. The resource blocks are chunks of data obtained from any ASCII-encoded file (data or image files that can be read as a string of ACSII characters) called the Resource File. The resource file could be a text file or a Word document or even an image. Each 8 byte block of a resource file forms an 8x8 resource block with the 8-bit binary representation of each byte forming the row of the 8x8 block. For example, a sequence of 8 characters from an MS word document 'This one' (8 characters including the blank space) would form an 8x8 resource block as shown in Figure 2.2 (a) and a block of 8 consecutive blank spaces would form a block as shown in figure 2.2 (b) since, 'T' is 01010100, 'h' is 01101000, blank space is 00100000 (32 in ASCII) and so on.

(a)

0	1	0	1	0	1	0	0	T
0	1	1	0	1	0	0	0	h
0	1	1	0	1	0	0	1	i
0	1	1	1	0	0	1	1	s
0	0	1	0	0	0	0	0	
0	1	1	0	1	1	1	1	o
0	1	1	0	1	1	1	0	n
0	1	1	0	0	1	0	1	e

(b)

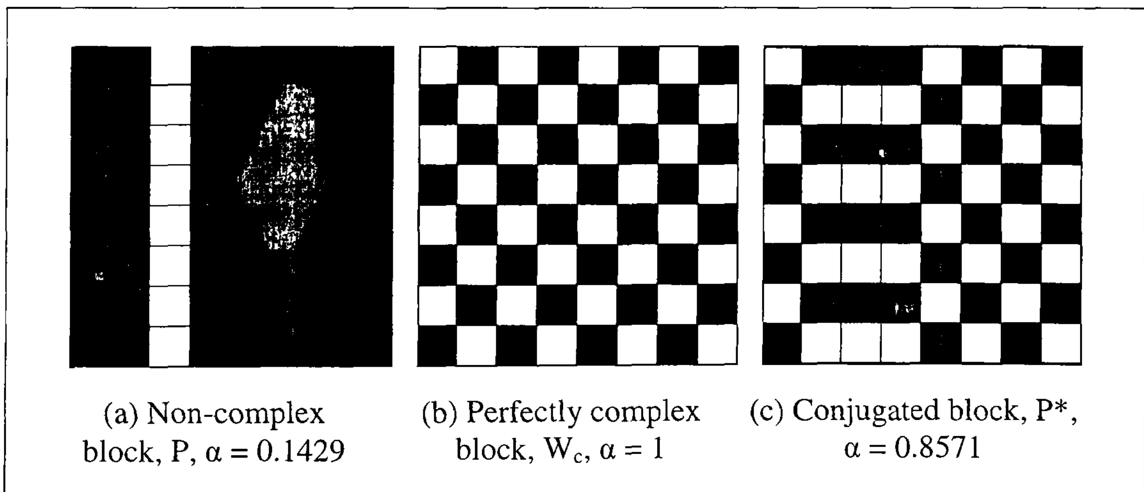
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0

Figure 2.2 Resource blocks corresponding to 2 different 8-character sequences

After the resource file is broken into 8-byte chunks and cast into 8x8 binary resource blocks, they are ready to replace the complex blocks in the bit-plane images, however, with one problem. A complex block denotes a block that appears noisy, and modifying such a block wouldn't be perceptible, unless, the modification results in making the block less complex than the threshold complexity value, α_0. Consider the block shown in Figure 2.2 (b). This is a frequently encountered block as Word documents contain numerous stretches of blank space.

The complexity of this block is 0.1429, which is far less than the α_0 value usually used. If this block replaces a complex block in the bit-plane image, a definite discrepancy arises, especially if it is in one of the higher order bit-planes. Also, the decoding module will not recognize the block, as it assumes that only the complex blocks have been replaced and hence only the complex blocks have valid information. To overcome this problem, the *conjugation operation* is introduced.

Figure 2.1 (c) shows the most complex 8x8 block possible, with a complexity of 1. This block is denoted as W_c with its top-left value being 1. A similar checker-board pattern with complexity 1 can be formed with the top-left value to be a 0, and that is denoted by B_c. The all white and all black blocks are denoted by W and B. W_c is used for all future explanations, although all of it would apply to B_c as well. The W_c block has a special property that when it is XORed (exclusive OR operation) with a non-complex block, P (say), of complexity say $\alpha_n < \alpha_0$, then the resulting block, P*, has a complexity of $(1 \quad \alpha_n) > \alpha_0$. As with any XOR operation, the block P can be easily retrieved by XORing again with W_c. This operation of changing the complexity of a block by XORing with W_c is called the *conjugation operation* and is denoted by '*'. Figure 2.3(a) shows a non-complex (or simple) block, P (say), 2.3(b) is the perfectly complex block, W_c, and 2.3(c) is the conjugated block, P*, obtained by XORing corresponding pixels in Figures 2.3(a) and (b).



(a) Non-complex block, P, $\alpha = 0.1429$

(b) Perfectly complex block, W_c, $\alpha = 1$

(c) Conjugated block, P*, $\alpha = 0.8571$

Figure 2.3 Example to illustrate the Conjugation operation

The 2 important properties of the conjugation operator can be summarized as:

1. $\alpha(P^*) = 1 - \alpha(P)$
2. $(P^*)^* = P$.

The first property is used at the encoder to make the non-complex resource blocks complex, while the second property is used at the decoder to retrieve the original block.

It happens that not all the resource blocks need to be made complex, as most of them are complex on their own. It becomes important to keep track of which blocks have been conjugated. This is done by using a *Conjugation Map*. For every 8 byte block of the resource file, one bit is appended to the conjugation map to indicate if the block has been conjugated. A '1' implies the resource block has been conjugated, while a '0' implies the resource block was embedded as is. The conjugation map is finally embedded after embedding all the resource blocks.

Again, it is possible that the conjugation map itself, when cast into 8x8 blocks, may not form a complex block, and it becomes necessary to specify a conjugation map for the conjugation map. This problem can be resolved in one of the following two ways:

a. Form the conjugation map for the conjugation map and embed the same into some known complex section of the image, for example, the LSB planes.
b. Read the bits in the conjugation map in blocks of 63 bits (padding the last block with zeros if necessary). Cast them into 8x8 blocks, making the first bit (top left bit) zero. If the block is complex, then embed it 'as is'. If it is not, then conjugate the block. The conjugation process automatically makes the top-left bit a one. This bit can be used by the decoder to understand that the block was conjugated at the encoder.

This implementation uses the second method as it offers greater flexibility when dealing with images that may not necessarily have complex LSB planes.

2.4 File Headers and Other Overheads

Each type of file (documents or images) has its own header which is included when the resource file is read, but some additional information about the file, like the file size, file name etc., must to be embedded with the file in order to perfectly reconstruct the files at the decoder. This information is attached as a 24-byte external header, for each file, and is embedded before each file. This is denoted as *FH* for *File Header*. The FH reserves 18 bytes for the file name with its extension, read as a string, and 6 bytes for the file size. Each byte in the 'file size' section of the header represents a digit. This fixes the maximum size of a single resource file to 999,999 bytes. Also, the encoder needs to make note of the number of files, n, it is embedding. This information is embedded as the first byte of an 8-byte *Overall Header* denoted *OH*. The rest of the OH is reserved for future use and the OH is embedded as the first block, before any file is embedded.

With all this information embedded with the resource files, the decoder only needs the encoded image (base image in which the resource files have been hidden) to retrieve all the information about the resource files including their names and sizes. After embedding all the headers and the respective resource files with their conjugation maps into the complex sections of the CGC bit-planes, the 24 bit-planes are put together, in sets of 8 each, to form an RGB image, and then they are converted back to PBC by using a Gray to Binary lookup table. This image is then saved as the encoded image, either with the same name as the original, wherein it replaces the original image, or with a new name.

2.5 Encoding Procedure

The steps involved in the encoding procedure can be summarized as follows:

1. Read the image, convert the intensity values into Gray code and perform bit-plane decomposition.
2. Determine a threshold for the complexity, α_{th}. For each exclusive 8x8 block in the bit-planes, calculate the complexity α. If $\alpha > \alpha_{th}$, mark up the 8x8 block to be

complex (say by marking it up with 1. For a 512x512 image, this "mark up" matrix would be 64x64, for each bit-plane).

3. Get the number of resource files to be embedded, n, make it the first byte of the Overall Header (OH), and embed that into the first complex block of the base image, conjugating it if necessary. Repeat steps 4 to 7 'n' times or till the maximum embeddable capacity is reached.
4. Read in the resource file and form it into a sequence (or vector) of ASCII values. Pad the sequence so that the number of bytes in the sequence is a multiple of 8. This is done because the encoder embeds blocks of 8 bytes at a time. Attach the 24 byte file header containing the file name and size to it.
5. Read the file header, 8 bytes at a time, and form it into 8x8 binary blocks. Calculate α for the block and do one of the following:
   (a) If $\alpha > \alpha_{th}$, then embed the resource block "as is" into the 8x8 block marked '1' in the base image (i.e., complex block in the base image) and append a '0' to the conjugation map to indicate that the block has not been conjugated.
   (b) If $\alpha < \alpha_{th}$, then conjugate the resource block to increase its complexity to $(1 - \alpha)$ (it is assumed that α_{th} is less than 0.5, which it usually is) and then embed the resource block "as is" into the 8x8 block marked '1' in the base image. Append a '1' to the conjugation map to indicate that the block has been conjugated.
6. Break the conjugation map into blocks of 63 bits each, padding with zeros for the final block, if necessary. Make the first bit (top-left bit) of an 8x8 block '0' and add the 63 bit block, into it, by rows. If the block is complex, embed it 'as is' into the next available complex block in the bit-plane base image. If the block is not complex, then conjugate it and embed it into the next available complex block in the bit-plane base image. For the file header there will be just one such block.
7. Repeat steps 5 and 6 substituting the resource file sequence for the file header.

8. Put back the 24 bit-planes together to form 3 color planes, R, G & B, convert from CGC to PBC, and save the image, either under a new name or under the same name as its original to eliminate suspicion. This is the encoded image.

2.6 Decoding Procedure

The work of the decoder is to systematically reverse the operations at the encoder. The way the encoder is organized, all the blocks that are complex in the original image are complex in the encoded image as well. All that the decoding module has to do is tread through each 8x8 block in the bit-plane decomposed image (after converting from PBC to CGC), check if complex and decode the relevant data. The decoding procedure can be summarized as follows:

1. Read the encoded image, convert the intensity values into Gray code and perform bit-plane decomposition.
2. Use the fixed threshold for the complexity, α_{th}, and mark up each 8x8 block in the bit-plane image with a '1' if it is complex.
3. Retrieve the value of the number of files embedded in the encoded image, n, from the first byte of the first complex 8x8 block. Repeat steps 4-6 'n' times.
4. Retrieve the next 4 complex blocks. The first three blocks contain the file header and the fourth block contains the conjugation map for this header (3 bits of this 63 bit sequence). If the top-left bit of this fourth block is '1', it means that the block was conjugated and has to be conjugated again to retrieve the original information (Property 2 of the conjugation operator). Convert this 8x8 block into a 63 bit sequence (excluding the top-left bit) and use the first 3 bits of this sequence to reconstruct the first 3 blocks. If a bit is one then it means that the corresponding block has to be conjugated to retrieve the original information. These reconstructed blocks form the 24 byte file header–an 18 byte file name and a 6 byte file size.
5. The number of complex 8x8 blocks that form the file, N_f, will be *ceil(file size/8)* where the 'ceil' function rounds a value to the nearest integer towards infinity. Retrieve these N_f blocks and their corresponding conjugation map (which would

have been embedded as the next *(ceil(N_f /63))* complex blocks after these N_f blocks) and reconstruct the original file sequence, using the same procedure as for the file header.

6. Save the recovered file either under the original name, which is the 18 byte 'file name' parameter of the file header, or under a different name, to avoid destroying the original file.

2.7 Calculation for the Total Overheads

Although the exact overhead will depend on the exact size of the file, an approximate estimate can be easily made. Let 'n' be the number of files to be embedded, N_i, i = 1, 2...n, be the size of each file and C_i be the number of bytes added to the conjugation map for each file i. For every 8 bytes in file i, one bit is added to the conjugation map and every 63 conjugation map bits form 8 bytes of conjugation map info (including the 1 bit for the conjugation info of the conjugation map blocks). Thus, in effect, 8 bytes of conjugate map info are added for every 63x8 bytes of the data file. Apart from this, an 8 byte overall header (OH), a 24-byte file header (FH_i) for every file i, and an 8 byte conjugation map for each file header are added. Let F_i be the size of each file. Putting this together,

$$\text{Total overhead} = \text{OH} + \sum_{i=1}^{n} FHi + 8*n + \sum_{i=1}^{n} Ci \tag{2.3}$$

$$\text{where} \quad Ci = \frac{Fi * 8}{63 * 8} \Rightarrow Ci = \frac{Fi}{63}. \tag{2.4}$$

Substituting for FH and OH and putting 2.2 in 2.1 we get,

$$\text{Total overhead (in bytes)} = 8 + 32 * n + \sum_{i=1}^{n} \frac{Fi}{63}. \tag{2.5}$$

The total overhead will be minimum when the number of bytes in the file is an exact multiple of 63. When a single large file is embedded, instead of a number of smaller files, the overhead tends to be of the order of (1/63)-rd of the file.

2.8 Variants and Suggestions for Improvement

While the basic BPCS encoder/decoder, described in sections 2.1 to 2.7, works well, there are some subtle nuances that could be added to improve its performance.

1. The resource blocks are embedded into complex sections of the CGC bit-planes. The choice of the order of bit-planes plays a significant role in keeping the encoded image unsuspicious. It is important that the embedded blocks be spread over all the 3 color planes evenly and that they do not modify any particular color plane abnormally. This rules out the traditional ascending or descending order of sequences ([1, 2 ... 24] or [24, 23 ... 1]) as they modify one color first before going to the next (red first in the former and blue first in the latter). It was found that the following order, [24, 16, 8, 23, 15, 7, 22, 14, 6 ... 17, 9, 1], gives the best results, both in terms of encoding speed and PSNR. This ensures that complex sections in the LSB planes are replaced before the MSB planes are touched and that no color plane is preferentially encoded.
2. The choice of the threshold for the complexity measure, α_{th}, controls two complimentary parameters – the maximum data that can be hidden and the distortion in the encoded image. The distortion of the original image due to embedded data is measured in terms of its Peak Signal to Noise Ratio (PSNR), given by:

$$PSNR = 10\log(\frac{1}{NMSE}) \tag{2.6}$$

where the Normalized Mean Square Error (NMSE) is given by

$$NMSE = \frac{1}{255*255*M*N*D}\sum_{l=1}^{D}\sum_{k=1}^{N}\sum_{j=1}^{M}(Ioriginal(j,k,l) - Idistorted(j,k,l))^2 . \tag{2.7}$$

Where $I_{original}$ is the original image, $I_{distorted}$ is the distorted, encoded image, MxNxD is the dimension of the image (Rows x Columns x Depth), and 255x255 is the maximum square intensity difference possible.

The value of α_{th} can be made arbitrarily small to accommodate more resource files, but that would cause the encoded image to look heavily distorted and would defeat the very purpose of steganography. Again, if α_{th} is kept high (very close to 0.5), the maximum amount of data that can be stored in the image is reduced drastically. Hence, α_{th} is fixed at 0.3 to strike a compromise between the 2 complimentary parameters.

A better way to determine α_{th} would be to fix it adaptively instead of using a fixed threshold for all the bit planes. Since the three lower bit-planes (LSB planes 8, 7, and 6) are invariably complex and don't have a significant bearing on the final distortion, a lower threshold can be fixed for these planes while progressively increasing the thresholds in the higher, more significant, bit-planes. In [7], the author suggests a method for varying the threshold for another complexity measure. It turns out that a similar method for fixing α_{th} gives significantly better results, not only in terms of PSNR but also in terms of the perceptible quality of the encoded image. For any bit-plane i of a particular color, α_{th}^{i} is fixed as:

$$\alpha_{th}^{i} = \begin{cases} \bar{\alpha} - (i-1).\sigma_{\alpha} & 1 \le i \le 5 \\ 0.0 & 6 \le i \le 8 \end{cases} \tag{2.8}$$

where $\bar{\alpha}$ and σ_{α} are the mean and standard deviation of α for all possible 8x8 blocks. The $\bar{\alpha}$ and σ_{α} values were calculated by fitting a Gaussian curve for the α values for 50,000 randomly generated 8x8 blocks were 0.5 and 0.0473 respectively. Using these values, the α_{th} for the various bit-planes, for each color R, G or B, is given by, [0, 0, 0, 0.3108, 0.3581, 0.4054, 0.4527, 0.5], from MSB plane to LSB plane.

3. Before embedding the resource blocks into the base image, the resource files could be passed through a lossless entropy encoder like an LZW encoder or an Adaptive Arithmetic Encoder. If the resource files were text files or MS Word documents, the lossless encoders provide a significant compression of the data. Apart from this, the lossless encoders also make the distribution of characters in the resource file

significantly random by eliminating redundancies, and hence more complex. Since these blocks would already be complex, they would not need to be conjugated, and hence would not leave a detectable pattern that a long sequence of blank spaces in the resource file could leave.

With all these suggested improvements in place, there is still scope for increasing the data storage capacity and the PSNR of the encoded image. These can be achieved by inflicting some fundamental changes to the process explained in this chapter, including using new measures for calculating the complexity and some new ways to reduce the overheads. These modifications form the crux of the discussion in the following chapter.

CHAPTER III
NEW COMPLEXITY MEASURES

The complexity measure, α, discussed in chapter II is probably not the best way to describe the complexity of an image (or a segment of the image). There are many cases where a really simple and regular pattern may return a high α value and may be wrongly deemed complex. In this chapter, the shortcomings of the α measure are discussed and new complexity measures which produce significantly better results when used with the BPCS scheme are suggested.

3.1 Shortcomings of the α measure

Consider the patterns shown in Figure 3.1. Both these patterns would qualify as a complex block if the value of α_{th} is fixed at its usual value of 0.3. It is readily seen that the block shown in Figure 3.1(a) is a regular pattern or alternating black and white rows. If this block were to be replaced by a noisy block, the replacement is sure to make some noticeable changes to the region.

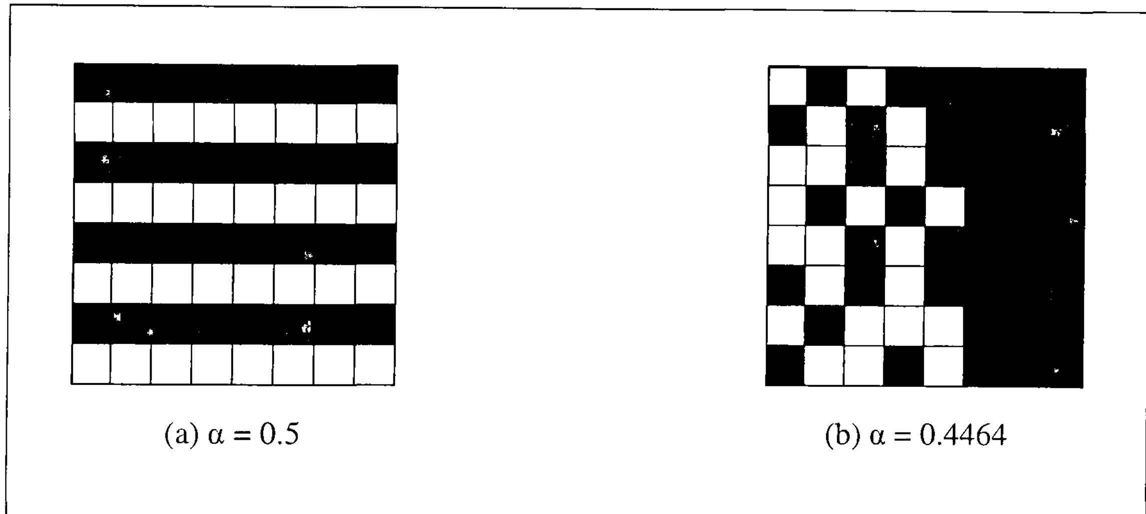


Figure 3.1 Two different blocks that are not actually complex

Blocks, like the one shown in Figure 3.1(b), occur frequently in regions that lie on the boundary of noise-like and informative regions, and modifying these regions may result in ruining the quality of the border regions and edges and making them prone to suspicion. Since these two appear to be completely different kinds of blocks, a complexity measure that works for one kind may not work for the other. In [7], the author proposes two different complexity measures Beta (β) and Gamma (γ) that can be used in combination to overcome the disadvantages of using α alone.

3.2 The β Complexity Measure

This β complexity measure is different from the measure based on the number of connected components discussed in chapter II. This measure is based on the irregularity of the runs of black and white pixels (in the binary bit-plane images) along each row and column of an 8x8 block and helps in overcoming the disadvantage of applying the α measure to blocks such as the one in Figure 3.1(a). The logic behind this method is that if the distribution of the black and white pixels in a block has a regular periodicity, then the block should not be used for embedding. The β value of a block is calculated based on the histogram of both black and white pixels along each row and column of the block.

Consider the arrangements of black and white pixels shown in Figure 3.2.

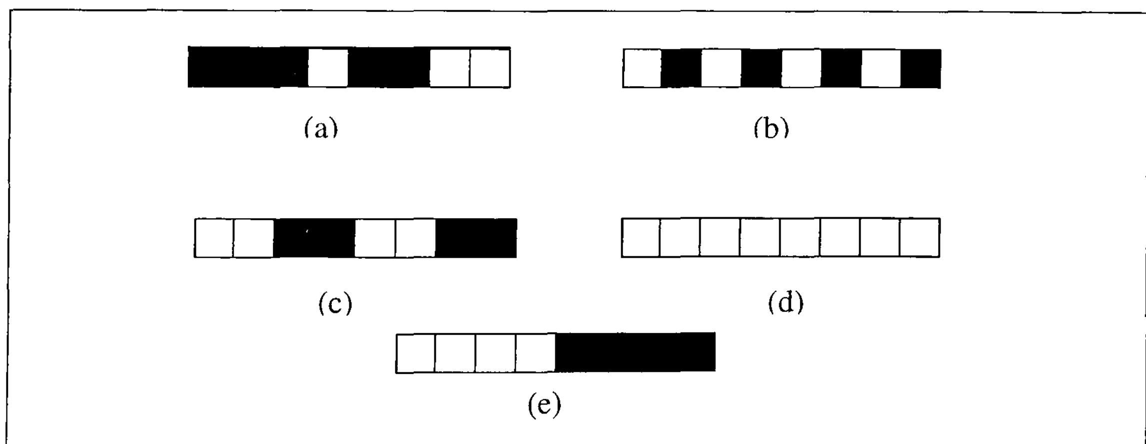


Figure 3.2 Some typical binary pixel sequences

While patterns like the one shown in Figure 3.1(a) are rather uncommon in practice, the runs of black and white pixels shown in figure 3.2 are mundane. Figure 3.2(a) shows one possible row of an 8x8 block that consists of: one run of three black pixels, one run of one white pixel, one run of two black pixels and one run of two white pixels. Note that only the maximum length of the run is considered. For example, a run of four white pixels should not be accounted for as two runs of two white pixels or four runs of one white pixel. The histogram of the run-lengths will be represented as,

$$h[1] = 1,\ h[2] = 2 \text{ and } h[3] = 1, \tag{3.1}$$

here, h[i] is the frequency of runs of i pixels, either black or white.

The inequality of the run-length distribution in a binary sequence (along a particular row or column) is represented by factor h_s, given by

$$h_s = -\sum_{i=1}^{n} h[i]\log_2 p_i, \tag{3.2}$$

where
$$p_i = \frac{h[i]}{\sum_{j=1}^{n} h[j]} \tag{3.3}$$

If a sequence is formed by a periodic arrangement of black and white pixels, as in the case of the sequences of Figure 3.2(b), (c), (d) and (e), its h_s value becomes zero. For each of those sequences, the probability of a particular run, p_i, becomes one, as there are sequences of only one run-length for each of the sequences (3.2(b) has 8 runs of length 1, 3.2(c) has 4 runs of length 2 each and so on). For the sequence of Figure 3.2(a), h_s is 6.

The h_s values are normalized so that they lie in [0, 1], and the normalized h_s is denoted as $\hat{h}_s$. The normalization factor used is the highest possible value for h_s, 6.8548. This is obtained for a sequence that has $p_1 = 3/5$, $p_2 = 1/5$ & $p_3 = 1/5$. The normalization factor can also be found by generating random black and white sequences of length 8, calculating the h_s for each of those sequences and finding the maximum value of those.

Let r_i and c_i, for i = 1, 2, ... 8, be the i-th row and column of an 8x8 block. The run-length irregularity, β, of the block is defined as the minimum of the average of $\hat{h}_s$ values along the rows and the columns i.e.

$$\beta = \min\{\overline{\hat{H}_s(r)}, \overline{\hat{H}_s(c)}\} \qquad (3.4)$$

Where, $\hat{H}_s(r) = \{\hat{h}_s(r_1), \ldots, \hat{h}_s(r_8)\}$ and

$$\hat{H}_s(c) = \{\hat{h}_s(c_1), \ldots, \hat{h}_s(c_8)\} \qquad (3.5)$$

and $\overline{X}$ represents the mean of all the elements in the vector X.

Figure 3.3 shows three 8x8 blocks and their corresponding α and β values. The block shown in figure 3.3(a) is the most complex block possible according to the α complexity measure, but if the β complexity measure were used, the complexity of this block becomes zero as there is no irregularity in the runs along the rows and columns of the block. The block in figure 3.3(b) is the same as the one in 3.1(b), and as mentioned earlier, a good complexity measure is supposed to return a small value for this block, unlike the α and β measures. Figure 3.3(c) shows an 8x8 block for which the β measure completely fails.

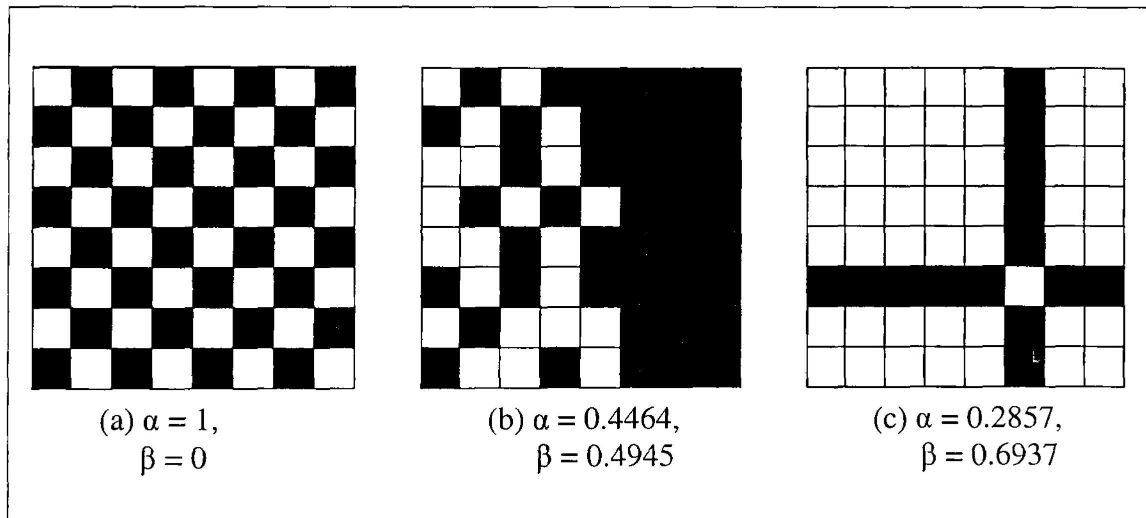


Figure 3.3 Three 8x8 blocks with α and β values

The block shown in Figure 3.3(c) is a typical example of a simple block for which the β measure returns a high value. In fact, even the α measure returns a value that reflects the non-complexity of the block more effectively. This calls for another complexity measure, which, when combined with the β measure, forms an effective measure to gauge the complexity of a block. This measure, γ (gamma), based on the noisiness of borders in a block, is discussed in the next section.

3.3 The γ Complexity Measure

If a resource file is embedded in regions on the boundary between noisy and informative regions (i.e., the regions that lie along the boundary of an object and the background), then the noisy regions tend to grow after embedding because they would be replaced by a completely noisy block. This results in making the changes to the blocks noticeable. Hence, such blocks should be avoided for embedding data. The γ complexity measure is designed to return low values for these blocks which lie on the boundary of informative and noisy regions, so that they will not be used for embedding the resource files. If the border noisiness, γ, of a block is large enough, it cannot be on the boundary of a noisy and an informative region.

The γ measure is defined based on the difference between adjacent rows and columns as shown in Figure 3.4. The number of pixels at which two adjacent rows, r_i and r_{i+1}, differ is the number of ones in $(r_i \oplus r_{i+1})$ where $\oplus$ represents a bit-wise XOR of corresponding pixels in the two rows. If r_i and c_j are the i-th row and j-th column of an nxn block, i, j = 1,... 8, the border noisiness, γ, of the block is defined as

$$\gamma = \frac{1}{n}\min\{\ E_f(P_x(r)), E_f(P_x(c))\}\ \text{, where} \qquad (3.6)$$

$$P_x(r) = \{\rho(r_1 \oplus r_2),\ldots, \rho(r_{n-1} \oplus r_n)\} \text{ and}$$

$$P_x(c) = \{\rho(c_1 \oplus c_2),\ldots, \rho(c_{n-1} \oplus c_n)\}. \qquad (3.7)$$

$\rho(x)$ is the number of ones in a binary sequence x, and

$$E_f(X) = \left(1 - \frac{V(X)}{\max\{V(X)\}}\right)\overline{X} \quad (3.8)$$

$$X = \{x_1, \ldots, x_m\}, \quad m = n - 1$$

$$V(X) = \textit{Variance of } X, \textit{ and } \overline{X} = \textit{Mean of } X.$$

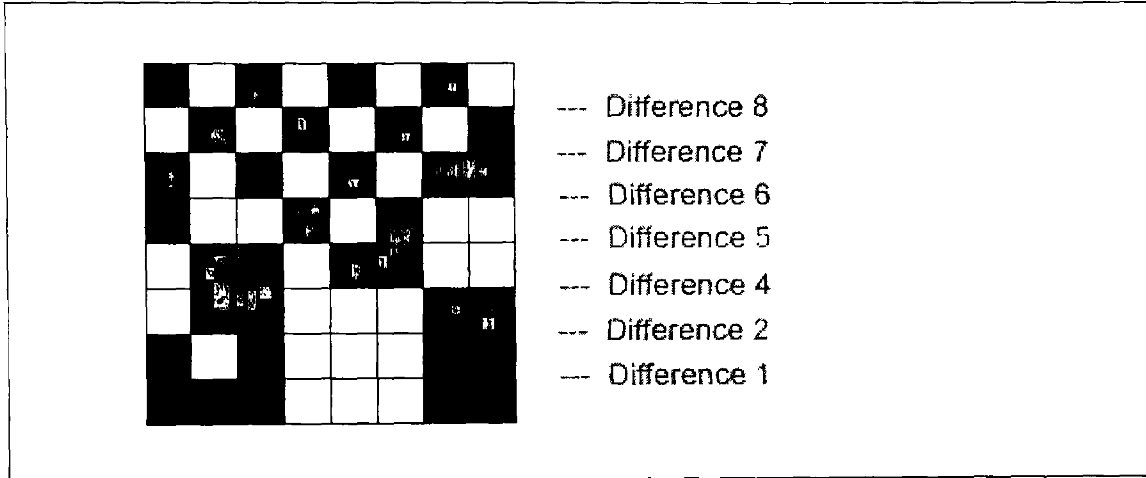


Figure 3.4 Difference between adjacent rows for an 8x8 block

For an 8x8 block, $P_x(r)$ and $P_x(c)$ are each sequences of 7 numbers, the black and white borders counted for every pair of adjacent rows and columns. E_f is a weight calculated based on the variance of these two sequences and it lies in the range [0, n]. The variance, V(x), is the second moment of the sample about the mean (sample variance), and the maximum value of the variance is found by calculating the maximum variance over 50,000 random 8x8 blocks and is found equal to 15.6735 (obtained for the sequence [8, 8, 8, 8, 0, 0, 0]). Taking the minimum of the two E_f's, one calculated along the rows and another along the columns, helps in excluding those blocks which lie on a horizontal boundary or a vertical boundary but not both. Due to the normalizing factor, n, in equation 3.6, γ lies in [0, 1]. A value of γ close to zero means that the block is not complex and a large γ value (close to 1) means that the black and white pixels are well-distributed throughout the block and hence the block is complex.

Figure 3.5 shows some 8x8 blocks and their corresponding α, β and γ values. Figure 3.5(a), (b) and (c) repeat from Figure 3.3. In fact, Figure 3.5(a) is a typical example of why the γ measure alone cannot be used to evaluate the complexity of a block. The black and white pixels in this block form a regular pattern and cannot be called complex. Only the β measure reflects the 'non-complexity' of the block while the α and γ measures indicate the block to be perfectly complex.

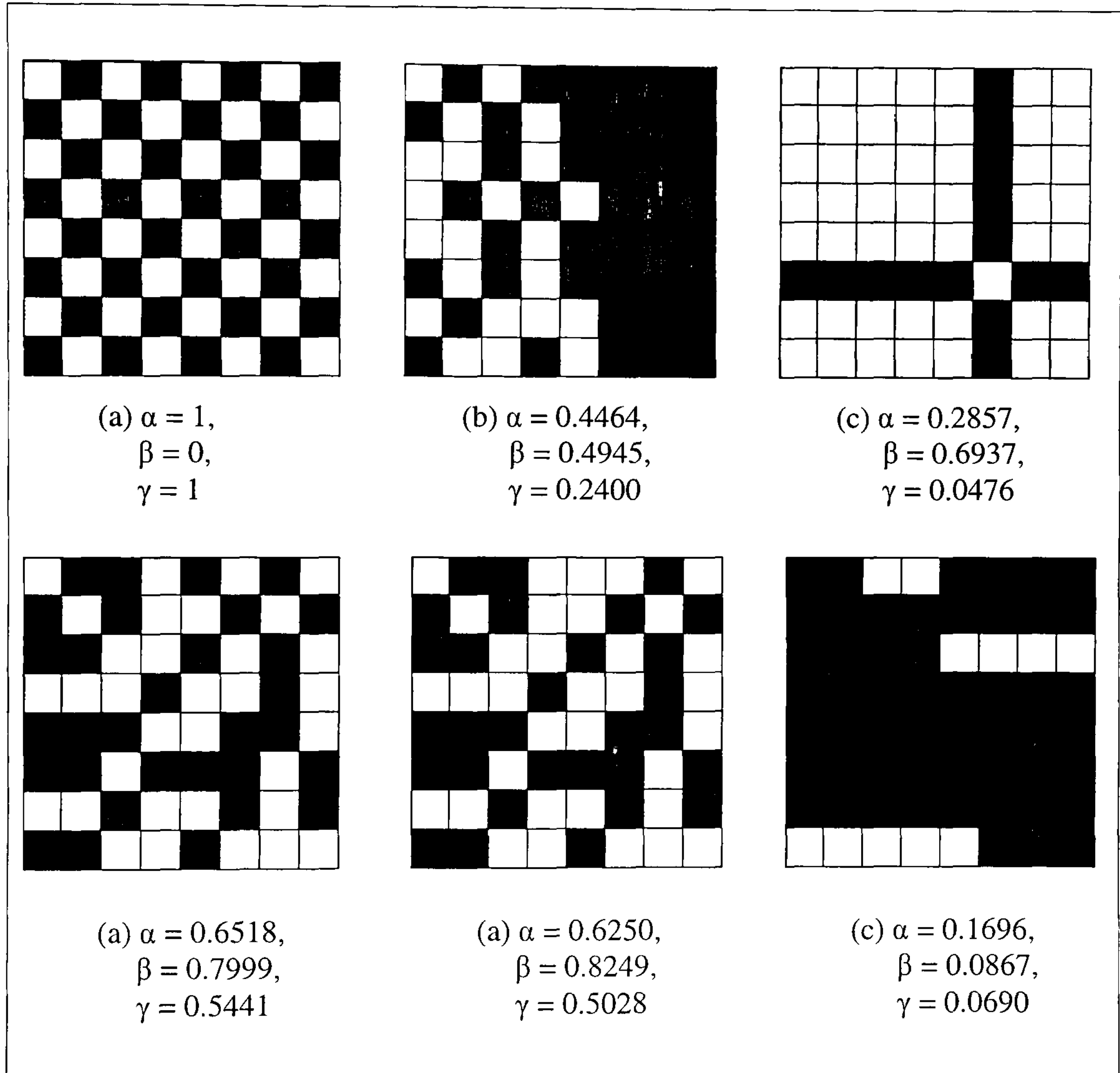


Figure 3.5 Blocks with various complexity values (α, β, γ)

3.4 Threshold Values

As in the case of the α complexity measure, a threshold has to be fixed for both β and γ (β_t and γ_t,), and a block B is said to be complex if and only if

$$\beta(B) \geq \beta_t \quad \text{and} \quad \gamma(B) \geq \gamma_t. \tag{3.9}$$

Again, the threshold values could be kept the same for all the planes or varied adaptively. The threshold values are determined by the mean (μ) and standard deviation (σ) of the distributions of β and γ, shown in Figure 3.6, and the order of planes. For any bit-plane i, of a particular color (say red), β_{th}^i and γ_{th}^i are fixed as:

$$\beta_{th}^i = \begin{cases} \bar{\beta} - (i-1)\sigma_\beta & 1 \leq i \leq 5 \\ 0.0 & 6 \leq i \leq 8 \end{cases} \tag{3.10}$$

$$\gamma_{th}^i = \begin{cases} \bar{\gamma} - (i-1)\sigma_\gamma & 1 \leq i \leq 5 \\ 0.0 & 6 \leq i \leq 8 \end{cases} \tag{3.11}$$

Where, i = 1 is the most significant bit-plane and i = 8 is the least significant bit-plane. Figure 3.6 shows the distribution of β and γ values over 100,000 randomly generated 8x8 blocks.

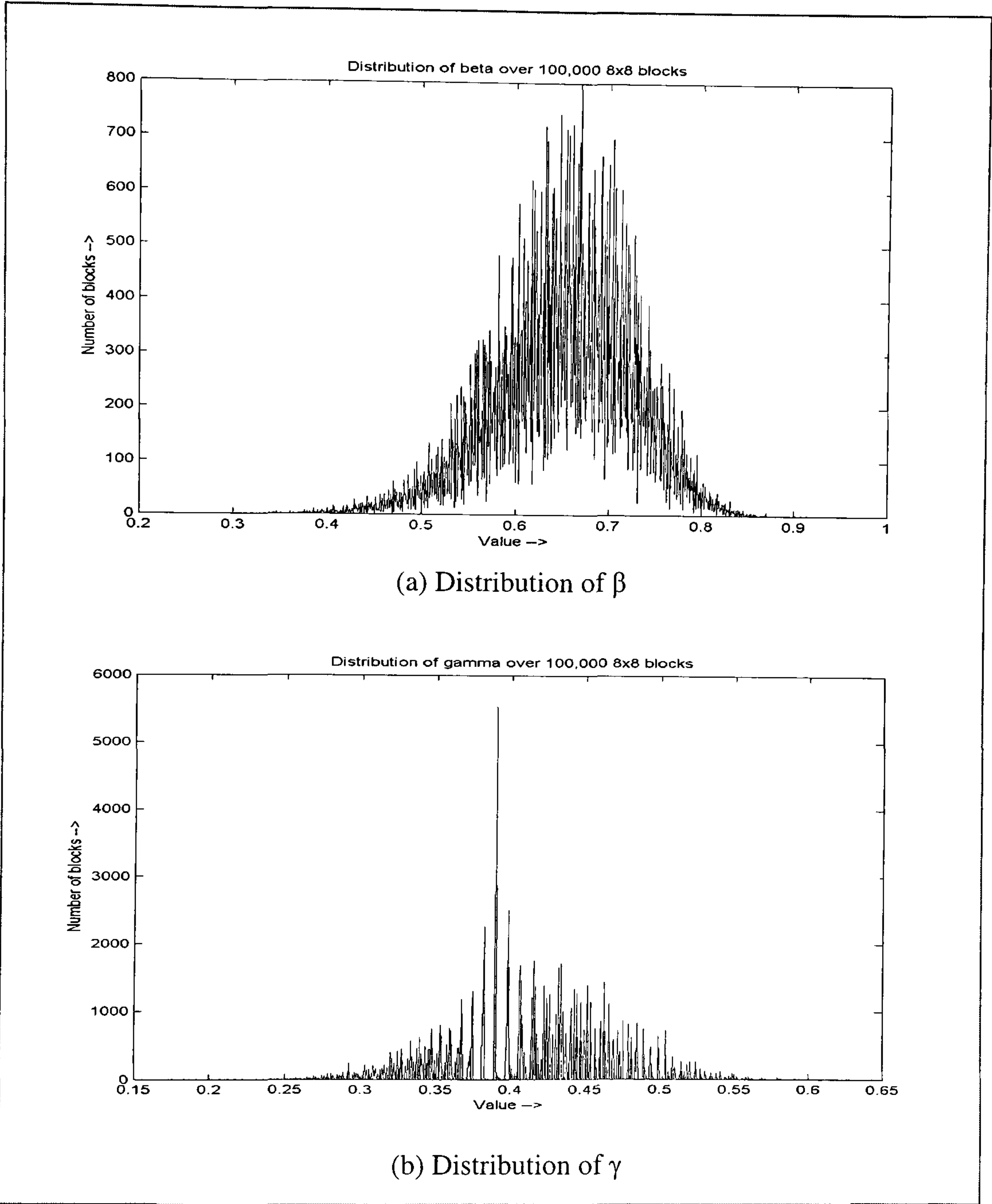
The mean and standard deviation of β and γ were found by fitting a Gaussian curve to the distributions shown in Figure 3.6(a) and (b) and they are as follows:

$$\bar{\beta} = 0.653\,;\ \sigma_\beta = 0.0728\,;\ \bar{\gamma} = 0.408\,;\ \sigma_\gamma = 0.0540; \tag{3.12}$$

The threshold for each plane can be fixed by substituting these values in equations 3.10 and 3.11. The threshold values determined for planes 1 to 8 (Red) are reflected onto planes 9 to 16 (Green) and planes 17 to 24 (Blue).

The default threshold values work well for most natural images, but the method makes for easy *steganalysis*, which refers to detecting and deciphering hidden data. For example, if an image region were all zeros (R = 0, G = 0, B = 0 corresponding to plain black) or all ones (R = 255, G = 255, B=255 corresponding to plain white), the three LSB planes of each of the three colors, R, G and B, will still be used for embedding data, as their thresholds are zero, and the embedded data shows

up as visible discrepancy on the perfect original image. For such images, a simple fallback routine could be used, where the thresholds for planes 6, 7 and 8 are not made zero but reduced to arbitrarily low values.



(a) Distribution of β

(b) Distribution of γ

Figure 3.6 Distribution of β and γ over 100,000 randomly generated 8x8 blocks

3.5 M-Sequences

Most of steps involved in encoding resource files into an image and decoding those files, using the β and γ complexity measures, are the same as ones used in the original BPCS scheme explained in Chapter 2. Just as in the original case, an RGB image is first read, its intensity values converted from PBC to CGC followed by bit-plane decomposition of the CGC planes, and the complexity of each exclusive 8x8 block determined according to the β and γ complexity measures. Once the complexity of the blocks has been determined, the complex blocks are ready to be replaced by the resource blocks. As in the case of the original scheme (of Chapter 2), care has to be taken to ensure that the resource blocks themselves are complex enough for the decoder to distinguish the block as a block with hidden information. However, the process of making the 'simple' resource blocks 'complex' is not as simple as conjugating it. It is difficult to define conjugation operations for β and γ as they are defined in a far more complicated way than α. The M-sequences provide a more generic solution to the problem of making simple resource blocks complex.

Shift register sequences having the maximum possible period for an r-stage shift register are called *maximal length sequences* or *M-sequences* [8]. The M-sequences are special cases of pseudo-random noise sequences (PN sequences) and can be implemented in much the same way using Linear Feedback Shift Register (LFSR) generators [9]. Figure 3.7 shows an m-stage linear feedback shift register where the square blocks represent a one-bit register (or flip-flop). The weight g_i for any given tap i, where i = 1, 2 ..., m-1, is either 0, meaning no connection, or 1, meaning it is fed back. The weights g_0 and g_m are always 1. In fact, g_m is not a feedback connection but the serial input to the shift register. Any LFSR can be represented as a polynomial of variable X, called the generator polynomial, G(X), as

$$G(X) = g_m X^m + g_{m-1} X^{m-1} + \ldots + g_2 X^2 + g_1 X + g_0 . \quad (3.13)$$

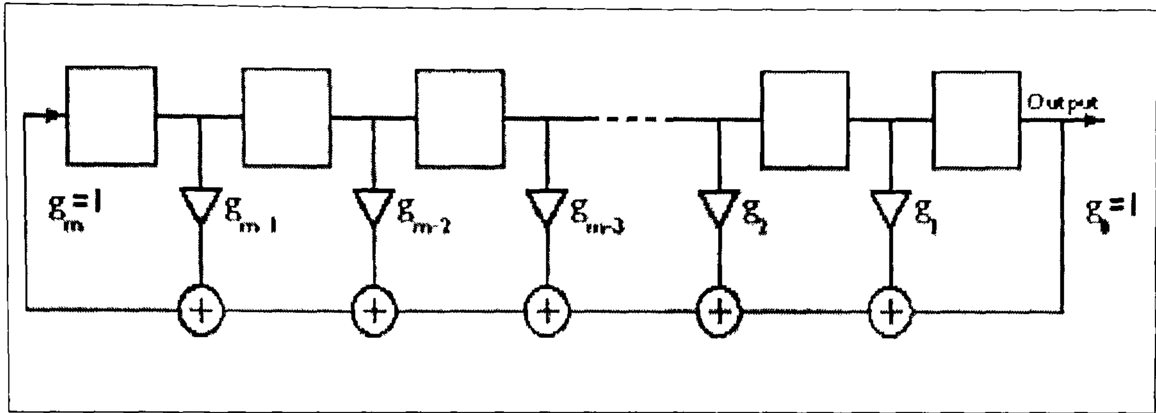


Figure 3.7 Fibonacci implementation of an m-stage Linear Feedback Shift Register

The reason for choosing g_m to be one is evident from equation 3.13. If g_m were not one, the order of LFSR would not be m. The m-bit number, with each bit corresponding to the state of each of the m registers, which indicates the initial state of the LFSR is called the *seed.* The weights g_i could be anything, and each combination of the g_i's produces a different sequence of m-bit pseudo-random numbers. However, they may not produce all possible m-bit numbers before starting to repeat the sequence. If the weights g_i, and in turn the polynomial G(X), are carefully chosen, the LFSR can be made to generate each of the 2^m possible m-bit numbers before starting to repeat them, or in other words, they can also be said to generate maximal length sequences. Such a polynomial is called a *primitive polynomial.* A polynomial G(X), of degree m, is said to be primitive if:

a. G(X), cannot be factored (i.e., it is prime), and
b. G(X) is a factor of $X^N + 1$, where the *length* of the sequence, N, is $2^m - 1$.

LFSR's built using primitive polynomials have several interesting properties, the most important being its ability to generate 2^m-1 of the 2^m possible m-bit numbers, in a random order, before starting to repeat the numbers. The only condition is to initialize the LFSR using a non-zero seed. The one remaining state is the trivial or all zero state, which is generated with an all-zero seed. The usefulness of the M-sequence stems from this "non-repeating" property (over a period of 2^m-1) of the

primitive polynomials. Equation 3.14 is an example of a primitive polynomial, G(X), defined

$$G(X) = X^{64} + X^{6} + X^{2} + X + 1. \tag{3.14}$$

This is a primitive polynomial of degree m = 64. Given a non-trivial seed (all registers in the LFSR are not uniformly zero), an LFSR based on this polynomial as its generator polynomial will make the 64-bit shift register emulate all of the possible 2^{64}-1 states (except the trivial state) before the repeating the seed. The actual maximal length sequence or the M-sequence is obtained by collecting the 2^{64}-1 output bits, i.e. the bit from the right-most register in figure 3.7, for each of the 2^{64}-1 exclusive states of the LFSR. Any N-tuple, where N equals the degree of the generator polynomial (N = 64 in this example), of the M-sequence at phase i, corresponds to a particular state of the LFSR. This N-tuple is represented as,

$$m_i^N = (m_i, m_{i+1}, \ldots, m_{i+N-1}). \tag{3.15}$$

This N-tuple is referred to as an M-block thus making the M-sequence a stream of M-blocks.

3.5.1 M-sequence Block Stream Conversion (MBSC)

Since the complexity is calculated for each 8x8 block of the base image, the resource files are divided into streams of 'k' 64 bit (8 byte) resource blocks, say R_0, R_1, ..., R_{k-1}. An M-sequence is generated using the LFSR shown in figure 3.7 using the generator polynomial of equation 3.14. Let $m_i^N, m_{i+1}^N, \ldots, m_{i+(k-1)N}$ represent 'k' N-tuple M-blocks, where N = 64, of the M-sequence, starting at phase i. The basic idea is to perform a bit-wise XOR between the stream of k resource blocks and the stream of k M-blocks such that

$$\beta(R_i) \geq \beta_t^{\pi(B_j)} \quad \text{and} \quad \gamma(R_i) \geq \gamma_t^{\pi(B_j)}, \tag{3.16}$$

here $\pi(B_j)$ represents the plane that contains the complex 8x8 base image block, B_j , and R_i is the resource block that is to be embedded in B_j, for i = 1, 2, ..., k-1.

Since the number of possible N-tuples (2^{64}-1) is far greater than the number of blocks in a stream, k, which is usually fixed anywhere between 65 and 1000, it is not too difficult to find a phase 'p' such that the blocks

$$R_0 \oplus m_p^N, R_1 \oplus m_{p+N}^N, \ldots, R_{k-1} \oplus m_{p+(k-1)N}^N$$

satisfy equation 3.16. Also, since no two N-tuples in an M-sequence are the same, instead of taking 'k' exclusive N-tuples, an overlap of d, where d is a positive integer, could be allowed and the M-block stream redefined as

$$m_p^N, m_{p+d}^N, \ldots, m_{p+(k-1)d}^N. \tag{3.17}$$

Fixing d to be 1 eliminates the need to generate and store the extremely long M-sequence because the N-tuple at each stage can be generated based on the previous state of the shift register and the generator polynomial. The MBSC was implemented with k = 200 and d = 1.

Once the k resource blocks have been made complex with respect to the β and γ complexity measures, they can be embedded in k successive complex blocks in the bit plane base image. However, the decoder needs some additional information to decode the resource blocks. Since the k complex resource blocks were obtained by XORing the original resource blocks with K N-tuples of the M-sequence, the initial seed of the LFSR, the phase p, and the generator polynomial are the only information the decoder needs for extracting the resource blocks. The K N-tuples are chosen with the first N-tuple starting at phase p and each subsequent N-tuple starting from bit 'd' of the previous N-tuple. The generator polynomial can be found out from the first 2N-tuple (i.e. the seed and the next N-tuple). This 2N-tuple is called *the M-sequence key.* The phase is specified by the N-tuple m_p^N. This N-tuple is called the *phase key.* The M-sequence key is embedded on two 8x8 blocks (since it is 16 bytes) before any of the resource blocks are embedded and the phase keys are embedded on one 8x8 block. There is one phase key for every k block resource file stream. Since the M-sequence key forms the first two blocks of information to be embedded, it is hidden in the least significant bit plane where the threshold complexity is zero. Hence the

complexity of the M-sequence key is not a big issue. However, this is not the case for the phase keys. The phase keys could be embedded anywhere on the base image, including the most significant bit-plane. Hence it is important to make sure that the phase keys themselves are complex. If a particular phase does not yield a complex N-tuple, the next phase is tried till a complex phase key can be found and only then is the MBSC attempted, starting from this phase.

It is useful to have a seed with reasonable complexity, with respect to the β and γ complexity measures, for the M-sequence generator. This seed, in turn, is generated using a 6-bit LFSR, with a generator polynomial $G(X) = X^6+X^5+X^2+X+1$ and seed 010101, which produces a 64-bit M-sequence. The implementation of the LFSR shown in figure 3.7 is called the Fibonacci implementation and is the one used for the MATLAB implementation. It is different from another implementation of the LFSR called the Galois implementation, where the contents of the shift register are modified at every step by a binary-weighted value of the output stage.

An important difference between using the α measure and the β and γ measures is that in the case of the former only the non-complex resource blocks are made complex by conjugating, while in the latter case, all the resource blocks are subjected to the MBSC scheme, regardless of whether they are simple or complex.

3.6 File Headers and Overall Headers

Just as in the original case, some additional information about the file, like the file size, file name etc., must to be embedded with the file in order to perfectly reconstruct the files at the decoder. The conventions explained in section 2.4 for the original BPCS scheme, are used here as well. The *file header (FH)* is again 24 bytes, with 18 bytes for the file name and 6 bytes for the file size, and is attached to every resource file that is to be embedded. The *overall header (OH)* is 8 bytes with the first byte containing the number of resource files embedded in the image and the rest of the bytes reserved for future use. With these headers in place, the files are all ready to be embedded into the base image, of course with all the necessary pre-processing.

3.7 Encoding Procedure

The steps involved in encoding using the new complexity measures can be summarized as follows:

1. Read the image, convert the intensity values into Gray code and perform bit-plane decomposition.
2. Use equations 3.10 and 3.11 to determine the complexity thresholds, β_{th} and γ_{th}, for each plane. For each exclusive 8x8 block in the bit-plane image, calculate the complexities β and γ. If the complexities of the block satisfy equation 3.9, mark the 8x8 block to be complex. This is done by marking up a '1' on another matrix, called the *complexity matrix*, whose dimensions are one-eighth the bit-plane base image along the rows and columns. A 512x512x3 image would have a 64x64x24 complexity matrix. The order of bit-planes followed for embedding is [24, 16, 8, 23, 15, 7, 22, 14, 6 ... 17, 9, 1]; as this ensures that the least significant bit-planes of each color are embedded before the more significant ones are touched.
3. Embed the 128-bit M-sequence key (2N-tuple) into the first two embeddable 8x8 regions (complex regions) of the bit-plane image. The first N-tuple of this 2N-tuple is the seed for the 64-bit M-sequence generator (LFSR) and is the M-sequence generated using a 6-bit LFSR with seed '010101'. The second N-tuple is simply the next phase of this initial seed.
4. Get the number of resource files to be embedded, n, make it the first byte of the overall header (OH), and make this 8x8 block the first block of the resource file stream. Repeat steps 5 to 8 'n' times or till the maximum embeddable capacity is reached.
5. Read in the resource file and form it into a sequence (or vector) of ASCII values. Pad the sequence so that the number of bytes in the sequence is a multiple of 8. This is done because the encoder embeds blocks 8 bytes at a time. Attach a 24-byte file header (FH) containing the file name and file size to this sequence. Then attach this whole sequence to the resource file stream and make the number of bytes in the resource file stream a multiple of 8xk bytes, where k is the number of 8x8

blocks for which a unique phase key is generated, by padding the stream with zeros. Repeat steps 6 and 7 till the whole resource stream has been embedded.

6. Read in k blocks, R_0, R_1,..., R_{k-1}, of 8 bytes (64-bits) each, from the resource file stream. Find a phase 'p' on the M-sequence, generated by the 64-bit LFSR using the generator polynomial in equation 3.14, such that k consecutive N-tuples starting at phase p, $m_p^N, m_{p+d}^N, \ldots, m_{p+(k-1)d}^N$, when XORed with the k corresponding resource blocks, yield a complex block stream. As a result, the following blocks are all complex with respect to the β and γ complexity measures: $R_0 \oplus m_p^N, R_1 \oplus m_{p+N}^N, \ldots, R_{k-1} \oplus m_{p+(k-1)N}^N$. The overlap between consecutive N-tuples is specified by d. These k blocks are called the *complex resource blocks* and the process of converting the resource file stream into complex blocks is called *M-sequence Block Stream Conversion* (MBSC).
7. Embed the phase key (the N-tuple starting at phase p) and the k complex resource blocks in (k+1) consecutive complex 8x8 regions in the bit-plane base image. These (k+1) blocks can be found out by finding the ones in the complexity matrix.
8. Clear the resource file stream.
9. Put back the 24 bit-planes together to form 3 color planes, R,G and B, and convert from CGC to PBC and save the image, either under a new name or under the same name as its original to eliminate suspicion. This is the encoded image.

3.8 Decoding Procedure

Again, the function of the decoder is to reverse the modifications done on the resource file and retrieve them without any errors. The encoding module ensures that those blocks that were complex in the bit-plane base image are again complex in the encoded image, and the decoder can tread through these complex blocks and decode the relevant information. The decoding procedure can be summarized as follows:

1. Read the encoded image, convert the intensity values into Gray code (CGC) and perform bit-plane decomposition.

2. Determine the complexity thresholds, β_{th} and γ_{th}, for each plane using equations 3.10 and 3.11. For each exclusive 8x8 block in the bit-plane image, calculate the complexities β and γ. If the complexities of the block satisfy equation 3.9, mark the 8x8 block to be complex. This is done by marking up a '1' on the *complexity matrix*, in the same way as in the encoder. Follow the same bit-plane order as in the encoder to ensure that the blocks are decoded in the same order as they were encoded.
3. From the first 2 complex 8x8 blocks, decode the 2N-tuple M-sequence key. If the generator polynomial for the LFSR is varied for each time the encoder is used, this M-sequence key can be used to find out the generator polynomial.
4. The next complex block has the phase key needed to decode the first k blocks. This phase key is used to generate the next k m-blocks, $m_p^N, m_{p+d}^N, ..., m_{p+(k-1)d}^N$, that can be simply XORed with the next k complex blocks (read into a stream of k 64-bit blocks in a row major order) to retrieve the original resource blocks. The first byte of the first retrieved block is the number of embedded files, n. The next three blocks are the external file header (FH) inserted by the encoder, the first 18 bytes of which contain the file name and the last 6 bytes contain the file size. Use the file size parameter to recover the rest of the files, keeping in mind that every $(k+1)^{th}$ block is a phase key that is to be used to recover the following k blocks. The number of k-block streams, including the first k-block stream that contains the headers, over which the first file is stored is $ceil((filesize+4)/(8*k))$, where ceil(x) of a real number x is the smallest integer greater than x. Save the recovered file either under the original name, which is the 18 byte file name parameter of FH, or under a different name. Repeat step 5, (n-1) times.
5. Retrieve the phase key, from the next complex block, and use it to recover the next k blocks. The first three 8x8 blocks have the 24 byte FH. Again, use the file size parameter, which is the last 6 bytes of the file header, to recover the rest of the file. The number of k-block streams to be recovered to entirely recover the file is $ceil((filesize+4)/(8*k))$. Save the recovered file (similar to the first file).

3.9 Calculation for the Total Overheads

As explained in section 2.7 for the original BPCS technique, the exact overheads can be calculated based only on the exact size of the embedded files. However, an estimate of the overheads can still be determined. Let 'n' be the number of files to be embedded, N_i, i = 1, 2 ... n, be the size of each file. A one-time 8-byte overall header and a 24-byte file header, for each file, are added to the encoded image. For every k 8x8 blocks of each resource file, an 8-byte (one 8x8 block) phase key, p (say), is added and a one-time 16-byte M-sequence key, M (say), is added to the encoded image. Let F_i be the size of each file. Putting all these together,

$$\text{Total overhead} = OH + \sum_{i=1}^{n} FH_i + M + \sum_{i=1}^{n} K_i p \qquad (3.18)$$

K_i is the number of phase keys needed for file i and is given by,

$$K_i = \begin{Bmatrix} ceil\ ((F_i + 4)/k * 8) & \text{for } i = 1 \\ ceil\ ((F_i + 3)/k * 8) & \text{for } i = 2, 3 \ldots n \end{Bmatrix} \qquad (3.19)$$

Substituting the values for OH, FH, M and p in 3.18,

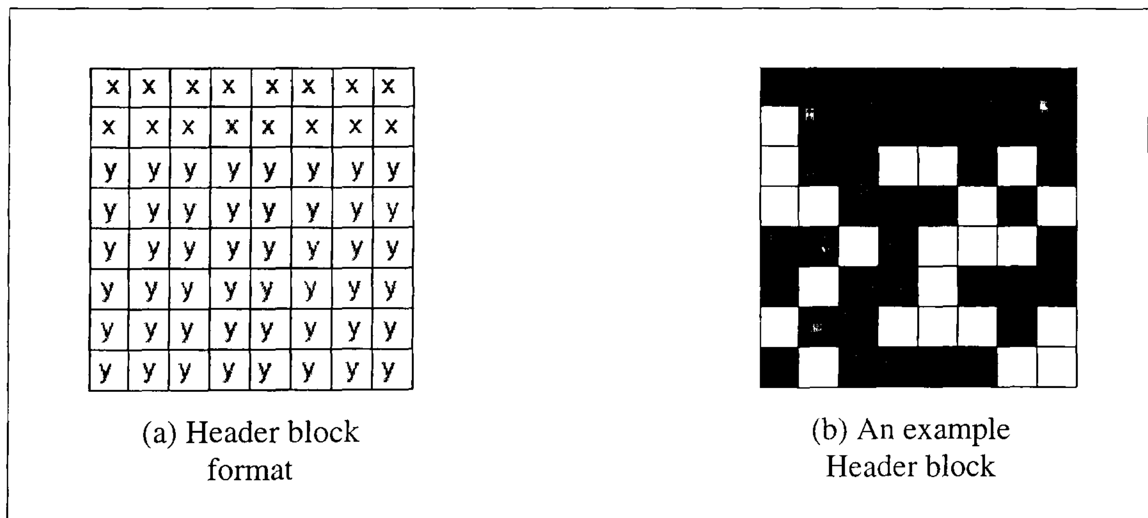
$$\text{Total overhead (in bytes)} = 24 + 24 * n + \sum_{i=1}^{n} (\mathrm{K_i} * \ 8) \qquad (3.20)$$

The total overhead will be minimized when the argument inside the *ceil* function in equation 3.19 is an integer. When a single large file is embedded, instead of a number of smaller files, the FH and OH tend to become very small in comparison to the phase key overhead. The total overhead then depends almost entirely on the factor k and is approximately equal to 1/k. For the value of k used in the implementation, k = 200, the overhead is approximately one byte for every 200 bytes of resource file. This is less than one third of the overhead for the original BPCS scheme, which was proved to be approximately equal to one byte for every 63 bytes of the resource block (see section 2.7).

3.10 Suggestions for Improvement

The biggest problem with the M-sequence method for making the blocks complex is that a phase key needs to be embedded once every k blocks. If k is too large then finding a phase of the M-sequence that produces k M-blocks that result in a complex resource block stream (of length k) becomes extremely time-consuming. This could be an even bigger problem for the higher bit-planes, where the threshold complexity is high. Consider a case where k is fixed at 1000. If a phase is found such that only 999 consecutive resource blocks can be converted into complex blocks, at this phase, the encoder has to restart the process of finding a phase and it could take a long time before 1000 consecutive M-blocks, to produce 1000 complex resource blocks, are found. However, the lower bit-planes (3 LSB planes for each color) have a zero threshold for β and γ, and hence any phase of the M-sequence would work. For these planes, k can be fixed arbitrarily large.

One solution would be to fix a different k for the lower bit-planes than for the more significant bit-planes. A more generic solution would be vary k adaptively and inform the decoder about it. An extra block could be embedded as a header to each k block stream, specifying the value of k. The difficult part, however, is to ensure that this header block is always complex. Figure 3.8 gives a solution for this problem.



(a) Header block format

(b) An example Header block

Figure 3.8 Format and example of the new block

In Figure 3.8(a), the pixels marked 'x' are the pixels (bits) used to embed the value of k. For this scheme to have a lower overhead than the original BPCS scheme, the value of k needs to be greater than 126 (approximately), because of the extra one block (8x8 block) overhead for each complex resource block stream. The encoder finds the first phase of the M-sequence, starting from a given seed, such that k lies in [127, 65535] (the upper limit results from using 2 bytes for k), which results in a complex resource block stream. Once the value of k has been found, the pixels marked 'y' in figure 3.8(a) are tweaked so that the complexity measures of this block are greater than the thresholds for the plane in which the block is to be embedded. An example, when k is 128, is shown in Figure 3.8(b). The last six rows in the 8x8 block have been adjusted so that the β value of the block is 0.7464 and γ value is 0.4232, which are greater than the thresholds for any plane. In fact, the values for the last six rows shown in Figure 3.8 (b) works for most values of k, and instead of scouting for completely random values, a template of values for the last six rows could be maintained and the random scouting for the pixel values could be used only when all the values in the template fail.

In summary, the β and γ complexity measures can be used to effectively describe the complexity of images. Combined with MBSC, these measures can be used with the BPCS scheme to hide large amounts of data in images. The effectiveness of the schemes presented in chapters II and III are compared, analyzed and interpreted in the following chapter.

CHAPTER IV

RESULTS AND CONCLUSION

In this chapter, the performance of the new method, presented in chapter III, is compared with the performance of the original BPCS scheme, presented in chapter II, and the preeminence of the former is substantiated with appropriate results. First, the original BPCS scheme was implemented as explained in sections 2.5 and 2.6. The results obtained were further improved by applying the methods suggested in section 2.8. The new complexity measures with MBSC were then implemented to overcome the shortcomings of the α measure used in chapter II. Finally, a Graphical User Interface (GUI) was developed to provide a user interactive interface to hide data files in images (See Appendix). All the coding and simulations were done using MATLAB.

4.1 Original BPCS Scheme Using α Complexity Measure

The results obtained by applying the original BPCS scheme and some variants suggested in chapter II are shown in figure 4.1. For any steganography scheme to work, it is necessary that the base image has a lot of high frequency features and colors. This ensures that there are a lot of complex regions in the image, which increases the amount of data that can be hidden in the image, and that the encoded image doesn't look visibly different from the original. The Baboon image, shown in figure 4.1(a), is probably the perfect image for steganographic applications. The image is a 512x512, 24 bits per pixel (bpp), bitmap color image. The image is about 786KB in size and, depending on the method used and the number of files embedded, it can be used to hide up to 520KB of data. The methods are evaluated based on amount of data they allow to hide and the distortion the hidden data causes on the original image.

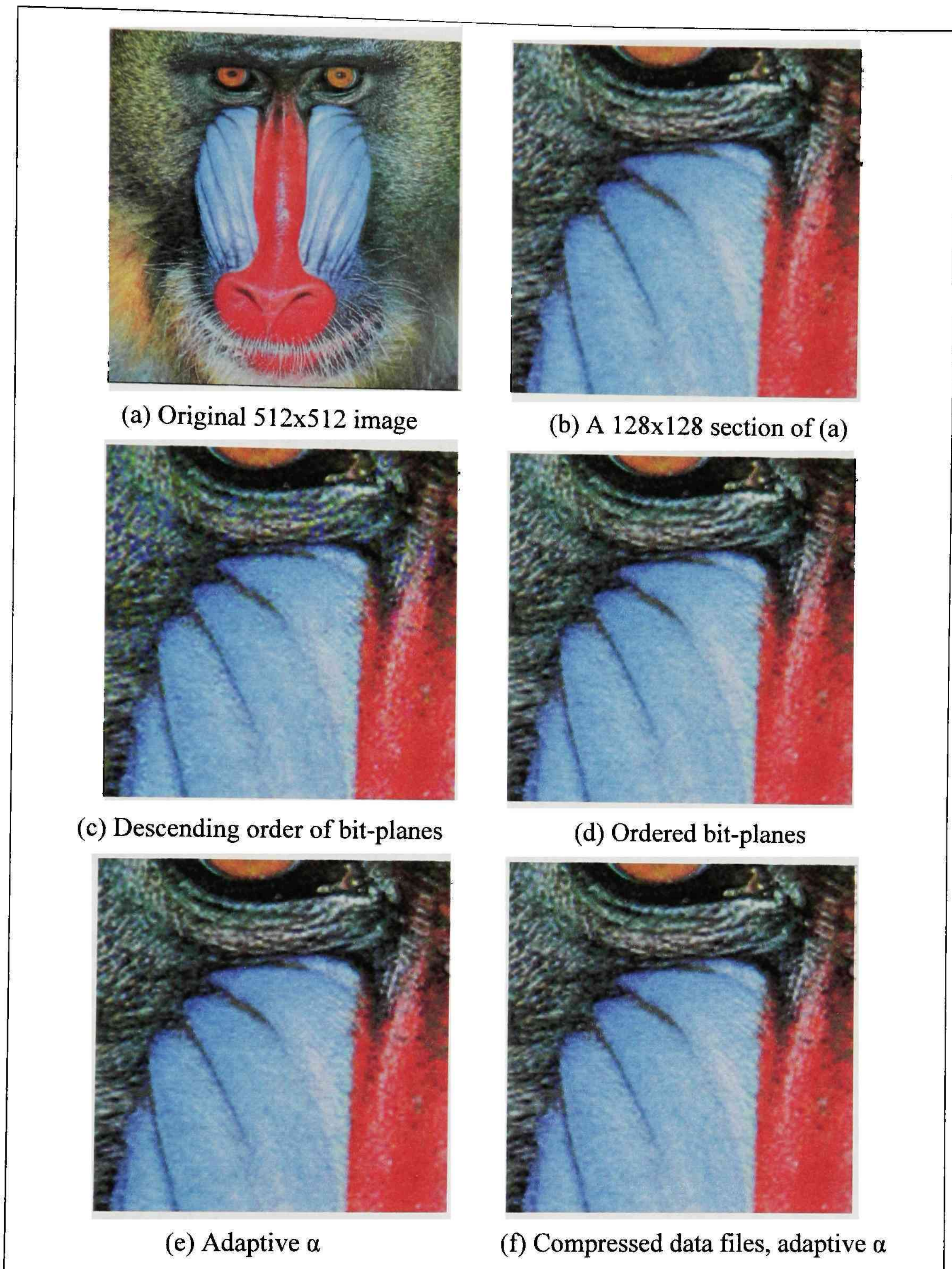


Figure 4.1 Images showing the result of applying the BPCS scheme and other variants suggested in chapter II

Image	Max. embeddable (bytes)	Bytes embedded (bytes)	% of max embedded	PSNR (dB)
(c)	532270	309090	58.07	22.6848
(d)	532270	309090	58.07	35.0760
(e)	469909	309090	65.77	35.2949
(f)	469933	308276	65.59	35.3363

(g) Table comparing the images (c)-(f) with respect to data embedded and PSNR

Figure 4.1 Continued

Other than in the case where the data is embedded using a descending order of bit-planes ([24, 23 ... 3, 2, 1]), there is no visible degradation of the original image if the images (original and the encoded image) are viewed in their original sizes. However, if the images were zoomed to larger sizes, say 2:1 or 3:1, the pixel patterns start to reveal some discrepancies from the original image. For this reason, only a small 128x128 section, containing a good sample of all the colors in the original image, is used for analysis. Figure 4.1(b) shows one such section, zoomed to twice its original size. Figure 4.1(c) shows the same 128x128 section of the encoded image, where the encoded image is the original image, shown in Figure 4.1(a), with 309090 bytes of hidden information. The encoder in this case follows the descending order of bit-planes rule wherein the blue plane is embedded before the green and red planes. It can be easily seen that this image has more blue regions than in the original image. Since the amount of data hidden in the original image is only 58.07% of maximum amount of data that can be hidden in the image (from Figure 4.1(g)), the data has been hidden almost completely in the blue planes. This causes the image to look visibly different from the original image and results in a low PSNR.

Figure 4.1(d) shows the result of embedding in the lower bit-planes of each color before starting with the higher bit-planes. The order explained in section 2.8 was used. It is easily seen that this image looks very similar to the original image and this is reflected on the high PSNR value obtained for this image. Figure 4.1(e) is the section of the encoded image in which the thresholds for the bit-planes are fixed according to equation 2.8. The improvement in image quality, over 4.1(d) which uses a fixed threshold of 0.3, is actually much better than indicated by the improvement in PSNR. Figure 4.2 shows the number of embeddable blocks in each bit-plane using a fixed threshold scheme and an adaptive threshold scheme.

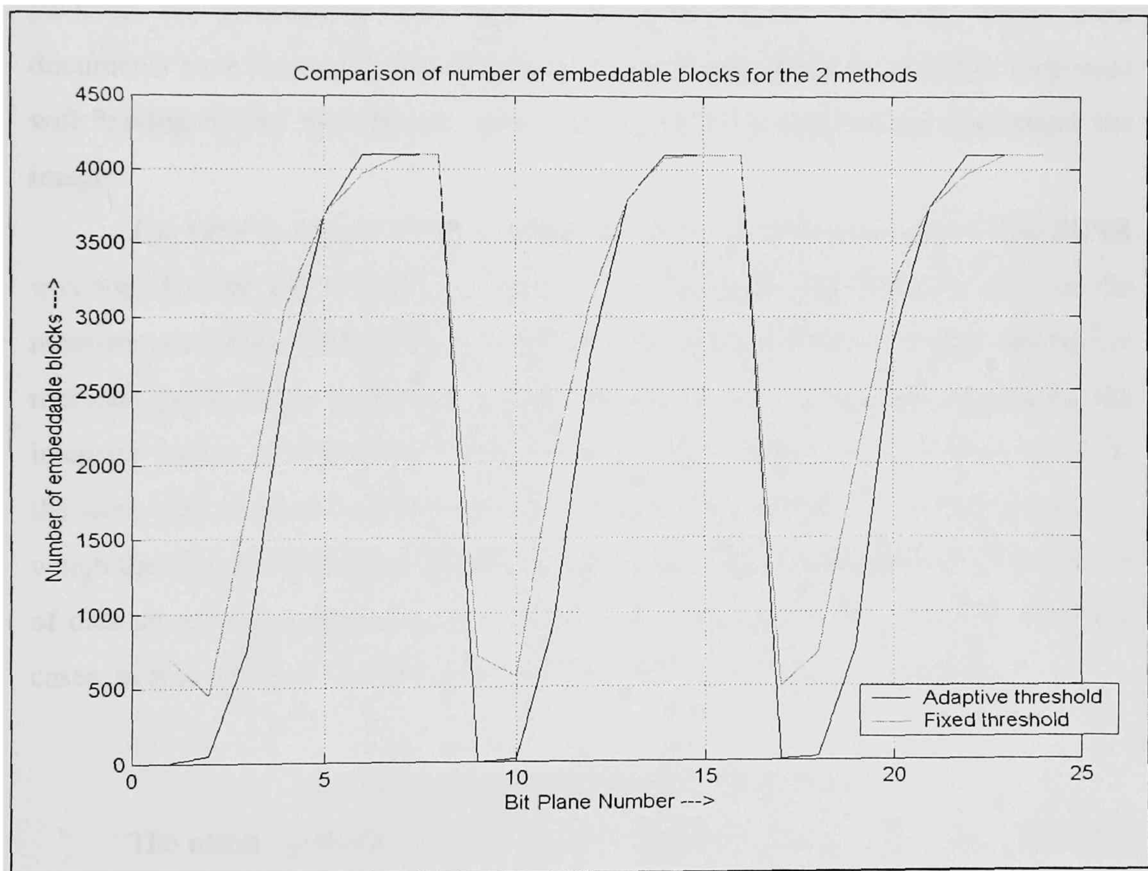


Figure 4.2 Graph comparing the number of blocks modified in each plane using fixed and adaptive thresholds

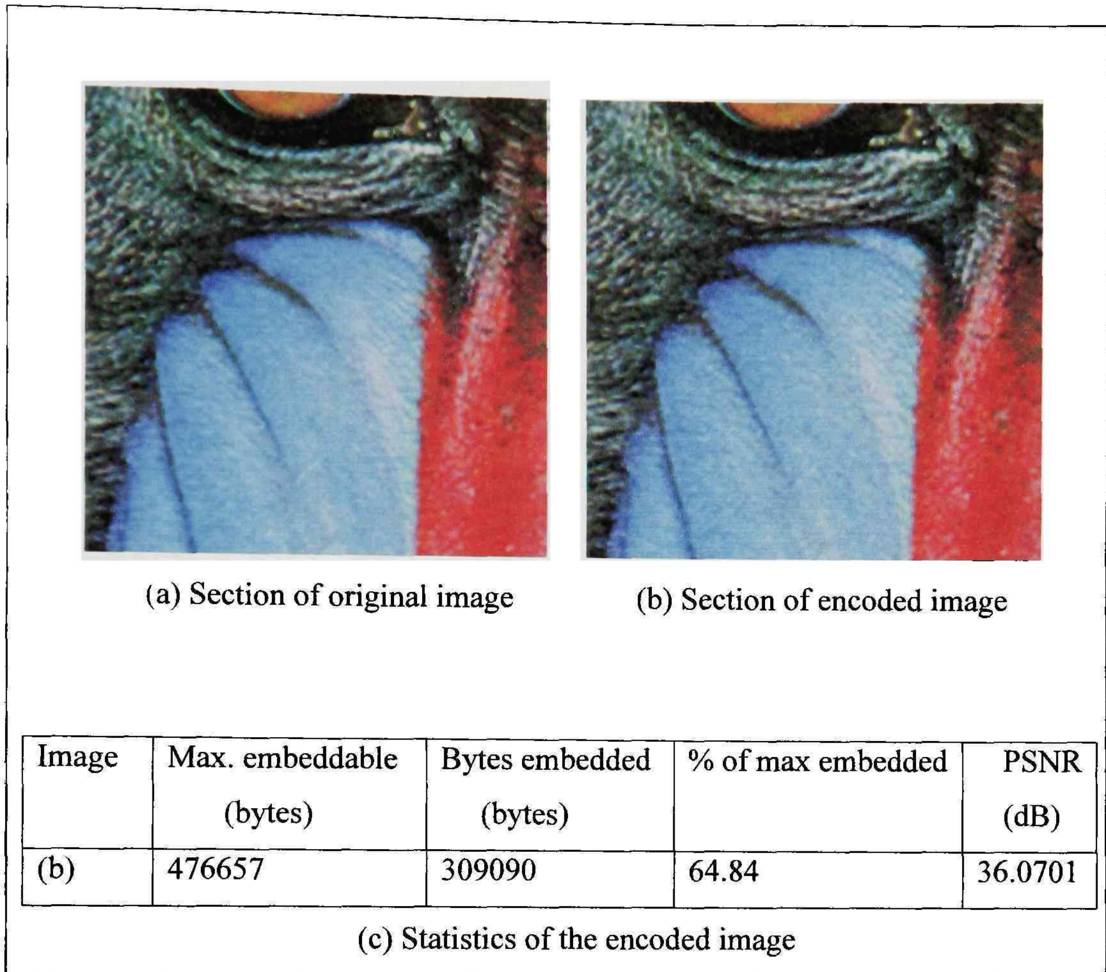
It is apparent that the adaptive threshold scheme uses fewer of the more significant bit-plane blocks than the fixed threshold scheme. This is important as

modifying the more significant bit-planes tend to cause visible distortion to the image and make them more suspicious. However, the compromise is the reduction in the maximum amount of data that can be hidden in the base image. The effective amount of information hidden in the image can be increased by passing the resource files through an entropy coder. In general, MS Word and PDF documents can be compressed to half their original sizes (on average). Figure 4.1(f) actually contains 641245 bytes, which is actually about 82% of the size of the image, compressed to less than half its size by passing it through an Adaptive Arithmetic Encoder [10]. The entropy encoder also helps in making the resource file blocks complex, as they try to pack in the information bits together by removing redundancies. Since most documents have long runs of blank spaces, using an entropy coding module dispenses with having blocks like the one shown in Figure 2.3 (c) embedded throughout the image.

The table in Figure 4.1(g) summarizes the results discussed above. The PSNR was found using the relation in equation 2.7. Although, the PSNR is used as the measure to evaluate the distortion, it does not describe the effectiveness of data hiding methods perfectly as most data hiding methods rely on explicitly modifying the intensity values, which affect the PSNR. The way the PSNR is calculated, even for the same size of the data embedded, the content of the resource files and the order in which the files are embedded can cause the PSNR values to be different. The amount of data stored in image 4.1(f) is not same as the amount of data stored in the other cases, as it is difficult to predict the exact size of the files post compression.

4.2 New Complexity Measures with MBSC

The result obtained by applying the complexity measures, β and γ, combined with MBSC, is shown in Figure 4.3. Figure 4.3(a) is the same 128x128 section of the Baboon image as in Figure 4.1(b) and Figure 4.3(b) is the corresponding section of the encoded image, encoded using the β and γ measures.



(a) Section of original image

(b) Section of encoded image

Image	Max. embeddable (bytes)	Bytes embedded (bytes)	% of max embedded	PSNR (dB)
(b)	476657	309090	64.84	36.0701

(c) Statistics of the encoded image

Figure 4.3 Data hiding using the β and γ measures

It is readily seen that there is no perceptible difference between the original and the encoded images. Also, using the β and γ measures provides a greater capacity than the corresponding BPCS variant (BPCS using adaptive α, Figure 4.1(e)). The PSNR of the encoded image in Figure 4.3(b) is significantly higher than the PSNR for any of the images obtained using the BPCS technique with the α complexity measure.

4.3 Variation of PSNR with Amount of Data Embedded

The PSNR of the encoded image depends almost exclusively on the amount of data embedded. As the amount of data embedded in the original image increases, the number of areas in the original image also increases. This causes the PSNR to decrease as the Mean Squared Error (MSE), and hence the Normalized Mean Square Error (NMSE), between the original and the encoded image increases. A plot of the PSNR for various percentages of data embedded (with respect to the maximum embeddable), for the baboon image, is shown in Figure 4.4. The variation is almost perfectly linear.

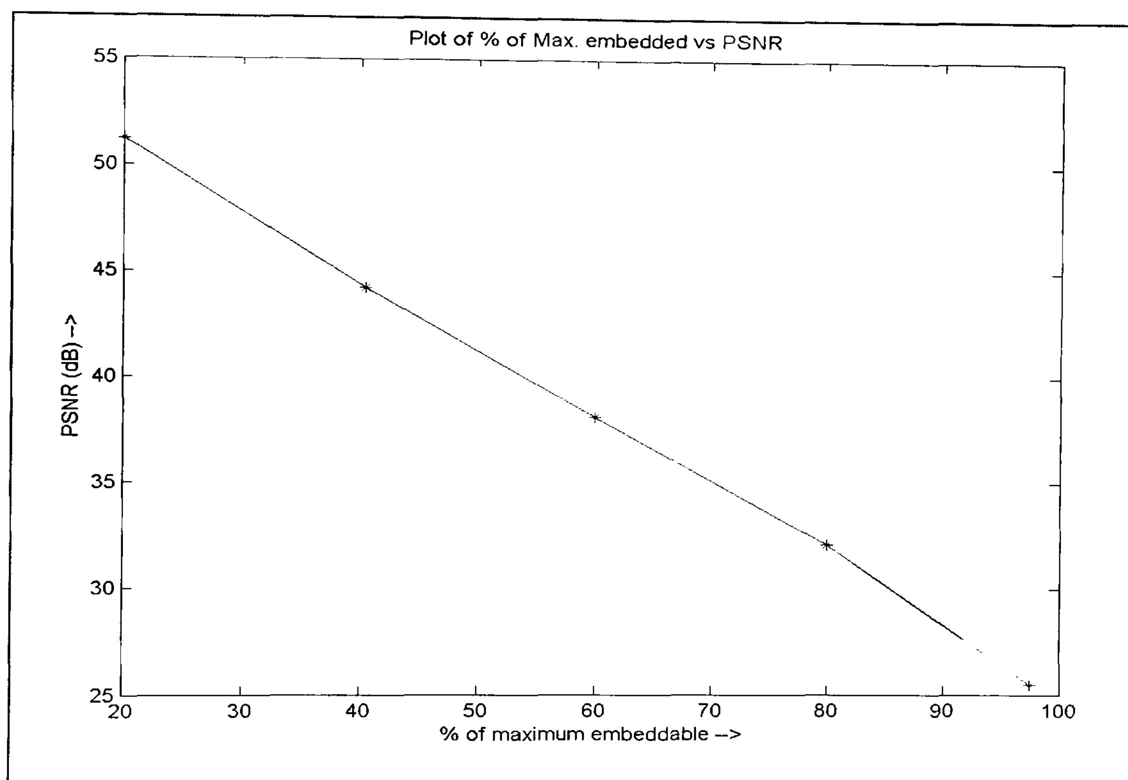
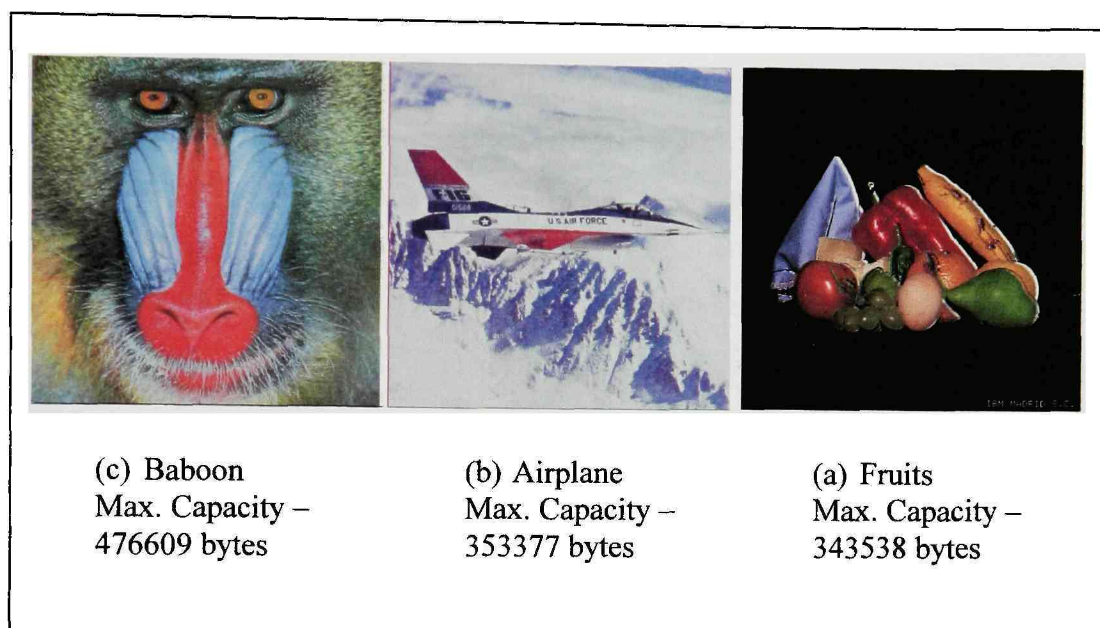


Figure 4.4 Plot of percentage of maximum data embedded with PSNR

4.4 Dependence of Embedding Capacity on the Base Image

The resource files are embedded in complex regions of the base image. If the image has a smooth background and if the objects in the image itself are plain, with very few features, the image forms a very bad base image with very low capacity.

Figure 4.5(a)-(c) show three standard images, which are used to test image processing algorithms, and the maximum amount of data that can be hidden in them, using the β and γ measures. In all the cases, it is assumed that a single large file is being embedded into the base image. Even from the appearance of the images, the Baboon image can be expected to have a much larger capacity than Airplane and the Fruits images. This is also reflected on the actual values.



(c) Baboon
Max. Capacity –
476609 bytes

(b) Airplane
Max. Capacity –
353377 bytes

(a) Fruits
Max. Capacity –
343538 bytes

Figure 4.5 Three test images and their maximum capacity

4.5 Conclusions

In conclusion, it can be seen that BPCS steganography can be effectively used to build a system that hides large chunks of data in images. The variations to the original scheme, introduced in section 2.8, significantly improve the performance of the original scheme. The β and γ complexity measures, used in combination, are better representative of the complex regions in the image than the α measure, and combined with the theory of M-sequences, they provide a high-capacity, low-overhead, technique for hiding data in images.

4.6 Future Scope

The future of this work lies in extending it to existing lossy compression schemes and providing improved security against hackers. In [13] the authors extend the BPCS technique to the popular *embedded zerotree wavelet* (EZW) scheme and claim that it could be extended to other wavelet-based, low bit-rate codecs. Since most images are compressed prior to transmission, applying the BPCS technique in the transform domain (wavelet domain in this case) improves the robustness of the embedded data to image compression. However, the embedding capacity is substantially reduced. The steganography method presented here can also be combined with some cryptography method to keep the data non-decipherable even if it were detected. The reserve bytes in the header could also be used to encode a password or a key, which would have to be matched to decode the rest of the data.

REFERENCES

1. Eiji Kawaguchi, Richard O. Eason, "Principle and Applications of BPCS-Steganography." SPIE's International Symposium on Voice, Video and Data Communications, (1998-11).

2. Neil F. Johnson, Zoran Duric, Sushil Jajodia, *Information hiding: Steganography and Watermarking- Attacks and Countermeasures*, Kluwer Academic Publishers, 2001.

3. John F. Wakerly, *Digital Design Principles and Practices*, II ed., Prentice Hall, 1990.

4. Eiji Kawaguchi, Michiharu Niimi, "Modeling Digital Image into Informative and Noise-Like Regions by Complexity Measure," *Information Modeling & Knowledge Bases IX*, IOS Press, pp.255-265, April, 1998.

5. Rafael C. Gonzalez, Richard E. Woods, *Digital Image Processing* 2nd ed., Pearson Education Asia ,Singapore, 2002.

6. Richard Eason, "A Tutorial On BPCS Steganography and Its Applications," Proceedings of Pacific Rim workshop on Digital Steganography 2003, pp. 18-31 (invited paper), Kitakyushu, Japan, July 2003.

7. Hioki Hirohisa, "A Data Embedding Method Using BPCS Principle With New Complexity Measures," Kyoto University, Japan. http://www.i.h.kyoto-u.ac.jp/%7Ehioki/research/DH/hioki_steg02_revised_paper.pdf

8. V. N. Yarmolik, S. N. Demidenko, *Generation and Application of Pseudorandom Sequences for Random Testing*, John Wiley & Sons, 1988.

9. New Wave Instruments, "Linear Feedback Shift Registers – Implementation, M-sequence Properties and Feedback Tables", http://www.newwaveinstruments.com/resources/articles/m_sequence_linear_feedback_shift_register_lfsr.htm

10. Ian H. Witten, Radford M. Neal, John G. Cleary, "Arithmetic Coding for data Compression," Communications of the ACM, June 1987, Volume 30, No. 6.

11. Stefan Katzenbeisser, Fabien A. P. Petitcolas, *Information Hiding – Techniques for Steganography and Digital Watermarking*, Artech House, Inc., 2000.

12. Peter Wayner, *Disappearing Cryptography – Information Hiding: Steganography &Watermarking*, II ed., Morgan Kaufmann Publishers, 2002.

13. Hideki Noda, Jeremiah Spaulding, Mahdad N.Shirazi, Michiharu Niimi, Eiji Kawaguchi, "Application of Bit-plane Decomposition Steganography to Wavelet Encoded Images", Proceedings of the IEEE International Conference on image Processing , Rochester, NY, September 2002, pp. II-909-II-912

14. X.-G. Xia, C. G. Boncelet, and G. R. Arce, "A multiresolution watermark for digital images," Proeedings of the IEEE International Conference on image Processing, Santa Barbara, CA, October 1997, pp. 548--551.

15. Ki-Hyeok Bae, Sung-Hwan Jung, "A New Information Hiding Using Wavelet Coefficient Relation in JPEG 2000", Dept. of Computer Engineering, Changwon National University, Korea.

16. The Mathworks Inc., Image Processing Toolbox 4.1.

APPENDIX

This section contains screenshots of the MATLAB GUI developed for embedding data into images and retrieving the information from encoded images. When the GUI is initially loaded, the menu shown in Figure A.1 is displayed. This menu lets the user select between the encoder and the decoder, which are shown in Figures A.2 and A.3, respectively. The encoder lets the user embed up to five files into the base image and displays the statistics like the maximum amount of data that can be embedded, size of the data actually embedded, encoding time, PSNR of the encoded image etc. The decoding module retrieves the resource files from an encoded image.

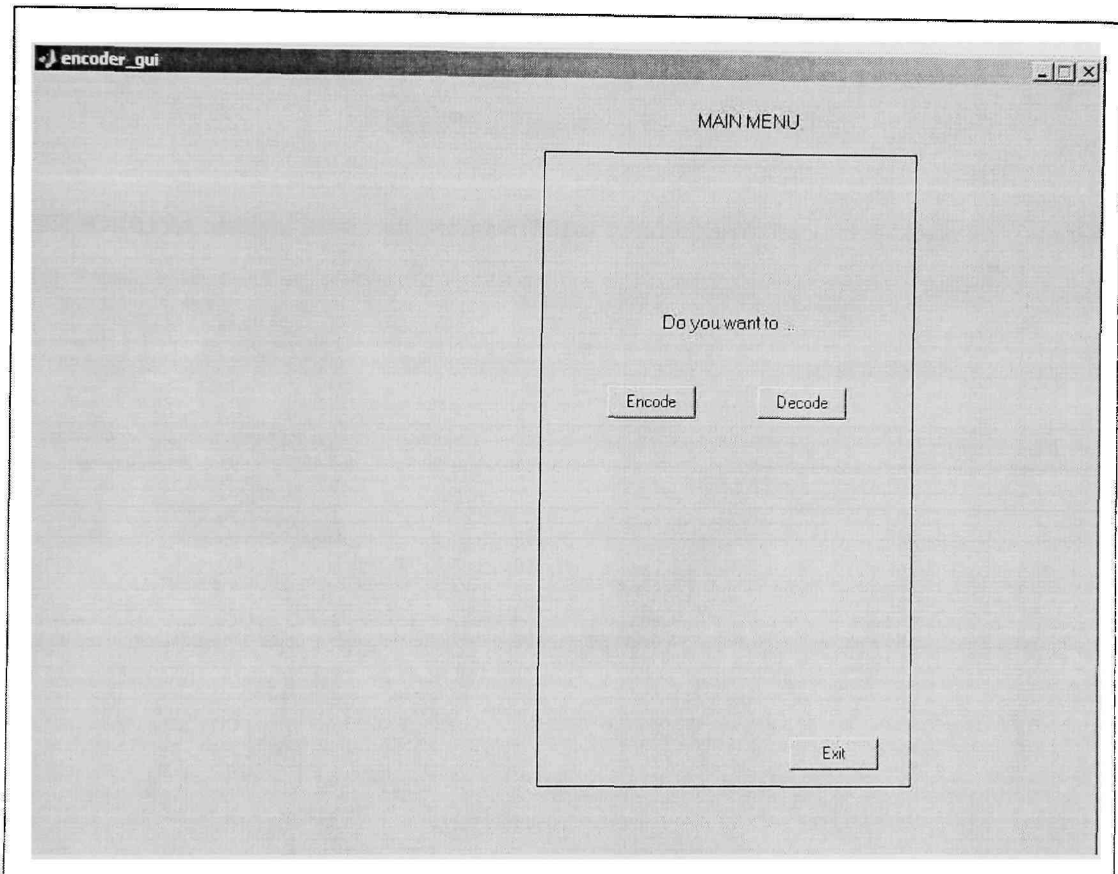


Figure A.1 Main menu

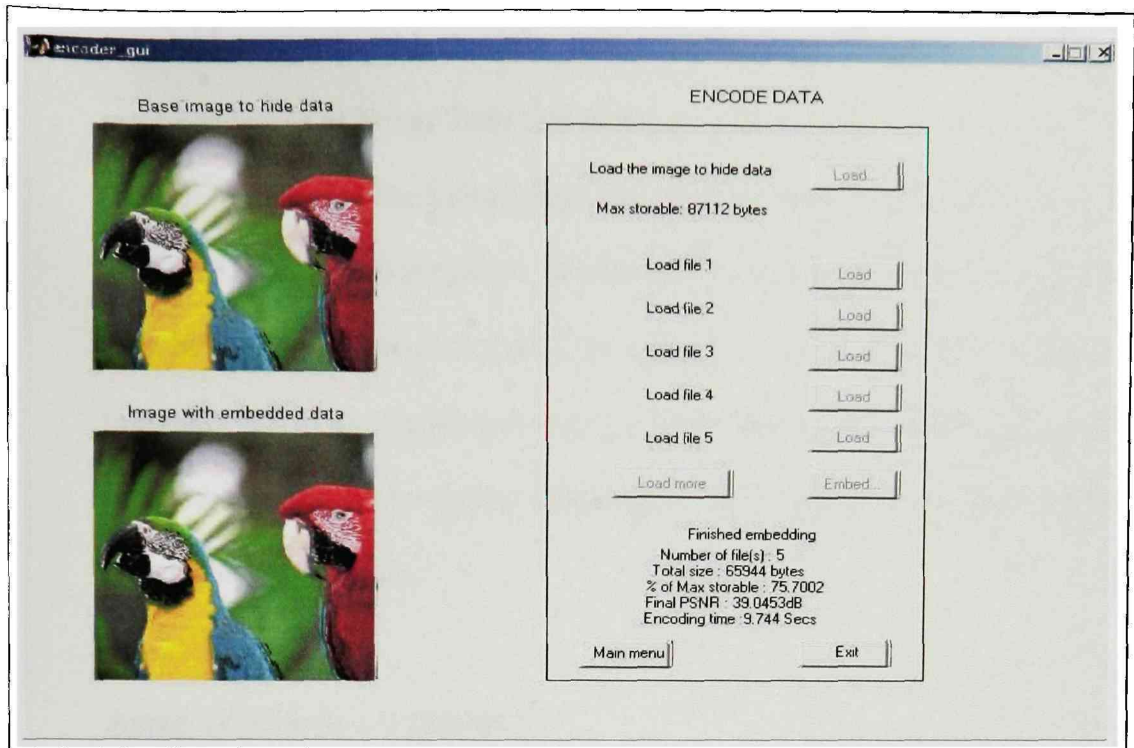


Figure A.2 Encoding module

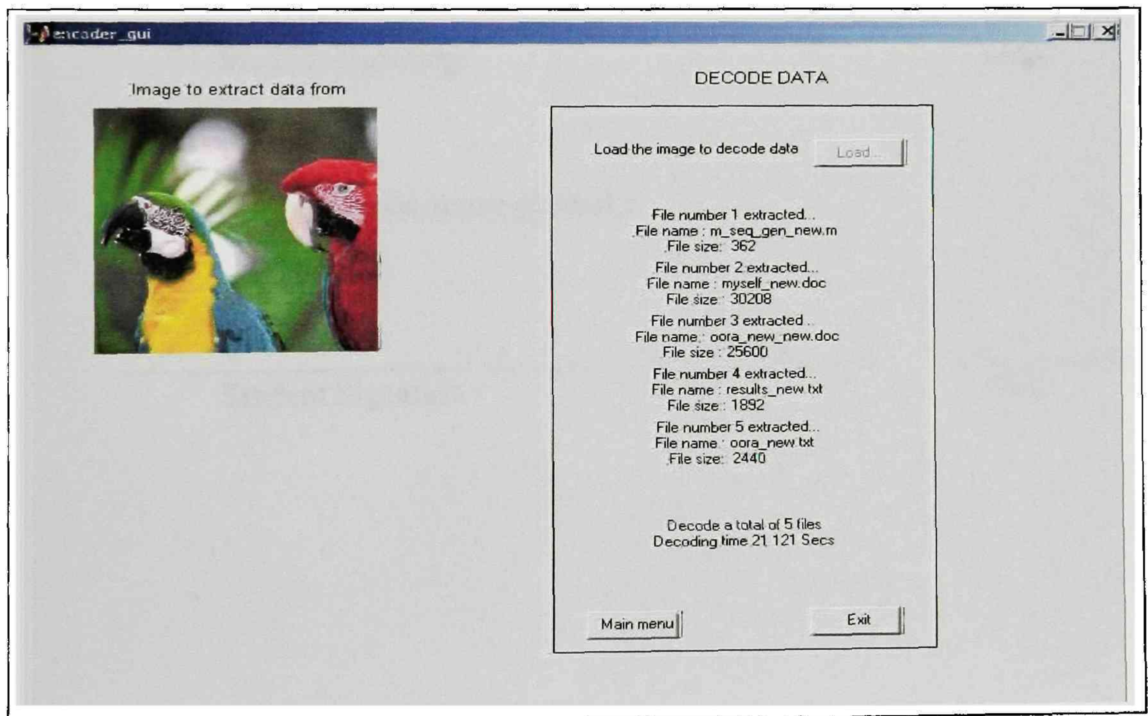


Figure A.3 Decoding module

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Texas Tech University or Texas Tech University Health Sciences Center, I agree that the Library and my major department shall make it freely available for research purposes. Permission to copy this thesis for scholarly purposes may be granted by the Director of the Library or my major professor. It is understood that any copying or publication of this thesis for financial gain shall not be allowed without my further written permission and that any user may be liable for copyright infringement.

Agree (Permission is granted.)

______________________________ ____________
Student Signature Date

Disagree (Permission is not granted.)

______________________________ ____________
Student Signature Date