# Hybrid Lossless Compression Technique

Project Report submitted in partial fulfillment of the requirement for the degree of

Bachelor of Technology.

in

**Information Technology**

Under the Supervision of

**AMIT KUMAR SINGH**

By

**JASJYOT SINGH KOHLI (111443)**

to

Jaypee University of Information and Technology

Waknaghat, Solan – 173234, Himachal Pradesh

# Certificate

This is to certify that project report entitled "**Hybrid Lossless Compression Technique**", submitted by **JASJYOT SINGH KOHLI** in partial fulfillment for the award of degree of Bachelor of Technology in Information Technology Engineering to Jaypee University of Information Technology, Waknaghat, Solan has been carried out under my supervision.

This work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.

**AMIT KUMAR SINGH**

**Assistant Professor**

**Date: 15-5-2015**

# Acknowledgement

I have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals and organizations. I would like to extend my sincere thanks to all of them.

I am highly indebted to Mr. AMIT KUMAR SINGH for their guidance and constant supervision as well as for providing necessary information regarding the project & also for their support in completing the project.

I would like to express my gratitude towards my parents & friends for their kind co-operation and encouragement which help me in completion of this project.

I would like to express my special gratitude and thanks to industry persons for giving me such attention and time.

.

Date: 15-5-2015 Jasjyot Singh Kohli

# Table of Content

# List of Figures

# List of Tables

# Abstract

This project creates software of one of the image processing application for image compression. It has been explicitly made for the image resizing. It manages space acquisition of computer hard disk. It includes the option to choose your image file and also save that image. The project also lets you compress and decompress the selected image. It also provides an option to make your image compress with three different algorithms. The algorithms include Huffman and Run Length Encoding and a Hybrid algorithm including features of both Huffman and Run Length Encoding.

It has capacity to show compressed and decompressed file on UI. In the UI, we pick image that user selects on which compression is to be performed. We save all files with save option and use write function.

The interface has been made very user friendly. The UI is created to perform one complete cycle for compression and decompression. We use image to binary and binary to image for compression. Overall objective of application is to reduce the size of image.

# CHAPTER 1

# COMPRESSION TECHNIQUES:
# AN OVERVIEW

## 1.1 INTRODUCTION

Compression is useful because it helps reduce resource usage, such as data storage space and increases transmission capacity. Data compression involves encoding information using fewer bits than the original representation. It works by finding patterns in data that occur frequently, and changing their representation to something short, so that the total amount of data is reduced without sacrificing any useful information. There are two types of compressions:

1) Lossy Compression

2) Lossless Compression

**1.2 Lossy compression** techniques reconstruct the original message with loss of some information. It reduces bits by identifying unnecessary information and removing it. It is also called irreversible compression. Lossy data compression schemes are informed by research on how people perceive the data in question. For example, the human eye is more sensitive to subtle variations in luminance than it is to variations in color. Data of some ranges which could not be recognized by the human brain can be neglected. Lossy compression is most commonly used to compress multimedia data like audio, video and still images.

The Lossy Techniques that are commonly used are:

1. JPEG
2. Fractal Compression
3. Block Truncation Coding

**1.2.1 JPEG** is a commonly used method of lossy compression for digital images, particularly for those images produced by digital photography. The degree of compression can be adjusted, allowing a selectable tradeoff between storage size and image quality. JPEG compression is used in a number of image file formats. JPEG/Exif is the most common image format used by digital cameras and other photographic image capture devices; along with JPEG/JFIF, it is the most common format for storing and transmitting photographic images on the World Wide Web.

**1.2.2 Fractal compression** is a lossy compression method for digital images, based on fractals. The method is best suited for textures and natural images, relying on the fact that parts of an image often resemble other parts of the same image. Fractal algorithms convert these parts into mathematical data called "fractal codes" which are used to recreate the encoded image.

**1.2.3 Block Truncation Coding (BTC)** is a type of lossy image compression technique for greyscale images. It divides the original images into blocks and then uses a quantiser to reduce the number of grey levels in each block whilst maintaining the same mean and standard deviation. BTC was first proposed by Robert Mitshell at Purdue University. Another variation of BTC is Absolute Moment Block Truncation Coding or AMBTC, in which instead of using the standard deviation the first absolute moment is preserved along with the mean.

**1.3 Lossless compression** techniques reconstruct the original data from the compressed file without any loss of data. It is also called reversible compression. Lossless data compression algorithms usually exploit statistical redundancy to represent data more concisely without losing information, so that the process is reversible. Lossless compression is possible because most real-world data has statistical redundancy. Lossless compression techniques are used to compress medical images, text, computer executable file and images preserved for legal reasons. Lossless compression results in a closer representation of the original media, and thus a higher quality end product. Lossy compression can give you a smaller file size, but the resulting end product may be in some ways inferior to the original. Run length encoding(RLE), Huffman, Arithmetic and Lempel Ziv Welch(LZW) Coding are the important lossless compression techniques.

The important applications of Lossless Compression Techniques are:

1. To compress medical images where loss of data can be a matter of concern.
2. To compress computer executable file.
3. Compressing images preserved for legal reasons.
4. Used in the ZIP file format and in the GNU tool gzip.

**1.3.1 Run Length Encoding (RLE)** is the simplest of the lossless compression algorithms. It replaces runs of two or more of the same character with a number which represents the length of the run, followed by the original character. Run-length encoding performs lossless data compression and is well suited to palette-based bitmapped images such as computer icons. It does not work well at all on continuous-tone images such as photographs, although JPEG uses it quite effectively on the coefficients that remain after transforming and quantizing image blocks.

Example of RLE:

Input: AAABBCCCCD

Output: 3A2B4C1D

**1.3.2 Huffman coding** algorithm was developed by David Huffman in 1951. In this algorithm fixed length codes are replaced by variable length codes. Huffman procedure works as follows:

1) Symbols with a high frequency are expressed using shorter encodings than symbols which occur less frequently.
2) The two symbols that occur least frequently will have the same length.

A binary tree is built up from the bottom up. Assume that the characters in a file to be compressed have the following frequencies:

A: 25 B: 10 C: 99 D: 87 E: 9 F: 66

C=00 D=01 F=10 A=110 B=1110 E=1111

**1.3.3 Arithmetic Coding** is useful for small alphabets with highly skewed probabilities. In this method, a code word is not used to represent a symbol of the text. Instead, it produces a code for an entire message. Arithmetic Coding assigns an interval to each symbol. The interval is then divided into sub-intervals. The number of sub-intervals is identical to the number of symbols in the current set and size is proportional to their probability of appearance. For each symbol a new internal, division takes place based on the last sub interval.

ARBER is coded as [0.14432, 0.1456)

**1.3.4 Lempel Ziv Welch (LZW)** uses fixed-length code words to represent variable-length strings of symbols that commonly occur together. The LZW encoder and decoder build up the same dictionary dynamically while receiving the data. It places longer and longer repeated entries into a dictionary, and then emits the code for an element, rather than the string itself, if the element has already been placed in the dictionary. LZW coding does not give efficient results for large data files but it gives good compression rates for small sized files. It cannot handle large files because of the increasing size of the dynamic dictionary involved.

**The project includes implementation of both Huffman Coding and Run length Encoding.**

## 1.4 OVERALL DESCRIPTION

### GOALS OF THE PROPOSED SYSTEM:

- **Planned approach towards working**: The working in the application will be well planned and organized. The data (Image) will be stored efficiently with optimal disk space consumption in user selected location which will help in retrieval of image while decompression.

- **Accuracy**: The level of accuracy in the proposed system will be higher. All operations would conform to integrity constraints and correctness and it will be ensured that whatever data is received at or sent from the centre is accurate.

- **Reliability**: The reliability of the proposed system will be high due to the above mentioned reasons. This comes from the fact that only the image which conforms to the accuracy clause would be allowed to commit back to the disk.

- **No redundancy**: In the proposed system it will be ensured that no repetition of information occurs; neither on a physical storage nor on a logical implementation level. This economizes on resource utilization in terms of storage space. Also even in case of concurrent access no anomalies occur and consistency is maintained.

- **Immediate retrieval of information**: the main objective of the proposed system is to provide a quick and efficient platform for retrieval of information. Data of images which is in side memory object is handled by image to binary and binary to image function for conversion .

- **Ease of operation**: The system should be simplistic in design and use. It is such that it can be easily developed within a short period of time and can conform to the financial and resource-related constraints of the organization.

# CHAPTER 2
# LITERATURE REVIEW

A large number of data compression algorithms have been developed and used throughout the years. Some of which are of general use and can be used to compress files of different types. Others are developed to compress efficiently a particular type of files. It has been realized that, according to the representation form of the data at which the compression process is performed, below is reviewing some of the literature review in this field.

Alarabeyyat et al. [1] proposed a method to design an efficient and effective lossless image compression scheme. It deals with the design of a lossless image compression method and is based on LZW algorithm and the BCH algorithm an error correcting technique, in order to improve the compression ratio of the image. It is a lossless image compression scheme which is applied to all types of image based on LZW algorithm that reduce the repeated value in image and BCH codes that detect/correct the errors. The BCH algorithm works by adding extra bits called parity bits, whose role is to verify the correctness of the original message sent to the receiver.

Chakraborty et al. [2] presented an approach in which image information had been firstly scanned for statistical redundancy following Run length encoding schemes. Then, the frequency-pixel pairs, thus generated, were encoded using Extended ASCII character-set (8 bit) using an efficient approach. The corresponding symbol-information pairs had been stored in dictionary format that was created and deployed individually with each image. No permanent storage of the dictionary was required. The compression algorithm had two interrelated procedures to facilitate the entire compression process. They were Run Length Encoding and Character Replacement scheme.

AlHashemi et al. [3] proposed a model based on BCH codes, for detecting/correcting errors in data transmission. The BCH code algorithm adds extra bits, called parity bits, whose role is to verify the correctness of the original message sent upon receipt. This BCH method converts the block of size $k$ bits into $n$ by adding m parity bits, depending upon the size of the message $k$, which is encoded into a codeword of length $n$.

Talu et al. [4] presented a lossless compression scheme for binary images which consists of a novel encoding algorithm which uses a new edge tracking algorithm. It has two sub-stages:

(i) Encoding binary image data using the proposed encoding method

(ii) Compression the encoded image data using any well-known image compression method such as Huffman, Run-Length or Lempel-Ziv-Welch (LZW).

The proposed encoding method contains two subsequent processes:

(i) Determining the starting points of independent objects.

(ii) Obtaining their edge points and geometrical shapes information.

Kaur et al. [5] proposed a Hybrid compression technique using the two lossless methodologies Huffman coding and Lempel Ziv Welch coding to compress data image. In the first stage, the image is compressed with Huffman coding and calculates the MSE, PSNR, CR and elapsed time of a data image. After that apply the LZW compression on same image and further we can use the both algorithms and recover the data image reduced the size of data image and calculate same results.

Abdmouleh et al. [6] proposed that compression is the coding of the data to minimize their representation by removing the redundancy present in them, keeping only sufficient information that can be effectively used in the decompressing phase to reconstruct the original data. The compression of images is motivated by the economic and logistic needs to conserve space in storage media and save bandwidth in communication.

Chaudhari et al. [7] proposed that frequently occurring and repetitive patterns are assigned to a shorter codeword. The less efficient codeword is assigned to the others. Based on this principle, the codeword table should be constructed to provide the fixed mapping relationship. Many famous methods, including Huffman coding, Run length coding , arithmetic coding, and LZW have been widely developed, and some of them are further applied in lossy compression standards.

**Table 2.1: Existing compression based methods**

| S.No | Author Name | Technique Used | Result (Average Compression Ratio) |
|---|---|---|---|
| 1 | A. Alarabeyyat, S. Al-Hashemi, T. Khdour | Hybrid of LZW and BCH | 1.67 |
| 2 | Debashis Chakraborty, Soumik Banerjee | RLE and Character Replacement Scheme | 1.292 |
| 3 | Rafeeq Al-Hashemi, Israa Wahbi Kamal | BCH Codes | 1.417 |
| 4 | M. Fatih TALU, İbrahim TÜRKOĞLU | Edge Tracking algorithm with Huffman | 1.364 |
| 5 | Dalvir Kaur, Kamaljeet Kaur | Hybrid of Huffman and LZW | 1.713 |

# CHAPTER 3

# A HYBRID COMPRESSION TECHNIQUE USING HUFFMAN AND RLE

The proposed method is based on two most important compression techniques: Huffman Coding and Run length encoding (RLE). The single compression technique can only save a limited purpose. In this work, we have combined the above two compression techniques to improve the performance of the proposed method in terms of compression factor. The theoretical background of the Huffman and RLE compression methods are given below:

## 3.1 RUN LENGTH ENCODING ALGORITHM

```
set color to 0
set count to 0
for each pixel in the image

    if current pixel not equal to color
        write count
        set color to current pixel color
        set count to 1
    else
        increment count by 1
if count not equal to 0
       write count
```

## 3.2 HUFFMAN CODING ALGORITHM

- Huffman (W, n)
- Input: A list W of n (positive) weights.
- Output: An extended binary tree T with weights taken from W that gives the minimum weighted path length.
- Procedure:

Create list F from singleton trees formed from elements of W

WHILE (F has more than one element) DO

Find T1, T2 in F that have minimum values associated with their roots

Construct new tree T by creating a new node and setting T1 and T2 as its children

Let the sum of the values associated with the roots of T1 and T2 be associated with the root of T

Add T to F

OD

Huffman: = tree stored in F

## 3.3 EXPERIMENTAL RESULTS AND ANALYSIS:

A data set of images of different formats is gathered to test and compare the compression factors of the individual compression techniques and also the hybrid compression technique. The results are recorded in a tabular form to get a better understanding of the various methods used.

**Table 3.1** shows the performance of the proposed method for png images using Huffman compression method. In this table, the highest compression factor has been found for Beach image. However, the least was found out to be for Portgas image. The average compression factor for the Huffman compression was calculated at **14.19**

**Table 3.2** shows the performance of the proposed method for png images using Run length encoding method. In this table, the highest compression factor has been found for Portgas image. However, the least was found out to be for Stream image. The average compression factor for the Huffman compression was calculated at **7.4**

**Table 3.3** shows the performance of the proposed method for png images using Hybrid compression method. In this table, the highest compression factor has been found for Beach image. However, the least was found out to be for Portgas image. The average compression factor for the Huffman compression was calculated at **16.37**

**Table 3.4** shows the performance of the proposed method for jpg images using Huffman compression method. In this table, the highest compression factor has been found for Desert image. However, the least was found out to be for Lighthouse image. The average compression factor for the Huffman compression was calculated at **7.82**

**Table 3.5** shows the performance of the proposed method for jpg images using Run length encoding compression method. In this table, the highest compression factor has been found for Desert image. However, the least was found out to be for Koala image. The average compression factor for the Huffman compression was calculated at **8.7**

**Table 3.6** shows the performance of the proposed method for jpg images using Hybrid compression method. In this table, the highest compression factor has been found for Desert image. However, the least was found out to be for Koala image. The average compression factor for the Huffman compression was calculated at **11.14**

The results show that Huffman Compression provides better compression factors than Run length encoding technique for png images. However, for the jpg images Run length encoding shows better compression factor than the Huffman Compression technique. The Hybrid method provided better results for both types of images and has proved to be more efficient than the original techniques.

**Table 3.1: Performance of the proposed method using Huffman compression technique for png files**

| S.No. | Name | Size Before Compression (kb) | Size after Compression (kb) | Compression Factor |
|---|---|---|---|---|
| 1 | Beach.png | 2885 | 179 | 16.07 |
| 2 | Portgas.png | 2146.6 | 195 | 10.98 |
| 3 | Car.png | 3725.9 | 262 | 14.20 |
| 4 | Stream.png | 2858.8 | 184 | 15.51 |
| **Average Compression Factor = 14.19** | | | | |

**Table 3.2: Performance of the proposed method using Run length Encoding technique for png files.**

| S.No. | Name | Size Before Compression (Mb) | Size after Compression (kb) | Compression Factor |
|---|---|---|---|---|
| 1 | Beach.png | 2885 | 387 | 7.45 |
| 2 | Portgas.png | 2146.6 | 228 | 9.38 |
| 3 | Car.png | 3725.9 | 553 | 6.73 |
| 4 | Stream.png | 2858.8 | 472 | 6.04 |
| **Average Compression Factor = 7.4** | | | | |

**Table 3.3: Performance of the proposed method using Hybrid compression technique for png files.**

| S.No. | Name | Size Before Compression (kb) | Size after Compression (kb) | Compression Factor |
|---|---|---|---|---|
| 1 | Beach.png | 2885 | 161 | 17.92 |
| 2 | Portgas.png | 2146.6 | 148 | 14.5 |
| 3 | Car.png | 3725.9 | 240 | 15.52 |
| 4 | Stream.png | 2858.8 | 163 | 17.54 |
| **Average Compression Factor = 16.37** | | | | |

**Table 3.4: Performance of the proposed method using Huffman compression technique for jpg files**

| S.No. | Name | Size Before Compression (kb) | Size after Compression (kb) | Compression Factor |
|---|---|---|---|---|
| 1 | Desert.jpg | 826 | 84 | 9.83 |
| 2 | Hydrangeas.jpg | 581 | 72.4 | 8.02 |
| 3 | Koala.jpg | 762 | 107 | 7.12 |
| 4 | Lighthouse.jpg | 548 | 86.7 | 6.32 |
| **Average Compression Factor = 7.82** | | | | |

**Table 3.5: Performance of the proposed method using Run length Encoding compression technique for jpg files**

| S.No. | Name | Size Before Compression (kb) | Size after Compression (kb) | Compression Factor |
|---|---|---|---|---|
| 1 | Desert.jpg | 826 | 66.2 | 12.48 |
| 2 | Hydrangeas.jpg | 581 | 79 | 7.35 |
| 3 | Koala.jpg | 762 | 183 | 4.16 |
| 4 | Lighthouse.jpg | 548 | 50.8 | 10.79 |
| **Average Compression Factor = 8.7** | | | | |

**Table 3.6: Performance of the proposed method using Hybrid compression technique for jpg files**

| S.No. | Name | Size Before Compression (kb) | Size after Compression (kb) | Compression Factor |
|---|---|---|---|---|
| 1 | Desert.jpg | 826 | 52.1 | 15.85 |
| 2 | Hydrangeas.jpg | 581 | 62.3 | 9.33 |
| 3 | Koala.jpg | 762 | 92 | 8.28 |
| 4 | Lighthouse.jpg | 548 | 49.4 | 11.09 |
| **Average Compression Factor = 11.14** | | | | |

# CHAPTER 4:

# PROJECT REQUIREMENTS

## 4.1 DESIGN

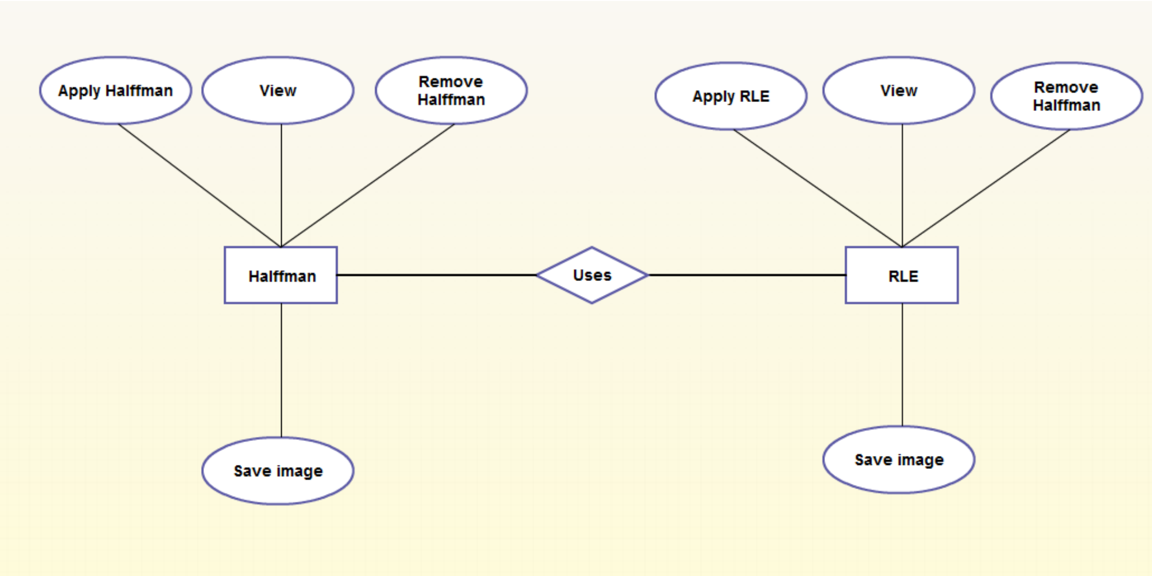



**Figure 1: ER Diagram**

An entity-relationship (ER) diagram is a graphical representation of entities and their relationships to each other, typically used in computing in regard to the organization of data within databases or information systems. An entity is a piece of data-an object or concept about which data is stored.

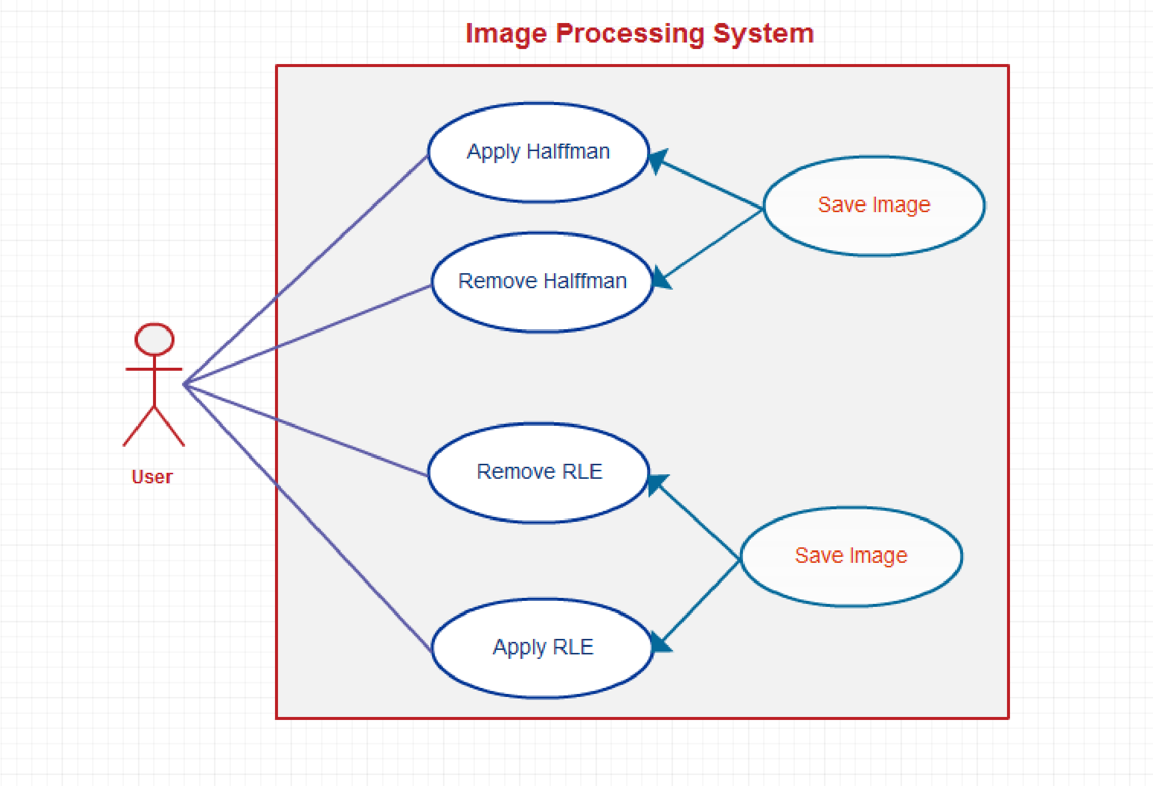



**FIGURE 2: Use Case Diagram**

Use case diagram is the representation of a user's interaction with the system and depicting the specifications of the use case. It can portray the different types of users of a system and the cases. It is a simple graphical formalism that can be used to represent a system in terms of the input data to the system, various processing carried out on these data, and the output data is generated by the system.

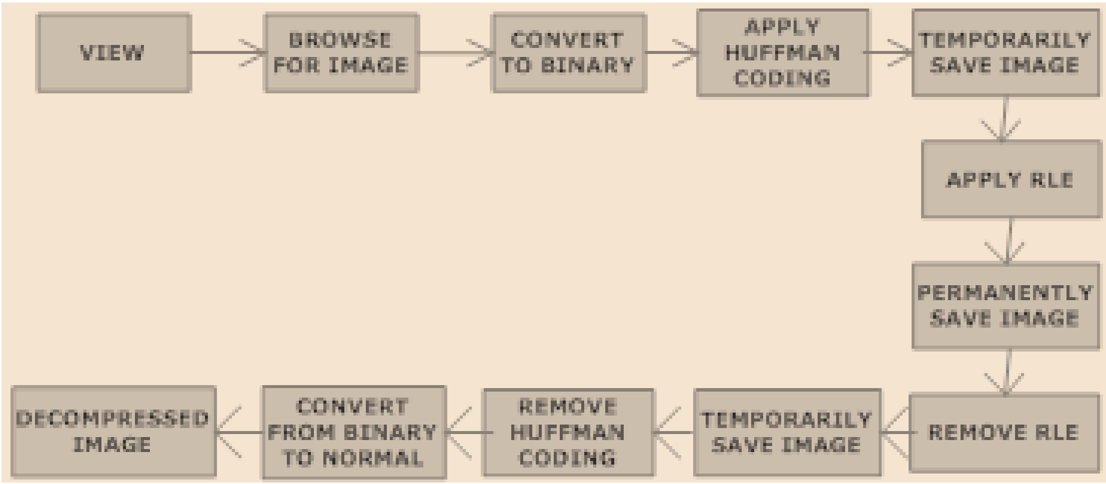



**FIGURE 3: FLOW CHART**

A flowchart is a formalized graphic representation of a logic sequence, work or manufacturing process, organization chart, or similar formalized structure. The purpose of a flow chart is to provide people with a common language or reference point when dealing with a project or process.

## 4.2 SOFTWARE SPECIFICATION

- Operating System: Windows 7/8
- Technology : MATLAB
- Tools : MATLAB 2014

## 4.3 HARDWARE SPECIFICATION

- Processor: x86 compatible processor
- RAM: 512 MB or greater
- Hard Disk: 20 GB or greater
- Monitor: VGA/SVGA
- Keyboard: 104 keys standard
- Mouse: 2/3 button. Optical/ Mechanical.

## 4.4 USER CHARACTERISTICS

Every user:

- Should be comfortable with basic working of the computer.
- Must have basic knowledge of English.
- Must carry a png image.

## 4.5 CONSTRAINTS

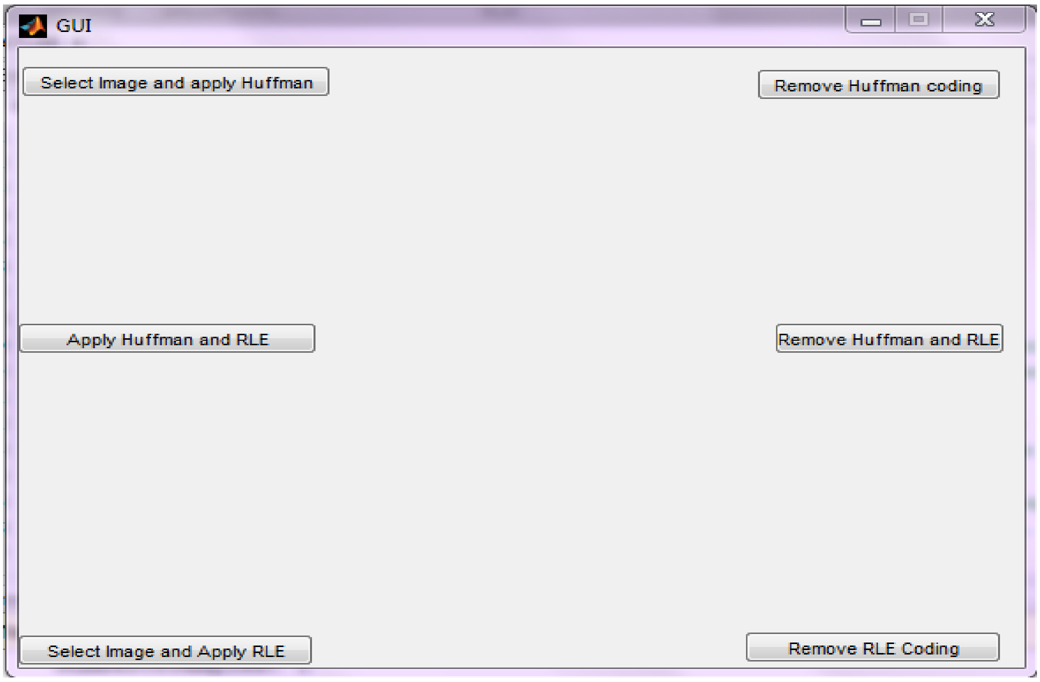
- The GUI is restricted to English
- PNG and JPG image is must.
- The application is applicable only to the images.

# CHAPTER 5

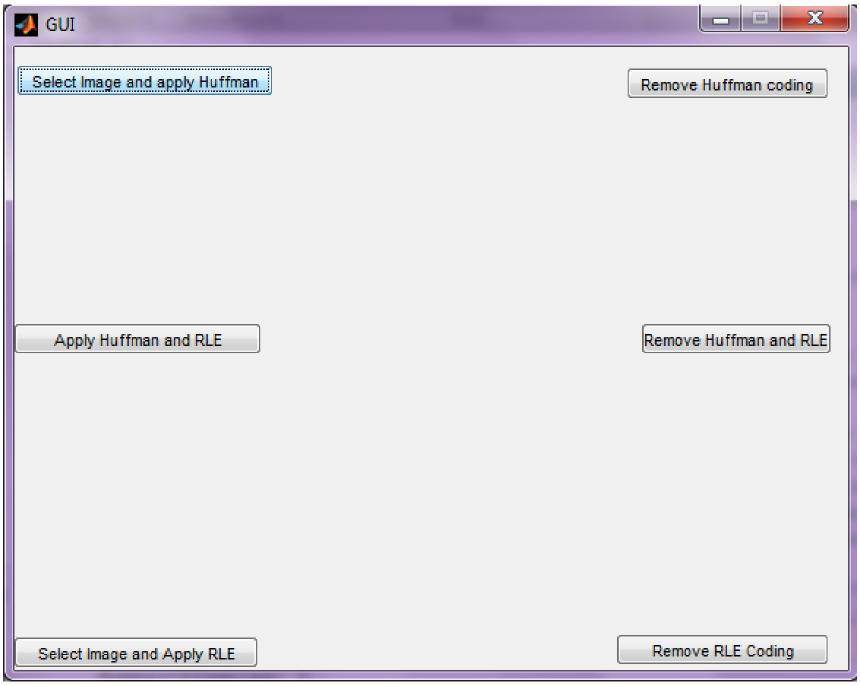
# IMPLEMENTATION

## 5.1 SNAPSHOTS

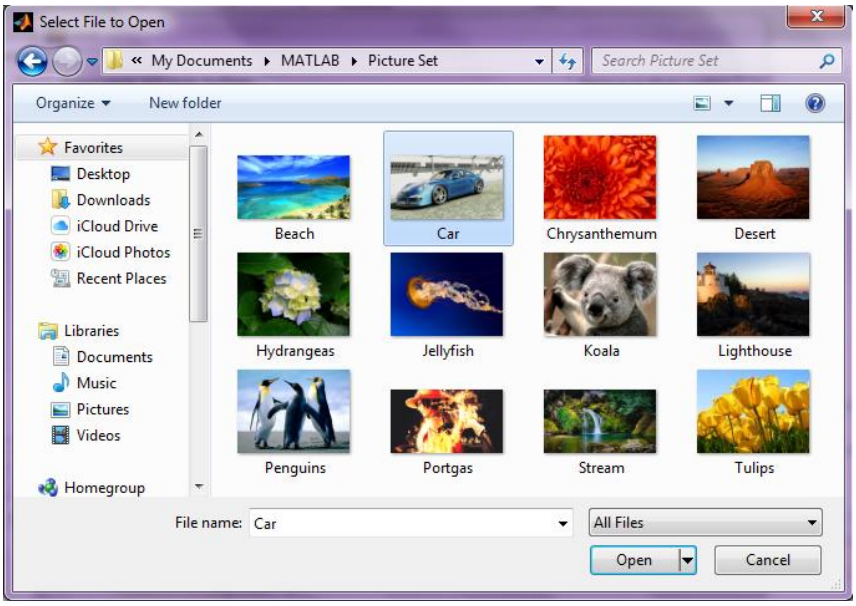
**This gives the First view of the Application. It has options like Select image and applies Huffman, Select image and applies RLE and Select image and applies Hybrid Technique. Right side buttons are for removing Huffman coding, RLE coding and Hybrid Technique.**



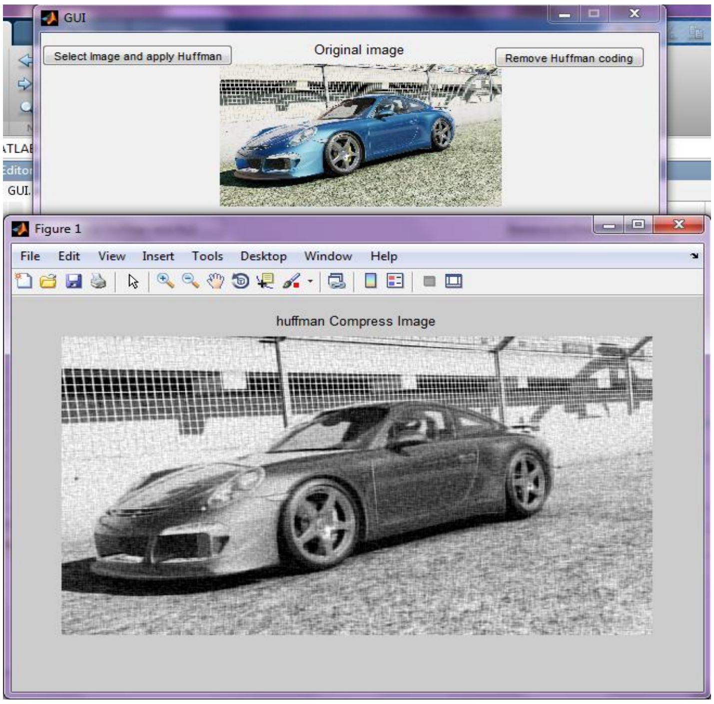
**The selected button is for choosing the image and to apply Huffman coding algorithm to compress**



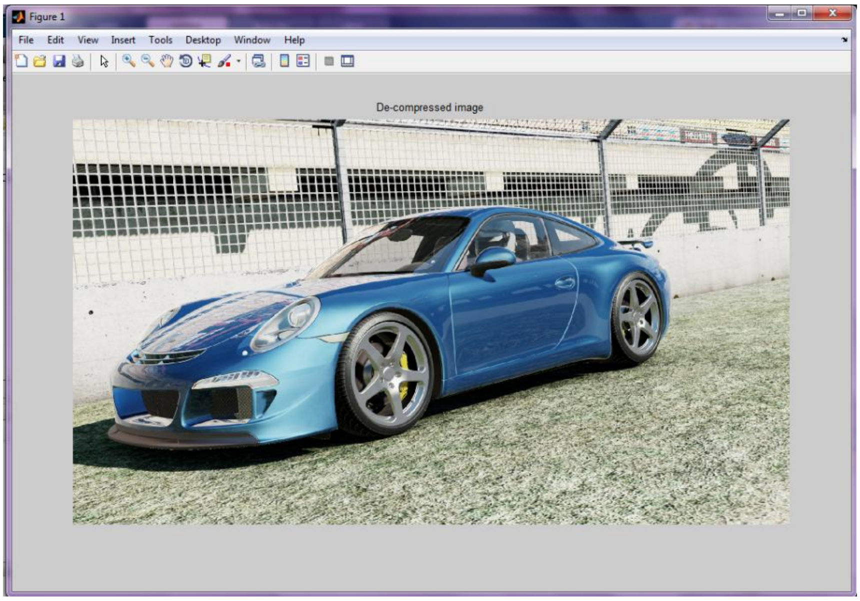

**This is the browse picture dialog box which helps us to choose the image to be compressed.**



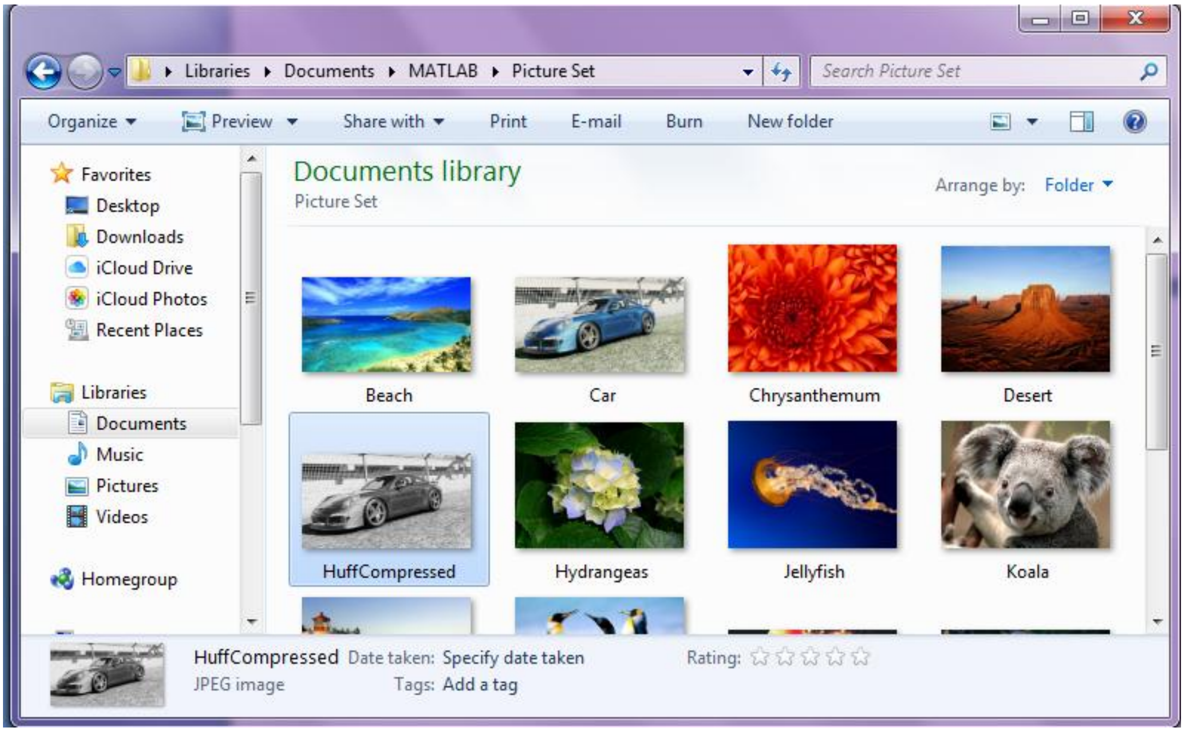
**This window shows the original picture and the compressed image through Huffman coding algorithm.**



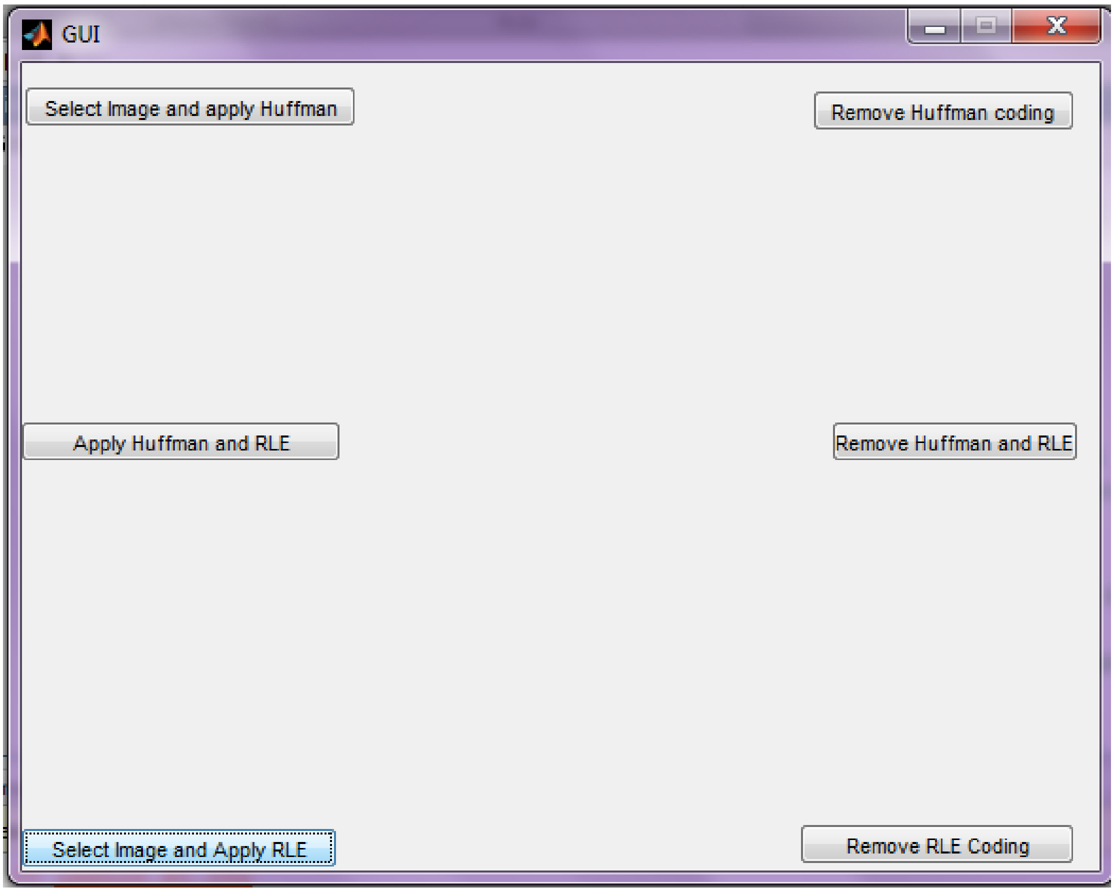

**This window shows the decompressed image on which Huffman coding was applied.**



**This window shows the compressed file which is saved at the same spot from where the image was taken.**

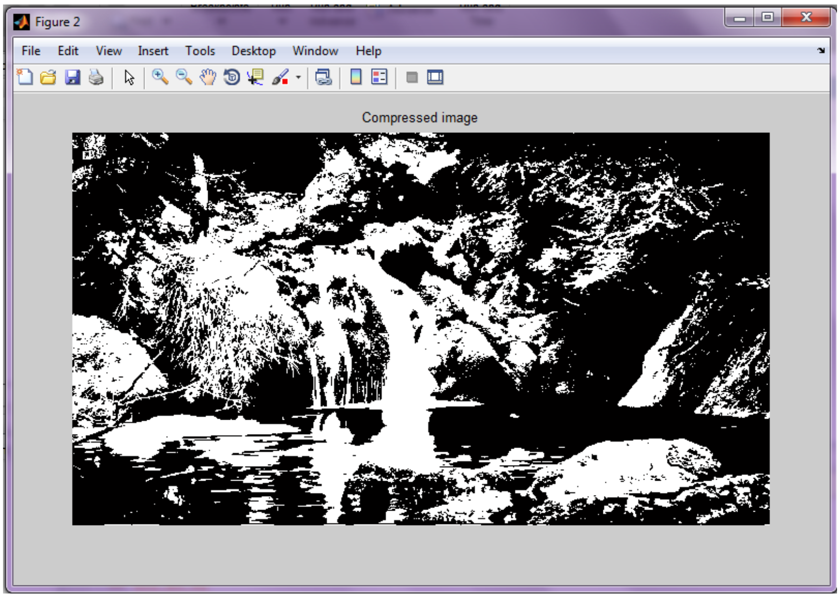


**The selected button is for choosing the image and to apply RLE algorithm to compress it.**

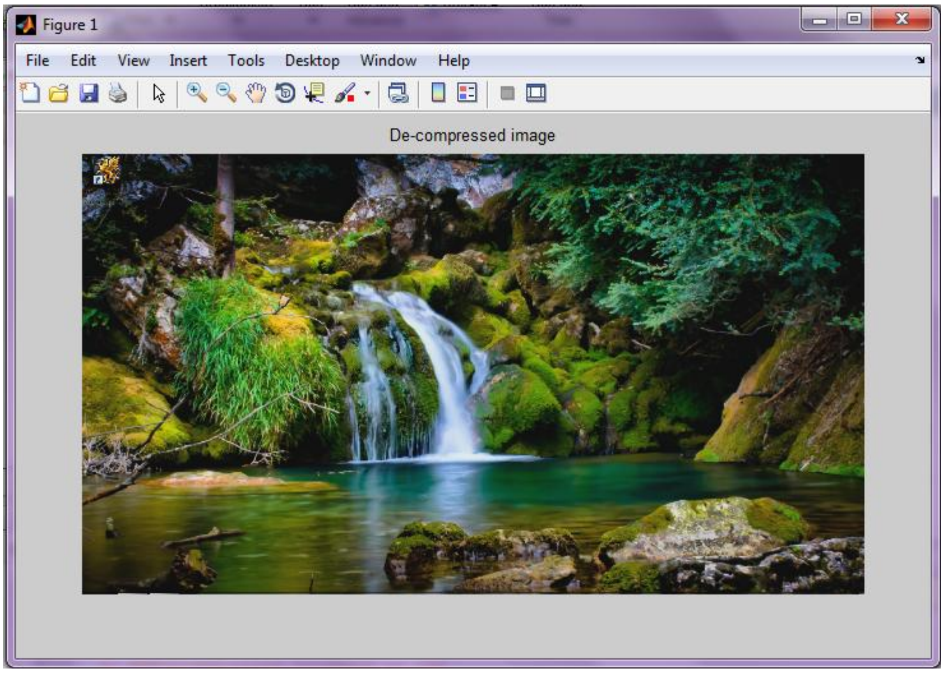


**The window shows the original image which is to be compressed.**

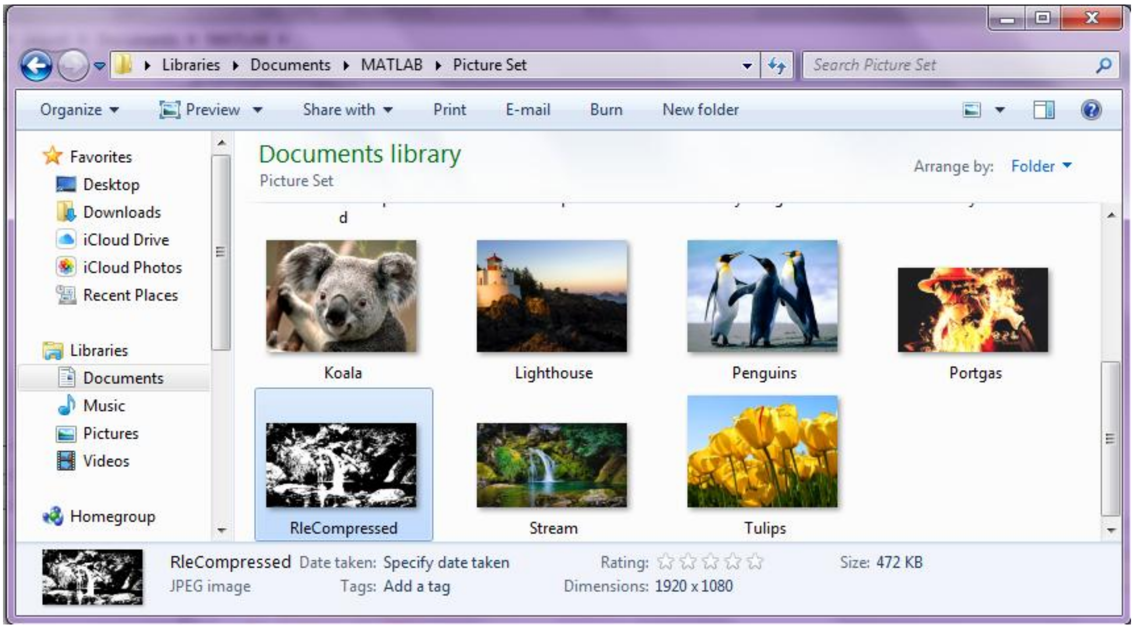

**This window shows the compressed image after applying RLE algorithm.**



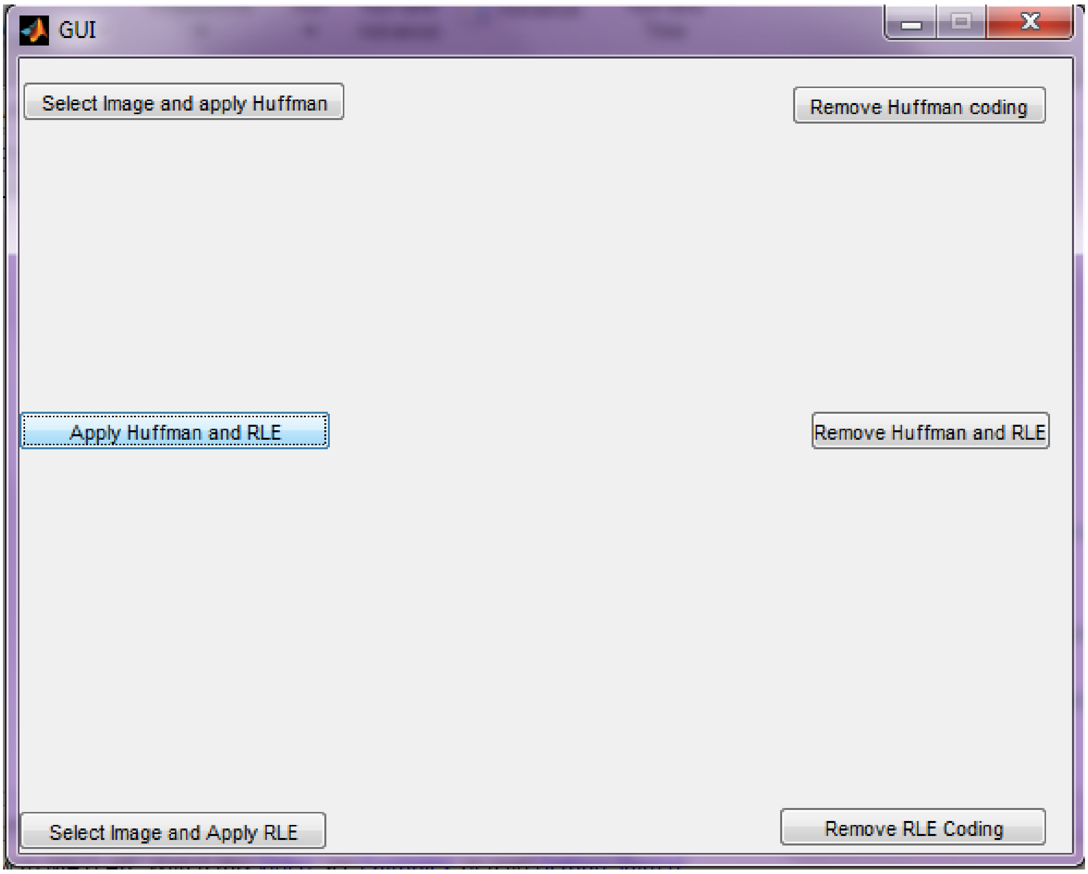
**This window shows the decompressed image on which RLE algorithm was applied earlier.**



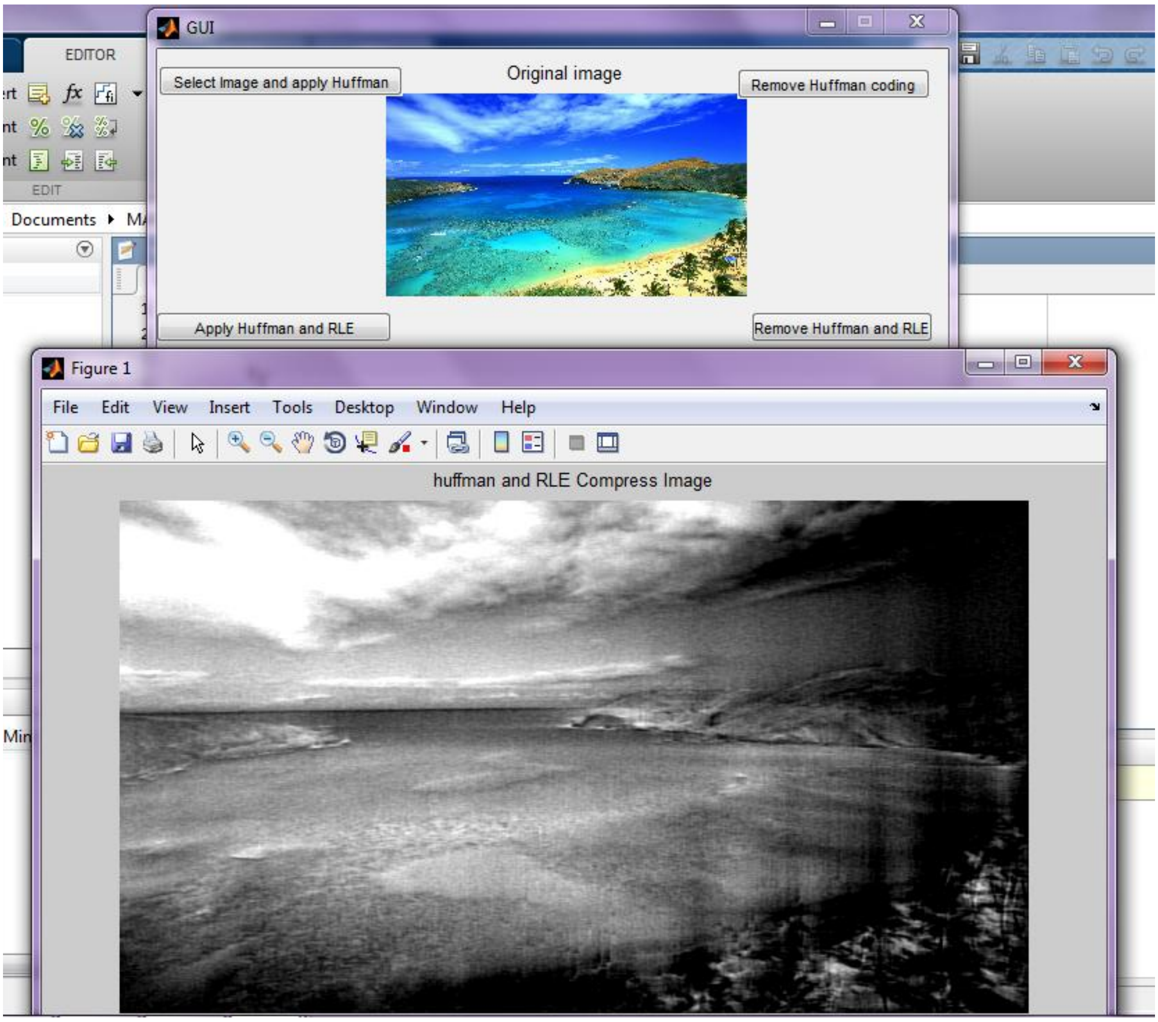

**This window shows that the compressed image is saved at the same location as that of the source image.**



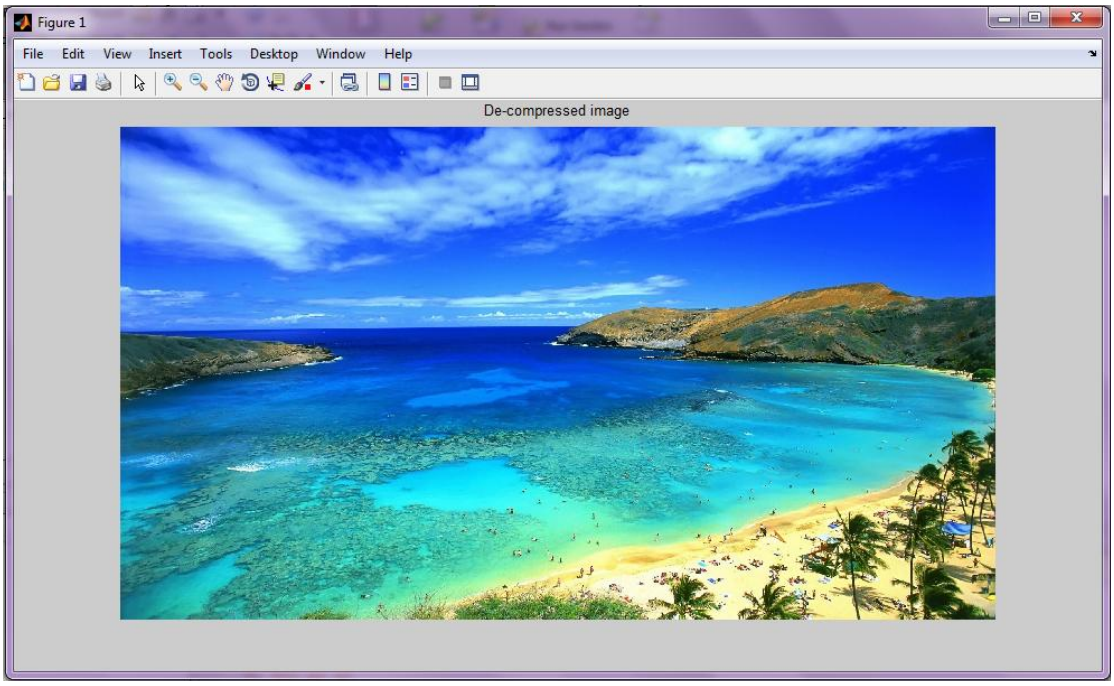
**The selected button is for choosing the image and to apply the Hybrid algorithm to compress it.**



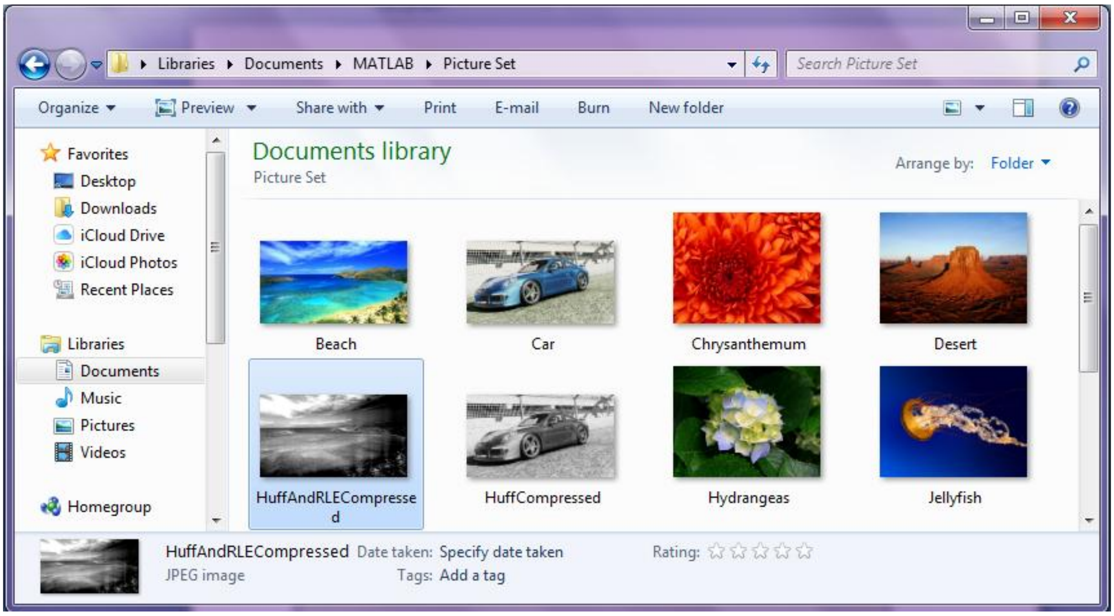
**This window shows the original image and also the image compressed with the Hybrid algorithm.**



**This window shows the decompressed image on which the Hybrid algorithm was applied.**




**This window shows that the compressed image is saved at the same location as that of the source file.**



# CHAPTER 6

# CONCLUSION

This project was motivated by the desire of improving the effectiveness of lossless image compression by improving the Huffman and RLE. We provided an overview of various existing coding standards lossless image compression techniques. We have proposed a high efficient algorithm which is implemented using the Huffman coding approach. The proposed method takes the advantages of the Huffman algorithm with the advantages of the RLE algorithm which is known for its simplicity and speed. The ultimate goal is to give a relatively good compression ratio and keep the time and space complexity minimum. The application software has been developed using MATLAB so as to meet the requirements of an image compression application, thereby ensuring quality performance. The image can be accessed, manipulated and retrieved very easily. To conclude this software has proved to be a user friendly interface. The application can be proved to be of great use to compress and decompression of the image.

# FUTURE WORK

We suggest for future work to use Huffman with another compression method and that enable to repeat the compression more than three times, and to investigate how to provide a high compression ratio for given images and to find an algorithm that decrease file. The experiment dataset in this project was somehow limited so applying the developed methods on a larger dataset could be a subject for future research and finally extending the work to the video compression is also very interesting. The application deals with all existing formats of images. This application is currently for windows OS we can also make it for MAC and Linux. I would be extending the application for some other doc format also and would be working more on compression algorithms. An android app can also be created to compress images in a mobile.

# REFERENCES


[1] A. Alarabeyyat, S. Al-Hashemi, T. Khdour, “Lossless Image Compression Technique Using Combination Methods”, Journal of Software Engineering and Applications, Vol-5, No.4, pp. 42-46, 2012.

[2] Debashis Chakraborty, Soumik Banerjee,, “Efficient Lossless Colour Image Compression Using Run Length Encoding and Special Character Replacement”, International Journal on Computer Science and Engineering, Vol-3, No.6, pp. 21-23, 2011.

[3] Rafeeq Al-Hashemi, Israa Wahbi Kamal, “A New Lossless Image Compression Technique Based on Bose, Chandhuri and Hocquengham (BCH) Codes”, International Journal of Software Engineering and Its Applications Vol-5, No. 3, pp.34-36, 2011.

[4] M. Fatih TALU, İbrahim TÜRKOĞLU, “Hybrid Lossless Compression Method For Binary Images”, IU-JEEE Vol-11, No. 2, pp. 12-14, 2011.

[5] Dalvir Kaur, Kamaljeet Kaur, “Data Compression on Columnar-Database Using Hybrid Approach (Huffman and Lempel-Ziv Welch Algorithm)”, International Journal of Advanced Research in Computer Science and Software Engineering, Vol-3, No. 11, pp. 47-50, 2013.

[6] Med Karim Abdmouleh, Atef Masmoudi, Med Salim Bouhlel, “A New Method Which Combines Arithmetic Coding with RLE for Lossless Image Compression”, Journal of Software Engineering and Applications, Vol-5, No. 6, pp. 41-44, 2012.

[7] Prof.Megha S.Chaudhari, Prof.S.S.Shirgan, " Implementation and Analysis of Efficient Lossless Image Compression Algorithm", International Journal of Engineering and Technical Research, Vol-2, No.6, pp.-37-39, 2014.

[8] R. C. Gonzalez, R. E. Woods and S. L. Eddins, "Digital Image Processing Using MATLAB," Pearson Prentice Hall, Upper Saddle River, 2003.

[9] www.dspguide.com/ch27/7.htm

[10] www.ou.edu/class/digitalmaedia/articles/Compressionmethods.htm

# APPENDIX

# CODE

## GUI

```
function varargout = GUI(varargin)
% GUI MATLAB code for GUI.fig
%      GUI, by itself, creates a new GUI or raises the existing
%      singleton*.
%
%      H = GUI returns the handle to a new GUI or the handle to
%      the existing singleton*.
%
%      GUI('CALLBACK',hObject,eventData,handles,...) calls the local
%      function named CALLBACK in GUI.M with the given input arguments.
%
%      GUI('Property','Value',...) creates a new GUI or raises the
%      existing singleton*.  Starting from the left, property value pairs are
%      applied to the GUI before GUI_OpeningFcn gets called.  An
%      unrecognized property name or invalid value makes property application
%      stop.  All inputs are passed to GUI_OpeningFcn via varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help GUI

% Last Modified by GUIDE v2.5 15-Apr-2015 09:56:47

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @GUI_OpeningFcn, ...
                   'gui_OutputFcn',  @GUI_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
```

```
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before GUI is made visible.
function GUI_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to GUI (see VARARGIN)

% Choose default command line output for GUI
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes GUI wait for user response (see UIRESUME)
% uiwait(handles.figure1);


% --- Outputs from this function are returned to the command line.
function varargout = GUI_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;


% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
[filename,pathname] = uigetfile('*.*');
if isequal(filename,0)

else
im=imread(fullfile(pathname, filename));
imfinfo(fullfile(pathname, filename))
im=imread(fullfile(pathname, filename));
jb=copyfile(fullfile(pathname,filename),fullfile(tempdir,filename));
set(handles.edit1,'String',filename);
```

```
im = double(im)/255;
I = rgb2gray(im);
subplot(211)
imshow(im)
size(im)
title('Original image');
img_dct=dct2(I);
img_pow=(img_dct).^2;
img_pow=img_pow(:);
[B,index]=sort(img_pow);%no zig-zag
B=flipud(B);
index=flipud(index);
compressed_dct=zeros(size(I));
coeff = 20000;% maybe change the value
for k=1:coeff
compressed_dct(index(k))=img_dct(index(k));
end
im_dct=idct2(compressed_dct);
subplot(212)
figure;imshow(im_dct);
title('huffman Compress Image');
Level=8;
Speed=0;
xC=cell(15,1);
[y, Res]=Huff06(xC, Level, Speed);
imwrite(im_dct,fullfile(pathname, 'HuffCompressed.jpg'));
imfinfo(fullfile(pathname, 'HuffCompressed.jpg'))
end

% --- Executes on button press in pushbutton2.
function pushbutton2_Callback(hObject, eventdata, handles)

% hObject    handle to pushbutton2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
%figure;imshow(fullfile(tempdir,get(handles.edit1,'String')));title('De-compressed
image');
input=get(handles.edit1,'String');
if (strcmp(input,'Selected File')==1)

else
path=fullfile(tempdir,get(handles.edit1,'String'));
img=imread(path);
imfinfo(path);
x=im2bw(img);
%copyfile(fullfile(tempdir,get(handles.edit1,'String')),fullfile('/Decompress/',get(hand
les.edit1,'String')));
% if iscell(x) % decoding
%   i = cumsum([ 1 x{2} ]);
%   j = zeros(1, i(end)-1);
```

```
%   j(i(1:end-1)) = 1;
%   data = x{1}(cumsum(j));
%     Huff04(x);
% end
figure;imshow(fullfile(tempdir,get(handles.edit1,'String')));title('De-compressed
image');
imwrite(img,get(handles.edit1,'String'));
end
% --- Executes on button press in pushbutton3.
function pushbutton3_Callback(hObject, eventdata, handles)
[filename,pathname] = uigetfile('*.*');
if isequal(filename,0)

else
I=imread(fullfile(pathname, filename));
imfinfo(fullfile(pathname, filename))
jb=copyfile(fullfile(pathname,filename),fullfile(tempdir,filename));
set(handles.edit2,'String',filename);
%I=imread('c:/gis4.jpg','jpg');
figure;imshow(I);title('original image');
level=graythresh(I);
imshow(I);title('Original')
bw=im2bw(I,level);
figure;imshow(bw);title('binary image');
a=bw'; a=a(:); a=a';
a=double(a);
rle(1)=a(1); m=2; rle(m)=1;
for i=1:length(a)-1
if a(i)==a(i+1)
rle(m)=rle(m)+1;
else
m=m+1; rle(m)=1; %dynamic allocation and initialization of next element of rle
end
end
display(rle);
imshow(bw);title('Compressed image');
imwrite(bw,fullfile(pathname, 'RleCompressed.jpg'));
imfinfo(fullfile(pathname, 'RleCompressed.jpg'));
end
% hObject    handle to pushbutton3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)


% --- Executes on button press in pushbutton4.
function pushbutton4_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton4 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

```
%figure;imshow(fullfile(tempdir,get(handles.edit2,'String')));title('De-compressed
image');
input=get(handles.edit2,'String');
if (strcmp(input,'Selected File')==1)

else
it=imread(fullfile(tempdir,get(handles.edit2,'String')));
imfinfo(fullfile(tempdir,get(handles.edit2,'String')));
x=im2bw(it);
%copyfile(fullfile(tempdir,get(handles.edit2,'String')),fullfile('Decompress',get(handle
s.edit2,'String')));
if iscell(x) % decoding
    i = cumsum([ 1 x{2} ]);
    j = zeros(1, i(end)-1);
    j(i(1:end-1)) = 1;
    data = x{1}(cumsum(j));

end
figure;imshow(fullfile(tempdir,get(handles.edit2,'String')));title('De-compressed
image');
imwrite(it,get(handles.edit2,'String'));
end
function edit1_Callback(hObject, eventdata, handles)
% hObject    handle to edit1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit1 as text
%        str2double(get(hObject,'String')) returns contents of edit1 as a double


% --- Executes during object creation, after setting all properties.
function edit1_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function edit2_Callback(hObject, eventdata, handles)
% hObject    handle to edit2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

```
% Hints: get(hObject,'String') returns contents of edit2 as text
%        str2double(get(hObject,'String')) returns contents of edit2 as a double


% --- Executes during object creation, after setting all properties.
function edit2_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in pushbutton5.
function pushbutton5_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton5 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
[filename,pathname] = uigetfile('*.*');
if isequal(filename,0)

else
im=imread(fullfile(pathname, filename));
imfinfo(fullfile(pathname, filename))
im=imread(fullfile(pathname, filename));
jb=copyfile(fullfile(pathname,filename),fullfile(tempdir,filename));
set(handles.edit1,'String',filename);
im = double(im)/255;
I = rgb2gray(im);
subplot(211)
imshow(im)
size(im)
title('Original image');
img_dct=dct2(I);
img_pow=(img_dct).^2;
img_pow=img_pow(:);
[B,index]=sort(img_pow);%no zig-zag
B=flipud(B);
index=flipud(index);
compressed_dct=zeros(size(I));
coeff = 20000;% maybe change the value
for k=1:coeff
compressed_dct(index(k))=img_dct(index(k));
end
```

```
im_dct=dct2(compressed_dct);
subplot(212)
figure;imshow(im_dct);
title('huffman and RLE Compress Image');
Level=8;
Speed=0;
xC=cell(15,1);
[y, Res]=Huff06(xC, Level, Speed);

imwrite(im_dct,fullfile(pathname, 'HuffAndRLECompressed.jpg'));
imfinfo(fullfile(pathname, 'HuffAndRLECompressed.jpg'))
end

% --- Executes on button press in pushbutton7.
function pushbutton7_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton7 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
input=get(handles.edit1,'String');
if (strcmp(input,'Selected File')==1)

else
path=fullfile(tempdir,get(handles.edit1,'String'));
img=imread(path);
imfinfo(path);
x=im2bw(img);
%copyfile(fullfile(tempdir,get(handles.edit1,'String')),fullfile('/Decompress/',get(hand
les.edit1,'String')));
% if iscell(x) % decoding
%   i = cumsum([ 1 x{2} ]);
%   j = zeros(1, i(end)-1);
%   j(i(1:end-1)) = 1;
%   data = x{1}(cumsum(j));
%    Huff04(x);
% end
figure;imshow(fullfile(tempdir,get(handles.edit1,'String')));title('D
e-compressed image');
imwrite(img,get(handles.edit1,'String'));
end
```

## HUFFMAN

```
function HK = HuffCode(HL,Display)
% HuffCode    Based on the codeword lengths this function find the Huffman
codewords
%
% HK = HuffCode(HL,Display);
% HK = HuffCode(HL);
% ----------------------------------------------------------------
% Arguments:
%  HL     length (bits) for the codeword for each symbol
%         This is usually found by the hufflen function
%  HK     The Huffman codewords, a matrix of ones or zeros
%         the code for each symbol is a row in the matrix
%         Code for symbol S(i) is: HK(i,1:HL(i))
%         ex: HK(i,1:L)=[0,1,1,0,1,0,0,0] and HL(i)=6 ==>
%             Codeword for symbol S(i) = '011010'
%  Display==1  ==> Codewords are displayed on screen, Default=0
% ----------------------------------------------------------------


if nargin<1
   error('huffcode: see help.')
end
if nargin<2
  Display = 0;
end
if (Display ~= 1)
   Display = 0;
end

N=length(HL);
L=max(HL);
HK=zeros(N,L);
[HLs,HLi] = sort(HL);
Code=zeros(1,L);
for n=1:N
   if (HLs(n)>0)
      HK(HLi(n),:) = Code;
      k = HLs(n);
      while (k>0)                % actually always!  break ends loop
         Code(k) = Code(k) + 1;
         if (Code(k)==2)
            Code(k) = 0;
            k=k-1;
         else
            break
         end
      end
   end
```

```
end

if Display
  for n=1:N
    Linje = ['  Symbol ',int2str(n)];
    for i=length(Linje):15
      Linje = [Linje,' '];
    end
    Linje = [Linje,'  gets code: '];
    for i=1:HL(n)
      if (HK(n,i)==0)
        Linje = [Linje,'0'];
      else
        Linje = [Linje,'1'];
      end
    end
    disp(Linje);
  end
end

return;
```

## RLE

```
function data = rle(x)
% data = rle(x) (de)compresses the data with the RLE-Algorithm
%   Compression:
%      if x is a numbervector data{1} contains the values
%      and data{2} contains the run lenths
%

if iscell(x) % decoding
    i = cumsum([ 1 x{2} ]);
    j = zeros(1, i(end)-1);
    j(i(1:end-1)) = 1;
    data = x{1}(cumsum(j));
else % encoding
    if size(x,1) > size(x,2), x = x'; end % if x is a column vector, tronspose
    i = [ find(x(1:end-1) ~= x(2:end)) length(x) ];
    data{2} = diff([ 0 i ]);
    data{1} = x(i);
end
```

## HYBRID

```
function varargout = Huff06(xC, ArgLevel, ArgSpeed)
% Huff06      Huffman encoder/decoder with (or without) recursive splitting
% Vectors of integers are Huffman encoded,
% these vectors are collected in a cell array, xC.
% If first argument is a cell array the function do encoding,
% else decoding is done.
%
% [y, Res] = Huff06(xC, Level, Speed);                 % encoding
% y = Huff06(xC);                                 % encoding
% xC = Huff06(y);                                 % decoding
% ----------------------------------------------------------------
% Arguments:
%  y        a column vector of non-negative integers (bytes) representing
%           the code, 0 <= y(i) <= 255.
%  Res      a matrix that sum up the results, size is (NumOfX+1)x4
%           one line for each of the input sequences, the columns are
%           Res(:,1) - number of elements in the sequence
%           Res(:,2) - zero-order entropy of the sequence
%           Res(:,3) - bits needed to code the sequence
%           Res(:,4) - bit rate for the sequence, Res(:,3)/Res(:,1)
%           Then the last line is total (which include bits needed to store NumOfX)
%  xC       a cell array of column vectors of integers representing the
%           symbol sequences. (should not be to large integers)
```

```
%          If only one sequence is to be coded, we must make the cell array
%          like: xC=cell(2,1); xC{1}=x; % where x is the sequence
%  Level    How many levels of splitting that is allowed, legal values 1-8
%          If Level=1, no further splitting of the sequences will be done
%          and there will be no recursive splitting.
%  Speed    For complete coding set Speed to 0. Set Speed to 1 to cheat
%          during encoding, y will then be a sequence of zeros only,
%          but it will be of correct length and the other output
%          arguments will be correct.
% ----------------------------------------------------------------

% SOME NOTES ON THE FUNCTION
% huff06 depends on other functions for Huffman code, and the functions in this file
%  HuffLen     - find length of codewords (HL)
%  HuffTabLen  - find bits needed to store Huffman table information (HL)
%  HuffCode    - find huffman codewords
%  HuffTree    - find huffman tree


global y Byte BitPos Speed Level

Mfile='Huff06';
Debug=0;    % note Debug is defined in EncodeVector and DecodeVector too

% check input and output arguments, and assign values to arguments
if (nargin < 1);
   error([Mfile,': function must have input arguments, see help.']);
end
if (nargout < 1);
   error([Mfile,': function must have output arguments, see help.']);
end

if (~iscell(xC))
   Encode=0;Decode=1;
   y=xC(:);            % first argument is y
else
   Encode=1;Decode=0;
   if (nargin < 3); Speed=0; else Speed=ArgSpeed; end;
   if (nargin < 2); Level=8; else Level=ArgLevel; end;
   if ((length(Speed(:))~=1));
      error([Mfile,': Speed argument is not scalar, see help.']);
   end
   if Speed; Speed=1; end;
   if ((length(Level(:))~=1));
      error([Mfile,': Level argument is not scalar, see help.']);
   end
   Level=floor(Level);
   if (Level < 1); Level=1; end;
   if (Level > 8); Level=8; end;
   NumOfX = length(xC);
```

```
end

if Encode
  Res=zeros(NumOfX,4);
  % initalize the global variables
  y=zeros(10,1);    % put some zeros into y initially
  Byte=0;BitPos=1;  % ready to write into first position
  % start encoding, first write VLIC to give number of sequences
  PutVLIC(NumOfX);
  if Debug
    disp([Mfile,' (Encode): Level=',int2str(Level),'  Speed=',int2str(Speed),...
        '  NumOfX=',int2str(NumOfX)]);
  end
  % now encode each sequence continuously
  Ltot=0;
  for num=1:NumOfX
    x=xC{num};
    x=full(x(:));        % make sure x is a non-sparse column vector
    L=length(x);Ltot=Ltot+L;
    y=[y(1:Byte);zeros(50+2*L,1)];  % make more space available in y
    % now find some info about x to better code it
    if (L>0)
      maxx=max(x); maxx=maxx(1);
      minx=min(x); minx=minx(1);
    else
      maxx=0;
      minx=0;
    end
    if (minx<0)
      Negative=1;
    else
      Negative=0;
    end
    if ( (((maxx*4)>L) || (maxx>1023)) && (L>1) && (maxx>minx))
      % the test for LogCode could be better, I think, (ver. 1.3)
      LogCode=1;    % this could be 0 if LogCode is not wanted
    else
      LogCode=0;
    end
    PutBit(LogCode);
    PutBit(Negative);
    I=find(x);                    % non-zero entries in x
    Sg=(sign(x(I))+1)/2;            % the signs may be needed later, 0/1
    x=abs(x);
    if LogCode
      xa=x;                       % additional bits
      x(I)=floor(log2(x(I)));
      xa(I)=xa(I)-2.^x(I);
      x(I)=x(I)+1;
    end
```

```
    [bits, ent]=EncodeVector(x);   % store the (abs and/or log) values
    if Negative                    % store the signs
      for i=1:length(Sg); PutBit(Sg(i)); end;
      bits=bits+length(Sg);
      ent=ent+length(Sg)/L;
    end
    if LogCode                     % store the additional bits
      for i=1:L
        for ii=(x(i)-1):(-1):1
          PutBit(bitget(xa(i),ii));
        end
      end
      bits=bits+sum(x)-length(I);
      ent=ent+(sum(x)-length(I))/L;
    end
    if L>0; Res(num,1)=L; else Res(num,1)=1; end;
    Res(num,2)=ent;
    Res(num,3)=bits;
  end
  y=y(1:Byte);
  varargout(1) = {y};
  if (nargout >= 2)
    % now calculate results for the total
    if Ltot<1; Ltot=1; end;   % we do not want Ltot to be zero
    Res(NumOfX+1,3)=Byte*8;
    Res(NumOfX+1,1)=Ltot;
    Res(NumOfX+1,2)=sum(Res(1:NumOfX,1).*Res(1:NumOfX,2))/Ltot;
    Res(:,4)=Res(:,3)./Res(:,1);
    varargout(2) = {Res};
  end
end

if Decode
  % initalize the global variables, y is set earlier
  Byte=0;BitPos=1;  % ready to read from first position
  NumOfX=GetVLIC;   % first read number of sequences
  if Debug
    disp([Mfile,'(Decode):  NumOfX=',int2str(NumOfX),'
length(y)=',int2str(length(y))]);
  end
  xC=cell(NumOfX,1);
  for num=1:NumOfX
    LogCode=GetBit;
    Negative=GetBit;
    x=DecodeVector;   % get the (abs and/or log) values
    L=length(x);
    I=find(x);
    if Negative
      Sg=zeros(size(I));
      for i=1:length(I); Sg(i)=GetBit; end;   % and the signs   (0/1)
```

```
      Sg=Sg*2-1;                    % (-1/1)
    else
      Sg=ones(size(I));
    end
    if LogCode        % read additional bits too
      xa=zeros(L,1);
      for i=1:L
        for ii=2:x(i)
          xa(i)=2*xa(i)+GetBit;
        end
      end
      x(I)=2.^(x(I)-1);
      x=x+xa;
    end
    x(I)=x(I).*Sg;
    xC{num}=x;
  end
  varargout(1) = {xC};
end

return    % end of main function, huff06

% the EncodeVector and DecodeVector functions are the ones
% where actual coding is going on.
% This function calls itself recursively
function [bits, ent] = EncodeVector(x, bits, HL, Maxx, Meanx)
global y Byte BitPos Speed Level
Debug=0;
Level = Level - 1;
MaxL=50000;       % longer sequences is split in the middle
L=length(x);
% first handle some special possible exceptions,
if L==0
  PutBit(0);      % indicate that a sequence is coded
  PutVLIC(L);     % with length 0 (0 is 6 bits)
  PutBit(0);      % 'confirm' this by a '0', Run + Value is indicated by a '1'
  bits=2+6;
  ent=0;
  Level = Level + 1;
  return   % end of EncodeVector
end
if L==1
  PutBit(0);      % indicate that a sequence is coded
  PutVLIC(L);     % with length 1 (6 bits)
  PutVLIC(x(1));  % containing this integer
  bits=1+2*6;
  if (x(1)>=16); bits=bits+4; end;
  if (x(1)>=272); bits=bits+4; end;
  if (x(1)>=4368); bits=bits+5; end;
  if (x(1)>=69904); bits=bits+5; end;
```

```
    if (x(1)>=1118480); bits=bits+4; end;
    ent=0;
    Level = Level + 1;
    return    % end of EncodeVector
end
if max(x)==min(x)
    PutBit(0);       % indicate that a sequence is coded
    PutVLIC(L);      % with length L
    for i=1:7; PutBit(1); end;   % write end of Huffman Table
    PutVLIC(x(1));   % containing this integer
    bits=1+6+7+6;
    if (x(1)>=16); bits=bits+4; end;
    if (x(1)>=272); bits=bits+4; end;
    if (x(1)>=4368); bits=bits+5; end;
    if (x(1)>=69904); bits=bits+5; end;
    if (x(1)>=1118480); bits=bits+4; end;
    if (L>=16); bits=bits+4; end;
    if (L>=272); bits=bits+4; end;
    if (L>=4368); bits=bits+5; end;
    if (L>=69904); bits=bits+5; end;
    if (L>=1118480); bits=bits+4; end;
    ent=0;
    Level = Level + 1;
    return    % end of EncodeVector
end
if (L <= 5)      % ver. 1.9 feb. 2010 KS
    PutBit(0);       % indicate that a sequence is coded
    PutVLIC(L);      % with length 1 (6 bits)
    bits=1+6;
    for i=1:L
        PutVLIC(x(i));   % containing this integer
        bits=bits+6;
        if (x(i)>=16); bits=bits+4; end;
        if (x(i)>=272); bits=bits+4; end;
        if (x(i)>=4368); bits=bits+5; end;
        if (x(i)>=69904); bits=bits+5; end;
        if (x(i)>=1118480); bits=bits+4; end;
    end
    ent=0;
    Level = Level + 1;
    return    % end of EncodeVector
end
% here we test if Run + Value coding should be done
I=find(x);   % the non-zero indices of x
if (L/2-length(I))>50
    Maxx=max(x);
    Hi=IntHist(x,0,Maxx);  % find the histogram
    Hinz=nonzeros(Hi);
    ent=log2(L)-sum(Hinz.*log2(Hinz))/L;  % find entropy
    % there are few non-zero indices => Run+Value coding of x
```

```
  x2=x(I);  % the values
  I=[I(:);L+1];   % include length of x
  for i=length(I):(-1):2; I(i)=I(i)-I(i-1); end;
  x1=I-1;   % the runs
  % code this as an unconditional split (like if L is large)
  if Speed
    Byte=Byte+1;    % since we add 8 bits
  else
    PutBit(0);       % this is idicated like when a sequence
    PutVLIC(0);      % of length 0 is coded, but we add one extra bit
    PutBit(1);       % Run + Value is indicated by a '1'
  end;
  [bits1, temp] = EncodeVector(x1);
  [bits2, temp] = EncodeVector(x2);
  bits=bits1+bits2+8;
  Level = Level + 1;
  return    % end of EncodeVector
end

if (nargin==1)
  Maxx=max(x);
  Meanx=mean(x);
  Hi=IntHist(x,0,Maxx);  % find the histogram
  Hinz=nonzeros(Hi);
  ent=log2(L)-sum(Hinz.*log2(Hinz))/L;  % find entropy
  HL=HuffLen(Hi);
  HLlen=HuffTabLen(HL);
  % find number of bits to use, store L, HL and x
  bits=6+HLlen+sum(HL.*Hi);
  if (L>=16); bits=bits+4; end;
  if (L>=272); bits=bits+4; end;
  if (L>=4368); bits=bits+5; end;
  if (L>=69904); bits=bits+5; end;
  if (L>=1118480); bits=bits+4; end;
  if Debug
    disp(['bits=',int2str(bits),'  HLlen=',int2str(HLlen),...
       '   HClen=',int2str(sum(HL.*Hi))]);
  end
else               % arguments are given, do not need to be calculated
  ent=0;
end
%
% Here we have: x, bits, L, HL, Maxx, Meanx, ent
if (L>MaxL)   % we split sequence anyway (and the easy way; in the middle)
  L1=ceil(L/2);L2=L-L1;
  x1=x(1:L1);x2=x((L1+1):L);
elseif ((Level > 0) & (L>10))
  xm=median(x); % median in MatLab is slow, could be calulated faster by using the
histogram
  x1=zeros(L,1);x2=zeros(L,1);
```

```
  x2(1)=x(1);i1=0;i2=1;
  for i=2:L
    if (x(i-1) <= xm)
      i1=i1+1; x1(i1)=x(i);
    else
      i2=i2+1; x2(i2)=x(i);
    end
  end
  x1=x1(1:i1);x2=x2(1:i2);
  % find bits1 and bits2 for x1 and x2
  L1=length(x1);L2=length(x2);
  Maxx1=max(x1);Maxx2=max(x2);
  Meanx1=mean(x1);Meanx2=mean(x2);
  Hi1=IntHist(x1,0,Maxx1);  % find the histogram
  Hi2=IntHist(x2,0,Maxx2);  % find the histogram
  HL1=HuffLen(Hi1);HL2=HuffLen(Hi2);
  HLlen1=HuffTabLen(HL1);
  HLlen2=HuffTabLen(HL2);
  bits1=6+HLlen1+sum(HL1.*Hi1);
  bits2=6+HLlen2+sum(HL2.*Hi2);
  if (L1>=16); bits1=bits1+4; end;
  if (L1>=272); bits1=bits1+4; end;
  if (L1>=4368); bits1=bits1+5; end;
  if (L1>=69904); bits1=bits1+5; end;
  if (L1>=1118480); bits1=bits1+4; end;
  if (L2>=16); bits2=bits2+4; end;
  if (L2>=272); bits2=bits2+4; end;
  if (L2>=4368); bits2=bits2+5; end;
  if (L2>=69904); bits2=bits2+5; end;
  if (L2>=1118480); bits2=bits2+4; end;
else
  bits1=bits;bits2=bits;
end
% Here we may have: x1, bits1, L1, HL1, Maxx1, Meanx1
% and               x2, bits2, L2, HL2, Maxx2, Meanx2
% but at least we have bits1 and bits2  (and bits)
if Debug
  disp(['Level=',int2str(Level),'  bits=',int2str(bits),'  bits1=',int2str(bits1),...
      '  bits2=',int2str(bits2),'  sum=',int2str(bits1+bits2)]);
end

if (L>MaxL)
  if Speed
    BitPos=BitPos-1;
    if (~BitPos); Byte=Byte+1; BitPos=8; end;
  else
    PutBit(1);      % indicate sequence is splitted into two
  end;
  [bits1, temp] = EncodeVector(x1);
  [bits2, temp] = EncodeVector(x2);
```

```
  bits=bits1+bits2+1;
elseif ((bits1+bits2) < bits)
  if Speed
    BitPos=BitPos-1;
    if (~BitPos); Byte=Byte+1; BitPos=8; end;
  else
    PutBit(1);      % indicate sequence is splitted into two
  end;
  [bits1, temp] = EncodeVector(x1, bits1, HL1, Maxx1, Meanx1);
  [bits2, temp] = EncodeVector(x2, bits2, HL2, Maxx2, Meanx2);
  bits=bits1+bits2+1;
else
  bits=bits+1;     % this is how many bits we are going to write
  if Debug
    disp(['EncodeVector: Level=',int2str(Level),'  ',int2str(L),...
        ' sybols stored in ',int2str(bits),' bits.']);
  end
  if Speed
    % advance Byte and BitPos without writing to y
    Byte=Byte+floor(bits/8);
    BitPos=BitPos-mod(bits,8);
    if (BitPos<=0); BitPos=BitPos+8; Byte=Byte+1; end;
  else
    % put the bits into y
    StartPos=Byte*8-BitPos;    % control variable
    PutBit(0);      % indicate that a sequence is coded
    PutVLIC(L);
    PutHuffTab(HL);
    HK=HuffCode(HL);
    for i=1:L;
      n=x(i)+1;    % symbol number (value 0 is first symbol, symbol 1)
      for k=1:HL(n)
        PutBit(HK(n,k));
      end
    end
    % check if one has used as many bits as calculated
    BitsUsed=Byte*8-BitPos-StartPos;
    if (BitsUsed~=bits)
      disp(['L=',int2str(L),'  max(x)=',int2str(max(x)),'  min(x)=',int2str(min(x))]);
      disp(['BitsUsed=',int2str(BitsUsed),'  bits=',int2str(bits)]);
      error(['Huff06-EncodeVector: Logical error, (BitsUsed~=bits).']);
    end
  end
end
Level = Level + 1;
return   % end of EncodeVector

function x = DecodeVector
global y Byte BitPos
MaxL=50000;     % as in the EncodeVector function (line 216)
```

```
if GetBit
  x1=DecodeVector;
  x2=DecodeVector;
  L=length(x1)+length(x2);
  if (L>MaxL)
    x=[x1(:);x2(:)];
  else
    xm=median([x1;x2]);
    x=zeros(L,1);
    x(1)=x2(1);
    i1=0;i2=1;
    for i=2:L
      if (x(i-1) <= xm)
        i1=i1+1; x(i)=x1(i1);
      else
        i2=i2+1; x(i)=x2(i2);
      end
    end
  end
else
  L=GetVLIC;
  if (L>5)
    x=zeros(L,1);
    HL=GetHuffTab;
    if length(HL)
      Htree=HuffTree(HL);
      root=1;pos=root;
      l=0;  % number of symbols decoded so far
      while l<L
        if GetBit
          pos=Htree(pos,3);
        else
          pos=Htree(pos,2);
        end
        if Htree(pos,1)          % we have arrived at a leaf
          l=l+1;
          x(l)=Htree(pos,2)-1;   % value is one less than symbol number
          pos=root;              % start at root again
        end
      end
    else    % HL has length 0, that is empty Huffman table
      x=x+GetVLIC;
    end
  elseif L>1  % ver 1.9 feb. 2010 KS
     x=zeros(L,1);
     for i=1:L
        x(i) = GetVLIC;
     end
  elseif L==0
    if GetBit
```

```
        % this is a Run + Value coded sequence
        x1=DecodeVector;
        x2=DecodeVector;
        % now build the actual sequence
        I=x1;     % runs
        I=I+1;
        L=length(I);  % one more than the number of values in x
        for i=2:L;I(i)=I(i-1)+I(i); end;
        x=zeros(I(L)-1,1);
        x(I(1:(L-1)))=x2;  % values
      else
        x=[];    % this was really a length 0 sequence
      end
    elseif L==1
      x=GetVLIC;
    else
      error('DecodeVector: illegal length of sequence.');
    end
end
return    % end of DecodeVector

% Functions to write and read the Huffman Table Information
% The format is defined in HuffTabLen, we repeat it here
% Function assume that the table information is stored in the following format
%       previous symbol is set to the initial value 2, Prev=2
%       Then we have for each symbol a code word to tell its length
%        '0'            - same length as previous symbol
%        '10'           - increase length by 1, and 17->1
%        '1100'         - decrease length by 1, and 0->16
%        '11010'        - increase length by 2, and 17->1, 18->2
%        '11011'        - One zero, unused symbol (twice for two zeros)
%        '111xxxx'      - set code length to CL=Prev+x (where 3 <= x <= 14)
%                         and if CL>16; CL=CL-16
%       we have 4 unused 7 bit code words, which we give the meaning
%        '1110000'+4bits - 3-18 zeros
%        '1110001'+8bits - 19-274 zeros, zeros do not change previous value
%        '1110010'+4bits - for CL=17,18,...,32, do not change previous value
%        '1111111'      - End Of Table

function PutHuffTab(HL)
global y Byte BitPos

HL=HL(:);
% if (max(HL) > 32)
%    disp(['PutHuffTab: To large value in HL, max(HL)=',int2str(max(HL))]);
% end
% if (min(HL) < 0)
%    disp(['PutHuffTab: To small value in HL, min(HL)=',int2str(min(HL))]);
% end
Prev=2;
```

```
ZeroCount=0;
L=length(HL);

for l=1:L
  if HL(l)==0
    ZeroCount=ZeroCount+1;
  else
    while (ZeroCount > 0)
      if ZeroCount<3
        for i=1:ZeroCount
          PutBit(1);PutBit(1);PutBit(0);PutBit(1);PutBit(1);
        end
        ZeroCount=0;
      elseif ZeroCount<19
        PutBit(1);PutBit(1);PutBit(1);PutBit(0);PutBit(0);PutBit(0);PutBit(0);
        for (i=4:-1:1); PutBit(bitget(ZeroCount-3,i)); end;
        ZeroCount=0;
      elseif ZeroCount<275
        PutBit(1);PutBit(1);PutBit(1);PutBit(0);PutBit(0);PutBit(0);PutBit(1);
        for (i=8:-1:1); PutBit(bitget(ZeroCount-19,i)); end;
        ZeroCount=0;
      else
        PutBit(1);PutBit(1);PutBit(1);PutBit(0);PutBit(0);PutBit(0);PutBit(1);
        for (i=8:-1:1); PutBit(1); end;
        ZeroCount=ZeroCount-274;
      end
    end
    if HL(l)>16
      PutBit(1);PutBit(1);PutBit(1);PutBit(0);PutBit(0);PutBit(1);PutBit(0);
      for (i=4:-1:1); PutBit(bitget(HL(l)-17,i)); end;
    else
      Inc=HL(l)-Prev;
      if Inc<0; Inc=Inc+16; end;
      if (Inc==0)
        PutBit(0);
      elseif (Inc==1)
        PutBit(1);PutBit(0);
      elseif (Inc==2)
        PutBit(1);PutBit(1);PutBit(0);PutBit(1);PutBit(0);
      elseif (Inc==15)
        PutBit(1);PutBit(1);PutBit(0);PutBit(0);
      else
        PutBit(1);PutBit(1);PutBit(1);
        for (i=4:-1:1); PutBit(bitget(Inc,i)); end;
      end
      Prev=HL(l);
    end
  end
end
for (i=7:-1:1); PutBit(1); end;      % the EOT codeword
```

```
return;  % end of PutHuffTab

function HL=GetHuffTab
global y Byte BitPos

Debug=0;
Prev=2;
ZeroCount=0;
HL=zeros(10000,1);
HLi=0;
EndOfTable=0;

while ~EndOfTable
  if GetBit
    if GetBit
      if GetBit
        Inc=0;
        for (i=1:4); Inc=Inc*2+GetBit; end;
        if Inc==0
          ZeroCount=0;
          for (i=1:4); ZeroCount=ZeroCount*2+GetBit; end;
          HLi=HLi+ZeroCount+3;
        elseif Inc==1
          ZeroCount=0;
          for (i=1:8); ZeroCount=ZeroCount*2+GetBit; end;
          HLi=HLi+ZeroCount+19;
        elseif Inc==2          % HL(l) is large, >16
          HLi=HLi+1;
          HL(HLi)=0;
          for (i=1:4); HL(HLi)=HL(HLi)*2+GetBit; end;
          HL(HLi)=HL(HLi)+17;
        elseif Inc==15
          EndOfTable=1;
        else
          Prev=Prev+Inc;
          if Prev>16; Prev=Prev-16; end;
          HLi=HLi+1;HL(HLi)=Prev;
        end
      else
        if GetBit
          if GetBit
            HLi=HLi+1;
          else
            Prev=Prev+2;
            if Prev>16; Prev=Prev-16; end;
            HLi=HLi+1;HL(HLi)=Prev;
          end
        else
          Prev=Prev-1;
```

```
          if Prev<1; Prev=16; end;
          HLi=HLi+1;HL(HLi)=Prev;
        end
      end
    else
      Prev=Prev+1;
      if Prev>16; Prev=1; end;
      HLi=HLi+1;HL(HLi)=Prev;
    end
  else
    HLi=HLi+1;HL(HLi)=Prev;
  end
end
if HLi>0
  HL=HL(1:HLi);
else
  HL=[];
end

if Debug
  % check if this is a valid Huffman table
  temp=sum(2.^(-nonzeros(HL)));
  if temp ~=1
    error(['GetHuffTab: HL table is no good, temp=',num2str(temp)]);
  end
end

return;  % end of GetHuffTab

% Functions to write and read a Variable Length Integer Code word
% This is a way of coding non-negative integers that uses fewer
% bits for small integers than for large ones. The scheme is:
%   '00'   +  4 bit  - integers from 0 to 15
%   '01'   +  8 bit  - integers from 16 to 271
%   '10'   + 12 bit  - integers from 272 to 4367
%   '110'  + 16 bit  - integers from 4368 to 69903
%   '1110' + 20 bit  - integers from 69940 to 1118479
%   '1111' + 24 bit  - integers from 1118480 to 17895695
%   not supported  - integers >= 17895696 (=2^4+2^8+2^12+2^16+2^20+2^24)
function PutVLIC(N)
global y Byte BitPos
if (N<0)
  error('Huff06-PutVLIC: Number is negative.');
elseif (N<16)
  PutBit(0);PutBit(0);
  for (i=4:-1:1); PutBit(bitget(N,i)); end;
elseif (N<272)
  PutBit(0);PutBit(1);
  N=N-16;
  for (i=8:-1:1); PutBit(bitget(N,i)); end;
```

```
elseif (N<4368)
  PutBit(1);PutBit(0);
  N=N-272;
  for (i=12:-1:1); PutBit(bitget(N,i)); end;
elseif (N<69940)
  PutBit(1);PutBit(1);PutBit(0);
  N=N-4368;
  for (i=16:-1:1); PutBit(bitget(N,i)); end;
elseif (N<1118480)
  PutBit(1);PutBit(1);PutBit(1);PutBit(0);
  N=N-69940;
  for (i=20:-1:1); PutBit(bitget(N,i)); end;
elseif (N<17895696)
  PutBit(1);PutBit(1);PutBit(1);PutBit(1);
  N=N-1118480;
  for (i=24:-1:1); PutBit(bitget(N,i)); end;
else
  error('Huff06-PutVLIC: Number is too large.');
end
return

function N=GetVLIC
global y Byte BitPos
N=0;
if GetBit
  if GetBit
    if GetBit
      if GetBit
        for (i=1:24); N=N*2+GetBit; end;
        N=N+1118480;
      else
        for (i=1:20); N=N*2+GetBit; end;
        N=N+69940;
      end
    else
      for (i=1:16); N=N*2+GetBit; end;
      N=N+4368;
    end
  else
    for (i=1:12); N=N*2+GetBit; end;
    N=N+272;
  end
else
  if GetBit
    for (i=1:8); N=N*2+GetBit; end;
    N=N+16;
  else
    for (i=1:4); N=N*2+GetBit; end;
  end
end
```

```
return

% Functions to write and read a Bit
function PutBit(Bit)
global y Byte BitPos
BitPos=BitPos-1;
if (~BitPos); Byte=Byte+1; BitPos=8; end;
y(Byte) = bitset(y(Byte),BitPos,Bit);
return

function Bit=GetBit
global y Byte BitPos
BitPos=BitPos-1;
if (~BitPos); Byte=Byte+1; BitPos=8; end;
Bit=bitget(y(Byte),BitPos);
return;

% this function is a variant of the standard hist function
function Hi=IntHist(W,i1,i2)
W=W(:);
%if (rem(i1,1) | rem(i2,1));   error('Non integers'); end;
L=length(W);
Hi=zeros(i2-i1+1,1);
if (i2-i1)>50
  for l=1:L
    i=W(l)-i1+1;
    Hi(i)=Hi(i)+1;
  end
else
  for i=i1:i2
    I=find(W==i);
    Hi(i-i1+1)=length(I);
  end
end
return;
```

## MATRIX TO VECTOR

```
function xC = Mat2Vec(W, Method, K, L)
% Mat2Vec     Convert an integer matrix to a cell array of vectors,
% several different methods are possible, most of them are non-linear.
% The inverse function is also performed by this function,
% to use this first argument should be a cell array instead of a matrix.
%

Mfile='Mat2Vec';
Debug=0;

% check input and output arguments, and assign values to arguments
if (nargin < 2);
   error([Mfile,': function must have two input arguments, see help.']);
end
if (nargout ~= 1);
   error([Mfile,': function must have one output arguments, see help.']);
end

if (~iscell(W))
   ToSeq=1;    % transform matrix W to xC
   if (nargin < 3); K=size(W,1); end;
   if (nargin < 4); L=size(W,2); end;
else
   ToSeq=0;    % transform cell array xC to W
   xC=W;
   clear W
   if (nargin < 4)
      error([Mfile,': function must have four input arguments, see help.']);
   end
end

% check given Method
Method=floor(Method);
if Method<0; Method=0; end;
if Method>19; Method=0; end;
% find number of sequences in xC from Method
if     (Method== 0); xCno=1;
elseif (Method== 1); xCno=2;
elseif (Method== 2); xCno=1;
elseif (Method== 3); xCno=2;
elseif (Method== 4); xCno=1;
elseif (Method== 5); xCno=3;
elseif (Method== 6); xCno=2;
elseif (Method== 7); xCno=2;
elseif (Method== 8); xCno=K;
elseif (Method== 9); xCno=2*K;
elseif (Method==10); xCno=log2(K)+1;
elseif (Method==11); xCno=2*log2(K)+2;
```

```
elseif (Method==12); xCno=K;
elseif (Method==13); xCno=2*K;
elseif (Method==14); xCno=log2(K)+1;
elseif (Method==15); xCno=2*log2(K)+2;
elseif (Method==16); xCno=1+(3/2)*log2(K);
elseif (Method==17); xCno=2+3*log2(K);
elseif (Method==18); xCno=1+(3/2)*log2(K);
elseif (Method==19); xCno=2+3*log2(K);
else                 xCno=0;
end;
%
if ToSeq
  [k,l]=size(W);
  if ((k~=K) || (l~=L))
    error([Mfile,': illegal size of W matrix, see help.']);
  end
  xC=cell(xCno,1);
  if sum(Method==[4:7,12:15,18,19])
    % make W with only positive values
    W=W*2;
    I=find(W<0);
    W(I)=-W(I)-1;
  end
else
  temp=length(xC);
  if temp~=xCno
    error([Mfile,': size of xC does not correspond to Method, see help.']);
  end
  W=zeros(K,L);
end

if Method==0                        % direct by columns
  if ToSeq
    xC{1}=W(:);
  else
    W=reshape(xC{1},K,L);
  end
elseif ((Method==1) || (Method==6))     % runs and values, column by column
  if ToSeq
    I=find(W(:));
    xC{2}=W(I);  % values
    for i=length(I):(-1):2; I(i)=I(i)-I(i-1); end;
    xC{1}=I-1;    % runs
  else
    I=xC{1};      % runs
    I=I+1;
    for i=2:length(I);I(i)=I(i-1)+I(i); end;
    W(I)=xC{2};  % values
  end
end
```

```
if Method==2                    % direct by rows
  if ToSeq
    W=W';
    xC{1}=W(:);
    W=W';
  else
    W=reshape(xC{1},L,K)';
  end
end
if ((Method==3) || (Method==7))     % runs and values, row by row
  if ToSeq
    W=W';
    I=find(W(:));
    xC{2}=W(I);  % values
    for i=length(I):(-1):2; I(i)=I(i)-I(i-1); end;
    xC{1}=I-1;    % runs
    W=W';
  else
    W=zeros(L,K);
    I=xC{1};      % runs
    I=I+1;
    for i=2:length(I);I(i)=I(i-1)+I(i); end;
    W(I)=xC{2};  % values
    W=W';
  end
end
if Method==4                    % EOB coded
  if ToSeq
    xC{1}=eob3(W);
  else
    W=eob3(xC{1},K);
  end
end
if Method==5                    % EOB coded, three sequences
  if ToSeq
    [xC{1},xC{2},xC{3}]=eob3(W);
  else
    W=eob3(xC{1},xC{2},xC{3},K);
  end
end
if ((Method==8) || (Method==12))    % each row coded as one sequence
  if ToSeq
    for k=1:K
      xC{k}=W(k,:)';
    end
  else
    for k=1:K
      W(k,:)=xC{k}';
    end
  end
```

```
end
if ((Method==9) || (Method==13))    % each row coded as runs and values
  if ToSeq
    for k=1:K
      I=find(W(k,:));
      if ~isempty(I)
        xC{2*k}=W(k,I)';    % values
        for i=length(I):(-1):2; I(i)=I(i)-I(i-1); end;
        xC{2*k-1}=(I-1)';    % runs
      else
        if Debug
          display('empty sequence.');
        end
        xC{2*k}=[];
        xC{2*k-1}=[];
      end
    end
  else
    for k=1:K
      I=xC{2*k-1};      % runs
      I=I+1;
      for i=2:length(I);I(i)=I(i-1)+I(i); end;
      W(k,I)=xC{2*k}';  % values
    end
  end
end
if ((Method==10) || (Method==14))   % each subband is coded as one sequence
  if rem(log2(K),1)
    error('Logical error: K is not a power of 2.');
  end
  i1=1;i2=1;
  if ToSeq
    for k=1:(log2(K)+1)
      xC{k}=reshape(W(i1:i2,:),L*(i2-i1+1),1);
      i1=i2+1;
      i2=i2*2;
    end
  else
    for k=1:(log2(K)+1)
      W(i1:i2,:)=reshape(xC{k},i2-i1+1,L);
      i1=i2+1;
      i2=i2*2;
    end
  end
end
if ((Method==11) || (Method==15))   % each subband is coded as runs and values
  if rem(log2(K),1)
    error('Logical error: K is not a power of 2.');
  end
  i1=1;i2=1;
```

```
  if ToSeq
    for k=1:(log2(K)+1)
      temp=reshape(W(i1:i2,:),L*(i2-i1+1),1);
      I=find(temp);
      xC{2*k}=(temp(I))';    % values
      for i=length(I):(-1):2; I(i)=I(i)-I(i-1); end;
      xC{2*k-1}=(I-1)';    % runs
      i1=i2+1;
      i2=i2*2;
    end
  else
    for k=1:(log2(K)+1)
      I=xC{2*k-1};      % runs
      I=I+1;
      for i=2:length(I);I(i)=I(i-1)+I(i); end;
      temp=zeros(i2-i1+1,L);
      temp(I)=xC{2*k};         % values
      W(i1:i2,:)=temp;
      i1=i2+1;
      i2=i2*2;
    end
  end
end
% new methods June 5. 2009
if ((Method==16) || (Method==18))   % each subband is coded as one sequence
  if rem(log2(K),2)
    error('Logical error: K is not a power of 4.');
  end
  nivaa = log2(K)/2;
  v = [1,2; 3,4]; vm = 4;
  for i = 1:(nivaa-1)
     v = [v, (vm+1)*ones(size(v)); (vm+2)*ones(size(v)), (vm+3)*ones(size(v))];
     vm = vm+3;
  end
  if ToSeq
    for k=1:vm
      xC{k} = reshape(W(find(v(:)==k),:), L*sum(v(:)==k), 1);
    end
  else
    for k=1:vm
      W(find(v(:)==k),:) = reshape(xC{k}, sum(v(:)==k), L);
    end
  end
end
if ((Method==17) || (Method==19))   % each subband is coded as runs and values
  if rem(log2(K),2)
    error('Logical error: K is not a power of 4.');
  end
  nivaa = log2(K)/2;
  v = [1,2; 3,4]; vm = 4;
```

```
  for i = 1:(nivaa-1)
     v = [v, (vm+1)*ones(size(v)); (vm+2)*ones(size(v)), (vm+3)*ones(size(v))];
     vm = vm+3;
  end
  if ToSeq
    for k=1:vm
      temp = reshape(W(find(v(:)==k),:), L*sum(v(:)==k), 1);
      I=find(temp);
      xC{2*k}=(temp(I))';    % values
      for i=length(I):(-1):2; I(i)=I(i)-I(i-1); end;
      xC{2*k-1}=(I-1)';    % runs
    end
  else
    for k=1:vm
      I=xC{2*k-1};      % runs
      I=I+1;
      for i=2:length(I);I(i)=I(i-1)+I(i); end;
      temp=zeros(i2-i1+1,L);
      temp(I) = xC{2*k};         % values
      W(find(v(:)==k),:) = reshape(temp, sum(v(:)==k), L);
    end
  end
end

if ~ToSeq
  if sum(Method==[4:7,12:15])
    W=W/2;
    I=find(rem(W,1));
    W(I)=-W(I)-0.5;    % make negative values in W appear again
  end
  xC=W;                % must return with W
end

return
```