PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA

Ministry of Higher Education and Scientific Research

UNIVERSITY OF MOHAMMED EL BACHIR EL IBRAHIMI

BORDJ BOU-ARRERIDJ

FACULTY OF SCIENCE AND TECHNOLOGY

ELECTRONICS DEPARTMENT

Memory

Presented to obtain

MASTER'S DIPLOMA

FILIERE: **ELECTRONICS**

SPECIALITY: INDUSTRIAL ELECTRONICS

By

- **ADEL BENAMIDA**
- **SEDRATI NASREDDINE**

Entitled

Data-Hiding Algorithms Implementation in a C++ Environment

Evaluated on : ……

Evaluated by:

First & Last Name		*Grade*	*Institution*
Mr. ABED TAREK	***President***	***MAA***	***Univ-BBA***
M.	***Examiner***		***Univ-BBA***
M.	***Supervisor***		***Univ-BBA***

University Year 2020/2021

ABSTRACT

Data hiding is a technique for carrying data in a suitable carrier for secure communication. Data hiding technique provides authentication and secure communication but there may be a chance of loss of carrier during communication. Data hiding techniques are mainly used for authentication, media registration, copyright protection, etc. But in the field of medicine and military distortion of the original image is not allowed. So, it needs high secure data hiding techniques. To overcome the problem faced during extracting the carrier with distortion was removed by RDH (Reversible Data Hiding) techniques. RDH technique gets the original data after extracting the secret encrypted data. RDH techniques are classified based on implementing method. In this paper we discussed about different techniques based on histogram shifting, compression embedding and different expansion for RDH are discussed.

TABLE OF CONTENTS

FIGURES

ABBREVIATIONS

GENERAL INTRODUCTION

A digital image is an image composed of picture elements, also known as pixels, each with finite, discrete quantities of numeric representation for its intensity or gray level that is an output from its two-dimensional functions fed as input by its spatial coordinates denoted with x, y on the x-axis and y-axis, respectively. Depending on whether the image resolution is fixed, it may be of vector or raster type. By itself, the term "digital image" usually refers to raster images or bitmapped images (as opposed to vector images). (22, s.d.)

Watermarking is a method of embedding useful information into a digital work (especially, thus, audio, image, or video) for the purpose of copy control, content authentication, distribution tracking, broadcast monitoring, etc. The distortion introduced by embedding the watermark is often constrained so that the host and the watermarked work are perceptually equivalent. However, in some applications, especially in the medical, military, and legal domains, even the imperceptible distortion introduced in the watermarking process is unacceptable. This has led to an interest in reversible watermarking, where the embedding is done in such a way that the information content of the host is preserved. This enables the decoder to not only extract the watermark, but also perfectly reconstruct the original host signal from the watermarked work. (24)

Due to information explosion, growth of digital content is ever increasing in the order of peta bytes. At present Internet has become a powerful source of information to the end users, at the same time unauthorized and copyright violated information is also easily available making them unsure about the originality of information. On the other Hand, establishing the legal ownership of digital content is important for the content providers. In such scenario, availability of authentic information from authorized sources is what the end users and content providers will be looking for. To ensure authenticity and legal ownership of digital content, various methods such as steganography cryptography, and watermarking have been proposed in the literature. Digital watermarking Is one such powerful information hiding technique extensively used to overcome illegal copying, modifying, and redistributing the digital content. (26)

So, to overcome the information explosion we must implement watermarking algorithms, and this what we'll be doing in the next chapters.

Our work is split into 4 chapters:

- General Overview of RDH techniques
- The theory of DE & PE data-hiding algorithms
- Implementing the algorithms
- Comparing the results

Chapter I Reversible Data Hiding Techniques

I.1 Introduction

This Internet has become more popular and common now-a-days. High risk is involved in sending the important data through the net. Data hiding is the one which provides secure communication without any loss of data. Number of approaches are present for data hiding techniques. Two main common approaches for protecting information leakage are Data hiding and Encryption. The former is used to protect the data, whereas the latter is used to protect the hidden data. Two main techniques Digital Steganography and watermarking are mainly used for communicating the secret data in appropriate carriers like audio, video, and image files. But these techniques may distort the original data after extracting the secret data. Data hiding techniques are mainly used for authentication, media registration, copyright protection, etc. But it is a problem for sensitive applications, such as medical images and military images. So Reversible data hiding (RDH) is best technique for these special applications, which aims to recover both embedded data and the original image. From the last few years many RDH algorithms have been proposed such as difference expansion (DE) based methods, lossless image compression-based methods, integer-to-integer transform-based methods, dual-image-based methods and histogram shifting (HS) based methods.

RDH embeds hidden data known as secret data, invisible data into a digital image known as cover image in a reversible manner

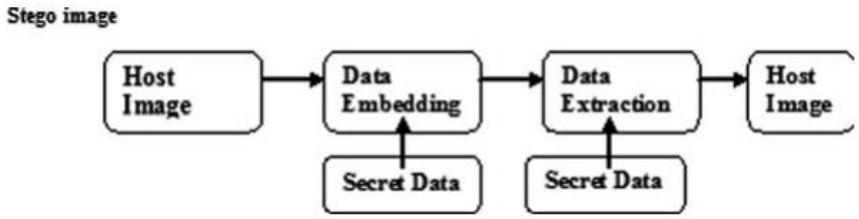

Fig. 1. Reversible data hiding

Figure 1 Reversible data hiding

The main task of RDH technique is getting the original image after extracting the secret data with little distortions and good quality. For the security purpose, RDH embeds some digitized information, so that only the authorized used can extract the secret data and restore the original image. An information hiding systems consisting of four different aspects.

- Security: refers to the ability to protect the secret key by an unauthorized user.
- Robustness: Ability to handle the modifications on the medium without distorting the secret information.
- Perceptibility: ability to find the secret information.
- Capacity: It refers to the amount of information the media can accommodate.

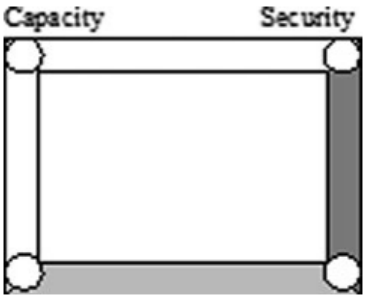

Figure 2 Characteristics of Reversible Data Hiding

RDH algorithms are mainly divides into three categories. The first kind of algorithm deals with lossless compression (1) embedding framework (LC). The algorithms first compute a pixel rate and then compressed. After that the left-out space will be allocated with messages. The second kind of algorithm is mainly based on DE (difference Expansion) (2). In this technique the difference between the pixel pair is extended to get the LSB's (Least Significant Bits) all values as zeroes so that it can be used for adding the data. The third technique is mainly based on HS (Histogram) (3). The value of histogram is always irregular, and the values can be modified by using histogram bins. All RDH techniques include 2 steps:

- By considering predicting errors (PE) host sequence with small entropy is generated.
- By changing the histogram values using HS or DE the messages are embedded reversibly into host sequence.

This results in payload maximization in over all or a given distortion condition. From the above techniques the HS based Techniques are widely used. HS methods are again divided into 3 types.

- Histogram Shifting
- Prediction Error Histogram Shifting
- Difference Histogram Shifting.

These techniques are mainly used to find the prediction error histogram. On modifying the histogram value, the host image is embedded reversibly along with the message. From last few years many algorithms of DHS (5) and PEHS (6) are proposed and they became very famous because of its high fidelity and huge embedding capability.

I.2 Literature Review on RDH.

Mintzer et al. (7) initiated reversible watermarks as a visible pattern. In this technique the image is marked with reversible visible watermark and that image is posted in internet and that water mark image will look like a puzzle. The user must pay money to remove that watermark after that the user will get the original image. Eastman Kodak (8) is the first one who to copyright the lossless invisible watermark. The similar thought was being proposed to extend the patchwork algorithm (9) but the outcome is not adequate. These technique cause salt and pepper visual artifacts. To get the watermarks here we used magnitude comparisons, which failed and cause visual artifacts. To avoid this problem Fridrich et al. (10) introduced a technique to change a bit plane of the image for watermarking and they extracted a complete bit plane and compress the losslessly for filling the watermark in the output space. But we get

disturbing artifacts due to different bit planes because the capacity may change from one image to another image. Fridrich's technique works strictly for the environment wherever the watermarked image is processed without any loss and results in modification of bits and the bit plane will contain the payload will concern the entropy synchronization and it may lose the hidden data permanently. To overcome the Fridrich's technique Vleeschouwer et al. proposed a technique. Vleeschouwer et al. (11) proposed a method that overcame Fridrich's method. Here they used circular interpretation of bijective transformations. Here they used circular interpretation of bijective transformations. Circle is mapped with the histograms for groups of pixels that was operated by the transform. The relative orientation among the histograms of two groups conveyed one bit of information. Here the reversibility process did not experience artifacts and the wrapped pixels were not altered.

Tian (12) is the first to use different expansion transform, the expansion is applied on a pixel pair to develop low distorted with high-capacity watermark. First the algorithm divides the image into pair of pixels which does not cause either overflow or underflow. Each single bit was embedded into difference pair of pixels. This technique results in embedding as high as 1 bit per one pixel in a single pass. Alattar (13) used vectors of an arbitrary size as parameter to difference expansion transforms. The algorithm results in computationally higher capacity with small distortion in image. Wang et al. (14) suggested to use 2D-vector maps for RDH. Two RDH difference expansion ideas were discovered by them. First idea, cover data is the coordinates of vertices and modifying the differences between the adjacent coordinates is used to hide the data. Second idea calculates the Manhattan distances between the neighboring vertices as the cover data and embedded the hidden data altering the differences among the neighboring distances. Here the two ideas result in high-capacity maps with high associated coordinates. The authors Dinu Cultic and Jean Mare has discussed about spatial domain reversible watermarking technique based on RCM contrast mapping and this technique results in high-capacity data embedding without any secondary compression stage, the remaining space is used by the LSBs for hiding the data. For improving RDH characteristics many different expansion transforms were concentrated. Xinpeng initially introduced RDH in encrypted image (15). In this real image is encrypted by a cipher text which is used to embed the secret information. In this technique, we can successfully extract the hidden data and the real image was completely claimed by using spatial correlation feature of the real image. By using this scheme, the author Xinpeng extended the idea of RDH in encrypted image. In this technique we introduced a new technique by using keys. In embedding phase, the data owner will encrypt the real uncompressed image by using encrypted key. LSB of the image is compressed by using the data hiding key and it creates some extra space so that the user can accommodate any extra data by the data hider. In this we get three possibilities that happens in encrypted image which contains data.

- When the receiver is having the data hiding key, then the user can extract the secret data without knowing the content of the image.
- In the next case if the receiver is having encryption key, then he can decrypt the received data to get an image which is like the original image, but he cannot extract the secret data.
- If the receiver is having both data hiding key and encryption key, then he can extract the hidden data and recover the original data without any loss of data. But the hidden data should not be too big.

But the problem in this technique is it did not utilize the pixels completely when analyzing the smoothness of every pixel and borders of the neighboring blocks were not considered. To avoid the problems in this technique the Hong (16) introduced a new smooth evaluation technique in which the pixels are fully exploited for calculating the pixel variation in images. For confusing blocks, we presented a side match mechanism to analyze the smoothness. In this technique (17) it automatically converts the large size secret data into a hidden segment visible mosaic image of same size. By dividing the secret image into fragments and converting the color feature to those of the equal blocks of the output image generated the mosaic image.

In (18)technique it includes three segments: content owner, the recipient, and the data hider. The owner of the data will encrypt the real image and upload the same into the target server. The recipient is the one who extracts the data by using secret key. The data hider is the one who divides the encrypted data into three groups and embed the message into each group which generates the marked encrypted image. By using the above technique, we can obtain the quality image on decryption by the receiver by using decryption key. If both keys are available, then we can get lossless original image. In distributed source coding was implemented in RDH encrypted images. Once the data owner encrypts the real image by using stream cipher, the data hider will compress some series of bits which is taken from the encrypted image to make it as a room for secret data. The selected series of bits is encoded by using low-density parity check codes by slepian-wolf. By using this he proposed (19) a new RDH scheme over encrypted domain. Here by using public key alteration mechanism we achieved data embedding, by which we cannot access encryption key. At the receiver side we use SVM classifier to differentiate between the encrypted and non-encrypted images and decodes the message from the encrypted image. Now a day's cloud computing is one of the emerging technologies so RDH also included the cloud communication usage (20). RDH-EI technique which is based on reversible image transformation was proposed. Unlike the earlier encryption schemes where cipher texts attracted the probing cloud, RIT-based framework allowed the operator to convert the content of original image into the content of another image of the same size. The converted image, that seemed to be the target image, was used as the "encrypted image," by the cloud. Another work using the cloud was implemented by the authors in (21). They proposed RDH scheme where the cipher text images were scrambled by public-key cryptosystems with probabilistic and homomorphic properties. In this scheme, the cipher text pixels were substituted with new values embedding the hidden data into different least significant bit planes of cipher text pixel using multilayer wet paper coding. The embedded data was then separated from the encrypted domain successfully.

I.3 Spatial domain techniques

In spatial domain steganography bits in the pixel's values are changed to hide the data. It deals with image pixels. Spatial techniques are mainly used for altering the individual pixels values.

I.4 Transform domain techniques

Transform domain is based on frequency components. In this domain, by using different transforms, the image is converted into the frequency domain from the image in the spatial domain. After that

embedding process will be done in appropriate transform coefficients. Transform domain is less exposed to compression, cropping etc.

Various techniques used for Data hiding is discussed below

I.5 Spatial Domain Techniques

I.5.a LSB substitution

LSB substitution is one of the data hiding technique which keeps the secret message into an image by hiding that data or message. So that the attacker will not be able to read the message. It selects 8-bit grayscale images as cover media for hiding the information which is called as cover images. LSB substitution gives stego-image, by applying local pixel adjustment process, the quality can be improved in stego-images. LPAP will not be optimal because it considers the last three LSB and fourth bit and not all. So, it cannot be used in hiding. Hence it uses Optimal Pixel Adjustment Process (OPAP).

Worst Mean Square Error (WMSE) gained by OPAP is less than 1⁄2 of that gained by LSB substitution technique. OPAP is applied to increase the capacity of simple LSB, and the quality of stego-image is improved effectively along with low extra computational complexity. LSB is the widely used simplest method where there is less chance for degradation of the original image.

I.5.b ISB Technique

ISB embedding technique keeps watermark pixels in empty/filled region in the place of original image pixels and it keeps the watermark pixels very closed. This technique mainly focuses on testing watermark pixel values. In grayscale images, it has 8-bit planes. while first bit planes include set of Most Significant Bits (MSB), while last bit planes include LSB and the sets in-between from second to seventh-bit planes are called as Intermediate Significant Bits (ISB). The value of a pixel between the range of middle and edge are the pixels in which the watermarked data can be protected from various attacks and it will retain less distortion of the watermarked image. Encryption of watermarked image is done by a random pixel manipulation technique to improve system security. Here robustness is improved by, embedding information according to the blocks of pixels.

I.6 Reversible Data Hiding Techniques

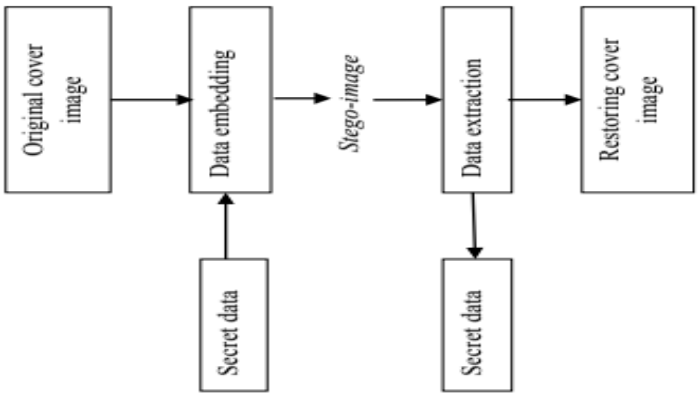

Figure 3 RDH Block Diagram

I.6.a Difference Expansion

Reversible data embedding can also be called as lossless data embedding. In reversible data embedding, it restores the original image. For reversible data embedding, quality degradation of the image should be low after data is embedded. This technique provides reversible data embedding, imperceptibility, enhanced capacity, and high visual quality for digital images. Here Difference Expansion provides extra storage by investigating repetition in the content of the image, and the main importance of this technique is limit of payload capacity and visual quality of embedded images, it is fragile technique if there is reversible data embedding because if the embedded image is lossy compressed, there will be no restoration of original content and the decoder will find it is not authentic.

I.6.b Interpolation technique

Interpolation is one of the techniques for Reversible Data Hiding. The original cover image is separated as five by five blocks among non-overlapping boundaries, and it is minimized into three-by-three blocks of each five-by-five block and generate again into five-by-five blocks with the help of interpolation technique. Secret data is embedded in the position where the total dissimilarity value between the interpolation value and the original Gray level is smaller than a threshold in the middle of the four three by three overlapping blocks of every five-by-five block. Here embedding steps are followed to take out embedded data with the help of an embedded region location. After that, by shuffling back the order secret data can be found. This technique provides high quality with imperceptibility and cover image can be recovered back after extracting the secret data.

I.6.c Prediction Error Expansion

PEE is one of the techniques for hiding the data which is considered for embedding the predictive errors. This technique enhances the accuracy in prediction of one channel through deriving a benefit from edge information from another channel. The prediction-error is reduced consequently; the performance of the algorithm is enhanced according to the rate of data hiding against distortion in embedding. Most vital role of the algorithm is improving the accuracy of prediction by utilizing the relationship between the channels. It also enhances the traditional sorting strategy by taking overflow/underflow issue into account.

I.6.d Histogram shifting

Histogram shifting is used to embed data or message in cover media by shifting the histogram of the image. This technique embeds data by shifting peak and zero points by detecting them in the histogram of the image. This technique divides the input image into blocks and histogram shifting is done on each block, by doing this data hiding capacity and visual quality is enhanced. It also provides a high capacity of data hiding with low distortion.

I.6.e Recursive histogram modification

RHM is one of the new techniques used for Reversible Data Hiding. This method embeds messages using the decompression and compression process recursively. It embeds the message by dividing the host set into disjoint blocks and by adjusting the histogram of every block recursive manner. The

message which is going to be embedded is encrypted before embedding. In the embedding process, the host set is divided as disjoint blocks for n blocks (n-1) that belongs to the same length and the last block have larger length. After dividing the host sequence message gets embedded into each block. It makes an equivalency between Reversible Data Hiding and lossless data compression and this technique gives improvements that are very important for larger images.

I.6.f PWLC data hiding technique

PWLC (Pair-Wise Logical Computation) is the proposed RDH technique for binary images. The main drawback of PWLC is, it fails to take out the secret data and it does not restore the original image. This technique either it uses spread spectrum technique or any other compression technique. For storing payload in host image, it uses XOR binary operations. The host images are scanned in some type of order. The sequences “000000” or “111111” that is situated close to the image boundaries are selected for hiding the data content.

I.6.g DHTC data hiding technique

DHTC (Data Hiding by Template ranking with symmetrical Central pixels) is defined as RDHTC which is depend on nonreversible data hiding. DHTC selects only imperceptible pixels to inject the secret data. The images that are marked by DHTC have brilliant visual quality, and there will be no salt-and-pepper noise. The result of a given pattern is according to the matching of the template with the lowest impact.

I.7 Frequency Domain Technique

I.7.a Discrete wavelets transform

Wavelet is a most secure domain for embedding of watermark and data hiding. It represents the signal in another form by transforming the signal, and it will not change the content in it. Here the original image is converted into frequency domain which is called as frequency domain technique. DWT decomposes the hierarchy of image by providing both descriptions of spatial and frequency. In DWT, an image will be decomposed in 3 directions of spatial that is vertical, horizontal, and diagonal in result it divides an image into 4 components that are high-low, low_low, low_high, high_high. Low level is said to be the lowest level which includes approximation part belongs to the original image and other 3 levels give full data of the original image.

Techniques	**Merits**	**Demerits**
LSB substitution	Improves stego-image quality	Embedding capacity is low.
ISB technique	Improves the quality of watermarked images.	Not robust against attacks
Difference Expansion (DE)	No loss of data due to compression and decompression	Cannot be applied to textured Image because of more sensitive to the smoothness of the image

Interpolation Technique	1. Less distortion 2. High visual quality	Low Embedding capacity
Prediction Error Expansion (PEE)	Enables to embed large payload while keeping distortion low	Low efficiency and security of RDH
Histogram shifting	Higher simplicity	Low capacity which is restricted by peak-pixel value frequency of the histogram
Discrete Wavelet Transform (DWT)	Improves image quality	Low-performance time
PWLC technique	Improved data hiding capacity and image quality	Does not extract secret data and not recovers the cover image
DHTC technique	Have brilliant visual quality and does not has salt-and-pepper noise	To insert secret data DHTC flips only low- visibility pixels
Recursive Histogram Modification	Gives improvement that is more significant for larger images	The issue of communicating multipeak and zero points

Figure 4 Comparison table of data hiding techniques

The table above defines how every technique has its drawbacks and benefits. Each technique differs from each other some methods provide more security whereas other methods have high capacity in hiding, some methods provide robustness, imperceptibility, and visual quality. Each method has a different level of applications according to their merits and demerits. One method cannot be suitable for other applications.

Chapter II DE & PE Algorithms Theory

II.1 Introduction

Over the last two decades, several RDH methods have been proposed based on lossless data compression techniques. These schemes involve methods that apply lossless compression to a selected set of features from the original image and then embed the message in the space that has been saved due to feature compression. One of the earliest works, have proposed a data embedding framework having the property that the original image can be fully recovered once the data has been extracted from the marked image. They have presented methods for both the uncompressed formats (BMP, TIFF) and for the JPEG format. Another technique has presented a lossless generalized LSB embedding scheme (G-LSB) which is based on grouping the pixels of an image and embedding data bits into the state of each group. To increase the embedding capacity of RDH, a different scheme called Difference Expansion (DE) has been proposed. The scheme is based on modifying the difference between a pair of pixel values while keeping the average of them unchanged. DE achieves good performance since natural images, in general, exhibiting high correlation between adjacent pixels. In novel methods were proposed to resolve two issues associated with DE method, i.e., the maximum number of embeddable location and the payload control capability. (25)

II.2 Difference Expansion with Histogram Shifting and Overflow Map

The DE Embedding technique is a technique that takes a Gray-Level image (White and Black) and calculates a bunch of numbers represented as low-pass image *L* containing the integer averages and a high-pass image containing the pixel differences.

II.2.a Calculating High Pass and Low Pass Values

Let's assume that *a* and *b* are two pixels besides each other, then the low-pass and high-pass values are defined as:

$$l = floor(\frac{a+b}{2})$$

Equation 1 Low-Pass

$$h = a - b$$

Equation 2 High-Pass

These two values *l* and *h* can be used again to restore the original pixel values *a* and *b* as follows:

$$a = l + floor(\frac{h+1}{2})$$

Equation 3 Value of a from l and h

$$b = l - floor(\frac{h}{2})$$

Equation 4 Value of b from l and h

```
a=168,b=170

a = 168
b = 170

l=floor((a+b)/2),h=a-b

l = 169
h = -2

a=l+floor((h+1)/2),b=l-floor(h/2)

a = 168
b = 170
```

Figure 5 Restoring a and b from l and h Example

The restored a and b are the same.

II.2.b Expansion Embedding the Information bit

We can store an information bit *i* in the high-pass value *h* by shifting it to the left by one then adding the bit to the LSB of h as explained below:

$$hw = 2 * h + i$$

Equation 5 Expansion-Embedded Difference

The value h_w is then used to calculate the watermarked pixels *a* and *b* using II.2.a

Note: the values a and b are 8-bit represented so they don't exceed 255.

To restore the original difference all we need to do is this:

$$h = floor(\frac{hw}{2})$$

Equation 6 Restoring Expansion-Embedded Difference

II.2.c Invertible Region Rd

Since the values a and b are 8-bit represented, the watermarked difference h_w need to satisfy the condition below so the watermarked values a and b don't overflow:

$$|h| \in Rd(l) = [0, \min(2 * (255 - l), 2 * l + 1)]$$

Condition 1 Difference h

Where Rd is called the invertible region, combining Equation 5 and Condition 1 we get the condition for DE for a pair-pixel *a* and *b*:

$$|2h + i| \in Rd(l) \quad for \ i = 0,1$$

Condition 2 Expandability for DE

This is called the expandability condition for DE.

II.2.d LSB Replacement Embedding the Information bit

In the DE Algorithm there is another technique that's used which called LSB replacement. In the LSB replacement technique, the LSB of the difference is replaced with the information bit.

Someone asks a question, the true LSB is replaced? so it is overwritten and lost isn't? don't worry, the true LSBs are saved with the payload to make sure that we can restore the original LSBs so the original pixels.

We can replace the LSB of a difference only if the pair-pixels *a* and *b* satisfy the changeability condition:

$$|2 * floor\left(\frac{h}{2}\right) + i| \in Rd(l) \quad for\ i = 0,1$$

Condition 3 Changeability for DE

Take note that an expandable difference is also a changeable difference and a changeable difference remains changeable even after its LSB is replaced.

Example

```
a=175;b=180;
l=floor((a+b)/2);h=a-b;
%Embedding bit (1) to h
h=2*h+1
```

```
h = -9
```

```
%Checking if the watermarked difference is
%in Rd range
if(abs(h)>=0 && abs(h)<=min(2*(255-l),2*l+1))
    a=l+floor((h+1)/2);
    fprintf('Its Expandable !\nThe Watermarked pixels are:\na=%d',a)
    b = l-floor(h/2);
    fprintf('\nb=%d',b)
else
    disp('Not expandable')
end
```

```
Its Expandable !
The Watermarked pixels are
a=173
b=182
```

Figure 6 Expansion Embedding Example

II.2.e Difference Expansion's Domains

Let D be the common domain of the high-pass and low-pass images, H and L, respectively. Each element of is associated with a difference and an integer-average (or equivalently a pair of pixel intensities). Expandable locations and changeable locations are subsets of D. The subset of D with corresponding changeable differences is denoted by C and is called the set of changeable locations. An important subset of C containing the locations with expandable differences is denoted E by and is called

the set of expandable locations. Using a selection criterion depending on the size of the payload, E is partitioned into E’ and the set difference, E \ E’. The differences at E’ are expansion embedded. The differences at C \ E’ are modified by LSB replacement. To ensure reconstruction, the original LSBs are saved and embedded along with the payload. To enable reconstruction, a binary location map indicating the selected locations, E’, is created and losslessly compressed. A bitstream is formed by concatenating the compressed location map, the saved LSBs and the payload, and this bitstream is then embedded into the high-pass image H. The locations of H are traversed in a predefined order (raster-scan order or a secret-key order), and the bits are embedded into the changeable locations, C. The watermarked image is calculated from the modified high-pass image and the low-pass image.

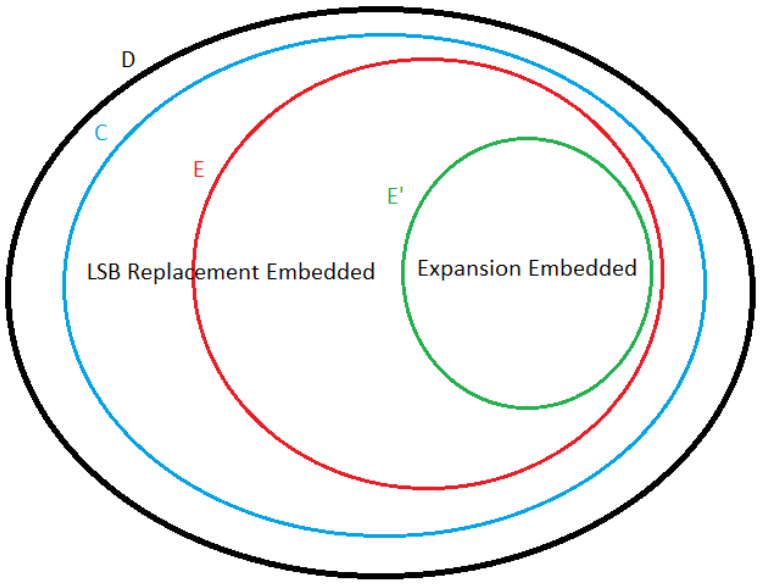

Figure 7 Difference Expansion Domains

II.2.f Differences’ Histogram

We take an example of this theory the Lena image:

Figure 8 Lena Image

We begin by calculating High-Pass and Low-Pass Values of the image using Equation 1 and Equation 2 in a raster-scan order for the pair-pixels, we obtain a 1-D array for both the High-Pass and Low-Pass Values.

From these two arrays we draw a histogram of expandable differences:

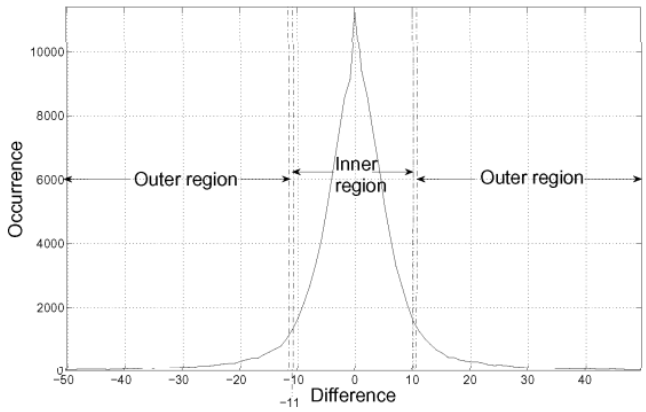

Figure 9 Expandable Differences Histogram

II.2.g Histogram Shifting

Consider the histogram of expandable differences of the Lena image, as shown in Figure 9. The difference histogram for most natural images would be like this, in the sense that difference values with small magnitudes occur more frequently. Consider a process of selecting locations for expansion embedding from the set of expandable locations that involves selecting suitable bins from the histogram of expandable differences. The bins whose differences have smaller magnitude are given preference in the selection process because the smaller the magnitude of the expandable difference, the smaller the resulting distortion. Therefore, the selection of locations for expansion embedding involves setting an appropriate threshold $\Delta \geq 0$, such that $\Delta + 1$ negative and $\Delta + 1$ nonnegative bins from the histogram are selected, resulting in $2\Delta + 2$ bins. These selected bins have differences in the range $[\Delta - 1, \Delta]$. This selection method divides the histogram into two nonoverlapping inner and outer regions as shown in Figure 9 for ($\Delta = 10$)

After we Expansion Embed the inner region as shown in Figure 9 using the methods in (II.2.bExpansion Embedding the Information bit), now the modified differences now occupy the range $[-2\Delta - 2, 2\Delta + 1]$:

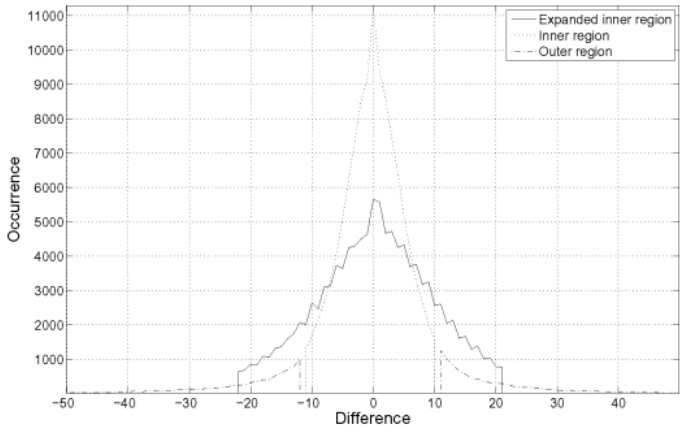

Figure 10 Histogram after expansion of inner region (selected bins from Figure 9)

Comparing this range with the range of the differences that constitute the outer regions, we see that they overlap in the range [-2Δ - 2, 2Δ + 1]. An appropriate histogram shift of the outer regions would cancel all overlap between the two regions. To achieve this, the negative differences and the nonnegative differences of the outer regions should be shifted left and right, respectively, by at least Δ + 1

$$hs = \begin{cases} h + \Delta + 1, & if\ h > \Delta \\ h - \Delta - 1, & if\ h < -\Delta - 1 \end{cases}$$

Equation 7 Histogram Shifting

From the above equation we get a histogram of:

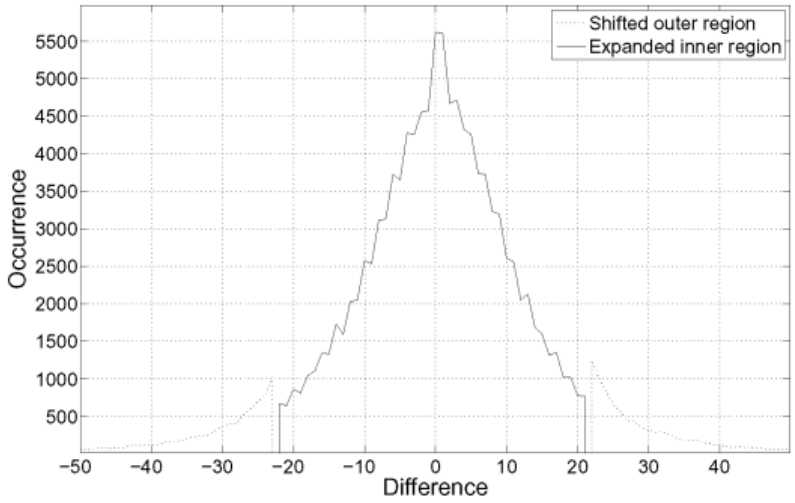

Figure 11 Shifted Histogram (not selected bins from Figure 10)

The histograms of the expanded inner region and the shifted outer regions are shown in Figure 11.

A histogram shift can be easily reversed if Δ is known

$$h = \begin{cases} hs - \Delta - 1, & if\ hs > 2\,\Delta + 1 \\ hs + \Delta + 1, & if\ hs < -2\Delta - 2 \end{cases}$$

Equation 8 Reverse Histogram Shift

Note that the discussion on histogram shifting has been restricted to expandable differences lying in the outer regions (i.e., differences outside the range [-Δ-1, Δ]). Histogram shifting causes a smaller change in these differences than difference expansion. Therefore, it is not necessary to check whether a histogram shift might cause overflow/underflow.

Incorporating histogram shifting along with DE also eliminates the need to have a location map of the selected expandable locations (they can be identified at the decoder from the histogram of the differences). Consequently, the amount of auxiliary information embedded is also significantly reduced. In addition, the computational intensity required for histogram shifting is much less than that required for the compression/ decompression engine.

II.2.h Notation and Functions

To simplify the explanation of the two approaches to extend Tian's algorithm in the subsequent sections, we define some notation and functions in this section. The operators, $\oplus$ and •, represent the DE and LSB-embedding operations, respectively. They operate on an integer *h* and a bit *b=0,1* and are defined as

$$h \oplus b = 2h + b$$

Equation 9 Expansion Notation

$$h \bullet b = 2 * floor\left(\frac{h}{2}\right) + b$$

Equation 10 LSB Embedding Notation

The concatenation operator, ⊙, operates on two bitstreams and concatenates them. The length of a bitstream is returned by the function *n ()*. Therefore, if *A* and *B* are two bitstreams, then

$$n(A \odot B) = n(A) + n(B)$$

Equation 11 Bitstream Length

II.2.i Embedding the Bitstream

We first decompose the image into differences and integer averages and determine the changeable and the expandable locations *E*. A 2-D overflow map, M, is formed, indicating the expandable locations. The overflow map is losslessly compressed. The compressed overflow map $\mathcal{M}$ and a header segment $\mathcal{Q}$ constitute the auxiliary information. The auxiliary information stream $\mathcal{A}$ is formed by concatenating the compressed bitstream $\mathcal{M}$ to the header segment $\mathcal{Q}$. The header segment has a fixed length, which is also known to the decoder. The total length of the auxiliary stream *n(A)* is

$$\eta(\mathcal{A}) = \eta(\mathcal{M}) + \eta(\mathcal{Q}).$$

Equation 12 Length of Auxiliary Information

The total number of bits embedded by expansion embedding is the sum of the size of the auxiliary bitstream $\eta(\mathcal{A})$ and the size of the payload *n(P)*. Let be the total number of bits embedded by expansion embedding

$$t = \eta(\mathcal{A}) \eta \eta(\mathcal{P}).$$

Equation 13 Total Size of the payload

The operating threshold Δs is selected such that the total number of locations that comprise the selected bins is at least as large as *t*. Based on Δs, we partition *E* into two sets E_e and E_s, and, where E_e is the set of expandable locations that comprise the selected bins and the remaining expandable locations comprise E_s. Histogram shifting of the bins that correspond to the set E_s is done next. The auxiliary information stream is then created, wherein the header segment is populated with information regarding the size of the compressed overflow map, *n(M)*, the operating threshold Δs and the size of the payload *n(P)*. The header segment *Q* is then appended to the compressed location map *M* to create the auxiliary information stream *A*

$$\mathcal{A} = \mathcal{Q} \odot \mathcal{M}.$$

Equation 14 Auxiliary Bitstream

The LSBs of the differences at the locations in C \ E_e are then saved into *L*. The auxiliary information stream *n(A)*, the saved LSBs *L*, and the payload *P* are then concatenated to create the bitstream *B* to be embedded

$$\mathcal{B} = \mathcal{A} \odot \mathcal{P} \odot \mathcal{L}.$$

Equation 15 Bitstream

The bitstream *B* is embedded at the locations in *C* —using DE at the locations in E_e and using LSB replacement at the locations in C \ E_e. From the resulting differences and integer averages, the watermarked image is computed by inverting the transform using Equation 3 and Equation 4.

Example

Let's say that we have an information to embed in Figure 8 Lena Image that consists of the name "Adel Benhamida", after we mark the expandable/changeable locations such as Figure 7, we start creating the header segment which is the size of the compressed overflow map and the threshold Δ and the size of the payload which is the size of the name 14 x 8, assuming that the size of compressed overflow map is 1000 and delta value is 0

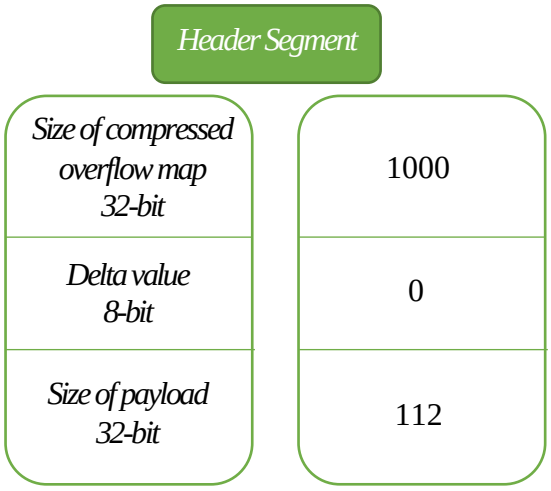

Figure 12 Header Segment

Now, we have created the header segment and it is ready to be concatenated with the other segments.

For the compressed overflow map, payload and the LSBs, they are just a collection of bits that are stick together.

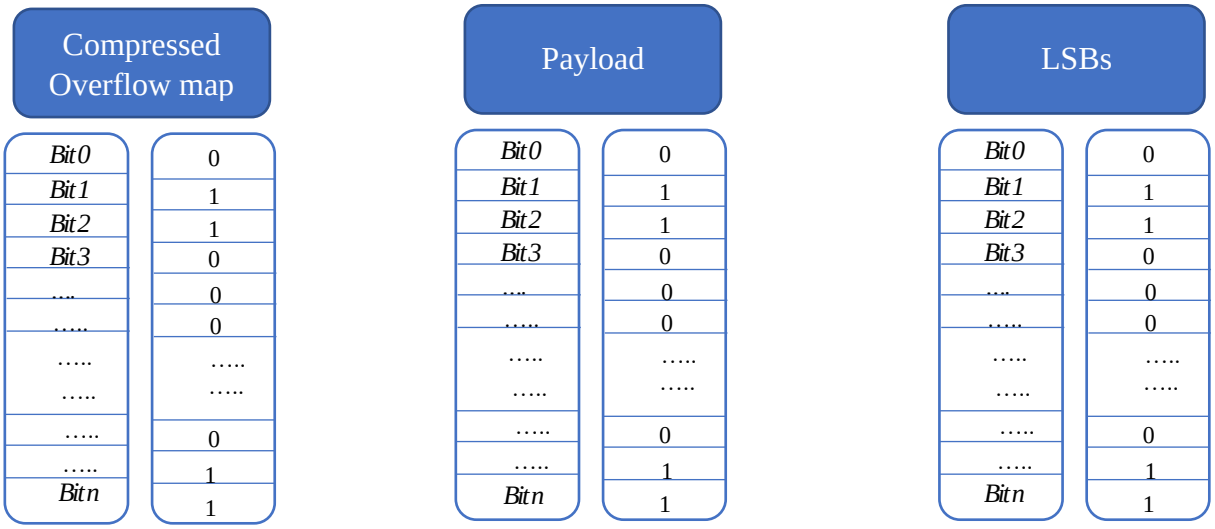

Figure 13 Compressed Overflow Map and Payload and LSBs

After we built these segments, we just stick them together in this order to end up with a bitstream that we then embed to the image using Equation 4 and Equation 3 and Equation 5:

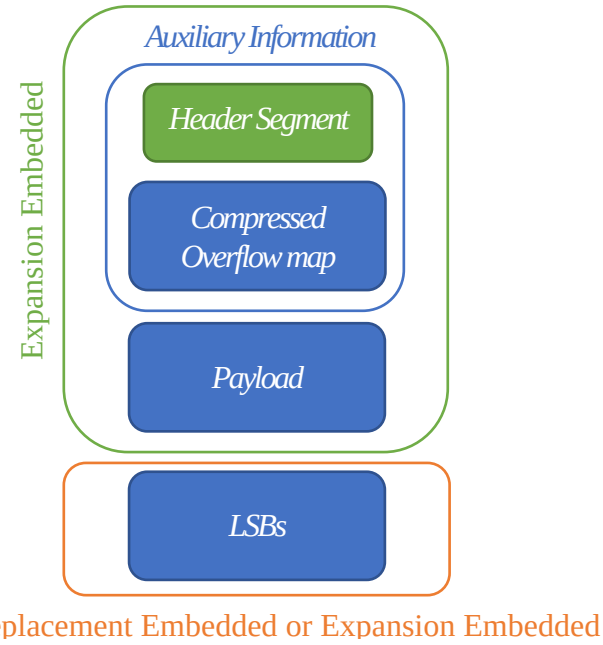

Figure 14 Bitstream Diagram

II.3 Decoding the bitstream from a watermarked image

II.3.a Extracting the bitstream

To extract the bitstream all we need to do is to calculate the high pass and low pass values (II.2.aCalculating High Pass and Low Pass Values) and then we check for the changeable differences and then we start reading their LSBs in a raster-scan order to build our bitstream.

II.3.b Restoring the Original Image

After we extract the bitstream, we decompress the overflow map to get the expandable locations, and using the threshold Δ we can derive the locations Ee and E_s, as we talked in (II.2.gHistogram Shifting) the modified differences Ee now occupy the range of $[-2\Delta - 2, 2\Delta + 1]$, so we use this range to identify E_e and E_s, the E_e locations are expansion restored (Equation 6 Restoring Expansion-Embedded Difference), and Es and C locations are LSB-Replacement restored by overwriting the differences' LSBs with the extracted LSBs from the bitstream, and we extract our payload.

II.4 Prediction-Error Expansion

II.4.a Pixel Context

The embedding process involves computing the prediction-error (PE) from the neighborhood of a pixel

c1	c2
c3	a

Figure 15 Pixel Context

Using this equation:

$$\tilde{a} = \begin{cases} \max(c2, c3), & \text{if } c1 \leq \min(c2, c3) \\ \min(c2, c3), & \text{if } c1 \geq \max(c1, c3) \\ c2 + c3 - c1, & \text{otherwise} \end{cases}$$

Equation 16 Predicted Value

II.4.b Embedding Information bit

Embedding the information bit in the expanded prediction error. The difference between the pixel intensity *and* its predicted intensity *à* is the prediction error

$$p = a - \grave{a}$$

Equation 17 Prediction Error

Embedding a bit *i* in *p* results in the watermarked value PE

$$p_w = p \oplus i = 2p + i.$$

Equation 18 Watermarked Prediction Error

II.4.c Invertible Region Rp

The invertibility region of PE that we need to check the watermarked error value for is described as follows:

$$R_p(\hat{a}) = [-\hat{a}, 2^n - 1 - \hat{a}]$$

Equation 19 Invertibility Region for PE

From this interval we can derive Expandability / Changeability conditions which are:

$$2 * p + i \in Rp(\grave{a}), \qquad i = 0,1$$

Condition 4 Expandability for PE

$$2 * floor\left(\frac{p}{2}\right) + i \in Rp(\grave{a}), \quad i = 0,1$$

Condition 5 Changeability for PE

II.4.d Embedding the bitstream

In the PE Algorithm is the same as DE Algorithm, we build the bitstream that consists of the Header, compressed overflow map, payload and the LSBs and we start embedding it to the predicted error using (Equation 18 Watermarked Prediction Error) and from it we derive the watermarked pixel using (Equation 17 Prediction Error), we do the whole image in a raster-scan order.

Since the predicted value relies on the 3 neighbor pixels and these neighbor pixels are also modified during the embedding process it would be impossible at the decoder to identify the changeable locations since the pixel's context has been changed, we came to an idea of making sure that all the locations of the image are changeable by restricting the predicted value at any location to be an even value.

$$\grave{a} = 2 * floor(\frac{\grave{a}}{2})$$

Equation 20 Even Predicted Value

II.5 Decoding the Bitstream from a Watermarked Image

II.5.a Extracting the bitstream

Since the prediction algorithm ensures that the predicted value for any location is an even number, the PE at any location is a changeable PE, so extracting the embedded bitstream is trivial because the embedded bits are the LSBs of the pixel values.

II.5.b Restoring the Original Image

The restoration process is sequential because the pixel context has also been changed, we start by decompressing the overflow map to get the expandable locations and then we start by the first pixel which has no context ($\grave{a}$ =0), we check if it is in the interval of [-2Δ - 2, 2Δ + 1] so restore the pixel using (Equation 6 Restoring Expansion-Embedded Difference) if it is out of that range we restore it using LSB-Replacement, after we restore the first pixel we have to commit it to the image so the context of the next pixel will be ready and correct to be read.

Chapter III Implementing the Algorithms in C++

III.1 Introduction

III.1.a What is an IDE?

An integrated development environment (IDE) is software for building applications that combines common developer tools into a single graphical user interface (GUI). An IDE typically consists of:

- **Source code editor**: A text editor that can assist in writing software code with features such as syntax highlighting with visual cues, providing language specific auto-completion, and checking for bugs as code is being written.
- **Local build automation**: Utilities that automate simple, repeatable tasks as part of creating a local build of the software for use by the developer, like compiling computer source code into binary code, packaging binary code, and running automated tests.
- **Debugger**: A program for testing other programs that can graphically display the location of a bug in the original code.

III.1.b Why developers use IDEs

An IDE allows developers to start programming new applications quickly because multiple utilities don't need to be manually configured and integrated as part of the setup process. Developers also don't need to spend hours individually learning how to use different tools when every utility is represented in the same workbench. This can be especially useful for onboarding new developers who can rely on an IDE to get up to speed on a team's standard tools and workflows. In fact, most features of IDEs are meant to save time, like intelligent code completion and automated code generation, which removes the need to type out full character sequences.

Other common IDE features are meant to help developers organize their workflow and solve problems. IDEs parse code as it is written, so bugs caused by human error are identified in real-time. Because utilities are represented by a single GUI, developers can execute actions without switching between applications. Syntax highlighting is also common in most IDEs, which uses visual cues to distinguish grammar in the text editor. Some IDEs additionally include class and object browsers, as well as class hierarchy diagrams for certain languages.

It is possible to develop applications without an IDE, or for each developer to essentially build their own IDE by manually integrating various utilities with a lightweight text editor like Vim or Emacs. For some developers the benefit of this approach is the ultra-customization and control it offers. In an enterprise context, though, the time saved, environment standardization, and automation features of modern IDEs usually outweigh other considerations.

Today, most enterprise development teams opt for a pre-configured IDE that is best suited to their specific use case, so the question is not whether to adopt an IDE, but rather which IDE to select.

III.1.c Visual Studio

Visual Studio is an Integrated Development Environment (IDE) developed by Microsoft to develop GUI (Graphical User Interface), console, Web applications, web apps, mobile apps, cloud, and web services, etc. With the help of this IDE, you can create managed code as well as native code. It uses the various platforms of Microsoft software development software like Windows store, Microsoft Silverlight, and Windows API, etc. It is not a language-specific IDE as you can use this to write code in C#, C++, VB (Visual Basic), Python, JavaScript, and many more languages. It provides support for 36 different programming languages. It is available for Windows as well as for macOS.

III.2 Using Visual Studio

III.2.a Create a New Project

This is the entry window of the program:

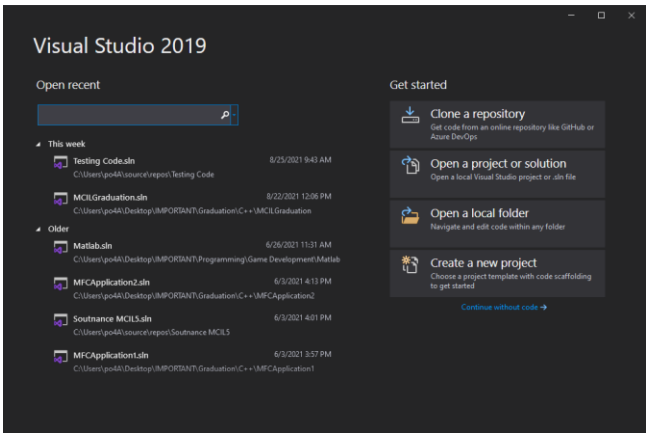

Figure 16 Entry-Window VS2019

We click at **Create a new project** and then we fill the information about our project (project name, location, select **Windows Desktop Application** as a project template) and we end up with:

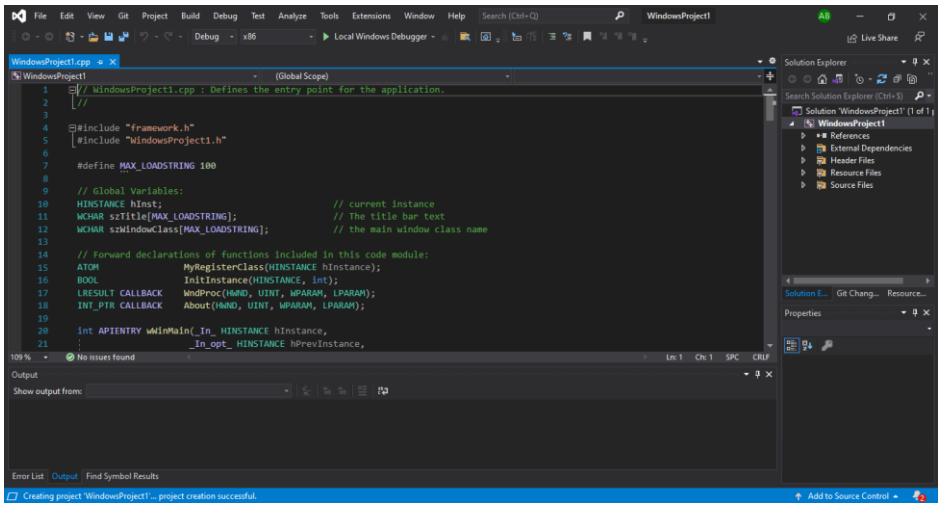

Figure 17 Project Window

III.2.b GUI Design Window

In top-right window we can find **Solution Explorer**, we use it to browse the project's files such as header and source files, it also includes the GUI design files used to describe how the user interface will look like:

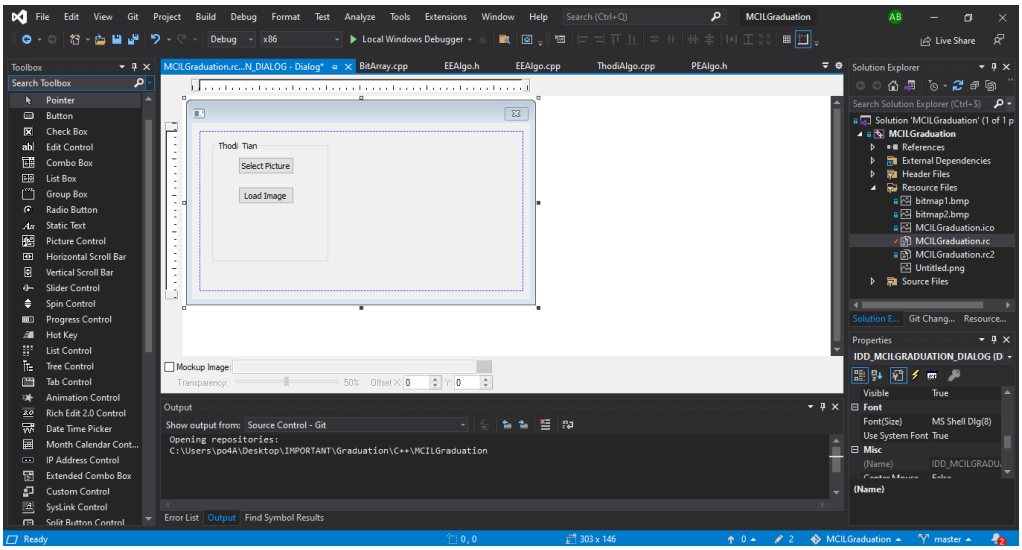

Figure 18 GUI Design Window

III.2.c Add New Source (Class) File

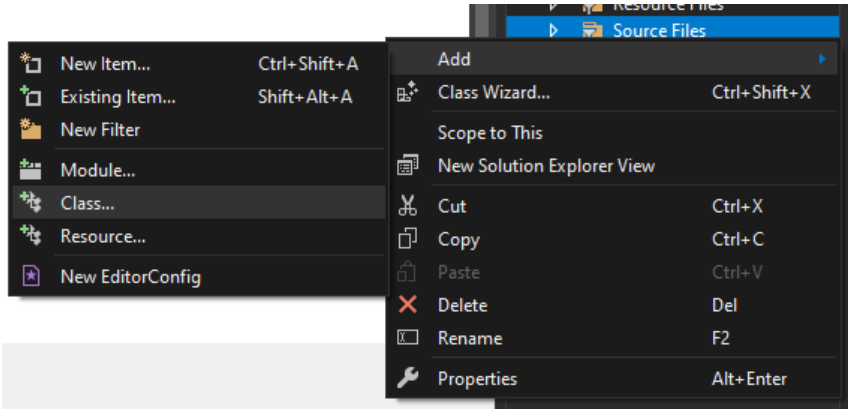

Figure 19 Adding a class

After we click at **Class,** a new window will appear asking for the class name, automatically creates the Header (.h) and the Class (.cpp) File and puts them in right folders (.h file in **Header Files** folder and .cpp in **Source Files** folder).

III.2.d Debugging Code

Debugging feature is a must-have feature in an IDE, it allows you to inspect the execution of code at run time and allows you to inspect values of the variables at run, you can set breakpoints at any statement, you can see this here:

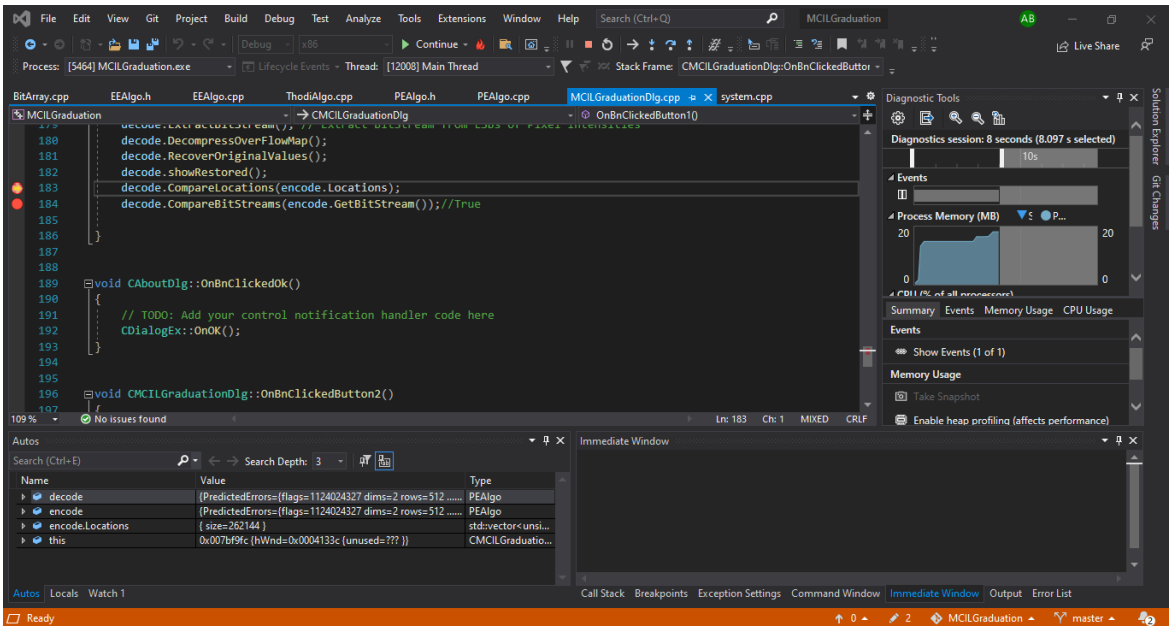

Figure 20 Debugging Feature

You can see the current being-executed statement and the values of the variables at that time, you can also check the memory usage of the process.

III.3 Implementing the Algorithm

III.3.a BitArray Class

Since we are dealing with bits everywhere, we can't access them directly in C++ because the smallest variable type in computer programming languages is char which is 8-bits, so we need some sort of code that manipulates Bits easily, for that we'll create a class for it, let's name it **BitArray.**

With this class we'll be able to:

- Read bit at any location
- Write bit to any location
- Compare Two BitArray Objects using the = = operator
- Get an Iterator for loop-based operations

Here is the class declaration BitArray.h (Header File):

```
class BitArray
{
public:
    unsigned int Size;
    unsigned int currIndex=0;
    unsigned int nextIndex=0;
    char* bitArray;
public:
    BitArray(int numArray);
    BitArray(char* input,unsigned int size=0);
    void set(unsigned int pos);

    void reset(unsigned int pos);
    void push(char Bit);
    char next();
    void resetNext() { nextIndex = 0; }
    void resetGet() { currIndex = 0; }
    char operator[](unsigned int i);
    bool operator== (BitArray& obj);
    void* Data() { return bitArray; }
    unsigned int size() { return Size; }
    unsigned int sizeInBytes() {
        return Size % 8 != 0 ? Size / 8 + 1 : Size/8;
    }
};
```

Figure 21 BitArray Header Class

As you can see in the picture above, we declared:

- 3 numbers, one for the size of the array (in bits), and the others are for iterator function.
- Two constructors, the first one takes a number (number of bits) and uses it to create an empty bit-array and the second one takes a pointer to another buffer and its size so you can manipulate its bits.
- A pointer for buffer that will hold the array of bits.
- Two functions (**set, reset**) to access the bits.
- next () function for the iterator accessing.
- The operators [ ] and = = will be explained later in this chapter.

Now we come to explaining the functions:

```
BitArray::BitArray(int numArray)
{
    Size = numArray;
    bitArray = (char*)malloc(sizeInBytes());
}
```

Figure 22 First Constructor Function

This function take number and uses it to create a buffer in the heap using malloc() function, it returns a pointer to the new created buffer, we save this pointer in the **bitArray** variable.

```
BitArray::BitArray(char* input, unsigned int size)
{
    bitArray = input;
    Size = size;
}
```

Figure 23 Second Constructor

Stores the input buffer and its size and now we can use the class to access bits in that array

```
void BitArray::set(unsigned int pos)
{
    *(char*)(bitArray + pos / 8) |= (0x80 >> pos % 8);
}

void BitArray::reset(unsigned int pos)
{
    *(char*)(bitArray + pos / 8) &= ~(0x80 >> pos % 8);
}
```

Figure 24 Reading Writing bits Functions

In these two functions we use the number **pos** to indicate the character position, since we are dealing with a char array (8-bit per item) we need to convert the **pos variable** from indexing a bit into indexing a character as explained in this picture:

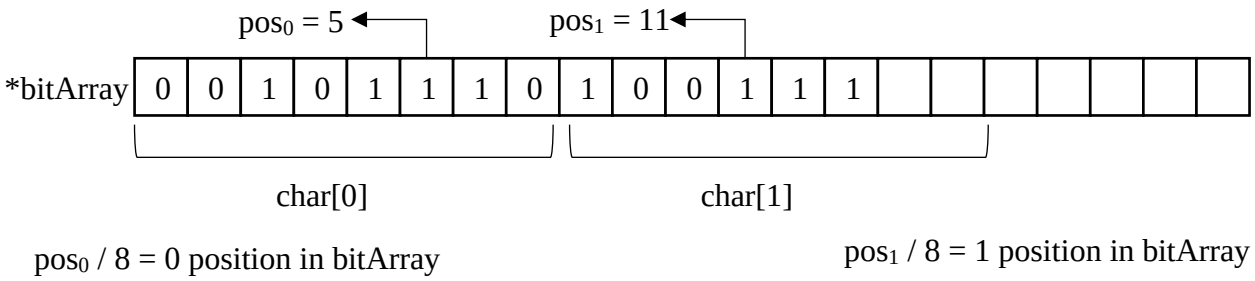

Figure 25 Reading Writing bits Diagram 1

Since the variable **pos** is declared **unsigned int so** any division operations on it will result in another int value (decimal digits after point gets ignored) unless it is being divided by a float / double value this will result in a float / double result.

After we got the position of the bit in which char it is located, now to get the position of the bit in that char, it is just as easy as this statement **pos % 8** which means the rest of **pos** divided by 8, the result will be used to shift the value **0x80** to the right then it will be used to perform an **OR** or **AND** operation with the char it is located in.

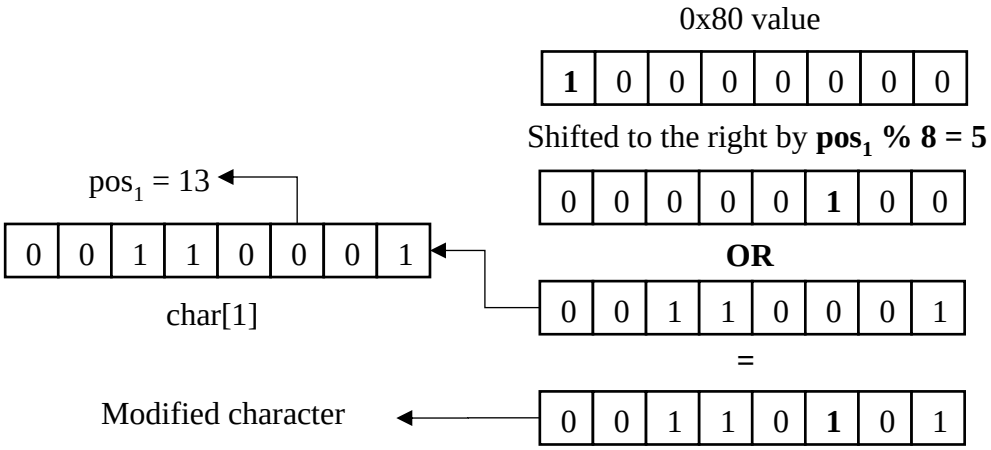

Figure 26 Reading Writing bits Diagram 2

The figure above explains how to set a bit to 1 in a char array, to set a bit to 0, all we need to do is after we shift the value 0x80 to right we perform a **NOT** operation on it and then we **AND** it with the desired character from the char array.

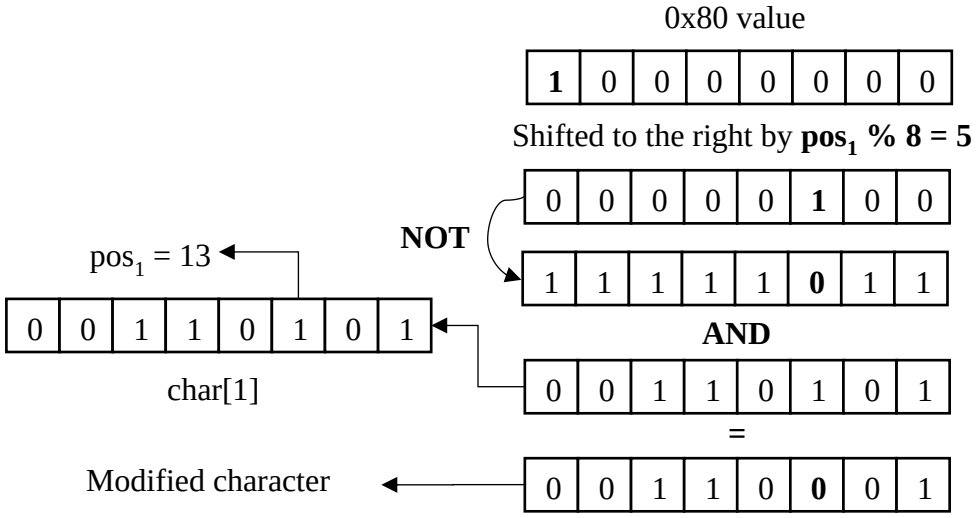

Figure 27 Reading Writing bits Diagram 3

```
void BitArray::push(char Bit)
{
    if (Bit & 0x01)
        set(currIndex++);
    else
        reset(currIndex++);
}
```

Figure 28 Push Function

It receives a char and then we AND it with 0x01 to extract the LSB which is the bit that we want to push to bit-array, if the result equals 1 which means **true** in C++ then we set the bit number **currIndex** to 1 else to 0

Note: currIndex starts from 0, and every time we want to overwrite the array, we need to reset this variable

```
char BitArray::operator[](unsigned int i)
{
    // TODO: insert return statement here
    return (*(unsigned char*)(bitArray + i / 8) >> (7 - i % 8))&0x01;
}
```

Figure 29 [ ] Operator Function

We use this function to read an array's bit by its position, first we read the byte where the bit is located in ***(char*)(bitArray + i/8)** then we shift the byte to the right by the current bit position, so we make the desired bit at the LSB of the byte and then we AND the byte with 0x01, so we return only the bit which is 0 or 1.

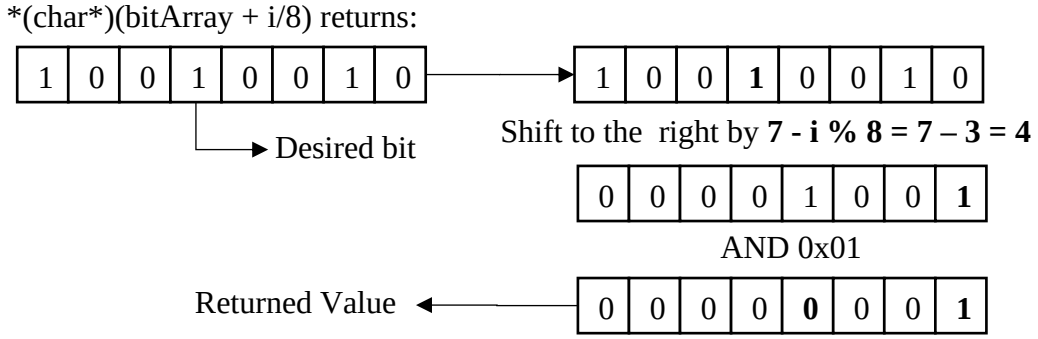

Figure 30 [ ] Operator Diagram

```
char BitArray::next()
{
    return (*this)[nextIndex++];
}
```

Figure 31 Next Function

We use this function to read bits starting for the index variable **nextIndex=0,** and this function uses (Figure 29 [ ] Operator Function)

```
bool BitArray::operator==(BitArray& obj)
{
    for (int i = 0; i < Size; i++)
        if ((*this)[i] != obj[i])
            return false;
    return true;
}
```

Figure 32 = = Operator Function

This is an overloaded operator (overloading is a C++ concept, please refer to https://en.cppreference.com/w/cpp/language/operators for more information), we use it to compare two BitArray objects, the parameter **obj** is the to-be-compared-with object, and **this object** is what we will compare to **obj**, it is a for loop that starts from 0 to the **Size** of the bitarray comparing bit by bit and if it finds two unequal bits then it returns false otherwise true.

III.3.b EEAlgo Class

This class will hold common functionalities between the next PEAlgo and DEAlgo Classes to avoid boilerplate code.

For loading the images' pixels we'll be using a library called **OpenCV,** I installed it using **vcpkg** (Please refer to https://github.com/microsoft/vcpkg for more information).

The common things between the PE and DE are:

- Bitstream (Header, Overflow Map…)
- Expand and Change bit Functions (Expansion and LSB Replacement Techniques)
- Input and Output Image

This is the Header File of the class:

```
class EEAlgo
{
protected:
    Mat _imagePixels;
    Mat _OriginalPixels;
    BitArray* OverFlowMapM;
    BitArray* Payload;
    BitArray* ComMap;
    BitArray* LSBs;
    uchar delta;
    size_t sizeOfLSBs;
    enum { NEITHER, EXPANDABLE, CHANGABLE, EXPANDABLE_IN_DELTA };
    short ExpandBit(short high, uchar Bit) { return high << 1 | (short)(Bit & 0x01); }
    short ChangeBit(short high, uchar Bit) { return (high >> 1) << 1 | (short)(Bit & 0x01); }
    bool isExpandable(short high, uchar low, bool (*isInRange)(short, uchar)) { return isInRange(high << 1 | (short)0x0001, low); }
    bool isChangable(short high, uchar low, bool (*isInRange)(short, uchar)) {
        return isInRange((short)(2 * floor((float)high / 2)) | (short)0x0001, low); }
    BitArray* GeneratePayload();
    uchar getCurrentRegion(unsigned int bitsEmbedded);
    enum { RANGE_HEADER, RANGE_COMPRESSED_OV_MAP, RANGE_PAYLOAD, RANGE_LSBS , RANGE_HEADER_EMPTY_BYTES};
    struct Header { ... };
    struct AuxilaryInformation { ... };
    struct BitStream { ... };
    BitStream BS;
public:
    BitStream GetBitStream() { return BS; }
    void showOriginal() { imshow("Original", _OriginalPixels); waitKey(1); }
    void showInjected() { waitKey(1); imshow("Injected", _imagePixels); waitKey(1); }
    void showRestored() { imshow("Restored", _imagePixels); waitKey(1); }
    Mat getPixels() { return _imagePixels; }
    Mat getOriginalPixels() { return _OriginalPixels; }
    bool isEqualTo(Mat imagePixels);
    bool CompareBitStreams(BitStream inBS);
    EEAlgo(const cv::String& filename, int flags = cv::IMREAD_GRAYSCALE);
    EEAlgo(Mat pixels);
```

Figure 33 EEAlgo Header

The ExpandBit Function takes two parameters, first one is the variable that we want to embed the bit to and the second one is the to-be-embedded bit using (Equation 5 Expansion-Embedded Difference)

The ChangeBit Function is just as same as the ExpandBit function it just implements (Equation 10 LSB Embedding Notation)

isExpandable / isChangeable Function checks if the value is expandable or not and takes a pointer to a function that checks the value if it's in the range (Invertible Region Rd, Invertible Region Rp)

GeneratePayload Function generate a random bit-array and returns a pointer to it to embed it later with the bitstream.

```
uchar EEAlgo::getCurrentRegion(unsigned int bitsEmbedded)
{

    if (bitsEmbedded >= sizeof(int)*8+8 && bitsEmbedded < 72)
        return RANGE_HEADER_EMPTY_BYTES;
    if (bitsEmbedded >= 72 && bitsEmbedded < 72 + BS.aInfo.header.SizeOfCompressedOverFlowMap)
        return RANGE_COMPRESSED_OV_MAP;
    if (bitsEmbedded >= 72 + BS.aInfo.header.SizeOfCompressedOverFlowMap &&
        bitsEmbedded < 72 + BS.aInfo.header.SizeOfCompressedOverFlowMap + BS.aInfo.header.SizeOfPayload)
        return RANGE_PAYLOAD;
    if (bitsEmbedded >= 72 + BS.aInfo.header.SizeOfCompressedOverFlowMap + BS.aInfo.header.SizeOfPayload)
        return RANGE_LSBS;
    return RANGE_HEADER;
}
```

Figure 34 getCurrentRegion Function

getCurrentRegion Function returns what region the current being-embedded bit is located at.

This is the declaration of the bitstream struct:

```
struct Header {
    unsigned int SizeOfCompressedOverFlowMap;
    uchar Delta;
    unsigned int SizeOfPayload;
};
struct AuxilaryInformation
{
    struct Header header;
    void* overflowComp;
};
struct BitStream
{
    struct AuxilaryInformation aInfo;
    void* payload;
    void* LSBs;
};
```

Figure 35 BitStream Declaration

Since we know that the size of header segment is 4 + 1 + 4 bytes but in this **struct** above the compiler allocates 4 + 4 + 4 bytes for it, for performance concerns it must be aligned to an address that is multiple to 4.

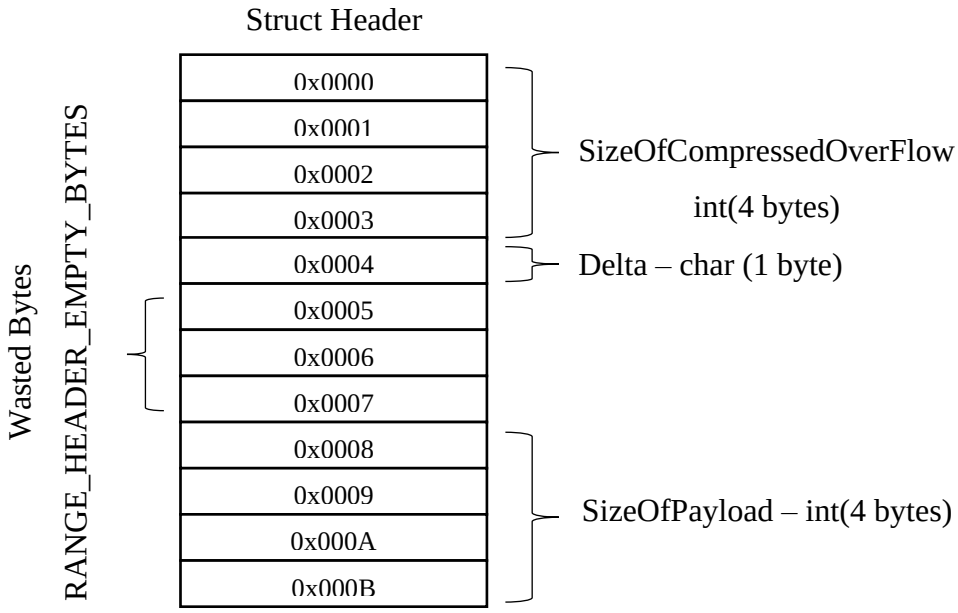

Figure 36 Header Struct Diagram

But if we declared another char variable after **delta** variable, the compiler would make use of the Wasted Bytes until we declare a variable that's larger than the wasted bytes.

III.3.c DEAlgo Class

In the DEAlgo Header Class we have these public functions:

```
#include "EEAlgo.h"
using namespace cv;
class DEAlgo : public EEAlgo
{
private:
    std::vector<uchar> Low;
    std::vector<uchar> Locations;
    unsigned __int32 imageSize;
    // a function that checks if value is in RD Range
    static bool isInRdRange(short val, uchar low);
public:
    std::vector<short> High;
    DEAlgo(const cv::String& filename, int flags = cv::IMREAD_GRAYSCALE);
    DEAlgo(Mat pixels);
    void Init(Mat pixels);
    void CalcHighPass();
    void DetermineLocations();
    void GetDelta();
    void OutterHistogramShift();
    void BuildBitStream();
    void EmbedBitStream();
    void CompressOverFlowMap();
    void CompileImage();
    // Decoding Functions
    void GetCLocations();
    void ExtractBitStream();
    void DecompressOverFlowMap();
    void IdentifyExpandedLocations();
    void RestoreLSBs();
    void ReverseShift();
    void RestoreExpanded();
    bool CompareHigh(std::vector<short> tHigh);
};
```

Figure 37 DEAlgo Header Class

As you can see this class extends the parent class EEAlgo, so we'll be able to use EEAlgo's functionalities on DEAlgo data.

In this class we have two constructors, one takes a full file path name of the image and how it is loaded, here we loaded it as a Grayscale image, and the other constructor take an already-loaded image or matrix to apply the algorithm on it.

The **CalcHighPass** Function is a simple loop that iterate all over the image's pixels and calculate the high pass and low pass value using (II.2.aCalculating High Pass and Low Pass Values)

```
void DEAlgo::DetermineLocations()
{
    sizeOfLSBs = 0;
    for (int i = 0; i < Locations.size(); i++)
    {
        // Checking if Hw is in Rd range
        if (isExpandable(High.at(i), Low.at(i), &isInRdRange))
        {
            Locations.at(i) = EXPANDABLE;
            OverFlowMapM->set(i);
            sizeOfLSBs++;
            continue;
        }
        else if (isChangable(High.at(i), Low.at(i), &isInRdRange))
        {
            Locations.at(i) = CHANGABLE;
            sizeOfLSBs++;
        }
        else
            Locations.at(i) = NEITHER;
        OverFlowMapM->reset(i);
    }
}
```

Figure 38 DetermineLocations Function

in **DetermineLocations** function, we are iterating over the high and low pass values and check every pair if they are expandable or changeable and mark them in another vector called Locations and mark the Expandable Locations which are the overflow map, the **sizeOfLSBs** variable will be used later to indicate the size of LSBs segment.

```
void DEAlgo::GetDelta()
{
    unsigned int bits=0;
    for (delta = 0; delta < 256; delta++)
    {
        for (int i = 0; i < High.size(); i++)
        {
            if (!(High.at(i) >= -(int)delta-1 && High.at(i) <= delta))
                continue;
            if (Locations.at(i) != EXPANDABLE)
                continue;
            Locations.at(i) = EXPANDABLE_IN_DELTA;
            sizeOfLSBs--;
            bits++;
        }
        if (bits >= 72 + BS.aInfo.header.SizeOfPayload + BS.aInfo.header.SizeOfCompressedOverFlowMap)
            break;
    }
}
```

Figure 39 GetDelta Function

GetDelta Function iterates over the high pass values and checks the number of the differences that are in the range of [-Δ-1, Δ], if that number is enough for the Auxiliary Information and the payload to be embedded in then we break the loop and save the threshold Δ (**delta** variable).

OutterHistogramShift Function iterates over the high pass values that are expandable and outside of the range [-Δ-1, Δ] then it shifts them to right and the left according to their sign, see (Equation 7 Histogram Shifting).

```
void PEAlgo::BuildBitStream()
{
    BS.aInfo.header.Delta = delta;
    BS.aInfo.header.SizeOfCompressedOverFlowMap = ComMap->size();
    BS.aInfo.header.SizeOfPayload = Payload->size();
    BS.aInfo.overflowComp = ComMap->Data();
    BS.payload = Payload->Data();
    LSBs = new BitArray(sizeOfLSBs);
    //Save LSBs of C \ Ee
    for (int i = 0; i < _imagePixels.rows; i++)
        for(int j =0;j<_imagePixels.cols;j++)
                if (Locations.at(i* _imagePixels.rows+j) == CHANGABLE || Locations.at(i * _imagePixels.rows + j) == EXPANDABLE)
                    LSBs->push(PredictedErrors.at<short>(i,j));

    BS.LSBs = LSBs->Data();
}
```

Figure 40 BuildBitStream Function

BuildBitStream Function fills the struct **BitStream** with the information and saves the LSBs of C \ E_e, see Figure 7 Difference Expansion Domains.

```
void DEAlgo::EmbedBitStream()
{
    BitArray headerBits((char*)&BS.aInfo.header,0),PayloadBits((char*)BS.payload,0),LSBsBits((char*)BS.LSBs,0);
    unsigned int bitsEmbedded=0;
    uchar location;
    for (int i = 0; i < Locations.size(); i++)
    {
        location = Locations.at(i);
        switch (getCurrentRegion(bitsEmbedded))
        {
        case RANGE_HEADER:
            if (location == EXPANDABLE_IN_DELTA)
                High.at(i) = ExpandBit(High.at(i), headerBits[bitsEmbedded++]);
            else if (location == EXPANDABLE || location == CHANGABLE)
                High.at(i) = ChangeBit(High.at(i), headerBits[bitsEmbedded++]);
            break;
        case RANGE_HEADER_EMPTY_BYTES:
            if (location == EXPANDABLE_IN_DELTA)
                High.at(i) = ExpandBit(High.at(i), headerBits[bitsEmbedded++ + 3 * 8]);
            else if (location == EXPANDABLE || location == CHANGABLE)
                High.at(i) = ChangeBit(High.at(i), headerBits[bitsEmbedded++ + 3 * 8]);
            break;
        case RANGE_COMPRESSED_OV_MAP:
            if (location == EXPANDABLE_IN_DELTA)
                High.at(i) = ExpandBit(High.at(i), (*ComMap)[bitsEmbedded++ - 72]);
            else if (location == EXPANDABLE || location == CHANGABLE)
                High.at(i) = ChangeBit(High.at(i), (*ComMap)[bitsEmbedded++ - 72]);
            break;
        case RANGE_PAYLOAD:
            if (location == EXPANDABLE_IN_DELTA)
                High.at(i) = ExpandBit(High.at(i), PayloadBits[bitsEmbedded++ - 72 - (int)BS.aInfo.header.SizeOfCompressedOverFlowMap]);
            else if (location == EXPANDABLE || location == CHANGABLE)
                High.at(i) = ChangeBit(High.at(i), PayloadBits[bitsEmbedded++ - 72 - (int)BS.aInfo.header.SizeOfCompressedOverFlowMap]);
            break;
        case RANGE_LSBS:
            if (location == EXPANDABLE_IN_DELTA)
                High.at(i) = ExpandBit(High.at(i), LSBsBits[bitsEmbedded++ - 72 - (int)BS.aInfo.header.SizeOfCompressedOverFlowMap - (int)BS.aInfo.header.SizeOfPayload]);
            else if (location == EXPANDABLE || location == CHANGABLE)
                High.at(i) = ChangeBit(High.at(i), LSBsBits[bitsEmbedded++ - 72 - (int)BS.aInfo.header.SizeOfCompressedOverFlowMap - (int)BS.aInfo.header.SizeOfPayload]);
            break;
        }
    }
}
```

Figure 41 EmbedBitStream Function

In this function, we are reading the **header struct (BS.aInfo.header)** as a stream of bits using our BitArray Class and the payload and the LSBs, we use the variable **bitsEmbedded** to indicate which region we currently are in and then we check if the location is Expandable_in_delta then we embed the

bit using Equation 5 Expansion-Embedded Difference and if it is Changeable or Expandable we embed the bit using Equation 10 LSB Embedding Notation

CompileImage Function calculates the new watermarked pixels using Equation 3 Value of a from l and h and Equation 4 Value of b from l and h

```
void DEAlgo::CompressOverFlowMap()
{
    std::string* output = new std::string;
    snappy::Compress((char*)OverFlowMapM->Data(), OverFlowMapM->sizeInBytes(), output);
    ComMap = new BitArray((char*)output->data(),output->size()*8);
}
```

Figure 42 CompressOverFlowMap Function

The compressing algorithm we used is Google's snappy compressing engine please refer to https://github.com/google/snappy for information

For now, we mentioned the encoding functions and explained them, now we go the decoding part

We call the **CalcHighPass** function to calculate the high and low pass values.

GetCLocations Function marks every high-low pass pair-values that is changeable.

```
void DEAlgo::ExtractBitStream()
{
    BitArray BitStreamBuilder((char*)&BS,0);

    //Fill the empty Space between char and int of the header variables which is 3*8

    for (size_t i = sizeof(int)*8+sizeof(char)*8; i < 3 * 8+ sizeof(int) * 8 + sizeof(char) * 8; i++)
        BitStreamBuilder.reset(i);

    //Extract Header
    int i = 0;
    for (i = 0; i < Locations.size(); i++)
    {
        if (Locations.at(i) == EXPANDABLE || Locations.at(i) == CHANGABLE)
        {
            if (BitStreamBuilder.currIndex >= sizeof(int) * 8 + sizeof(char) * 8 && BitStreamBuilder.currIndex < 64)
                BitStreamBuilder.currIndex += 3 * 8;
            if (BitStreamBuilder.currIndex >= sizeof(Header) * 8)
                break;
            BitStreamBuilder.push(High.at(i));
        }
    }
    BitArray* ComMap = new BitArray(BS.aInfo.header.SizeOfCompressedOverFlowMap);
    BS.aInfo.overflowComp = ComMap->bitArray;

    //Extract Compressed Map
    for (; i < Locations.size(); i++)
    {
        if (Locations.at(i) == EXPANDABLE || Locations.at(i) == CHANGABLE)
        {
            if (ComMap->currIndex >= BS.aInfo.header.SizeOfCompressedOverFlowMap)
                break;
            ComMap->push(High.at(i));
        }
    }
    // Extract Payload
    BitArray *payload = new BitArray(BS.aInfo.header.SizeOfPayload);
    BS.payload = payload->bitArray;
    for (; i < Locations.size(); i++)
    {
        if (Locations.at(i) == EXPANDABLE || Locations.at(i) == CHANGABLE)
        {
            if (payload->currIndex >= BS.aInfo.header.SizeOfPayload)
                break;
```

Figure 43 ExtractBitStream Function

In this function, we read the bitstream variable **BS** as a stream of bits.

According to Figure 36 Header Struct Diagram we need to fill the wasted bytes RANGE_HEADER_EMPTY_BYTES with zeros and then start extracting the LSBs of high pass values to build our BitStream.

DecompressOverFlowMap Function decompresses the extracted overflow map using Google's snappy library

```
void DEAlgo::IdentifyExpandedLocations()
{
    // Mark E locations
    for (int i = 0; i < Locations.size(); i++)
        if ((*OverFlowMapM)[i])
            Locations.at(i) = EXPANDABLE;

    // Get Ee Locations
    for (int i = 0; i < Locations.size(); i++)
        if (Locations.at(i) == EXPANDABLE)
            if (High.at(i) >= -2 * (short)BS.aInfo.header.Delta - 2 && High.at(i) <= 2 * BS.aInfo.header.Delta + 1)
            {
                Locations.at(i) = EXPANDABLE_IN_DELTA;
                sizeOfLSBs--;
            }

}
```

Figure 44 IdentifyExpandedLocations Function

IndentifyExpandedLocations Function uses the decompressed overflow map to mark the expanded locations E and derive the E_e from it by marking the values that in the range of $[-2\Delta - 2, 2\Delta + 1]$. (according to II.2.gHistogram Shifting)

```
void DEAlgo::RestoreLSBs()
{
    BitArray LSBs = BitArray((char*)BS.LSBs, 0);
    LSBs.resetNext();
    for (int i = 0; i < Locations.size(); i++)
        if (Locations.at(i) == CHANGABLE || Locations.at(i) == EXPANDABLE)
            if (LSBs.next())
                High.at(i) |= 0x0001;
            else
                High.at(i) &= ~0x0001;
}
```

Figure 45 Restore the LSBs Function

We read the LSBs from the bitstream we extracted and start recovering the differences that are LSB-Replacement embedded $C \setminus E_e$.

RestoreExpanded Function restores the E_e Locations using Equation 6 Restoring Expansion-Embedded Difference.

```
void DEAlgo::ReverseShift()
{
    for (int i = 0; i < Locations.size(); i++)
        if (Locations.at(i) == EXPANDABLE)
            if (High.at(i) > 2 * (short)BS.aInfo.header.Delta + 1)
                High.at(i) += -(short)BS.aInfo.header.Delta - 1;
            else if (High.at(i) < -2 * (short)BS.aInfo.header.Delta - 2)
                High.at(i) += BS.aInfo.header.Delta + 1;
}
```

Figure 46 ReverseShift Function

This function reverse shifts the expandable differences that are outside of [-2Δ - 2, 2Δ + 1], according to II.2.g Histogram Shifting.

Example

Now we declared the DEAlgo class, how to use it?

- Create an instance of it and pass the full file path name of the image to the constructor
- Call CalcHighPass function
- Call DetermineLocations function
- Call CompressOverFlowMap function
- Call GetDelta Function
- Call OutterHistogramShift function
- Call BuildBitStream function
- Call EmbedBitStream function
- Call CompileImage function

And now we've embedded a random payload to the image, how to extract the payload and restore the original image?

- Create another instance of the DEAlgo class and pass the result image of the object encoder.
- Call CalcHighPass function
- Call GetCLocations function
- Call ExtractBitStream function
- Call DecompressOverFlowMap function
- Call IdentifyExpandedLocations function
- Call RestoreLSBs function
- Call RestoreExpanded function
- Call ReverseShift function
- Call CompileImage function

And now you're done, you have extracted the bitstream thus the payload and restored the original image.

III.3.d PEAlgo Class

The PEAlgo class also extends the parent class **EEAlgo** class

```
#include "EEAlgo.h"
class PEAlgo : public EEAlgo
{
private :
    Mat PredictedErrors = Mat(_imagePixels.rows, _imagePixels.cols, CV_16FC1);
    Mat PredictedVal= Mat(_imagePixels.rows, _imagePixels.cols, CV_8UC1);

    void Init(Mat pixels);
    uchar PixelVal(int row, int col);

public:
    PEAlgo(String fileName);
    PEAlgo(Mat pixels);
    void CalcPE();
    std::vector<unsigned char> Locations = std::vector<unsigned char>(_imagePixels.rows * _imagePixels.cols);
    void GetLocations();
    void GetDelta();
    void CompressOverFlowMap();
    void OutterHistogramShift();
    void BuildBitStream();
    void EmbedBitStream();
    void CompileImage();
    void GetCLocations();
    void ExtractBitStream();
    void DecompressOverFlowMap();
    void IdentifyExpandedLocations();
    void RecoverOriginalValues();
    //Compare
    bool CompareLocations(std::vector<uchar> inLocations);
    static bool isInRpRange(short prErr, uchar prVal);
};
```

Figure 47 PEAlgo Header Class

With a little bit of modifications.

```
uchar PEAlgo::PixelVal(int row, int col)
{
    if (row<0 || col<0 )
        return 0;
    return _imagePixels.at<uchar>(row,col);
}
```

Figure 48 PixelVal Function

To calculate the predicted value of a pixel we need its neighbor pixels, but the first pixel for example has no neighbor pixels (II.4.a Pixel Context), so when we try to access a row or a column that doesn't exist in the image matrix, we simply make it 0, for that case, we created a dedicated function that checks if the row or column number is negative then return 0 else return the true pixel value.

```
void PEAlgo::CalcPE()
{
    for (int i = 0; i < _imagePixels.rows; i++)
        for (int j = 0; j < _imagePixels.cols; j++)
        {
            uchar c1 = PixelVal(i - 1, j - 1), c2 = PixelVal(i - 1, j), c3 = PixelVal(i, j - 1);
            if (c1 <= std::min(c2, c3))
                PredictedVal.at<uchar>(i, j) = 2*floor((float)std::max(c2, c3)/2);
            else if (c1 >= std::max(c2, c3))
                PredictedVal.at<uchar>(i, j) = 2 * floor((float)std::min(c2, c3)/2);
            else
                PredictedVal.at<uchar>(i, j) = 2 * floor((float)(c2 + c3 - c1)/2);
        }
    //Calculating Predicted Errors
    for (int i = 0; i < _imagePixels.rows; i++)
        for (int j = 0; j < _imagePixels.cols; j++)
            PredictedErrors.at<short>(i, j) = (short)_imagePixels.at<uchar>(i, j) - (short)PredictedVal.at<uchar>(i, j);
}
```

Figure 49 CalcPE Function

We calculate the predicted values and errors using Equation 16 Predicted Value and Equation 17 Prediction Error

GetLocations function same as DEAlgo's, iterates over the predicted error values and marks the expandable and changeable ones and creates the overflow map that indicates the expandable ones

GetDelta function is also same DEAlgo's, iterates over the predicted errors and marks the errors that are in the range of $[-\Delta-1, \Delta]$.

CompressOverFlowMap function, compresses the overflow map using Google's snappy engine.

OutterHistogramShift function, iterates over the expandable predicted errors that are outside of the range $[-\Delta-1, \Delta]$ and shifts them by Δ to the right (positive side) and $-\Delta - 1$ to the left (negative side).

BuildBitStream function, same as DEAlgo's, it combines the segments together (Header, compressed ov map…).

EmbedBitStream function, same as DEAlgo's, it reads the bitstream variable **BS** as a stream of bits and then embeds it bit by bit to the predicted errors according to their state, EXPANDABLE_IN_DELTA ones by Expansion and CHANGEABLE \ EXPANDABLE ones by LSB-Replacement.

CompileImage function, same as DEAlgo's, calculates the watermarked pixels using the predicted values and errors.

These are the encoding functions that embeds the bitstream to an image, now we'll talk about the decoding functions

GetCLocations function, iterates over the predicted errors and marks them

ExtractBitStream function, same as DEAlgo's, it reads the LSBs of the changeable predicted errors and pushes them into the bitstream variable **BS** using our BitArray Class.

DecompressOverFlowMap function, decompresses the extracted bitstream using Google's snappy engine.

```
void PEAlgo::RecoverOriginalValues()
{
    BitArray LSBs = BitArray((char*)BS.LSBs, _imagePixels.rows * _imagePixels.cols);
    for (int i = 0; i < _imagePixels.rows; i++)
        for (int j = 0; j < _imagePixels.cols; j++)
        {
            uchar c1 = PixelVal(i - 1, j - 1), c2 = PixelVal(i - 1, j), c3 = PixelVal(i, j - 1);
            uchar predictedValue = 0;
            short predictedError = 0;
            if (c1 <= std::min(c2, c3))// a = c3 b = c2 c = c1
                predictedValue = std::max(c2, c3);
            else if (c1 >= std::max(c2, c3))
                predictedValue = std::min(c2, c3);
            else
                predictedValue = (c2 + c3 - c1);
            predictedValue = 2 * floor((float)predictedValue / 2);
            predictedError= (short)_imagePixels.at<uchar>(i, j) - (short)predictedValue;
            if ((*OverFlowMapM)[i * _imagePixels.rows + j])
                if (predictedError >= -2 * (short)BS.aInfo.header.Delta - 2 && predictedError <= 2 * BS.aInfo.header.Delta + 1)
                    Locations.at(i * _imagePixels.rows + j) = EXPANDABLE_IN_DELTA;
                else
                    Locations.at(i * _imagePixels.rows + j) = EXPANDABLE;
            else if (isChangable(predictedError, predictedValue, &isInRpRange))
                Locations.at(i * _imagePixels.rows + j) = CHANGABLE;
            else
                Locations.at(i * _imagePixels.rows + j) = NEITHER;
            // Based on the location map we recover the original bit of the predictedError
            switch (Locations.at(i * _imagePixels.rows + j))
            {
            case CHANGABLE: //C /E
                predictedError =ChangeBit(predictedError, LSBs.next());
                break;
            case EXPANDABLE:// Es
                //TODO: SHIFT
                predictedError =ChangeBit(predictedError, LSBs.next());
                if (predictedError < -2 * (short)BS.aInfo.header.Delta - 2)
                    predictedError += BS.aInfo.header.Delta + 1;
                else if (predictedError > 2 * BS.aInfo.header.Delta + 1)
                    predictedError -= BS.aInfo.header.Delta + 1;
                break;
            case EXPANDABLE_IN_DELTA: // Ee
                predictedError = floor((float)predictedError / 2);
                break;
            default:
                continue;
            }
            //RecoverPixel
            _imagePixels.at<uchar>(i, j) = predictedValue + (uchar)predictedError;
        }
```

Figure 50 RecoverOriginalValues Function

According to II.5.b Restoring the Original Image, the recovery process is sequential, since the first pixel has no context (equal to 0) so it's the same after the embedding the bitstream, so we start recovering the first pixel and then use the recovered pixel to restore the second pixel and so on.

We calculated the predicted error and value and then we use them to mark the C, E_e, E_s locations and then using the marked locations we restore the original pixel based on its location E_e by Equation 6 Restoring Expansion-Embedded Difference and $C \setminus E_e$ by Equation 10 LSB Embedding Notation, we repeat the same process for the next pixel until we restore all the image.

Example

For now, we've covered the class's functions and declarations, how to use it?

- Create an instance of the PEAlgo class giving the full image file path name as an argument for the constructor.
- Call CalcPE function
- Call GetLocations function

- Call CompressOverFlowMap function
- Call GetDelta function
- Call OutterHistogramShift function
- Call BuildBitStream function
- Call EmbedBitStream function
- Call CompileImage function

For now, we've completed the embedding process, now we need the code on how to recover the payload and restore the original image.

- Create an instance of the PEAlgo class giving the encoded image matrix as an argument for the constructor
- Call ExtractBitStream function
- Call DecompressOverFlowMap function
- Call RecoverOriginalValues function

And you're done, you've extracted the payload and recovered the original image.

To see the source code, you can check it out at:
https://github.com/h3x3ncr7pt/MCILGraduation/tree/master/MCILGraduation

Chapter IV Comparing the results

IV.1 Introduction

We'll be comparing the algorithms we've implemented so far (PE and DE) on multiple images and then we compare the results of each algorithm (pros and cons).

IV.1.a Images Test List

The images are 512 x 512 and 256 x 256 Grayscale:

Figure 51 Test Image 1 (256 x 256)

Figure 52 Test Image 2 (256 x 256)

Figure 53 Test Image 3 (512 x 512)

Figure 54 Test Image 4 (512 x 512)

Figure 55 Test Image 5 (512 x 512)

IV.1.b PSNR

The term **peak signal-to-noise ratio (PSNR)** is an expression for the ratio between the maximum possible value (power) of a signal and the power of distorting noise that affects the quality of its representation. Because many signals have a very wide **dynamic range**, (ratio between the largest and smallest possible values of a changeable quantity) the **PSNR** is usually expressed in terms of the logarithmic decibel scale.

Image enhancement or improving the visual quality of a digital image can be subjective. Saying that one method provides a better-quality image could vary from person to person. For this reason, it is necessary to establish quantitative/empirical measures to compare the effects of image enhancement algorithms on image quality. Using the same set of tests images, different image enhancement algorithms can be compared systematically to identify whether a particular algorithm produces better results. The metric under investigation is the **peak-signal-to-noise ratio**. If we can show that an algorithm or set of algorithms can enhance a degraded known image to resemble more closely the original, then we can more accurately conclude that it is a better algorithm. (23, s.d.)

$$PSNR = 20\log_{10}\left(\frac{MAX_f}{\sqrt{MSE}}\right)$$

Figure 56 PSNR Equation

Where MSE is Mean Squared Error which is the mean of the difference between the original image and the encoded image squared.

IV.2 Tests

We'll draw charts for every test image starting by Image 1:

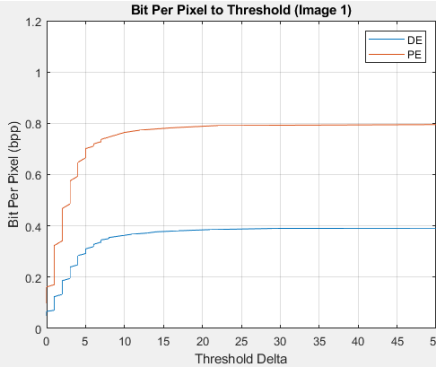

Figure 57 Bpp to Delta 1

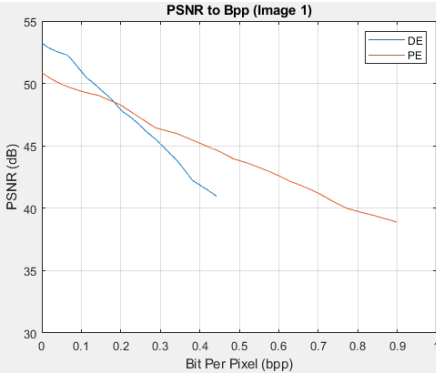

Figure 58 PSNR to Bpp 1

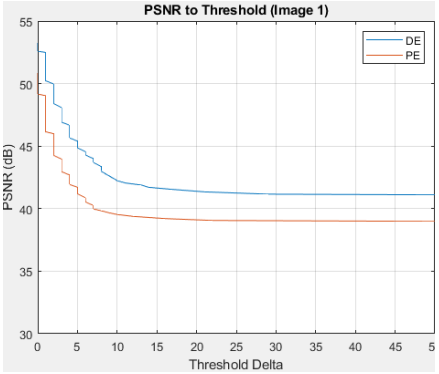

Figure 59 PSNR to Delta 1

With a max Payload capacity of **29032** bits for DE and **58932** bits for PE.

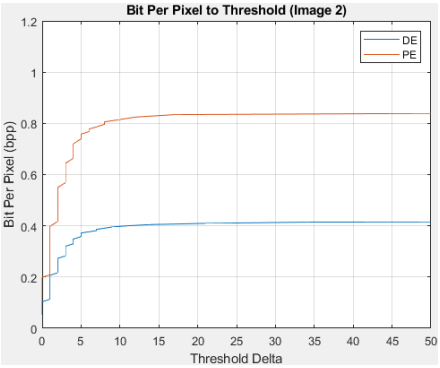

Figure 60 Bpp to Delta 2

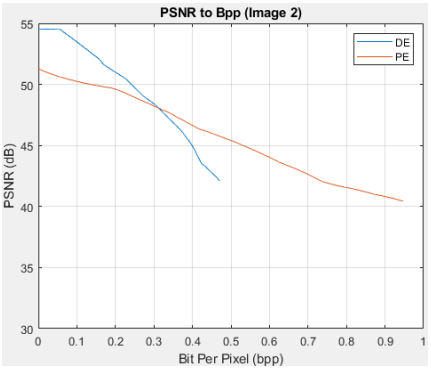

Figure 61 PSNR to Bpp 2

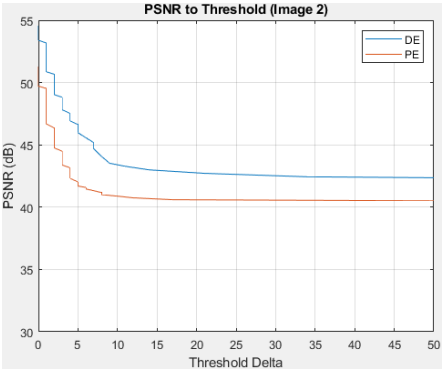

Figure 62 PSNR to Delta 2

With a max Payload capacity of **30856** bits for DE and **62113** bits for PE.

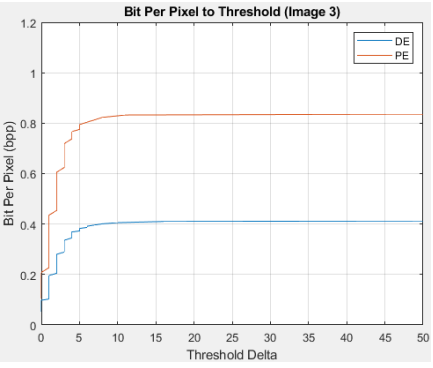

Figure 63 Bpp to Delta 3

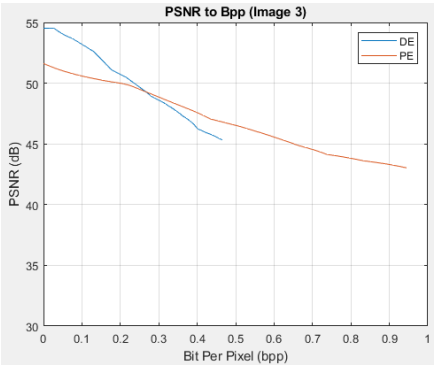

Figure 64 PSNR to Bpp 3

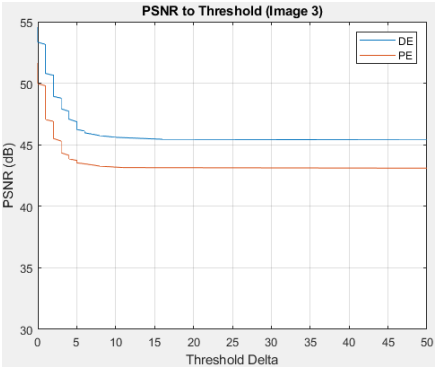

Figure 65 PSNR to Delta 3

With a max Payload capacity of **122210** bits for DE and **247962** bits for PE.

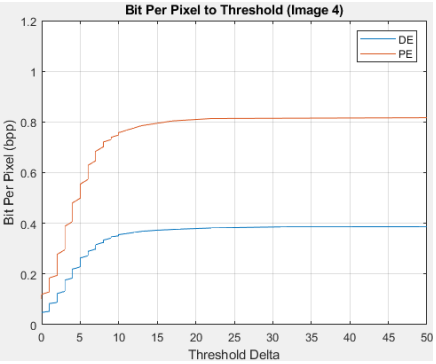

Figure 66 Bpp to Delta 4

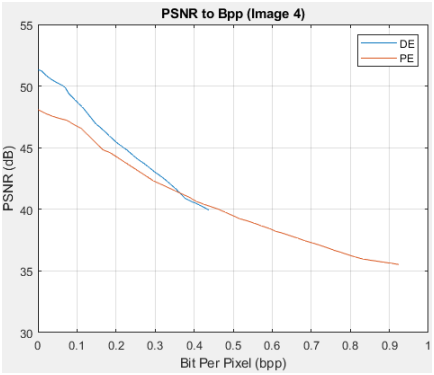

Figure 67 PSNR to Bpp 4

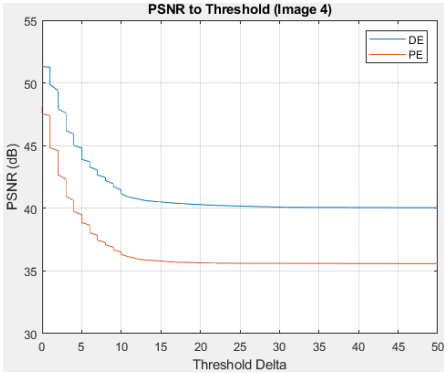

Figure 68 PSNR to Delta 4

With a max Payload capacity of **114952** bits for DE and **242295** bits for PE.

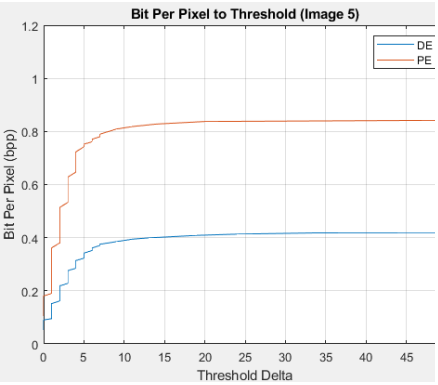

Figure 69 Bpp to Delta 5

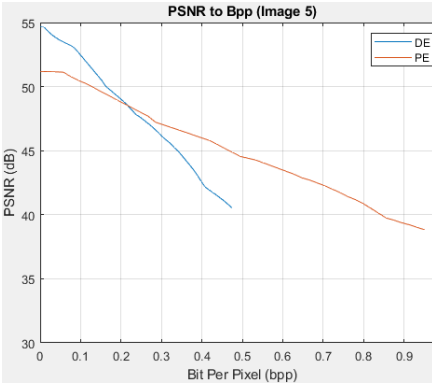

Figure 70 PSNR to Bpp 5

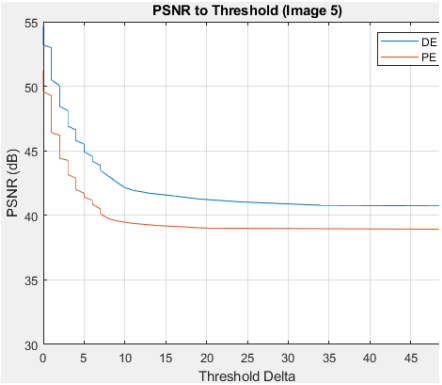

Figure 71 PSNR to Delta 5

With a max Payload capacity of **124563** bits for DE and **249457** bits for PE.

As we can clearly see on the results that the PE Algorithm has more (almost double) payload capacity but less PSNR quality unlike the DE Algorithm that has less payload capacity but more PSNR compared to BPP.
For BPP to Delta, we can see that both the algorithms reach their max BPP together (Same Delta), the PSNR at the max BPP rate for both algorithms we see that the DE algorithm has better PSNR than PE one, but if we see at the same BPP we see that the PE algorithm is always the better, also the DE algorithm has better PSNR at low BPP values (for all test images).

Finally, we say that if we want more embedding capacity, we should go for the PE algorithm, but we lose some PSNR and if the capacity isn't important but the PSNR is, we should go for DE then.

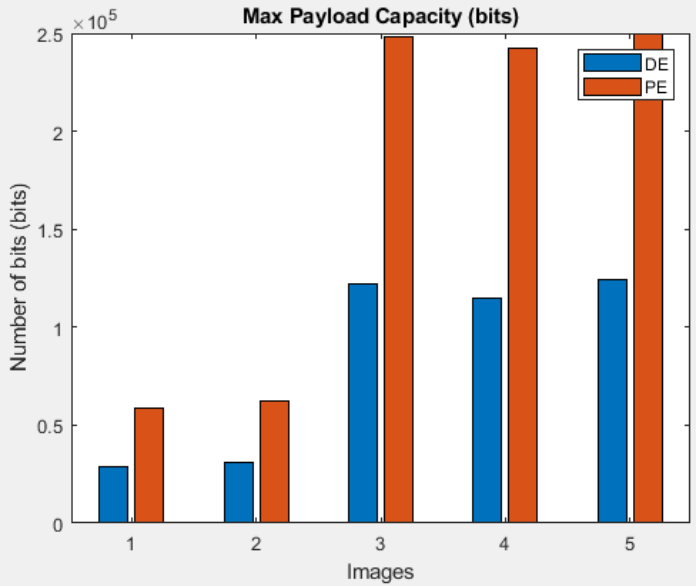

Figure 72 Max Payload Capacity

IV.3 Conclusion

In This article, we've studied and implemented two Reversible Data Hiding algorithms which are the Difference Expansion and the Prediction Error algorithms, they both have their own pros and cons which means that the optimal (the perfect) embedding technique is still not achieved yet, the major difference between them is the embedding capacity, the PE algorithm has double DE algorithm's payload capacity.

GENERAL CONCLUSION

We did a complete analysis of different data hiding techniques. Data hiding techniques become more important these days because of security maintaining requirements. These techniques have its own merits and demerits. Distortion level and payload capacity differ on every technique. Some methods reduce distortion levels whereas, other methods increase payload capacity. The main aim of these techniques is to provide security, undetectability, robustness, capacity. The methods in each work have a different level of applications. These methods overcome distortion problems and enhance image quality and contrast. This review paper presents how every technique differs and has its individual benefits and disadvantage. By using the techniques discussed above, data hiding and RDH can be achieved effectively. In the future, there is scope to invent applications by combining the above methods for improved data hiding with better security which can be also used in networks with lower bandwidth.

Watermarking is a very active research field with a lot of applications. Although it is a relatively new field, it has produced important algorithms for hiding messages into digital signals, RDH techniques recover the real data after doing encryption of the secret data. These techniques are used in the applications like military, medical imagery and forensics use this method for media registration, copyright protection, integrity authentication, etc. In this paper we discussed the techniques used for RDH (Reversible Data Hiding) over the last few years like lossless compression, histogram shifting, differential expansion, prediction error. All the above-mentioned techniques were implemented for better results in terms of robustness, capacity, security, and perceptibility.

REFERENCES

1. (n.d.). In *Barton JM (1997) Method and apparatus for embedding authentication information within digital data. US Patent 5646997, July 8.*

10. (n.d.). In *Fridrich J, Goljan J, Du R (2001) Invertible authentication. In: Proceedings of SPIE 2001, security and watermarking of multimedia content, San Jose, CA, Jan 2001.*

11. (n.d.). In *De Vleeschouwer C, Delaigle JF, Macq B (2003) Circular interpretation of bijective transformations in lossless watermarking for media asset management. IEEE Trans Multimed 5:97–105.*

12. (n.d.). In *Tian J (2002) Reversible watermarking by difference expansion. In: Dittmann J, Fridrich J, Wohlmacher P (eds) Proceedings of the workshop on multimedia and security: authentication, secrecy, and steganalysis, Dec 2002, pp 19–22.*

13. (n.d.). In *Alattar M (2004) Reversible watermark using the difference expansion of a generalized integer transform. IEEE Trans Image Process 13:1147–1156.*

14. (n.d.). In *Wang X, Shao C, Xu X, Niu X (2007) Reversible data hiding scheme for 2D vector maps based on difference expansion. IEEE Trans Inf Forensics Secur 2:311–320.*

15. (n.d.). In *Zhang X (2011) Reversible data hiding in encrypted image. IEEE Signal Process Lett.*

16. (n.d.). In *Hong W, Chen T-S, Wu H-Y (2012) An improved reversible data hiding in encrypted images using side match. IEEE Signal Process Lett 19:199–202.*

17. (n.d.). In *Lee Y-L, Tsai W-H (2014) A new secure image transmission technique via secret-fragment visible mosaic images by nearly reversible color transformations. IEEE Trans Circuits Syst Video Technol 24:695–703.*

18. (n.d.). In *Qian Z, Zhang X, Feng G (2016) Reversible data hiding in encrypted images based on progressive recovery. IEEE Signal Process Lett 23:1672–1676.*

19. (n.d.). In *Qian Z, Zhang X (2016) Reversible data hiding in encrypted images with distributed source.*

2. (n.d.). In *Tian J (2003) Reversible data embedding using a difference expansion. IEEE Trans Circuits Syst Video Technol 13(8):890–896.*

20. (n.d.). In *Zhang W, Wang H, Hou D, Yu N (2016) Reversible data hiding in encrypted images by reversible image transformation. IEEE Trans Multimed 18:1469–1479.*

21. (n.d.). In *Zhang X, Long J, Wang Z, Cheng H (2016) Lossless and reversible data hiding in encrypted images with public-key cryptography. IEEE Trans Circuits Syst Video Technol 26:1622–1631.*

22. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Digital_image

23. (n.d.). Retrieved from National Instruments: https://www.ni.com/en-lb/innovations/white-papers/11/peak-signal-to-noise-ratio-as-an-image-quality-metric.html

24. (n.d.). In *Expansion Embedding Techniques for Reversible Watermarking.*

25. (n.d.). In *Reversible data hiding techniques with high message embedding capacity in images.*

26. (n.d.). In *Adaptive Digital Watermarking for Copyright Protection of Digital Images in Wavelet Domain.*

3. (n.d.). In *Ni Z, Shi Y-Q, Ansari N, Su W (2006) Reversible data hiding. IEEE Trans Circuits Syst Video Technol 16(3):354–362.*

5. (n.d.). In *Lee S-K, Suh Y-H, Ho Y-S (2006) Reversible image authentication based on watermarking In: Proceedings of the IEEE international conference on multimedia expo (ICME), Toronto, ON, Canada, July 2006, pp 1321–1324.*

6. (n.d.). In *Li X, Yang B, Zeng T (2011) Efficient reversible watermarking based on adaptive prediction error expansion and pixel selection. IEEE Trans Image Process 20(12):3524–3533.*

7. (n.d.). In *Mintzer F, Lotspiech J, Morimoto N (1997) Safeguarding digital library contents and users: digital watermarking. D-Lib Magazine, Dec 1997.*

8. (n.d.). In *Honsinger CW, Jones P, Rabbani M, Stoffel JC (1999) Lossless recovery of an original image containing embedded data. US Patent application, Docket No 77 102/E-D.*

9. (n.d.). In *Macq B (2000) Lossless multiresolution transform for image authenticating watermarking. In: Proceedings of EUSIPCO 2000, Tampere, Finland, Sept 2000.*