ECE572 Digital Image Processing

Final project: Stegnography Implementation

Name: Sang Hyeb Lee

Student ID: 000330428

NetID: slee91

Abstract

Steganography is the technique to hide a secret message within a plain message in such way that others cannot discern the presence or content of the hidden message. It is widely used in copy-right marking and covert communication. For this project, four different steganography techniques are implemented. These are (i) a basic LSB substitution, (2) a pseudo-random interval LSB substitution, (3) DWT and Huffman coding method, and (4) High Capacity and Security Steganography using DWT. The result shows that each steganography has its advantage and disadvantage. For example, DWT and Huffman coding method increases privacy and robustness by performing Huffman coding on the secret message before embedding it in DWT domain.

1. Introduction

Steganography is derived from the Greek for “covered writing” and essentially means “to hide in plain sight” [1]. The main objective of steganography is to allow communication to be invisible by hiding secret information in innocuous messages. That means, unlike cryptography which only hides information from attackers, steganography even hides the very existence of the communication. Moreover, steganography makes it hard to remove a secret message without significantly altering the data in which it is embedded. This makes steganography suitable for many tasks for which cryptography is not such as copyright marking. Steganography is used in various areas: (i) protection of data alteration, (ii) copy-right marking, (iii) confidential communication, and etc.

The aim of this project is to investigate and implement available steganography techniques. For transform domain methods, two different discrete wavelet transform based approaches are examined: high capacity and security steganography using discrete wavelet transform by Reddy[2], and a DWT and Huffman encoding based technique by Nag[3]. For substitution systems, a basic LSB substitution and a pseudo random interval LSB technique are examined.

2. Steganography

2.1 What is steganography?

Steganography is the art and science of communicating in such a way that the presence of a message cannot be detected [1]. In steganography, a secret message is embedded in the cover image and exchanged in such a way that the existence of the secret message is undetectable. Figure 1 shows an example where steganography is used to embed a secret message.

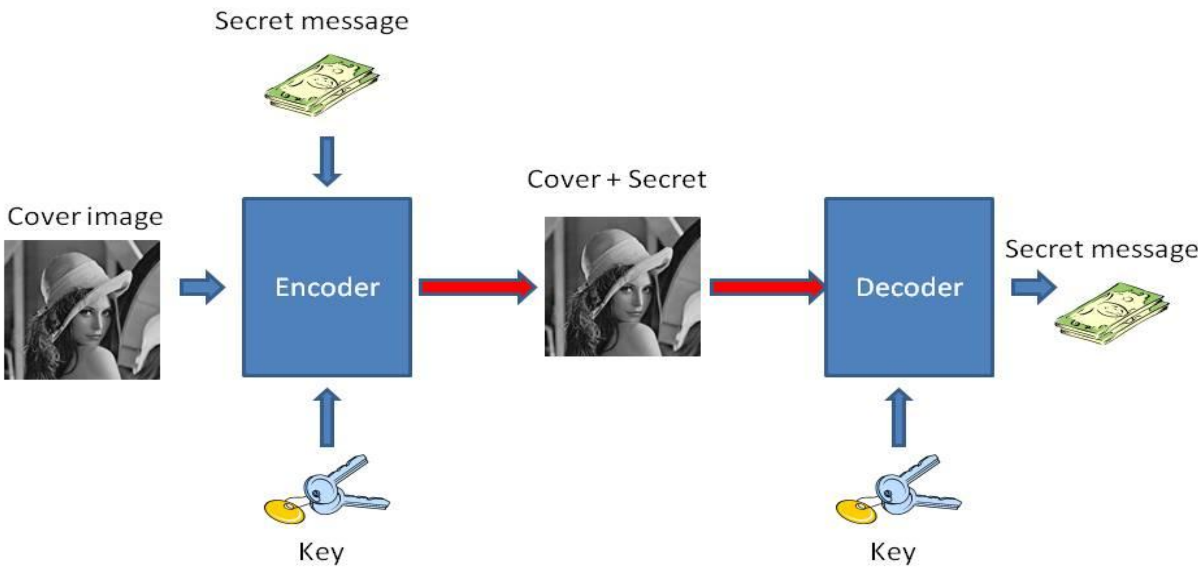


Figure 1 application of steganography for invisible communication

Steganography techniques can be classified into six main categories[1]: (i) substitution techniques, (ii) transform domain techniques, (iii)spread spectrum techniques, (iv) statistical techniques, (v) distortion techniques, and (vi) cover generation methods. Substitution technique works by substituting redundant parts of a cover with a secret message. It is a simplest method for data hiding but also the weakest in resisting attacks such as lossy compression. Transform domain technique embeds a secret message in a transform space of the cover image. Commonly used transform domain techniques are Discrete Cosine Transform (DCT), Discrete Wavelet

Transform (DWT), and Fast Fourier Transform (FFT). Spread spectrum technique spreads secret information over a wide spectrum, making it very difficult to remove message without destroying the cover image. In Distortion technique, a secret message is hidden in the signal distortion. Cover generation technique is different from others in that it creates a customized cover image to embed a secret message rather than using an existing cover image to hide a secret image. For this project, substitution technique and transform domain technique are examined.

Performance of a steganography technique can be measured in terms of robustness, utility, and imperceptibility. Robustness measures how robust an algorithm is against attacks like lossy compression. Utility measures how much data can be embedded into a cover image. Imperceptibility measures how likely an attacker can find out the existence of a secret message in a cover image.

2.1 Substitution technique

Substitution technique is a type of steganography technique which works by substituting redundant parts of a cover with a secret message. For this project, Least Significant Bit (LSB) substitution techniques are examined.

Least Significant Bit (LSB) Substitution

LSB substitution techniques work by using the least significant bits of each pixel in one image to embed the most significant bits of the message. Suppose that a pixel is represented by an 8-bit as shown in Figure 2.

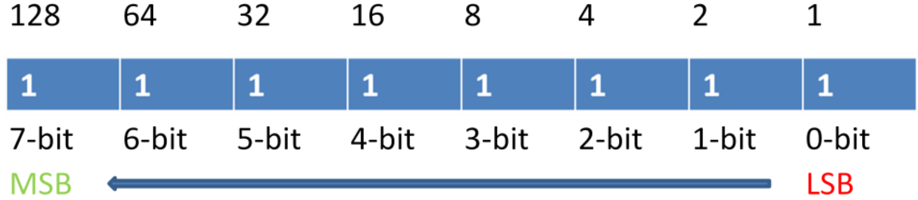
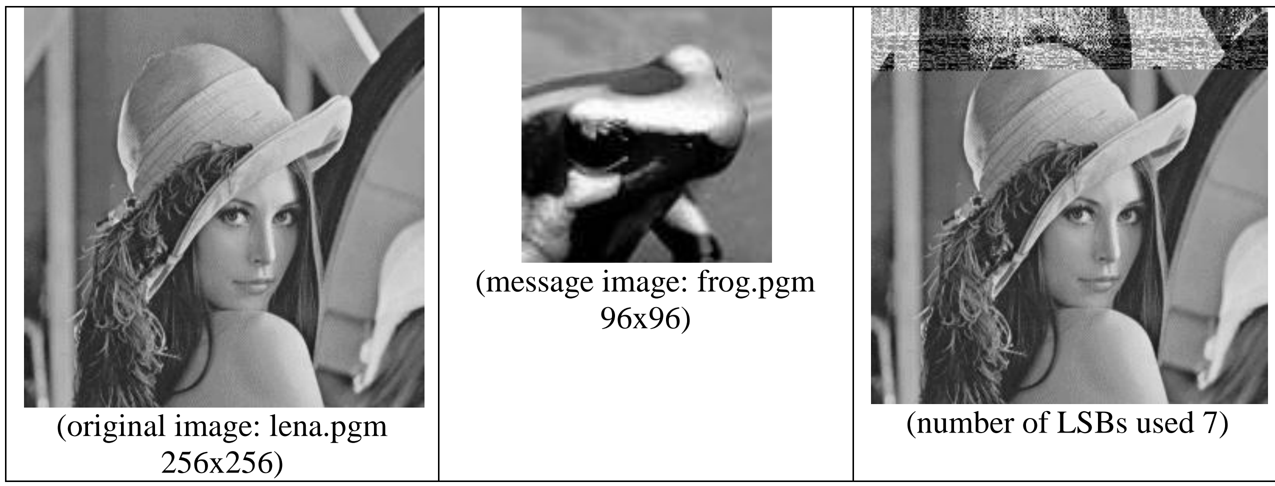


Figure 2 8-bit for a pixel

If we change the value of the 7-bit from 0 to 1, the pixel value will increase by 128. However, if we change the value of the 0-bit from 0 to1, the pixel value will increase by 1. Our goal is to keep the change to the minimal; moreover, the lower level bits are likely to contain noise. For this reason, a secret message is always hidden in the least significant bits.

The embedding process consists of replacing least significant bits in the cover image with most significant bits of the secret message. The length of LSB to replace in the original image depends

on the size of the secret message. It determines the robustness and the capacity of the algorithm. Figure 3 shows an example where too many LSBs are used to store the message.If we use more bits, the capacity will increase; however, it will be more venerable to the attacks as the cover image changed more. If we use fewer bits, the capacity will decrease but it will be more robust.



(original image: lena.pgm 256x256) (message image: frog.pgm 96x96) (number of LSBs used 7)

Figure 3 an example of using too many LSBs to store message

Figure 4 shows basic LSB substitution algorithm. Implementation of this method is discussed in Section 3. So we now have a basic LSB substitution algorithm. But how do we decide which pixel to embed the secret message into?

Embedding process

1. Convert the secret message into bit array.
2. For each bit in the array
    1. Compute the pixel index to store the bit
    2. Perform bitwise operation to embed the bit

Extraction process

1. For each bit in the array
    1. Compute the pixel index where the message bit is stored.
    2. Perform bitwise operation to extract the bit
2. Convert bit array to form the message.

Figure 4 basic LSB substitution algorithm

The simplest approach is sequentially using all pixels in the cover image, starting at the first element. However, a secret message will normally be smaller than the cover image. If this happens, the first part of the cover image will have different statistical properties than the second part where no pixels were modified. This is a serious security problem.

Pseudorandom interval

A more sophisticated way to choose a pixel index is the use of a pseudorandom number generator to randomly spread the secret message over the cover [1]. If the sender and the receiver have the same seed value for the random number generator, they can generate the same random number sequence and use this value to determine the pixel indices. Figure 5 shows pseudorandom interval algorithm. Implementation of this method is discussed in Section 3.

Embedding process

1. Convert the secret message into bit array.
2. Feed the seed value to the random number generator.
3. For each bit in the array
    1. Generate a random value, k.
    2. Index = (index + k) % MAX
    3. Perform bitwise operation to embed the bit into $\text{pixel}_{\text{index.}}$

Extraction process

1. Feed the seed value to the random number generator.
2. For each bit in the array
    1. Generate a random value, k.
    2. Index = (index +k) & MAX
    3. Perform bitwise operation to extract the bit from $\text{pixel}_{\text{index}}$
3. Convert bit array to form the message.

Figure 5 pseudorandom interval method

The pseudorandom interval method increases the complexity for the attacker since the pixel indices are randomly distributed. However, one index could appear more than once in the sequence, thus inserting more than one message bit into the same pixel value. This problem can be solved by using method like pseudo random permutation [1].

2.2 Transform Domain technique

While LSB substitution techniques provide easy ways to embed information in the cover image, they are highly vulnerable to simple attacks such as compression. If the attacker performs lossy compression on the cover image, the secret information could be destroyed. There has been much research to find steganography techniques which are more robust to such attacks. Transformation domain technique is one of them. Actually, most robust steganographic systems known today actually operate in some sort of transform domain [1]. There are various transform domain techniques: Discrete Cosine Transform (DCT), Discrete Fourier Transform (DFT),

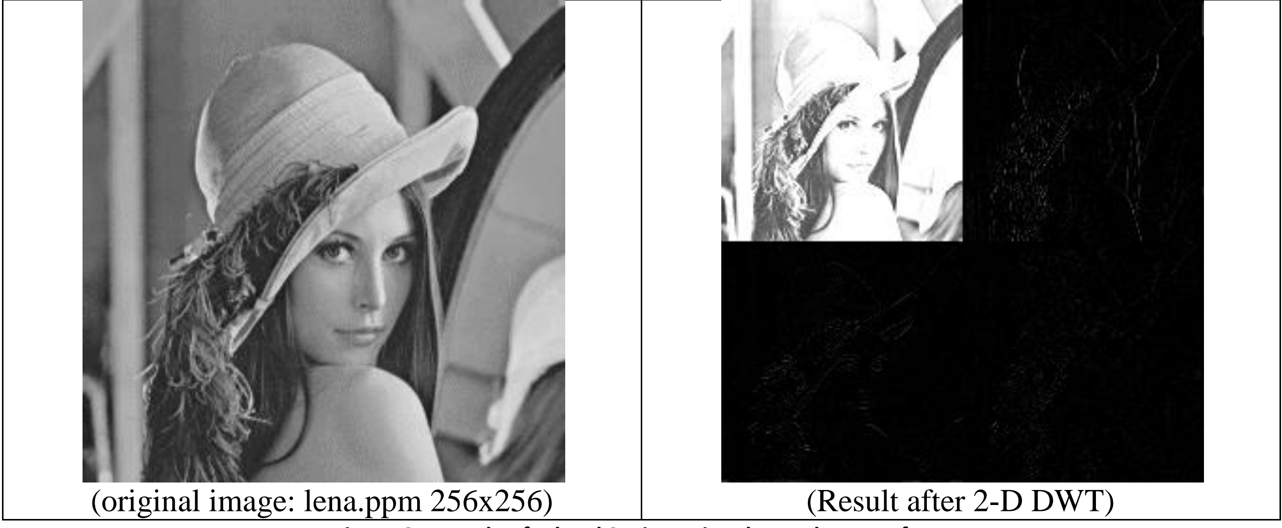
Discrete Wavelet Transform (DWT), and etc. For this project, I implemented DWT based techniques.

Discrete Wavelet Transform

LL **(approximation image)**	**HL** **(vertical edges)**
LH **(horizontal edges)**	**HH** **(detail components)**

Figure 6 components of 1-level 2-Dimensional Wavelet Transform

Wavelet transform is a mathematical transformation which converts a spatial domain into frequency domain. Unlike Fourier transformation, wavelet transform preserve the temporal information. That is, wavelet transformation can tell us which frequency components occur at what time. Moreover, wavelet transform clearly separates the high frequency and low frequency information on a pixel by pixel basis [2]. DWT is preferred over DCT because image in low frequency at various levels can offer resolution needed. For 2D images, applying DWT separates the image as shown in Figure 6. The approximate image in DWT consists of low frequency wavelet coefficients and the other three components contain the edge and texture details. An example of DWT components are shown in Figure 7.



(original image: lena.ppm 256x256) (Result after 2-D DWT)

Figure 6 example of 1-level 2-Dimensional Wavelet Transform

A Image Steganography based on DWT and Huffman Encoding

This method was proposed by Amitava Nag and et al in “A Novel technique for Image Steganography Based on DWT and Huffman Encoding” [2]. The basic idea behind this approach is that since human eyes are not sensitive to the small changes in the edges and the textures of an image, we can embed the secret message at the HH, HL, LH components of discrete wavelet transform. HH, HL, LH components store the edges and the textures whereas LL component store the smooth part of the image. The following figure shows the block diagram of the basic procedure.

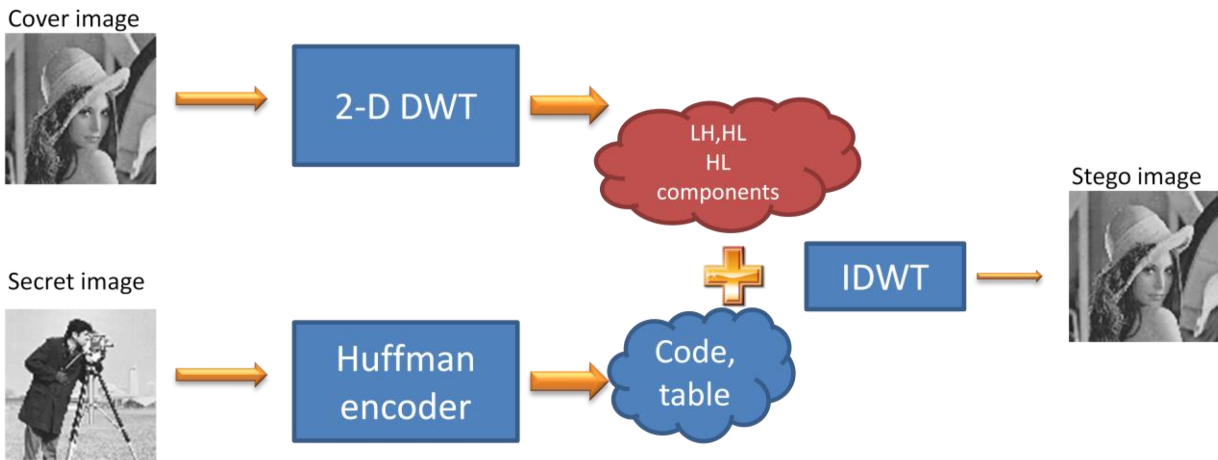


Figure 7 DWT + Huffman Encoding steganography technique

It works by first encoding the secret image using Huffman encoder to reduce the coding length and also to increase its privacy. It performs the 1-level 2-dimensional DWT on the cover image to get 3 subcomponents. It then embeds the Huffman table and Huffman code into three DWT subcomponents. So how do we decide the positions of the coefficient in the DWT component to which the secret image will be embedded? This paper does not describe how coefficients are chosen. I used a pseudo-random interval method when deciding the position of the coefficients. The implementation of this method is discussed in Section 3.

High Capacity and Security Steganography using DWT

This method was proposed by Manjunatha Reddy and Raja in “High Capacity and Security Steganography using Discrete Wavelet Transform”. Basically, their method works by adding coefficients of secret message and the cover image by using the equations below.

$$F(x,y) = \alpha C(x,y) + \beta p(x,y)$$
$$\alpha + \beta = 1$$

where F is modified DWT coefficients, C is the original DWT coefficients and P is the approximate band DWT coefficients of the message. the value of alpha and beta are chosen such that the secret message is not predominantly shown in the final image. Figure 8 consists of a block diagram of the basic procedure.

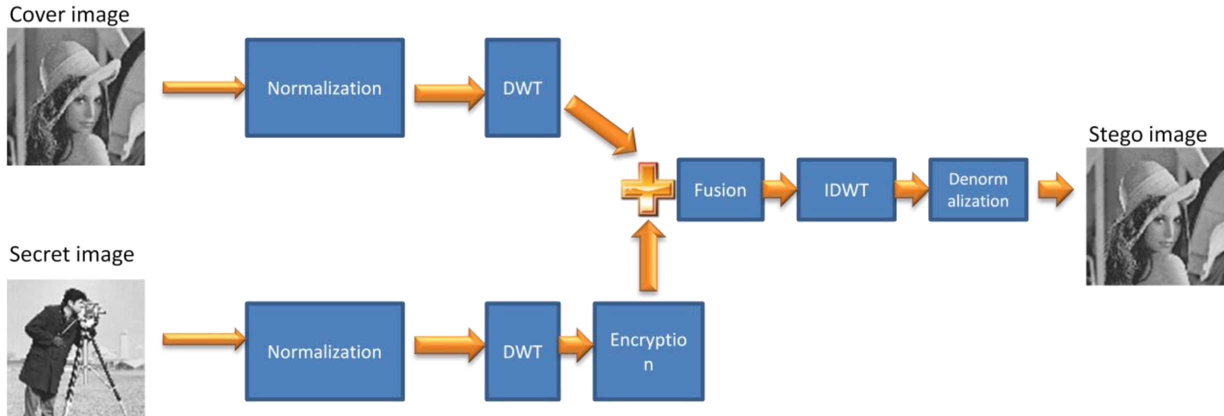


Figure 7 High Capacity and Security Steganography using DWT

This algorithm allows larger secret image to be embedded in the cover image by only embedded approximate component of the image. Major drawback of this algorithm is that the sender needs to have the original image to be able to extract the secret message from the stego image.

3. Implementation and performance analysis

Two LSB substitution techniques and two DWT domain techniques are implemented based on the Image library version 2 provided Dr. Qi. For the Image Steganography based on DWT and Huffman Encoding technique, I used an open source Huffman coding library [4]. To analyze performance each technique, I calculated PSNR and MSE to measure the difference between the original image and the stego image.

Two experiments were performed with each implementation. Figure 8 shows the cover image and the payload image used for the first experiment. The cover image is lena.pgm with size 256x256. The payload image is frog2.pgm with size 69x53.

(cover image, lena.pgm, 256x256)	(the payload image, frog2.pgm, 69x53)

Figure 8 the cover image and payload image used for test1

For the second experiment, the following cover image and payload image is used. The cover image is barbara.pgm with size 512x512. The payload image is cameraman2.pgm with size 156x127.

	(payload image, 156x127)

(cover image, 512x512)	

Figure 9 the cover image and payload image used for test2

In the second experiment, much larger payload image is used.

Basic Least Significant Bit substitution method

(stego image with 1 bit)

(restored message)

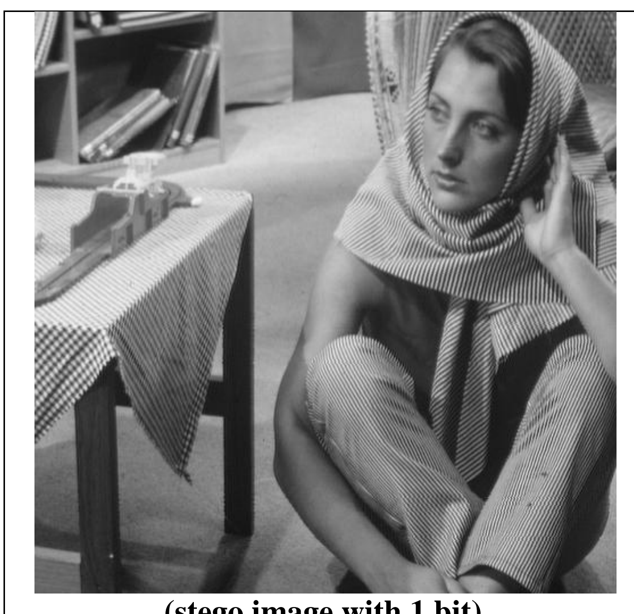
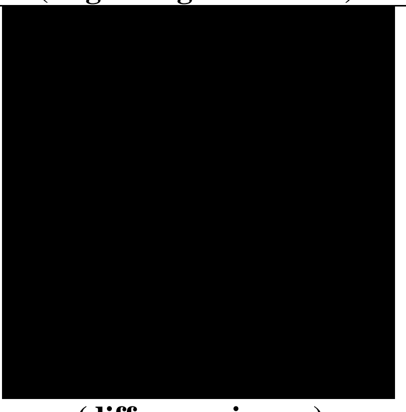
(difference image)

MSE: 0.464696
PSNR: 27.3766
CAPACITY: 29256

For the capacity of 29256, this technique generated a stego image with 0.464696 MSE and 27.3766 PSNR. I experimented with various size of LSBs. Here are the results.

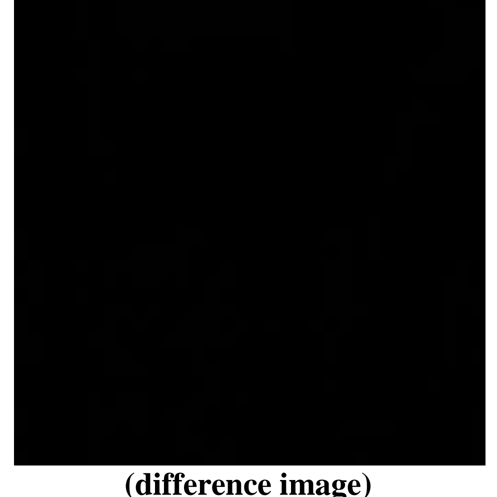
Size of LSBs	1 bit	2 bit	4 bit	6 bit
MSE	0.464696	0.786089	2.00663	7.51092
PSNR	27.3766	25.0936	21.0237	15.2914

As you can see in the above table, PSNR decreases as the number of LSBs used increases. This implies reciprocal relationship between robustness and imperceptibility.

<table>
<tr><td>
(stego image with 1 bit)</td><td>
(restored message)</td></tr>
<tr><td>
(difference image)</td><td>MSE: 0.68432
PSNR: 24.3766
CAPACITY: 158496</td></tr>
</table>

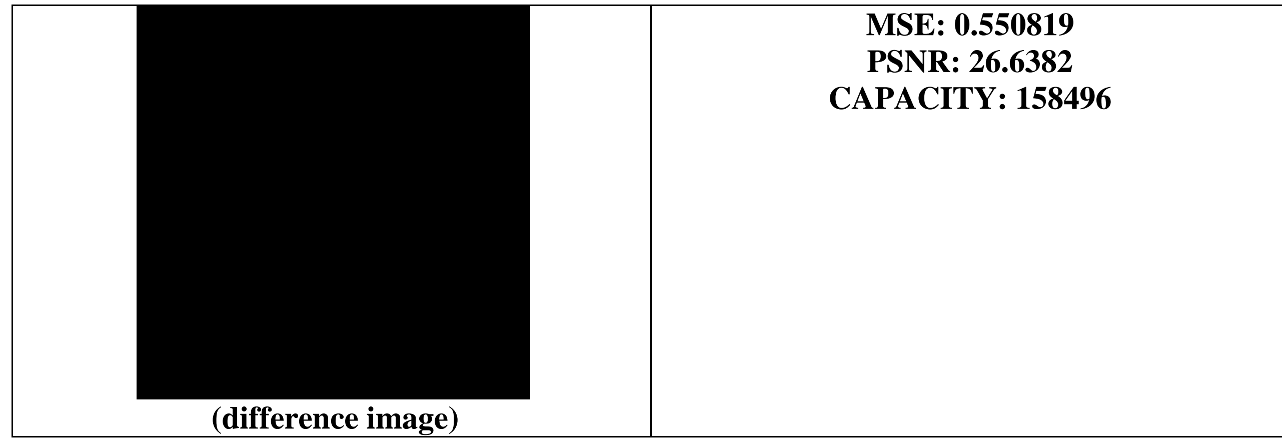
Pseudo-random interval

<table>
<tr><td>
(stego image with interval 25)</td><td>
(restored message)</td></tr>
</table>

<table>
<tr><td>
(difference image)</td><td>**MSE: 0.471428**
PSNR: 27.3142
CAPACITY: 29256</td></tr>
</table>

One of the problems I had with this implementation was that since the capacity is high, the same pixel index was used to encode more than one bit; therefore, the restored image contained many errors. I solved this problem by recording all the used pixel indices to prevent the same pixel from being used more than once. This slowed the computation; however, I got correct output.

<table>
<tr><td>
(stego image with 1 bit,interval=25)</td><td>
(restored message)</td></tr>
</table>

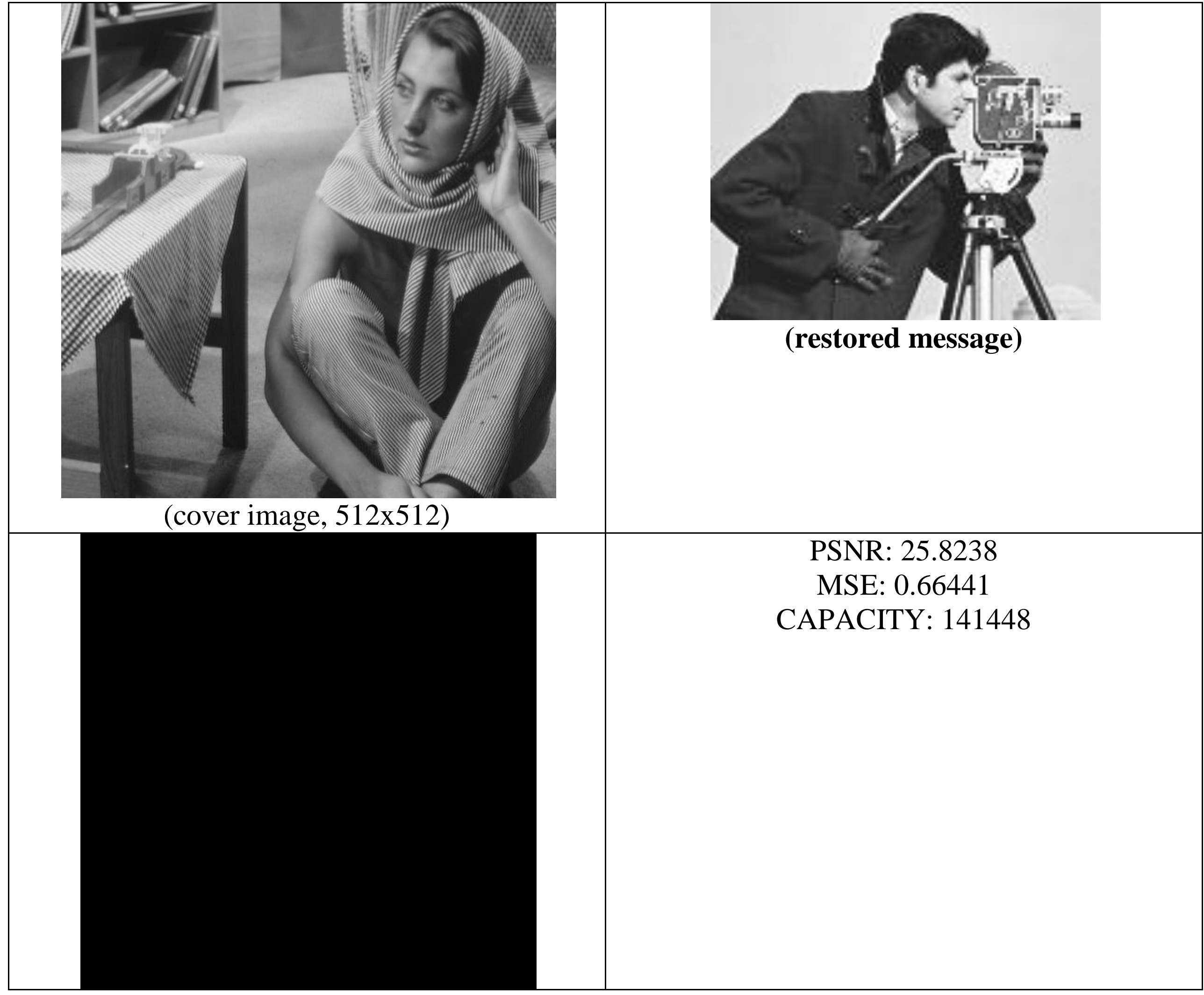
 (difference image)	MSE: 0.550819 PSNR: 26.6382 CAPACITY: 158496

A Image Steganography based on DWT and Huffman Encoding

I used a pseudo-random interval technique to determine the position of the coefficients in the DWT components. Here are the results.

(stego image with interval 25)	(restored message)
(difference image)	MSE: 0.661963 PSNR: 25.84 CAPACITY: 35384

One important thing to notice is that PSNR of this method is high compared to PSNR of LSB substitution techniques. This is because the capacity is much higher compared to those of LSB substitution techniques. Since the payload image is quite small, 69x53 and the capacity consists of Huffman code as well as Huffman table.



(cover image, 512x512)

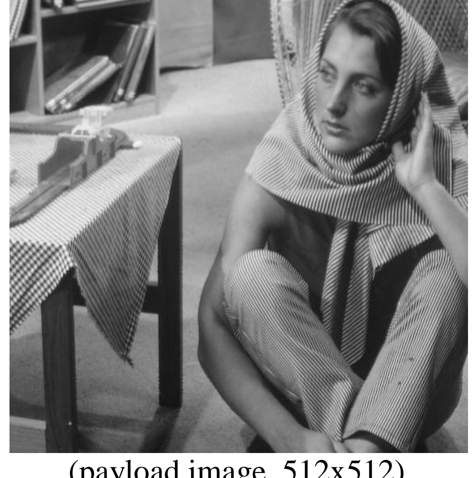
(restored message)

PSNR: 25.8238
MSE: 0.66441
CAPACITY: 141448

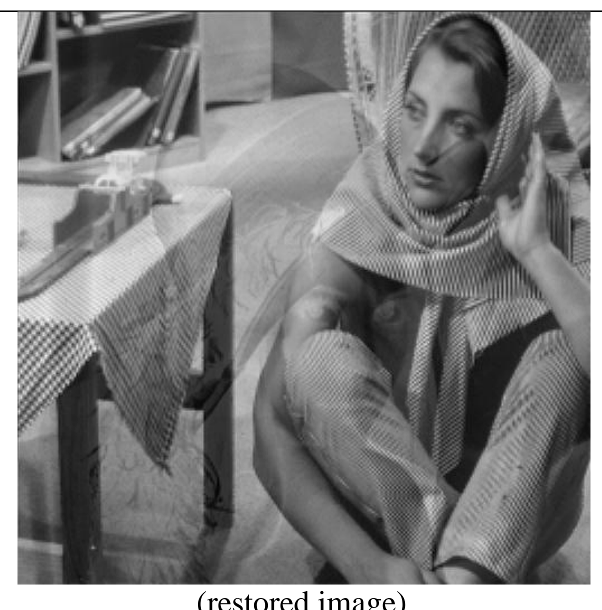
As you can see above, the capacity of this method is smaller compared to those of LSB techniques. This is because Huffman encoding reduces the capacity.

High Capacity and Security Steganography using DWT

My implementation does not encrypt or decrypt the payload image when embedding it into the cover image. For this implementation, I used the following cover and payload image for testing. The payload image is barbara.pgm with size 512x512. The cover image is lena.pgm with size 256x256.

<table>
<tr><td>
(cover image, 256x256)</td><td>
(payload image, 512x512)</td></tr>
</table>

This implementation allows high capacity to be embedded into the cover image by using only the approximate band.

<table>
<tr><td>
(stego image, alpha=0.9,beta=0.1)</td><td>
(restored image)</td></tr>
</table>

(difference image)	PSNR: 13.2224 MSE: 12.0946 Capacity: 2097152

PSNR is 13.2224 and MSE is 12.0946. However, the capacity is really high 2097152. The original implementation in the paper gave PSNR of 44.9 when the payload is twice large as the cover image. However, I could not achieve similar results.

4. Conclusion

Steganography is widely used when communication needs to be invisible. For this project, two LSB substitution and two DWT domain techniques are studied and implemented. These are (i) a basic LSB substitution, (2) a pseudo-random interval LSB substitution, (3) DWT and Huffman coding method, (4) High Capacity and Security Steganography using DWT. All these techniques allow the sender and the receiver to exchange a payload message by embedding it in a cover image. Each has its own advantage and disadvantage. LSB substitution techniques are simple to implement; however, they are vulnerable to attacks such as lossy compression. The DWT and Huffman coding based technique works in DWT domain. Moreover, this method compresses a payload image by using Huffman coding. However, in the first experiment where the payload image size is small, it actually increased the capacity as this method required the Huffman code table to be embedded as well. For the second test where the payload image is larger, the capacity was smaller and produced better PSNR.

References

[1] Stefan Katzenbeisser, and Fabien A. P. Petitcolas, editors, Boston:: Artech House, *Information hiding techniques for steganography and digital watermarking*, Artech House, 2000

[2] H S Manjunatha Reddy, "High Capacity and Security Steganography Using Discrete Wavelet Transform", *Journal of Computer Science and Security,* vol. 3, issue 6, 2009, pp. 462 - 472

[3] Amitava Nag, et al "a Novel Technique for Image Steganography Based on DWT and Huffman Encoding", *Journal of Computer Science and Security,* vol. 4, issue 6, 2010, pp. 561-570

[4] *libhuffman – an open source huffman coding library in C*, http://huffman.sourceforge.net, accessed at December 4 2011

```
/*********************************************************
 * steganography.cpp - steganography techniques
 *
 *   LSBEncoderText -- LSB substitution enboder for text
 *   LSBDecoderText -- LSB substitution decoder for text
 *   LSBEncoderImage -- LSB subsitution encoder for image
 *   LSBDecoderImage -- LSB subsitution decoder for image
 *   WTEncoderImage  -- high capcity DWT substitution
 *   WTDecoderImage -- high capacity DWT substitution
 *   WTDecoder2 -- DWT & Huffman decoder
 *   WTEncoder2 -- DWT & Huffman decoder
 *
 * Author: Sang Hyeb Lee slee91@utk.edu
 *
 * Created: December 1 2011
 *
 *********************************************************/

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <string>
#include <vector>
#include <cstdio>
#include "huffman/huffman.h"

using namespace std;

int globalsize;
Image stegoglobal;

/**
 * DWT & Huffman encoder
 * @param inimg cover image
 * @param filename file name of the message to be embedded in the cover image
 * @param seed seed for the random number generator
 * @param maxinterval max interval for the random number generator
 * @return DWT image
 */
Image WTEncoder2(Image &inimg, string filename,int seed, int maxinterval,int num_LS
B) {
  BitArray* bitarray;
  Image inimg_wt;
  Image LH,HH,HL;
  int nr,nc;
  int nr2,nc2;

  //(1) Decompose the cover image by using DWT
  inimg_wt = wt2d(inimg,0,1);

  nr = inimg.getRow();
  nc = inimg.getCol();
  nr2 = nr/2;
  nc2 = nc/2;

  writeImage(inimg_wt,"image_wt.pgm");

  //(2) Get huffman coding
  FILE* in = fopen(filename.c_str(),"r+");
  FILE* huffman = fopen("huffman.dat","w+");
  huffman_encode_file(in,huffman);
  fclose(huffman);

  //reading huffman code into memory
  huffman = fopen("huffman.dat","r+");
  vector<unsigned char> input;
  int read = fgetc(huffman);
  while(read!=EOF) {
    input.push_back((unsigned char)read);
    read = fgetc(huffman);
  }
  fclose(huffman);

  //(3) Convert huffman coding into bitstream
  bitarray = new BitArray();

  for(int i=0;i<input.size();i++) {
    for(int j=0;j<8;j++) {
      unsigned char target = input[i];
      unsigned char tester = 1<<(7-j);
      tester = tester & target;
      if(tester == 0)
        bitarray->push_back(0);
      else
        bitarray->push_back(1);
    }
  }

   //(3) hide information by altering the magnitude of the DWT coefficients of thre
e sub-bands, HH, LH,HL
  LH = subImage(inimg_wt, nr2,0,nr-1,nc2-1);
  HH = subImage(inimg_wt, nr2,nc2,nr-1,nc-1);
  HL = subImage(inimg_wt, 0,nc2,nr2-1,nc-1);

 //feed the seed value to the random number generator
  srand(seed);

  vector<int> positionbank;
  int position=0;
  //embedded the image
  cout<<"Bitarray size: "<<bitarray->size()<<endl;
  for(int i=0;i<bitarray->size();) {
    //work out the position
    position = position + ((double)rand()/RAND_MAX)*maxinterval+1;
    while(true) {
    if(position>(nr2*nc2)) {
      cout<<"encoder: position gone out of the range.. using mod now"<<endl;
      position = position %(nr2*nc2);
    }
    bool match=false;
    for(int k=0;k<positionbank.size();k++) {
      if(positionbank[k]==position) {
        match=true;
        //cout<<"decoder: "<<i<<endl;
      }
    }
    if(!match) break;
    else position++;
    }
   positionbank.push_back(position);
    //embedded into LH
    int pixel = (int)LH(0,position); //retrieve the original pixel
    for(int j=0;j<num_LSB&&(i+j)<bitarray->size();j++) {
      pixel = (pixel & (0xFFFFFFFF-(int)pow(2,j))) | ((*bitarray)[i+j]<<j);
    }
    LH(0,position) = (float)pixel;
    i+=num_LSB;

    //embedded into HL
    pixel = HL(0,position); //retrieve the original pixel
    for(int j=0;j<num_LSB&&(i+j)<bitarray->size();j++) {
      pixel = (pixel & (0xFFFFFFFF-(int)pow(2,j))) | ((*bitarray)[i+j]<<j);
```

```
    }
    HL(0,position) = (float)pixel;
    i+=num_LSB;

    //embedded into HH
    pixel = HH(0,position); //retrieve the original pixel
    for(int j=0;j<num_LSB&&(i+j)<bitarray->size();j++) {
      pixel = (pixel & (0xFFFFFFFF-(int)pow(2,j))) | ((*bitarray)[i+j]<<j);
    }
    HH(0,position) = (float)pixel;
    i+=num_LSB;
  }
  //delete bitarray;
  globalsize = bitarray->size();

  //(4) assemble all the pieces back
  for(int i=0;i<nr;i++) {
    for(int j=0;j<nc;j++) {
      //cout<<i<<","<<j<<endl;
      if(i>=nr2&&j>=nc2) {
        //HH component
        inimg_wt(i,j) = HH(i-nr2,j-nc2);
      }else if(i>=nr2&&j<nc2) {
        //LH component
        inimg_wt(i,j) = LH(i-nr2,j);
      }else if(i<nr2&&j>=nc2) {
        //HL component
        inimg_wt(i,j) = HL(i,j-nc2);
      }
      //LL component is not modified
    }
  }

  cout<<inimg_wt.getRow()<<".."<<inimg_wt.getCol()<<endl;
  cout<<"here"<<endl;
  //(5) apply IDWT to acuire the image
  writeImage(inimg_wt,"image_wt_e.pgm");
  cout<<"1"<<endl;
  Image stego = wt2d(inimg_wt,0,-1);
  stegoglobal = inimg_wt;

  return stego;

}

/**
 * Decoder for wavelet transform version 2
 * @param stego stego-image
 * @param size of the message
 * @param seed random number generator seed
 * @param maxinterval max random number size
 * @param num_LSB number of LSB
 * @param string output -> output file name
 */
void WTDecoder2(Image &stego,int size,int seed,int maxinterval,int num_LSB,string o
utput) {
  vector<unsigned char> input;
  BitArray* bitarray;
  Image stego_dwt;
  Image LH,HL,HH;
  int nr,nc;
  int nr2,nc2;
  Image stego_wt;
  int position;
  unsigned char value;

  //(1)apply DWT to the stego image
  stego_wt = stegoglobal;
  nr = stego_wt.getRow();
  nc = stego_wt.getCol();
  nr2 = nr/2;
  nc2 = nc/2;

  //(2) extract the bit array from the LL and HL, HH
  LH = subImage(stego_wt, nr2,0,nr-1,nc2-1);
  HH = subImage(stego_wt, nr2,nc2,nr-1,nc-1);
  HL = subImage(stego_wt, 0,nc2,nr2-1,nc-1);

  bitarray = new BitArray();
  position=0;
  // size = size * 8; //size of the bit array
  srand(seed); //initialize the random number generator
  vector<int> positionbank;

  cout<<"Size is "<<size<<endl;
  //extract the hidden message
  for(int i=0;i<size;) {
     //work out the position
    position = position + ((double)rand()/RAND_MAX)*maxinterval+1;
    while(true) {
    if(position>(nr2*nc2)) {
      cout<<"decoder: position gone out of the range.. using mod now"<<endl;
      position = position %(nr2*nc2);
    }
    bool match=false;
    for(int k=0;k<positionbank.size();k++) {
      if(positionbank[k]==position) {
        match=true;
        //cout<<"decoder: "<<i<<endl;
      }
    }
    if(!match) break;
    else position++;
    }
   positionbank.push_back(position);

    //extract from LH
    int pixel = LH(0,position); //retrieve the original pixel
    for(int j=0;j<num_LSB&&(i+j)<size;j++) {
      value = (pixel & ~(0xFFFFFFFF-(int)pow(2,j))) >> j;
      //cout<<"value: "<<(int)value<<endl;
      bitarray->push_back(value);
    }
    i+=num_LSB;

    //extract from HL
    pixel = HL(0,position); //retrieve the original pixel
    for(int j=0;j<num_LSB&&(i+j)<size;j++) {
      value = (pixel & ~(0xFFFFFFFF-(int)pow(2,j))) >> j;
      bitarray->push_back(value);
    }
    i+=num_LSB;

    //extract from HH
    pixel = HH(0,position); //retrieve the original pixel
    for(int j=0;j<num_LSB&&(i+j)<size;j++) {
      value = (pixel & ~(0xFFFFFFFF-(int)pow(2,j))) >> j;
      bitarray->push_back(value);
    }
    i+=num_LSB;
  }
  cout<<"Size of the bitarray: "<<bitarray->size()<<endl;

  //convert bitarray to huffman code
```

```
  input.clear();
  unsigned char temp;
  for(int i=0;i<bitarray->size();i+=8) {
    temp = 0x00;
    for(int j=0;j<8;j++) {
      temp = temp | (*bitarray)[i+j]<<(7-j);
    }
    //cout<<(int)temp<<endl;
    input.push_back(temp);
  }

  //unnnecessary code...
  //write the result to the file
  FILE *testout = fopen("huffman3.dat","w+");
  for(int i=0;i<input.size();i++) {
    fputc((int)input[i],testout);
  }
  fclose(testout);

  cout<<"HERE"<<endl;

  //conver the huffman code to the image
  FILE *huff = fopen("huffman3.dat","r+");
  FILE *restored = fopen(output.c_str(),"w+");
  cout<<"Decoding!!"<<endl;
  huffman_decode_file(huff,restored);
  cout<<"Decoded!!"<<endl;
  fclose(huff);
  fclose(restored);

  cout<<"BYE"<<endl;
  return;
}
/**
 * High capacity and security steganography using discrete wavelet transform
 * @param inimg input image
 * @param original original image
 * @param float alpha -> coefficient of the original image
 * @param float beta -> coefficient of the message image
 * @return image DWT image
 */
Image WTDecoderImage(Image &inimg, Image &original, float alpha, float beta) {
  Image inimg_normalized;
  Image original_normalized;
  Image message_dwt;
  Image message_idwt;
  Image inimg_dwt;
  Image original_dwt;
  Image message;

  //(1) normalize
  inimg_normalized = rescale(inimg,0,1);
  original_normalized = rescale(original,0,1);

  //(2) transform s into 2 levels of wavelet decomposition
  inimg_dwt = wt2d(inimg_normalized,0,1);
  original_dwt = wt2d(original_normalized,0,1);

  //(3) subtract DWT coefficient of c from DWT
  message_dwt = inimg_dwt- original_dwt*alpha;
  writeImage(message_dwt,"temp.pgm");
  message_dwt = message_dwt / beta;

  //(4) Decrypt

  //(5) Apply IWT of all the sub bands of p
  message_idwt = wt2d(message_dwt,0,-1); //this is an approximate

  //(6) Apply IWT of payload obtained with respect to approximate band
  Image restored;
  int nr = message_idwt.getRow();
  int nc = message_idwt.getCol();
  int nr2 = message_idwt.getRow() * 2;
  int nc2 = message_idwt.getCol() *2;
  restored.createImage(nr2,nc2);

  for(int i=0;i<nr;i++) {
    for(int j=0;j<nc;j++) {
      restored(i,j) = message_idwt(i,j);
    }
  }

  Image restored_2 = wt2d(restored,0,-1);

  //(6) Denormlize
  message = rescale(restored_2,0,255);

  return message;

}
/**
 * High capacity and security steganography using discrete wavelet transform
 */
Image WTEncoderImage(Image& inimg, Image &message,float alpha,float beta) {
  Image inimg_normalized;
  Image message_normalized;
  Image inimg_dwt;
  Image message_dwt;
  Image fusion;
  Image IDWT;
  Image rescaled;
  Image message_approximate;
  Image message_approximate_dwt;
  int nr,nc;

  //statistics
  nr = message.getRow();
  nc = message.getCol();

  //(1) normalization
  inimg_normalized = rescale(inimg,0,1);
  message_normalized = rescale(message,0,1);
  //   message_normalized = rescale(message,0,255);

  //(2) DWT on cover and the message
  inimg_dwt   =  wt2d(inimg_normalized,0,1);
  message_dwt =  wt2d(message_normalized,0,1);
  writeImage(message_dwt,"message_dwt.pgm");

  //(3) apply DWT on the approximate band of the payload
  message_approximate = subImage(message_dwt,0,0,(nr/2)-1,(nc/2)-1);
  message_approximate_dwt = wt2d(message_approximate,0,1);

  cout<<message_approximate.getRow()<<endl;
  writeImage(message_approximate,"approximate.pgm");
  //(3) Encrypt the message

  //(3) Fusion
  fusion = inimg_dwt*alpha+ message_approximate_dwt*beta;

  //(4) IDWT
  IDWT = wt2d(fusion,0,-1);

  //(5) Denormalization
```

```
  rescaled = rescale(IDWT,0,255);

  //(6) Stego Image
  return rescaled;
}

/**
 * Convert the given string into bit array
 */
BitArray* getBitArray(string& input) {
  BitArray* bitarray = new BitArray();

  for(int i=0;i<input.length();i++) {
    for(int j=0;j<8;j++) {
      unsigned char target = input[i];
      unsigned char tester = 1<<(7-j);
      //cout<<"Target is "<<(int)target<<endl;
      //cout<<"Tester is "<<(int)tester<<endl;
      tester = tester & target;
      //cout<<(int)tester<<endl;;
      if(tester == 0)
        bitarray->push_back(0);
      else
        bitarray->push_back(1);
      //bitarray->push_back((input[i]<<(7-j))>>7);
      //cout<<(int)'A'<<endl;
    }
  }

  return bitarray;
}

/**
 * convet image to an array of bits
 * @param image image to convert to an array of bits
 */
BitArray* getBitArrayFromImage(Image& inimg) {
  int nr,nc;
  vector<unsigned char> input;
  nr = inimg.getRow();
  nc = inimg.getCol();

  cout<<"getBitArrayFromImage: Size of the image is"<<nr*nc<<endl;
  //convert image to a string
  for(int i=0;i<(nr*nc);i++) {
    input.push_back(inimg(0,i));
  }

  BitArray* bitarray = new BitArray();

  for(int i=0;i<input.size();i++) {
    for(int j=0;j<8;j++) {
      unsigned char target = input[i];
      unsigned char tester = 1<<(7-j);
      tester = tester & target;
      if(tester == 0)
        bitarray->push_back(0);
      else
        bitarray->push_back(1);
    }
  }

  return bitarray;
}

/**
 * Convert the given bit array into string
 * @param bitarray array of bits
 */
string* getString(BitArray& bitarray) {
  string* input = new string();
  unsigned char temp;

  for(int i=0;i<bitarray.size();i+=8) {
    temp = 0x00;
    for(int j=0;j<8;j++) {
      temp = temp | bitarray[i+j]<<(7-j);
    }
    input->append(1,temp);
  }
  return input;
}

/**
 * convert bit-array into Image
 * @param bitarray array of bits
 * @param nr number of rows
 * @param nc number of columns
 */
Image getImage(BitArray &bitarray,int nr, int nc) {
  Image outimg;
  vector<unsigned char> input;
  unsigned char temp;

  for(int i=0;i<bitarray.size();i+=8) {
    temp = 0x00;
    for(int j=0;j<8;j++) {
      temp = temp | bitarray[i+j]<<(7-j);
    }
    //cout<<(int)temp<<endl;
    input.push_back(temp);
  }

  cout<<"getImage: size of string is "<<input.size()<<endl;
  outimg.createImage(nr,nc);

  for(int i=0;i<(nr*nc);i++) {
    outimg(0,i,0) = (float)(input[i]);
  }

  return outimg;
}
/**
 * Embed given image into the image
 * @param inimg the input image in RGB model
 * @param text a text which needs to be embedded
 * @param seed the seed value
 * @param num_LSB number of LSB to use
 * @param maxinetrval maximum value of the random number
 */
Image LSBEncoderImage(Image &inimg, Image& input, int seed,int num_LSB,int maxinter
val) {
  int nr, nc, nt, nchan; //image characteristics
  int *interval; //random interval
  Image outimg(inimg); //output image
  int position=0;
  BitArray *bitarray;

  //find out the image characteristics
  nr = inimg.getRow();
  nc = inimg.getCol();
  nt = inimg.getType();
  nchan = inimg.getChannel();
```

```
  //convert the input string into bit array
  bitarray = getBitArrayFromImage(input);

  cout<<"Encoder: size of bitarray is: "<<bitarray->size()<<endl;
  cout<<endl;
  //feed the seed value to the random number generator
  srand(seed);

  vector<int> positionbank;
  position=0;
  //embedded the image
  for(int i=0;i<bitarray->size();i+=num_LSB) {
    //cout<<"processing "<<i<<endl;

    position = position + ((double)rand()/RAND_MAX)*maxinterval+1;
    while(true) {
    if(position>(nr*nc)) {
      cout<<"encoder: position gone out of the range.. using mod now"<<endl;
      position = position %(nr*nc);
    }
    bool match=false;
    for(int k=0;k<positionbank.size();k++) {
      if(positionbank[k]==position) {
        match=true;
        //cout<<"Encoder: "<<i<<endl;
      }
    }
    if(!match) break;
    else position++;
    }
    positionbank.push_back(position);
    //cout<<"position "<<position<<endl;
    int pixel = outimg(0,position); //retrieve the original pixel
    for(int j=0;j<num_LSB;j++) {
      pixel = (pixel & (0xFFFFFFFF-(int)pow(2,j))) | ((*bitarray)[i+j]<<j);
    }
    outimg(0,position) = (float)pixel;
  }

  return outimg;
}

/**
 * extract text from the image
 * @param inimg the input image in RGB model
 * @param seed the seed value
 * @param number of LSB to use
 * @param image_nr row of the restored image
 * @param image_nc column of the restored image
 * @param maxinterval maximum value of random number
 */
Image LSBDecoderImage(Image &inimg, int seed,int num_LSB,int image_nr,int image_nc,
int maxinterval) {
  int nr, nc, nt, nchan; //image characteristics
  int *interval; //random interval
  Image outimg(inimg); //output image
  int position=0;
  BitArray bitarray;
  unsigned char value;
  int size;

  //find out the image characteristics
  nr = inimg.getRow();
  nc = inimg.getCol();
  nt = inimg.getType();
  nchan = inimg.getChannel();

  //feed the seed value to the random number generator
  srand(seed);

  position=0;
  vector<int> positionbank;
  size = image_nr*image_nc*8;
  //cout<<"Size is "<<size<<endl;
  //extract the image
  for(int i=0;i<size;i+=num_LSB) {

    position = position + ((double)rand()/RAND_MAX)*maxinterval+1;
    while(true) {
    if(position>(nr*nc)) {
      cout<<"decoder: position gone out of the range.. using mod now"<<endl;
      position = position %(nr*nc);
    }
    bool match=false;
    for(int k=0;k<positionbank.size();k++) {
      if(positionbank[k]==position) {
        match=true;
        //cout<<"decoder: "<<i<<endl;
      }
    }
    if(!match) break;
    else position++;
    }
   positionbank.push_back(position);
    //cout<<"position "<<position<<endl;
    int pixel = outimg(0,position); //retrieve the original pixel
    //extract the data
    for(int j=0;j<num_LSB&&(i+j)<size;j++) {

      value = (pixel & ~(0xFFFFFFFF-(int)pow(2,j))) >> j;
      bitarray.push_back(value);
    }

  }

  cout<<"Decoder: Size of bitarray: "<<bitarray.size()<<endl;
  cout<<endl;

  outimg = getImage(bitarray,image_nr,image_nc);
  return outimg;
}

/**
 * extract text from the image
 * @param inimg the input image in RGB model
 * @param seed the seed value
 * @param number of LSB to use
 * @param size of the text
 * @param maxinterval maximum random number value
 */
string* LSBDecoderText(Image &inimg, int seed,int num_LSB,int size,int maxinterval)
 {
  int nr, nc, nt, nchan; //image characteristics
  int *interval; //random interval
  Image outimg(inimg); //output image
  int position=0;
  BitArray bitarray;
  string* text;
  unsigned char value;

  //find out the image characteristics
  nr = inimg.getRow();
  nc = inimg.getCol();
  nt = inimg.getType();
```

```
  nchan = inimg.getChannel();

  //feed the seed value to the random number generator
  srand(seed);

  position=0;
  vector<int> positionbank;
  size = size * 8;
  //cout<<"Size is "<<size<<endl;
  //extract the image
  for(int i=0;i<size;i+=num_LSB) {

   position = position + ((double)rand()/RAND_MAX)*maxinterval+1;
    while(true) {
    if(position>(nr*nc)) {
      cout<<"decoder: position gone out of the range.. using mod now"<<endl;
      position = position %(nr*nc);
    }
    bool match=false;
    for(int k=0;k<positionbank.size();k++) {
      if(positionbank[k]==position) {
        match=true;
        //cout<<"decoder: "<<i<<endl;
      }
    }
    if(!match) break;
    else position++;
    }
   positionbank.push_back(position);
  //cout<<"position "<<position<<endl;
    int pixel = outimg(0,position); //retrieve the original pixel
    //extract the data
    for(int j=0;j<num_LSB&&(i+j)<size;j++) {

      value = (pixel & ~(0xFFFFFFFF-(int)pow(2,j))) >> j;
      bitarray.push_back(value);
    }

  }

  cout<<"Value of bitarray is "<<endl;
  for(int i=0;i<size;i++) {
    cout<<(int)bitarray[i];
  }
  cout<<"Size of bitarray: "<<bitarray.size()<<endl;
  cout<<endl;
  text = getString(bitarray);
  return text;
}

/**
 * Embed given text into the image
 * @param inimg the input image in RGB model
 * @param text a text which needs to be embedded
 * @param seed the seed value
 * @param number of LSB to use
 * @param maxinterval maximum value for the random number generation
 */
Image LSBEncoderText(Image &inimg, string input, int seed,int num_LSB,int maxinterv
al) {
  int nr, nc, nt, nchan; //image characteristics
  int *interval; //random interval
  Image outimg(inimg); //output image
  int position=0;
  BitArray *bitarray;

  //find out the image characteristics
  nr = inimg.getRow();
  nc = inimg.getCol();
  nt = inimg.getType();
  nchan = inimg.getChannel();

  //convert the input string into bit array
  bitarray = getBitArray(input);
  cout<<"Value of bitarray is "<<endl;
  for(int i=0;i<bitarray->size();i++) {
    cout<<(int)(*bitarray)[i];
  }
  cout<<endl;
  //feed the seed value to the random number generator
  srand(seed);

  position=0;
  vector<int> positionbank;
  //embedded the image
  for(int i=0;i<bitarray->size();i+=num_LSB) {
    position = position + ((double)rand()/RAND_MAX)*maxinterval+1;
    while(true) {
    if(position>(nr*nc)) {
      cout<<"decoder: position gone out of the range.. using mod now"<<endl;
      position = position %(nr*nc);
    }
    bool match=false;
    for(int k=0;k<positionbank.size();k++) {
      if(positionbank[k]==position) {
        match=true;
        //cout<<"decoder: "<<i<<endl;
      }
    }
    if(!match) break;
    else position++;
    }
   positionbank.push_back(position);

    //cout<<"position "<<position<<endl;
    int pixel = outimg(0,position); //retrieve the original pixel
    for(int j=0;j<num_LSB;j++) {
      pixel = (pixel & (0xFFFFFFFF-(int)pow(2,j))) | ((*bitarray)[i+j]<<j);
    }
    outimg(0,position) = (float)pixel;
  }

  return outimg;
}
```

```
// test code for lsb

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cstdlib>

using namespace std;

#define Usage "test_lsb2 inimg outimg messageimg size interval\n"

int main(int argc, char **argv)
{
  Image inimg, outimg;    // the original image
  int size;
  int interval;

  Image outimg2;
  Image messageimg;
  if(argc<6) {
    cout<<Usage;
    exit(-1);
  }
  // read in image
  inimg = readImage(argv[1]);
  messageimg = readImage(argv[3]);
  size  = atoi(argv[4]);
  interval = atoi(argv[5]);

  outimg = LSBEncoderImage(inimg, messageimg, 0,size,interval);
  outimg2 = LSBDecoderImage(outimg,0,size,messageimg.getRow(),messageimg.getCol(),i
nterval);

  writeImage(outimg,argv[2]);
  writeImage(outimg2,"restored.pgm");
cout<<"PSNR: "<<psnr(inimg,outimg)<<endl;
  cout<<"MSE: "<<rmse(inimg,outimg)<<endl;
  Image diff = inimg - outimg;
  writeImage(diff,"diff.pgm");
  int nr=messageimg.getRow();
  int nc=messageimg.getCol();
  for(int i=0;i<(nr*nc);i++) {
    //cout<<messageimg(0,i)<<endl;
    if( messageimg(0,i)>L) {
      cout<<"Over L: "<<messageimg(0,i)<<endl;
    }

  }

  return 0;
}
```

```
// test code for lsb

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cstdlib>

using namespace std;

#define Usage "test_lsb inimg outimg size interval\n"

int main(int argc, char **argv)
{
  Image inimg, outimg;    // the original image
  int size;
  int maxinterval;

  string test = "ECE572 is fun";

  if(argc<5) {
    cout<<Usage;
    exit(-1);
  }
  // read in image
  inimg = readImage(argv[1]);
  size  = atoi(argv[3]);
  maxinterval = atoi(argv[4]);



  outimg = LSBEncoderText(inimg, test, 0,size,maxinterval);
  cout<<"PSNR: "<<psnr(inimg,outimg)<<endl;
  cout<<"MSE: "<<rmse(inimg,outimg)<<endl;
  Image diff = inimg - outimg;
  writeImage(diff,"diff.pgm");

  string* outcome = LSBDecoderText(outimg,0,size,test.length(),maxinterval);
  cout<<"Retrived message"<<*outcome<<endl;
  cout<<"Size is "<<outcome->length()<<endl;
  writeImage(outimg,argv[2]);

  if(outcome->compare(test)==0)
    cout<<"Exact match between the initial message and the decoded message"<<endl;


  return 0;
}
```

```
// test code for contrast stretching

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cstdlib>
using namespace std;

#define Usage "testwt2 inimg1 message seed maxinterval lsb decoded\n"

extern int globalsize;

int main(int argc, char **argv)
{
  Image inimg;    // the original image
  Image outimg;
  int seed;
  int maxinterval;
  int lsb;


  // check if the number of arguments on the command line is correct
  if (argc <7) {
    cout << Usage;
    exit(3);
  }

  // read in image
  string filename(argv[2]);
  inimg = readImage(argv[1]);
  seed = atoi(argv[3]);
  maxinterval = atoi(argv[4]);
  lsb = atoi(argv[5]);
  string decoded(argv[6]);

  outimg = WTEncoder2(inimg,filename,seed,maxinterval,lsb);
  WTDecoder2(outimg,globalsize,seed,maxinterval,lsb,decoded);

  writeImage(outimg, "encoded.pgm");

  cout<<"PSNR: "<<psnr(inimg,outimg)<<endl;
  cout<<"MSE: "<<rmse(inimg,outimg)<<endl;
  Image diff = inimg - outimg;
  writeImage(diff,"diff.pgm");

  return 0;
}
```

```cpp
// test code for contrast stretching

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cstdlib>
using namespace std;

#define Usage "testwt inimg1 message outimg alpha beta\n"

int main(int argc, char **argv)
{
  Image inimg, message;    // the original image
  Image outimg;
  float alpha;
  float beta;

  // check if the number of arguments on the command line is correct
  if (argc < 6) {
    cout << Usage;
    exit(3);
  }

  alpha = atof(argv[4]);
  beta  = atof(argv[5]);

  // read in image
  inimg = readImage(argv[1]);
  message = readImage(argv[2]);

  outimg = WTEncoderImage(inimg,message,alpha,beta);
  Image outimg2 = WTDecoderImage(outimg,inimg,alpha,beta);
  // output the image
  writeImage(outimg, argv[3]);
     writeImage(outimg2,"retrieved.pgm");
  cout<<"PSNR: "<<psnr(inimg,outimg)<<endl;
  cout<<"MSE: "<<rmse(inimg,outimg)<<endl;
  Image diff = inimg - outimg;
  writeImage(diff,"diff.pgm");
  cout<<"capacity: "<<message.getRow()*message.getCol()*8<<endl;
  return 0;
}
```

```
/********************************************************************
 * Dip.h - header file of the Image processing library
 *
 * Author: Hairong Qi, hqi@utk.edu, ECE, University of Tennessee
 *
 * Created: 01/22/06
 *
 * Modification:
 ********************************************************************/

#ifndef DIP_H
#define DIP_H

#include "Image.h"
#include<vector>
#include<cmath>
#include<cstdlib>
typedef vector<unsigned char> BitArray;
#define PI 3.1415926

//////////////////////////////////////////////
// point-based image enhancement processing

Image cs(Image &,                     // contrast stretching
         float,                       // slope
         float);                      // intercept

/////////////////////////////////////////
// Substitution Systems
BitArray* getBitArray(string& input); //creates an array of bit from a string
string* getString(BitArray& bitarray);//creates a string from bitarray
Image LSBEncoderText(Image &inimg, string input, int seed,int num_LSB,int interval)
; //LSB text encoder
string* LSBDecoderText(Image &inimg, int seed,int num_LSB,int size,int interval); /
/LSB text decoder
Image LSBDecoderImage(Image &inimg, int seed,int num_LSB,int image_nr,int image_nc,
int maxinterval); //lsb image decoder
Image LSBEncoderImage(Image &inimg, Image& input, int seed,int num_LSB,int maxinter
val); //lsb image encoder

////////////////////////////////////////////
// Transform Domain steganogy techniques
Image WTEncoderImage(Image& inimg, Image &message,float alpha,float beta); //high c
apacity implementation
Image WTDecoderImage(Image &inimg, Image &original, float alpha, float beta) ; //hi
gh capacity implementation decoder
Image WTEncoder2(Image &inimg, string text,int seed,int maxinterval,int num_LSB); /
/DWT&Huffman Encoder
void WTDecoder2(Image &stego,int size,int seed,int maxinterval,int num_LSB,string o
utput); //DWT&Huffman Encoder

///////////////////////////////////
// wavelet transform
Image wt2d(Image &,                   // the 2-D image
           int,                       // level of wt decomposition/coefficients
           int);                      // forward transform (>=0) or inverse (-1)
Image wt1d(Image &,                   // the 1-D row vector
           int);                      // forward transform (>=0) or inverse (-1)
void daub4(Image &,                   // the 1-D row vector
           int,                       // level of transform
           int);                      // forward transform (>=0) or inverse (-1)

int floorPower2(int);                 // the largested power of 2 <= the given

////////////////////////////////////
// utility
float rmse(Image &inimg1, Image &inimg2); //MSE
float psnr(Image &inimg1, Image &inimg2); //PSNR
#endif
```