Final Project: Steganography Implementation

Steven Young: ECE 572

Due: December 5, 2011

Abstract

The purpose of this project is to explore some Steganography techniques. Steganography refers to hiding information inside a cover, such as an image, such that not only is the information hidden, but the existence of the information is hidden. Steganography has many requirements it must satisfy. Among these are large capacity, high security level, and high imperceptibility, but it does not have to be robust against cover modifications. Steganography is a way that two or more people can communicate in a covert manner. A typical application example is to transfer secret messages embedded in images using a public website. The images in this example are accessible to all users, but only those with the secret key(s) can view the hidden information. In this project, two Steganography implementations will be discussed. The first will attempt to hide information in the least significant bits of an image and the second will use the coefficients of the Discrete Cosine Transform to hide information.

Tasks

In order to complete this project it was necessary for the following tasks to be completed:

1. Implement random sequence generator
2. Implement DCT
3. Implement Inverse DCT
4. Implement LSB Steganography Encryption Method
5. Implement LSB Steganography Decryption Method
6. Implement DCT Steganography Encryption Method
7. Implement DCT Steganography Decryption Method

Introduction (problems with state-of-the-art algorithms, briefly state the advantages of your approach)

Background and Motivation

The classic Steganography problem as described in the reference material for this project is the Prisoner's problem. Two friends, Alice and Bob, are arrested and thrown into separate cells. They can communicate, but all communications between them are arbitrated by Wendy. Wendy will not let Alice and Bob communicate through encryption, and if she detects encryption (even if she doesn't decrypt the message) she will shut down all communication between Alice and Bob. Thus, Alice and Bob must find some discreet way to hide meaningful information in messages that appear completely harmless to Wendy.

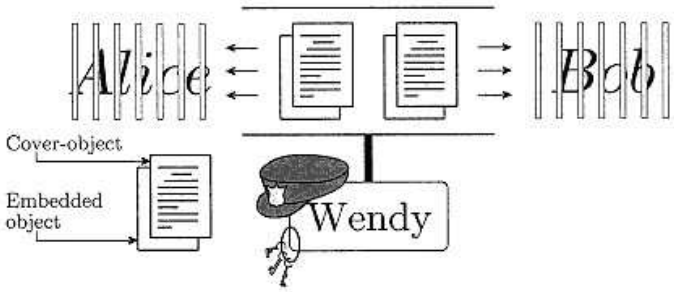


Figure 1: Diagram of Prisoner's Problem

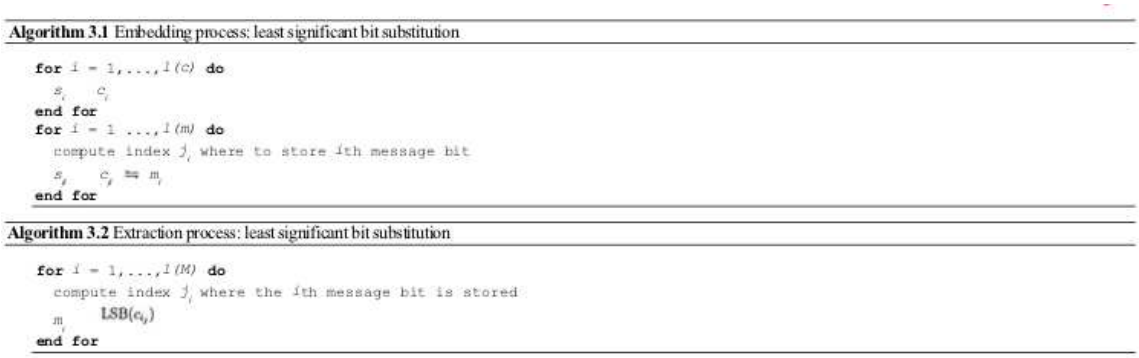
Thus, when Alice and Bob embed secret messages in the harmless object, the cover, they need to make sure they don't leave behind a detectable statistical footprint or visible distortion in the cover.

Existing Algorithms

LSB Method

This method works by placing the message in the LSB of elements of the cover. In and 8-bit carrier, a surprising amount of information can be hidden without any perceptible impact to the carrier. The embedding process consists mainly of choosing which elements of the cover to use to carry the message. Both the Alice and Bob need to know the method and the key(s) (or seed(s)) used to choose the elements used to carry the message. The biggest problem with this approach is they are typically used in lossless compression formats and are susceptible to the message being lost via lossy compression or noise injection.

The simplest way to choose which cover elements is to start with the first element and stop once you have encoded all the message bits. This algorithm is shown in the figure below.



Algorithm 3.1 Embedding process: least significant bit substitution

```
for i = 1,...,l(c) do
  s_i  c_i
end for
for i = 1 ...,l(m) do
  compute index j_i where to store ith message bit
  s_j  c_j ← m_i
end for
```

Algorithm 3.2 Extraction process: least significant bit substitution

```
for i = 1,...,l(M) do
  compute index j_i where the ith message bit is stored
  m_i  LSB(c_j)
end for
```

Figure 2: Simple LSB Algorithm

This has a serious security flaw though. The first part of the message will have significantly different statistical properties than the rest of the image. This will make the fact that an encoded message exists easy to detect. Thus, it is best to spread the message across the entire message using some random process. The algorithm below outlines such a process.

Algorithm 3.3 Embedding process: random interval method

```
for i = 1...,l(c) do
  s_i ← c_i
end for
generate random sequence k_i using seed k
n ← k_1
for i = 1,...,l(m) do
  s_n ← c_n ⊕ m_i
  n ← n + k_i
end for
```

Algorithm 3.4 Extraction process: random interval method

```
generate random sequence k_i using seed k
n ← k_1
for i = 1,...,l(m) do
  m_i ← LSB(c_n)
  n ← n + k_i
end for
```

Figure 3: More Robust LSB Algorithm

Transform Method

This method works by taking the Discrete Cosine Transform of 8x8 patches of a cover, encoding the message in the transform coefficients, and transforming it back to a carrier resembling the original cover. The message is encoding by making sure the comparison of two coefficients within the block encodes the message. The coefficients chosen should represent middle frequencies since most of the information in in image is at high frequencies (edges) and low frequencies (background). Also, to ensure that the image can withstand some tampering and compression, it should be made sure that the difference in the coefficients is greater than some threshold. Such a process is outline below.

Algorithm 3.8 DCT-Steg encoding process

```
for i = 1,...,l(M) do
  choose one cover-block b_i
  B_i = D(b_i)
  if m_i = 0 then
     if B_i(u_1, v_1) > B_i(u_2, v_2) then
        swap B_i(u_1, v_1) and B_i(u_2, v_2)
     end if
  else
      if B_i(u_1, v_1) < B_i(u_2, v_2) then
         swap B_i(u_1, v_1) and B_i(u_2, v_2)
      end if
  end if
  adjust both values so that |B_i(u_1, v_1) - B_i(u_2, v_2)| > x
  b'_i = D^-1(B_i)
end for
create stego-image out of all b'_i
```

Algorithm 3.9 DCT-Steg decoding process

```
for i = 1,...,l(M) do
  get cover-block b_i associated with bit i
  B_i = D(b_i)
  if B_i(u_1, v_1)    B_i(u_2, v_2) then
     m_i = 0
  else
     m_i = 1
  end if
end for
```

Figure 4: DCT Transform Algorithm

This method is fairly robust to JPEG compression because it also uses an 8x8 DCT to compress the image. A larger x value will result in a method that is more robust to compression, but it will distort the cover image more.

Technical Approach (detail description of the approaches you designed)

Random Sequence Generator

I implemented a random sequence generator with the goal of spreading the message randomly over the entire image. It needs 3 parameters to operate. It needs a seed, the length of the message, and the length of the image in pixels (or blocks for the DCT method). It then generates a random number between 1 and the length of the image left to use and the number of bits of the message left to encode. This random number is then added to the previous index that was chosen to form the next number in the sequence. The process is outline below.

Algorithm 0.1 Random Sequence Generator

```
seq[0]=rand(1:(img_len/msg_len))
for i=1:N
img_left = img_len - seq[i-1]
msg_left = msg_len-i;
seq[i]=rand(1:(img_left/msg_left))
endfor
```

DCT and Inverse DCT Transforms

The DCT transform is a simple transform commonly used in image processing. However, it did not exist in the DIP library provided so it had to be implemented. The algorithms used for the normal and inverse transforms are shown below.

$$S(u,v) = \frac{2}{N}C(u)C(v)\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}s(x,y)\cos\left(\frac{\pi u(2x+1)}{2N}\right)\cos\left(\frac{\pi v(2y+1)}{2N}\right)$$

$$s(x,y) = \frac{2}{N}\sum_{u=0}^{N-1}\sum_{v=0}^{N-1}C(u)C(v)S(u,v)\cos\left(\frac{\pi u(2x+1)}{2N}\right)\cos\left(\frac{\pi v(2y+1)}{2N}\right)$$

Figure 5: DCT and Inverse DCT Transforms

LSB Steganography Encryption and Decryption Method

The method used is the same as given in the Existing Methods discussion except that it uses my random sequence generator.

DCT Steganography Encryption Method

The method used is similar to the method in the Existing Methods discussion except that instead of just using sequential blocks starting from the beginning of the image, blocks to be used are selected using my random sequence generator.

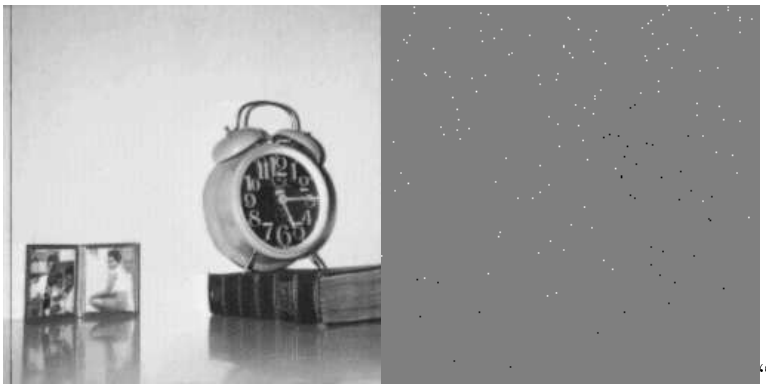
Experiments and Results (design the experiment, results, comment on the results)

We will be working with the following image during the experiments.

Figure 6: Original Image

Basic Experiment

For the first experiment we will simply take an image, encrypt a message into it, and then try to decrypt the message. This method is designed mainly to test the function of my code and to display that my random sequence generate spreads the message across the entire image. The test message will be "Test message for the final ECE 572 Project." Below are the results.


"Test message for the final ECE 572 Project."

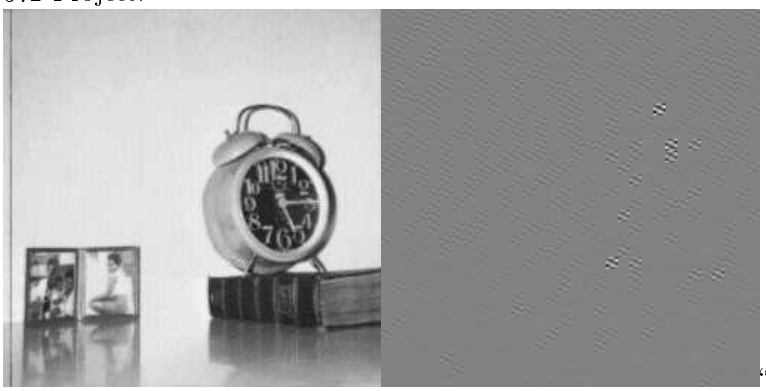
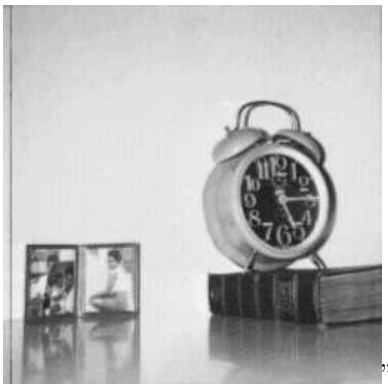

"Test message for the final ECE 572 Project."

Figure 7: Image with Encryption, Rescaled Difference Between Original and Encrypted Image, and Decrypted Message (LSB on Top; DCT on Bottom)

When there is no compression or other interference, the LSB method could be viewed as better because it can hold a 64 times longer message. However, the Transform method could still be viewed as better because it is more complex, making it more difficult to detect.

Compression Effect Experiment

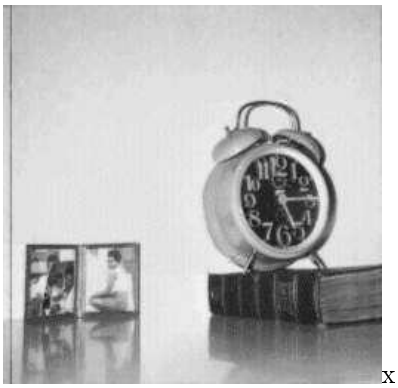
The goal of this experiment will be to determine the effect compression has on being able to decipher and read the message. For this experiment, we will be encrypting a message, using JPEG compression with varying compression levels and then try to decipher the message. GIMP will be used to perform the compression.

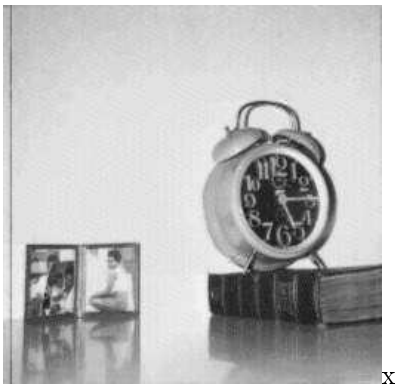

”%Z%%7%%L%#Y%]%%%%G%y%
3%qG%>SsC%%%A%J%%”

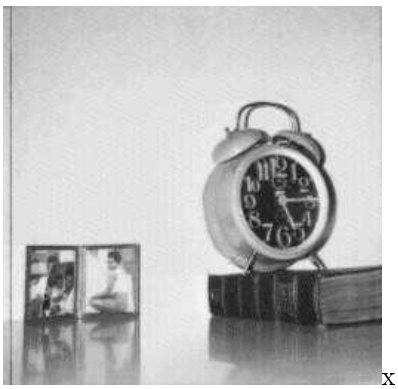
“Tmsv %es3age fMr%the Fi%al EcE 572 Project.”

Figure 8: Image with Encryption and Compression to 95% and Decrypted Message (LSB on Top; DCT on Bottom(x=0.5))

As you can see, the LSB method fails quickly and the Transform method is still readable, even with the small value of x. Below we experiment with the DCT method and different levels of compression and different values of x.


x=4; “[%%%%t%:Z%"w%%%%%%%+k4%o%o{ce~S%gkj%/%>”


x=8; “#%%%*Z%"%%%%%%%+n4%%OoCa~?%fkja/%?”


x=16; “Test message for the Final ECE 572 Project.”

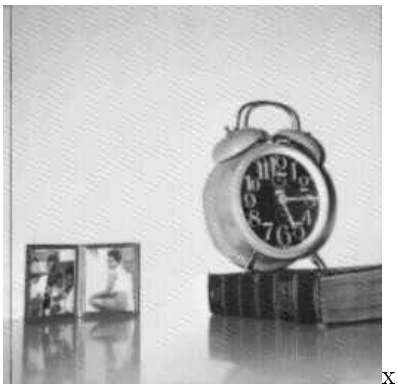
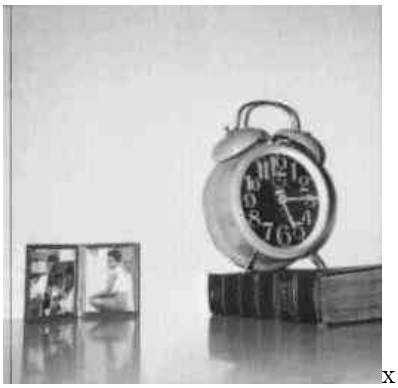

x=64; “Test message for the Final ECE 572 Project.”

Figure 9: Image with Encryption and DCT Compression to 65% and Decrypted Message

Above you can see when x=64, the large value of x is beginning to corrupt the image. However, smaller values of x such as x=8 are not large enough to withstand compression.


x=16; “null”


x=24; “%%0%‘%G>%%%ƒj%%%%%%%w%%nm%C%%e%?B%%z%6%91?”


x=64; “Test message for the Final ECE 572 Project.”


x=128; “Test message for the Final ECE 572 Project.”

Figure 10: Image with Encryption and DCT Compression to 35% and Decrypted Message

As you can see, the necessary value of x to recover the message grows as we compress the image to a smaller size with JPEG.

Summary/Discussion/Future Work

Results were much as expected from reading the reference paper. The LSB method was quick, simple, and had little effect on the cover image. However, it could not handle compression. The transform method was robust against compression. However, there is a necessary balance between the value of X and the amount of compression. If X is too large, it will leave artifacts in the image, and it will be obvious that the image is holding a message. If X is too small, the message will be destroyed by compression. The desired value of X is just large enough to withstand the amount compression applied to it. There are several items I think would make interesting future work. One would be to explore the effects of scaling has on the ability to decrypt the message and methods to get any problems caused. Another would be to explore different ways of placing the message in the image in some redundant manner in order to be able to reduce X and still be able to decrypt the message. This project was very interesting. It was very different from the previous projects this semester and forced the student to explore new ideas independently.

```
/*************************************************************************
 * stegMain.cpp
 *
 * Author: Steven Young
 *
 * Created: 12/1/11
 *
 * Modification: This file takes does encryption and decryption.
 ************************************************************************/

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cstdlib>
#include <cmath>             // new: for log()
#include <steg.h>
using namespace std;

#define Usage "./stegMain seed input_img message X\n"

int main(int argc, char **argv)
{
  Image inimg, outimg, mag, phase, maglog;
  int nr, nc;
  int i, j;
  int seedval;
  float x;

  if (argc < 4) {
    cout << Usage;
    exit(3);
  }

  // Read Image
  inimg = readImage(argv[2]);

  // Define Some Parameters
  seedval = atoi(argv[1]);
  x = atof(argv[4]);

  // encrypt lsb
  outimg = encryptLSB(inimg, argv[3], seedval);
  int msg_len = getMsgSz(argv[3]);
  decryptLSB(outimg, msg_len, seedval);

  // write encrypted image
  writeImage(outimg,"../output/encrypted.pgm");

  // write difference image
  outimg = outimg - inimg;
  writeImage(outimg,"../output/diff.pgm", 1);

  // dct of image
  outimg = dct(inimg, 8);
  writeImage(outimg,"../output/dct.pgm", 1);

  // idct image
  outimg = idct(outimg, 8);
  writeImage(outimg,"../output/redone.pgm",1);

  // encrypt with dct transform
  outimg = encryptTransform(inimg, argv[3], seedval, x);
  writeImage(outimg,"../output/transformE.pgm");
  outimg = readImage("../output/transformE.pgm");
  cout << "Decrypting....\n";
  decryptTransform(outimg, msg_len, seedval);

  // writeDiff
  outimg = outimg - inimg;
  writeImage(outimg,"../output/diff2.pgm", 1);

  return 0;
}
```

```
/*************************************************************
 * compressionTests.cpp
 *
 * Author: Steven Young
 *
 * Created: 12/1/11
 *
 * Modification: This file takes an image that has been compressed
 *               and tries to decrypt it using the key provided.
 *************************************************************/

#include "Image.h"
#include "Dip.h"
#include <iostream>
#include <cstdlib>
#include <cmath>             // new: for log()
#include <steg.h>
using namespace std;

#define Usage "./compressionTests seed input_LSB input_DCT message\n" // message is
 needed only to calculate message length
                                                                      // which is o
ne of the keys

int main(int argc, char **argv)
{
  Image inimg, outimg, mag, phase, maglog;
  int nr, nc;
  int i, j;
  int seedval;
  float x;

  if (argc < 2) {
    cout << Usage;
    exit(3);
  }

  // Read Image
  inimg = readImage(argv[2]);
  seedval = atoi(argv[1]);

  // decrypt lsb
  int msg_len = getMsgSz(argv[4]);
  decryptLSB(inimg, msg_len, seedval);

  // decrypt dct
  inimg = readImage(argv[3]);
  decryptTransform(inimg, msg_len, seedval);

  return 0;
}
```

```
/*************************************************************************
 * steg.cpp
 *
 * Author: Steven Young
 *
 * Created: 12/1/11
 *
 * Modification: Utilities for the final project.
 ************************************************************************/

#include <steg.h>


void decryptTransform(Image inimg, int msg_len, int seedval){

  Image outimg, dctimg;
  int img_len;
  int nr, nc;
  int block_x, block_y;
  int c_pos, mask;
  char c_char = -1;
  int i, bit_set, idx;
  int* array;
  int u1, v1, u2, v2;

  int form_char;
  char* r_char;

  // define what indices to compare
  u1 = 4;
  v1 = 1;
  u2 = 3;
  v2 = 2;

  // get dct of image
  dctimg = dct(inimg, 8);

  // number of rows and cols of blocks
  nr = dctimg.getRow()/8;
  nc = dctimg.getCol()/8;

  img_len = (nr * nc);

  //  use key to create random sequence
  array = new int[msg_len];
  steg_randseq(seedval, msg_len, img_len, array);

  outimg = dctimg;
  for(i=0;i<msg_len;i++){
    c_pos = i % 8;
    if(c_pos == 0){
      c_char++; // next char
      form_char = 0;
    }

    // locate next block to use
    idx = array[i];
    block_x = (idx / (nr)) * 8;
    block_y = (idx % (nr)) * 8;

    // see if bit is set
    bit_set = (((int)(outimg(block_x+u1,block_y+v1)>outimg(block_x+u2,block_y+v2)))
);

    // add bit to character representation
    form_char = form_char | (bit_set << (7-c_pos));

    // print character
    if(c_pos == 7){
      if (form_char == 0) break;

      cout << ((char)('\0' + form_char));
    }
  }
  cout << endl;
}

Image encryptTransform(Image inimg, char* msg, int seedval, float x){
  Image outimg, dctimg;
  int msg_len, img_len;
  int nr, nc;
  int block_x, block_y;
  int c_pos, mask;
  char c_char=-1;
  int i, bit_set, idx;
  int* array;
  int u1, v1, u2, v2;

  // define the coefficients to swap
  u1 = 4;
  v1 = 1;
  u2 = 3;
  v2 = 2;

  // get dct of image
  dctimg = dct(inimg, 8);

  // number of rows and cols of blocks
  nr = dctimg.getRow() / 8;
  nc = dctimg.getCol() / 8;

  // calculate message length and image length
  msg_len = getMsgSz(msg);
  img_len = (nr * nc);

  // generate random sequence using key
  array = new int[msg_len];
  steg_randseq(seedval, msg_len, img_len, array);

  // encrypt message into dct image
  outimg = dctimg;
  for(i=0;i<msg_len;i++){
    c_pos = i % 8;
    if(c_pos == 0){
      c_char++; // next char
      mask = 128;
    }
    bit_set = (bool)((mask & msg[c_char]) >> (7 - c_pos));
    mask = mask >> 1;


    idx = array[i];
    block_x = (idx / (nr)) * 8;
    block_y = (idx % (nr)) * 8;

    // code message based on bit value
    if (bit_set){
      outimg(block_x+u1,block_y+v1) = max(dctimg(block_x+u1,block_y+v1),dctimg(bloc
k_x+u2,block_y+v2));
      outimg(block_x+u2,block_y+v2) = min(dctimg(block_x+u1,block_y+v1),dctimg(bloc
k_x+u2,block_y+v2));
      float diff = outimg(block_x+u1,block_y+v1)-outimg(block_x+u2,block_y+v2);
      if(diff < x){ // ensure difference between coefficients
        float spread = (x - diff)/2;
```

```
        outimg(block_x+u1,block_y+v1) = outimg(block_x+u1,block_y+v1) + spread;
        outimg(block_x+u2,block_y+v2) = outimg(block_x+u2,block_y+v2) - spread;
      }
    }else{
      outimg(block_x+u1,block_y+v1) = min(dctimg(block_x+u1,block_y+v1),dctimg(bloc
k_x+u2,block_y+v2));
      outimg(block_x+u2,block_y+v2) = max(dctimg(block_x+u1,block_y+v1),dctimg(bloc
k_x+u2,block_y+v2));
      float diff = outimg(block_x+u2,block_y+v2)-outimg(block_x+u1,block_y+v1);
      if(diff < x){ // ensure difference between coefficients
        float spread = (x - diff)/2;
        outimg(block_x+u1,block_y+v1) = outimg(block_x+u1,block_y+v1) - spread;
        outimg(block_x+u2,block_y+v2) = outimg(block_x+u2,block_y+v2) + spread;
      }
    }
  }

  // produce image by taking the inverse dct
  outimg = idct(outimg, 8);

  return outimg;
}

Image idct(Image S, int blk_sz){

  Image outimg;
  int nr, nc;
  int nb_r, nb_c;
  int x, y;
  int u, v;
  int r_off, c_off;
  float c_u, c_v;
  int i, j;

  nr = S.getRow();
  nc = S.getCol();

  outimg.createImage(nr,nc);

  // calculate number of rows and cols of blocks in image
  nb_r = (nr / blk_sz) + ((nr % blk_sz) > 0);
  nb_c = (nc / blk_sz) + ((nc % blk_sz) > 0);

  // do inverse dct
  for (i=0;i<nb_r;i++){
    for(j=0;j<nb_c;j++){

      r_off = i * blk_sz;
      c_off = j * blk_sz;
      for(x=0;x<blk_sz;x++){
        for(y=0;y<blk_sz;y++){
          if(u == 0){
            c_u = 1/sqrt(2);
          }else{
            c_u = 1;
          }
          if(v == 0){
            c_v = 1/sqrt(2);
          }else{
            c_v = 1;
          }
          outimg(x+r_off,y+c_off) = 0.0;
          for(u=0;u<blk_sz;u++){
            for(v=0;v<blk_sz;v++){
              if(u == 0){
                c_u = 1/sqrt(2);
              }else{
                c_u = 1;
              }
              if(v == 0){
                c_v = 1/sqrt(2);
              }else{
                c_v = 1;
              }
              outimg(x+r_off,y+c_off) = outimg(x+r_off,y+c_off) + c_u * c_v * S(u+r
_off,v+c_off)*cos(u*PI*(2*x+1)/(2*blk_sz))*cos(v*PI*(2*y+1)/(2*blk_sz));
            }
          }
          outimg(x+r_off,y+c_off) = outimg(x+r_off,y+c_off) * 2 / blk_sz;
        }
      }
    }
  }
  return outimg;
}

Image dct(Image s, int blk_sz){

  Image outimg;
  int nr, nc;
  int nb_r, nb_c;
  int x, y;
  int u, v;
  int r_off, c_off;
  float c_u, c_v;
  int i, j;

  nr = s.getRow();
  nc = s.getCol();

  outimg.createImage(nr,nc);

  // calculate number of rows and cols of blocks in image
  nb_r = (nr / blk_sz) + ((nr % blk_sz) > 0);
  nb_c = (nc / blk_sz) + ((nc % blk_sz) > 0);

  // do dct
  for (i=0;i<nb_r;i++){
    for(j=0;j<nb_c;j++){
      r_off = i * blk_sz;
      c_off = j * blk_sz;
      for(u=0;u<blk_sz;u++){
        for(v=0;v<blk_sz;v++){
          if(u == 0){
            c_u = 1/sqrt(2);
          }else{
            c_u = 1;
          }
          if(v == 0){
            c_v = 1/sqrt(2);
          }else{
            c_v = 1;
          }
          outimg(u+r_off,v+c_off) = 0.0;
          for(x=0;x<blk_sz;x++){
            for(y=0;y<blk_sz;y++){
              if((x < nr) && (y < nc)){ // pad zeros otherwise
                outimg(u+r_off,v+c_off) = outimg(u+r_off,v+c_off) + s(x+r_off,y+c_o
ff)*cos(u*PI*(2*x+1)/(2*blk_sz))*cos(v*PI*(2*y+1)/(2*blk_sz));
              }
            }
          }
          outimg(u+r_off,v+c_off) = outimg(u+r_off,v+c_off) * 2 / blk_sz * c_u * c_
v;
```

```
        }
      }
    }
  }
  return outimg;
}


int getMsgSz(char* msg){
  int msg_len = 0;

  while(msg[msg_len++] != '\0'); // count number of characters

  msg_len = msg_len * 8; // calculate number of bits

  return msg_len;
}

void decryptLSB(Image inimg, int msg_len, int seedval){

  int nr, nc, img_len;
  int i;
  int* array;
  int c_char = -1;
  int mask;
  int bit_set;
  int c_pos;
  int idx, x, y;
  int form_char;
  char* r_char;


  nr = inimg.getRow();
  nc = inimg.getCol();

  img_len = nr*nc;

  // generate random sequence from key
  array = new int[msg_len];
  steg_randseq(seedval, msg_len, img_len, array);


  // do lsb decryption
  for(i=0;i<msg_len;i++){
    c_pos = i % 8;
    if(c_pos == 0){
      form_char = 0;
      c_char++; // next char
    }

    idx = array[i];
    x = idx / nr;
    y = idx % nr;

    bit_set = (((int)inimg(x,y)) % 2);
    form_char = form_char | (bit_set << (7-c_pos));

    if(c_pos == 7){
      if (form_char == 0) break;
      cout << ((char)('\0' + form_char));
    }
  }

  cout << endl;

  delete array;
}


Image encryptLSB(Image inimg, char* msg, int seedval){

  int nr, nc, img_len, msg_len;
  int i;
  int* array;
  int c_char = -1;
  int mask;
  bool bit_set;
  int c_pos;
  Image outimg;
  int idx, x, y;

  nr = inimg.getRow();
  nc = inimg.getCol();
  outimg.createImage(nr,nc);

  msg_len = getMsgSz(msg);

  img_len = nr*nc;

  // calculate random sequence from key
  array = new int[msg_len];
  steg_randseq(seedval, msg_len, img_len, array);

  // do lsb encryption
  outimg = inimg;
  for(i=0;i<msg_len;i++){
    c_pos = i % 8;
    if(c_pos == 0){
      c_char++; // next char
      mask = 128;
    }
    bit_set = (bool)((mask & msg[c_char]) >> (7 - c_pos));
    mask = mask >> 1;


    idx = array[i];
    x = idx / nr;
    y = idx % nr;
    if (bit_set){
      outimg(x,y) = inimg(x,y) + (1 - (((int)inimg(x,y)) % 2)); // make sure LSB is
 set;
    }else{
      outimg(x,y) = inimg(x,y) - ((((int)inimg(x,y)) % 2)); // make sure LSB is not
 set;
    }
  }

  delete array;
  return outimg;
}

void disp_seq(int msg_len, int* array){
  int i;

  cout << endl;
  for(i=0;i<msg_len;i++){
    if ((i % 5) == 0) cout << endl;
    cout << array[i] << " , ";
  }
}

void steg_randseq(int steg_seed, int msg_len, int img_len, int* array){
  int max_int;
  int i;
  int prev_ind = 0;
```

```
  int msg_left;
  int img_left;

  srand(steg_seed);

  if (msg_len > img_len){
    cout << "Message cannot fit in image!\n";
    exit(2);
  }

  msg_left = msg_len;
  img_left = img_len;
  for(int i=0; i<msg_len; i++){
    max_int = img_left/msg_left;
    array[i] = prev_ind + steg_randint(max_int);
    prev_ind = array[i];
    msg_left--;
    img_left = img_len - prev_ind;
  }
}


int steg_randint(int max_int){
  return rand() % max_int + 1;
}
```

```
/*************************************************************************
 * steg.h
 *
 * Author: Steven Young
 *
 * Created: 12/1/11
 *
 * Modification: Utilities for the final project.
 ************************************************************************/

#ifndef STEG
#define STEG

#include "Dip.h"
#include "Image.h"
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <math.h>

using namespace std;

void decryptTransform(Image inimg, int msg_len, int seedval);
Image encryptTransform(Image inimg, char* msg, int seedval, float x);
Image idct(Image S, int blk_sz);
Image dct(Image s, int blk_sz);
void decryptLSB(Image inimg, int msg_len, int seedval);
Image encryptLSB(Image inimg, char* msg, int seedval);
int getMsgSz(char* msg);
void disp_seq(int msg_len, int* array);
void steg_randseq(int steg_seed, int msg_len, int img_len, int* array);
int steg_randint(int max_int);

#endif
```