**Steganography Technique Based On Block-DCT And Huffman Encoding**



Team 11 CS4953

Zachary Salazar | Marvin Lopez | Marcus Lorenzana

**Introduction**

For our project we will be using a robust and secure Block-DCT (Discrete Cosine Transform) steganography technique with Huffman coding to embed and decode a secret message into an image. The image will be an 8-bit Grayscale/Binary Image of any size that will be divided into 8x8 blocks. In the block diagram, it shows the necessary steps and methods/algorithms that are needed to hide the image or message. DCT is applied to the blocks and then Huffman encoding is applied to the message and image before the embedding process. Huffman coding program will also handle the decoding of the extracted message when it is received from the stego image. To embed the image or message we are going to do the following five steps:

Step 1: Divide the carrier image into non overlapping blocks of size 8×8 and apply DCT on each of the blocks of the cover image to obtain F using eqn (1).

Step 2: Perform Huffman encoding on the 2-D secret image S of size M2 × N2 to convert it into a 1-D bits stream H.

Step 3: .Huffman code H is decomposed into 8-bits blocks B.

Step 4: The least significant bit of all of the DCT coefficients inside 8×8 block is changed to a bit taken from each 8 bit block B from left to right. The method is as follows:

For k=1 ; k<1; k=k+1

LSB( ( F(u,v))2 ) <- B(k) ;

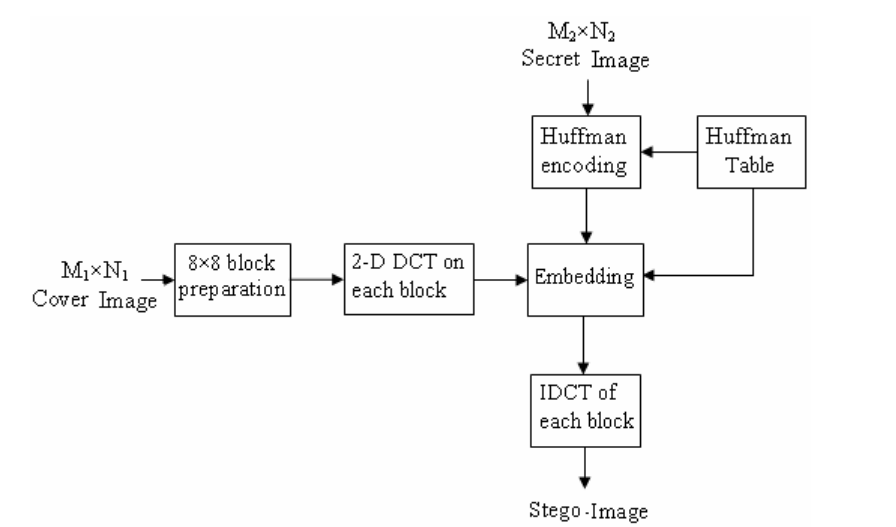
Where B(k) is the kth bit from left to right of a block B and (F(u,v)2 ) is the DCT coefficient in binary form.

Step 5: Perform the inverse block DCT on F and obtain a new image cover image which contains secret image.

Our main goal is to first have the program to run properly by hiding the image or message. As this is the first type of assignment like this we have had as a group, it will be a challenge. The next goal for this project is to implement the all the methods/algorithms to make a secure and robust steganographic information hiding technique. Finally, if we have more time we intend on making better ways of implementing the image as well as a different image that best fits with our implement to show less distortion of the original image.

**Functional Block Diagram**

**Insertion:**



**Extraction:**

**Discussion**

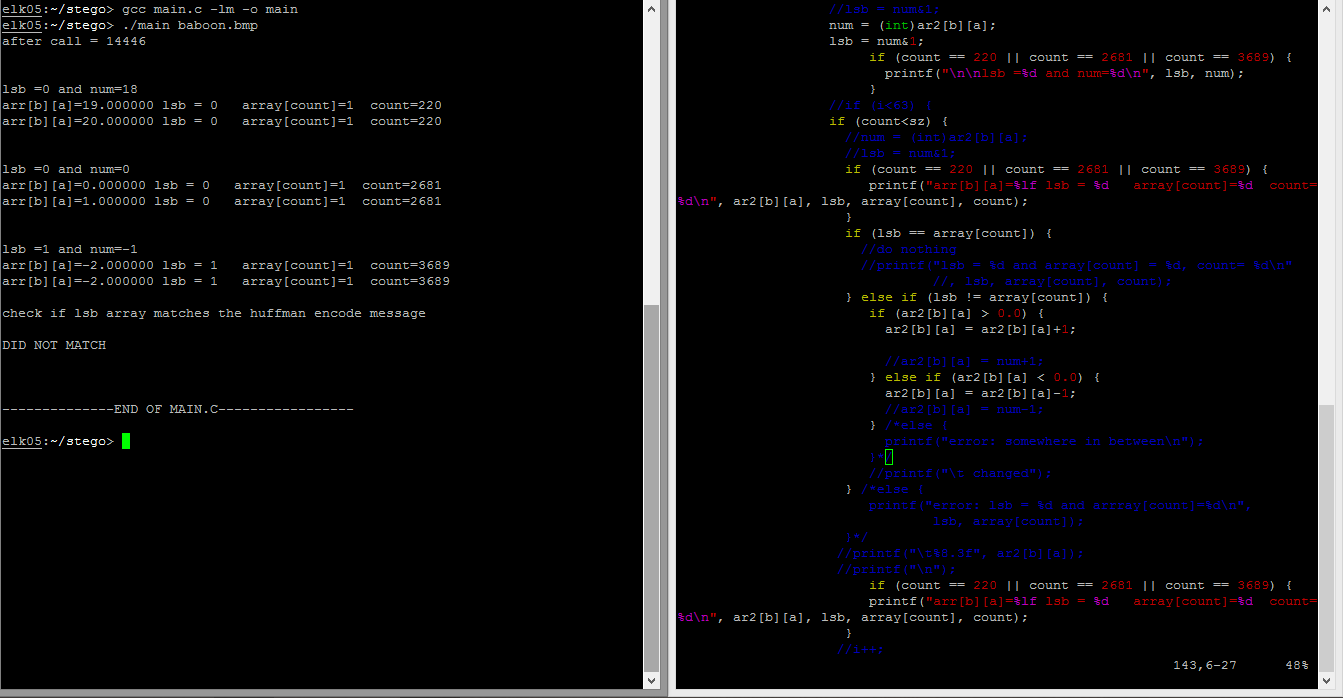
In are program we made three functions to manipulate the array of 8x8 pixels the first is Compute8x8Dct(double in[][],double out[][]). This function gets the 2-d array in[][] and applies DCT and puts the coefficients in the out[][] array so we can manipulate later on. Compute8x8Idct(double in[][], double out[][]) This formula reverses the DCT to generate the pixels into a 8x8 array so we can generate a new image. The last function is print8x8(char \*title, double in[][]) so we can see the arrays and make sure the functions are working , and to see what is the output of the two previous functions.

For our project we utilized huffman encoding technique to create a bit stream that will be used to replace the least significant bits of each DCT coefficient. Overview: Huffman encoding is a tree data structure that compresses text by assigning byte codes to each letter. If the character is used frequently it will have a shorter byte code, and vice versa for character that is rarely used. Our Huffman.c program uses a struct that contains the character, the frequency, and the left and right nodes. Overview of functions:

* new\_node : create a new node and assign the frequency and character to it, initializing the next nodes.
* q\_insert: a priority queue that will sort the frequency of the letters from most used to least used.
* q\_remove: remove a node from the tree and fix if necessary
* build\_code: walk through the tree and generate the 0’s and 1’s for the correspoding left and right nodes.
* init: initialize the character string into the huffman tree and get the huffman table
* encode: walk through the tree and get the binary value for each letter
* decode: use the huffman table to identify the bit stream and spit out the original message
* main: in our main we first search for the file “big.txt” as read only and get the message to be converted. You could alternatively use a simple char array in the main function to pass into encode. Use fseek to read in the file and assign it to a character string.
* Initialize the string into a huffman table and tree.
* Pass the string into encode to get the binary stream
* Get the bit length of the stream create a fixed 1-D integer array.
* Output the stream into a text file to be used for embedding process, which will later be fed into getArr that will convert the stream into a 1-D integer array.
* After the embedding process and extraction process, pass the bit stream back into huffman.c for decoding to get the original character message back.

The huffman encoding was some what difficult to implement for large amounts of data, but was not able to implement due to the problem of casting. We were able to get huffman code for semi-large data like 8 long paragraphs but when we tried adding say 100 paragraphs is when it didn’t work as intended. At first we thought it was because the buffer size was too small but the buffer fit the number of characters. We also would be able to huffman encode an image by changing the image to a string in order for it to be compatible since we encode a string. The discrete cosine transform algorithm worked well for us. We were able to apply the algorithm get the DCT-coefficients, make changes to the values to match our bit stream (in most cases), apply the inverse discrete cosine transform algorithm, take the values and add to the output image(stego-image), and then apply discrete cosine transform algorithm to get the bitstream.

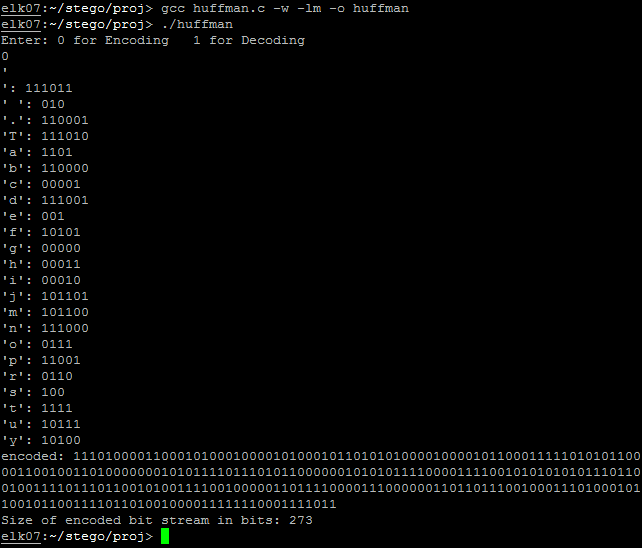
We had most of what we planned to do for the project, but our biggest problem was the casting problem. If we had overcome the casting problem we would have been able to hide large amounts of data as well as an image in our cover image. In the following picture it shows the problem with the casting:



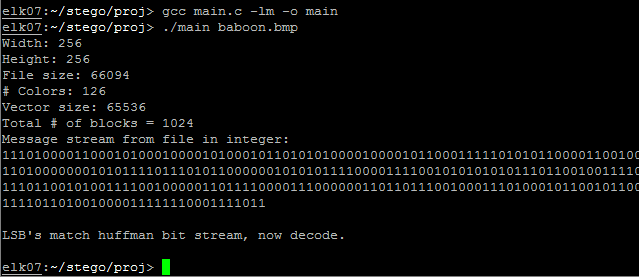
In the picture, you can see that when casting a double to an int the double is really 19.000 but is 18 as a int. So when grabbing the lsb and comparing to the huffman encoded bit stream it does not come out to the proper bit stream. This was the most frustrating part of our project, because if we overcome this small piece of adversity we would have a lot more data to work with. Overall, we got done a lot of what we wanted to get done. Even though it is not working with large data, we are proud of what we accomplished.

**Example of the program**

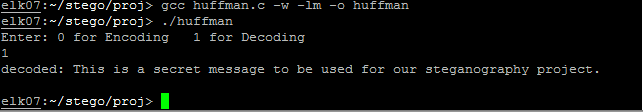
First, we run the huffman.c program to get the bit stream of the secret message by opening the file msg.txt to get the string and huffman encode it. The program prompts the user to determine what they want to do by asking whether or not they want to encode by entering 0 or decode by entering 1. After the program runs we get the following output, which include the huffman table that we will use to decode the bitstream in the last step:



Next, we run the main.c program that includes the DCT.c program in the header. This program main.c reads in the bitmap information, copies the header information and is added to the output StegoImage.bmp along with the altered values. The output of the program is the header info, the bit stream message that we will embed in the DCT-coefficients, and whether or not the bit stream and lsbs extracted were the same. If they are the same then you are prompted to run the huffman program to decode the message. If they do not match, then it would display the non matching pared values.



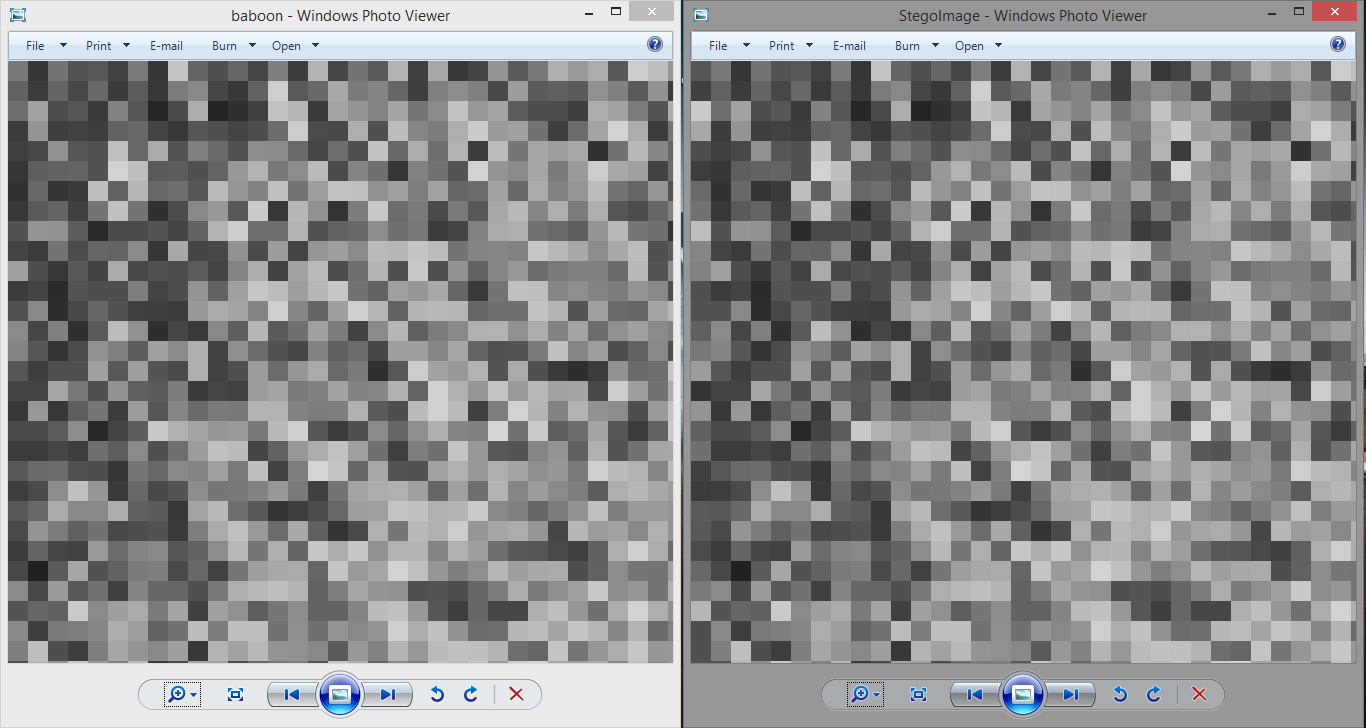
Finally, after the running huffman.c and main.c for the first time you can now decode the lsbstream.txt file by selecting 1 when prompted and the output should be the same message as was entered in the msg.txt.



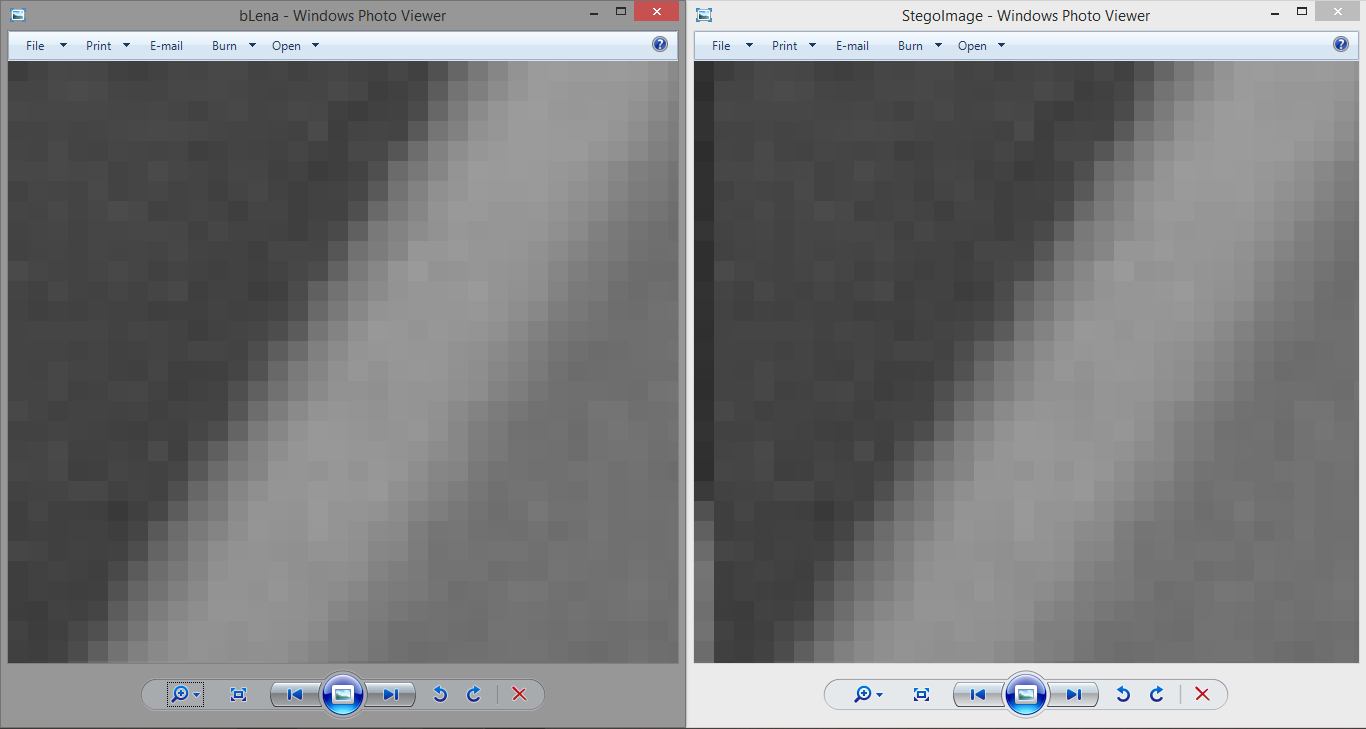
**Statistical Analysis Results**

The program we have now is to run the program that has a maximum of 63 bytes in each 8x8 block. We have to skip the first element because if we alter it it makes the inverse values of the DCT-coefficients change drastically. The method of hiding with these steps has improve security and better image quality. If we had able to hide big data, we would have to adjust out code to move one to the second least significant bit after all the first least significant bits were used for hiding. So for example if the image is 256x256, you would be able to use 64,512 bytes. Then have to move to the next least significant bit for the remaining information and continue until all information is embedded into the cover image. In the article that we based our project they show that their capacity was 299520 bits which in theory we would have liked to do.

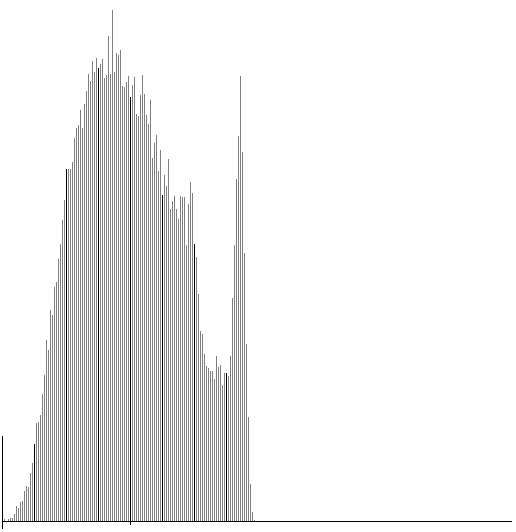
We tested our program with a couple images; baboon.bmp which is 256x256 and bLena.bmp which is 640x640. We choose these images to test that the program would work with different size images with legal dimensions. Since we only have the program working with small amounts of data we had to zoom all the way into the image to show that pixels change and the new information has been added. The following image shows the differences from the images in the top-left pixels for the baboon.bmp.



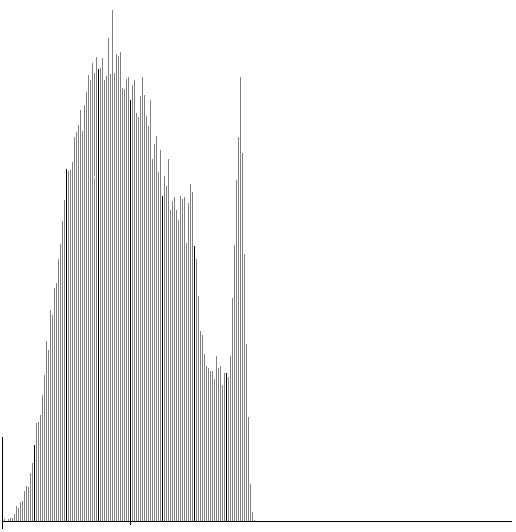
The following image shows the differences from the images in the top-left pixels for the bLena.bmp.



We ran the histogram program on the baboon.bmp, but since its a small amount of data in the file there is visually no change. We couldn’t run the DCT-coefficients histogram program because it was for jpeg file. But even if we were to run it there would be small changes to the coefficients as well. The following is the histograms:



baboon.bmp



StegoImage.bmp

**Suggestions for future work**

* Be able to hide an image in our cover image, which would have been able to do if we had large data working.
* Have the hidden information hide in permuted places in the 8x8 block as well as in different blocks.

**Bibliography**

* Nag, A., Biswas, S., Sarkar, D., & Sarkar, P. P. (0210, June 1). A novel technique for image steganography based on Block-DCT and Huffman Encoding. . Retrieved June 30, 2014, from http://arxiv.org/ftp/arxiv/papers/1006/1006.1186.pdf
* Yang, C., & Wang, S. (2010, January 1). Transforming LSB Substitution for Image-based Steganography in Matching Algorithms. . Retrieved July 2, 2014, from http://www.iis.sinica.edu.tw/page/jise/2010/201007\_03.pdf
* Hamdy, S., El-Messiry, H., Roushdy, M., & Kahlifa, E. Retrieval of Bitmap Compression History. (IJCSIS) International Journal of Computer Science and Information Security, 8, 6. Retrieved July 10, 2014
* Huffman Coding." *Huffman Cod*. Rosettacode, n.d. Web. <<http://rosettacode.org/wiki/Huffman_coding>>.
* "Discrete Cosine Transform." *Wikipedia*. Wikimedia Foundation, 22 July 2014. Web. 04 Aug. 2014. <<http://en.wikipedia.org/wiki/Discrete_cosine_transform>>
* Green, Bill. "Raster Data Tutorial." *Raster Data Tutorial*. N.p., 2002. Web. 04 Aug. 2014. <<http://dasl.mem.drexel.edu/alumni/bGreen/www.pages.drexel.edu/_weg22/raster.html>>
* Trying to implement The inverse of a dct 8\*8 matrix. (n.d.). *c*. Retrieved July 4, 2014, from http://stackoverflow.com/questions/8553717/trying-to-implement-the-inverse-of-a-dct-88-matrix/8553966#8553966