Avery Anderson
4/22/2018

Final Project: Steganography

In this project I sought to create a set of programs that would allow me to attach secret messages to photos in such a way that an observer wouldn't know there's anything weird about the image. I then wanted to be able to decode the image, reconstructing the original message. In researching this project I found that it had been largely already done by a set of students in the University of Wisconsin-Madison. Their website can be found here.
https://sites.google.com/site/cs534steganographyproject/home

They were a large inspiration during this project, as their system is where I got the idea to store the message in the Least Significant Bits of the image. My final system can encode or decode .txt and .wav files into a .jpg, or .png file. The encoded image is returned by the function, or it can be saved as a bitmap (.bmp) file. Any other type of image format compresses the image, and destroys the message. After encryption, messages can be decrypted and either displayed at the terminal, or saved to the project directory.

Phase One: *Encoding text messages:*

The first hurdle that had to be overcome for this project was defining a system to encode a basic text message into an image. This was done in three major stages. First the program verifies the validity of the inputs, and imports them. Providing us with a row vector of characters for the message, and a standard RGB image. It then transforms the 1xn character vector into an nx8 matrix where the nth row is the binary representation of the nth element in our character array. Now that we have the message in an encodable format, we iterate through the image matrix, and overwrite the LSB of each pixel to instead be a pixel of our image matrix and save it.

When we transfer these messages, I figured it would also be nice to preserve the name of our original message file. As this is just another array of characters, this functionality imply involved adding the name of the file at the front of our message. In the code this is done by vertically concatenating the binary representation of the message.

The actual encoding process is relatively simple. I calculate the total number of bits I need to encode, and iterate from across each byte directly mapping the bit to the image. The only complicated part of the process comes from indexing. I trivially index the image matrix (I call image(k) for k = 1:bitsToEncode) which means if I was working with the original image I would first write to only red pixels, then only green and then only blue pixels. Meaning for relatively short messages only one color will be altered by the program. I figured that it makes more sense to instead work through a single pixel altering every RGB value, and only then moving to a neighboring pixel. In order to accomplish this I permute the image array immediately before and after the encoding process to make the RGB index the first dimension of the system.

I also had to get creative with indexing the binary message. I knew that every consecutive row marked a consecutive character, and I had to move through a set number of bits within each row before I could start the next row. This relationship between index within row and the row index is that of a modulus. So I index the message vector by taking the quotient of my index divided by 8 as the row index, and the remainder as my index within the row. It was actually only in writing this paragraph that I now realized that it would be much simpler for the program to instead take the transpose of the message matrix and index the message in the exact same way that I index the image matrix. This change has now been reflected in the code.

Phase Two: *Decoding text messages:*

Once I had created a system to hide the messages I needed to create an inverse function that would be able to reconstruct the message. The first two issues that I wanted to resolve were those of recognizing that there was a message in the image, and only decoding the necessary frames in order to reduce decoding time. My initial solution was to simply designate the first pixel of the image as a a length pixel. Completely overwriting the data of the square. With an mxn message matrix, I was going to store m in image(1), n in image(2) and image(3) was going to be the sampling rate when I moved on to encoding audio files. However, upon implementing this method there were two key issues. First this method was not discrete, the first pixel was quite obviously overwritten. The second issue was far more important. As each frame of image is only a uint8, the maximum value it could store was 2^8 - 1, or 255. Meaning I would never be able to decode a text message of more than 255 characters. While I do love tweeting, I was hoping to get more usability out of my system.

My solution to this problem was again inspired by the students of the University of Wisconsin-Madison. I modified the encryption code to include a section where I took a count of the amount of rows I needed to encode, turned that number into a string, and added it to my encoded binary message. Separating it from the name of the file with a period, as file names by convention cannot have more than one period.

Length of Message
‘.’
Message

This process also provided me with a very simple way to verify that the image contained a message. I assumed that I will never try to encode more than 1Mbyte of information. With this assumption I know any message that I encode ought to have a period stored in the file within the first seven characters that I would decode. So I only allow the program to decode seven bytes

before throwing an error stating that no message was found. The decryption process is exactly the same as the encryption, except this time rather than assigning a value to the image, we assign a value to the message.

Once a message has been fully read, we need to appropriately format it. For a text message this simply means converting the binary back to a number, and that number to its representative characters. The program tells the user the name of the file, and asks them if they would like to save the message, see it just once in the terminal, or do both. At which point the program ends. Note once a portion of the message has been processed, such as the length, the message is resized to remove the processed indices, allowing the later sections to work more directly with the information.

Phase Three: *Working with Wav Files*

Once I had the the encryption decryption system set up, I figured the transition to audio files would be trivial, but I had forgotten that the primary vector for an audio file is made of floating point numbers, so the bitwise interpretations were far less intuitive. My initial attempts to work with the floats was to simply cast the doubles to be of type uint64. As I figured if I never directly alter the numbers I should be able to easily reverse the cast and regain my signal. However, I quickly ran into an issues with this approach as doubles cannot have more than 53 bits, so I couldn't simply cast the numbers back. When I tried to truncate the numbers so they'd be the appropriate size, I would get floats out, but they had completely different values.

When I tried to find a solution to this problem I found that people seemed to be running into it all over the place. There are tons of discussion boards that all claim it's impossible to get the binary numbers over 53 bits, or it is at least too much effort and they should just change how they are doing their project. However, halfway down one such message board I found the following link to the functions another user had made to resolve this issue. The float2bin function is used in my encryption code, and the bin2float in the decryption. You can find these functions again in the directory "Third party programs"." They are also copy pasted into the bottom of the files.
https://www.mathworks.com/matlabcentral/fileexchange/39113-floating-point-number-conversion

Using these functions I was able to begin processing and encoding the audio messages into the images. The output of these float to binary functions is 64 bits. In order to make the audio information mesh with the code I had previously written in for the text files. I needed to reshape the 64 bit numbers into sets of 8 bytes. Once I had fully done this reshaping, I could encode and decode the message in the exact same way I had done with the text message.

The only other addition to the code I needed to make in order to implement audio files was the passing of the sampling rate. I resolved this issue in the exact same way I resolved the rows issue previously. I added the values as a group of chars immediately after the name of the file. Again separated from the message by a period.

Moving Forward:

There are many directions in which this project could continue to expand. The most readily available next steps would be begin encoding other types of files such as images or videos. Now that I have implemented encoding for floating point numbers as well as fixed points, I think I will probably have encountered a majority of issues I'd face while attempting other formats. Though there would probably be additional issues of weird concatenations and needing to reshape matrices, but I think it would be very doable to add other formats, just tedious.

While I did succeed in hiding audio files in the images, my current method of encryption is very memory inefficient, as every single sample point needs 8*8/3 or 21.3 pixels in the image. This is simply not scalable. I would like to create a system where I could hide messages of more than a couple seconds without needing the world's largest image.

Lastly it would be pretty exciting to explore better paths of encryption. Looking again to the students from the University of Wisconsin-Madison, we can see that there are a lot of issues with the method of "sequential encoding" that I used in this project. They resolved this issue by using a set of random indices, and I'm sure there are a lot of cool and fun ways to implement similar situations.

Running the Code:

A full explanation of how to run the code can be found in the READ_AND_ENCODE_ME.txt file.

Other Resources:

In this project I also encoded a novel into the code. I found the text that I used for the "Hidden_Text.bmp" example at the following website. There were a couple percent signs and dashes I needed to delete in order get the text to print, so I'm pretty sure this is a very illegitimate copy of the book. But its purpose was served in the example.
https://archive.org/stream/warandpeace030164mbp/warandpeace030164mbp_djvu.txt