DSP Final Project Team 12
Image Encrypting and Decrypting of JPEG

- Members: 106061101 曾泓硯, 106061209 廖宏儒

GitHub: https://github.com/hankliao87/10720EE366000

I. Identify question and Application

In this project, we aim to construct a compressor, which could both compress the capacity of file and encrypt the image with arbitrary key.

We refer to the procedure of JPEG to implement our compressor and adjust a few steps to encrypt the image. A successfully encrypted image could hide our real image perfectly. Whether the hacker believes the authenticity of image depends on some factors, color, resolution, noise, etc. We will discuss the rate of compression and the effect of encryption in this project. We use MATLAB for research and python for implementation.

In this document, we call the real image that we want to send as origin image, the image we used to encrypt as key image.

II. Problem Analysis

We can separate the problem into two parts, compression and encryption.

For compression, the balance between compression rate and loss of information is our major problem. The error may mostly happen during quantization and dequantization, since rounding function may discard lots of data.

As for encryption, there are two main problem. First, the ability to hide the image. The user can select arbitrary image as key to encrypt origin image. However, the addition in each 8*8 matrix results in the overlapping both images. Therefore, in order to hide image perfectly, we should increase the weight of key image. Sup-Figure-1 (origin image), Sup-Figure-2 (key image), Sup-Figure-3 (key image*0.9+origin image*0.1 overlapping), Sup-Figure-4 (key image*0.5+origin image*0.5 overlapping). We can observe that Sup-Figure-3 hides the image better than Sup-Figure-4.

Second, the ability to recover the image. The key image has higher weight than real image has, this may result in severe error during quantization. Sup-Figure-1 (origin image), Sup-Figure-5 (key image * 0.9 + origin image * 0.1), Sup-Figure-6 (key image * 0.5 + origin image * 0.5). As we can see, Sup-Figure-6 looks more clearly and similar to Sup-Figure-1 than Sup-Figure-5 does.

Besides, we are also curious about the effect of noise, different scanning method and quantization or not. The hardest part in this project is Huffman coding, the implementation of Huffman coding includes forward merging and backward splitting. Huffman coding approximate the average length of code to theoretical value. Besides, the loss of data also result in noise in recover image.

III. Implementation

A. The procedure of origin JPEG includes:

RGB to YUV → Downsampling to 8*8 matrix → DCT → Quantization → zigzag scan → Run length encode → Hoffman coding.

1. DCT:

As we know, Fourier transform has imaginary part. The information doubles after Fourier transform, which conflicts with compression. Therefore, we use DCT, which preserves the real part of Fourier transform.

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right] \qquad k = 0, \ldots, N-1.$$

2. Quantization:

Most of information are lost during this procedure, we divide the DCT 8*8 matrix with quantization matrix (Sup-Figure-7). The high frequency part in 8*8 matrix after DCT concentrated at larger index. Since naked eyes all less sensitive to high frequency images, we decide to abandon high frequency part by dividing larger integer.

3. Zigzag Scan:

Zigzag scan rearranges the coefficient from low frequency to high frequency respectively. The first term in matrix is DC term, while others are AC terms. (Sup-Figure-8)

4. Run Length Encode:

We will encode DC term and AC terms respectively. For DC term, we should first apply DPCM, but we skip this part.

For AC terms, we will encode the zigzag sequence. For each non-zero term, we use (num, size, value) to represent it. Num represents the number of zeros before non-zero term, size represents the size of bit needed for value, and value represents the actual value of non-zero term. If we find out the rest of terms are zeros, we will use (0,0,0) to represent it. In Sup-Figure-9 we will get, (0,3),(5),(0,3),(-4),(1,2),(2),(3,1),(-1),(0,0) as code.

5.Huffman coding:

Before we construct the heap tree, we should get the frequency table first, the RL code that appears most frequently should have the shortest Huffman code, vice versa. We will construct a Huffman table to store the code each RL code represents, and transfer the whole RL code into binary file. (Sup-Figure-10 is the heap tree, Sup-Figure-11 is the Huffman table).

B. The procedure to decode origin JPEG includes:
Transfer Hoffman coding → Run length decode → inverse zigzag scan → Dequantization → IDCT → YUV to RGB
We inverse the whole procedure in the part A.

C. Some factor that affects encryption:
1. Overlapping the origin image with key image:
We intend to add DCT of key image with different weight to the origin image.

2. Without quantization

Since quantization will lose a lot of data, we come up with the idea that we can simply skip this part.

3. Add noise

We are curious about the effect of noise. In this project, we will use random function with fixed seed to generate random matrix for each 8*8 matrix.

4. Different scanning method

The origin JPEG scan the 8*8 matrix by zigzagging, from (Sup-Figure-12) we can observe that the same column also shares similar pattern. We decide to scan each column instead of zigzagging.

Our final procedure to encrypt:
RGB to YUV → Downsampling to 8*8 matrix → Add DCT of origin & key image → Add noise → zigzag scan → Run length encode → Hoffman coding.

Our final procedure to decrypt:
Transfer Hoffman coding → Run length decode → inverse zigzag scan → remove noise → remove key image → IDCT → YUV to RGB

IV. Result

We will discuss two parts, compression rate and recovery & encryption effect.

1. JPEG (our version)

We can see from Sup-Figure-13, there is a subtle green line. It is because the dimension of origin image is 732*1072, it can't be divided by 8*8 matrix completely; therefore, there is a green line. Overall, the image recovers well.

2. Overlapping the origin image with key image

As we can see, Sup-Figure-6 (0.5 origin+0.5 key) looks more clearly than Sup-Figure-5 (0.1 origin+0.9 key). We think it comes from quantization error, in Sup-Figure-5 most of the value comes from key image, the origin image is likely to lose during rounding. Therefore, if we want to hide the image perfectly, we have to in exchange of the quality of recover image.

3. Without quantization

Since the origin JPEG already has quantization error, hiding the origin image will deteriorate the error; therefore, we come up with the idea not to quantize it. Compare Sup-Figure-5 and Sup-Figure-14, we can see clearly that Sup-Figure-14 recovers the image perfectly. The reason that our quantization matrix end with 99 is because we hope all 8*8 matrix ends with 0, therefore, we can use JPEG to decompress it.

4. Add noise

We are curious about the effect of noise added to the image. In order to recover the image back, we need to store the noise matrix. Sup-Figure-15 is the origin matrix, Sup-Figure-16 is the recover image. As we can see, the effect of protecting the image works well in image comprised of similar color. What if we want to get clear image back? We can simply skip the step of quantization. Sup-Figure-17 is the result without quantization.

5. Different scanning method

We scan the matrix by each column, we can expect that the image won't be clear. From Sup-Figure-18, we are surprise that we can't even see the origin image using origin decompressor of JPEG.

Since we refer to the method of JPEG, the compression rate should also be considered seriously. From Sup-Figure-19, the file we compressed is 983KB, while the origin file is 2.4MB, the **compression rate (Sup-Figure-19 is about 2.409, Sup-Figure-20 is about 2.588)**. Besides, our JPEG version not only compress the file but also encrypt it.

V. Conclusion

The origin problem we encountered is the ability to hide the image and the quality of recover image. We solve it with not using quantization matrix.

We also discuss the effect of adding noise and different scanning method. Adding noise can protect the image with similar image more well. However, we think using different scanning method has no benefit to encrypt, because the scanning each column didn't help to hide the details.

Overall, we are satisfied with our project, no matter the compression rate or the ability to encrypt.

VI. Contribution

A. Method Contribution

The part of encryption and decryption come up by ourselves. However, the procedure of JPEG come from reference [1]~[3].

B. Implementation Contribution

MATLAB: Huffman_code.m, source_coding.m, source_decoding.m comes from reference [4], other .m and .mlx file are done by 曾泓硯.

Python: Huffman.py comes from reference [5], JEPGIO.py is modified from reference [5]. Other .py files are done by 廖宏儒.

VII. Team Work

106061101 曾泓硯: MATLAB code, Algorithm of JPEG, encryption and decryption, team12.pdf & supplementary_team12.pdf

106061209 廖宏儒: Python code, convert MATLAB code to python, GUI of python code, Binary file I/O using python, Readme file

VIII. Reference

[1] JPEG Wikipedia Available: https://en.wikipedia.org/wiki/JPEG

[2] JPEG 解碼器 (A JPEG decoder in C++) Available: https://github.com/MROS/jpeg_decoder

[3] JPEG Compression, Quality and File Size Available: https://www.impulseadventure.com/photo/jpeg-compression.html

[4] PPT for Chapter 9 of "MATLAB/Simulink for Digital Communication" Available: https://ww2.mathworks.cn/matlabcentral/fileexchange/26384-ppt-for-chapter-9-of-matlab-simulink-for-digital-communication

[5] JPEG encoder/decoder written in Python Available: https://www.github.com/ghallak/jpeg-python/

[6] PyQT5 Available: https://pythonspot.com/pyqt5/