

Data Compression Project 2

Author

- 112522083
 - 中央資工碩一
 - 鄧祺文
-

Environment

- Operating System: VMware Workstation with Ubuntu 20.04 LTS (因為 Windows 不知道為何壓一壓會出 Bug，導致 RGB 執行出來的結果會直接變成扭曲的圖像，最後的解決方法是通過 VM 換一個環境來進行實驗，然後問題就解決了...)
- C/C++ Compiler: GCC
- Project IDE: JetBrains CLion with CMake

開啟 JetBrains CLion 並完成 C/C++ Compiler 配置後，開啟 JPEG 的 Project 的資料夾，即可執行並於 Terminal 進入功能選單。

JPEG Function Menu

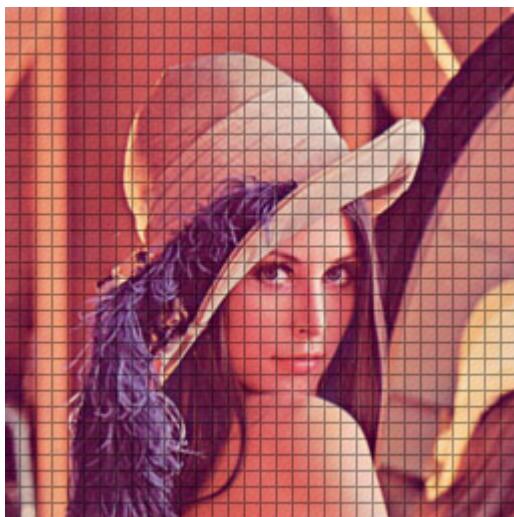
```
int main() {
    decode(isRGB: false);
    PSNR(isRGB: false, width: 512, height: 512, filename);
    break;
    case 2:
        cout << "Enter which file to encode: ";
        cin >> filename;
        cout << "Enter the quality factor to encode: ";
        cin >> QF;
        encode(isRGB: true, width: 512, height: 512, filename, QF);
        decode(isRGB: true);
        PSNR(isRGB: true, width: 512, height: 512, filename);
        break;
    case 3:
        exit( Code: 0 );
    default:
        cout << "Invalid choice!" << endl;
}
```

C:\Users\toby0\Documents\GitHub\NCU-Data-Compression\Project2-JPEG-Compressor\JPEG\cmake-build-debug\JPEG.exe
which function to use?
1) encode gray level image and decode
2) encode color image and decode
3) exit

JPEG Design

Compress

- 由於 JPEG 壓縮的操作方法是將圖片分割成多個區塊來進行壓縮，所以一開始要先把原圖切成許多 8×8 的區塊。如下圖所示：

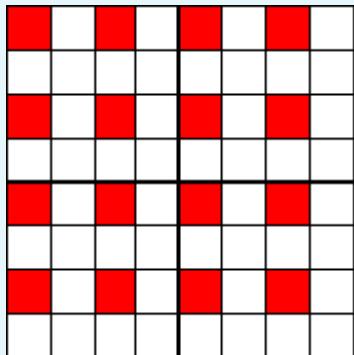


- 接下來要進行色彩空間轉換（RGB 圖片需要此步驟，Gray Image 就不需要了），將 RGB 轉換為 YCbCr 來進行壓縮，YCbCr 分別代表的是亮度以及彩度。

- 再來因為人類肉眼對於亮度的敏感程度相較於彩度要高，所以彩度可以使用 **Downsampling** 的方式來減少資料量。單就本次實現而言，採用的是最普遍的 **4:2:0** 方式來進行處理。

Info

- 把圖片切成 2×2 的區塊只取左上角的區塊來進行壓縮



- 8×8 的區塊就只取紅色的部分來做壓縮，資料量就變成原本的「四分之一」

- 之後再對切出來的區塊做離散餘弦變換 (**Discrete Cosine Transform, DCT**)，會把這個 8×8 區塊的內容轉換成左上到右下，低頻到高頻分布。一個簡單的例子如下圖：

1. 假設有一個長這樣的 8×8 (8 Bit, 0 ~ 255) 子區域

52	55	61	66	70	61	64	73
63	59	55	90	109	85	69	72
62	59	68	113	144	104	66	73
63	58	71	122	154	106	70	69
67	61	68	104	126	88	68	70
79	65	60	70	77	68	58	75
85	71	64	59	55	61	65	83
87	79	69	68	65	76	78	94

2. 推移 128，使其範圍變為 (-128) ~ 127，得到結果為

$$\begin{bmatrix} -76 & -73 & -67 & -62 & -58 & -67 & -64 & -55 \\ -65 & -69 & -73 & -38 & -19 & -43 & -59 & -56 \\ -66 & -69 & -60 & -15 & 16 & -24 & -62 & -55 \\ -65 & -70 & -57 & -6 & 26 & -22 & -58 & -59 \\ -61 & -67 & -60 & -24 & -2 & -40 & -60 & -58 \\ -49 & -63 & -68 & -58 & -51 & -60 & -70 & -53 \\ -43 & -57 & -64 & -69 & -73 & -67 & -63 & -45 \\ -41 & -49 & -59 & -60 & -63 & -52 & -50 & -34 \end{bmatrix}$$

3. 執行 DCT，和捨位取最接近的整數，得到結果為

$$\begin{bmatrix} -415 & -30 & -61 & 27 & 56 & -20 & -2 & 0 \\ 4 & -22 & -61 & 10 & 13 & -7 & -9 & 5 \\ -47 & 7 & 77 & -25 & -29 & 10 & 5 & -6 \\ -49 & 12 & 34 & -15 & -10 & 6 & 2 & 2 \\ 12 & -7 & -13 & -4 & -2 & 2 & -3 & 3 \\ -8 & 3 & 2 & -6 & -2 & 1 & 4 & 2 \\ -1 & 0 & 0 & -2 & -1 & -3 & 4 & -1 \\ 0 & 0 & -1 & -4 & -1 & 0 & 1 & 2 \end{bmatrix}$$

- DCT 結束後再進行量化工作，由於人類眼睛在高頻率亮度變動之確切強度的分辨上沒有想像中敏感，這讓我們能在高頻率成份上極佳地降低資訊的數量。例子如下：

1. 假設一個長這樣的量化矩陣

$$\begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

2. 使用這個量化矩陣與前面所得到的 DCT 結果逐項相除，得到結果

-26	-3	-6	2	2	-1	0	0
0	-2	-4	1	1	0	0	0
-3	1	5	-1	-1	0	0	0
-3	1	2	-1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

- 量化完成，最後就利用 Huffman Encode 做無損壓縮（採查表形式）。最左上角的是 DC 因為相鄰區塊的差異不大所以使用 DPCM 的方式來壓縮；其他 63 個是 AC 先做 Zigzag 會得到一連串 0 和非 0 的數字，因為越右小角是高頻的部分量化完會變成很多 0，所以先做 Zigzag 可以讓壓縮比較有效率。後面再來做 Run Length Coding 去處理成前面有幾個 0 跟後面非 0 的數字是多少並查表，就相當於做完 JPEG 的壓縮了。

Summary

壓縮流程總結

- Gray：切塊 → DCT → 量化 → Huffman Encode
- RGB：切塊 → 採樣 → DCT → 量化 → Huffman Encode

Decompress

- 解壓縮的過程就是把壓縮的流程反著做一遍，只要把壓縮後的 Bit Stream 去查表還原成 Run Length Coding 的樣子，再把 Run Length Coding 更原始的狀態，再按照 Zigzag 的位置把所有的值填回去就會回到沒做 Zigzag 前的樣子，最後再來做反量化跟 IDCT 就會還原回原本 8 x 8 的區塊。而 RGB 的彩度部分因為有做 Downsampling，並且該操作為不可還原，所以就用採樣的那個值來當作 2 x 2 區塊的彩度。

Summary

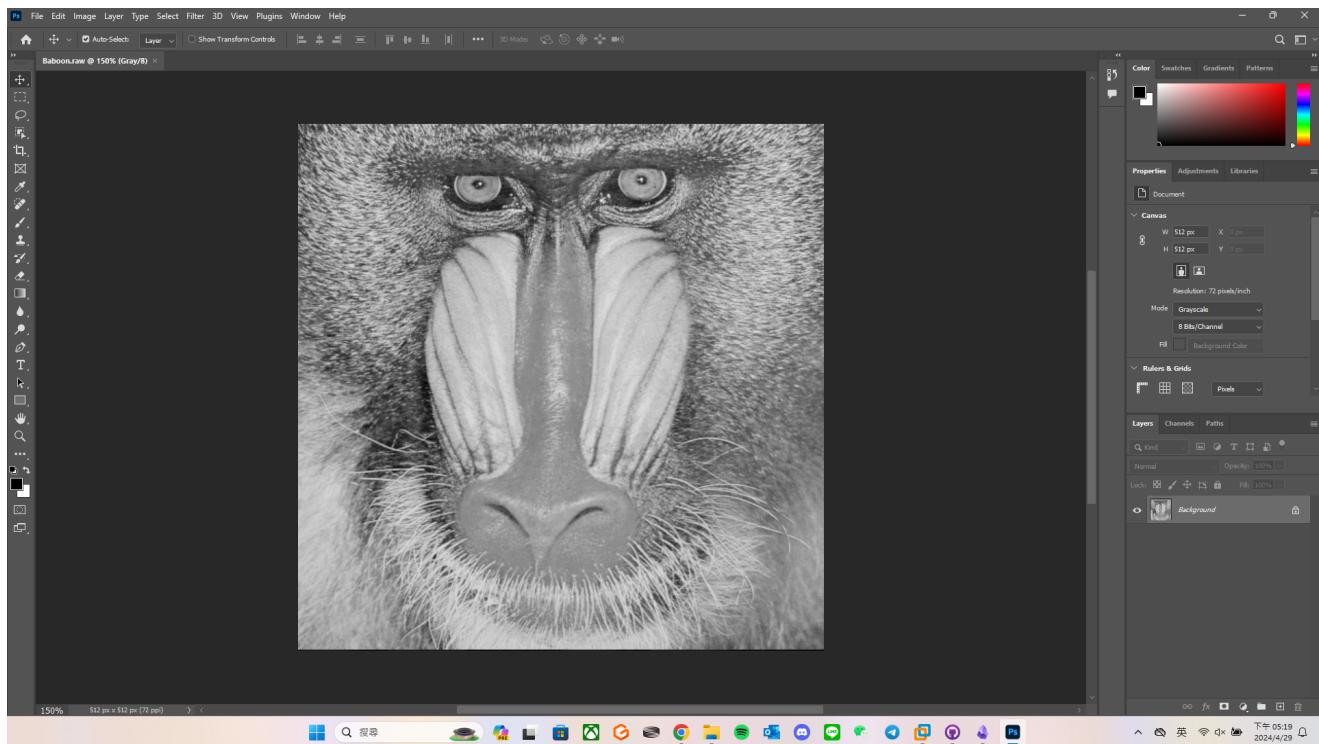
解壓縮流程總結

- Gray : Huffman Decode → 反量化 → IDCT → 還原成原本 8×8 區塊
- RGB : Huffman Decode → 反量化 → IDCT → 升採樣 → 還原成原本 8×8 區塊

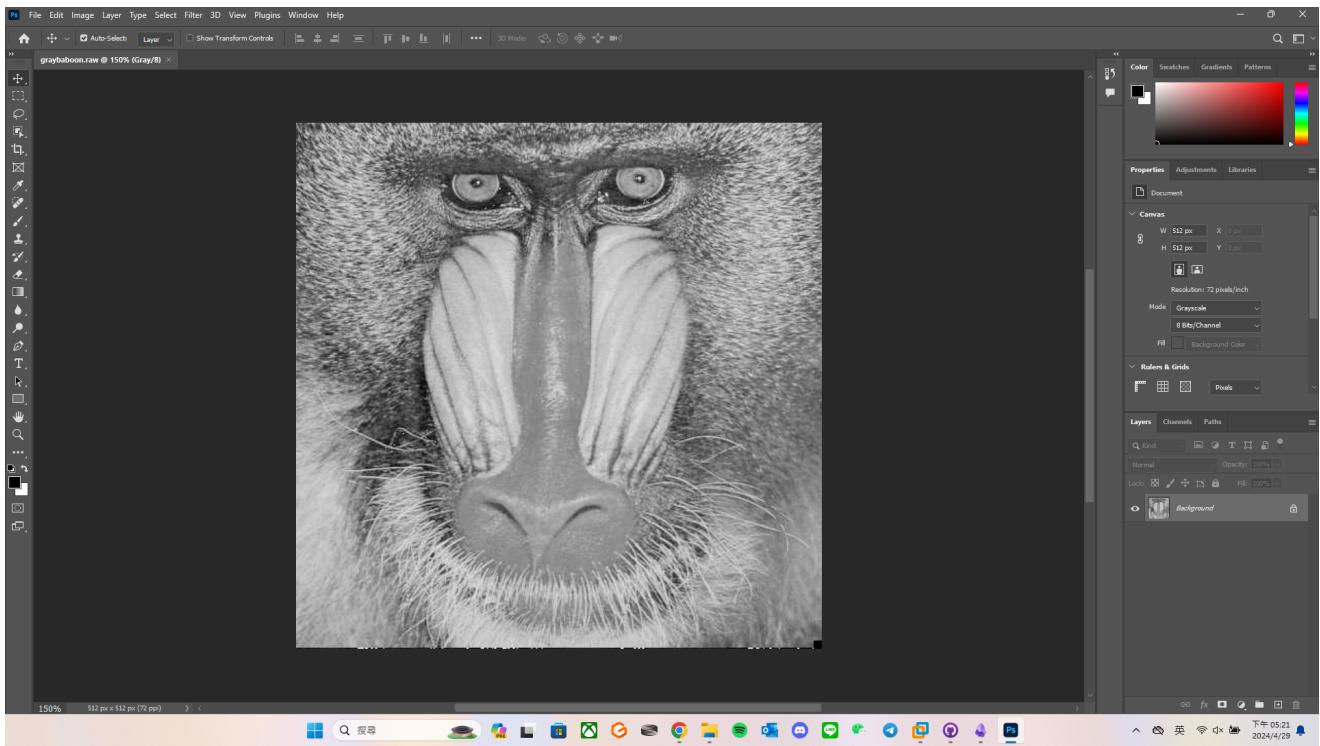
Results

Gray Image

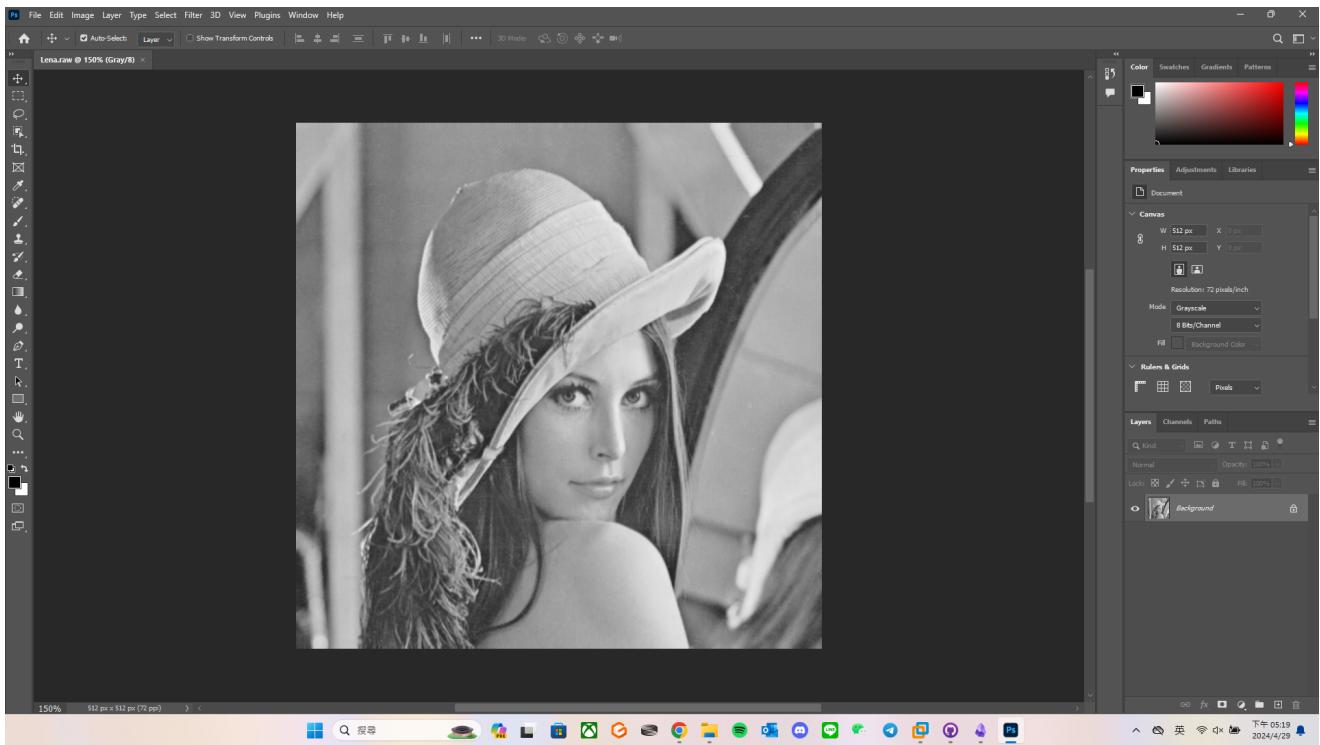
- Original "Baboon" RAW



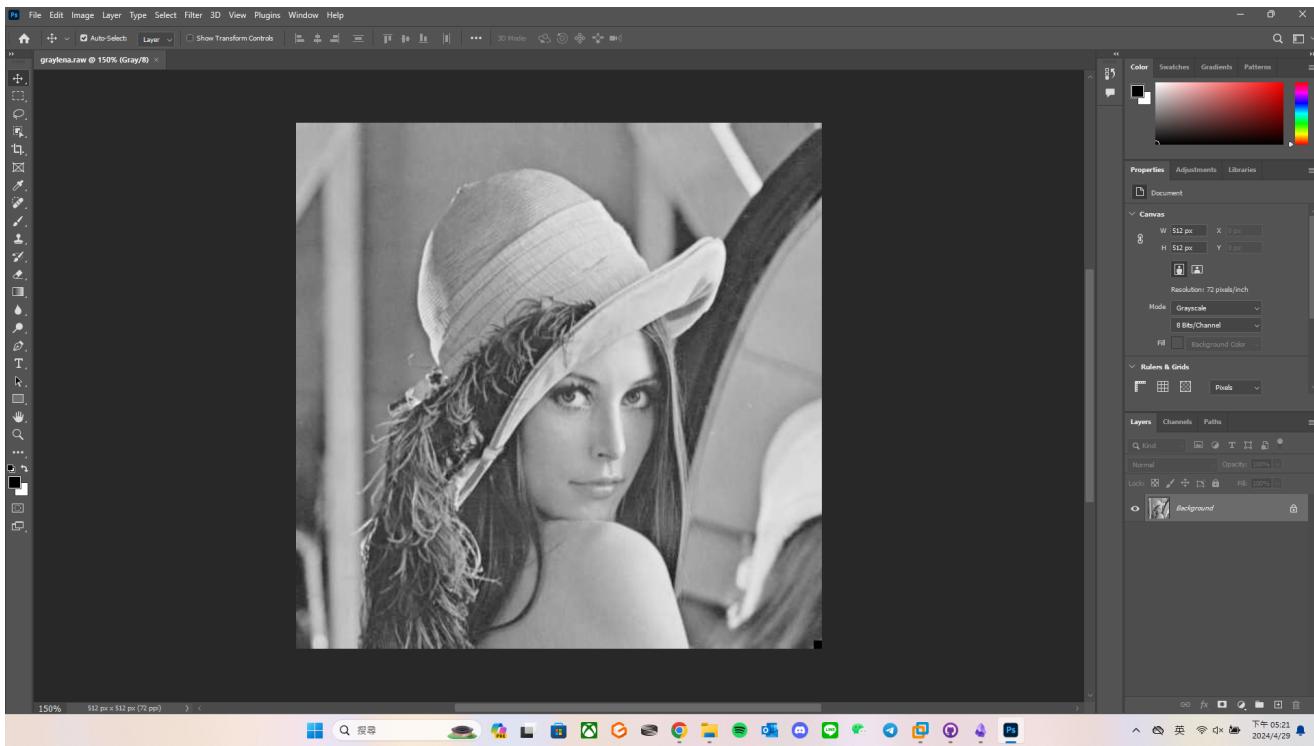
- Encode/Decode "Baboon"



- Original "Lena" RAW

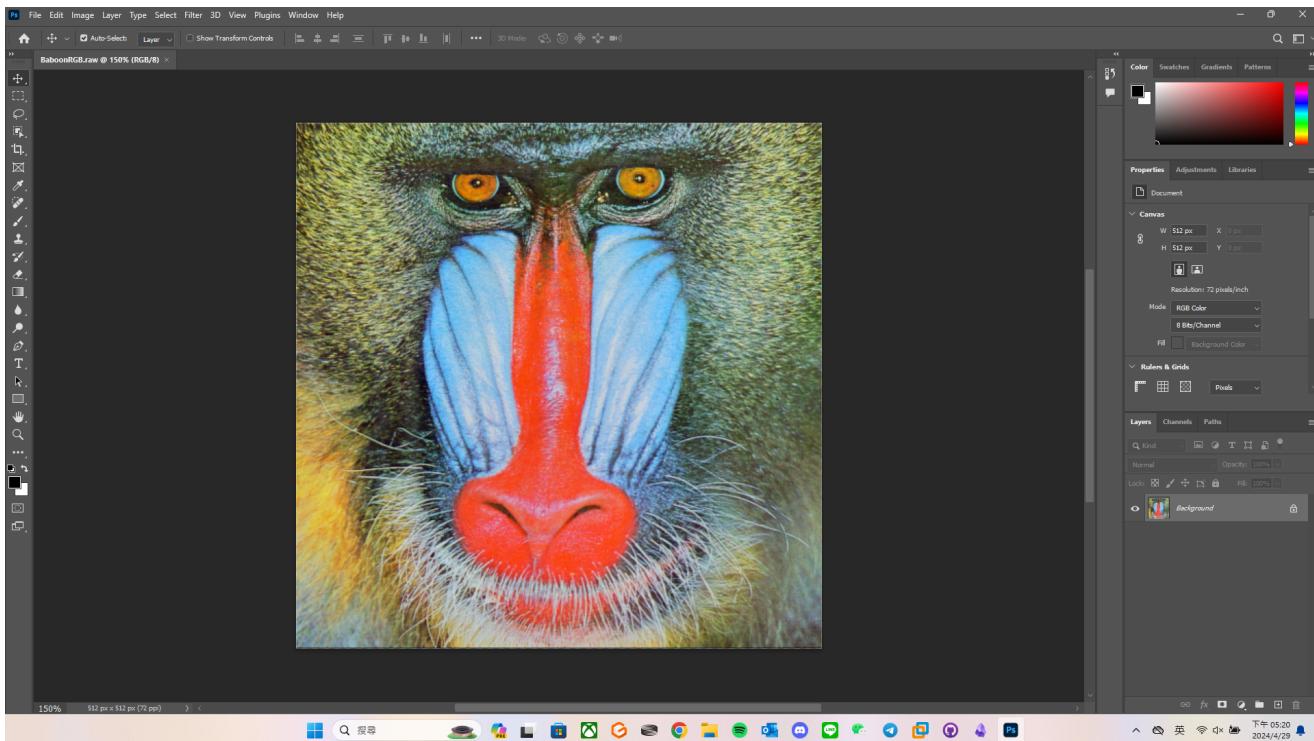


- Encode/Decode "Lena"

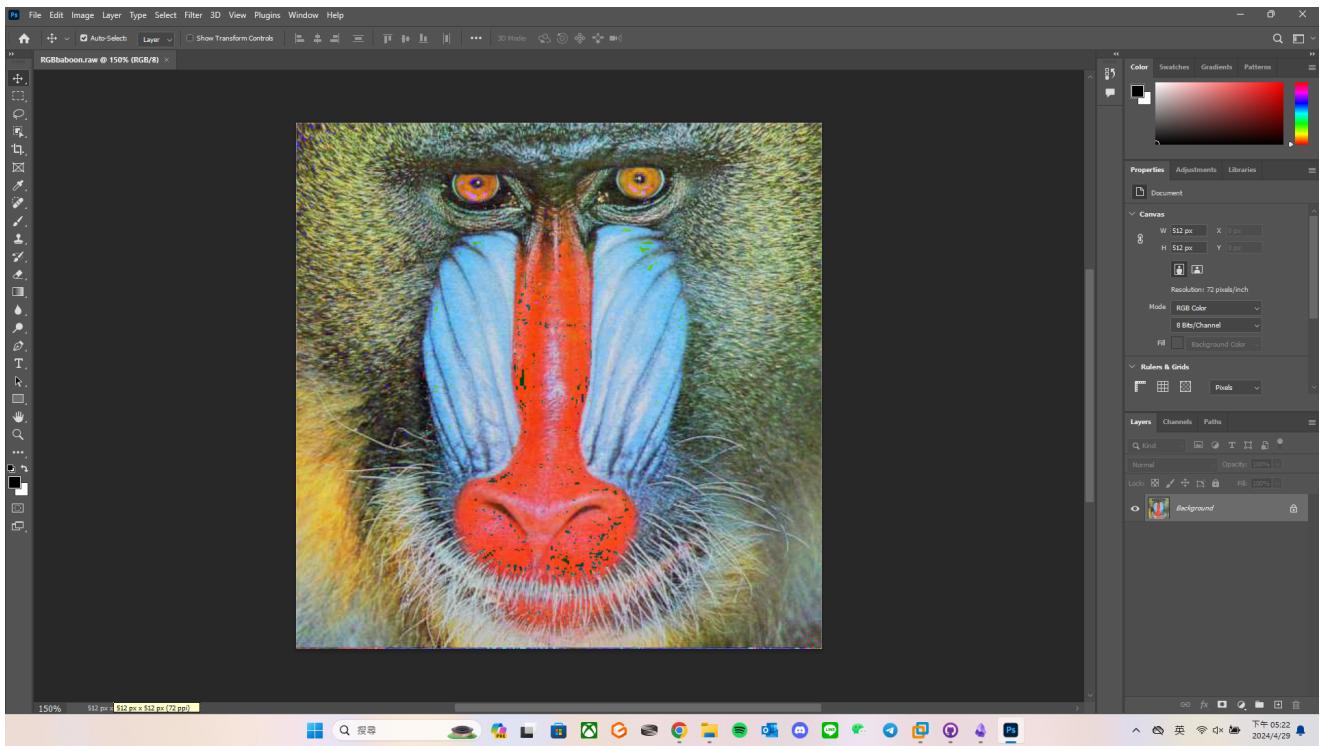


RGB Image

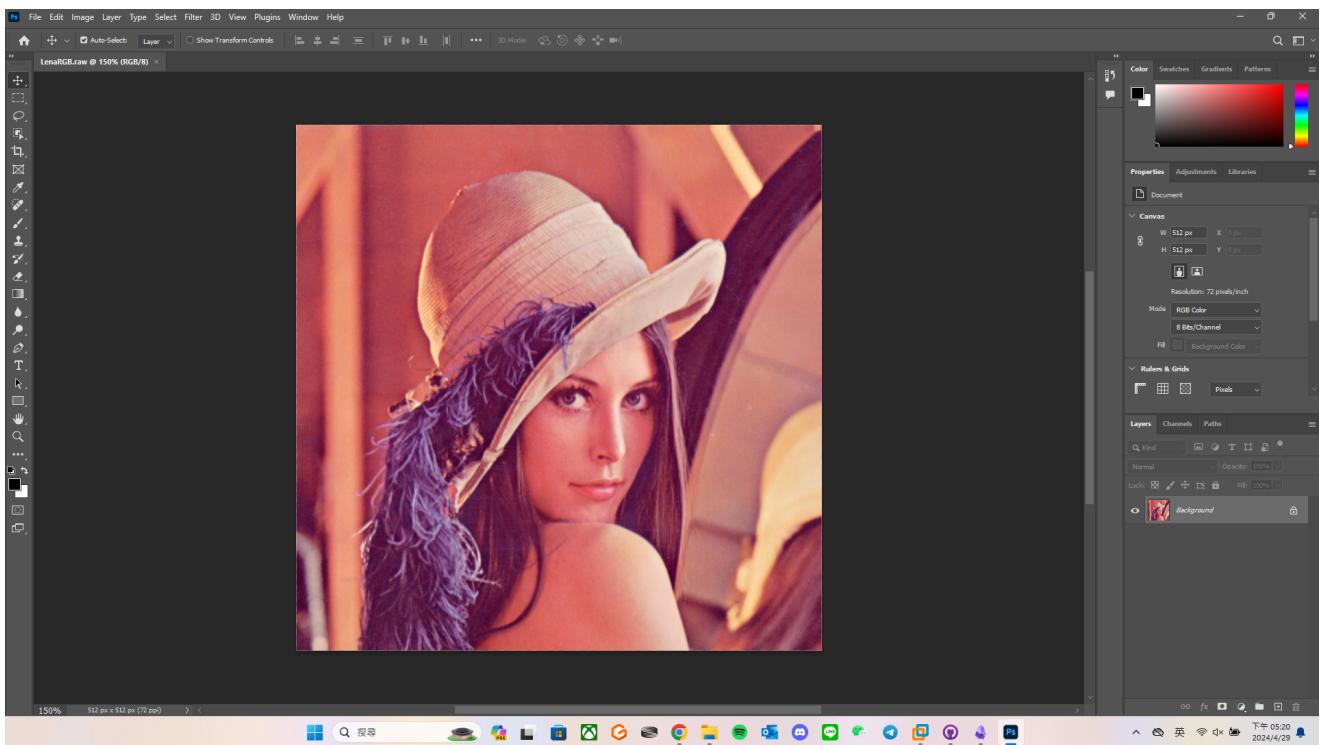
- Original "BaboonRGB" RAW



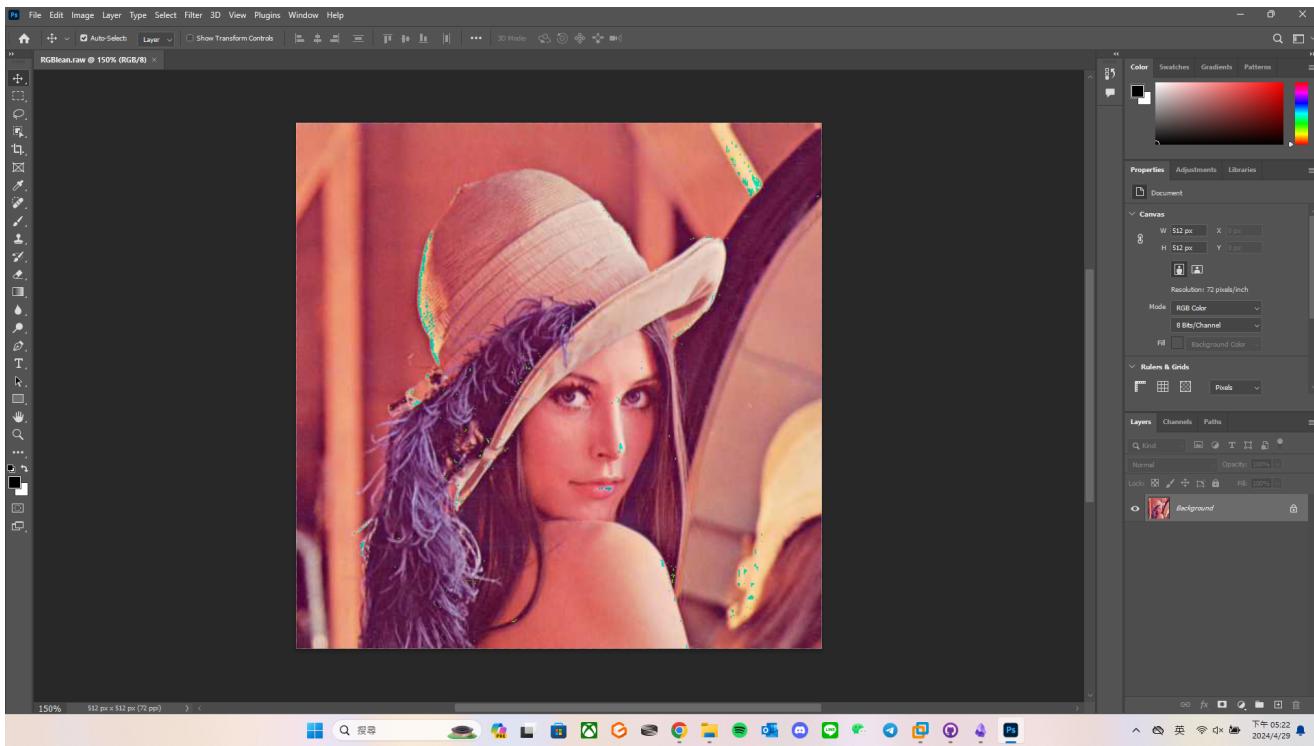
- Encode/Decode "BaboonRGB"



- Original "LenaRGB" RAW



- Encode/Decode "LenaRGB"



PSNR 與 QF 比較

PSNR/QF	90	80	50	20	10	5
Gray Baboon	34.614	31.049	27.3	24.824	23.039	20.998
Gray Lena	39.445	37.705	35.807	32.942	30.118	26.796
RGB Baboon	21.74	21.983	21.812	20.556	19.475	17.639
RGB Lena	29.982	29.232	27.534	25.085	22.827	19.504

- PSNR 越高代表壓縮後的結果與原圖的差異越小，由 Quality Factor 的操作性質可以知道 QF 的設的值越高，PSNR 理論上也要越高，這次的實驗成功證明了這點。
- Gray 的影像壓縮後的 PSNR 普遍比 RGB 的影像縮後的 PSNR 更高，主要是因為有顏色這件事本身也有影響。RGB 圖片經過 Sampling 會把彩度從原本的大小往下降為四分之一，而且過程不可逆，導致除量化以外的額外損失，進而使得在相同 Quality Factor 的情況下，RGB 會擁有低得多的 PSNR 結果。

Quantization 的影響

- 因為在做量化的時候是拿 DCT 後的結果跟量化的表相除，除完之後做四捨五入取整數把小數的部分捨掉，導致 JPEG 為有損壓縮。
- Quality Factor 的結果公式如下，也就是影響量化的原因：

$$f(QF) = \begin{cases} \frac{5000}{QF}, & \text{if } QF < 50 \\ 200 - 2QF, & \text{if } QF \geq 50 \end{cases}$$

- $Q = S * f(QF) / 100$ ，S表示的是量化表的值，DCT 的結果再去除上 Q 得到量化後的結果。
 - 也就是說，當 Quality Factor 越小的時候會使得量化造成的損失越多，壓縮後的品質就越差，不過相對的壓縮後的檔案大小就越小。
-