PURDUE UNIVERSITY

ECE 637 Laboratory Exercise 9
Achromatic Baseline JPEG encoding Lab

Tong Shen

May 5, 2017

1 INTRODUCTION

Nothing to report for this section.

2 DCT BLOCK TRANSFORMS AND QUANTIZATION

The source image is first broken into 8x8 blocks. The pixel values Syx in each of these blocks are then transformed by the FDCT into an 8 x 8 block of 64 DCT coefficients.

2.1 CODE LIST OF TRANSFORMING, QUANTIZATION AND STORING

```

I = imread('img03y.tif');
I = double(I);
I = I - 128;

gamma = 1;
fn = @(x) round(dct2(x.data,[8,8])./(Quant*gamma));
dct_blk = blockproc(I,[8,8],fn);
dct_blk = dct_blk';
[r,c] = size(dct_blk);
fileID = fopen('img03y.dq','w');
fwrite(fileID,r,'integer*2');
```

```
fwrite(fileID,c,'integer*2');
fwrite(fileID,dct_blk,'integer*2');
fclose(fileID);

image_1 = readfile('img03y.dq',gamma);

imshow(image_1')
image_different = image_1' - I1;
figure(2)
imshow(image_different)
```

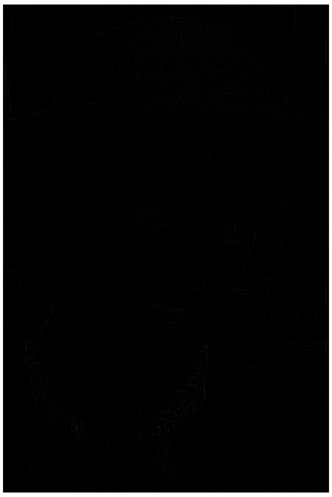
```

function [image] = readfile(file,gamma)
run('Qtables');
f = fopen(file,'r');
data = fread(f,'integer*2');
image = reshape(data(3:end),[data(1),data(2)]);
fn = @(x) round(idct2(x.data.*Quant*gamma,[8,8]));
image = blockproc(image,[8,8],fn);
image = image + 128;
image = uint8(image);
end
```

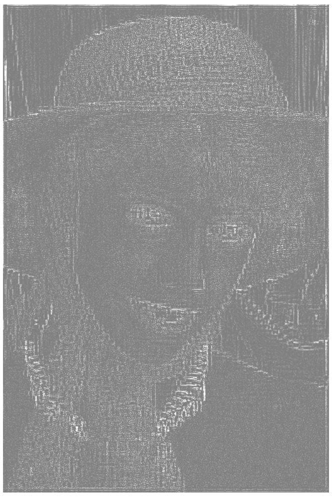
2.2 Result report

Figure 2.1: The original image

(a) *img0.25.tif*



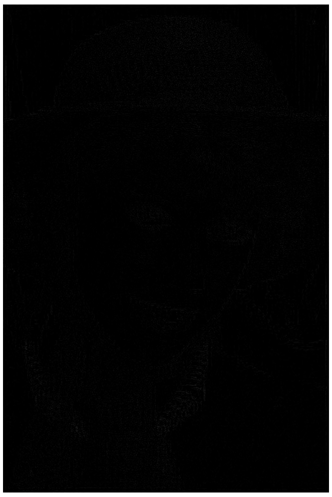
(b) *img0.25d.tif*



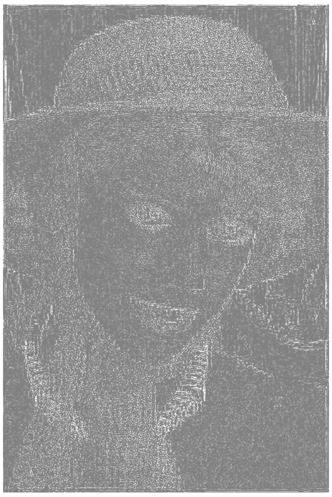
(c) *img0/25Enhanced.tif*

Figure 2.2: Restored Image and the difference with a r of 0.25

(a) *img1.tif*



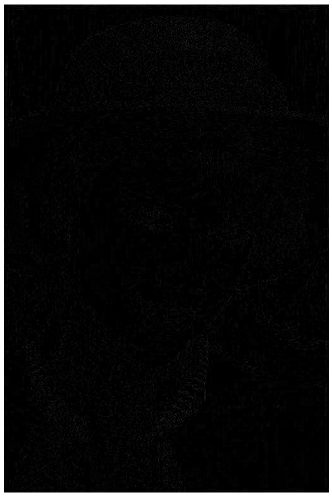
(b) *img1d.tif*



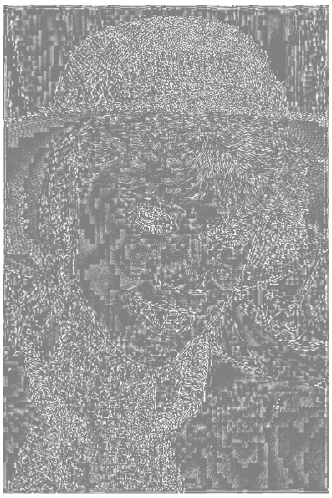
(c) *img1Enhanced.tif*

Figure 2.3: Restored Image and the difference with a γ of 1

(a) *img4.tif*



(b) *img4d.tif*



(c) *img4Enhanced.tif*

Figure 2.4: Restored Image and the difference with a γ of 4

PS: In this part, the error img was enhanced with a formula of 10 times multiply and adding 128 pixel gray value.

2.3 Comment on the Result

According to the result, the γ represent a quantization factor in the process. With the increasing of γ, the quantization blocks are bigger and the resolution decreases.

3 Differential Encoding and the Zig-Zag Scan Pattern

To improve image quality and reduce bit rate, the DC coefficient is differentially encoded. This means that, using a raster ordering of the blocks, only the difference between the current and previous DC coefficients is coded.

3.1 Result report

Figure 3.1: The DC component formed image

This looks a little dizzy, like a resized form of the original image. this is probably because the image information is mainly stored in the DC components.

The DC components have the highest energy and represent the main information in the whole coding block. And in an image, nearby pixels are correlated. So, the adjacent DC components are correlated.

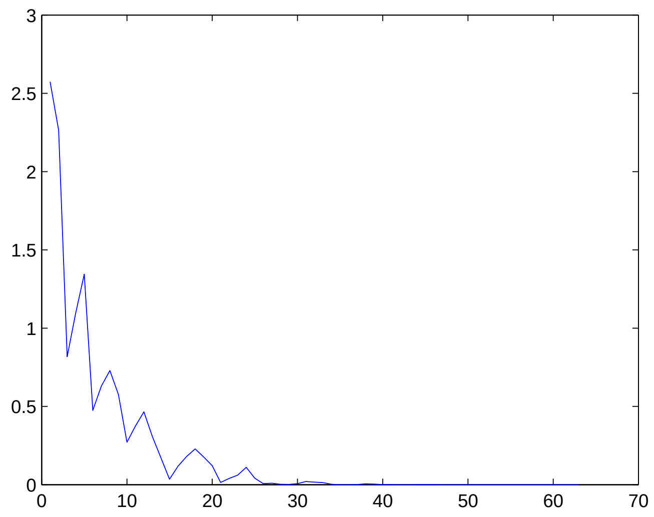


Figure 3.2: plot of the mean value of the magnitude of the AC coefficients for $\gamma = 1.0$.

In this figure, the mean value of magnitude of the AC coefficients goes down. This is probably because in one DCT block, the energy goes down when it goes far away from the upper-left corner.

3.2 Code Listing

```

I = imread('img03y.tif');
I = double(I);
I = I - 128;

gamma = 1;
fn = @(x) round(dct2(x.data,[8,8])./(Quant*gamma));
dct_blk = blockproc(I,[8,8],fn);

[r,c] = size(dct_blk);
DC = zeros(r/8,c/8);
AC = zeros(r*c/64,63);

for i = 0:r/8-1
for j = 0:c/8-1
DC(i+1,j+1) = dct_blk(i*8+1,j*8+1) + 128;
temp = dct_blk(i*8+1:i*8+8,j*8+1:j*8+8);
temp = temp(Zig);
AC(i*c/8 + j + 1,:) = temp(2:end);
```

```
end
end

DC = uint8(DC);
figure(1)
imshow(DC)
AC_mean = mean(abs(AC));
t = 1:63;
figure(2)
plot(t,AC_mean)
```

4 Entropy Encoding of Coefficients

In order to reduce the number of bits required to represent the quantized image, both the differential DC and AC coefficients must be entropy encoded. To do this, JPEG uses two basic encoding schemes, the *Variable-Length Code* (VLC) and the *Variable-Length Integer* (VLI). The VLC encodes the number of bits used for each coefficient, and the VLI encodes the signed integer efficiently.

4.1 Subroutine Coding List

4.1.1 BitSize

```

int BitSize(int value)
{
int bitsize=0;

if(value<0)
value=-value;

while( (value-(1<<bitsize) >= 0) && (bitsize<16) )
bitsize++;

return(bitsize);
}
```

4.1.2 VLI_encode

```

void VLI_encode(int bitsize, int value, char *block_code)
{
int k;
char buffer[17];   /* max VLI code length should be 16 */
```

```
if(bitsize>16)
fprintf(stderr,"Error");

if(value<0)
value = value-1;

for(k=bitsize; k>0; k--){
if( (value & (1<<(k-1))) == 0)
buffer[bitsize-k]='0';
else
buffer[bitsize-k]='1';
}
buffer[bitsize]='\0';
strcat(block_code,buffer);
}
```

4.1.3 ZIGZAG

```
void ZigZag(int ** img, int y, int x, int *zigline) {
fprintf(stdout, "in ZigZag()\n");
int i, j;

for (i = 0; i < 8; i++) {
for (j = 0; j < 8; j++) {
zigline[Zig[i][j]] = img[y+i][x+j];
}
}
}
```

4.1.4 DC_ENCODER

```
void DC_encode(int dc_value, int prev_value, char *block_code) {
int diff, size;

diff = dc_value - prev_value;
size = BitSize(diff);

strcat(block_code, dcHuffman.code[size]);
VLI_encode(size, diff, block_code);
}
```

4.1.5 AC_ENCODER

```
  int idx = 1;
int zerocnt = 0;
int bitsize;

while (idx < 64) {
if (zigzag[idx] == 0) {
zerocnt++;
} else {
for (; zerocnt > 15; zerocnt -= 16) {
strcat(block_code, acHuffman.code[15][0]);
}

bitsize = BitSize(zigzag[idx]);
strcat(block_code, acHuffman.code[zerocnt][bitsize]);
VLI_encode(bitsize, zigzag[idx], block_code);

zerocnt = 0;
}

idx++;
}
```

4.1.6 BLOCK_ENCODER

```
void Block_encode(int prev_dc, int *zigzag, char *block_code)
{
DC_encode(zigzag[0],prev_dc,block_code);
AC_encode(zigzag,block_code);
}
```

4.1.7 CONVERT_ENCODER

```
int Convert_encode(char *block_code, unsigned char *byte_code) {
int len = strlen(block_code);
int bytes = len / 8;
int idx;
int i, j;

idx = 0;
for (i = 0; i < bytes; i++) {
for (j = 0; j < 8; j++) {
byte_code[idx] <<= 1;

if (block_code[i*8 + j] == '1') {
byte_code[idx]++;
```

```
}
}

if (byte_code[idx] == 0xff) {
byte_code[++idx] == 0x00;
bytes++;
}

idx++;
}

strcpy(block_code, block_code + len / 8 * 8);

return bytes;
}
```

4.1.8 Zero_pad

```
unsigned char Zero_pad(char *block_code)
{
unsigned char byte_value=0;
int k=0;
char mask;

while( block_code[k] != '\0' ) {
mask= 0x80 >> (k%8);

if( block_code[k] == '1' )
byte_value |= mask;

if( block_code[k] == '0' )
byte_value &= (~mask);

k++;
}

return(byte_value);
}
```

4.2 Main Image Coding Program

```


#include <stdio.h>
#include <stdlib.h>
#include <string.h>

```

```
#include "Htables.h"
#include "JPEGdefs.h"
#include "allocate.h"


int main(int argc, char* argv[])
{
int   **input_img;
FILE   *outfp;
int      row;          /* height   */
int      column;       /* width   */
double gamma;          /* scaling parameter */
int temp, k,i,j;
char tempc[100];
int tempi[64];
int **img;
unsigned char byte_code[100];
unsigned char tempuc;

input_img = get_arguments(argc,argv,&row,&column,&gamma,&outfp) ;

if( gamma > 0 )
change_qtable(gamma) ;
else {
fprintf(stderr, "\nQuantizer scaling must be > 0.\n") ;
exit(-1) ;
}

jpeg_encode(input_img,row,column,outfp) ;

return 1 ;
}


void change_qtable(double scale)
{
int      i,j ;
double   val ;

for(i=0;i<8;i++){
for(j=0;j<8;j++){
val = Quant[i][j]*scale ;
/* w.r.t spec, Quant entry can be bigger than 16 bit */
Quant[i][j] = (val>65535) ? 65535 : (int)(val+0.5) ;
}
}
}


int **get_arguments(int argc,
char *argv[],
int *row,
int *col,
double *gamma,
```

```
FILE **fp )
{
/* float val ; */
FILE * inp ;
short** img ;
int ** in_img ;
short tmp ;
int i,j ;

switch(argc){
case 0:
case 1:
case 2:
case 3: usage(); exit(-1) ; break ;
default:

sscanf(argv[1],"%lf",gamma) ;

*fp = fopen(argv[3],"wb") ;
if(*fp==NULL) {
fprintf(stderr,
"\n%s file error\n",argv[3]) ;
exit(-1) ;
}

inp = fopen(argv[2],"rb") ;
if( inp == NULL ) {
fprintf(stderr,
"\n%s open error\n",argv[2]) ;
exit(-1) ;
}

fread(&tmp,sizeof(short),1,inp) ;
*row = (int) tmp ;
fread(&tmp,sizeof(short),1,inp) ;
*col = (int) tmp ;

img = (short **)get_img(*col,*row,sizeof(short)) ;
fread(img[0],sizeof(short),*col**row,inp) ;
fclose(inp) ;

break ;
}

in_img = (int **)get_img(*col,*row,sizeof(int)) ;
for( i=0 ; i<*row; i++ ){
for( j=0 ; j<*col; j++ ){
in_img[i][j] = (int) img[i][j] ;
}
}
free_img((void**)img) ;
return( in_img ) ;
}
```

```
void jpeg_encode(int **img, int h, int w, FILE *jpgp)
{
int     x, y, i, j, length ;
int     prev_dc = 0 ;
unsigned char val ;
static int     zigline[64] ;
static char    block_code[8192] = {0} ;
static unsigned char byte_code[1024] ;

printf("\n JPEG encode starts...") ;
/* JPEG header writes */
put_header(w,h,Quant,jpgp) ;

printf("\n Header written...\n Image size %d row  %d column\n",h,w) ;
/* Normal block processing */
for( y = 0 ; y < h ; y += 8) {
for( x = 0 ; x < w ; x += 8 ){
/* read up 8x8 block */
ZigZag(img,y,x,zigline) ;

Block_encode(prev_dc,zigline,block_code) ;


prev_dc = zigline[0] ;
length = Convert_encode(block_code,byte_code) ;
fwrite(byte_code,sizeof(char),length,jpgp) ;
}
printf("\r (%d)th row processing    ",y) ;
}
printf("\nEncode done.\n") ;

if( strlen(block_code) ){
val = Zero_pad(block_code) ;
fwrite(&val,sizeof(char),1,jpgp) ;
}

put_tail(jpgp) ;
fclose(jpgp) ;
free_img((void **)img) ;
}
```

4.2.1 RESULT REPORT

(a) *output_.25.jpg*

(b) *img_1.jpg*

(c) *img_4.jpg*

Figure 4.1: Restored Image and the difference with a r of 0.25, 1 and 4