CS3570 Introduction to Multimedia

Homework #1 Report

103062234 張克齊

1. DCT image compression

[實作方法]

The DCT Transform Matrix

There are two ways to compute the DCT using Image Processing Toolbox™ software. The first method is to use the `dct2` function. `dct2` uses an FFT-based algorithm for speedy computation with large inputs. The second method is to use the DCT *transform matrix*, which is returned by the function `dctmtx` and might be more efficient for small square inputs, such as 8-by-8 or 16-by-16. The M-by-M transform matrix `T` is given by

$$T_{pq} = \begin{cases} \frac{1}{\sqrt{M}} & p = 0, \quad 0 \le q \le M-1 \\ \sqrt{\frac{2}{M}} \cos\frac{\pi(2q+1)p}{2M} & 1 \le p \le M-1, \quad 0 \le q \le M-1 \end{cases}$$

For an M-by-M matrix `A`, `T*A` is an M-by-M matrix whose columns contain the one-dimensional DCT of the columns of `A`. The two-dimensional DCT of `A` can be computed as `B=T*A*T'`. Since `T` is a real orthonormal matrix, its inverse is the same as its transpose. Therefore, the inverse two-dimensional DCT of `B` is given by `T'*B*T`.

```
function output = blockDivide(size)
%blockDivide Summary of this function goes here
%   Detailed explanation goes here
    for i = 1 : size;
        for j = 1 : size;
            if i ~= 1
                output(i,j) = sqrt(2/size)*cos(pi*(2*j-1)*(i-1)/(2*size));
            else
                output(i,j)=1/sqrt(size);
            end
        end
    end
end
```

<- Calculate Tpq

在 Matlab 官方文件中，有提到實作 DCT 的方法有兩種，一種是利用 FFT-based algorithm，另一種則是利用 DCT transform matrix 來完成，我的方法是利用後者。因此，我們要先利用上圖公式計算出 8 x 8 的 transform matrix，這樣我們就可以利用 B = Tpq * A * Tpq' 計算出 DCT 後的結果，其中 Tpq' 是 Tpq 的 transpose matrix；做這個矩陣相乘時我們要利用到 blockproc()這個函式，可以幫助我們快速的處理分塊而不用使用到 for 迴圈，用法是要先建立一個 structure 表示我們想要處理的函數 `func = @(block_struct) Tpq * block_struct.data * Tpq';`，然後在利用 blockproc 表示我們運算時要切的大小，切除來的就是上面的 block_struct.data 的值，再進行 func 中的運算 `output = blockproc(input, [8 8], func);`。

做 IDCT 前，我們要先對做完 DCT 的結果做題目要求的處理，就是先建造一個 8 x 8 的 mask 並依照指定的 n 給值 = 1，對 DCT 轉換後的結果做點乘，這樣才能再利用跟 DCT 一樣的方法做 IDCT 的轉換。轉換的方式就是從 B = Tpq * A * Tpq' 變成 A = Tpq' * B * Tpq (Linear algebra method)。

```
func_mask = @(block_struct)mask .* block_struct.data;
input_mask = blockproc(input, [8, 8], func_mask);
```

<- applying mask

```
func = @(block_struct)Tpq' * block_struct.data * Tpq;
output = blockproc(input_mask, [8, 8], func);
```

<- A = Tpq* B * Tpq

至於 YIQ 的部分很容易，透過公式創立一個 matrix K 並取出 input image 的 RGB 各項進行矩陣相乘，做回 IYIQ 的部分就是變成 inv(K)與 YIQ 各項進行矩陣相乘，也就是從 YIQ = K *RGB 變成 inv(K) * YIQ = RGB，就能得到結果。

From RGB to YIQ

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.211 & -0.523 & 0.312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

PSNR 的部分也是利用公式，但要注意的是因為彩色圖有 RGB，所以要對三個維度分開計算在取平均，才是我們想要的 MSE。圖中的的 255 是指峰值，但因為我們傳進去的是 single (0~1)所以應該改成 1。

$$\mathrm{PSNR} = 10 \times \lg\left(\frac{255^2}{\mathrm{MSE}}\right)$$

$$\mathrm{MSE} = \frac{1}{M \times N} \sum_{i=1}^{N} \sum_{j=1}^{M} \left[I(i,j) - I'(i,j) \right]^2$$

```
MSER= sum(sum((first(:,:,1)-second(:,:,1)).^2)) / M / N;
MSEG= sum(sum((first(:,:,2)-second(:,:,2)).^2)) / M / N;
MSEB= sum(sum((first(:,:,3)-second(:,:,3)).^2)) / M / N;
MSE = (MSER + MSEG + MSEB) / 3;
output = 10 * log10(1 / MSE);
```

[結果圖]

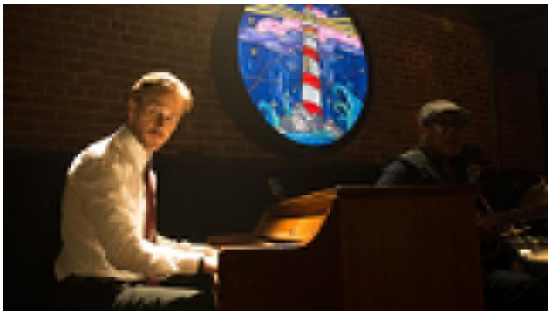
(a) DCT results:

n = 2, PSNR: 26.8887

n = 4, PSNR: 33.3722

n = 8, PSNR: `Inf`

(b) YIQ color model



n = 2, PSNR: `26.8887`

n = 4, PSNR: `33.3722`

n = 8, PSNR: `Inf`

[結論]

(1) n = 8 時的討論: 我一開始在做 n = 8 時，做出來的結果是 300 左右，但是理想中 mask 8 x 8 在 block 8 x 8 上作用應該會使 PSNR 中 MSE 為 0(因為圖片所有部分都保留)，答案要是 Inf.才對。後來與同學討論發現應該是因為在做 PSNR 運算時，我們要先將圖片取成 uint8 type 再轉成 single，才可以解決 matlab 中浮點數精度的問題，把一些接近於 0 的數給變成 0，就能得到正確的 PSNR 值。

```
p2 = myPSNR(im2single(im2uint8(I)),im2single(im2uint8(II2)));
```

(2) PSNR 的意義: PSNR 愈大，代表圖片的失真現象較小，因為 n 取愈大代表我們圖片保留的部分愈多，所以 PSNR 會隨著 n 變大而變大；當 n = 8 時，代表著圖片被全保留，就是跟原圖一樣，所以 PSNR 無限大。(a)和(b)的結果相同，因為我們可以知道他只是從 RGB 的表示法經過線性轉換成 YIQ，並沒有對圖片直接產生影響。

(3) (a)和(b)的差異: RGB 的方式雖然和 YIQ 一樣，但因為資料量比較大，所以透過公式中那個 3 x 3 的矩陣(K)運算後，不僅能保留原本的圖片，也能達到稍微壓縮資料的效果。

2. Image filtering

[實作方法]

這一題是要實現 filtering 的功能，分成利用了 Gaussian 和 Median 的方法。

(a) Gaussian blurring: 首先依照題目提示我們可以利用 fspecial()的函式創建出我們想要的 Gaussian masks。

```
% Create the gaussian filter with size and sigma
G1 = fspecial('gaussian',[3 3], 0.3);
G2 = fspecial('gaussian',[9 9], 1);
```

接著，我們要運用卷積(CONV)的概念，算出每一點 filter 後的值，我對於邊界的處理是運用忽略法，就是邊界範圍的值不做 CONV 而直接取原圖的值，而運算卷積的技巧就是要先找出 filter 的中間點，接著一一對應 Input 算出中心點經過 CONV 處理後的值。

```
output(i, j) = sum(sum(filter.*now));
```

(b) Median filtering: 邊界處理的部分跟 Gaussian 一樣可以用忽略法，但在做運算的部分就有所不同；因為 Median 是給想劃分區塊並抓出該區塊的中位數，代表劃的區塊中間那個位置對應的值，所以我用了一個 K 存下當前我要處理的方塊。

```
K = input(i-mid_x:i+mid_x, j-mid_x:j+mid_x);
```

然後，我利用 reshape()的將 K 轉成 column 等於 1，

```
K = reshape(K, [f_size*f_size, 1]);
```

這樣就能利用 matlab 中的 median() 函式算出正確的中位數(詳細原因請看討論)。

```
output(i, j) = median(K);
```

[結果圖]

(a) Gaussian blur filter:

mask sizes 3 x 3 mask sizes 9 x 9

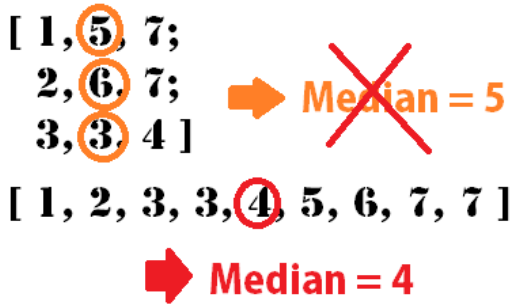
(b) Median filter:

mask sizes 3 x 3 mask sizes 9 x 9

[討論]

(1) 邊界處理的方法: 有關邊界處理的方法有很多，像是這邊的忽略方法或是 zero-padding 等。忽略法的優點在於程式簡潔而且不會對原本的 matrix 進行操作，但缺點是會產生像是上面結果圖所呈現的邊界明顯的問題；採用 zero-padding 的話，必須依據 filter 的 size 在 image matrix 的四周向外補零，直到第一次 filter 的中心落在 image 第一個位置，這樣就不會產生忽略法造成邊界未處理的缺點，但是就必須對原本的 matrix 進行操作。

(2) 對 matrix 取中位數的方法: 在實作方法中，有提到要將原本方陣轉成 total_row*1 的矩陣才能做 median 而非對 row 和 column 各做一次 median，這是因為再取中位數的時候並不能截成不同的段落取完再整合，因為它只代表在這個段落而非整個段落的中位數，如以下我自己舉的反例，第二個有經過 sort 比較好判斷，實際是直接跑內建函式 median()。



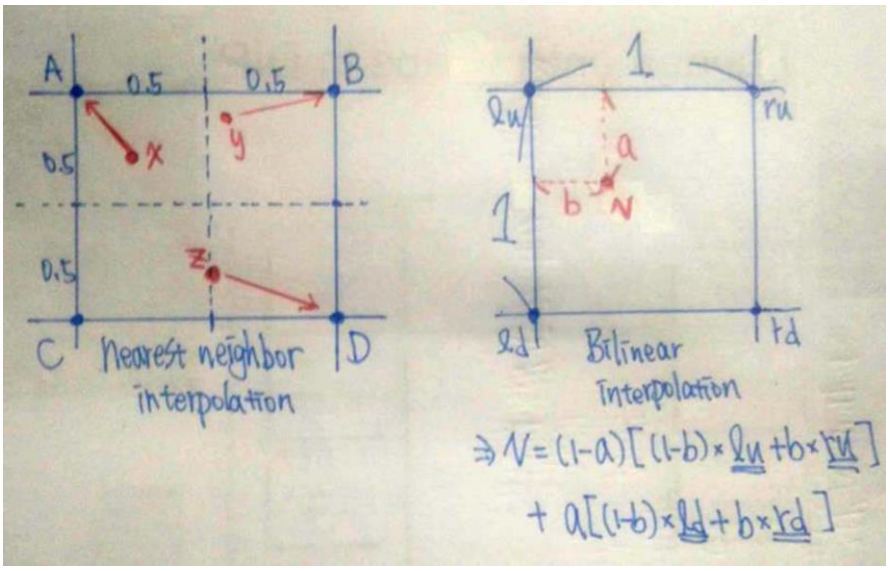
(3) Gaussian blurring 和 Median filtering 比較: 從得到的結果來看，我們可以發現當 mask 的 size 愈大時對圖片的影響會更大；更進一步來說，當 mask 大小為 9 x9 的時候，Median filtering 比較能夠濾掉所謂的胡椒鹽雜訊，但也可以發現整張圖片相較於經過 Gaussian blurring 變得較為模糊，原因是因為 Median filtering 只取了中位數的點，而 Gaussian blurring 則要進行 CONV 的運算才能得到該點的值，所以不難理解後者對原圖保留的較多，結果也較不模糊。

3. Interpolation

[實作方法]

這一題主要實作兩個 interpolation 的方法，Nearest Neighbor 和 Bilinear。

(a) Nearest Neighbor Interpolation (簡稱 NN): 這個方法是很基本的方法，概念是將 input 依照指定的比例放大縮小，並將坐標系也一起縮放，但這樣會產生有些點落在分數的問題，NN 的概念就是找到那些分數的點最接近坐標系的哪個整數點，而在 color channel 的部分就直接沿用該對應點的顏色。



(b) Bilinear Interpolation: 如上圖我們可以知道我們的目標是要找到對應點周圍的四個點並做 interpolation，但要注意我們不能只取 floor()和 ceil()找到四個點，因為會有圖片邊界的問題，所以我們還要對取完的點做判斷，這樣我們就能正確取出我們想要做 interpolation 的四個點和手繪圖中的 a 和 b。最後，找到四個點和 a, b 後，就能利用手繪圖中推得算式求出結果。

```
row_u = floor(x);
if row_u <1
    row_u = row_u + 1;
end
row_d = ceil(x);
if row_d > M
    row_d = row_d - 1;
end
```

```
col_l = floor(y);
if col_l < 1
    col_l = col_l+1;
end
col_r = ceil(y);
if col_r > N
    col_r = col_r-1;
end
```

```
lu = input(row_u, col_l, :);
ld = input(row_d, col_l, :);
ru = input(row_u, col_r, :);
rd = input(row_d, col_r, :);
```

```
a = x - row_u; b = y - col_l;
output(i, j, :) = (1-a) * ((1-b)*lu + b*ru) + a * ((1-b)*ld + b*rd);
```

其中還要注意一開始取的點，我這邊是參照網路上的方法(Stack Overflow)來初始化，做了這樣的操作能確保所有的圖片邊界都會處理到，如下圖。

http://stackoverflow.com/questions/1550878/nearest-neighbor-interpolation-algorithm-in-matlab

```
x = (i-0.5) * (M/zoom_M) + 0.5;
y = (j-0.5) * (N/zoom_N) + 0.5;
```

【結果圖】

(a) Nearest-neighbor interpolation

PSNR: 27.7087

(b) Bilinear interpolation

PSNR: 29.2059

【討論】

(1)兩種方法的比較: 因為做 NN 時，我們是直接強制將原本縮放後小數的點對應到整數點上，從計算上來看就會很明顯地知道這個方法產生的圖片鋸齒狀會很嚴重，尤其是倍率高的時候；而當我們做 Bilinear 時，我們會利用到他四周的四個點來進行內插運算，因此得到的值會較為精確，產生的結果也會較為平滑。兩者共同的是因為都不是取道最精確的點，所以原本顏色會些微跑掉，縮放後的圖片整體看起來都是模糊的。

(2) PSNR 的意義: PSNR 值愈高時，代表圖片的失真愈少，算是一種客觀的評比數據。所以在這一題中，我們可以知道經過 Bilinear 縮放後的 PSNR 應該要比經過 NN 縮放後的 PSNR 還來的高，因為我們算到比較精確的值，因此對於圖片的失真也較小。