Part 1.

本題需要實作的內容可分為三個部分，分別為 DCT、compression、 YIQ 轉換和 PSNR 的計算。

首先，利用 2D 的 DCT 公式，我們將圖片進行離散餘弦變換，透過化簡，繁雜的公式可以變化成右邊簡單的公式方便我們來轉換 matrix。

$$F(u,v) = c(u)c(v)\sum_{i=0}^{N-1}\sum_{i=0}^{N-1} f(i,j)\cos[\frac{(i+0.5)\pi}{N}u]\cos[\frac{(i+0.5)\pi}{N}v]$$

$$F = AfA^T$$

$$A(i,j) = c(i)\cos[\frac{(j+0.5)\pi}{N}i]$$

透過這樣的轉換，我們就可以產生大小為 8*8 的 matrix。

接下來，依據公式，我們要做 A * input * A' 的運算，將原影像藉由 DCT 轉換，從 spatial domain 轉為 frequency domain。IDCT 的部份，則是運算 A'* input * T 來實作。N 值的改變，只是在做 DCT 時需取不同的 upper left 範圍，因此只要稍作改動即可，其餘都是相同的。

在 RGB 與 YIQ 轉換這部分，我們可以根據已知公式來做矩陣轉換即可得到，公式如下：

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.523 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

在計算 PSNR 這部分，我們利用已知的公式，即可完成 PSNR 計算。在計算中要特別注意計算單位我們究竟是使用 double 抑或是 uint；除此之外，影像為 RGB 三維，因此 MSE 是所有維度方差之和除以影像尺寸再除以 3。

$$PSNR = 10 \cdot \log_{10}\left(\frac{MAX_I^2}{MSE}\right) = 20 \cdot \log_{10}\left(\frac{MAX_I}{\sqrt{MSE}}\right)$$

```
[w1,h1,~] = size(img);
d = (img - c2) .^ 2;
MSE = sum(d(:)) / (3 * w1 * h1);
psnr_c2 = log10(1 / MSE) * 10;
```

結果：

(1). RGB，n = 2，PSNR = 27.2584

(a) n=2

(2). RGB，n = 4，PSNR = 35.6371

(a) n=4

(3). RGB，n = 8，PSNR = Inf

(4). YIQ，n = 2，PSNR =27.2584

(5). YIQ，n = 4，PSNR = 35.6371

(6). YIQ，n = 8，PSNR = Inf

觀察與比較：

由肉眼來看，可以發現 n 值不同時，壓縮品質會有差異，隨著 n 值愈大，品質愈好。

比較 PSNR 的結果可以發現，當 n 值愈大時，PSNR 值也愈大，n 愈大代表壓縮比例愈小，與原圖之間的差異也愈小。當 n = 8 時，PSNR 到達 infinite，代表其結果與原圖其實已無差異。

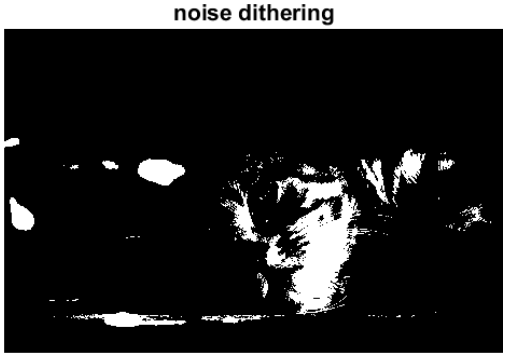
至於 RGB 和 YIQ 之間，以肉眼觀察沒有明顯的差異，計算所得到的 PSNR 值也完全相同，但由理論來看，由於人類的眼睛對於亮度差異的敏感度高於色彩變化，藉由轉換至 YIQ，在實作上可以更有效率地壓縮影像，經過 YIQ 轉換後的圖品質會越好。因此得知， RGB 經過轉換到 YIQ 是做顏色模式的轉換，並不會影響做不同的影像壓縮後的壓縮品質。

Part 2.

(a). noise dithering

noise dithering 又稱為 random dithering，在這裡我的實作方法與講義相同，首先使用 random = unidrnd(256) - 1 來產生 0~255 之間的亂數因子，並比較每個 pixel 的 color value 與 random 的大小，以此來決定此 pixel 的顏色黑或白，透過 for 迴圈跑完就可以得到最終結果。因為是 random 產生 value，因此每次得到的最終 image 會有不同結果。

noise dithering

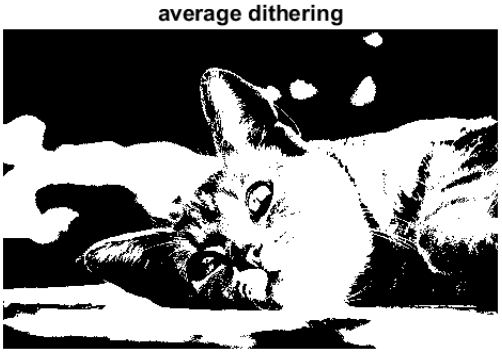


1. Generate a random number from 0 to 255
2. If the pixel's color value is greater than the number then it is white, otherwise black
3. Repeat step 2 for each pixel in the image

(b). average dithering

首先，我們要計算所有 pixel value 的平均，因此利用 sum(origin(:))來得到 total value，並將 total value 除以 image size 來得到 average pixel value；接著我們利用 for 迴圈來比較 average pixel value 與每個 pixel value 的大小，並利用結果來決定 pixel 的顏色黑與白，如此即可完成此 dithering。

average dithering



1. Calculate an average pixel value
2. If a pixel value is above this average, then set it to white, else black
3. Repeat step 2 for each pixel in the image

(c). error diffusion dithering

首先，我們需要 define 一個 mask，而在這裡我是採用跟講義一樣的 masking 方法，如右圖所示。

	p	7
3	5	1

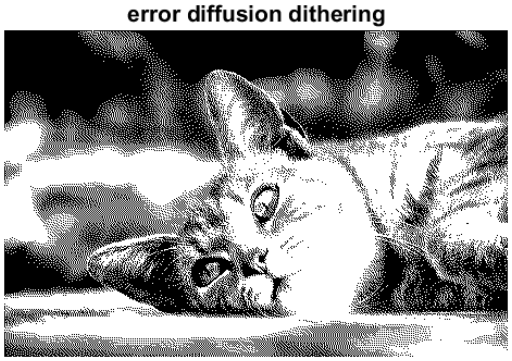
接下來如講義一樣，依據每個 pixel，我們要定義 e 值，並利用以下的四條公式來完成 dithering。

$$p(x+1,y) = p(x+1,y) + (7/16)e$$
$$p(x-1,y+1) = p(x-1,y+1) + (3/16)e$$
$$p(x,y+1) = p(x,y+1) + (5/16)e$$
$$p(x+1,y+1) = p(x+1,y+1) + (1/16)e$$

$$\text{if } p < 128,\ e = p;\ \text{otherwise } e = p - 255$$

在完成所有 pixel 的調整後，利用 for 迴圈跑過所有 pixel value，比較每個 pixel value 與 128 的大小，將 pixel 顏色調整成黑或白，即可完成 error diffusion dithering。

error diffusion dithering



三種方法的比較：

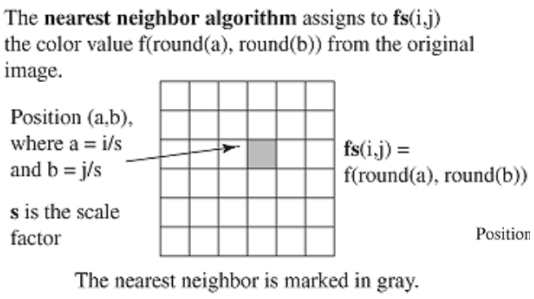
noise dithering 的結果是三種方法中最差的，無法明確地呈現原圖的貓咪輪廓，這也與其簡單的 dithering 運算方法有關；average dithering 是取所有 pixel 的平均與原本個別 pixel value 比較，因此完成 dithering 的結果能呈現原圖的大結構，但是細微部分就會稍微失真，比 error diffusion dithering 效果稍差一些； error diffusion dithering 經過運算後，其顯示的每個 pixel color，會與原圖的 pixel color 保持在一個固定範圍的誤差內，因此最終 dithering 的結果與原圖最為相近，不會有太大的失真問題，圖片細節上的處理也是最好的。

Part 3.

本題題目要求實作三種不同演算法，並利用該演算法將原影像放大 4 倍，再進一步計算放大後的影像與原圖的 PSNR 值。

(a). nearest-neighbor interpolation

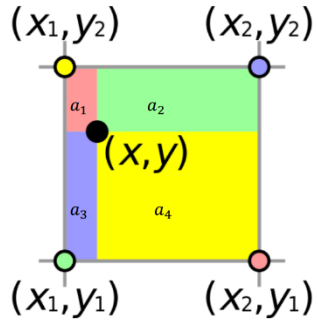
首先，先運用 imread()讀取原圖，並取得原圖的 dimension。依據此演算法的定義與講義的說明，我們可以直接將放大四倍後的點座標取其/s(scale = 4)後的最鄰近座標，並利用 floor()，找出小於該座標值的最大整數點。




```
floor_i = 1 + floor((i-1)/4);
floor_j = 1 + floor((j-1)/4);
i1 = min(w, floor_i);
j1 = min(h, floor_j);
nn_image(i, j, :) = image(i1, j1, :);
```

(b). bilinear interpolation

依據演算法的定義，我們可以利用加權來計算所求值，距離越近的點相對的權重越重，因此對此點的影響會愈大，如此一來，運用線性內插的概念，我們就可以對 x 和 y 兩個方向進行線性內插來求得答案。




```
output(i, j, :) = ...
    (image(x, y, :) * (1 - distance_a) * (1 - distance_b) ...
    + image(x, y + 1, :) * (1 - distance_a) * (distance_b) ...
    + image(x + 1, y, :) * (distance_a) * (1 - distance_b) ...
    + image(x + 1, y + 1, :) * (distance_a) * (distance_b));
```

(c). bicubic interpolation

在這種方法中，函數 v 在點 (x, y)的值可以透過矩形網格中，最近的 4*4 個 neighbor pixel 的加權平均得到，與 bilinear 比較不同的是，在這裡我們需要使用兩個多項式插值三次函數。對每個 pixel 來說，我們會把其上方兩點做三次內插、下方兩點做三次內插，得到兩點再做三次內插，而透過以下公式我們則可以實現其實作。

$$v(x,y) = \sum_{i=0}^{3}\sum_{j=0}^{3} a_{ij} P_{ij}$$

$$a_{ij} = a_i * b_j$$

$$a_i = \prod_{k=0,\ k\neq i}^{3} \frac{(x-ceil(s*(c+k)))}{ceil(s*(c+i))-ceil(s*(c+k))}$$

$$b_j = \prod_{k=0,\ k\neq j}^{3} \frac{(y-ceil(s*(r+k)))}{ceil(s*(r+j))-ceil(s*(r+k))}$$

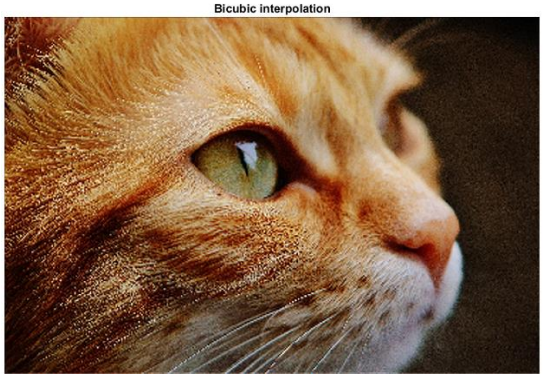
```
Matrix = [  (i-m2)*(i-m3)*(i-m4)/((m1-m2)*(m1-m3)*(m1-m4)) ...
            (i-m1)*(i-m3)*(i-m4)/((m2-m1)*(m2-m3)*(m2-m4)) ...
            (i-m1)*(i-m2)*(i-m4)/((m3-m1)*(m3-m2)*(m3-m4)) ...
            (i-m1)*(i-m2)*(i-m3)/((m4-m1)*(m4-m2)*(m4-m3))];
```

比較：

(a). PSNR = 19.7314

(b). PSNR = 20.1120

(c). Bicubic，PSNR = 19.6347



以肉眼來看的話，使用 bilinear 所得到的放大結果會比 nearest-neighbor 或 Bicubic 更為優異。nearest-neighbor 所得的圖片中，顏色交界處產生明顯的鋸齒狀，較為不自然，這是因為它的演算法容易，所有的放大後的 4*4 方格都會是同一個 pixel value。反之，在 bilinear 中，色彩分布的十分均勻，無任何鋸齒狀的人工效果，可以說是十分成功的放大結果。
理論上，使用 Bicubic 的放大結果會比 bilinear 與 nearest-neighbor 來得好，因為他是做了雙三次插值的結果。(理論結果的 PSNR 大小應該依序為 nearest-neighbor < bilinear < Bicubic)。但依據我自己實作後的實際上 PSNR 計算結果，可以發現 nearest-neighbor 與 Bicubic 的結果差不多，而 bilinear 相對來說較好，我覺得這部分可能是因為自己在演算法的運算時不夠細緻產生了誤差所導致，導致 PSNR 沒有預期中的高。

參考資料：
https://thilinasameera.wordpress.com/2010/12/24/digital-image-zooming-sample-codes-on-matlab/
http://deep-free.blogspot.tw/2012/06/2d-dct.html