Homework 1 Report

1. Method

這次作業使用了 numpy 來做矩陣運算，以及 opencv 來讀圖。

- Rotate Image

我定義了一個 function 來旋轉圖片：

```
def get_rotmtx(deg):
    ang = deg * np.pi/180
    return np.array([[np.cos(ang), -np.sin(ang)],
                     [np.sin(ang), np.cos(ang)]])

def rotate_image_cw(image, deg, method='nearest'):
    if method == 'nearest':
        method = nearest
    elif method == 'bilinear':
        method = bilinear
    elif method == 'bicubic':
        method = bicubic

    h, w, c = image.shape
    new_image = np.zeros((h, w, c), dtype=np.uint8)

    x_center, y_center = w//2, h//2
    rotmtx = get_rotmtx(-deg)
    for yy in range(h):
        for xx in range(w):
            x, y = rotmtx @ np.array([xx - x_center, yy - y_center]) + \
                            np.array([x_center, y_center])
            if 0 <= y < h and 0 <= x < w:
                new_image[yy, xx] = method(image, x, y)

    return new_image
```

其中計算新圖片坐標轉換到原圖片坐標的方式，我是先將軸移到中點，然後用旋轉矩陣算出該點應該會在原圖的哪個點（因為 y 軸方向相反，所以角度要加個負號）。如果算出來的點在原圖的範圍內，就去取值，反之就不處理（留黑）。

- Upscale Image

我也定義了一個 function 來處理放大圖片的部分：

```
def upscale_image_2x(image, method='nearest'):
    if method == 'nearest':
        method = nearest
    elif method == 'bilinear':
        method = bilinear
    elif method == 'bicubic':
        method = bicubic

    h, w, c = image.shape
    new_image = np.zeros((h*2, w*2, c), dtype=np.uint8)

    for yy in range(h*2):
        for xx in range(w*2):
            new_image[yy, xx] = image[yy//2, xx//2] if xx%2 == 0 and yy%2 == 0 else
method(image, xx/2, yy/2)

    return new_image
```

放大的部分比較簡單，可以先判斷對應的原坐標是不是整數，如果是的話就可以直接取值，反之就使用指定的方法來算出值來。

- Nearest Neighbor Interpolation
  這方法很簡單，只要把坐標取整再去取值即可（照理來說要四捨五入，但這樣會導致最後一個點取不到值，因此我是直接取整，這樣出來的圖片就是剛好一個點複製四次）。

```
def nearest(image, x, y):
    return image[int(y), int(x)]
```

- Bilinear Interpolation
  套用線性內插方法兩次就可以得到想要的效果，其中因為左右兩個點的差值都是 1，所以可以不用除。

```
def bilinear(image, x, y):
    h, w = image.shape[:2]
    x0, y0 = int(x), int(y)
    x1, y1 = x0 + 1, y0 + 1

    return (image[y0, x0] * (x1 - x) * (y1 - y) +
            image[y0, x1] * (x - x0) * (y1 - y) +
            image[y1, x0] * (x1 - x) * (y - y0) +
            image[y1, x1] * (x - x0) * (y - y0)).round().astype(np.uint8) \
    if x1 < w and y1 < h else image[y0, x0]
```

- Bicubic Interpolation
  套用助教給的公式，每個點需要四個相鄰的點來近似曲線，然後要取兩次，因此需要 16 個點來獲得近似值。這程式使用了矩陣乘法來加速計算。

```
def bicubic(image, x, y):
    def cubic_interpolation(p, x):
        coef = np.array([[-1/2,  3/2, -3/2,  1/2],
                         [1   , -5/2,    2, -1/2],
                         [-1/2,    0,  1/2,    0],
                         [0   ,    1,    0,    0]])

        x = np.array([x**3, x**2, x, 1])

        return (coef @ p).T @ x

    def get_points_x(img, y, x0, x1, x2, x3):
        return np.array([img[y, x0], img[y, x1], img[y, x2], img[y, x3]],
                        dtype=np.int32)

    h, w = image.shape[:2]

    計算 x0~x3, y0~y3 以及邊界條件判斷…

    dx,dy = x - x1, y - y1

    p = get_points_x(image, y0, x0, x1, x2, x3)
    q0 = cubic_interpolation(p, dx)

    p = get_points_x(image, y1, x0, x1, x2, x3)
    q1 = cubic_interpolation(p, dx)

    p = get_points_x(image, y2, x0, x1, x2, x3)
    q2 = cubic_interpolation(p, dx)

    p = get_points_x(image, y3, x0, x1, x2, x3)
    q3 = cubic_interpolation(p, dx)

    return np.clip(cubic_interpolation(np.array([q0, q1, q2, q3]), dy), 0,
255).astype(np.uint8)
```

2. Result

在經過旋轉的圖片中，可以發現使用 Nearest Neighbor 會有很明顯的鋸齒狀邊緣，而 Bilinear 跟 Bicubic 則會好很多，仔細看可以發現 Bicubic 會更平滑一點。

		
Nearest Neighbor	Bilinear	Bicubic

經過兩倍放大的圖片也可以得到差不多的結果，使用 Nearest Neighbor 會有較明顯的顆粒感，邊緣的部分也有一些割裂感，使用 Bilinear 跟 Bicubic 則好很多。

		
Nearest Neighbor	Bilinear	Bicubic

3. Feedback

經由這次的作業，我對影像處理的操作更加熟悉，也更能理解作業用到的幾種插值法到底是怎麼實作出來的。之前玩遊戲的時候，總是會在畫質設定看到“材質過濾”這個選項，但直至現在我才知道背後的原理，但似乎從來沒看到 Bicubic 的選項，看來在效能跟效果上還是各向異性過濾會比較好一些。

這次的作業我也在做 Bicubic interpolation 的時候遇到了一個 bug：取點的時候如果不先把點的型態換成 int32，後面在做運算的時候就會爆掉，出來的圖片就會很奇怪。不過當我把計算改成用矩陣運算的時候發現就算不換型態也不會出事。之後在寫這種算數的程式的時候要小心資料型態。