數位影像處理 HW1

R11942137 張舜程

Problem 1 - Prove

$$T(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y)\, r(x,u,y,v)$$

$\because r(x,u,y,v)$ *is a* ***sperable*** *and* ***symmetric*** *kernel*
$\therefore r(x,u,y,v) = r_1(x,u)\, r_2(y,v)$

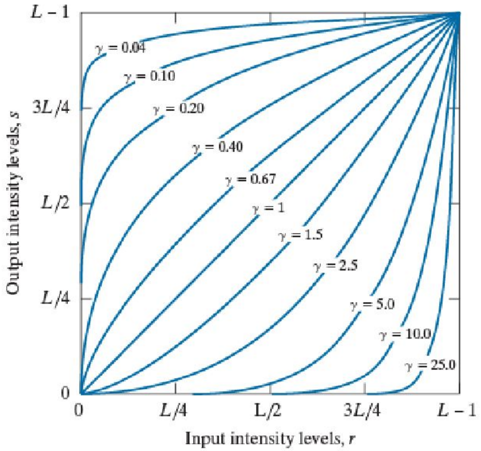
$$T(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y)\, r_1(x,u)\, r_2(y,v)$$

$$= \sum_{x=0}^{M-1} f(x,y)\, r_1(x,u) \sum_{y=0}^{N-1} r_2(y,v)$$

$$= T(u,y) \sum_{y=0}^{N-1} r_2(y,v)$$

$$= \sum_{y=0}^{N-1} T(u,y) r_2(y,v)$$

Problem 2 - Gamma Correction(GC)



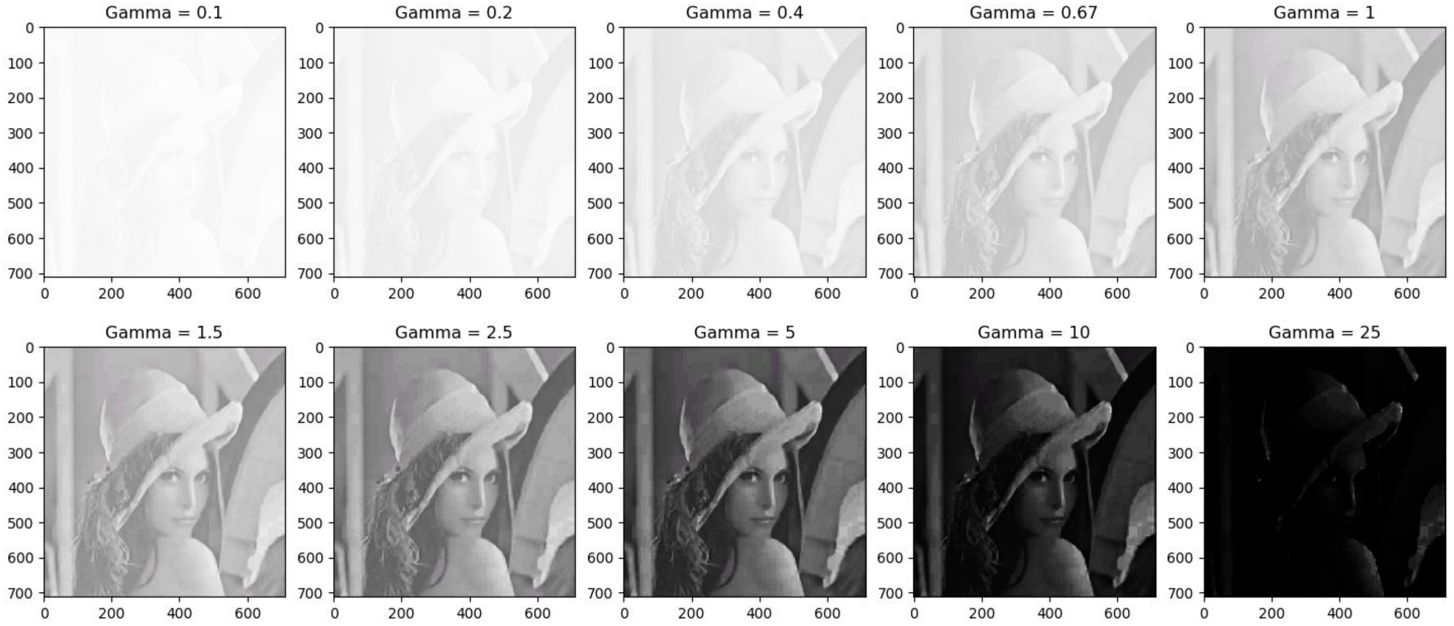
$$s = cr^{\gamma}$$

- 依照講義公式刻:

```python
def GC(img, gamma):
    intensity = img/float(np.max(img))
    result = np.power(intensity, gamma)
    return result
```

Problem 2 - Gamma Correction(GC)

- Result 在不同 gamma 值下的結果



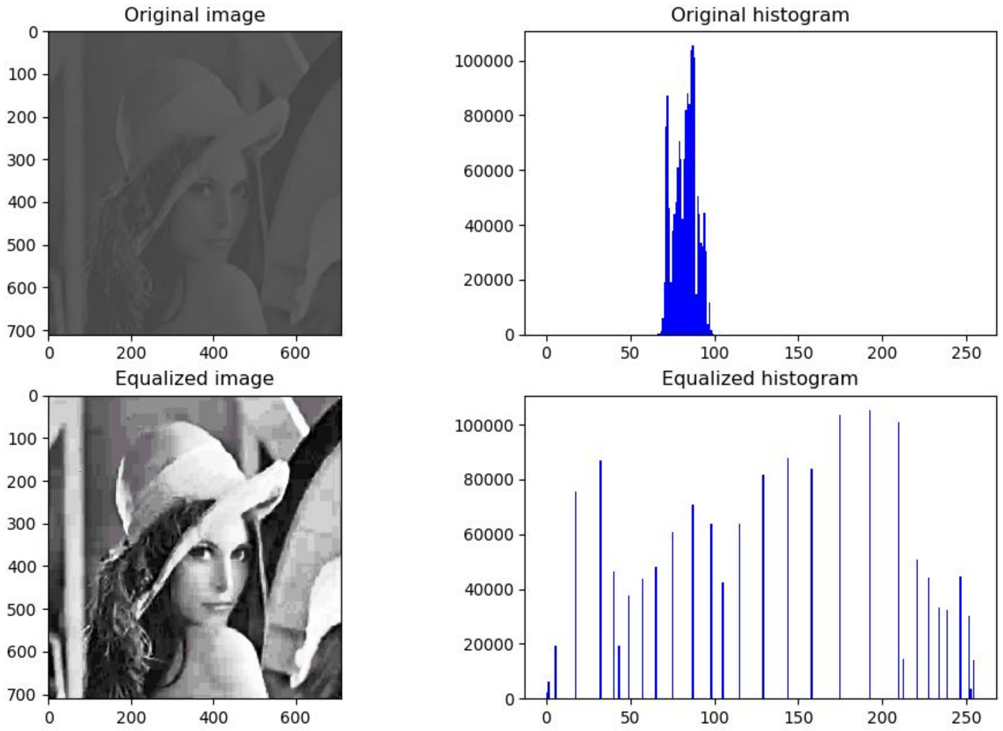
Problem 2 - Histogram Equalization(HE) 實作

```python
def HE(img):
    hist, bins = np.histogram(img.ravel(), 256, [0, 255])
    pdf = hist/img.size
    cdf = pdf.cumsum()
    equ_value = np.around(cdf * 255).astype('uint8')
    result = equ_value[img]
    return result
```

先用 numpy 的 histogram，去統計此張 image 各 pixel 值出現的次數
同除以總 pixel 數量，得到 PDF
再用 .cumsum 算出 CDF
最後將CDF映射到色彩空間(在這邊是用0~255灰階) 變成新的 pixel 值
這麼做可以讓新的相片的色彩分布呈現 **均勻分布**

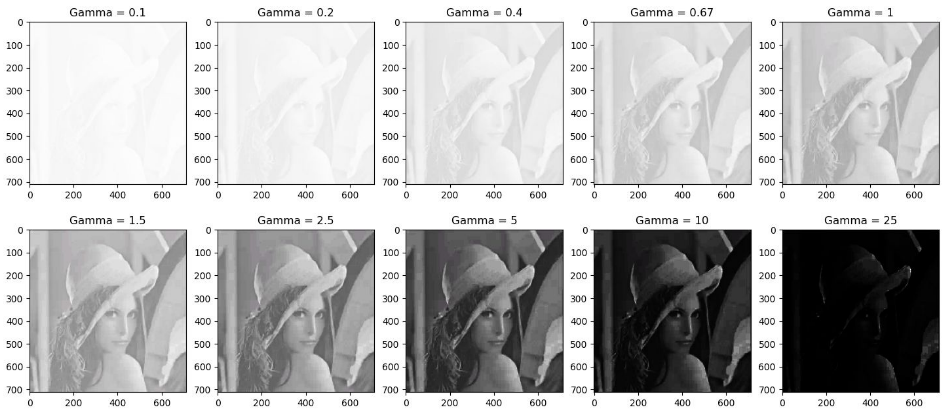
Problem 2 - Histogram Equalization(HE)

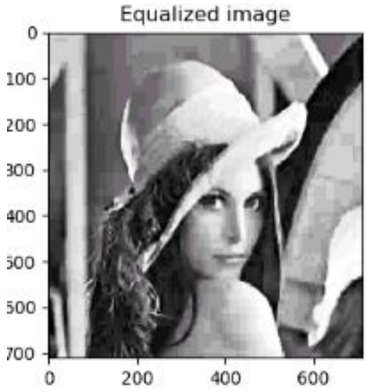
- Result



Problem 2 - What is the gamma value that leads to minimal difference between the results of these two methods?

Gamma = 5





Problem 3 - Image Interpolation

我實作了三個 function:

```
def nearest_interpolation(image, scale, degree):
    ...
def bilinear(image, scale, degree):
    ...
def bicubic(image, scale, degree):
    ...
```

在主程式中，分別用三種 Interpolation 來實現**旋轉(-45度)**，再**縮小 0.7 倍**。

Problem 3 - Image Interpolation

```
def nearest_interpolation(image, scale, degree):

    rads = math.radians(degree)

    image_width = image.shape[0]
    image_height = image.shape[1]

    resized_image_width = math.floor(image_width * scale)
    resized_image_height = math.floor(image_height * scale)

    midx, midy = (resized_image_width // 2 + 40, resized_image_height // 2 + 20)

    image_output = np.zeros((resized_image_width, resized_image_height), dtype=np.uint8)
    image_output.fill(255)
```

根據縮放倍率(scale), 計算影像大小的一些參數, 並且先製作一個 **全白的canvas**
另外, 也根據 degree 算出影像要旋轉的角度(radians)
還有算出旋轉的中心點 midx, midy

Problem 3 - Image Interpolation

```python
for x in range(resized_image_width):

    x_ori = (x / resized_image_width) * image_width
    x_interp = x_ori - np.floor(x_ori)

    # find the nearest neighbour of current x
    if x_interp < 0.5:
        x_int = int(np.floor(x_ori))
    else:
        x_int = int(np.ceil(x_ori))
        # if x_int is out of bound force it back inbound
        if x_int >= image_width:
            x_int = int(np.floor(x_ori))

    for y in range(resized_image_height):
        y_ori = (y / resized_image_height) * image_height

        y_interp = y_ori - np.floor(y_ori)

        # find the nearest neighbour of current y
        if y_interp < 0.5:
            y_int = int(np.floor(y_ori))
        else:
            y_int = int(np.ceil(y_ori))
            # if y_int is out of bound force it back inbound
            if y_int >= image_height:
                y_int = int(np.floor(y_ori))
```

利用 double for loop 對整個 image 做 interpolation

過程中使用 numpy 的 **floor** 和 **ceil** 函數來找出距離縮小後的pixel最近的點

Problem 3 - Image Interpolation

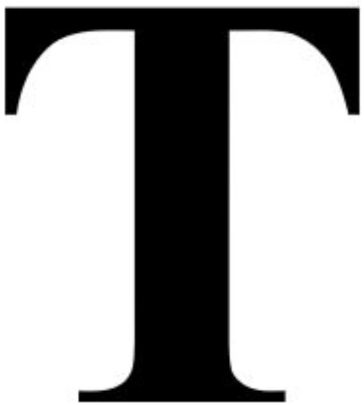
```python
        X_int = (x_int - midx) * math.cos(rads) + (y_int - midy) * math.sin(rads)
        Y_int = -(x_int - midx) * math.sin(rads) + (y_int - midy) * math.cos(rads)
        X_int = round(X_int) + midx
        Y_int = round(Y_int) + midy
        if (X_int >= 0 and Y_int >= 0 and X_int < image.shape[0] and Y_int < image.shape[1]):
            image_output[x, y] = image[X_int, Y_int]

return image_output
```

最後，再把剛才縮小完的 pixel，做**旋轉矩陣**的運算，得到最後的 值
並且 assign 到新的影像上

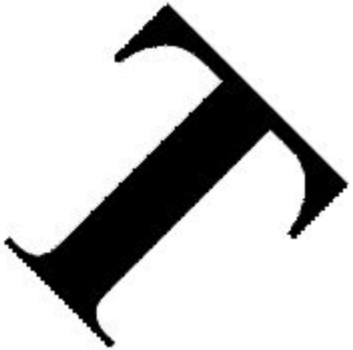
Bilinear 與 Bicubic 的做法也大致相同，差在 assign新的pixel時的做法不同，詳見程式碼

Problem 3 - Image Interpolation

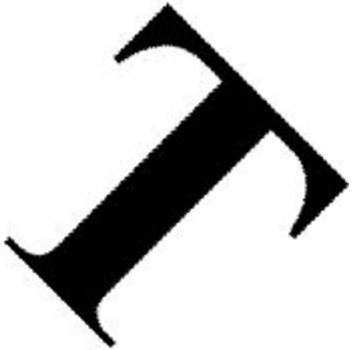
- Result



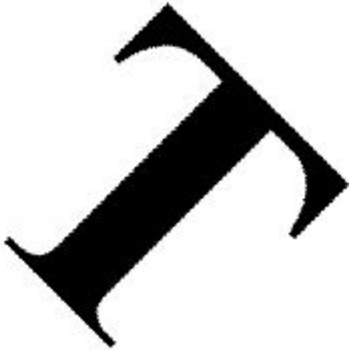
Origin



nearest



bilinear



bicubic

觀察發現 **Bicubic 的效果最好**，最差的為 nearest

Problem 4 - Shading correction

我實作了兩個 function:

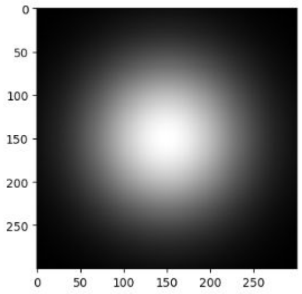
- `gaussian_kernel(l=300, sigma=64)`
- `conv(target, kernel, stride=1)`

實作上，以高斯濾波器作為**低通濾波器**用來濾除棋盤格的高頻成分。
為了取得 Shading Pattern，我嘗試了很多次的參數，以**300x300的 Kernel** 為結果。
另外，我是採用 **Zero Padding** 的方式來做卷積。

Problem 4 - Gaussian Kernel 實作

```python
def gaussian_kernel(l, sigma):
    ax = np.linspace(-(l-1)/2., (l-1)/2., l)
    gauss = np.exp(-0.5*np.square(ax)/np.square(sigma))
    kernel = np.outer(gauss, gauss)
    return kernel / np.sum(kernel)
```

因為高斯函數是**對稱函數**，根據 Problem1 的經驗，我製作了兩個一維的高斯函數，之後再對他們兩個做**外積**，就可以得到二維的高斯濾波器。



Problem 4 - Convolution 實作

```python
def conv(image, kernel, stride=1):
    if kernel.shape[0] % 2 == 1:
        padding = kernel.shape[0] // 2
    else:
        padding = kernel.shape[0] // 2 + 1

    target_shape = image.shape
    kernel_shape = kernel.shape
    result = np.zeros_like(image)


    padding_left = np.zeros((image.shape[0], padding), np.float32)
    padding_right = padding_left.copy()
    print(image.shape, padding_right.shape, padding_left.shape)
    image = np.concatenate((padding_left, image, padding_right), 1)

    padding_top = np.zeros((padding, image.shape[1]), np.float32)
    padding_bottom = padding_top.copy()
    image = np.concatenate((padding_top, image, padding_bottom), 0)

    for i in range(0, target_shape[0], stride):
        for j in range(0,target_shape[1], stride):
            window = image[i:i + kernel_shape[0], j:j + kernel_shape[1]]
            val = window * kernel
            result[i,j] = val.sum()

    return result
```

Determine some size

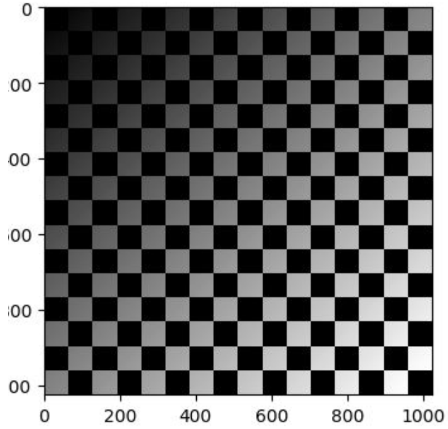
Zero Padding

Convolution

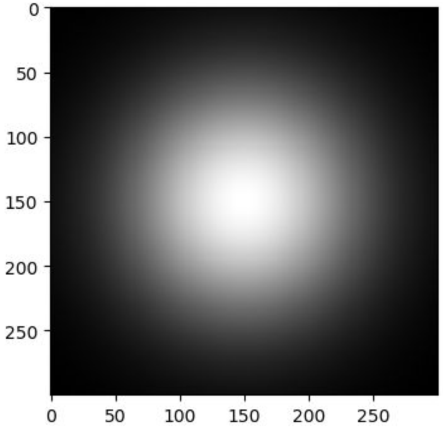
Problem 4 - Shading correction

- Result

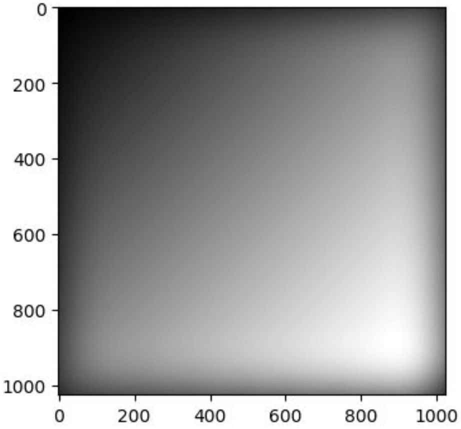
Oringial



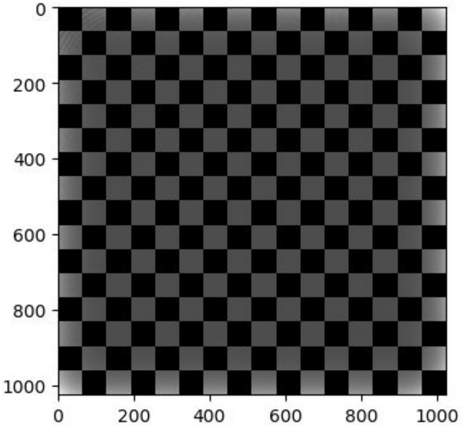
Kernel



shading pattern

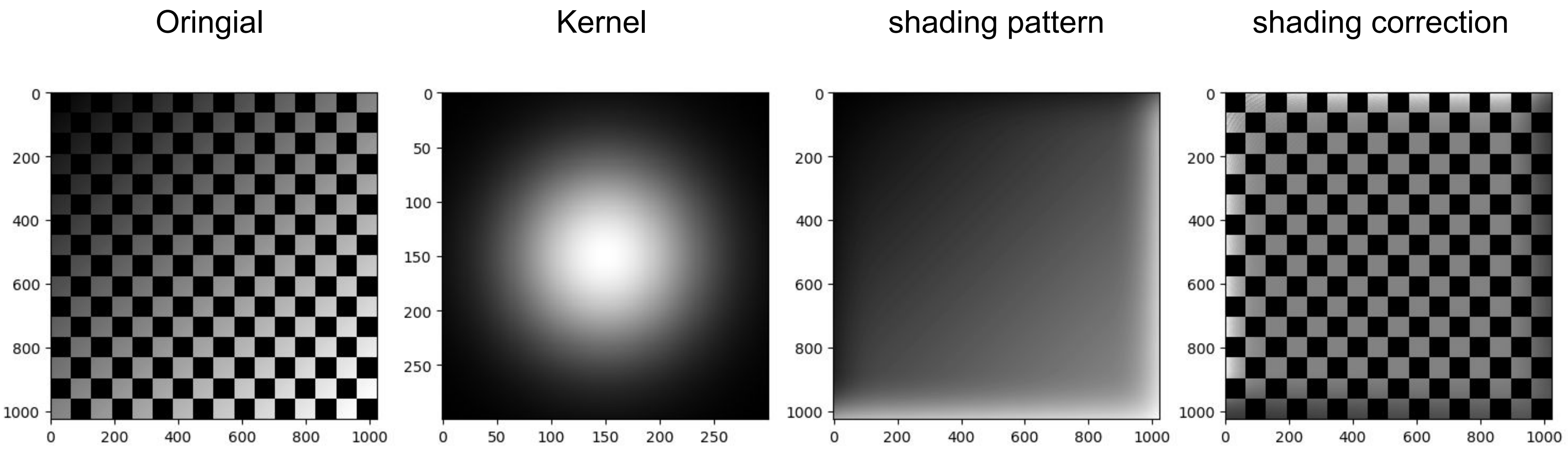


shading correction



Problem 4 - Shading correction

- 嘗試**右邊**和**底部**的padding改成**白色**



Problem 4 - Shading correction

因為 Padding 的關係讓右下角多了一塊很大的黑色角落。

高斯濾波器在卷積時，會把這個黑色角落加進去一起計算。

所以 Shading Pattern 的邊邊角角會是暗色的。