

## Problem1(a)

一、Output :

檔名: resize&rotate\_nn.png, resize&rotate\_bilinear.png, resize&rotate\_bicubic.png

大小為原本的 0.6 倍，再轉 15 度 ( 先 resize 再 rotate )

resize&rotate\_bicubic.png



resize&rotate\_bilinear.png



resize&rotate\_nn.png



原本給的 T\_transformed.png

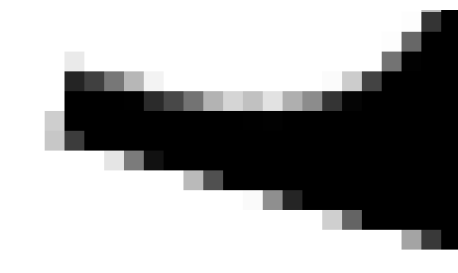


## 二、結果分析：

Bicubic



Bilinear



Nearest neighbor



可以看出雖然 Bicubic 和 Bilinear 的效果差不多，但是相較 Nearest neighbor(NN)就好多了，NN 的鋸齒感特別明顯。

## 三、各 interpolation 的計算方法(詳細實作都寫在 code 註解)

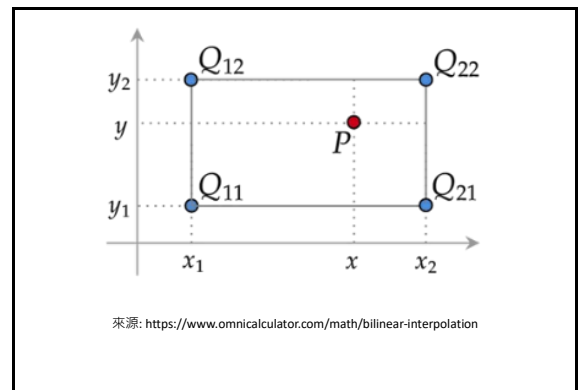
### 1. Nearest neighbor

若新圖的某 pixel  $P$  落在舊圖 pixel

$Q_{11}, Q_{12}, Q_{21}, Q_{22}$  之間，則取距離  $P$  最近的那個

pixel 的值作為 pixel  $P$  的值，在這個例子就是右上

角的  $Q_{22}$ 。



### 2. Bilinear

先對水平方向(x axis)做一次 linear interpolation，再

對垂直方向(y axis)做一次 linear interpolation。公式

裡的  $Q_{11}, Q_{12}, Q_{21}, Q_{22}$  一樣是前面那四個 pixels。

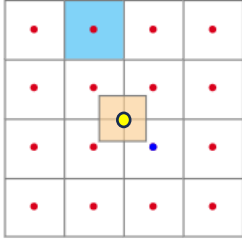
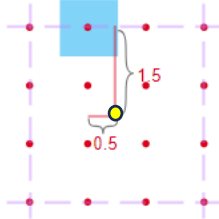
$$R_1 = \frac{x_2 - x}{x_2 - x_1} Q_{11} + \frac{x - x_1}{x_2 - x_1} Q_{21}$$

$$R_2 = \frac{x_2 - x}{x_2 - x_1} Q_{12} + \frac{x - x_1}{x_2 - x_1} Q_{22}$$

$$P = \frac{y_2 - y}{y_2 - y_1} R_1 + \frac{y - y_1}{y_2 - y_1} R_2$$

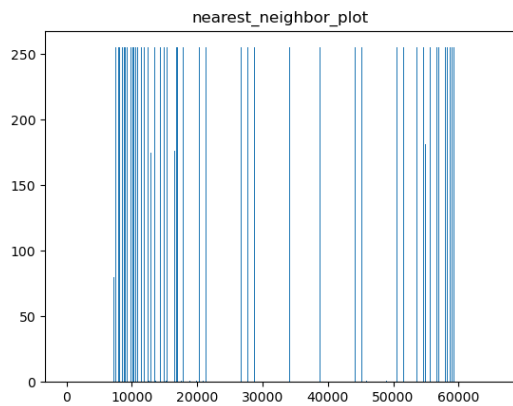
來源: <https://www.omnicalculator.com/math/bilinear-interpolation>

### 3. Bicubic 來源: <https://y7snki.axshare.com/?id=sesz04&p=%E5%8F%8C%E4%B8%89%E6%AC%A1%E6%8F%92%E5%80%BC%E6%B3%95&sc=3>

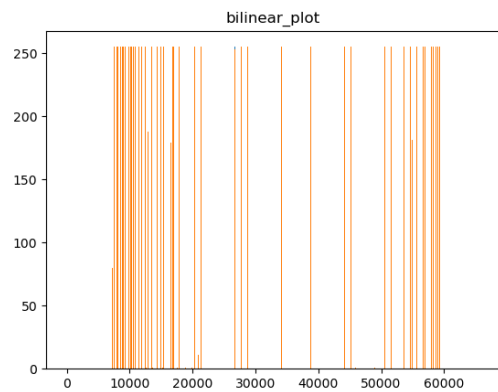
|   |   |
|---|---|
| <p>(a) 令紅點是舊圖的 pixel，灰線 grid 是新圖的 pixel。現在要用這 16 個紅點去計算出黃點 (new pixel 的值)。</p>  |   |
| <p>(b) 取個紅點與黃點的距離，舉例來說：黃點與藍底的那個紅點，x 方向的距離為 0.5，y 方向的距離為 1.5 (這裡都以 pixel 為單位)。</p>   |   |
| <p>(c) Apply 到每個紅點後，會得到一組 16 個數字的 x 軸距離，與一組 16 個數字的 y 軸距離，分別把這兩組數據帶入 function <math>W</math>，並各自存成一個 array <math>weight\_x</math> 和 <math>weight\_y</math>。</p> | $W(x) = \begin{cases} (a+2) x ^3 - (a+3) x ^2 + 1 & \text{for }  x  \leq 1 \\ a x ^3 - 5a x ^2 + 8a x  - 4a & \text{for } 1 \leq  x  \leq 2 \\ 0 & \text{others} \end{cases}$ <p>x: 上一步算的距離</p> |
| <p>(d) 得到兩個權重矩陣後，做 Hadamard product：紅點值 * <math>weight\_x</math> * <math>weight\_y</math>，最後把乘完後的矩陣每個值加起來，就會得到黃點應該填入的值。</p>                                     | $p(x, y) = \sum_{i=0}^3 \sum_{j=0}^3 a_{ij} x^i y^j.$   |

#### 四、Interpolation 和原圖的比較分析 Output：

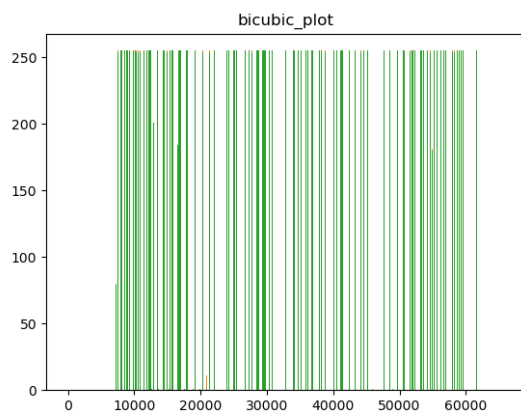
nearest\_neighbor\_plot.png



bilinear\_plot.png



bicubic\_plot.png



#### 五、做法：

X 軸為每個 pixel index (input size=256\*256，所以共有 65536 個 pixels)，

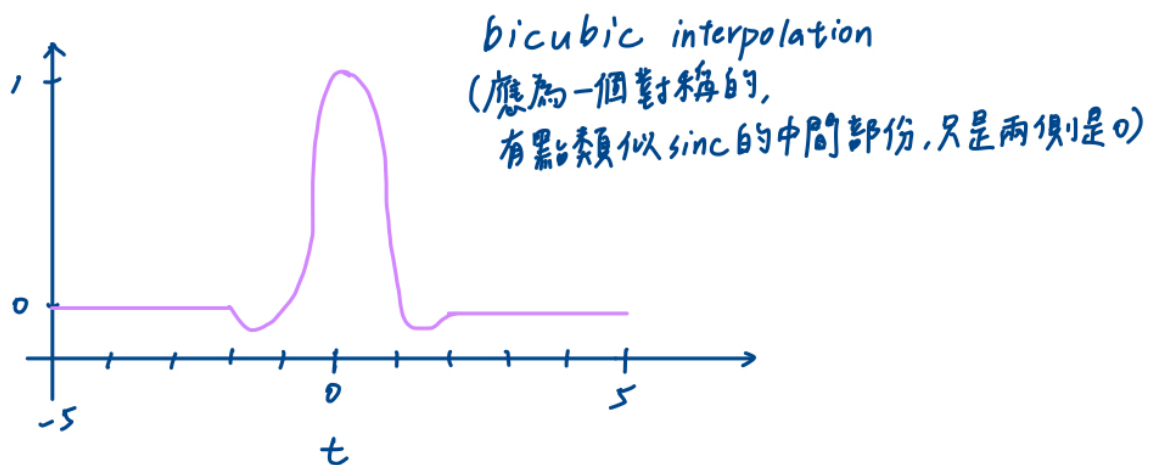
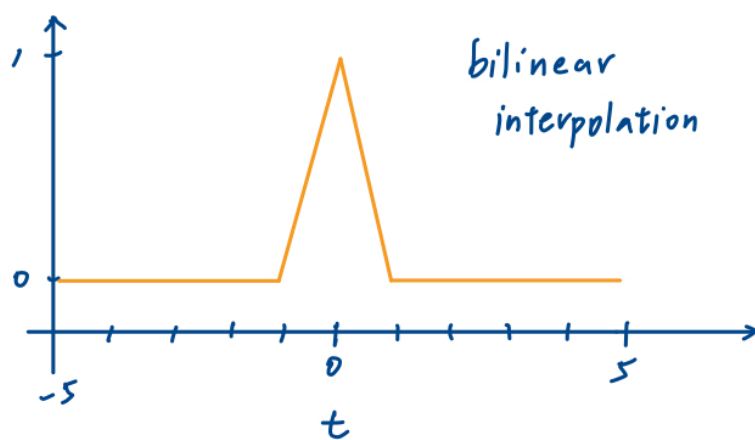
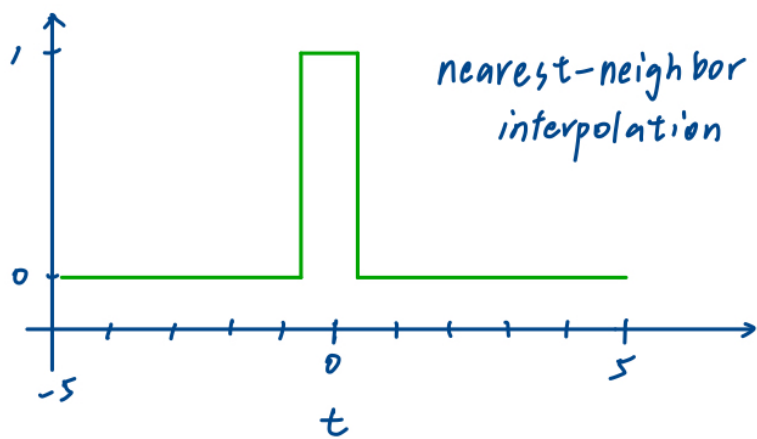
Y 軸為 $(\text{做完 interpolation 的 pixel value}) - (\text{原始圖片的 pixel value})$ ，介於 0~255

#### 六、分析：

bicubic 的 bar 數明顯比其他兩個多，我認為是因為它在做 bicubic 的時候，參考的點多很多，所以出來的 pixel value 也很容易和原始圖片不一樣。因為我們的 input 黑和白的地方都挺集中，所以 NN 和 bilinear 才會比較集中在兩側。

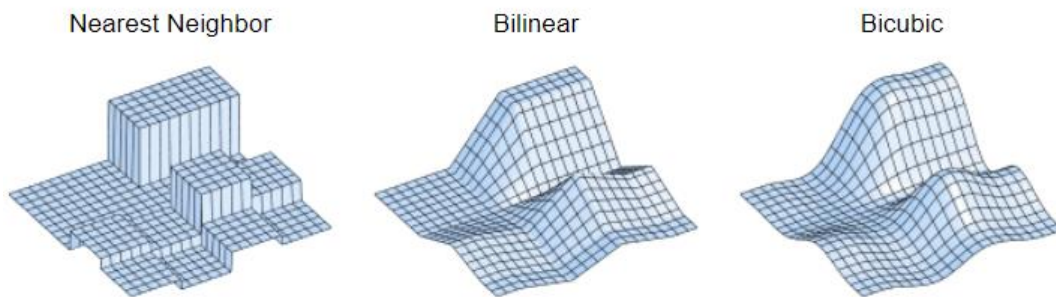
## Problem 1(b)

一、 impulse response 圖



這些圖都是 based on problem 1a 寫的那些公式畫的。

## 二、分析



*Nearest neighbor, bilinear, and bicubic applied to the same uniformly-spaced input data.*

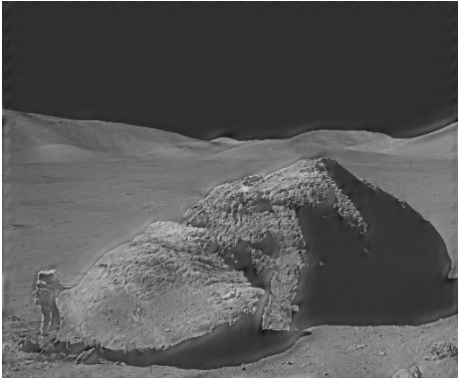
來源: [http://www.ipol.im/pub/art/2011/g\\_lmii/revisions/2011-09-27/g\\_lmii.html](http://www.ipol.im/pub/art/2011/g_lmii/revisions/2011-09-27/g_lmii.html)

Impulse response 之所以會長得像第一部分那樣，也可以從這張圖看出來。在這張圖做垂直的 slicing，就可以得到類似第一部分的結果。Nearest-neighbor interpolation 是像 0 or 1 的 signal 那樣長方形的，bilinear 是類似三角形的，bicubic 是類似 sinc 的曲線，但左右測是 0(曲線結果會因取的 a constant 而異，若 a 取-0.5 會最接近 sinc)。

## Problem 2

### 一、Output：

檔名: recover\_filtered\_img.png



Note: 程式執行時間約需 3 分鐘左右

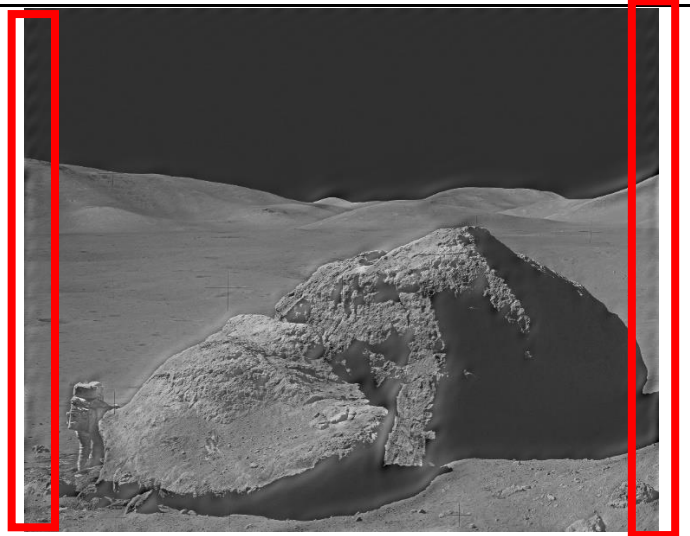
### 二、結果分析：

#### 現象 1:

左右兩邊有些微柵欄的痕跡沒有辦法消除掉。(詳細的可看後面的**補充 1**)

我猜測的原因:

因為 padding 對左右側只有 pad 少許 pixel，上下側就 pad 很多，所以比較不明顯

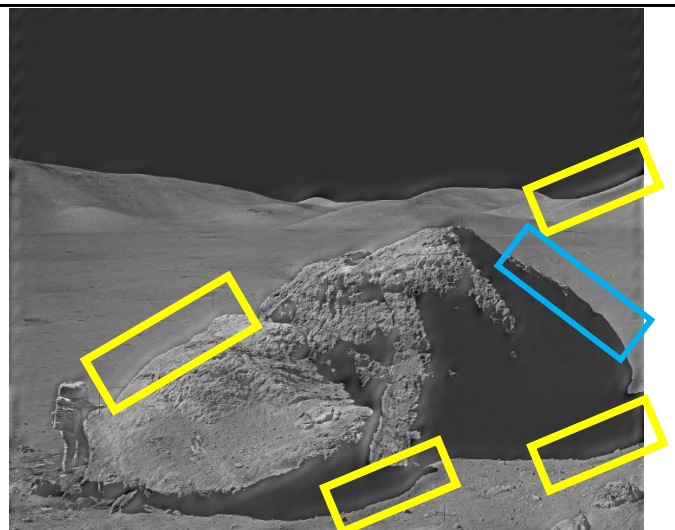


#### 現象 2:

部分地方的邊緣有點糊糊的(黃框部分)

我猜測的原因:

因為這些黃框部分的 edge 剛好和原本的柵欄是同一個方向，所以在做 filter 後會受到影響。這也解釋了為什麼藍框的 edge(和柵欄反向)沒有糊糊的。



### 三、計算過程：

1. 對原圖(astronaut\_interference.tif)做 padding，padding 後的圖片大小為 1024\*1024(取最接近的 2 的次方)，我在這裡有寫兩個版本的 padding：

- i. Wrap padding (相當於 `cv2.copyMakeBorder(..., borderType=cv2.BORDER_WRAP)`):

採用 wrap padding 的原因: 因常見的 padding 方法中，只有這個能延伸我們的柵欄，而不要讓它反向。若用其他方法邊邊會有反向的線條，如下圖



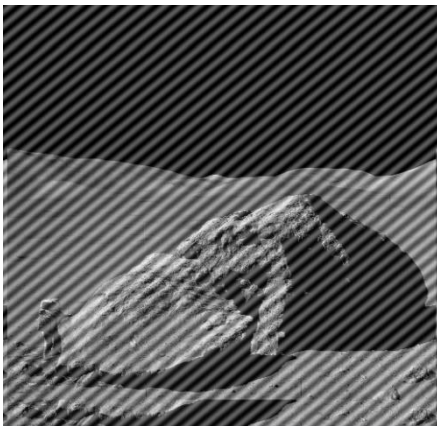
以 replicate padding 為例，做完 filtering 後會殘留 padding 完的反向柵欄(紅框)，見補充 2。



小圖是

Wrap padding 的結果

- ii. Customized padding:



依照完全依照 astronaut\_interference.tif 這張圖做客製化

padding，Padding 結果如左圖，詳細方法寫在 code 註解

優點: 上面的黑邊直接延伸，下面也是較多白色部分做延

伸。程式預設用這個，可在 line 251 改成標準 warp pad

2. 做 FFT2D (相當於 `np.fft.fft2`): 二維的 fft 相當於做兩次 fft，分別在不同維度

原本是手刻二維 dft，但發現 complexity 太高要跑超級久。

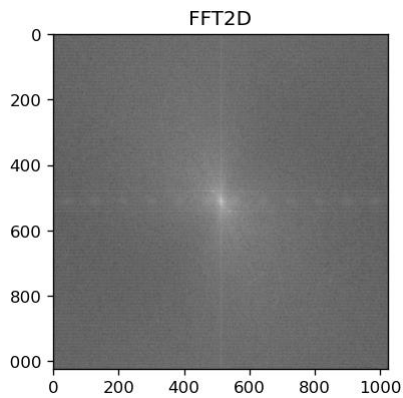
3. 做 FFTSHIFT (相當於 `np.fft.fftshift`): 把圖片切四塊做象限交換

目標: block 1, 3互換, block 2, 4互換

|   |   |   |   |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 4 | 3 | 2 | 1 |

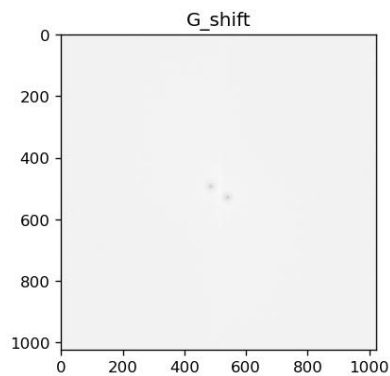
->





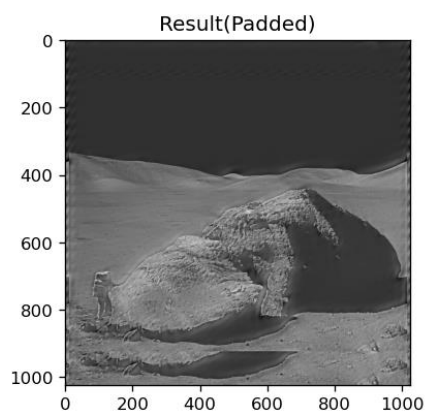
得到的 magnitude\_spectrum 如左圖

4. 把它和 butterworth notch filter 相乘，結果如下

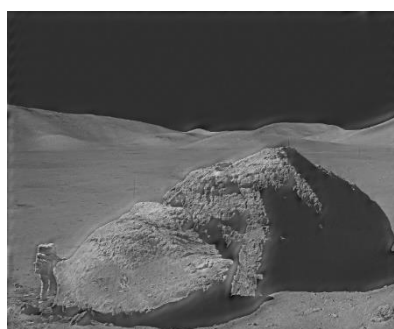


Butterworth 畫出來就是中間的那兩黑點，和前面的 fft 做結合。

5. 做 IFFT\_SHIFT (與 step3 的相反，相當於 `np.fft.ifftshift`)
6. 做 IFFT2D (相當於 `np.fft.ifft2`)
7. 得 crop 前的結果如下



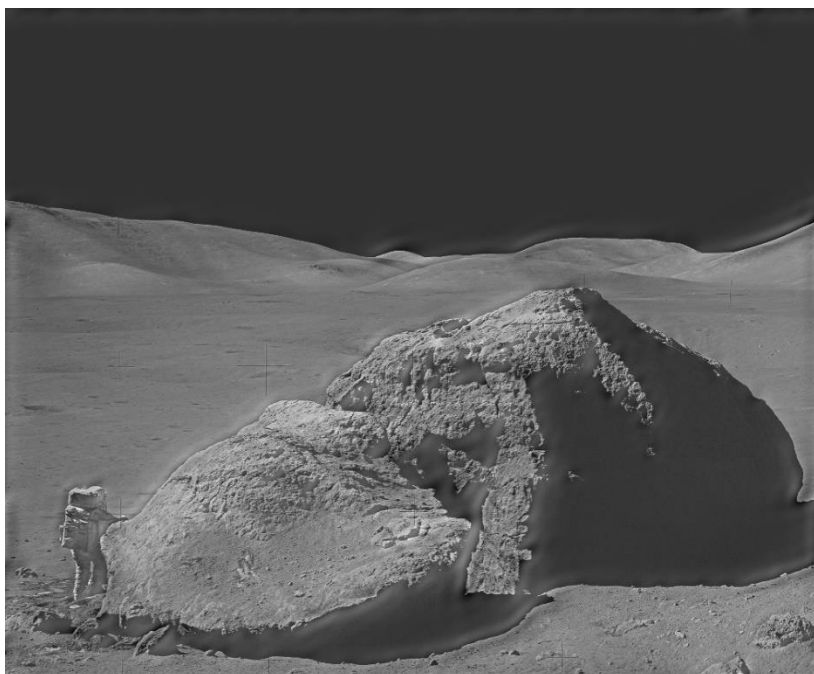
8. 再把這個 crop 成原圖的 1000\*824，並 normalize (單純是讓結果好看一點)
9. 最終 output 如右圖



## 補充 1: 與 numpy 的差異

為了確保手刻的 `fft`, `fftshift`, `iff`, `ifftshift` 都和 `numpy` 提供的一樣，我有寫一個版本是全部用 `numpy` 去 `call` 的 (程式碼在 p2 最底下的註解)，然後把我手刻的 `output` 和 `numpy` 的做比較，使用 `np.allclose()` 去驗證兩者的差距，各步驟得到的 `allclose()` 都是 `true`。

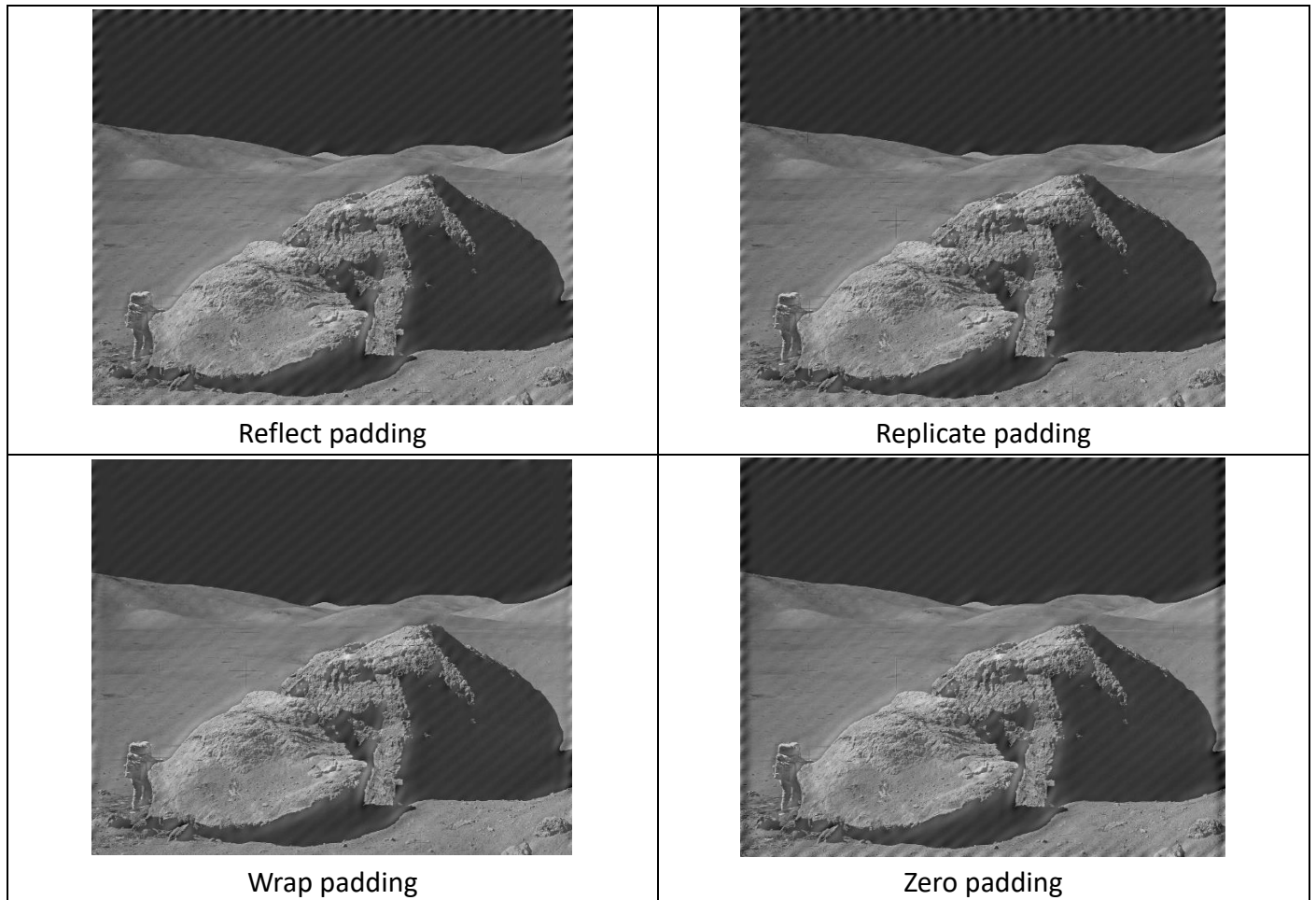
這是我跑 `numpy` 的結果，可以看到相比手刻的，它的左右側少了柵欄的痕跡，這是因為 `numpy` 的 `fft` 不需要做 `padding`，直接丟入原圖即可，所以手刻版的 `output` 左右側的柵欄殘留可以推測是因為 `padding` 導致。



## 補充 2: 與不同 padding 方式的比較

### 一、Filter 完的結果結果

Note: 這裡為了要凸顯 padding 的影響，所以 notch filter 參數設得和我前面的 output 不太一樣。



### 二、結果分析

Reflect 和 replicate 的上側都有都有反向柵欄的痕跡

Zero padding 的上側柵欄次份比 wrap 明顯，左右側的黑邊也比 wrap 嚴重。

## 參考資料

1. <https://www.omnicalculator.com/math/bilinear-interpolation>
2. <https://reurl.cc/2EW3za>
3. [https://www.bilibili.com/video/BV1wh411E7i9/?spm\\_id\\_from=333.788.recommend\\_more\\_video.-1&vd\\_source=f5890c5408bb2787293f07971e05cc83](https://www.bilibili.com/video/BV1wh411E7i9/?spm_id_from=333.788.recommend_more_video.-1&vd_source=f5890c5408bb2787293f07971e05cc83)
4. <https://reurl.cc/3eOm7V>
5. [https://github.com/Enzo-MiMan/cv\\_related\\_collections/blob/main/deep\\_learning\\_basic/interpolation/bicubic.py](https://github.com/Enzo-MiMan/cv_related_collections/blob/main/deep_learning_basic/interpolation/bicubic.py)
6. [https://www.bilibili.com/video/BV1Rc411P7WY/?spm\\_id\\_from=333.788&vd\\_source=f5890c5408bb2787293f07971e05cc83](https://www.bilibili.com/video/BV1Rc411P7WY/?spm_id_from=333.788&vd_source=f5890c5408bb2787293f07971e05cc83)
7. <https://jakevdp.github.io/PythonDataScienceHandbook/04.12-three-dimensional-plotting.html>
8. [https://docs.opencv.org/3.4/de/dbc/tutorial\\_py\\_fourier\\_transform.html](https://docs.opencv.org/3.4/de/dbc/tutorial_py_fourier_transform.html)
9. <https://reurl.cc/r6LX9b>
10. <https://github.com/adenarayana/digital-image-processing/blob/main/Python%2008%20Butterworth%20Filter.py>
11. <https://github.com/saurabhraj23/Denoising-Image/blob/main/main.py>
12. [https://www.bilibili.com/video/BV1E14y1k7Ky/?share\\_source=copy\\_web&vd\\_source=2084b3574eac7db2a94a671299845144](https://www.bilibili.com/video/BV1E14y1k7Ky/?share_source=copy_web&vd_source=2084b3574eac7db2a94a671299845144)
13. <https://blog.csdn.net/wjpwpwjp0831/article/details/118424311>
14. <https://reurl.cc/MyRDjX>
15. <https://stackoverflow.com/questions/48572647/recursive-inverse-fft>
16. [https://blog.csdn.net/wenhao\\_ir/article/details/51689960](https://blog.csdn.net/wenhao_ir/article/details/51689960)
17. <https://blog.csdn.net/youcans/article/details/122839594>
18. <https://reurl.cc/3eOm6L>
19. <https://www.jianshu.com/p/98f493de01db>
20. <https://blog.csdn.net/itnerd/article/details/88854199>
21. <https://stackoverflow.com/questions/48201729/difference-between-np-dot-and-np-multiply-with-np-sum-in-binary-cross-entropy-lo>
22. [http://www.ipol.im/pub/art/2011/g\\_lmii/revisions/2011-09-27/g\\_lmii.html](http://www.ipol.im/pub/art/2011/g_lmii/revisions/2011-09-27/g_lmii.html)