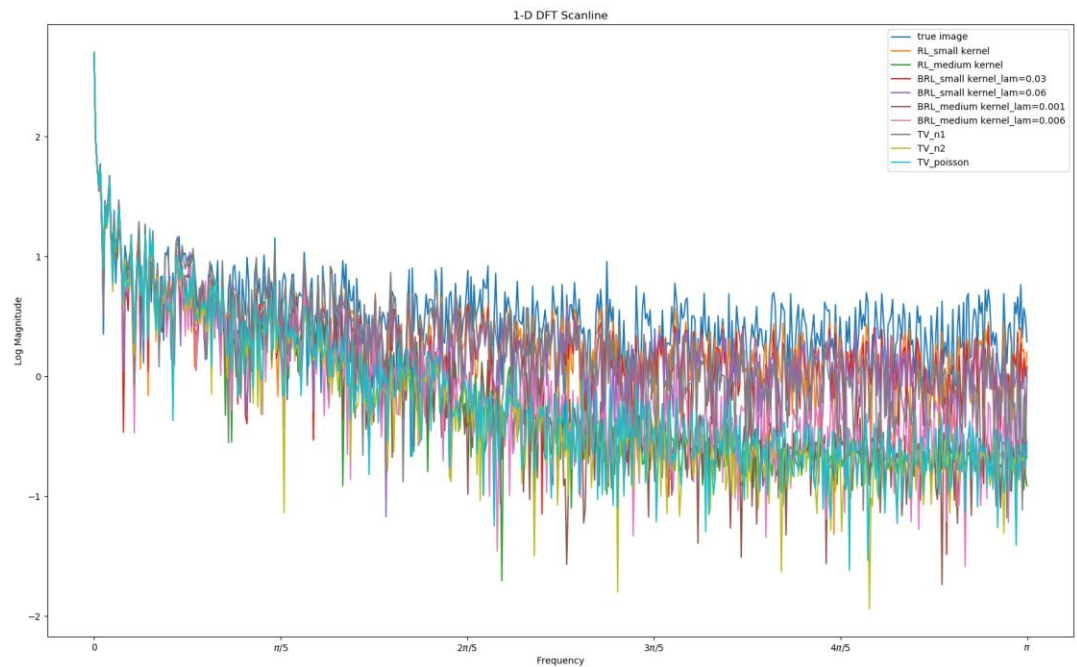


- Experiment – a

- Try to plot a same figure for the deblurred results and a non-blurred image data/curiosity.png



註:上圖的水平白線處為我選的 scanline

- How do you plot the figure? Do you apply any additional steps? Why?
首先將 Implementation 中所有 deblur method 產出的圖與 non-blurred

image 都用 np array 儲存成三維陣列，並從圖中選定一條水平線當作 scanline，我選定的水平線位在 shape[0] 為 350 的位置。並將剛剛上面所有轉成 np array 的圖片之該水平線位置取一維 DFT，一維 DFT 步驟如下：

1. 水平線位在 shape[0] 為 350 的位置之 pixel value 值，先對其 3 個 channel 取平均
2. 取平均後的 array 即為一維陣列，每個 element 都除以 255，做 normalization，將 value 壓在 0 到 1 之間
3. 接著用 np.fft.fft() 做 one-dimensional discrete Fourier Transform
4. 再將 np.fft.fft() 之 output (也是一維陣列) 的每個 element 取 log (以 10 為底)，需特別注意的是，因為 log 的定義域為大於零的數，故須將 np.fft.fft() 之 output 取絕對值再取 log。最後產出的一維陣列即為一維 DFT 的結果 (DFT_output)。

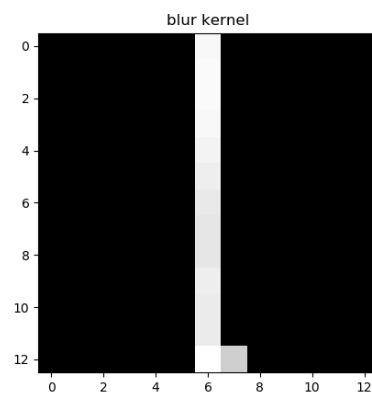
而作業說明中所附圖之 X 軸 frequency domain 是 0 到 π ，因為是一半的週期，所以這邊我們將一維的 DFT 結果取前半當作 Y 軸，X 軸則是 0 到 π 的等差數列 (總共有 $(\text{len}(\text{DFT_output})/2)$ 個數字)，並以 plt.plot 作圖即可得到某一張圖的 1-D DFT scanline。以此類推總共有十張圖都重複上述步驟即可。

■ How this figure can help you to explain the result?

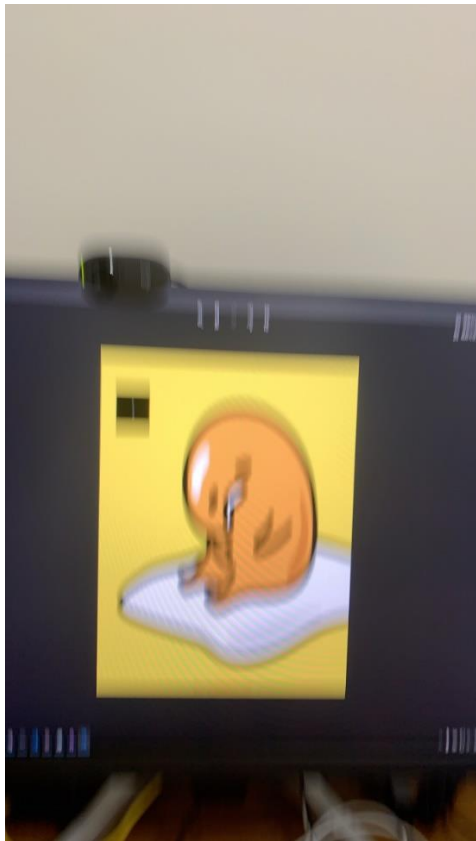
我們可以看出，沒有晃動的 true_image 其 Log Magnitude 最大，然後 TV_poisson 最小，且 deblur 效果越好的方法，其 Log Magnitude 越大，越逼近 true_image 的 Log Magnitude，並且可由該折線圖清楚發現 total variation 的作法，不論是 norm-1 或 norm-2 或 poisson 其 deblur 效果都輸給 RL 或 BRL，因為 total variation method 的 Log Magnitude 幾乎都是墊底，而 BRL 因為多了 spational & range penalty，預期 BRL 的 Log Magnitude 應該會比 RL 都高一個檔次，但從折線圖中發現也不盡然是如此。

● Experiment – b

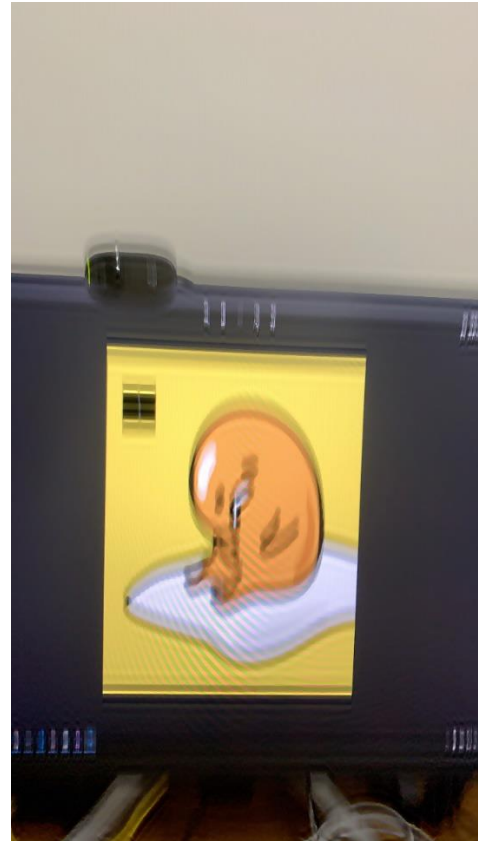
- blurred images with straight kernel (程式碼位置：/code/Experiments_b_straight_kernel.py)
 - ◆ straight kernel (13*13)



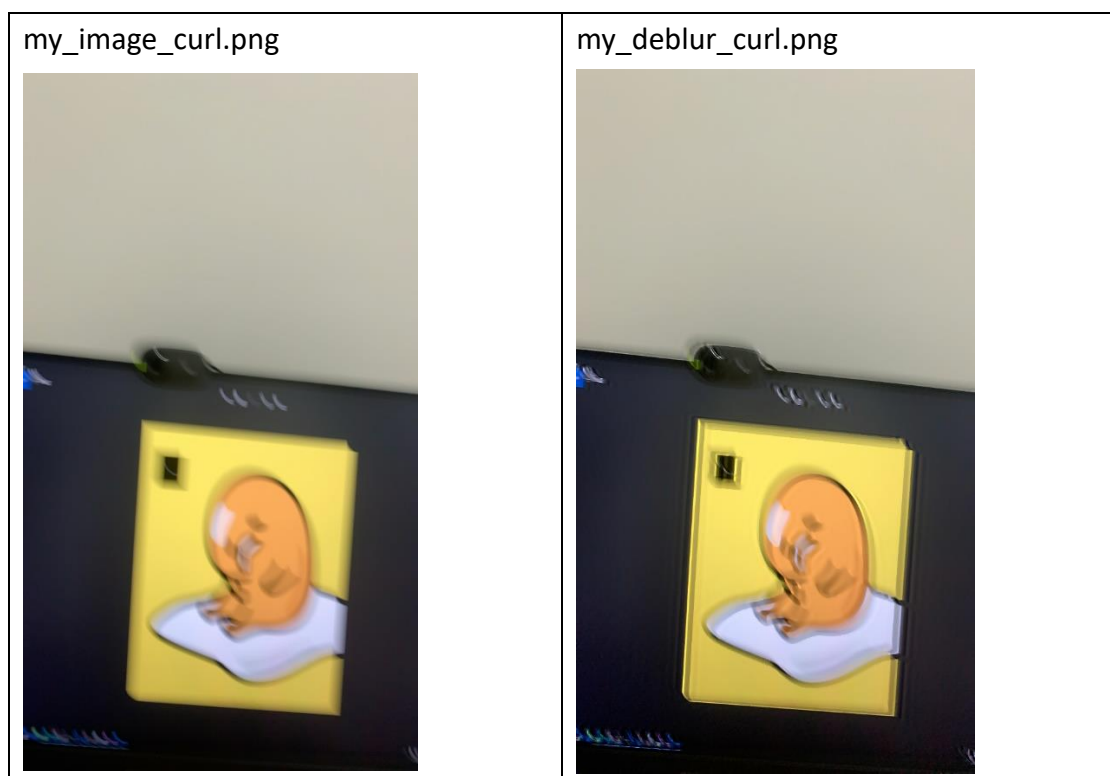
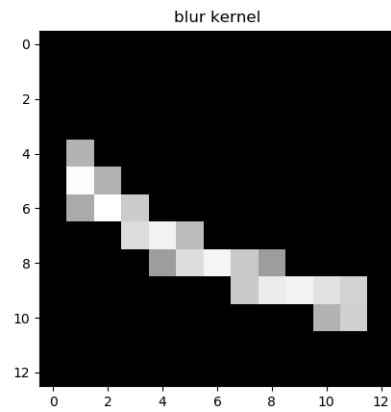
my_image_straight.png



my_deblur_straight.png



- blurred images with curl kernel(程式碼位置：/code/Experiments_b_curl_kernel.py)
 - ◆ curl kernel(13*13)



■ how you choose your parameters

Straight kernel 與 curl kernel 的 BRL 參數我都設為下表:

max_iter_BRL	25
rk	6
Sigma_r	50.0/255/255
lamb_da	0.03/255
to_linear	'False'
r_omega	0.5*rk

試過非常多組參數組合，幾乎都與上表所產出結果雷同。

■ some extra preprocess on your blur kernels

blur kernel 的作法如下：

1. 用小畫家剪出蛋黃哥左上角的方框
2. 用 opencv resize 成 downsample 成 13*13 的圖片，並以灰階儲存
3. 再對這 13*13 個 pixels 做一個 threshold 判斷，若沒有超過某個 threshold，都將其設為 0。
4. 因為 curl kernel 做 resize 後在邊界有些許雜訊，故 curl kernel 有額外做一個判斷，讓 13*13 的邊界處的值都設成 0。

註:步驟 3 與步驟 4 的目的都是希望讓 kernel 的形狀很接近 straight 與 curl，並且使其清晰易見。

■ compare the the deblur effect for the two kernels

Deblur 效果沒有很好的原因，推估是因為我將照片拍得太晃了。不過還是可以看出 deblur 成效，像是圖下方的工作列有變比較清楚，且蛋黃哥圖片左上角的黑色方框區域內 kernel 線，可看出不論是 straight 或是 curl，雖然沒有變回一個點，但都從完整的一條線變成三個分段，可見其 deblur 成效。而不論是 straight kernel 或是 curl kernel，用 BRL 做完都有一點 ringing effect。

● Free study – Problem 1.

■ Compare the RL and BRL results of Curiosity-small and Curiosity-medium

◆ Assumption 1

因為沒有任何的 penalty 限制，更新時容易造成圖像物體邊緣處 (pixel value 差距較大的部分) 能量傳遞擴散，故 RL 會有 ringing effect。且當 kernel size 越大，能量可以傳遞越遠，ringing effect 的波紋就會越大且越明顯。

◆ Assumption 2

BRL 加入了 spatial penalty(下式紅框處)與 range penalty(下式藍框處)。尤其是 range penalty，在圖像的物體邊緣 $I(x)$ 與 $I(y)$ 差距較大，此時 range penalty 變大，造成 $\nabla E_B(I^t)$ 變大，此時的更新就較不依賴當前的 I^t ，讓 ringing effect 較 RL 不明顯。

◆ Assumption 3

$$I^{t+1} = \frac{I^t}{1 + \lambda \cdot \nabla E_B(I^t)} \circ (K^* \otimes \frac{B}{I^t \otimes K}), \quad (4)$$

$$\nabla E_B(I^t) = 2 \cdot \sum_{y \in \Omega} \left(\exp\left(-\frac{|x-y|^2}{2\sigma_s}\right) \right) \cdot \left(\exp\left(-\frac{|I(x)-I(y)|^2}{2\sigma_r}\right) \cdot \frac{I(x)-I(y)}{\sigma_r} \right). \quad (6)$$

當 BRL 之 rk 值越大， σ_s 值就會越大，此時 spatial penalty(上式紅框處)變小， $\nabla E_B(I^t)$ 會跟著變小，而再根據上面式(4)， $\nabla E_B(I^t)$ 變小時，表示某次的更新較依賴當前的 I^t ，所以在消除晃動表現上，會較不明顯。

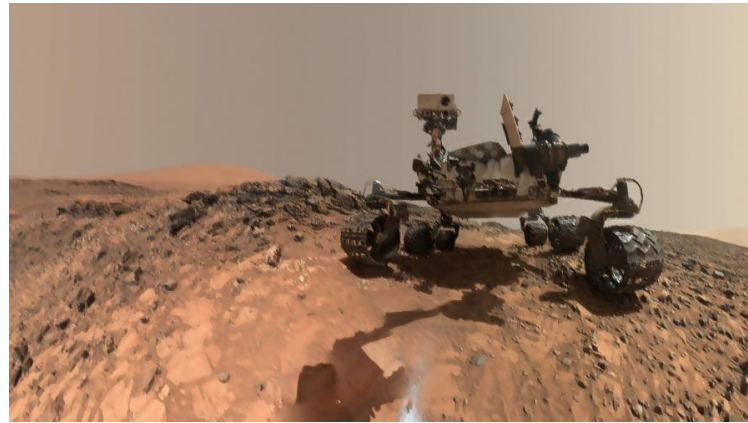
◆ Justification

為了驗證我的假設，我做了如下的對照：(取自 Implementation)

RL, Curiosity-small, iter=25



BRL, Curiosity-small, iter=25, $rk=6$, $\sigma_r=50$, $\lambda_{da}=0.03$



RL, Curiosity-medium, iter=55



BRL, Curiosity-medium, iter=55, $rk=12$, $\sigma_r=25$, $\lambda_{da}=0.001$



可以發現 RL 都會存在 ringing effect，且 Curiosity-medium(kernel size=25*25)比 Curiosity-small (kernel size=13*13)的 ringing effect 更明顯，這邊就驗證了我上面的 Assumption 1(上表的左上圖跟左下圖比較)。而 BRL 因為加入 spatial penalty 與 range penalty，讓 ringing effect 較 RL 不明顯(上表的左圖跟右圖比較，驗證 Assumption 2)，但是會有一層糊糊的油畫感，且 rk 值越高這種模糊感越明顯(上表的右上圖跟右下圖比較)，這邊就驗證了我上面的 Assumption 3，因為 rk 值高造成每次的更新梯度低，解決方法之一是讓更新次數變得更多，就有機會消除模糊感。

- Compare the BRL results of Curiosity-medium with different parameters.

- ◆ Compare lamb_da (其他參數都固定，iteration=25, sigma_r=50/255/255, rk=6)

- Assumption

$$I^{t+1} = \frac{I^t}{1 + \lambda \cdot \nabla E_B(I^t)} \circ \left(K^* \otimes \frac{B}{I^t \otimes K} \right), \quad (4)$$

當 lamb_da 值越大，分母越大，使得 spatial penalty 與 range penalty 的權重提高，所以畫面更加 smooth。並可由上式發現更新時受 I^t 影響也會越小，且更新多半受上面 4 式後面黑框處所控制，所以能夠更有效消除 ringing effect。

- Justification

lamb_da=0.03/255



lamb_da=0.3/255 (圖片位

置:/my_RL_BRL_result/Free_study_p1/

BRL_s_iter25_rk6_si50.00_lam0.300.png)



為了驗證我的 assumption，我將其他參數固定(iteration=25, sigma_r=50/255/255, rk=6)，只改動 lamb_da 值，當 lamb_da 放大十倍的情況下，可以看到圖像更加 smooth，並且有一點油畫不真實感。

- ◆ Compare sigma_r (其他參數都固定，lamb_da=0.03/255, iteration=25, rk=6)

- Assumption

$$\nabla E_B(I^t) = 2 \cdot \sum_{y \in \Omega} \left(\exp\left(-\frac{|x-y|^2}{2\sigma_s}\right) \right) \cdot \left(\exp\left(-\frac{|I(x)-I(y)|^2}{2\sigma_r}\right) \cdot \frac{I(x)-I(y)}{\sigma_r} \right). \quad (6)$$

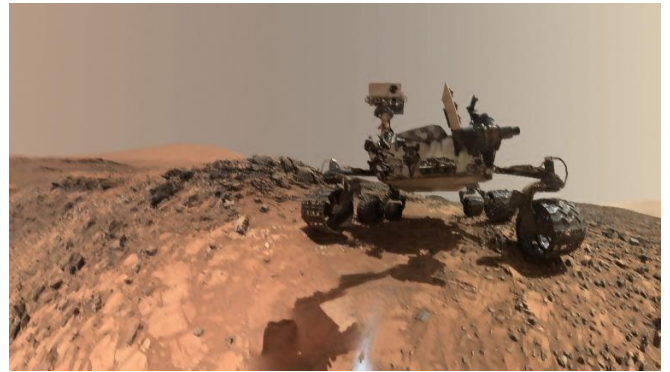
由上式可發現，當 sigma_r 越大，後面藍框處的 range penalty 就會越小，當圖像的物體邊緣 $I(x)$ 與 $I(y)$ 差距不變的情況下，因 sigma_r 在分母，造成 $\nabla E_B(I^t)$ 變小，此時的更新就較依賴當前的 I^t ，故推論 ringing effect 會較明顯。

- Justification

$\sigma_r = 50.0/255/255$



$\sigma_r = 100.0/255/255$ (圖片位置:
置:/my_RL_BRL_result/Free_study_p1/
BRL_s_iter25_rk6_si100.00_lam0.030.png)



為了證明我的推論，我將 σ_r 放大兩倍變成 $100.0/255/255$ ，其他參數固定($\lambda_{da}=0.03/255$, $iteration=25$, $rk=6$)，可發現在右圖好奇號的物體邊緣處，有些微的 ringing effect 產生。由此可驗證我的假設。

- Free study - Problem 2. Compare the BRL results of Curiosity-medium with different boundary condition.

- Assumption

Implementation 部分採用 **symmetric padding** 的方式，而這邊我想嘗試採用 **zero padding** 來看看效果差異，因為 **zero padding** 是將突出原圖的部分 **pixel value** 都補 0，而這樣可能造成圖像邊角處的 **pixel value** 差距會很大，可能比圖片中物體的邊緣 **pixel value** 差距還大，所以推測在圖像邊角處的 **ringing effect** 應該會相當明顯。

- Justification

Medium kernel, $\lambda_{da}=0.001/255$, $iter=55$,
 $\sigma_r=25/255/255$, $rk=12$, symmetric padding



Medium kernel, $\lambda_{da}=0.001/255$, $iter=55$,
 $\sigma_r=25/255/255$, $rk=12$, zero padding(圖片位置:
my_RL_BRL_result/ Free_study_p2/
BRL_s_iter55_rk12_si25.00_lam0.001zero_padding.png)



為了驗證我的假設，上表右圖我在其他參數不變的情況下，實作 zero padding，明顯可見圖像邊角處有很明顯的 ringing effect，甚至比圖像物體的邊緣處所產生的 ringing effect 還要明顯許多。而 symmetric padding 的作法讓圖像邊角處的 pixel value 值很接近，使的 BRL 做完時，圖像邊角不太會有 ringing effect 情況產生。

- Free study – Problem 3. Compare some different λ on TVL1 deconvolution.

- Assumption

$$E(I) = \|I \otimes k - B\| + \lambda \|\nabla I\|_{tv}$$

如上式所示， λ 在 TVL1 deconvolution 用來控制照片 total variation 的 gradient 之參數，當 λ 越大，相同的 gradient 下所造成的影響越大，在同樣的 update iteration 次數下，其晃動振波感會比較少。

- Justification

lamb_da = 0.01, iter=1000



lamb_da = 0.1, iter=1000(圖片位置: my_RL_BRL_result/Free_study_p3/ deblur_edgetaper_norm1_lam0.1.png)




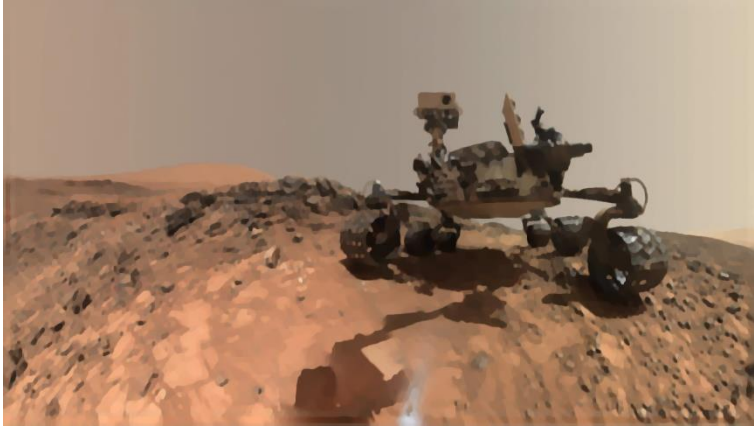
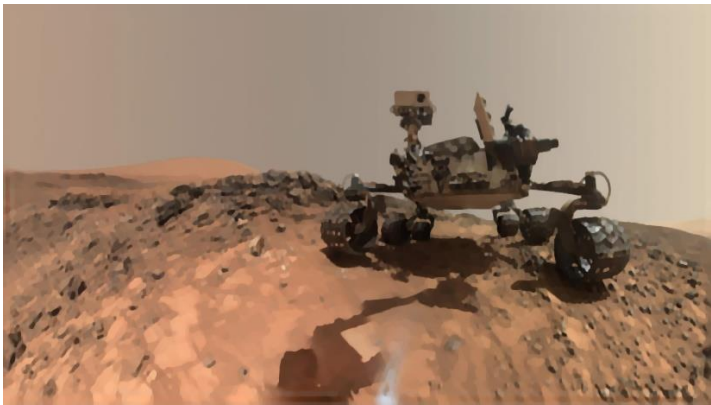
為了驗證我的推論，我用 TVL1.py 跑了兩種測試，上表右圖的 λ 是左圖的 10 倍，可發現振波晃動感的確隨著 λ 變大而有所消失，但是在同樣的 update 次數下，因為 λ 變大造成每次更新的梯度較大，會有一點失真感，某些細微處(如好奇號的胎紋)變得有點模糊。

- Free study – Problem 4. Compare the results of TVL1, TVL2 and TVpoisson.

- Assumption

TVL2 因為採用 L2 norm，根據 gradient 的更新會比 TVL1 的 L1 norm 更加顯著，所以 TVL2 的效果類似將 TVL1 的 λ 值調很大，消除了振波感，但是畫面會有點失真。而 TVPoisson 是將 input blurred image 當作 Poisson noise 來做處理，而我們的 task 是 deblur 而不是 denoise，所以推估用 TVpoisson 做 deblur 的 task 效果不會太好。

- Justification

TVL1 test_solver = 'pc' lam_da=0.01 eps=0.001	
TVL2 test_solver = 'pc' lam_da=0.01 eps=0.001	
TVpoisson test_solver = 'pc' lam_da=0.01 eps=0.001	

可發現在其他參數都不變的情況下，TVL1 做出來效果最佳(圖像較細緻)，而 TVL2 就如同我所說的類似於將 TVL1 的 λ 值調很大的效果(Free study – Problem 3)，而 TVpoisson 的效果在這邊類似於 TVL2，都帶模糊感，並且也有做到 deblur 的效果，這邊跟我原先假設相違背，我原本以為 TVpoisson 主要是用來消除 poisson 雜訊的，所以覺得在 deblur 表現上不會太好。

- Free study – Problem 5. Try to speed up BRL and report the execution time difference

- Assumption 1

因為 BRL function 內有大量的矩陣相關運算，而且需要三個

channel(RGB)分開計算，在運算上會花費不少時間。而我是對式 6 做優化，式 6 的數學式如下：

$$\nabla E_B(I^t) = 2 \cdot \sum_{y \in \Omega} \left(\exp\left(-\frac{|x-y|^2}{2\sigma_s}\right) \cdot \exp\left(-\frac{|I(x)-I(y)|^2}{2\sigma_r}\right) \cdot \frac{I(x)-I(y)}{\sigma_r} \right). \quad (6)$$

我原本程式碼的實作方式是將三個 channel 分開並且依序加總，如下：

```
E_B_I_t[i-pdsize,j-pdsize,0] = np.sum(total_kernel[:, :, 0]) # scalar
E_B_I_t[i-pdsize,j-pdsize,1] = np.sum(total_kernel[:, :, 1]) # scalar
E_B_I_t[i-pdsize,j-pdsize,2] = np.sum(total_kernel[:, :, 2]) # scalar
```

而我認為這會花費不少時間，可改成直接用一行 numpy 的 api 做平行運算，如下：

```
E_B_I_t[i-pdsize,j-pdsize,:] = np.sum(total_kernel, axis=(0,1))
```

一樣都是分別對三個 channel 做加總，但是優化後的寫法可以同時對三個 channel 分開加總，我想這樣會讓運行效率更好。

■ Assumption 2

將式 6 的數學式子(如下圖)

$$\nabla E_B(I^t) = 2 \cdot \sum_{y \in \Omega} \left(\exp\left(-\frac{|x-y|^2}{2\sigma_s}\right) \cdot \exp\left(-\frac{|I(x)-I(y)|^2}{2\sigma_r}\right) \cdot \frac{I(x)-I(y)}{\sigma_r} \right). \quad (6)$$

根據 exp 性質，將紅框與藍框處先做加總再取 exp，應該會比分別取 exp 再做相乘的效率還高，因為相加的效率比相乘還要快，並且優化後的寫法少做一次 exp。

■ Justification

為了驗證我上述的兩個 Assumption 有讓執行效率提升，且執行出來結果一樣，我在 code/Free_study_p5.py 做實驗，分別有 BRL_no_optimization()與 BRL_optimization()兩個 function，而程式執行順序是先做 BRL_no_optimization()再做 BRL_optimization()，兩個 function 傳入參數都一樣，rk=6, sigma_r=50.0/255/255, lamb_da=0.03/255, max_iter_BRL=25，並且都是對 curiosity_small.png 做 deblur。做完後都與 reference answer 算一次 psnr，確認兩個優化前後的輸出結果一致。結果如下：

```
[Running] python -u "c:\Users\f6405\Desktop\EE6620_
iter:1/25
iter:2/25
iter:3/25
iter:4/25
iter:5/25
iter:6/25
iter:7/25
iter:8/25
iter:9/25
iter:10/25
iter:11/25
iter:12/25
iter:13/25
iter:14/25
iter:15/25
iter:16/25
iter:17/25
iter:18/25
iter:19/25
iter:20/25
iter:21/25
iter:22/25
iter:23/25
iter:24/25
iter:25/25
BRL process time(no optimization) = 557.455571 sec
psnr = 102.080431
```

```
iter:1/25
iter:2/25
iter:3/25
iter:4/25
iter:5/25
iter:6/25
iter:7/25
iter:8/25
iter:9/25
iter:10/25
iter:11/25
iter:12/25
iter:13/25
iter:14/25
iter:15/25
iter:16/25
iter:17/25
iter:18/25
iter:19/25
iter:20/25
iter:21/25
iter:22/25
iter:23/25
iter:24/25
iter:25/25
BRL process time(optimization) = 410.831470 sec
psnr = 102.080431
[Done] exited with code=0 in 972.091 seconds
```

上面左圖是執行 `BRL_no_optimization()`，上面右圖是執行 `BRL_optimization()`，可發現與 reference answer 比較 psnr 都一樣的情況下，優化後的執行時間比優化前快了約 147 秒，可見差異之顯著。