Image Deblurring and QR Factorizations - Report

- Li Lu
- 121090272

1 Kernel Implementation and Image Blurring

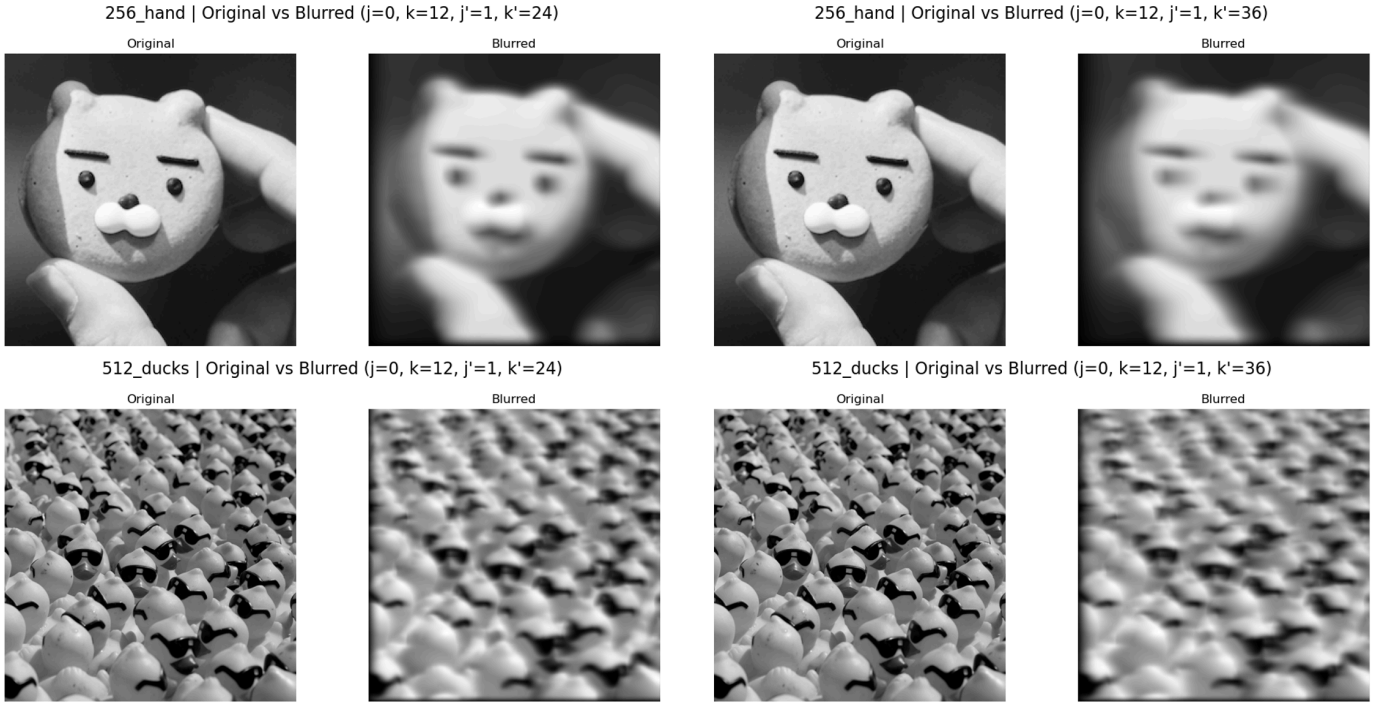
Two set of kernels are set following the format:

$$\begin{aligned}&\text{for } \boldsymbol{A}_\ell: \quad j=0,\ k=12 \quad (\boldsymbol{A}_\ell \text{ is an upper triangular matrix})\\&\text{for } \boldsymbol{A}_r: \quad j=1,\ k\in\{24,36\} \quad (\boldsymbol{A}_r \text{ is upper triangular with an extra subdiagonal})\end{aligned}$$

Blurred images are created following the **linear system** below

$$\boldsymbol{A}_\ell \boldsymbol{X} \boldsymbol{A}_r = \boldsymbol{B}. \tag{1}$$

Four pairs of original images and blurred images:



2 Analysis of the Deblurring Problem

Notice that A_l is already an upper triangular matrix, while A_r **has subdiagonal entries**, which possibly makes its **conditional number very large**, leading to the unstability and inaccuracy for solving the linear system.

Observations for $cond(A_r)$:

k \ n	256	512	1024	2048
24	6.4994e+12	1.5240e+17	2.6645e+17	8.2926e+16
36	8.9135e+09	1.8527e+16	3.4650e+17	3.5760e+17
100	6.8499e+07	4.2795e+09	1.4875e+14	2.8909e+17

From the table above, we can see that **cond(A_r)** varies with different matrix size n and number of diagonals k. Generally, we have $cond(A_r)$

- increases as n increases. Thus, as the image size grows, it may become harder to deblur it.
- roughly decreases as k increases. Thus, images blurred by kernels with larger k, deblurring ususally becomes easier.
- decreases faster as k increases for smaller n. Thus, the effects of deblurring varies greater for image with smaller sizes.

3 Image Deblurring by Builtin LU and QR functions

We only need to decompose A_r and then solve the linear system $A_l X L_r U_r = B$ or $A_l X Q_r R_r = B$.

```
from numpy.linalg import pinv
from scipy.linalg import lu, qr

def recover_by_LU(B, Al, Ar, lu_func=lu):
    _, Lr, Ur = lu_func(Ar, permute_l=False)
    X_rec = pinv(Al) @ B @ pinv(Ur) @ pinv(Lr)
    return np.clip(X_rec, 0, 1)

def recover_by_QR(B, Al, Ar, qr_func=qr):
    Qr, Rr = qr_func(Ar, pivoting=False)
    X_rec = pinv(Al) @ B @ pinv(Rr) @ Qr.T
    return np.clip(X_rec, 0, 1)
```

4 Householder QR Decomposition

Codes for householder QR decomposition **without column permutation** are shown below: (Unfortunately, the version with column permutation does not work well for deblurring.)

```
def my_qr(A, pivoting=False):
    '''Householder QR decomposition (without column pivoting)'''
    m, n = A.shape
    Q = np.eye(m)
    R = np.copy(A)

    for j in range(n):
        v = R[j:, j].copy()

        v[0] = v[0] + np.sign(v[0]) * norm(v) * v[0]
```

```
        v = v / norm(v)
        H = np.eye(len(v)) - 2 * np.outer(v, v)

        H_full = np.eye(m)
        H_full[j:, j:] = H
        R = H_full @ R
        Q = Q @ H_full.T
    return Q, R[:n, :]
```

5 Potential Improvements and Other Methods

5.1 Least Square Formulation

Instead of solving the linear system, we can consider solve the problem by considering the least square formluation below:

$$\min_{\boldsymbol{X}} \|\boldsymbol{A}_\ell \boldsymbol{X} \boldsymbol{A}_r - \boldsymbol{B}\|_F^2,$$

```
from scipy.linalg import lstsq

def recover_by_lstsq(B, Al, Ar):
    Y, _, _, _ = lstsq(Al, B)
    X_rec_t, _, _, _ = lstsq(Ar.T, Y.T)
    X_rec = X_rec_t.T
    return np.clip(X_rec, 0, 1)
```

5.2 Solve by Pure Psudo-Inverse

Though A_r is ill-conditioned, we may still consider solving the linear system by computing its **(Moore-Penrose) pseudo-inverse** and analyze the performance.

```
from numpy.linalg import pinv
def recover_by_pure_pinv(B, Al, Ar):
    X_rec = pinv(Al) @ B @ pinv(Ar)
    return np.clip(X_rec, 0, 1)
```

5.3 Padding A_r to Make it Diagonal

After obtaining $Y = A_l^{-1}B$, we can consider solving the linear system

$$\hat{X}\hat{A}_r = \hat{Y}$$

to get $X = \hat{X}[:-1, :-1]$, where $\hat{A}_r$ and $\hat{Y}$ are obtained by padding A_r and Y. This system is expanded as

$$\begin{bmatrix} X_{11} & \cdots & X_{1n} & 0 \\ X_{21} & \cdots & X_{2n} & 0 \\ \vdots & \ddots & \vdots & \vdots \\ X_{n1} & \cdots & X_{nn} & 0 \\ 0 & \cdots & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & (A_r)_{11} & (A_r)_{12} & \cdots & (A_r)_{1(n-1)} & (A_r)_{1n} \\ 0 & (A_r)_{21} & (A_r)_{22} & \cdots & (A_r)_{2(n-1)} & (A_r)_{2n} \\ 0 & 0 & (A_r)_{32} & \cdots & (A_r)_{3(n-1)} & (A_r)_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & (A_r)_{n(n-1)} & (A_r)_{nn} \\ 0 & 0 & 0 & \cdots & 0 & 1 \end{bmatrix} = \begin{bmatrix} \tilde{X}_{11} & Y_{11} & \cdots & Y_{1n} \\ \tilde{X}_{21} & Y_{21} & \cdots & Y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \tilde{X}_{n1} & Y_{n1} & \cdots & Y_{nn} \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

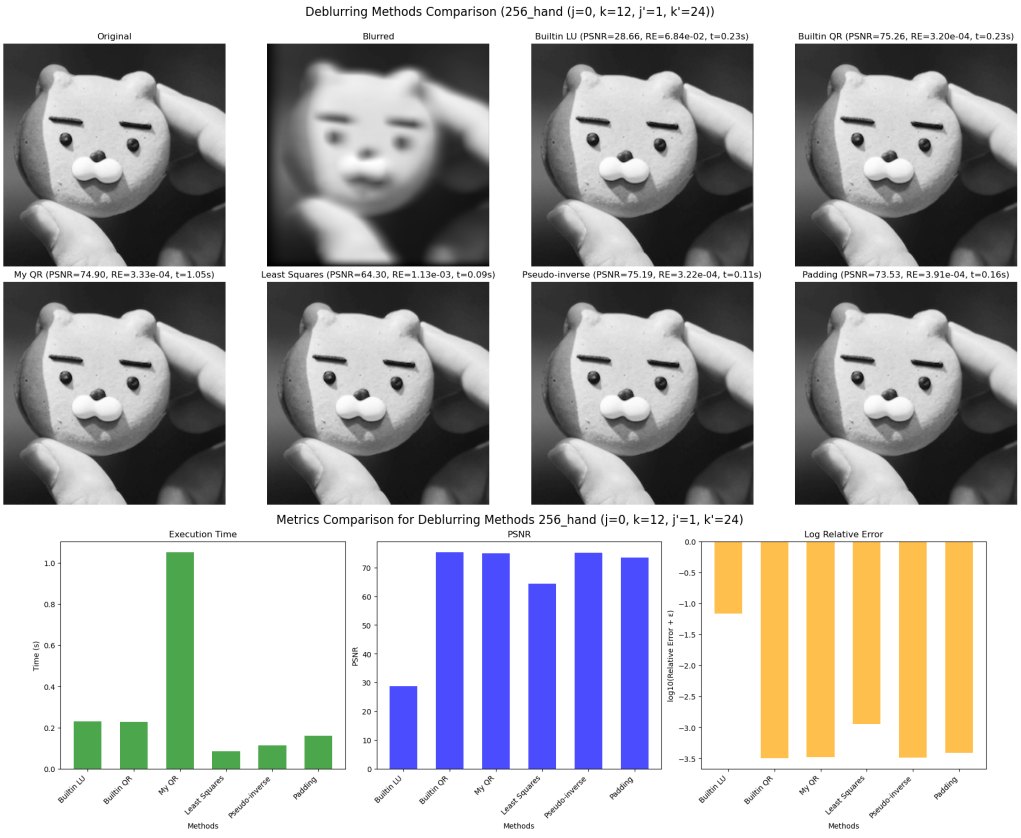
Assuming $X[:,0] \approx X[:,1]$, we can assign values for $\tilde{X}_{k1}$ in $\hat{Y}$ by estimating X_{k1} following the formular

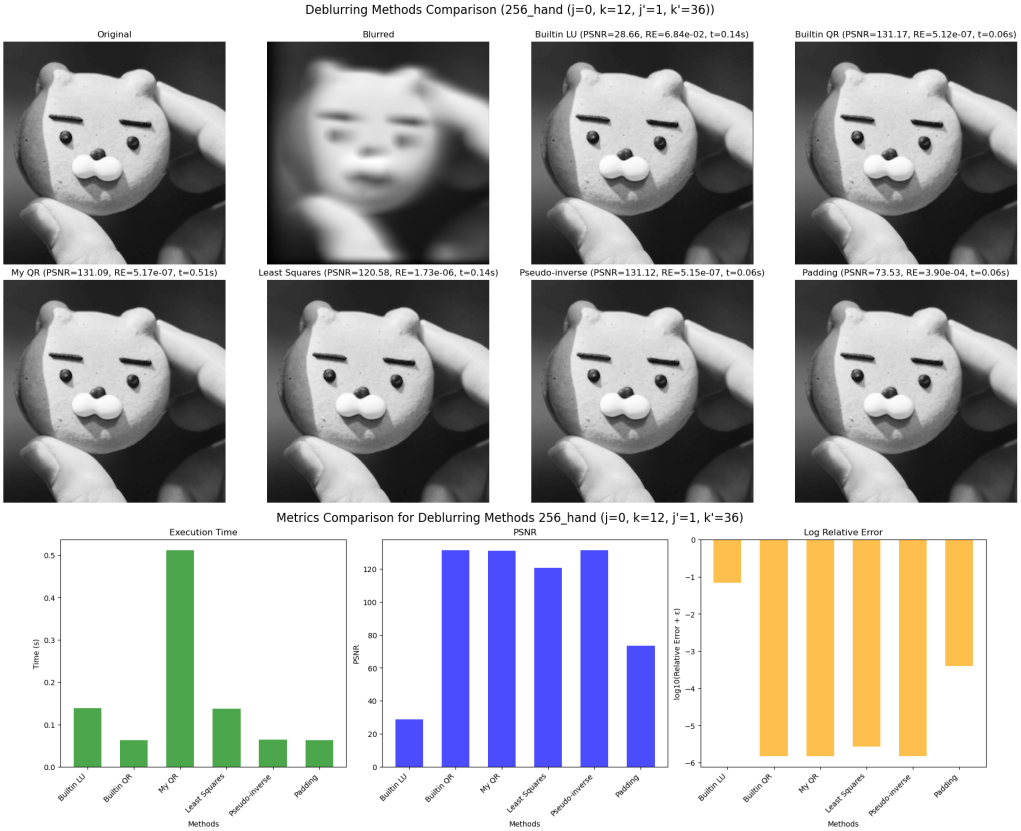
$$\tilde{X}_{k1} = \frac{Y_{k1}}{(A_r)_{11} + (A_r)_{21}} \quad \text{for } k = 1, \cdots, n$$

```python
def recover_by_pad(B, Al, Ar):
    Y = pinv(Al) @ B

    Y = np.pad(Y, ((0,1), (1,0)), mode='constant')
    Y[:-1, 0] = Y[:-1, 1] / (Ar[0,0] + Ar[1,0])

    Ar = np.pad(Ar, ((0,1), (1,0)), mode='constant')
    Ar[0,0] = 1; Ar[-1,-1] = 1

    X_rec = (Y @ pinv(Ar))[:-1, :-1]
    return np.clip(X_rec, 0, 1)
```

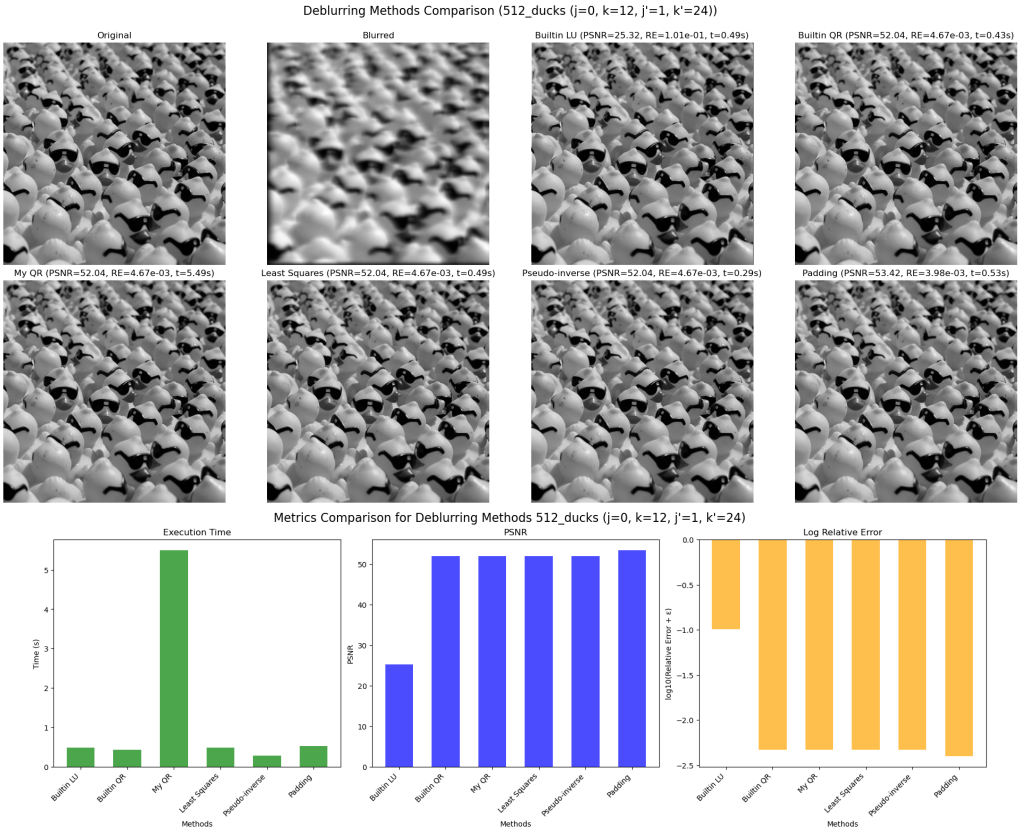
6 Results of Deblurring by Six Methods

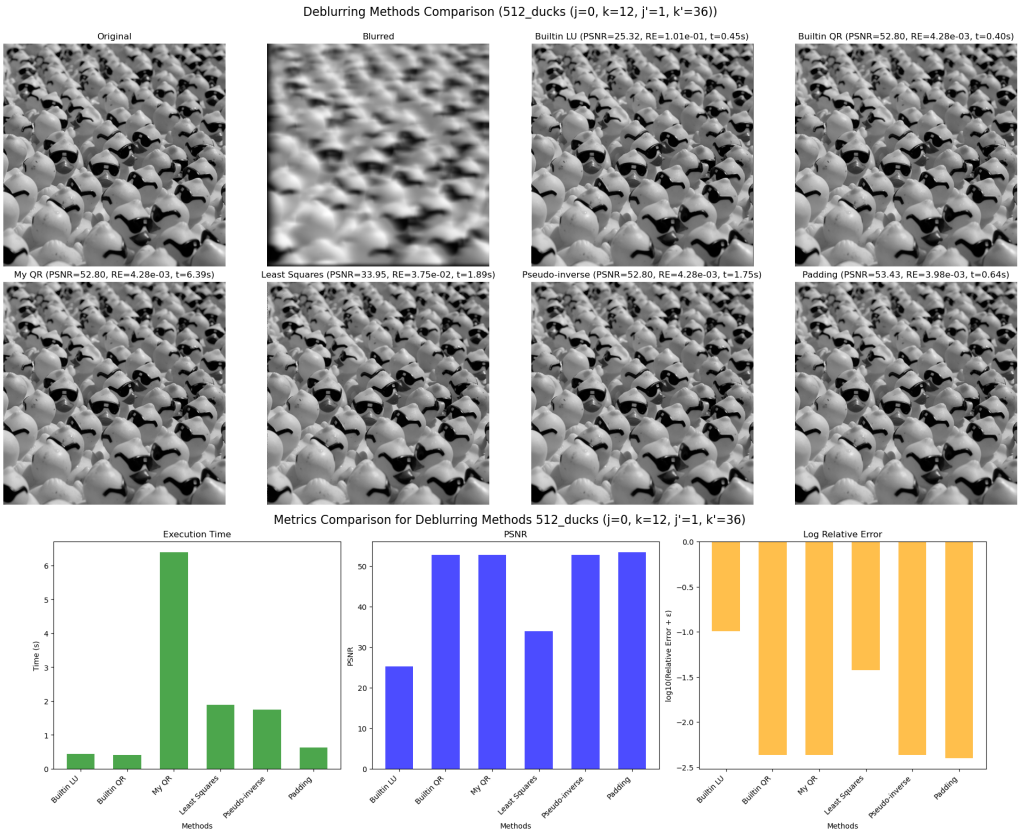
6.1 256_hand




Deblurring Methods Comparison (256_hand (j=0, k=12, j'=1, k'=36))
Original
Blurred
Builtin LU (PSNR=28.66, RE=6.84e-02, t=0.14s)
Builtin QR (PSNR=131.17, RE=5.12e-07, t=0.06s)
My QR (PSNR=131.09, RE=5.17e-07, t=0.51s)
Least Squares (PSNR=120.58, RE=1.73e-06, t=0.14s)
Pseudo-inverse (PSNR=131.12, RE=5.15e-07, t=0.06s)
Padding (PSNR=73.53, RE=3.90e-04, t=0.06s)
Metrics Comparison for Deblurring Methods 256_hand (j=0, k=12, j'=1, k'=36)
Execution Time
Time (s)
PSNR
Log Relative Error
log10(Relative Error + ε)
Builtin LU
Builtin QR
My QR
Least Squares
Pseudo-inverse
Padding
Methods

6.2 512_ducks

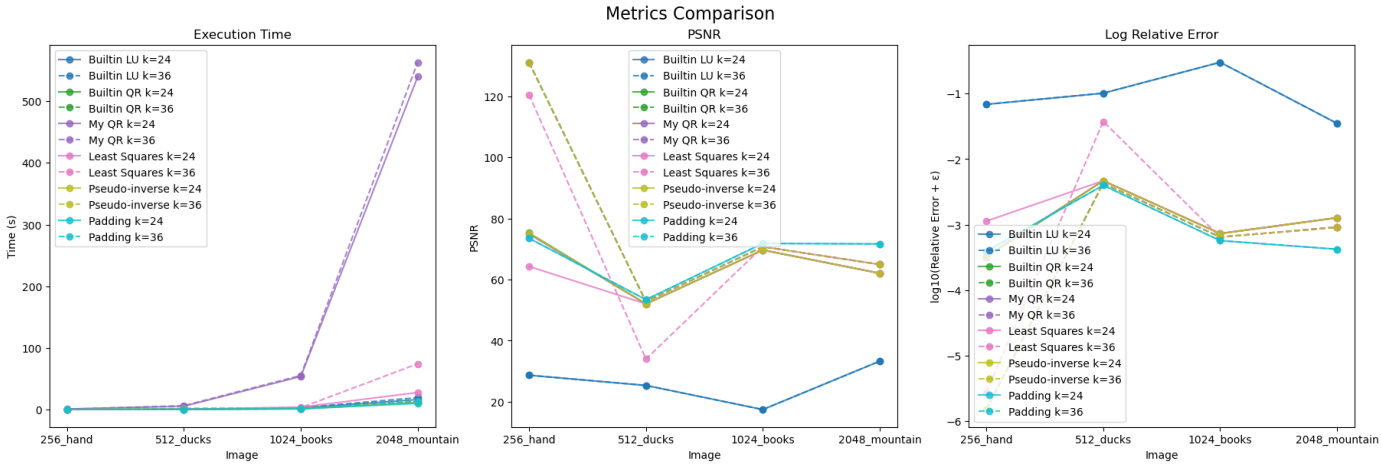

Deblurring Methods Comparison (512_ducks (j=0, k=12, j'=1, k'=24))
Original
Blurred
Builtin LU (PSNR=25.32, RE=1.01e-01, t=0.49s)
Builtin QR (PSNR=52.04, RE=4.67e-03, t=0.43s)
My QR (PSNR=52.04, RE=4.67e-03, t=5.49s)
Least Squares (PSNR=52.04, RE=4.67e-03, t=0.49s)
Pseudo-inverse (PSNR=52.04, RE=4.67e-03, t=0.29s)
Padding (PSNR=53.42, RE=3.98e-03, t=0.53s)
Metrics Comparison for Deblurring Methods 512_ducks (j=0, k=12, j'=1, k'=24)
Execution Time
Time (s)
PSNR
Log Relative Error
log10(Relative Error + ε)
Builtin LU
Builtin QR
My QR
Least Squares
Pseudo-inverse
Padding
Methods



7 Comparison among Six Methods and Discussions

In this project, **six** methods are tested on **four** images deblurred by **two** sets of kernels. Deblurred results for the last two images are shown in the appendix.

We consider **three** metrics to evaluate the efficiency of these six methods. Good methods are those with **less** execution time, **high** PSNR, and **low** (log) relative forward error (using the Frobenius norm). Below shows the metrics comparison:



Observations and discussions

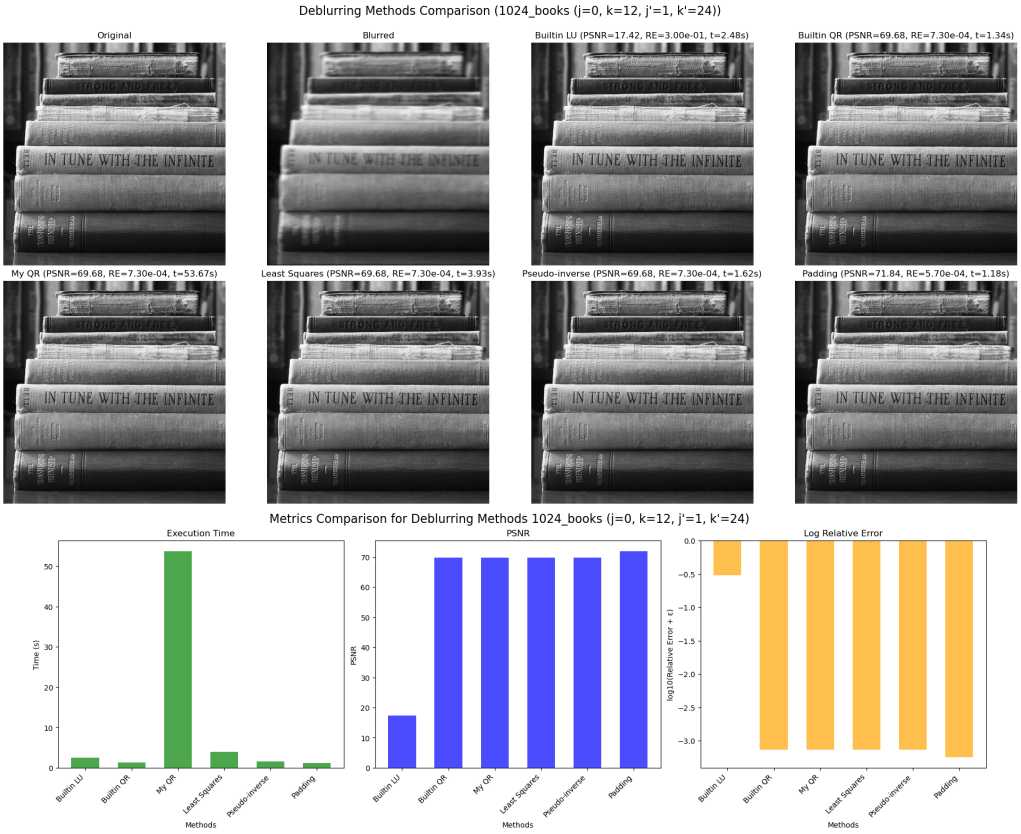
- As the image size increases, the excution time increases dramatically, especially for **the my_qr method (self-implemented Householder decomposition)**. The **padding method** requires least execution time, since it needs no decomposition for A_r. While the **pure psudo-inverse method** does not involve matrix decomposition, computing the (Moore-Penrose) pseudo-inverse of the non-triangular matrix A_r requires more time for computing that of the triangular matrix $\hat{A}_r$.

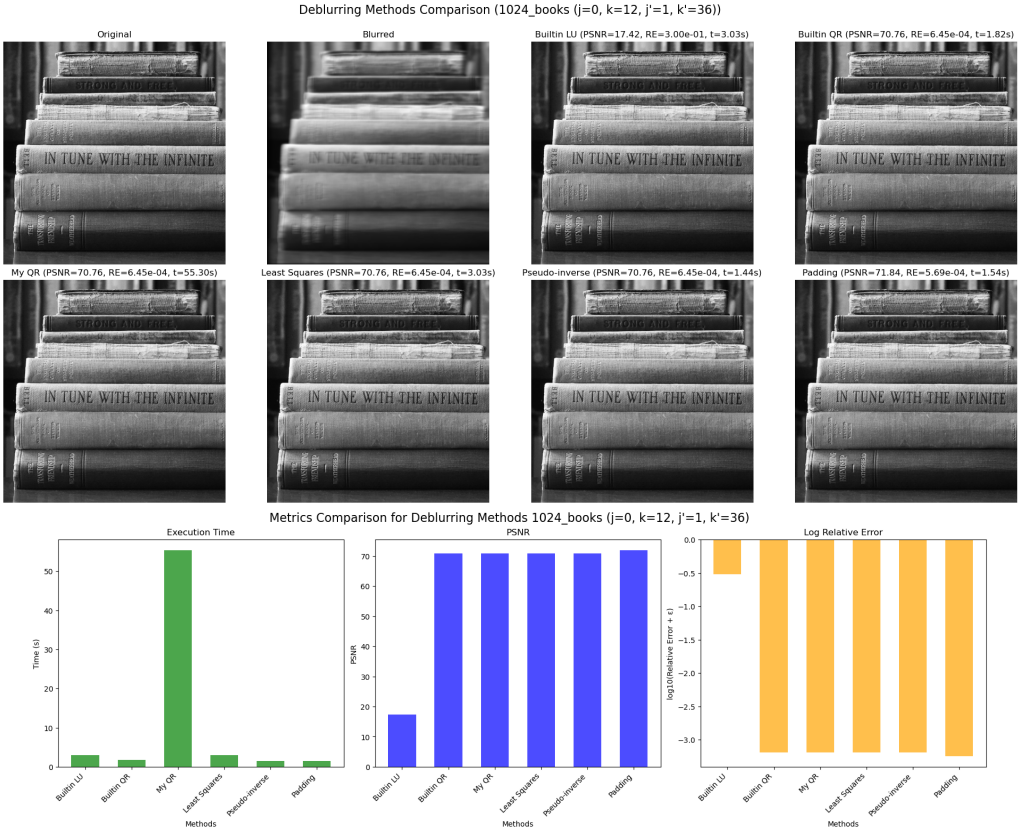
- On all images, the **builtin LU method** performs worst considering both PSNR and relative forward error. This is due to the **instability of LU decomposition** compared to QR decomposition especially for ill-conditioned matrices.
- The **least squares method** and the **builtin QR method** show similar performance on very large images, though the latter one requires less execution time.
- Though the **padding method** requires least execution time, it is not recommended for deblurring small-sized images, as the estimation error of one column is non-negligible.
- It must be admitted that the **pure psudo-inverse method** performs well. If the **padding method** is unpractical when the explicit form of blurring kernels is not given, it is one of the optimal choice for deblurring.
- The **my_qr method** gives exactly the same PSNR and relative forward error as the **builting QR method**, since these two essentially do the same operations. However, since the latter one is highly accelerated by more efficient codes (implemented by different programming language), it requires much less execution time.
- Overall, for small-sized images, **pure psudo-inverse method** and **least squares method**, are preferred; for larger images, the **padding method** is optimal.
- As analyzed for $cond(A_r)$ in section 2, the performance of different methods varies slightly for kernels with $k = 24$ and $k = 36$ when n is small, and is nearly the same for large n.

---

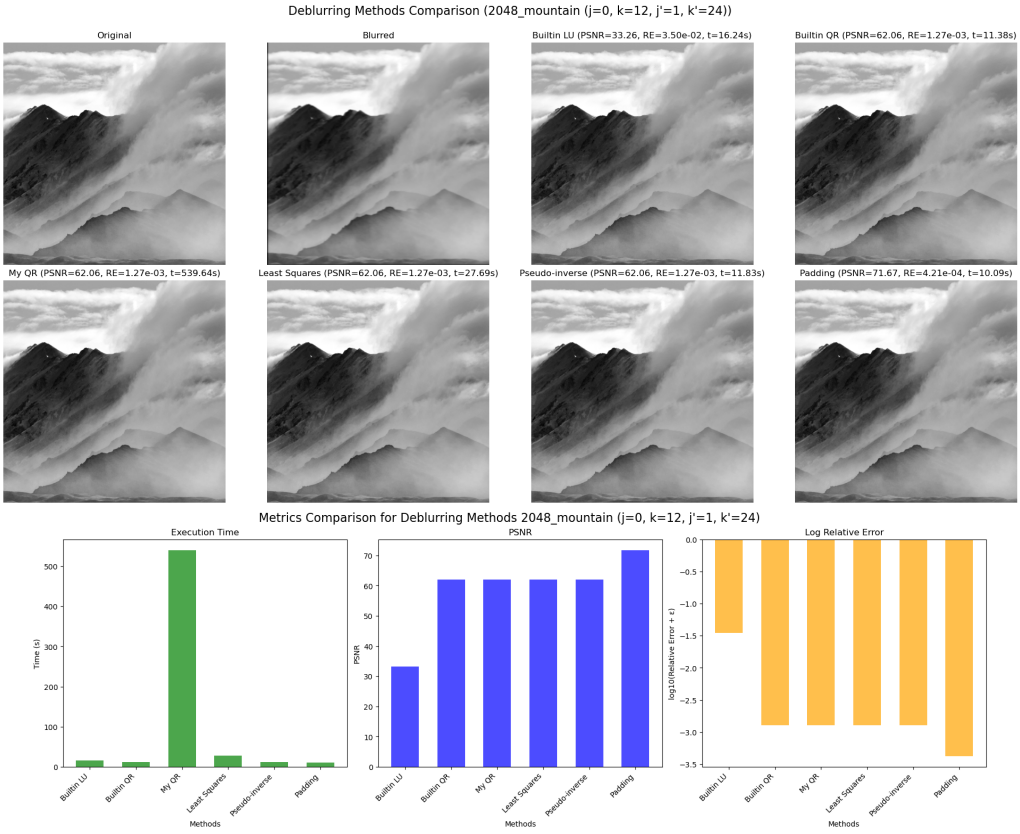
8 Appendix: Results for "1024_books" and "2048_mountain"

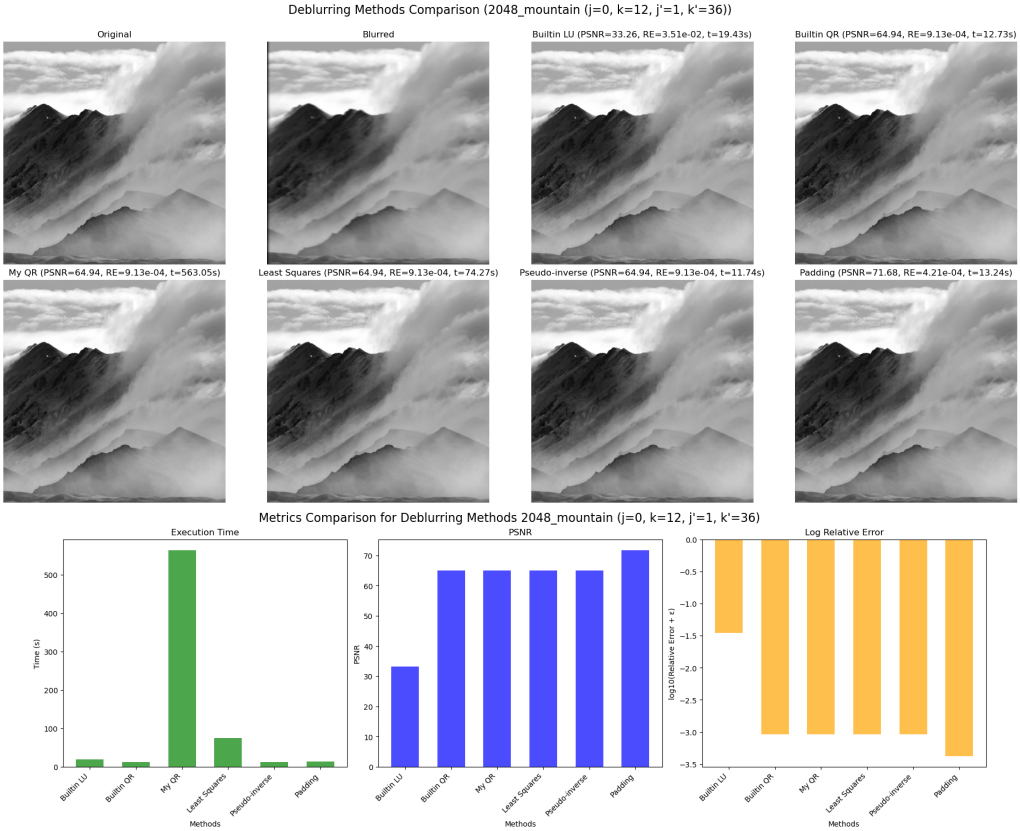
8.1 1024_books





8.2 2048_mountain




Deblurring Methods Comparison (2048_mountain (j=0, k=12, j'=1, k'=36))
Original
Blurred
Builtin LU (PSNR=33.26, RE=3.51e-02, t=19.43s)
Builtin QR (PSNR=64.94, RE=9.13e-04, t=12.73s)
My QR (PSNR=64.94, RE=9.13e-04, t=563.05s)
Least Squares (PSNR=64.94, RE=9.13e-04, t=74.27s)
Pseudo-inverse (PSNR=64.94, RE=9.13e-04, t=11.74s)
Padding (PSNR=71.68, RE=4.21e-04, t=13.24s)

Metrics Comparison for Deblurring Methods 2048_mountain (j=0, k=12, j'=1, k'=36)
Execution Time
Time (s)
PSNR
Log Relative Error
log10(Relative Error + ε)
Builtin LU
Builtin QR
My QR
Least Squares
Pseudo-inverse
Padding
Methods