Image Deblurring and QR Factorizations

DDA 3005 Numerical Methods

Course Project Report

Lexuan Chen 122090029

---

1. Project Introduction

This project investigates image deblurring techniques using matrix factorization methods. The main goal is to reconstruct original images (i.e., deblur) from blurred versions using different numerical approaches, including LU and QR factorizations.

Problem Description Given a blurred image $B \in R^{n\times n}$ and two blurring kernels A_l, $A_r \in R^{n\times n}$, we aim to solve the linear system: $\boldsymbol{A_lXA_r = B}$ ***(1),*** where $X \in R^{n\times n}$ is the original image to be recovered. The blurring kernels follow a specific structure for motion-type blurring, with elements defined by: $\mathbf{[a_{n+j}\ a_{n+j-1} \dots a_{n+j-k+1}] = (2 / k (k + 1))\ [k\ k\text{-}1 \dots 1]}$, which creates realistic motion blur effects while maintaining mathematical tractability.

The most intuitive way to solve ***(1)*** is perform matrix inverse: $\boldsymbol{X = A_l^{-1}BA_r^{-1}}$ ***(2)***. However, large condition numbers of kernels could commonly lead to the instability of direct inverse, impeding the reconstruction process. In this project, we try to resolve the instability utilizing different decomposition methods. The project encompasses three main tasks:

- Implementing and analyzing different matrix factorization methods (LU and QR)
- Developing a custom Householder QR factorization algorithm
- Investigating numerical stability and improving solution quality

2. Part (a) Image Blurring

To ensure consistent processing of different image types and sizes, we developed a preprocessing system that converts all images to grayscale and normalizes the pixel values to the [0,1] range. This step is essential for accurate blur simulation and quality assessment.

Blurring Process For the blurring operation, we implemented two key kernel construction functions following the specified motion-type format. The left kernel ($\boldsymbol{A_l}$)is constructed as an upper triangular matrix with j = 0 and k = 12, while the right kernel ($\boldsymbol{A_r}$) features an additional sub-diagonal with j = 1 and variable k values (48 and 96 in our experiments). This asymmetric kernel configuration creates realistic directional motion blur effects.

Padding To address boundary artifacts that commonly occur in image processing, we developed an edge padding mechanism. Our padding strategy extends the image borders by max (k_l, k_r) pixels using edge replication, which significantly reduces boundary distortions in the blurred images. This

approach proves particularly effective for larger kernel sizes, where boundary effects are more pronounced. Moreover, you can also choose not to conduct padding by setting parameter ***Pad = False***.

Our experiments were conducted on images of varying sizes (256×256, 1250×1250, 3500×3500) under different blurring parameters (48, 96). The results demonstrate consistent blur effects across all test cases, with larger k_r values producing more pronounced motion blur while maintaining numerical stability.

Figure 1: Blurred Example 1 (256 x 256)

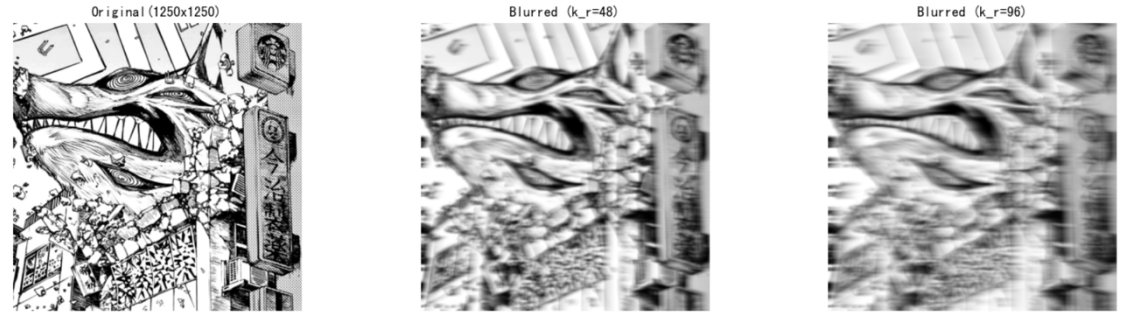


Figure 2: Blurred Example 2 (1024 x 1024)

Figure 3: Blurred Example 3 (3500 x 3500)

The results are robust, as it is evident that for the same original images, a larger k_r leads to more blurred results. Additionally, images with smaller sizes become blurrier when the same k_r is applied.

3. Part (b) Image Deblurring

In this problem, I will try to solve problem with traditional methods, LU decomposition and QR factorization.

3.1 LU Decomposition

Formulation The way to solve the deblurring problem with LU decomposition can be mathematically represented in the following way:

$$A_l X A_r = L_l U_l X L_r U_r = B \quad (3)$$

$$L_r U_r X = L_l^{-1} U_l^{-1} B \quad (4)$$

$$U_r^T L_r^T X^T = C^T \quad (5)$$

$$X^T = \left(L_r^T\right)^{-1} \left(U_r^T\right)^{-1} C^T \quad (6)$$

Efficiency and Stability The LU decomposition is computed using *scipy.linalg.lu*, and forward and backward substitution are employed to solve the triangular systems. This approach ensures numerical stability, as the inverses of matrices are calculated through substitution rather than direct matrix inversion, which enhances efficiency. Moreover, the result *X* is clipped to the valid range [0, 1] to ensure proper image pixel values.

3.2 QR Decomposition

Formulation The way to solve the deblurring problem with QR decomposition can be mathematically represented in the following way:

$$A_l X A_r = Q_l R_l X Q_r R_r = B \quad (7)$$

$$R_l X Q_r = Q_l^T B \quad (8)$$

$$X = (R_l^{-1} Q_r^T) Q_l^T B \quad (9)$$

Efficiency and Stability The QR decomposition is performed using *scipy.linalg.qr*, which ensures the orthogonality of the matrices Q_l and Q_r. Solving the system involves triangular and orthogonal matrix transformations, requiring efficient matrix multiplication and inversion. This process minimizes rounding errors due to the properties of orthogonal matrices, enhancing numerical stability.

3.3 Results and Comparison

Quality Assessment To evaluate the effectiveness of the deblurring process, we used the Peak Signal-to-Noise Ratio (PSNR) as a quality metric: $\boldsymbol{PSNR = 10 \cdot \log_{10}(1 / MSE)\ (3)}$ where MSE represents the Mean Squared Error, and the images are normalized to the [0,1] range. We also record the required CPU-time and relative forward error (i.e., RFE in Table 4) for each reconstruction process as further comparison perspectives.

Using the same three images as in part (a), I selected different values for k_r for them. Specifically, for the 256 x 256 image, I choose k_r = 48; or the 1250 x 1250 and 3500 x 3500 images, I selected k_r = 96 for both, which produced better blurring results. Figure 4 presents the reconstruction results for both LU decomposition and QR decomposition

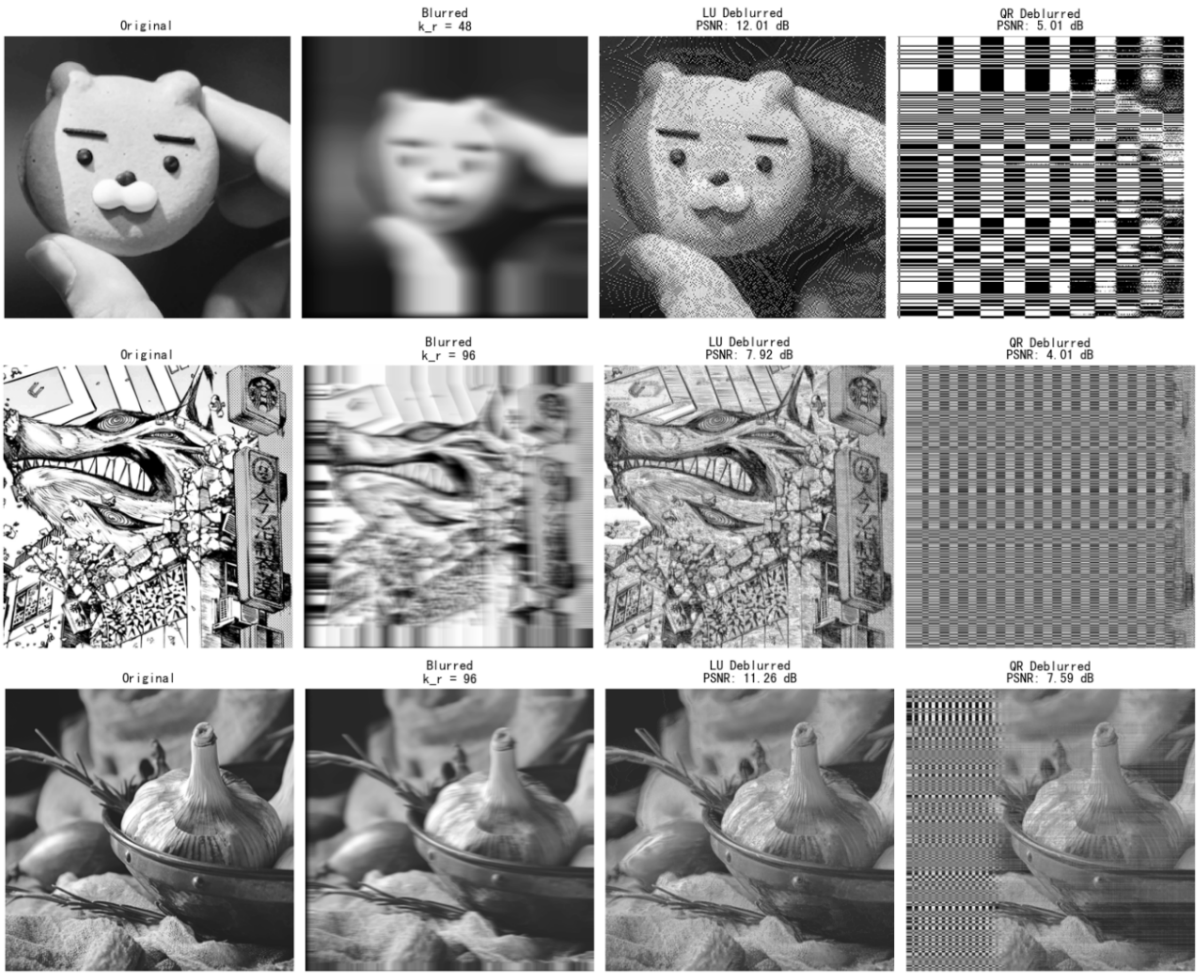


Figure 4: LU Reconstruction and QR Reconstruction

Image Size	K_r	Method	CPU-time	RFE	PSNR
256 x 256	48	**LU**	0.919s	0.465	12.01 dB
		QR	0.307s	1.041	5.01 dB
1250 x 1250	96	**LU**	1.730s	0.531	7.92 dB
		QR	1.023s	0.833	4.01 dB
3500 x 3500	96	**LU**	6.897s	0.534	11.26 dB
		QR	10.832s	0.816	7.59 dB

Table 1: Comparison Results for LU and QR

Given the structure of the blurred kernel matrix, LU decomposition shows faster computation and higher PSNR, especially for smaller matrices (256x256). This can be attributed to LU's ability to efficiently handle the upper triangular matrix and capture the low-frequency features induced by the blurring process, which results in better image reconstruction quality. On the other hand, QR decomposition, while having a shorter computation time, leads to lower PSNR and RFE, particularly for larger matrices (1250x1250 and 3500x3500). This may be due to the additional sub diagonal in the matrix, which QR decomposition struggles to adapt to, resulting in suboptimal performance in terms of image quality. The blurred kernel likely introduces smoother structures that LU decomposition handles better, offering superior computational efficiency and image quality for this specific matrix structure. Thus, for this task, LU decomposition appears more effective overall.

3.4 Further Discussion

Building on the previous results, we further investigated whether the choice of k_r values influence the outcomes. To this end, we selected a 1250x1250 image and tested a wider range of Kr values for comparison. The results were quite unexpected. As shown in Figure 5, the QR method significantly outperformed the LU method. One possible explanation for this could be the different sensitivity of the methods to kernel size variations or the stability of the QR decomposition with larger kernels.

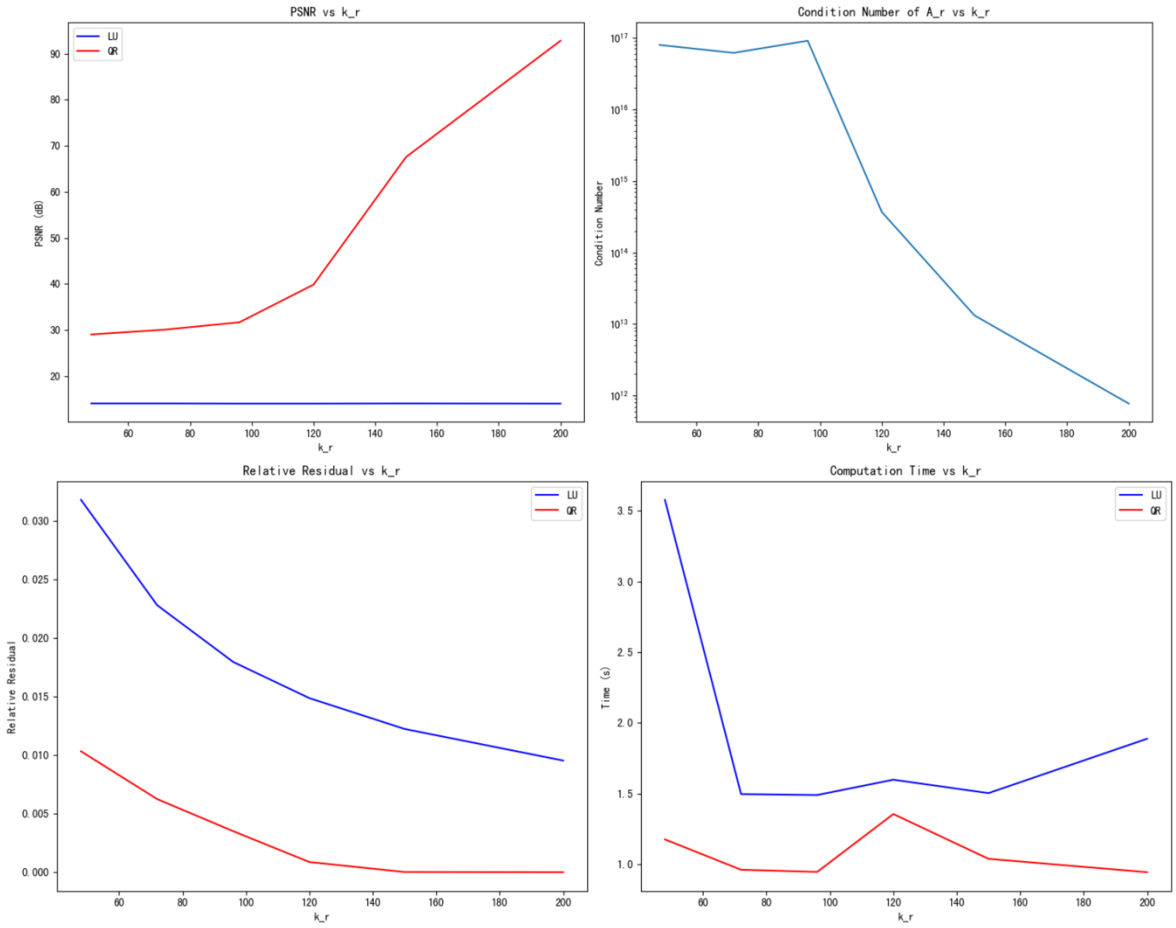


Figure 5: Further comparison

Although in our initial, smaller-scale tests, the LU method showed superior performance over the QR method, the results from the broader range of Kr values suggest that QR may ultimately be the better choice. This contrast indicates that while LU might perform better under certain conditions, QR could prove to be more stable and efficient when dealing with larger or more complex kernels. Therefore, further investigation is necessary to fully understand the circumstances under which each method excels, and to determine which is more reliable across a wider array of scenarios.

4. Part (c) Householder Reflection

I implement the Householder QR decomposition with pivoting through three key stages:

Pivoting Strategy The implementation begins with a strategic pivoting process that enhances numerical stability. For each column k, the algorithm identifies the element with maximum absolute value and performs row permutations to position it on the diagonal. These permutations are tracked through a permutation matrix P, ensuring reversibility of the transformations.

Householder Reflections The core decomposition employs carefully constructed Householder reflections. For each column, a reflection vector v is computed using the formula $v = x \pm ||x||_2 e_1$, where the sign is chosen to maximize numerical stability. These reflections systematically zero out elements below the diagonal, gradually transforming A into an upper triangular matrix R while maintaining the orthogonal matrix Q through cumulative updates.

Matrix Adjustments and Solution In the final phase, the algorithm adjusts Q by incorporating the permutation matrix P, ensuring the decomposition satisfies AP = QR. The implementation includes a solver component that efficiently handles linear systems using this decomposition, offering flexibility through optional pivoting.

Efficiency and Stability The implementation stands out for its robust handling of numerical precision and modular design. The clear separation between pivoting, reflection calculations, and matrix transformations ensures both stability and maintainability.

```python
def pivoting(A):

    m, n = A.shape
    P = np.eye(m)
    A_pivoted = A.copy()

    for k in range(min(m, n)):
        # Find the index of the largest absolute value in column k
        max_index = np.argmax(np.abs(A_pivoted[k:, k])) + k

        if max_index != k:
            # Swap rows in A
            A_pivoted[[k, max_index]] = A_pivoted[[max_index, k]]
            # Swap corresponding rows in P (permutation matrix)
            P[[k, max_index]] = P[[max_index, k]]

    return P, A_pivoted
```

```python
def householder_reflection(x):
    x = np.asarray(x, dtype=float)
    v = np.zeros_like(x)
    alpha = np.linalg.norm(x)
    if x[0] <= 0:
        v[0] = x[0] - alpha
    else:
        v[0] = x[0] + alpha
    v[1:] = x[1:]
    beta = 2 / (np.linalg.norm(v) ** 2)
    return v, beta
```

Coding1: Matrix Pivoting and Householder reflection for column

```python
def qr_with_permutation(A):
    m, n = A.shape
    Q = np.eye(m)

    # Apply column pivoting
    P, R = pivoting(np.copy(A))

    # Perform Householder reflections to compute Q and R
    for k in range(min(m, n)):  # Use min(m, n) to avoid out-of-bounds error
        x = R[k:, k]
        v, beta = householder_reflection(x)
        H = np.eye(m)
        H[k:, k:] -= beta * np.outer(v, v)
        R = np.dot(H, R)
        Q = np.dot(Q, H)

    # Reorder Q based on P
    Q = np.dot(Q, P.T)

    return Q, R, P
```

Coding 2: Main Loop for Householder QR

```python
def solve_with_qr(A, B, use_permutation=False):
    if not use_permutation:
        Q, R = np.linalg.qr(A)
        y = Q.T @ B
        x = np.linalg.inv(R) @ y
    else:
        Q, R, P = qr_with_permutation(A)
        y = Q.T @ B
        x = P @ np.linalg.inv(R) @ y
    return x

def deblur_householder_qr(B, A_l, A_r, use_permutation=False,pad_width=None):
    start_time = time.time()

    C = solve_with_qr(A_l, B, use_permutation)
    X = solve_with_qr(A_r.T, C.T, use_permutation).T
    X = np.clip(X, 0, 1)

    if pad_width is not None:
        X = X[pad_width:-pad_width, pad_width:-pad_width]

    cpu_time = time.time() - start_time

    return X, cpu_time
```

Coding 3: Solution and Final Implementation

5. Part (d) Test for Householder QR

In this part, I test my own Householder QR decomposition and compare the deblurred results with those in part (b). I first reconstruct the image whose size is 1250 x 1250 with k_r = 48. Figure 6 shows

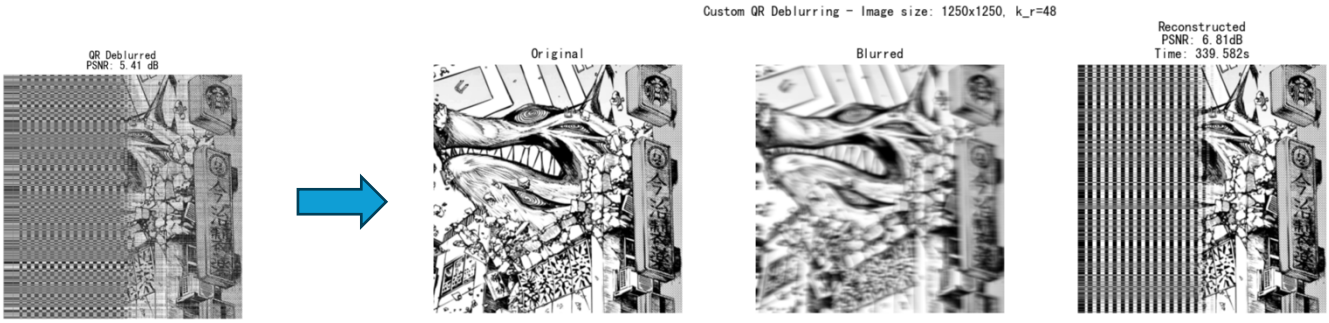


Figure 6: Householder QR Example Result 1

my first result. Although the reconstruction image is still incomplete, it has apparent improvement compared with the result used in-built QR decomposition method using k_r =48. Its right half is very perfect that we can clearly watch word on the image.

Next, I use a smaller image (640 x 640) with different k_r values (48, 72, 96) to further test the validity of my own Householder QR. The results are extraordinary.

Figure 7: Further test on Householder QR

Discussion and Potential improvement In the image deblurring experiments, both the handwritten QR implementation and the built-in QR decomposition failed to fully restore the images, although the handwritten version showed some optimizations. The primary reason for incomplete restoration lies in the stability of matrix decomposition. Image data often has high dimensions and complex structures, which can lead to numerical instability in the QR decomposition, especially when dealing with singular or ill-conditioned matrices from the blurring process. Despite the handwritten version showing improved stability and lower relative errors (5.650e-02, 1.177e-04, and 1.286e-06), it still could not fully recover the image details.

Deblurring is inherently a nonlinear problem, while QR decomposition is designed for linear systems. While QR can recover some global image structures, it struggles to restore finer local details, leading to incomplete image restoration. Additionally, QR is a direct method and lacks the iterative refinement typical in other deblurring algorithms. This lack of iterative adjustment results in poor recovery of fine details, especially in larger images like the 1250 size. These challenges, including local distortions, will be further discussed in section part (e).

In conclusion, while the handwritten QR version performs well in terms of stability and numerical precision with very low relative error, it still fails to completely restore the image due to the complexity of the deblurring problem and the lack of regularization and iterative refinement. Future improvements could involve combining regularization, nonlinear modeling, and iterative optimization to significantly enhance restoration, especially for larger images.

6. Part (e) Further improvements

Matrix Conditioning Analysis In Part (b) of the testing, I observed a significant impact of the matrix condition number on the deblurring performance. In the tests, A_l generally maintains a relatively stable condition number (i.e., A_l's cond = 1.30×10^1 in all cases), but A_r shows substantial variation. As A_r's condition number increases, especially for larger images (e.g., A_r's cond = 2.66×10^18 for a 1024 x 1024 image), the matrix approaches singularity, which can lead to failures in certain deblurring methods. For instance, QR deblurring fails with a singular matrix when the condition number of A_r is extremely large, while LU deblurring shows significant performance degradation due to the ill-conditioning of A_r.

The large condition number of the right kernel A_l has been a persistent issue throughout the project, causing numerical instability and poor image reconstruction. In this part, I try different regularization algorithms like Tikhonov regularization method, iteration algorithms and other decomposition methods like SVD, trying to derive the optimal solution to the given least square problem:

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

Here, I mainly focus on **Tikhonov regularization**. An additional extension is to explore how to choose λ for improved results.

Why Regularization When a matrix is nearly singular (or close to non-invertible), its determinant approaches zero, meaning some eigenvalues are very small. This causes instability in solving linear systems, as even small perturbations in the input can lead to large variations in the solution, making the result unreliable. This instability can be seen as a form of "overfitting," where the model becomes overly sensitive to small changes in the input, leading to a solution that deviates from the true value. By

introducing regularization (like Tikhonov regularization), we can reduce the condition number of the matrix, suppressing this sensitivity, and improving solution stability. Regularization penalizes the solution, preventing it from fitting noise too closely, resulting in a smoother and more reliable solution.

Regularization Matters This algorithm serves as an optimized version of QR factorization of part(b). In this method, the regularization term $\lambda\|X\|_F^2$ adds a penalty for large values of X, preventing overfitting to noise and ill-conditioning. It acts as a smoothness constraint, forcing the solution X to be more stable.

Pre-set Regularization parameter I first pre-set λ to different values (e.g., 1e-4, 1e-6, 1e-8) to explore its effect on the solution when solving the least-squares problem. The conjugate gradient (CG) method is used, with the regularization term directly embedded in the *matvec* function. *Figure 8* shows the results for an image size of 256 x 256 with k_r = 48. From the results, we can see that the choice of λ has a significant impact on the deblurring performance, affecting metrics like PSNR, computation time, and residuals.

Figure 8 Tikhonov regularization with pre-set λ

Adaptive Regularization parameter Based on the previous result, I develop a new approach that adaptively adjusts the regularization parameter λ based on the condition numbers of the left and right kernels. By using the inverse square root of the maximum condition number as a baseline, λ is dynamically tuned to ensure a stable and robust solution. This adjustment helps to strengthen regularization when either kernel is ill-conditioned, improving the deblurring process and preventing excessive blurring or instability in the reconstructed image.

Figure 9 shows results for this method. We use two examples, image 1024 x 1024 with k_r = 48, image 3500 x 3500 with k_r = 96. We could find that the adaptive QR defeat LU decomposition from the perspective of PSNR.

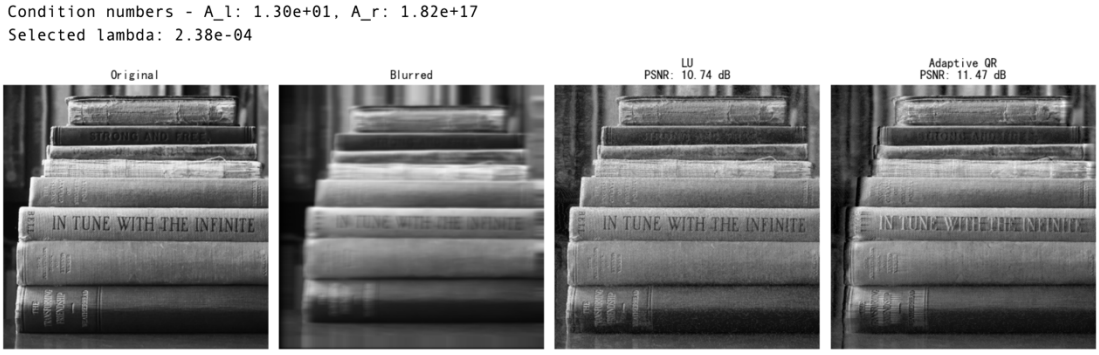
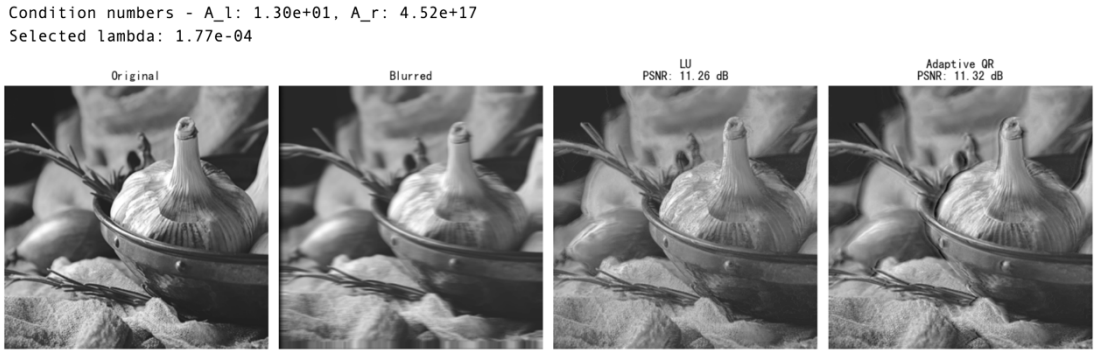


Figure 9: Example Results for auto-Tikhonov regularization into the QR deblurring method



Other Attempts In this work, I further explored two advanced deblurring methods under the guidance of ChatGPT. These methods demonstrated impressive results in both image quality and computation time. However, as they were outside the scope of the course material, I am presenting the results here solely as a reference for further exploration, from same images and under same k_r values in auto-Tikhonov regularization.

Iterative deblurring with adaptive regularization: This method involves using gradient descent with momentum to iteratively minimize the error between the blurred image and the reconstruction. It adapts the regularization term to balance the solution, making it more stable and improving performance over time.

Truncated SVD deblurring: This technique uses Singular Value Decomposition (SVD) to decompose the blurring kernels and then truncates small singular values to reduce noise and computational complexity. The adaptive truncation level is determined by analyzing the drop in singular values, ensuring that only the most significant components are used for the deblurring process.

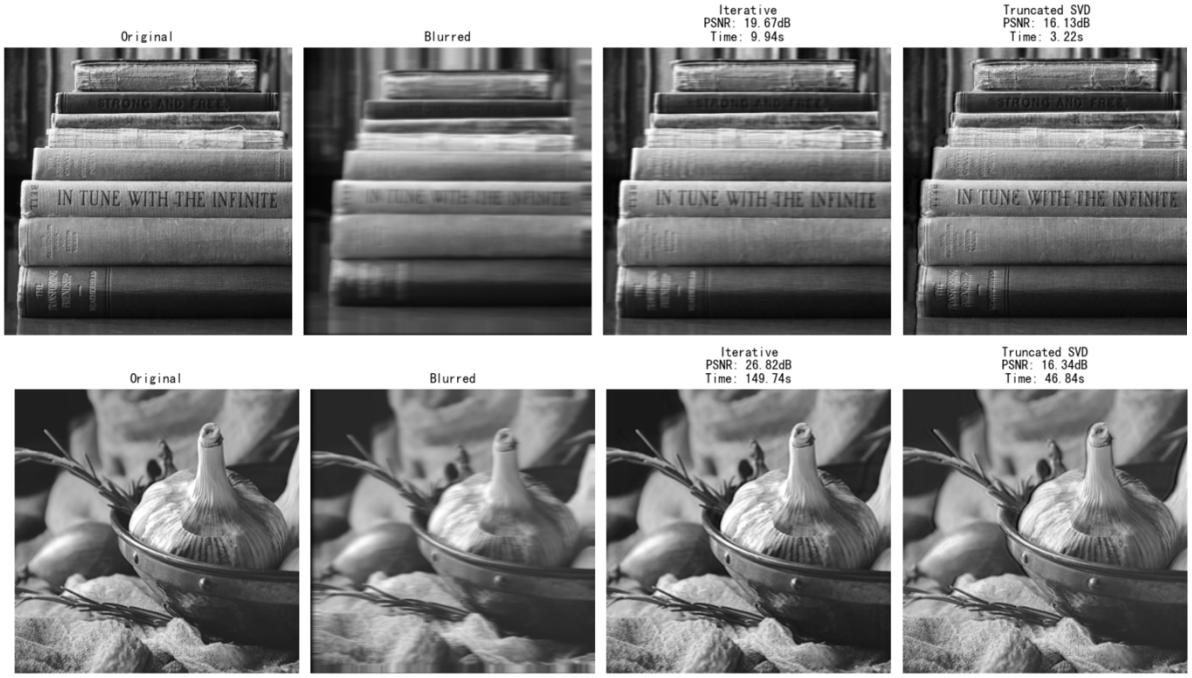


Figure 10: Example Results for Iterative Methods and Truncated SVD

6. Appendix

```python
def build_blur_kernel(n, j, k, upper_triangular=True, extra_subdiagonal=False):
    A = np.zeros((n, n))
    weights = np.array([k - i for i in range(k)])
    weights = (2 / (k * (k + 1))) * weights  # Normalize weights

    for i in range(n):
        for offset in range(len(weights)):
            if upper_triangular:
                if i + offset < n:
                    A[i, i + offset] = weights[offset]
            if extra_subdiagonal:
                if i + offset - j < n and i + offset - j >= 0:
                    A[i, i + offset - j] = weights[offset]
    return A

def apply_blur(X, A_l, A_r):
    """Apply blur using equation (1): A_l @ X @ A_r"""
    return A_l @ X @ A_r

def compute_psnr(original, reconstructed):
    """Compute Peak Signal-to-Noise Ratio"""
    mse = np.mean((original - reconstructed) ** 2)
    if mse == 0:
        return float('inf')
    return 10 * np.log10(1.0 / mse)  # Images are in [0,1] range
```

```python
def create_blurred_image(img_path, n, j_l=0, k_l=12, j_r=1, k_r=24, pad=True):
    """Create blurred version of an image with kernels of original size

    Returns:
        X: Original image
        B: Blurred image
        A_l_orig: Left kernel (original size)
        A_r_orig: Right kernel (original size)
        A_l_padded: Left kernel (padded size) if pad=True, else None
        A_r_padded: Right kernel (padded size) if pad=True, else None
    """
    X = load_image(img_path)

    A_l_orig = build_blur_kernel(n, j_l, k_l,upper_triangular=True, extra_subdiagonal=False)
    A_r_orig = build_blur_kernel(n, j_r, k_r,upper_triangular=False,extra_subdiagonal=True)

    if pad:
        pad_width = max(k_l, k_r)
        X_padded = np.pad(X, pad_width, mode='edge')
        padded_size = X_padded.shape[0]

        A_l_padded = build_blur_kernel(padded_size, j_l, k_l,upper_triangular=True, extra_subdiagonal=False)
        A_r_padded = build_blur_kernel(padded_size, j_r, k_r,upper_triangular=False,extra_subdiagonal=True)

        B_padded = apply_blur(X_padded, A_l_padded, A_r_padded)
        B = B_padded[pad_width:-pad_width, pad_width:-pad_width]

        return X, B, B_padded, A_l_padded, A_r_padded, pad_width
    else:
        B = apply_blur(X, A_l_orig, A_r_orig)
        return X, B, None, A_l_orig, A_r_orig, None
```

Coding 5: Construction of blurring images with control of padding

When constructing blurring images with required kernels, we develop a flexible algorithm, that we can freely control the padding condition. This modifiability extends to all code structures in this project, that all of them could automatically deal with the padding issues in a consistent way.

```python
def deblur_lu(B, A_l, A_r,pad_width=None):

    start_time = time.time()

    try:
        P_l, L_l, U_l = scipy.linalg.lu(A_l)
        P_r, L_r, U_r = scipy.linalg.lu(A_r)

        Y1 = solve(L_l, P_l @ B, check_finite=True)

        Z = solve(U_l, Y1, check_finite=True)

        Y2 = solve(L_r.T, P_r.T @ Z.T, check_finite=True)

        X = solve(U_r.T, Y2, check_finite=True).T

        # Handle any potential numerical instabilities
        X = np.clip(X, 0, 1)  # Ensure values are in valid range

        if pad_width is not None:
            X = X[pad_width:-pad_width, pad_width:-pad_width]

    except np.linalg.LinAlgError as e:
        print(f"Linear algebra error in LU method: {str(e)}")
        X = np.zeros_like(B if pad_width is None else B[pad_width:-pad_width, pad_width:-pad_width])
        return X, time.time() - start_time
    except Exception as e:
        print(f"Unexpected error in LU method: {str(e)}")
        X = np.zeros_like(B if pad_width is None else B[pad_width:-pad_width, pad_width:-pad_width])
        return X, time.time() - start_time

    return X, time.time() - start_time
```

Coding 2 LU decomposition for part (b)

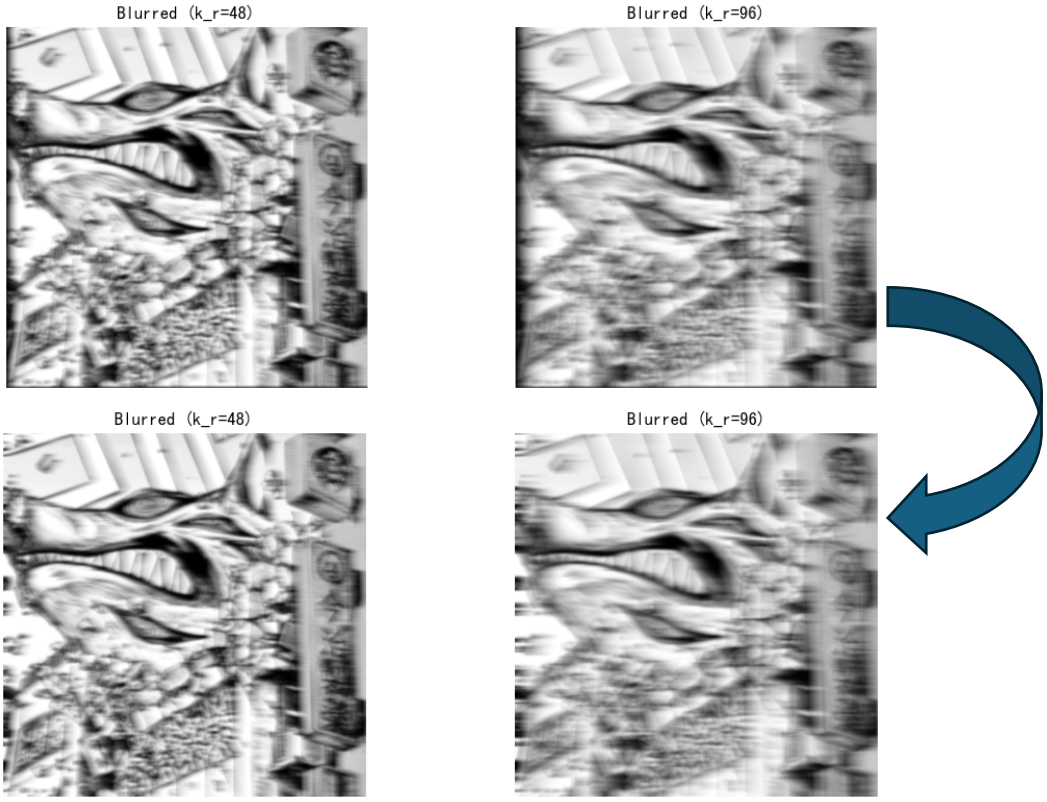
```python
def deblur_qr(B, A_l, A_r,pad_width=None):

    start_time = time.time()

    try:
        Q_l, R_l = scipy.linalg.qr(A_l)
        Q_r, R_r = scipy.linalg.qr(A_r)

        Y = np.linalg.solve(R_l, Q_l.T @ B)

        Rr_inv = np.linalg.inv(R_r)

        XQr = Y @ Rr_inv

        X = XQr @ Q_r.T

        X = np.clip(X, 0, 1)  # Ensure values are in valid range

        if pad_width is not None:
            X = X[pad_width:-pad_width, pad_width:-pad_width]

    except np.linalg.LinAlgError as e:
        print(f"Linear algebra error in QR method: {str(e)}")
        X = np.zeros_like(B if pad_width is None else B[pad_width:-pad_width, pad_width:-pad_width])
        return X, time.time() - start_time
    except Exception as e:
        print(f"Unexpected error in QR method: {str(e)}")
        X = np.zeros_like(B if pad_width is None else B[pad_width:-pad_width, pad_width:-pad_width])
        return X, time.time() - start_time

    return X, time.time() - start_time
```

Coding 6: LU decomposition for part (b)

Image	Condition Number (A_l)	Condition Number (A_r)	LU Deblurring (Time)	LU Deblurring (PSNR)	QR Deblurring (Time)	QR Deblurring (PSNR)
256x256 (kr48)	1.30E+01	3.41E+10	2.834s	12.01 dB	1.109s	5.01 dB
256x256 (kr96)	1.30E+01	4.58E+09	0.885s	9.49 dB	0.351s	4.75 dB
1024x1024 (kr48)	1.30E+01	1.82E+17	1.617s	10.74 dB	0.629s	Error
1024x1024 (kr96)	1.30E+01	2.66E+18	3.626s	9.89 dB	2.247s	4.90 dB
1250x1250 (kr48)	1.30E+01	7.62E+16	1.902s	10.32 dB	1.015s	5.31 dB
1250x1250 (kr96)	1.30E+01	1.91E+18	1.709s	7.92 dB	1.068s	4.01 dB
3500x3500 (kr48)	1.30E+01	1.71E+18	6.574s	12.19 dB	10.197s	8.31 dB
3500x3500 (kr96)	1.30E+01	4.52E+17	6.717s	11.26 dB	9.735s	7.59 dB

Table 2: Detailed results for part (b)

Why Padding



```
"""
 1. Optimize QR Deblurring Method using Tikhonov Regularization
"""

def solve_least_squares(B, A_l, A_r, lambda_reg=1e-6):
    """
    Solve min ||A_l X A_r - B||_F^2 + lambda||X||_F^2
    using normal equations
    """
    m, n = B.shape

    # Form A^TA and A^Tb without explicitly forming Kronecker product
    def matvec(x):
        """Compute (A_r^T ⊗ A_l)^T(A_r^T ⊗ A_l)x efficiently"""
        X = x.reshape(m, n)
        Y = A_l.T @ (A_l @ X @ A_r) @ A_r.T + lambda_reg * X
        return Y.reshape(-1)

    # Form right-hand side without explicitly forming Kronecker product
    rhs = A_l.T @ B @ A_r.T
    rhs = rhs.reshape(-1)

    # Solve normal equations using conjugate gradient
    x, info = splinalg.cg(
        splinalg.LinearOperator((m*n, m*n), matvec=matvec),
        rhs,
        tol=1e-10,
        maxiter=10000
    )

    if info != 0:
        print(f"Warning: CG did not converge, info={info}")

    # Reshape solution back to matrix form
    X = x.reshape(m, n)
    return np.clip(X, 0, 1)
```

Coding 7: Tikhonov Regularization with Pre-set Regularization Parameter

```
def get_reg_parameter(A_l, A_r):

    cond_l = np.linalg.cond(A_l)
    cond_r = np.linalg.cond(A_r)
    n = A_l.shape[0]

    base_lambda = 1.0 / np.sqrt(max(cond_l, cond_r))

    if max(cond_l, cond_r) > 1e10:
        base_lambda *= 1e5
    elif max(cond_l, cond_r) > 1e5:
        base_lambda *= 1e3

    size_factor = np.log10(n) / 3
    base_lambda *= size_factor

    lambda_reg = np.clip(base_lambda, 1e-6, 1e-2)

    print(f"Condition numbers - A_l: {cond_l:.2e}, A_r: {cond_r:.2e}")
    print(f"Selected lambda: {lambda_reg:.2e}")

    return lambda_reg

def deblur_qr_auto(B, A_l, A_r,pad_width):

    lambda_reg = get_reg_parameter(A_l, A_r)

    return deblur_qr_regularized(B, A_l, A_r, lambda_reg, pad_width)
```

Coding 9: Tikhonov Regularization with Auto Adaptive Regularization Parameter-2

```
"""
2.
 Auto adaptive regularization parameters

    The regularization parameters are selected adaptively according to the problem characteristics:
    i)  The base lambda value is based on the condition number
    ii) The ill-posed problem requires stronger regularization, adjust the base lambda by multiplying the adjustment coefficient
    iii) Make sure that lambda is within a reasonable range

"""
def deblur_qr_regularized(B, A_l, A_r, lambda_reg=1e-1,pad_width=None):
    try:
        start_time = time.time()

        Q_l, R_l = linalg.qr(A_l)
        Q_r, R_r = linalg.qr(A_r)

        Y1 = Q_l.T @ B
        Z = solve(R_l.T @ R_l + lambda_reg * np.eye(R_l.shape[1]),
                  R_l.T @ Y1,
                  check_finite=True)

        Y2 = Q_r.T @ Z.T
        X = solve(R_r.T @ R_r + lambda_reg * np.eye(R_r.shape[1]),
                  R_r.T @ Y2,
                  check_finite=True).T

        X = np.clip(X, 0, 1)

        if pad_width is not None:
            X = X[pad_width:-pad_width, pad_width:-pad_width]

        return X, time.time() - start_time

    except Exception as e:
        print(f"Error in regularized QR method: {str(e)}")
        return np.zeros_like(B), 0
```

Coding 8: Tikhonov Regularization with Auto Adaptive Regularization Parameter-1

```python
def iterative_deblur(B, A_l, A_r, lambda_reg=1e-3, max_iter=50, tol=1e-4):

    X = np.zeros_like(B)
    prev_X = np.zeros_like(B)

    # Precompute matrices for efficiency
    A_l_T = A_l.T
    A_r_T = A_r.T

    # Gradient descent with momentum
    momentum = 0.9
    v = np.zeros_like(B)
    step_size = 0.1  # Initial step size

    for i in range(max_iter):
        # Compute residual
        residual = A_l @ X @ A_r - B

        # Compute gradient
        gradient = A_l_T @ residual @ A_r_T + lambda_reg * X

        # Update with momentum
        v = momentum * v - step_size * gradient
        X = X + v

        # Compute relative change
        rel_change = np.linalg.norm(X - prev_X) / np.linalg.norm(X)
        if rel_change < tol:
            print(f"Converged after {i+1} iterations")
            break

        prev_X = X.copy()

        # Project to ensure valid image values
        X = np.clip(X, 0, 1)

    return X
```

Coding 10: Iterative deblurring algorithm

```python
def trunc_level_adaptive(s):
    """Determine truncation level adaptively based on singular values"""
    # Find where singular values drop significantly
    ratios = s[1:] / s[:-1]
    drop_idx = np.where(ratios < 0.1)[0]
    if len(drop_idx) > 0:
        return drop_idx[0] + 1
    return len(s) // 2  # Default to half if no clear drop
```

Coding 11: Truncation methods for SVD decomposition

```python
def truncated_svd_deblur(B, A_l, A_r, trunc_ratio=0.5):
    """Deblurring using truncated SVD with consistent dimensions"""
    U_l, s_l, Vh_l = np.linalg.svd(A_l, full_matrices=False)
    U_r, s_r, Vh_r = np.linalg.svd(A_r, full_matrices=False)

    # Determine truncation rank adaptively
    n = A_l.shape[0]
    k_max = min(n, int(n * trunc_ratio))
    k_l = min(k_max, len(s_l))
    k_r = min(k_max, len(s_r))
    k = min(k_l, k_r)

    print(f"Using rank {k} out of {n} possible components")

    # Truncate components
    U_l_t = U_l[:, :k]
    s_l_t = s_l[:k]
    Vh_l_t = Vh_l[:k, :]

    U_r_t = U_r[:, :k]
    s_r_t = s_r[:k]
    Vh_r_t = Vh_r[:k, :]

    # Print shapes for debugging
    print(f"U_l_t shape: {U_l_t.shape}")
    print(f"Vh_l_t shape: {Vh_l_t.shape}")
    print(f"U_r_t shape: {U_r_t.shape}")
    print(f"Vh_r_t shape: {Vh_r_t.shape}")
    print(f"B shape: {B.shape}")

    # Transform B with regularization
    B_transformed = U_l_t.T @ B @ U_r_t

    # Create regularization matrix
    outer_prod = np.outer(s_l_t, s_r_t)
    reg_matrix = outer_prod / (outer_prod**2 + 1e-10)

    # Solve the transformed system
    X_transformed = B_transformed * reg_matrix

    # Transform back
    X = U_l_t @ X_transformed @ U_r_t.T

    return np.clip(X, 0, 1)
```

Coding 12: SVD Algorithm with Truncation