Università della Svizzera italiana
Institute of Computing CI

Numerical Computing **2024**

Student: POLAD BAKHISHZADE

Solution for Project 4 **Due date:** Wednesday, 27 November 2024, 11:59 PM

---

1. General Questions [10 points]

1.1. Size of matrix A

The transformation matrix A has a size of $n^2 \times n^2$, where n represents the number of pixels along one side of the image. For $n = 250$, the matrix size is computed as:

$$\text{Size of } A = 250^2 \times 250^2 = \mathbf{62500} \times \mathbf{62500}.$$

This size can be verified both manually and programmatically through MATLAB.

The following MATLAB code computes and displays the size of the matrix A:

```
load("blur_data/A.mat");
sizeA = size(A);
fprintf("Size of A is %d x %d \n", sizeA(1), sizeA(2));
```

Running this code confirms that the size of A is **62500 x 62500**.

1.2. Number of diagonal bands in matrix A

The transformation matrix A is constructed using a kernel. A kernel is a small $d \times d$ matrix that determines how the value of each pixel is influenced by its neighbors. For example, in image blurring, the kernel specifies weights for averaging the pixel's intensity (brightness) with the intensities of nearby pixels.

In this project, the kernel is 7×7 in size ($d = 7$), meaning it operates on a grid of 49 neighboring pixels. The matrix A is sparse and banded, meaning most entries are zero except for non-zero elements concentrated along the main diagonal and adjacent diagonals. The total number of these diagonals (referred to as diagonal bands) is given by:

$$\text{Number of diagonal bands in } A = d^2 = 7^2 = \mathbf{49}.$$

1.3. Length of vectorized blurred image b

The blurred image B is a grayscale image represented as an $n \times n$ matrix, where each entry corresponds to the intensity of a pixel. Pixel intensity refers to the brightness of a pixel, where 0 represents black, and higher values (e.g., 255 in an 8-bit grayscale image) represent lighter shades of gray.

To perform mathematical operations such as solving the system $Ax = b$, the 2D matrix B needs to be transformed into a 1D column vector $b \in \mathbb{R}^{n^2}$. This transformation, called vectorization, involves stacking the rows of B one after another to create a single column. For example:

$$B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \quad \text{becomes} \quad b = \begin{bmatrix} b_{11} \\ b_{12} \\ b_{21} \\ b_{22} \end{bmatrix}.$$

For an image of size 250×250 ($n = 250$), the length of the vectorized image b is:

$$\text{Length of } b = n^2 = 250 \times 250 = \mathbf{62500}.$$

2. Properties of A [10 points]

2.1. Effect of A not being symmetric on $\tilde{A}$

If the transformation matrix A is not symmetric, solving the linear system $Ax = b$ using the Conjugate Gradient (CG) method becomes infeasible because CG requires A to be symmetric and positive-definite. To handle this, we create an augmented system:

$$\tilde{A}x = \tilde{b},$$

where:

$$\tilde{A} = A^T A \quad \text{and} \quad \tilde{b} = A^T b.$$

The matrix $\tilde{A}$ is always symmetric because it is the product of a matrix A and its transpose A^T, a property that holds for any matrix. Symmetry is necessary for CG to work.
The condition number $\kappa(A)$, which measures the sensitivity of a linear system to changes in b, becomes squared when using $\tilde{A}$:

$$\kappa(\tilde{A}) = \kappa(A)^2.$$

This increase in κ worsens the ill-conditioning of the system, leading to slower convergence of the CG method. Preconditioning can help mitigate this.

2.2. Why solving $Ax = b$ is equivalent to minimizing $\frac{1}{2}x^T Ax - b^T x$

We need to show that solving $Ax = b$ is equivalent to minimizing:

$$f(x) = \frac{1}{2}x^T Ax - b^T x.$$

Let us compute the derivative:

$$f'(x) = \frac{d}{dx}\left(\frac{1}{2}x^T Ax - b^T x\right).$$

Expanding this, we get:

$$f'(x) = \frac{1}{2}A^T x + \frac{1}{2}Ax - b.$$

Since A is symmetric ($A^T = A$), this simplifies to:

$$f'(x) = \frac{1}{2}Ax + \frac{1}{2}Ax - b = Ax - b.$$

The minimum of $f(x)$ occurs when $f'(x) = 0$, which gives:

$$Ax - b = 0 \quad \Rightarrow \quad Ax = b.$$

Thus, we have shown that solving $Ax = b$ is equivalent to minimizing $f(x) = \frac{1}{2}x^T Ax - b^T x$.

3. Conjugate Gradient [30 points]

3.1. Conjugate Gradient Solver Implementation

The Conjugate Gradient (CG) method is an iterative technique for solving systems of linear equations of the form $Ax = b$, where A is a symmetric and positive-definite matrix. In this context, A is symmetric if $A = A^T$, meaning that the matrix is equal to its transpose, a property ensuring that its eigenvalues are real. A matrix is positive-definite when all its eigenvalues are positive, guaranteeing that the system has a unique solution.

The method begins with an initial guess x_0 and computes the residual $r = b - Ax$, which represents the error in the current approximation. Smaller residuals indicate closer approximations to the true solution. The dot product of r with itself, stored as p_old in the code, measures the size of the residual and is used to compute key parameters such as the step size α and the scaling factor β for updating the search direction. The algorithm iteratively minimizes the residual until either convergence is achieved (when $\sqrt{p_{\text{new}}} \leq$ `tol`) or the maximum number of iterations (`max_itr`) is reached.

The implementation of the CG method is provided below:

```
function [x, rvec] = myCG(A, b, x0, max_itr, tol)
    x = x0;
    r = b - A * x0;
    d = r;
    rvec = [];
    p_old = dot(r, r);

    for k = 1:max_itr
        s = A * d;
        alpha = p_old / dot(d, s);
        x = x + alpha * d;
        r = r - alpha * s;
        p_new = dot(r, r);
        rvec = [rvec, sqrt(p_new)];
        if sqrt(p_new) <= tol
            break;
        end
        beta = p_new / p_old;
        d = r + beta * d;
        p_old = p_new;
    end
end
```

The inputs to the solver include A, which is the symmetric and positive-definite system matrix, b, the right-hand side vector ($Ax = b$), x_0, the initial guess, `max_itr`, the maximum number of iterations, and `tol`, the tolerance for convergence. The outputs are x, the approximate solution to $Ax = b$, and `rvec`, a vector storing the square root of the dot product of r with itself ($\sqrt{\text{dot}(r, r)}$) at each iteration. This represents the magnitude of the residual, effectively providing a single scalar that quantifies how far the solution is from being correct at the current iteration.

3.2. Validation of the Conjugate Gradient Solver

To validate the implementation of the Conjugate Gradient (CG) method, we solve a system defined by the test matrices A_test and b_test. The goal is to analyze the behavior of the algorithm and assess its convergence by plotting the residual magnitude at each iteration.

The system is solved using the myCG as follows:

```
load('test_data/A_test.mat', 'A_test');
load('test_data/b_test.mat', 'b_test');

m = sqrt(length(b_test));
n = m;

guess = zeros(size(A_test, 1), 1);
max_itr = 200;
tol = 1e-4;

[x, rvec] = myCG(A_test, b_test, guess, max_itr, tol);
figure;
plot(rvec);
xlabel("Iterations");
ylabel("Residual Values");
set(gca, "YScale", "log");
title("Convergence of myCG");
```

The residual magnitudes (`rvec`) are plotted against the iteration count in a logarithmic scale to evaluate the convergence behavior. The convergence plot is shown below:

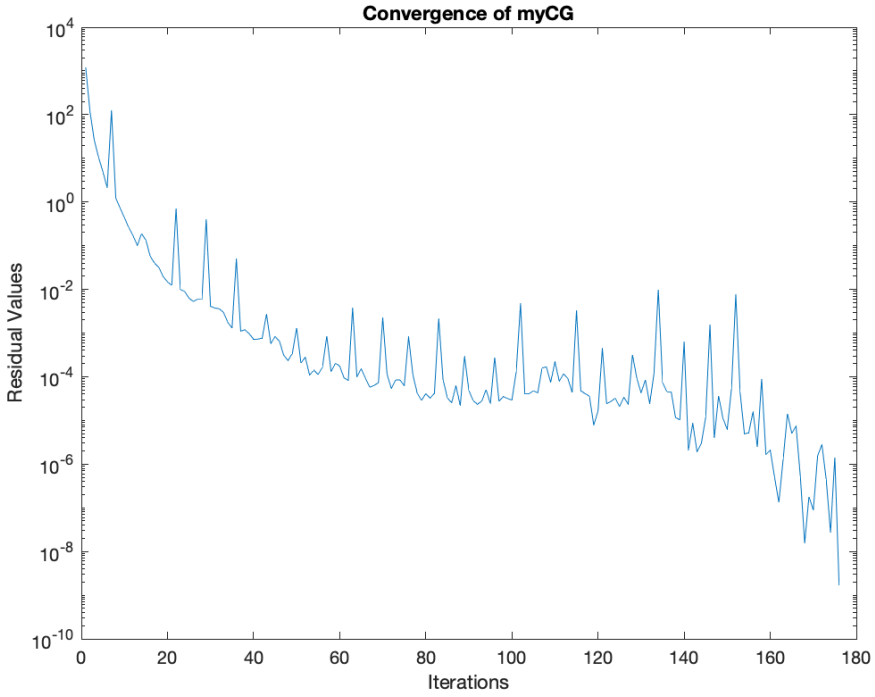


Figure 1: Convergence of the Conjugate Gradient Method. The plot shows the residual values decreasing as the iteration count increases, with the scale set to logarithmic.

The convergence plot demonstrates a steady decrease in the residual magnitude with iterations, which indicates that the method is working correctly. The logarithmic scale reveals that the residual values decrease exponentially as the algorithm approaches the solution, a behavior expected for well-conditioned systems. The stopping condition $\sqrt{\text{dot}(r,r)} \leq$ `tol` ensures that the solution is sufficiently accurate within the specified tolerance (1e-4).

3.3. Eigenvalues, Condition Number, and Convergence Rate

In this exercise, we analyze the eigenvalues of the matrix A_test.mat to estimate its condition number and discuss its effects to the convergence rate of the Conjugate Gradient (CG) method.

The condition number (A) is a property on linear systems of equations. It measures how much the result of the equation changes with respect to the input arguments. A system with a high condition number is called ill-conditioned, implying that it is harder to compute a solution. A system with a low condition number is instead called well-conditioned.

The eigenvalues of A_{test} exhibit a significant spread, as illustrated in the following plot:

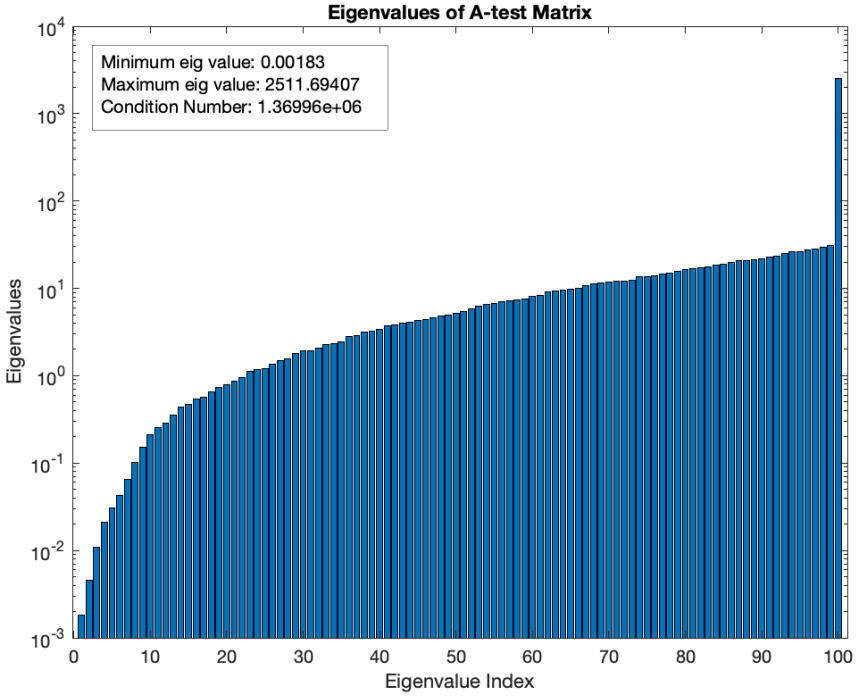


Figure 2: Eigenvalue distribution of A_{test} on a logarithmic scale. The wide range of eigenvalues indicates a high condition number.

The condition number $\kappa(A)$ is calculated using the ratio of the largest eigenvalue $\lambda_{\max}$ to the smallest eigenvalue $\lambda_{\min}$:

$$\kappa(A) = \frac{\lambda_{\max}}{\lambda_{\min}}.$$

For A_{test}, the eigenvalues were computed as:

$$\lambda_{\max} = 2511.69, \quad \lambda_{\min} = 0.001833.$$

The condition number is therefore:

$$\kappa(A) \approx 1.37 \times 10^6.$$

The high condition number indicates that A_{test} is ill-conditioned. This has two key implications. First, small changes in the right-hand side vector b ($Ax = b$) can cause significant changes in the solution x. Second, the convergence of the CG method is slowed due to the large spread of eigenvalues.

The convergence rate describes how quickly the error (residual) $r_k = b - Ax_k$ decreases as the CG method progresses through iterations. The convergence rate of the CG method is directly influenced by $\kappa(A)$. Specifically, the number of iterations required to achieve convergence is proportional to $\sqrt{\kappa(A)}$, meaning that a high condition number in an ill-conditioned system results in a significantly lower convergence rate. Preconditioning is an effective strategy to address this issue, as it narrows the range of eigenvalues and reduces $\kappa(A)$, thereby improving convergence speed.

3.4. Does the Residual Decrease Monotonically?

From the plotted residual values, it is clear that residual decreases overall as the CG method progresses in iterations but does not do so strictly monotonically. Monotonicity refers to a consistently decreasing trend in the residual values, where no temporary increases occur. Disruptions to the conjugacy of search directions, which are critical for efficient residual reduction, can lead to temporary increases in the residual. Conjugacy ensures that each search direction minimizes the residual independently, without undoing progress made in earlier steps. However, ill-conditioning disrupts conjugacy by causing the algorithm to revisit components that should have been resolved in earlier iterations. This inefficiency can result in temporary increases in the residual, though the overall trend remains a steady decrease.

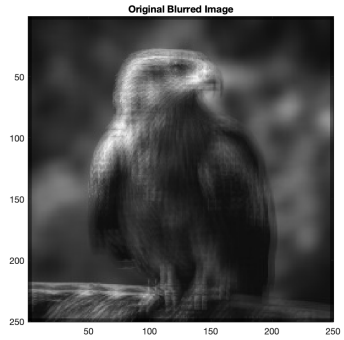
4. Deblurring problem [35 points]

4.1 Preconditioning using the incomplete Cholesky factorization

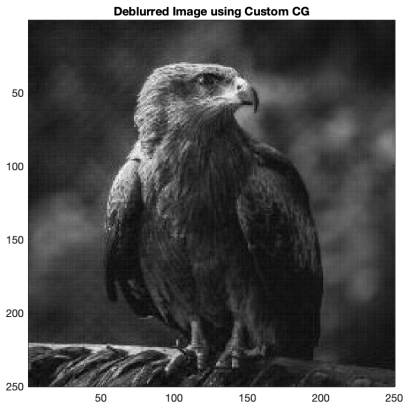
This exercise focuses on solving the deblurring problem using both the custom Conjugate Gradient (CG) solver myCG and Preconditioned CG (pcg) solver. The deblurring process aims to reconstruct the original image X from a blurred image B using the known transformation matrix A.

The problem involves solving the system $Ax = b$, where b is the vectorized form of the blurred image B, and x is the vectorized solution representing the deblurred image X. The solvers were configured with the following parameters, maximum iterations: 200, tolerance: 10^{-6}, preconditioner for PCG: incomplete Cholesky (`ichol`) with no-fill and a diagonal shift $\alpha = 0.01$.

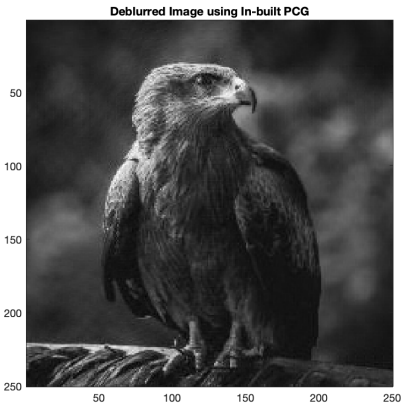
The images produced include the blurred image, the deblurred image from `myCG`, and the deblurred image from `pcg`:



(a) Original Blurred Image



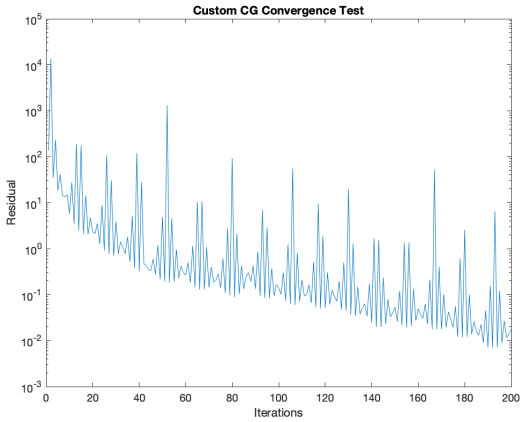
(b) Deblurred Image using `myCG`



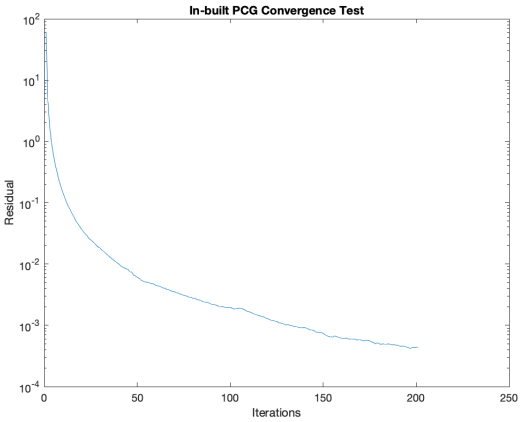
(c) Deblurred Image using `pcg`

Figure 3: Image Deblurring Results

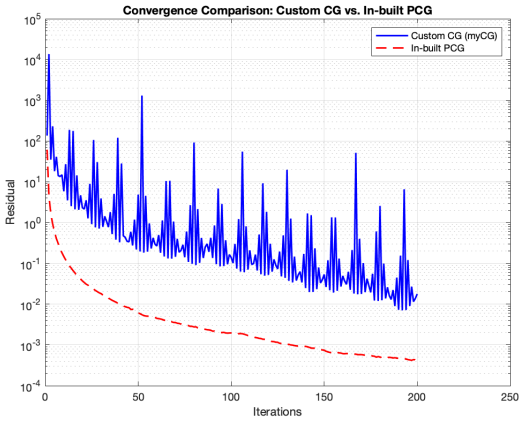
Additionally, the following convergence plots compare the residual variation and iterations for both solvers myCG and PCG:



(a) Convergence of `myCG`



(b) Convergence of `pcg`



(c) Convergence Comparison: `myCG and pcg`

Figure 4: Convergence Plots for both solvers `myCG and pcg`

Both solvers produce high-quality deblurred images, effectively reconstructing the original image from the blurred input. The results from both the `myCG` solver and the `pcg` solver are visually indistinguishable, demonstrating the reliability of these methods.

The `myCG` solver shows a decreasing residual norm with occasional fluctuations due to ill-conditioning in the matrix A. These fluctuations disrupt the conjugacy of search directions, affecting its efficiency. However, `myCG` is simple to implement and performs well for moderately conditioned systems, where the condition number $\kappa(A)$ is not excessively large.

In contrast, the `pcg` solver achieves a strictly decreasing residual norm due to the incomplete Cholesky preconditioner, which narrows the eigenvalue range of A. This reduces the condition number, improving convergence speed. The comparison plot shows that `pcg` converges faster and more smoothly than `myCG`, especially for ill-conditioned systems.

4.2 When would pcg be worth the added computational cost? What about if you are deblurring lots of images with the same blur operator?

Both solvers successfully solve the deblurring problem, but the preconditioned `pcg` method is more robust for ill-conditioned systems. The preconditioner enhances convergence by narrowing the eigenvalue range of the system matrix A. This makes the `pcg` solver particularly valuable for *large-scale problems*, which involve matrices and images with high dimensions (e.g., large-resolution images). In such cases, the computational cost of solving $Ax = b$ is high due to the size of the data.

The preconditioner reduces this cost by speeding up convergence. Additionally, `pcg` is advantageous for *repeated deblurring tasks*, such as when multiple blurred images share the same system matrix A. In these scenarios, *precomputing* the preconditioner—meaning the preconditioner is calculated once and reused across multiple runs—can save significant computational effort and improve efficiency. Despite the initial computational effort required to construct the preconditioner, its reuse across multiple tasks justifies the cost. PCG's compatibility with modern parallel computing architectures, such as *GPU-accelerated systems* or *multi-core CPUs*, enhances its scalability and performance. For instance, NVIDIA CUDA-enabled GPUs can distribute matrix-vector multiplications and preconditioning operations across thousands of cores, significantly reducing computation time for large-scale tasks. An example of a *large-scale problem* is deblurring high-resolution satellite images, where the size of the transformation matrix A and the vectorized form of the image x can involve millions of entries. Such problems demand efficient solvers like PCG to ensure timely and accurate solutions without excessive computational cost.