Università della Svizzera italiana

Institute of Computing CI

Numerical Computing **2023**

Student: Kyla Kaplan **Discussed with:**

Solution for Project 4 **Due date:** Wednesday, 6 December 2023, 11:59 PM

Numerical Computing 2023 — Submission Instructions
(Please, notice that following instructions are mandatory: submissions that don't comply with, won't be considered)

- Assignments must be submitted to iCorsi (i.e. in electronic format).
- Provide both executable package and sources (e.g. C/C++ files, MATLAB). If you are using libraries, please add them in the file. Sources must be organized in directories called:

  Project_number_lastname_firstname

  and the file must be called:

  project_number_lastname_firstname.zip
  project_number_lastname_firstname.pdf
- The TAs will grade your project by reviewing your project write-up, and looking at the implementation you attempted, and benchmarking your code's performance.
- You are allowed to discuss all questions with anyone you like; however: (i) your submission must list anyone you discussed problems with and (ii) you must write up your submission independently.

1. General Questions [10 points]

1. **What is the size of matrix A?**

   The size of matrix A (according to the given file) is $62500 * 62500$. We know the value of n is 250, and A is a $n^2 * n^2$ matrix, therefore A is $250^2 * 250^2 = 62500 * 62500$.

2. **How many diagonal bands does A have?**

   Since we have given the matrix kernel of size $(n * n)$ $7 * 7$, and since d can be less than OR equal to n; by setting d to 7, the number of diagonal bands of A would be equal to 49 (which is d^2 as a d-matrix).

3. **What is the length of the vectorized blurred image b?**

   Since matrix A operates on a 'row-vectorized' matrix of the image, the length of b would be the $n * n$, and with $n = 250$, b would be 625000.

2. Properties of A [10 points]

1. **If A is not symmetric, how would this affect $\tilde{A}$?**

   In the Conjugate Gradient (CG) method, the matrix A must be symmetric positive definite for the standard formulation of the algorithm. If A is not symmetric, you typically use the preconditioned Conjugate Gradient method with a symmetric positive definite matrix M s.t. $M^{-1}A$ is symmetric (or nearly symmetric).

   In the standard CG algorithm, $\tilde{A}$ (the matrix used in the algorithm) is defined as A, and the method exploits the symmetry of A to optimize the computation. When A is not symmetric, you may introduce a preconditioner M and solve the system $M^{-1}Ax = M^{-1}b$ instead.

   $$A^T Ax = A^T b$$

   This is because the CG method needs A to be positive-definite. Since $k(\tilde{A}) = k(A)^2$, the conditional number (k) also gets affected. We can also assume A is symmetric if its complement is a square matrix. Then,

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

2. **Explain why solving $Ax = b$ for x is equivalent to minimizing $\frac{1}{2}x^T Ax - b^T x$ over x, assuming that A is symmetric positive-definite.**

   We know that A is symmetric, therefore $A = A^T$, and we can compute the derivative of the formula and find the minimum (where $= 0$):

   $$\begin{aligned} f'(x) &= \frac{d}{dx}(\frac{1}{2}x^T Ax - b^T x) \\ &= \frac{1}{2}A^T x + \frac{1}{2}Ax - b \\ &= \frac{1}{2}Ax + \frac{1}{2}Ax - b \\ &= Ax - b \end{aligned}$$

   Then, by equating it to 0:

   $$Ax - b = 0$$

   $$Ax = b$$

So, when A is symmetric positive-definite, solving $Ax = b$ is equivalent to minimizing $\frac{1}{2}x^T Ax - b^T x$ over x. This equivalence is a result of the quadratic form representing a convex quadratic function, and the critical point of this function corresponds to the solution of the linear system. The positive-definiteness of A ensures the existence of a unique minimum.

3. Conjugate Gradient [30 points]

1. **Write a function for the conjugate gradient solver, where *x* and *rvec* are, respectively, the solution value and a vector containing the residual at every iteration.**

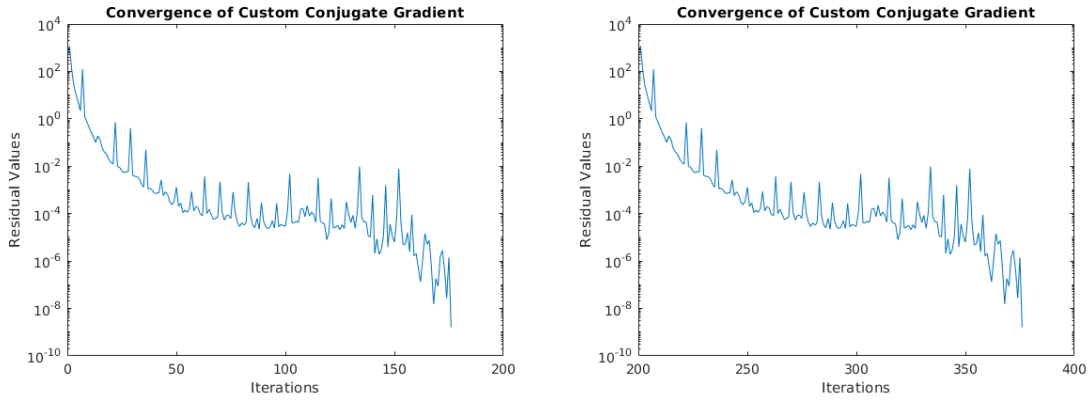
   Implementation can be found in the attached files.

```
function [x, rvec] = myCG(A, b, x0, max_itr, tol)
    % Initialize variables
    rvec = zeros(1, max_itr); % Preallocate rvec;
    rvec = [];                % Not pre-allocated
    x = x0;
    residual = b - A * x;
    direction = residual;
    residual_norm_old = dot(residual, residual);

    % Conjugate Gradient iteration
    for itr = 1:max_itr
        % Compute matrix-vector product
        product = A * direction;

        % Update solution
        alpha = residual_norm_old / dot(direction, product);
        x = x + alpha * direction;

        % Update residual
        residual = residual - alpha * product;

        % Update squared residual norm
        residual_norm_new = dot(residual, residual);

        % Update conjugate direction
        beta = residual_norm_new / residual_norm_old;
        direction = residual + beta * direction;

        % Update previous squared residual norm
        residual_norm_old = residual_norm_new;

        % Store squared residual norms for analysis
        rvec = [rvec, residual_norm_new];

        % Check convergence
        if sqrt(residual_norm_new) <= tol
            disp('Converged');
            break;
        end
    end
end
```

   As we will see later in the convergence test, the amount of iterations changed depending on the pre-allocation of *rvec*. Matlab rose a warning about the pre-allocation for *rvec*, so I include both results to show the difference in optimization.

2. **In order to validate your implementation, solve the system defined by *A_test.mat***

and *b_test.mat*. Plot the convergence (residual vs iteration).



(a) (b)

Figure 1: Plot of the residual values against iteration using custom Conjugate Gradient function (a) without a pre-allocated *rvec* (b) with pre-allocated *rvec*

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

% Determine matrix dimensions
[m, n] = size(A_test);

% Initial guess and iteration parameters
initialGuess = zeros(n, 1);
maxIterations = 200; %above
tolerance = 1e-4;

% Solve using custom Conjugate Gradient method
[x, rvec] = myCG(A_test, b_test, initialGuess, maxIterations, tolerance);

% Plot the convergence
figure;
plot(rvec);
xlabel('Iterations');
ylabel('Residual Values');
set(gca, 'YScale', 'log');
title('Convergence of Custom Conjugate Gradient');
```

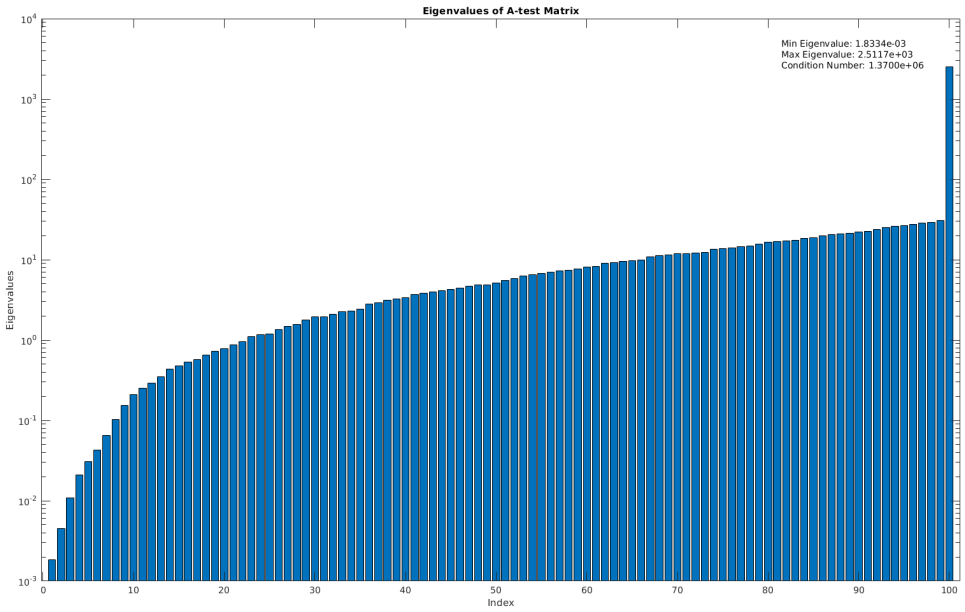
3. **Plot the eigenvalues of A_test.mat and comment on the condition number and convergence rate.**

   Implementation can be found in the attached files.

   This code uses a threshold ($threshold = 1e^{-8}$) to exclude eigenvalues below a certain magnitude, and then only includes the valid eigenvalues in the plot and calculations.

```
% Compute eigenvalues of A_test
eigenvalues = eig(A_test);

% Set a threshold for including eigenvalues in the plot
threshold = 1e-8;
validIndices = abs(eigenvalues) > threshold;

% Plot the eigenvalues
```

```
figure;
bar(1:length(eigenvalues(validIndices)), abs(eigenvalues(validIndices)));
set(gca, 'YScale', 'log');
xlabel('Index');
ylabel('Eigenvalues');
title('Eigenvalues of A-test Matrix');

% Compute min, max, and condition number of eigenvalues
minEigenvalue = min(abs(eigenvalues(validIndices)));
maxEigenvalue = max(abs(eigenvalues(validIndices)));
conditionNumber = maxEigenvalue / minEigenvalue;

% Display information on the side of the plot
annotation('textbox', [0.75, 0.6, 0.1, 0.3], 'String', sprintf('Min Eigenvalue:
    %.4e\nMax Eigenvalue: %.4e\nCondition Number: %.4e', minEigenvalue,
    maxEigenvalue, conditionNumber), 'FitBoxToText', 'on', 'EdgeColor', 'none');
```



(a)

Figure 2: Plot of eigenvalue distribution for matrix A_test.mat

The plotted eigenvalues exhibit a significant spread, with the largest and smallest absolute eigenvalues contributing to a substantial condition number, expressed as $k(A) = \frac{\sigma_{max}}{\sigma_{min}}$. This condition number is a key determinant of the Conjugate Gradient (CG) method's convergence behavior.

The formula for the convergence rate, denoted as $CR = \sqrt{k(A)}$, establishes a clear relationship between the condition number and the speed of convergence in iterative solvers like CG. A higher condition number implies a slower convergence rate, signaling potential challenges in finding a solution. In practical terms, a matrix with a high condition number is labeled as "ill-conditioned", indicating sensitivity to numerical precision issues.

In scenarios where the condition number is notably high, a technique called preconditioning becomes essential. Preconditioning involves adjusting the matrix to improve its conditioning, thereby addressing the challenges posed by ill-conditioning on the efficiency of iterative solvers. This strategic adjustment is particularly relevant when the inherent characteristics

of the matrix impede the convergence performance of numerical algorithms.

4. **Does the residual decrease monotonically? Why or why not?**

   Although not strictly monotonically, I observe that the residual plot of the Conjugate Gradient (CG) method displays a decreasing trend from Figure 1, aligning with the expected behavior of the algorithm. The CG method aims to iteratively minimize the residual, gradually approaching convergence towards a solution. The decreasing pattern in the residual plot is indicative of the algorithm's progress in reducing the error between consecutive approximations.

   While some fluctuations may occur due to factors such as orthogonality properties, numerical precision, and eigenvalue distribution, the overall trajectory reflects the successful reduction of the residual towards convergence. The persistent decline in the residual values supports the efficacy of the CG method in iteratively improving the solution approximation.

4. Deblurring problem [35 points]

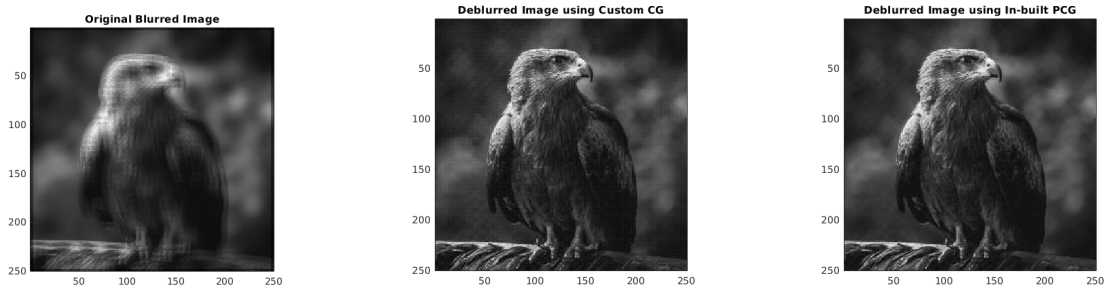
1. **Solve the deblurring problem for the blurred image matrix *B.mat* and transformation matrix *A.mat* using your routine *myCG* and Matlab's preconditioned conjugate gradient *pcg*. As a preconditioner, use *ichol* to get the incomplete Cholesky factors and set routine type to nofill with $\alpha = 0.01$ for the diagonal shift (see Matlab documentation). Solve the system with both solvers using $max\_iter = 200$ and $tol = 10^6$. Plot the convergence (residual vs iteration) of each solver and display the original and final deblurred image. Comment on the results that you observe.**

   Implementation can be found in the attached files.

```
% Load blurred image and create vectorized version
load('blur_data/B.mat', 'B');
img = B;
height = size(img, 1);

% Vectorize the blurred image
b = B(:);

% Load system matrix A and set up parameters
load('blur_data/A.mat', 'A');
guess = ones(size(b));
max_iterations = 200;
tolerance = 1e-6;
alpha = 0.01;

% Configure options for incomplete Cholesky preconditioner
options.type = 'nofill';
options.diagcomp = alpha;

% Form the normal equations and apply the incomplete Cholesky preconditioner
height_A = transpose(A) * A;
height_B = transpose(A) * b;
Q = ichol(height_A, options);
P = transpose(Q) * Q;

% Solve the linear system using custom CG
[x_myCG, rvec_myCG] = myCG(A, b, guess, max_iterations, tolerance);

% Solve the preconditioned linear system using in-built PCG
[x_pcg, ~, ~, iter, rvec_pcg] = pcg(height_A, height_B, tolerance, max_iterations);
```

```
% Display the original blurred image
figure;
imagesc(reshape(img, [height, height]));
colormap gray;
title('Original Blurred Image');
axis image;

% Display the deblurred image using custom CG
figure;
imagesc(reshape(x_myCG, [height, height]));
colormap gray;
title('Deblurred Image using Custom CG');
axis image;

% Display the deblurred image using in-built PCG
figure;
imagesc(reshape(x_pcg, [height, height]));
colormap gray;
title('Deblurred Image using In-built PCG');
axis image;

% Display the convergence plot for custom CG
figure;
semilogy(rvec_myCG);
title('Custom CG Convergence Test');
xlabel('Iterations');
ylabel('Residual');

% Display the convergence plot for in-built PCG
figure;
semilogy(rvec_pcg);
title('In-built PCG Convergence Test');
xlabel('Iterations');
ylabel('Residual');
```

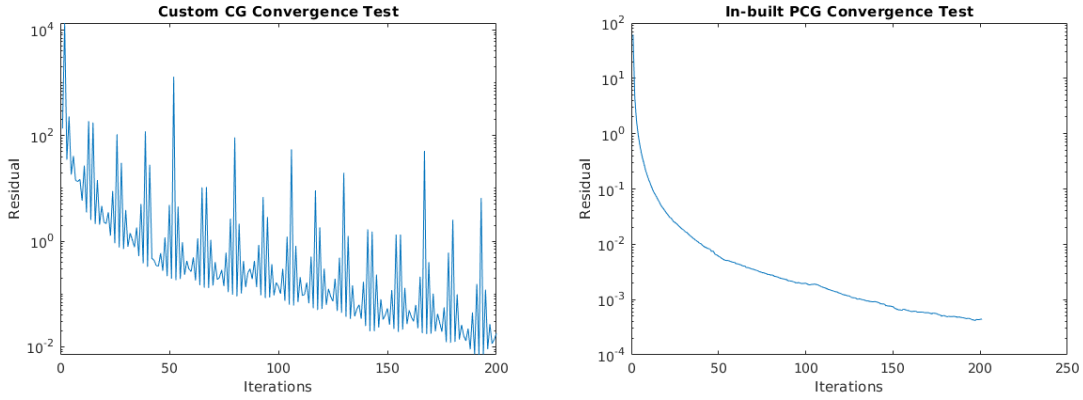
Producing the following images:



(a) (b) (c)

Figure 3: (a) Original Blurred Image (b) De-blurred Custom (c) De-blurred pcg

Followed by the following plots:



(a) (b)

Figure 4: (a) Original Blurred Image (b) De-blurred Custom (c) De-blurred pcg

When considering the visual outcome for image deblurring, both methods exhibit effective performance, yielding nearly indistinguishable results with minimal discernible differences. However, a more profound distinction emerges when analyzing their convergence behaviors.

The myCG method displays a monotonically decreasing trend during its convergence, albeit not strictly adhering to a linear decline. Eventually, it successfully reaches convergence, demonstrating a characteristic stabilization in its iterative process, aligning with the anticipated behavior based on its implementation.

Conversely, the pcg method follows a strictly decreasing trajectory as it converges. Although both methods appear to follow a similar inverse logarithmic trend, the pcg method's adherence to a strictly decreasing pattern contrasts with the myCG method's occasional spikes and subsequent stabilization. This divergence in convergence behavior sheds light on the nuanced differences between the two methods, each exhibiting unique characteristics in their iterative paths.

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

   When confronted with the challenge of an ill-conditioned system, PCG effectively mitigates the adverse effects of such conditions, resulting in a more rapid convergence process. Moreover, its inherent compatibility with parallel computing architectures positions it as a preferable choice in contemporary computational environments where parallelization offers significant performance gains over traditional methods.

   PCG demonstrates its greatest utility when tackling ill-conditioned linear systems, providing a notable enhancement in convergence rates compared to standard methods like the Conjugate Gradient (CG) approach, such as the myCG method. Its design is optimized for parallel computing architectures, leveraging the speed advantages offered by modern parallelization techniques.

   In scenarios where the same blur operator is consistently applied, PCG emerges as a highly efficient method. Despite the initial computational expense associated with setting up the preconditioner—an operation performed only once and then reused—the subsequent acceleration in convergence for multiple image deblurring tasks justifies this cost. The preconditioner, strategically applied to the system matrix, plays a pivotal role in enhancing its conditioning, thereby facilitating the swift convergence characteristic of the PCG method.