
**Numerical Computing** **2024**

**Student:** Lorenzo Galli **Discussed with:**

---

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

---

## 1. General Questions [10 points]

In image deblurring, the goal is to recover the original image from a blurred version. In our case, we will assume that the image is noise-free and we might think that each matrix entry corresponds to one pixel value of the grayscale image matrix. This process is modeled using a transformation matrix $A$ that describes how the original image $X$ was blurred to create the blurred image $B$.

Let $X$ be the original image, represented as a matrix of pixel values. For computational purposes, this image is often vectorized into a 1D column vector $x$.

Let $K$ be the blurring kernel, a matrix that defines how each pixel in the image is influenced by its neighbors during the blurring process, to make it smooth. Usually, the pixels really close to the center pixel have the largest weight, but it depends on what type of "blurring" we are looking for.

Let $A$ be the transformation matrix, that encodes the blurring effect by applying the kernel $K$ to every pixel in the image. Each row in $A$ corresponds to how a pixel is affected by its neighbors, based on the kernel. Since the kernel only influences neighboring pixels, $A$ is sparse (mostly zeros), resulting in a banded structure, with non-zero elements close to the diagonal.

The blurring effect applied to the original image is modeled as a linear transformation and this transformation is represented by, as we said, the transformation matrix $A$, which, when multiplied by the original image vector $x$, produces the blurred image vector $b$ (the vectorized form of the blurred image $B$). In other words, the blurred image vector $b$ is obtained by:

$$A \cdot x = b$$

To deblur the image, we need to solve this system in function of $x$. Since $A$ is typically large and sparse, we use methods like Conjugate Gradient to solve this equation. Once that the original image vector x is recovered, it can be restored back into a 2D matrix to visualize the deblurred image.

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

Let the original image $X$ be of size $n \times m$, where $n$ is the number of rows and $m$ is the number of columns of the image (in the matrix $X$ each matrix entry corresponds to one pixel value and $n \times m$ is the number of pixels). After vectorizing this image matrix, we get a column vector $x$ of size $N = n \cdot m$, where each entry in $x$ corresponds to a pixel in the image.

We have seen that the kernel $K$ defines how the neighboring pixels affect the current pixel during the blurring process. The transformation matrix $A$ describes how the blurring operation acts on each pixel of the image, allowing us to convert the original image $x$ into the blurred image $b$. In particular, each row of $A$ corresponds to how a single pixel in the original image is influenced by the surrounding pixels (according to the kernel).

Since each pixel of the image is influenced by its nearest pixels (with a weight that depends on the blurring kernel), the transformation matrix $A$ is a square matrix of size $N \times N$, where $N = n \times m$. In fact, each row and each column of $A$ represents a single pixel of the image, in its vectorized form.

## 2. How many diagonal bands does $A$ have?

Since, as we have seen before, the kernel $K$ defines how each pixel in the original image is influenced by its neighbors during the blurring process, I would assume that the dimension of the matrix $K$ is the key factor in understanding the number of diagonal bands of $A$.

In particular, in the matrix $A$, the main diagonal corresponds to the center pixel in the kernel, representing the pixel's value in the original image. The off-diagonal values represent the influence of the neighboring pixels on the center pixel (or vice-versa), according to the kernel. The further the off-diagonal values are from the main diagonal, the less influence the corresponding neighboring pixels have on the center pixel in the image.

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

We said that the original image $X$ have a size of $n \times m$, where each entry in the matrix $X$ corresponds to a pixel in the image, and the total number of pixels is $n \cdot m$. The column vector $x$ has the same size.

Since the blurring process does not change the number of pixels but only their values, the blurred image $B$ will have the same dimensions as the original image $X$, with $n$ rows and $m$ columns. Obviously, the vectorized version of the blurred image $b$ will also be a column vector of length $n \cdot m$.

## 2. Properties of A [10 points]

The Conjugate Gradient (CG) method is an iterative algorithm for solving systems of linear equations of the form:

$$A \cdot x = b$$

where:

$A$ is a symmetric, positive-definite matrix.
$b$ is a known vector, in our case the blurred image vectorized.
$x$ is the vector to be solved for, in our case the original image vectorized.

Differently from direct methods (like Gaussian elimination), the Conjugate Gradient method uses an iterative approach. This makes it very useful for large matrices, especially when the transformation matrix $A$ is very sparse.

However, we don't always deal with symmetric and positive-definite matrices. In particular, in our case, we are assuming that the transformation matrix $A$ is symmetric and of full rank but not positive-definite.

Let's remember that a square matrix like the transformation matrix $A \in R^{n \times n}$ is positive definite if, for every non-zero vector $\mathbf{x} \in R^n$, it's true that

$$\mathbf{x}^T A \mathbf{x} > 0.$$

In particular, a positive definite matrix is always symmetric, which means $A = A^T$, but the opposite is not guaranteed. Furthermore, in a positive definite matrix all eigenvalues of $A$ are strictly positive ($\lambda_i > 0$) and the determinants of all sub-matrices obtained by eliminating successive rows and columns are positive. I will make a short example before answering the questions because it leads me to a better understanding.
Let's verify if the following matrix is positive definite (this is just a matrix with simple numbers but it could be our big transformation matrix $A$ multiplied by the original image vectorized $x$):

$$A = \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}$$

by using a generic vector $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$.
We want to compute:

$$\mathbf{x}^T A \mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

So first of all let's compute $A\mathbf{x}$:

$$A\mathbf{x} = \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2x_1 - x_2 \\ -x_1 + 2x_2 \end{bmatrix}$$

Then, we compute $\mathbf{x}^T A\mathbf{x}$:

$$\mathbf{x}^T A \mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} 2x_1 - x_2 \\ -x_1 + 2x_2 \end{bmatrix}$$

$$\mathbf{x}^T A \mathbf{x} = x_1(2x_1 - x_2) + x_2(-x_1 + 2x_2) = 2x_1^2 - x_1x_2 - x_1x_2 + 2x_2^2$$

$$\mathbf{x}^T A \mathbf{x} = 2x_1^2 - 2x_1x_2 + 2x_2^2$$

$$\mathbf{x}^T A \mathbf{x} = (x_1 - x_2)^2 + x_1^2 + x_2^2.$$

Since $(x_1 - x_2)^2$, $x_1^2$, and $x_2^2$ are all non-negative and their sum is strictly positive for any nonzero vector $\mathbf{x}$, we conclude that $A$ is positive-definite. Of course, our matrices will be a way bigger.

In the Conjugate Gradient method the problem of solving the linear system $A\mathbf{x} = \mathbf{b}$ can be interpreted as minimizing the quadratic function:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T\mathbf{x},$$

and the solution to the equation $A\mathbf{x} = \mathbf{b}$ corresponds to the point where $f(\mathbf{x})$ is minimized. This is because the gradient of $f(\mathbf{x})$ with respect to $\mathbf{x}$ is:

$$\nabla f(\mathbf{x}) = A\mathbf{x} - \mathbf{b}.$$

Setting the gradient to zero ($\nabla f(\mathbf{x}) = 0$) gives the solution $\mathbf{x}$ that satisfies $A\mathbf{x} = \mathbf{b}$.

In practice, the Conjugate Gradient iteratively finds the solution by searching along a series of directions that are $A$-conjugate to one another. Two directions $\mathbf{p}_i$ and $\mathbf{p}_j$ are $A$-conjugate if:

$$\mathbf{p}_i^T A\mathbf{p}_j = 0, \quad \text{for } i \neq j.$$

Having understood that, now we might be wondering how to use the Conjugate Gradient method if we are dealing with symmetric and full rank but not positive-definite transformation matrix $A$.

In fact, when $A$ is not positive-definite, the quadratic form doesn't describe a convex function anymore and the Conjugate Gradient method cannot guarantee convergence.

To solve this problem, we need to pre-multiply the system with $A^T$:

$$A^T A\mathbf{x} = A^T\mathbf{b}.$$

Where:

$\tilde{A} = A^T A$ is the augmented matrix, which is symmetric and positive-definite ($A$ is full rank),
$\tilde{\mathbf{b}} = A^T\mathbf{b}$.

This operation transforms the original system to the following one:

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

Since $\tilde{A} = A^T A$ is the product of $A^T$ and $A$ (two symmetric matrices), it is symmetric by construction. Furthermore, we can verify that it's positive-definite because for any non-zero vector $\mathbf{x}$, we have:

$$\mathbf{x}^T\tilde{A}\mathbf{x} = \mathbf{x}^T A^T A\mathbf{x} = (A\mathbf{x})^T(A\mathbf{x}) = \|A\mathbf{x}\|^2 > 0,$$

Since now $\tilde{A}$ is symmetric and positive-definite, and the Conjugate Gradient method can be applied to the new augmented transformation matrix $\tilde{A}$.

After this quick part of theory, now let's answer the questions.

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

If we assume that the transformation matrix $A$ is not symmetric, the result is that the matrix $\tilde{A} = A^T A$ will still be symmetric. This happens because the product $A^T A$ is always symmetric, regardless of whether $A$ is symmetric or not.

Remembering that a matrix is symmetric if it's equal to its transposed matrix, we can show that:

$$\tilde{A}^T = (A^T A)^T = A^T (A^T)^T = A^T A = \tilde{A}$$

So, $\tilde{A}$ will still remain symmetric even if $A$ is not symmetric.

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

As we have seen earlier, in the Conjugate Gradient method, the problem of solving the linear system $A\mathbf{x} = \mathbf{b}$ is equivalent to minimizing the quadratic function:

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

We now want to deeply understand why this is true. Since we want to minimize that quadratic function, we need to find the critical points of $f(\mathbf{x})$. In order to do that, we compute the gradient of $f(\mathbf{x})$ with respect to $\mathbf{x}$.

The gradient of the first term $\frac{1}{2}\mathbf{x}^T A\mathbf{x}$ is:

$$\nabla\left(\frac{1}{2}\mathbf{x}^T A\mathbf{x}\right) = A\mathbf{x}$$

because the derivative of $\mathbf{x}^T A\mathbf{x}$ with respect to $\mathbf{x}$ is $2A\mathbf{x}$ (because $A$ is symmetric), and the factor of $\frac{1}{2}$ cancels out the 2.

Then, the gradient of the second term $-\mathbf{b}^T\mathbf{x}$ with respect to $\mathbf{x}$ is:

$$\nabla(-\mathbf{b}^T\mathbf{x}) = -\mathbf{b}$$

Putting the results together, the gradient of $f(\mathbf{x})$ is:

$$\nabla f(\mathbf{x}) = A\mathbf{x} - \mathbf{b}$$

To minimize the function $f(\mathbf{x})$, we set the gradient equal to zero:

$$A\mathbf{x} - \mathbf{b} = 0$$

which simplifies to:

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

and this is, in fact, the linear system we want to solve.

Since $A$ is symmetric and positive-definite, the function $f(\mathbf{x})$ is convex (a quadratic function with a positive-definite matrix $A$ is convex). Indeed, the function being convex guarantees that the critical point where the gradient is zero is a global minimum.

In conclusion, minimizing the function $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T\mathbf{x}$ is equivalent to solving the linear system $A\mathbf{x} = \mathbf{b}$, and the solution corresponds to the point where the gradient of $f(\mathbf{x})$ is zero.

## 3. Conjugate Gradient [30 points]

As we said earlier, the Conjugate Gradient method is very useful to solve systems in the form

$$A \cdot x = b$$

using an iterative approach. This way is computationally better than solving it by direct methods like, for example, Gaussian elimination or Matrix inversion.

Initially, we want to initialize the initial guess ($x0 = 0$), the residual r (which measures how far the current solution $x\_k$ is from satisfying the equation), and the search direction $d$.

At the beginning the search direction is set to the initial residual (which is the gradient of the function and it tells us the steepest direction to move in to keep minimizing the function), and then it's updated to be conjugate to all previous search directions.

In the Conjugate Gradient method, we don't want to just follow the steepest direction repeatedly, but we want to find independent directions that together help us to find the the solution. In fact, each direction contributes independently to the optimization process, which speeds up convergence.

1. **Write a function for the conjugate gradient solver $[x, rvec] = myCG(A, b, x0, max\_itr, tol)$, where $x$ and $rvec$ are, respectively, the solution value and a vector containing the residual at every iteration.**

Below, you can see the algorithm that follows this approach:

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

    for itr = 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);
        beta = p_new / p_old;
        d = r + beta * d;
        p_old = p_new;
        rvec = [rvec, p_new];

        if sqrt(p_new) <= tol
            disp('Converged');
            break;
        end
    end
end
```

## 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).

As a quick reminder, the goal of the algorithm is to find a solution vector $x$ such that the residual value $r$, which measures how close the current solution is to the solution, becomes sufficiently small (in practice we want the residual values to approach *tol* rather than zero).

In order to use my Conjugate Gradient algorithm with those two test sets, I need to create a little script to set the initial parameters, call the function and plot the graph:

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

[m, n] = size(A_test);
x0 = zeros(n, 1);
max_itr = 200;
tol = 1e-4;

[x, rvec] = myCG(A_test, b_test, x0, max_itr, tol);

...plot the graph...
```

Now, let's see the graph that shows us the residual values against the number of iterations:

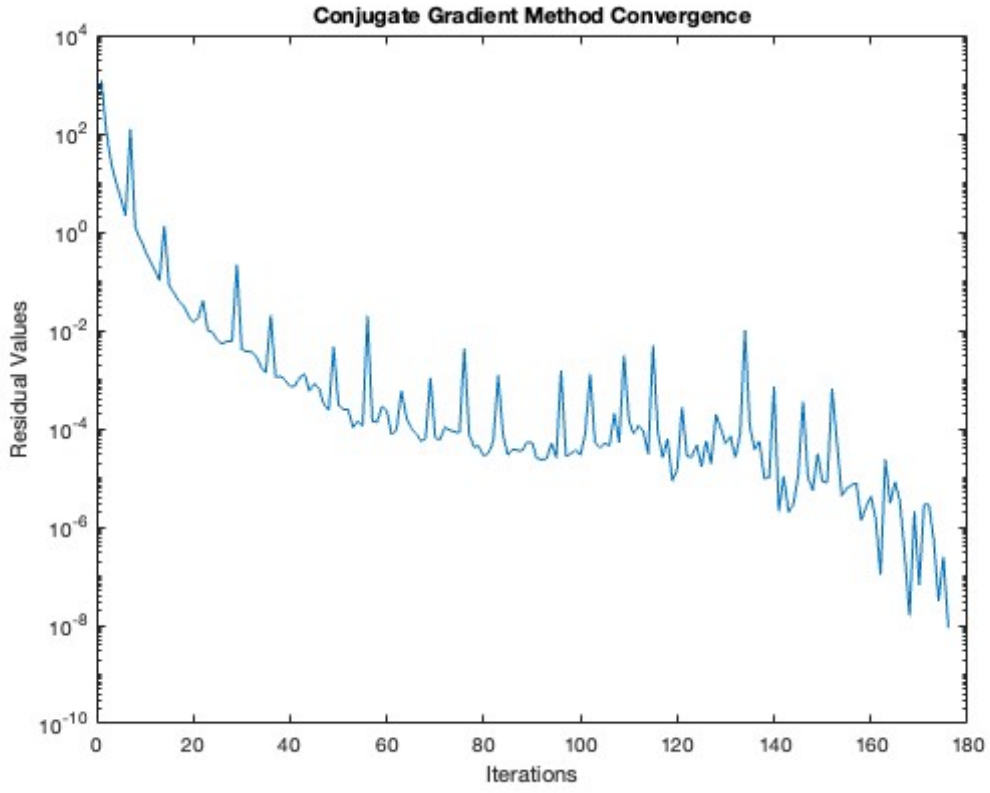



Figure 1: Plot of the residual values against iteration using Conjugate Gradient method algorithm

The implementation of Conjugate Gradient seems to converge very quickly, since it takes less than 200 iterations, which was our maximum.

### 3. Plot the eigenvalues of A test.mat and comment on the condition number and convergence rate.

The script to plot the eigenvalues of A test.mat is the following:

```
eigenvalues = eig(A_test);

minEig = min(abs(eigenvalues));
maxEig = max(abs(eigenvalues));
conditionNum = maxEig / minEig;

...plot the graph...
```

And the plotted result is:

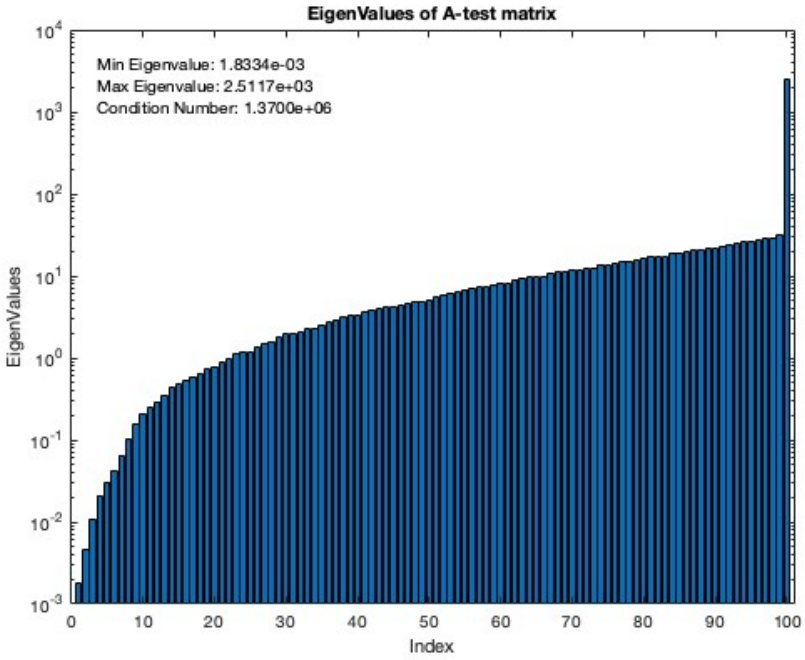



Figure 2: plot of EigenValues of matrix A test.mat

As we can see from the plot and from the data, the difference between the smallest and the largest absolute eigenvalues of the matrix is significant. This leads to an high condition number, which is calculated as follows:

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

where $\sigma_{\max}$ and $\sigma_{\min}$ represent the largest and smallest eigenvalues of the matrix $A$, respectively.

The condition number has a direct impact on the convergence rate of the Conjugate Gradient method, in fact, the convergence rate can be calculated as follows:

$$\text{convergence rate} = \sqrt{\kappa(A)}$$

If the condition number is high, the algorithm will converge slower. Furthermore, when the condition number is very large, the matrix $A$ is considered *ill-conditioned*. On the other hand, a small condition number indicates that the matrix is *well-conditioned*, and it will converge faster.

If a matrix is ill-conditioned, we can use a technique called *preconditioning*, in order to improve the condition number and make the matrix well-conditioned, with a faster convergence rate.

Preconditioning is a technique used to improve the convergence of Conjugate Method when solving linear systems, especially when the system matrix is ill-conditioned. Preconditioning technique basically makes the linear system easier to solve, by modifying the original matrix such that its condition number is reduced.

This is typically done by multiplying both sides of the equation by a preconditioner matrix $P^{-1}$, which approximates the inverse of $A$, as follows:

$$P^{-1}Ax = P^{-1}b$$

To find an efficient preconditioner, we usually use the Cholesky factorization, which decomposes a matrix into the product of a lower triangular matrix $L$ and its transpose:

$$A = LL^T$$

For preconditioning, we want to approximate $A^{-1}$ as close as possible, so we use the Cholesky factorization of $A$ to define the preconditioner

$$P = LL^T$$

In practice, Cholesky factorization can be computationally expensive, especially for large matrices. That's why we might prefer to use an incomplete Cholesky factorization (IC), which computes the Cholesky factorization only for the non-zero elements of the matrix.

This allows us to construct a sparse preconditioner that is less expensive to compute. However, the incomplete factorization could result in a matrix that is not positive-definite. To avoid this problem, we can apply a diagonal shift to the matrix to ensure that the preconditioner remains positive-definite (adds a scalar multiple of the identity matrix I to a such that: $\mathrm{A}_{\mathrm{shifted}} = A + \mu I$).

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

From figure 1 we can see that the residual value tends to decrease over time, even if we can see some fluctuations. Since the Conjugate Gradient method generates search directions that are supposed to be orthogonal, it happens that they might not always be perfectly orthogonal, especially when the matrix is ill-conditioned. In fact, if the matrix has a large difference between the smallest and the largest absolute eigenvalues, the convergence might occur slower and this can cause small oscillations in the residual values during the iterative process.

However, even if the decrease is not strictly monotonically, there is an evident overall decreasing trend of the the residual, which indicates that the Conjugate Method algorithm is approaching convergence.

## 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 $= 0.01$ for the diagonal shift (see Matlab documentation). Solve the system with both solvers using max iter $= 200$, 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.
Hint : Check the Matlab description of the pcg() routine to make sure that you compute the residual norm in the same way to be able to properly compare them.

Earlier, I described how the preconditioning technique works, and now we want to use it in order to solve the system with both myCG and in-built pcg algorithms. Here, the script for the deblurring problem:

```
load('blur_data/B.mat', 'B');
img = B;
n = size(img, 1);
b = B(:);

load('blur_data/A.mat', 'A');
initGuess = ones(size(b));
max_itr = 200;
tol = 1e-6;
alpha = 0.01;

options.type = 'nofill';
options.diagcomp = alpha;

tilde_A = transpose(A) * A;
tilde_B = transpose(A) * b;
L = ichol(tilde_A, options);
P = transpose(L) * L;

[x_myCG, rvec_myCG] = myCG(A, b, initGuess, max_itr, tol);
[x_pcg, ~, ~, iter, rvec_pcg] = pcg(tilde_A, tilde_B, tol, max_itr);

...plot the graphs...
```

Running the script, we obtain the following interesting results.

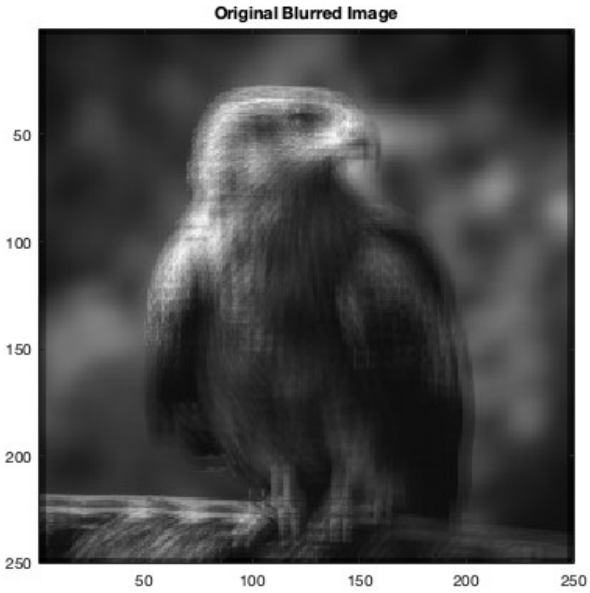
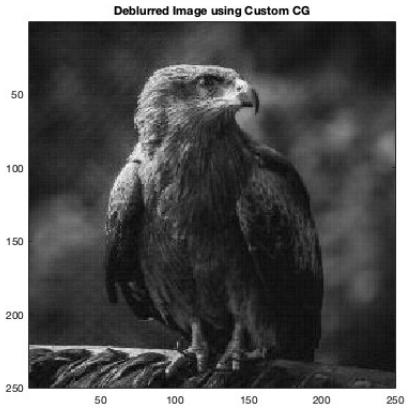



Figure 3: Blurred Image



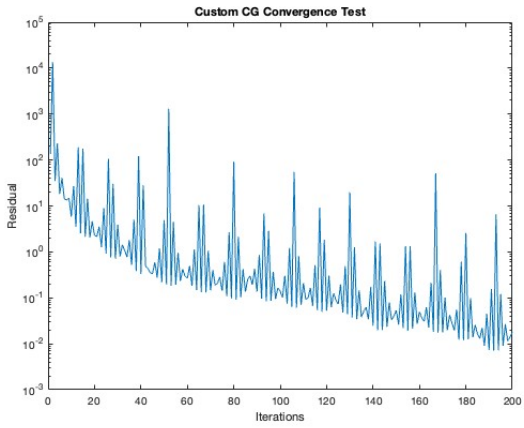
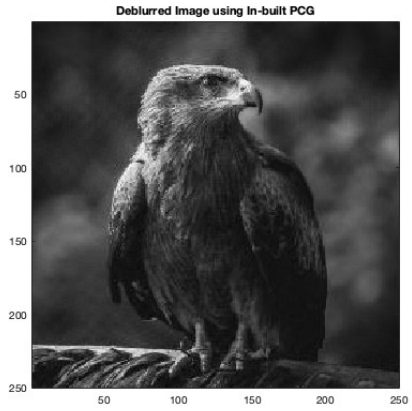




Figure 4: myCG algorithm



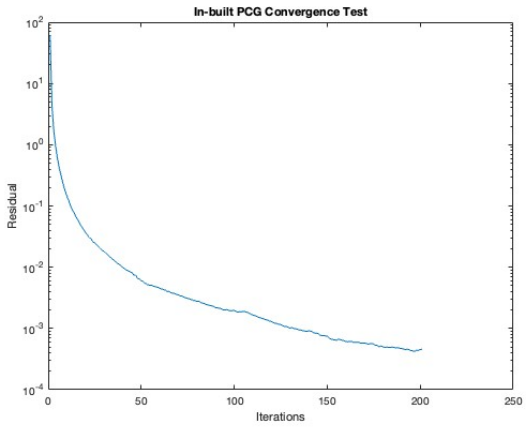




Figure 5: In-Built algorithm

Both deblurring algorithms are very effective, producing two very similar images. Maybe the in-built pcg yields a very little bit of a better image, but we are talking about minimal details. However, when examining their convergence behaviors, we can notice some more important differences.

The myCG method shows a gradual, non-linear convergence. In fact it's characterized by many fluctuation before it starts to stabilize. Besides that, in the overall, it gets closer to the expected pattern.

On the other hand, the pcg method follows a stricter, consistently decreasing convergence path. While both methods seem to reach convergence after a few hundreds of iterations, the pcg method stands out because its convergence continuously decreases without any major fluctuations, unlike myCG, which experiences a lot of variations before stabilizing.

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

The Conjugate Gradient method typically converges in fewer iterations if the matrix $A$ is well-conditioned. However, we often have to deal with ill-conditioned matrices due to the type of blurring applied. In this case, we might prefer to use the Preconditioned Conjugate Gradient for our purposes.

Since the preconditioner improves the conditioning of the system, the PCG method typically converges faster than CG, especially when the condition number of A is high. In practice, this leads to a reduction in the number of iterations required for convergence. Having said that, for ill-conditioned problems (like severe blur), PCG can be significantly more efficient than CG.

Like CG, PCG also requires matrix-vector products and inner products. However, it adds the cost of applying the preconditioner M during each iteration. This is the negative side of PCG, that makes it computationally more expensive.

In conclusion, PCG is more efficient for ill-conditioned problems, despite potentially higher computational costs due to the preconditioner.