Università della Svizzera italiana | Faculty of Informatics | Institute of Computing CI

Numerical Computing **2022**

Student: Albert Cerfeda

Solution for Project 5

Due date: Wednesday, 7 December 2022, 11:59 PM

Numerical Computing 2022 — 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, Julia). 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.

The purpose of this assignment is to gain insight on the theoretical and numerical properties of the **Conjugate Gradient method**. Here we use this method in an image processing application with the goal of **deblurring** an image given the exact (*noise-free*) blurred image and the original transformation matrix. Note that the "noise-free" simplification is essential for us to solve this problem in the scope of this assignment.

Contents

1. General Questions [10 points]

1.1. Size of matrix A

A is the sparse symmetric transformation matrix computed by repeateadly applying the *image kernel matrix*[1].
$B \in \mathbb{R}^{n \times n}$ in our case is the transformed,blurred image of size n by n pixels obtained by transforming the original deblurred image X by the transformation matrix A.
The blurring computation can therefore be defined with the following linear equation:

$$Ax = b$$

where $x, b \in \mathbb{R}^{n^2}$ are the vectorized representations of original image X and blurred image B respectively, given that A is a square matrix of size $n^2 \times n^2$ where n is the size of the image along any dimension.
The Julia code for retrieving the size of matrix A is rather trivial:

```
# Load default image data
B = read(matopen("./Data/Blur/B.mat"), "B");
img = B;
n = size(img, 1)
sizeA = n*n
```

Figure 1: Julia code for computing the size of transformation matrix A given blurred image B

Matrix A is a 62500×62500 matrix.

1.2. Diagonal bands of matrix A

We know that A is a d^2-*banded*[2] **symmetric** square matrix where d is the size of the kernel image matrix $K \in \mathbb{R}^{d \times d}$. This makes it easy to compute the amount of bands:
Since we know that our kernel image matrix is of size 7×7, our transformation matrix A has d^2 bands i.e 49 bands.

1.3. Lenght of vectorized blurred image b

We require the blurred image $B \in \mathbb{R}^{n \times n}$ to be converted into a *vectorized* column vector $b \in \mathbb{R}^{n^2}$ for transforming it with transformation matrix A.
We know that $n = 256$, therefore $n^2 = 62500$.

[1] Image Kernel Matrix: It is a small matrix used to transform and apply effects (eg "*filters*") to images. It can be therefore used for applying sharpening,desharpening,embossing ecc. It calculates the new value of a pixel through a weighted sum of the pixel and its neighbors.)

[2] Banded Matrix: A band matrix is a sparse matrix whose non-zero entries are stored on the main diagonal and zero or more diagonals on each side.

2. Properties of A [10 points]

2.1 Effects on $\tilde{A}$ when A is not symmetric

Since A is not positive-definite, we require the computation of $\tilde{A}$ for using the *Conjugate Gradient* method when solving the before-mentioned linear system [Eq. 1].

The *augmented system* is defined as follows:

$$\underbrace{A^T A}_{\tilde{A}} x = \underbrace{A^T b}_{\tilde{b}}$$

Figure 2: The *augmented system*

We compute the augmented system when the coefficient matrix A is not positive definite, like in our case. This operation affects the *condition number* $\kappa(A)$ [See Section 3.3] as $\kappa(\tilde{A}) \approx \kappa(A)^2$.

Was matrix A not to be symmetric, its augmented counterpart would still be symmetric as any square matrix B multiplied by its transposed B^T i.e $B^T B$ is symmetric.

2.2 Proof for the minimization problem

We need to show that the solving $Ax = b$ is equal to minimizing $\frac{1}{2}A^T x + \frac{1}{2}Ax - b$ over x.
Let us compute the derivative $f'(x) = \frac{d}{dx}\frac{1}{2}A^T x + \frac{1}{2}Ax - b$.

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

Note: Since matrix A is symmetric, $A^T = A$.
We can see how computing the derivative and then substituting A^T with A leads to equation $Ax = b$, therefore proving the above statement.

3. Conjugate Gradient [30 points]

3.1 Conjugate Gradient solver implementation

We want to find the original,deblurred image X by solving the linear system that represents the blurring operation [See Eq. 1] by the vectorized image x, given that transformation matrix A and vectorized blurred image b are known.

The methods for computing a linear system fall under two main categories: *Direct methods* and *Iterative methods*.
Direct methods find the exact solution (e.g Gaussian Elimination). Gaussian Elimination becomes unfeasably expensive for very large matrices like in our case. Iterative methods on the other hand start with a random solution converging more towards the exact result with each iteration, deriving an approximated result.

The *Conjugate Gradient solver* is an iterative method for finding an *approximated* solution to a very large linear system of equations like in our case. It's more specific than the Gaussian method as it requires matrix A to be a symmetric, positive-definite matrix.

Figure 3 shows the implementation of `function myCG(A,b,x0,max_itr,abstol)`:

Note: Input parameter `tol` was renamed to `abstol` to avoid confusion.
Function `IterativeSolvers.cg(A, b; kwargs...)`[3] features named parameters `abstol` and `reltol` for specifying either the relative tolerance ratio between iterations or the absolute tolerance value. Since in our implementation we are effectively using the absolute tolerance for probing convergence, the parameter was renamed to ensure consistency.

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

        push!(rvec,p_new);
        if sqrt(p_new) <= abstol
            println("Converged")
            break
        end
    end
    return x, rvec
end
```

Figure 3: Julia code defining the implementation of the *Conjugate Gradient solver*

3.2 Solving the system defined by A_test.mat and b_test.mat

Let us solve the provided sample linear system:

[3] IterativeSolvers.cg(...): See documentation iterativesolvers.julialinearalgebra.org

```
# Load test data
b_test = read(matopen("./Data/Test/b_test.mat"), "b_test");
img = b_test

# Blurred image
heatmap(reshape(img, (10, 10)), c=:greys, yflip=true,
    legend=:none, axis=nothing, aspect_ratio=:equal, xlabel="Blurred")

# Load transformation matrix A
A_test = read(matopen("./Data/Test/A_test.mat"), "A_test");

## Run myCG() on B_test
guess = ones(size(A_test,1));
maxiter = 200;
abstol = 1e-4;

x,residuals = myCG(A_test,b_test,guess,maxiter,abstol)
# Deblurred image
heatmap(reshape(x, (10, 10)), c=:greys, yflip=true,
    legend=:none, axis=nothing, aspect_ratio=:equal, xlabel="Deblurred")

## Plot convergence
plot(residuals,  ylims=(-Inf,1e5), xlabel="Iterations",ylabel="Residual value",
    yscale=:log10)
```

Figure 4: Julia code solving the sample linear system by using function *myCG* [Fig. 3]

Let us plot the blurred and the obtained deblurred counterpart:

Figure 5: The *B_test* image. On the left, the blurred one. On the right, the deblurred image obtained though function *myCG* [Fig. 3]

Its difficult to verify the correctness solely from the visual comparison as the images are too low-resolution for us to intuitively detect correct deblurring. However, convergence is achieved as shown by the plot representing the residual value variation [See Fig.6]:

It can be observed how the *Conjugate Gradient solver* method quickly finds convergence in few iterations.

3.3 Condition number and convergence rate in relation to the Eigenvalues of A_test.mat

The condition number $\kappa(A)$ is a property on linear systems of equations. It measures how much the result of the equation changes in 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 condition number directly affects the convergence rate of the *Conjugate Gradient* method, as

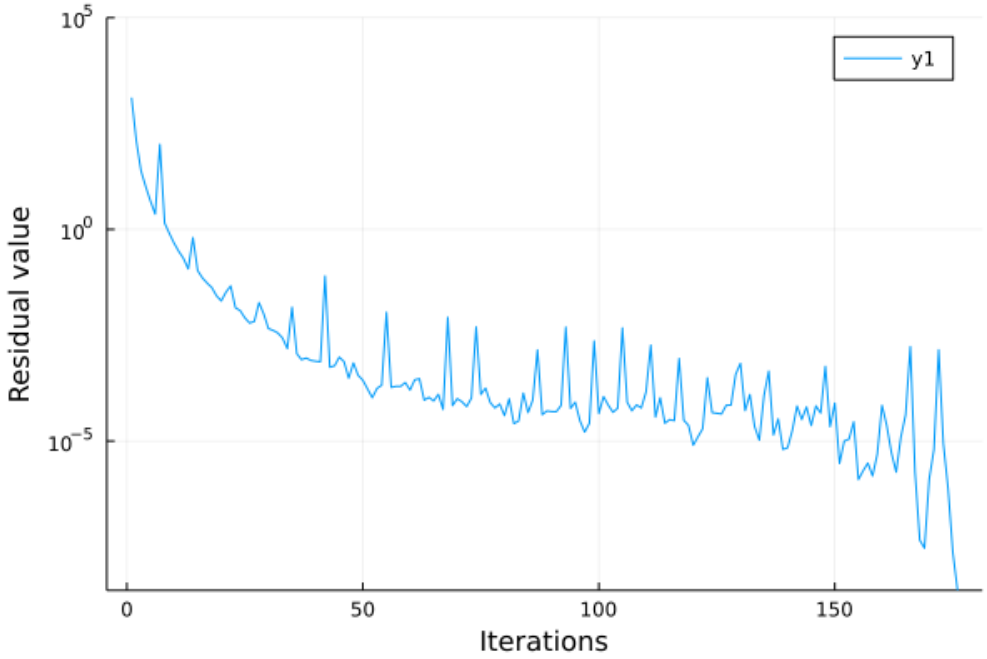

Figure 6: Plot representing the residual value variation for each iteration

running it on an ill-contitioned system results in significantly lower convergence rate.
The *Condition Number* of a square matrix is defined as follows:

$$\kappa(A) = \frac{\sigma_{\max}}{\sigma_{\min}} \qquad \text{where } \sigma \text{ are the singular values of matrix } A$$

The singular values of a symmetric matrix are equal to the absolute values of its eigenvalues, i.e $\sigma = |\lambda|$. This property holds for our transformation matrix A as it is symmetrical.
We can intuitively infer from the previous statements that the condition number of a system comes from the maximal ratio between eigenvalues of the *coefficient matrix* of the system (e.g matrix A [See Eq.1])
Let us plot the eigenvalues of provided transformation matrix `A_test.mat`

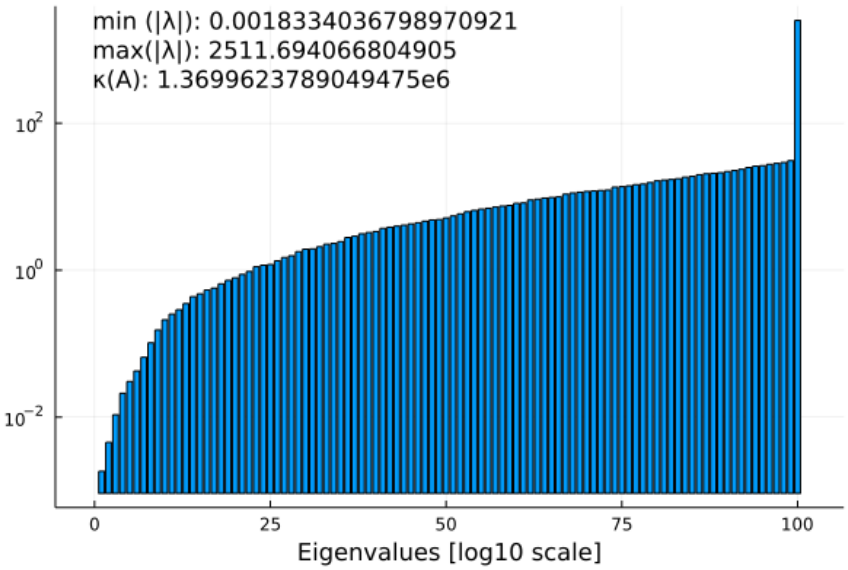

Figure 7: Bar graph representing the residual value variation for each iteration when running function *myCG* on matrix *A_test.mat* and vectorized blurred image *b_test.mat*, in *log10* scale

As we can see the distance between the smallest absolute eigenvalue $\kappa_{\min}$ and the greatest abso-

lute eigenvalue $\kappa_{\max}$ is rather large. In fact $\kappa(A) \approx 1.37 * 10^6$.
The condition number gets drastically worse if matrix `A_test.mat` was not to be positive definite, requiring us to solve for the *augmented system* $\tilde{A}x = \tilde{b}$ where $\tilde{A} = A^T A$ and $\tilde{b} = A^T b$, making the condition number grow exponentially. This case does not present itself as matrix `A_test.mat` is already positive definite.
We can use a technique known as *preconditioning* for reducing the distance of the eigenvalues and therefore lower the condition number.

3.4 Residual variation non-monotonically decreasing

As we can see from Figure 7, the residual variation is *not* decreasing monotonically.
Function `myCG` minimizes the distance of the vectorized unblurred image x from the actual vectorized unblurred image (the optimal x). This makes the residual variation generally decrease like expected leading towards convergence, but not monotonically.
Instead, if function `myCG` was to minimize the residual, the residual variation would decrease monotonically.

4. Deblurring problem [35 points]

4.1 Preconditioning using the incomplete Cholesky factorization

As mentioned before the condition number directly depends on the *range* of the eigenvalues of the coefficient matrix (eg transformation matrix A), given that it is a symmetric matrix.
An already adverse condition number $\kappa(A)$ only gets worse if we take in account some pre-manipulation done on system, like computing the *augmented system* [See Fig 2].

The *Identity Matrix* $I \in \mathbb{R}^{n \times n}$ by definition has n eigenvalues valued 1 (i.e $\lambda_{1,..n} = 1$). This ensures the lowest possible distance between eigenvalues and therefore $\kappa(A) = \frac{1}{1} = 1$. The condition number being equal to 1 implies that the algorithm will not arbitrarily diverge because of errors in the input and that the precision of the approximated result will be no different from the precision of the input data.

We can find matrix P^{-1} that approximates $\tilde{A}^{-1}$ meaning that once multiplied, the resulting matrix is closer to the Identity Matrix i.e:

$$P^{-1}\tilde{A} \approx I \quad \text{where } P \text{ is the symmetric positive-definite } \textit{preconditioner}$$

By approximating $\tilde{A}$ to the identity matrix I we ensure a smaller eigenvalue range and a lower condition number $\kappa(A)$. This technique is known as *preconditioning.*
We can decompose the preconditioner P such that $P = LL^T$, where L is the Cholesky Factor.

The Cholesky factorization decomposes a matrix A into a lower-triangular matrix L and its upper-triangular counterpart L^T. For sparse matrices, we use the *Incomplete* Cholesky factorization, a sparse approximation of the regular method. It computes the Cholesky factors of only the non-zero elements of A resulting in cheaper computational cost for sparse matrices.

Finally we can implement the Julia script for deblurring the provided `B.mat` image. The transformation matrix A has to be first augmented to ensure symmetricity and then shifted by 0.01 along its diagonal to ensure the existance of IC factors. After that, the final steps are rather trivial: we make use of the `CholeskyPreconditioner(..)` function inside package `Preconditioners.jl`[4]
As requested the *maximum iterations* is set to 200 and the *absolute tolerance* to 10^{-4}. The Julia

[4]Preconditioners.CholeskyPreconditioner(..): See codebase [github.com]

code is shown in Figure 8.

```
# Load blurred image B
B = read(matopen("./Data/Blur/B.mat"), "B");
img = B;
n = size(img, 1)
# Vectorize image
b = vec(B)
# Load transformation matrix A
A = read(matopen("./Data/Blur/A.mat"), "A");

# The initial guess
guess = ones(size(b,1));
# The maximum amount of iterations
maxiter = 200;
# The absolute tolerance
abstol = 1e-4;

# Compute augmented matrix to ensure positive-definedness and symmetricity
augA = A'A
# Shift diagonal by 0.01 to ensure the existance of IC factors
augA = 0.01*I+augA
# Retrieve the Cholesky preconditioner matrix with memory fill set to 1
P = CholeskyPreconditioner(augA,1);

# Run function IterativeSolvers.cg(...)
x, residuals  = IterativeSolvers.cg(A,b,log=true,Pl=P,maxiter=maxiter,abstol=abstol)
# Run function myCG()
x,residuals = myCG(A,b,guess,maxiter,abstol)
```

Figure 8: Julia code for deblurring the provided blurred `B.mat` image with both function `myCG()` [Fig.3] and `IterativeSolvers.cg(...)`

Let us plot the obtained deblurred images with the respective methods: [See results]

We can clearly see how the implementation is correct as the images are correctly deblurred. Both methods do not converge, as 200 iterations in this instance are too few. Nevertheless, we still obtain a sufficiently deblurred image.

Plotting the residual value variation yields the following results [See results].

Even tough both methods do not converge we notice how the residual variation is way lower in Figure 12. This is clearly due to the benefits of performing preconditioning on the augmented matrix $\tilde{A}$. As stated before4 the preconditioning step is performed to approximate $\tilde{A}$ to the identity matrix I to ensure a smaller eigenvalue range and a lower condition number $\kappa(A)$.

4.2 Amortized cost of preconditioned Conjugate Gradient method

Even though performing the preconditioning step leads to some performance overhead, the overall *amortized* cost in the long run, in the case where the same transformation matrix A is used to deblur multiple images which share a common image kernel matrix, makes it worth using preconditioning given its faster convergence rate.

5. Reproducing the obtained results

In the `src/` folder inside the submission archive you can find a `Makefile`.

Run command `make` while having the current working directory set as the `src/` folder to plot and store all the results used for this report. All the plots and blurred/unblurred images are saved inside the `src/out` folder. Make sure to uncomment the `png(...)` statements for the files you want to generate.

Figure 9: Provided blurred image `B.mat`

Figure 10: Obtained deblurred image using function `IterativeSolvers.cg(...)`

Figure 11: Obtained deblurred image using function `myCG(...)`

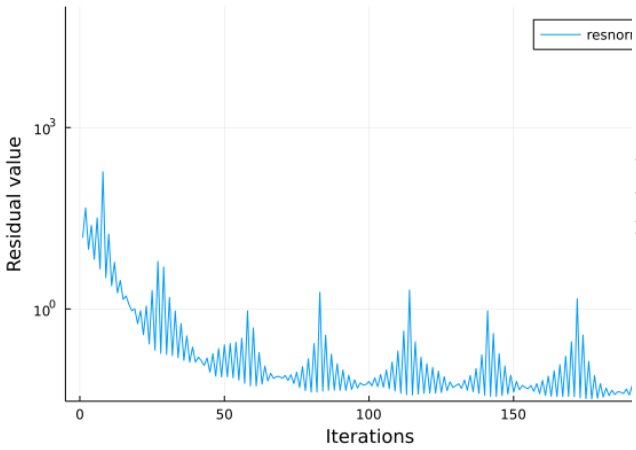

Figure 12: Obtained deblurred image using function `IterativeSolvers.cg(...)`

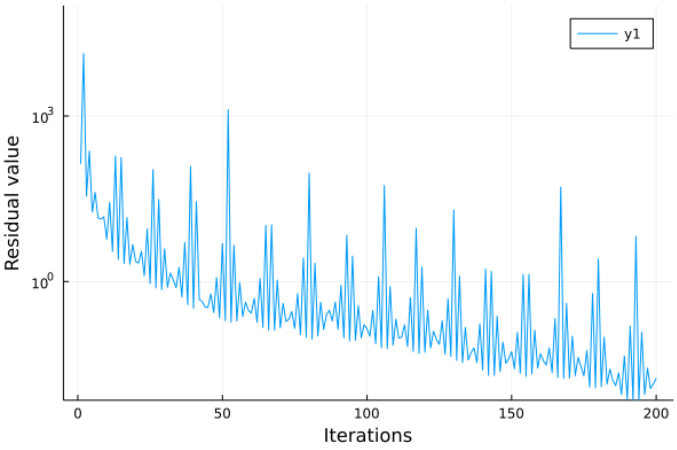

Figure 13: Obtained deblurred image using function `myCG(...)`