Numerical Methods for Differential Equations
Homework 2th

Lorenzo Schiavone

January 12, 2025

Contents

1 Exercise 1

My implementation of the PCG method with Cholesky preconditioner is the following.

```
function [x, resvec, iter] = mypcg(A, b, tol, maxit, L)
x = zeros(size(b));
r = b - A *x;
resvec = zeros(maxit);resvec(1)=norm(r);
p = L' \ (L \ r); g=p;rho = r'*g;
exit_test= tol*norm(b); k=0;
while ((resvec(k+1)>exit_test) && (k<maxit))
    z = A*p;
    alpha = rho/(z'*p);
    x = x + alpha*p;
    r = r -alpha*z;
    g= L' \ (L \ r);
    rho_new = r'*g;
    beta = rho_new/rho;
    p = g + beta*p;
    k=k+1;
    rho = rho_new;
    resvec(k+1)=norm(r);
end
resvec = resvec(1:(k+1)); iter = k;
```

To verify its correctness, Figure 1 compares the residue vector norms `resvec` provided by `mypcg` with those from MATLAB's `pcg` in a test case. We consider `A=delsq(numgrid('S',102))`, `n=size(A,1)`, `b=A*ones(n,1)`, `tol=1e-8` for both the Conjugate Gradient without preconditioner, i.e., with `L=speye(size(A,1))`, and for PCG with `L=ichol(A)`. In both cases, `mypcg`'s `resvec` match MATLAB's `pcg` ones and `mypcg` reaches convergence in the same number of iterations of MATLAB's `pcg`.

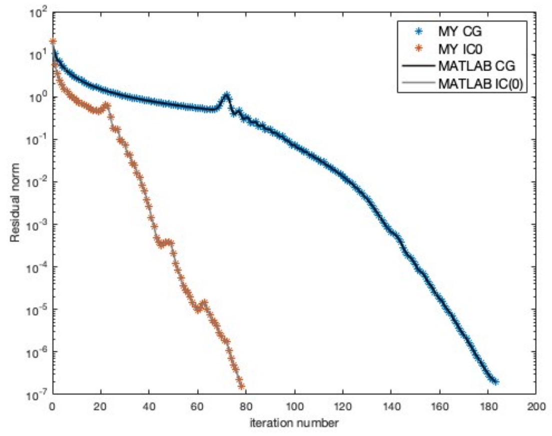


Figure 1: Comparison between `mypcg`'s and MATLAB `pcg`'s `resvec`.

2 Exercise 2

We would like to verify the dependence of the number of CG iterations on the square root of the condition number $\sqrt{\kappa(A)}$ by solving 4 linear systems `Ax = b` with `A=delsq(numgrid('S',nx))` and $nx = 102, 202, 402, 802$. The parameters `tol = 1e-8` and `maxit = 5000` are fixed and the right hand side b is set to correspond to the exact solution `x = 1./sqrt(1:size(A,1))`. Table 1 reports the number of iterations required by the method with different nx and `droprol` for the Cholesky preconditioners. We observe see that, every time the mesh discretization parameter is halved, in any column, the number of iterations doubles in agreement with the theory, since the condition number of the Poisson matrix doubles.

$nx - 1 \approx h^{-1}$	CG	IC(0)	IC(2)	IC(3)
101	283	87	45	17
201	532	159	78	30
401	948	282	137	53
801	1792	533	258	97

Table 1: Number of CG iterations with different nx. Respectively, with no preconditioner: CG, with Cholesky preconditioner with the same sparsity pattern of A: IC(0), with `droptol = 1e-2`: IC(2) and with `droptol = 1e-3`: IC(3).

3 Exercise 3

In this exercise, we show that for a diagonal matrix $A = \text{diag}(200, 400, 600, 800, 1000, 1, 1, \dots, 1)$ of size $n = 10^4$, the PCG method converges exactly in six iterations.

Indeed, at each iteration k, the error vector e_k lies in the subspace

$$e_0 + \text{span}\{Ae_0, A^2e_0, \dots, A^ke_0\} \quad \text{i.e.,} \quad e_k = e_0 + \sum_{j=1}^{k} c_j A^j e_0 = P_k(A)e_0,$$

for some coefficients c_j and polynomial of degree k denoted $P_k(x)$ with $P_k(0) = 1$. By the minimality property of the CG method, we have that

$$\|e_k\|_A = \min_{P_k\,:\,P_k(0)=1} \|P_k(A)e_0\|_A.$$

In our case, for $k = 6$, we can choose the polynomial

$$P_6(x) = (1-x)\prod_{i=1}^{5}\left(1 - \frac{x}{200\cdot i}\right)$$

and, since A is diagonalizable, there exists a basis of eigenvectors $(u_i)_{i=1,\dots,n}$, for which $e_0 = \sum_{i=1}^n \alpha_i u_i$. Thus,

$$P_6(A)e_0 = \sum_{i=1}^{n} \alpha_i P_6(A)u_i$$

but $P_6(A)u_i = 0$ for all $i = 1, \dots, n$ because the eigenvalues of A are exactly $\{1, 200, 400, 600, 800, 1000\}$, and for each eigenvector u_i the factor related to the corresponding eigenvalue λ_i vanishes:

$$P_6(A)u_i = (1-\lambda_i)\prod_{j=1}^{5}\left(1 - \frac{\lambda_i}{200j}\right)u_i = 0.$$

Hence, the PCG method converges in at most six iterations. Moreover, this argument cannot be applied for $k < 6$ because A has six distinct eigenvalues, and any polynomial of degree k can have at most k real

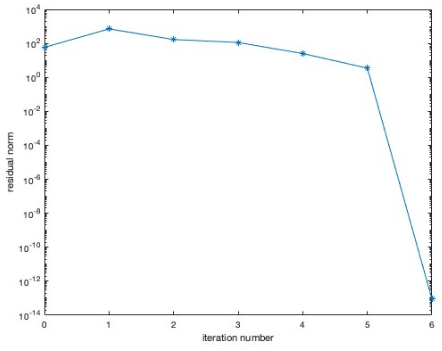


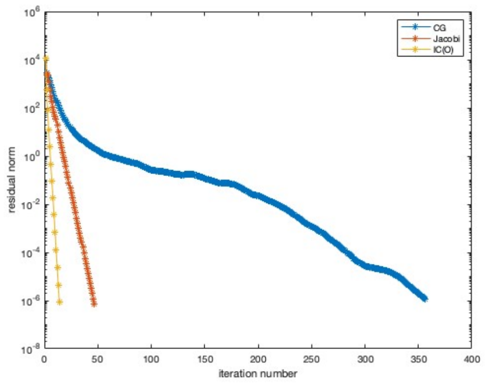
Figure 2: Semilogarithmic plot of the residual norm vs the iteration number.

roots. Therefore, for $k < 6$, it is impossible for the polynomial $P_k(x)$ to vanish at all six eigenvalues of A. Therefore, in this case, the PCG method converges in at least six iterations and thus in exactly six iterations.

Figure 2 shows `resvec` for iteration number taken by mypcg applied to the matrix A above and random right hand side `b = rand(n,1)`. In agreement with the theory, the method converges exactly in six iterations.

4 Exercise 4

In order to compare the non preconditioned CG and the PCG with Jacobi or Incomplete Cholesky preconditioners, we solve with PCG a linear system `Ax=b` with `A = gallery ('wathen',100,100)` as the coefficient matrix and `b` the right hand side corresponding to a random vector solution.



method	**CPU time (sec)**	**n° iterations**
CG	0.45	369
Jacobi	0.08	46
IC(0)	0.05	13

Table 2: Average CPU time and iteration number for 100 run with random vector solution.

Figure 3: Convergence profile of one case.

From Figure 3 and Table 2, it is evident that, even relatively simple preconditioner like Jacobi or the Incomplete Cholesky with the same sparsity pattern of the matrix A, helps to drastically reduce the CPU time and number of iterations to reach convergence.

5 Exercise 5

My personal GMRES without restart is implemented in MATLAB as follows:

```
function [x,iter,resvec,flag] = mygmres(A,b,tol,maxit,x0)
r0 = b - A*x0; beta=norm(r0);
resvec = zeros(maxit); resvec(1)=beta;
```

```
V = zeros(size(A,1), maxit); V(:,1) = r0/beta;
exit_test = tol*norm(b); flag = 0; k = 0;
H = zeros(maxit,maxit);
if (resvec(1) < exit_test) x=x0; else
while (resvec(k+1) > exit_test) && (k < maxit)
    k = k+1;
    % arnoldi method
    v_new= A*V(:,k); % v_{k+1}
    h = zeros(k,1);
    for j =1:k
        h(j)=v_new'*V(:,j); %h(j) are the component of the new column of H
        v_new = v_new - h(j)*V(:,j);
    end
    H(1:k,k)=h; % H is a square matrix here
    h_new = norm(v_new);
    if h_new < 1e-14 % happy breakdown
        disp('happy breakdown'); flag = -1;
        [Q,R]=qr(H(1:k,1:k));
        break
    else
    V(:,k+1)=v_new/h_new;
    H(k+1,k) = h_new;
    end
    [Q,R]=qr(H(1:k+1,1:k));
    resvec(k+1)=abs(beta*Q(1,k+1));
end
rhs = beta* Q(1,1:k)'; y = R(1:k,:)\(rhs);
x=x0 + V(:,1:k)*y;
end
iter = k; resvec=resvec(1:(k+1));
end
```

To test it, it is used for solving the linear system `Ax = b` where A is the matrix stored in mat13041.rig and `b` corresponds to the exact solution with components $x_i = 1/\sqrt{i}$, `tol = 1e-1; maxit = 550`, and `x0` the all zero vector. As we can see from Figure 4, the method converges in 511 iterations.

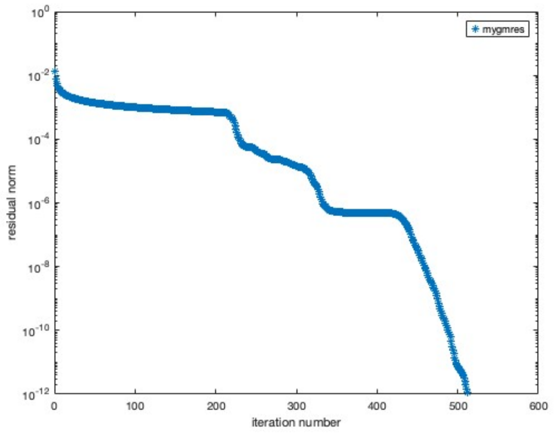


Figure 4: Semilogarithmic profile of Residue vs Iteration number for `mygmres`.

6 Exercise 6

My implementation of the preconditioned GMRES is exactly as in Exercise 5 where we have only changed these following three lines:

```
function [x,iter ,resvec ,flag] = myprecgmres(A,b,tol ,maxit,x0,L,U)
r0 = U\(L\(b - A*x0));
...
exit_test = tol*norm(U\(L\(b)));
...
v_new=U \ (L\(A*V(:,k)));
...
end
```

Method	Iterations	Final residue	"True" residue
`myprecgmres`	93	$8.921 \cdot 10^{-11}$	$5.45 \cdot 10^{-13}$
MATLAB `gmres`	93	$8.919 \cdot 10^{-11}$	$5.45 \cdot 10^{-13}$

Table 3: Comparison between `myprecgmres` and MATLAB's `gmres` for iterations number, final residue and true residue $\|Ax - \mathbf{b}\|$.

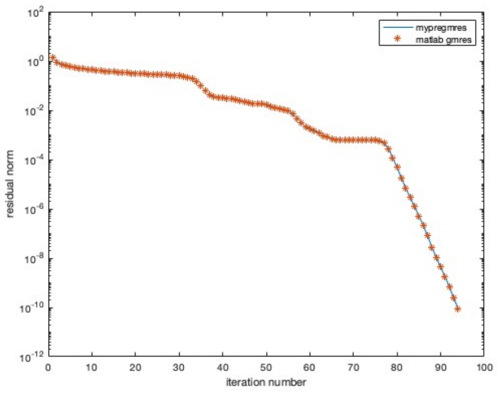


Figure 5: Convergence Plot comparison between `myprecgmres` and MATLAB's `gmres`.

Again, to test the function `myprecgmres`, it is applied for solving the same linear system as in Exercise 5 with the ILU preconditioner with drop tolerance 0.1. Table 3 compares `myprecgmres` with MATLAB's `gmres` for number of iterations, final residue, and true residue. As shown in Figure 5, the `myprecgmres` produces results very similar to those of MATLAB's `gmres` and behaves similarly.

7 Exercise 7

For the same linear system, we would like to investigate the effects of different restart values. We use the MATLAB's `gmres` with `tol = 1e-12`, `restart = 10,20,30,50` and the ILU preconditioner with `droptol = 1e-2`. For each of the four values of restart, we record in Table 4 the number of total iterations, the relative residual at convergence, and the CPU time.

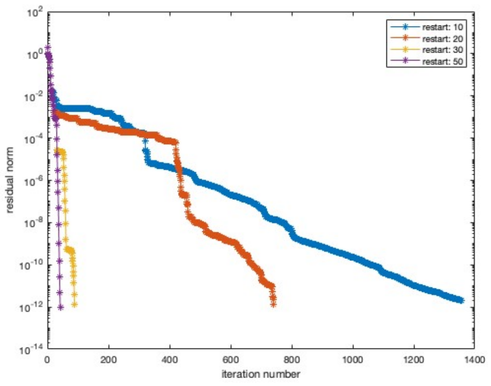


Figure 6: Convergence profiles for different restart values.

Restarts	n° iterations	Relres	CPU time (sec)
10	1367	$2.03 \cdot 10^{-12}$	1.9685
20	760	$1.33 \cdot 10^{-12}$	1.4441
30	118	$1.38 \cdot 10^{-12}$	0.18664
50	91	$9.94 \cdot 10^{-13}$	0.10118

Table 4: Summary for different restart values.

From Figure 6 and Table 4, we observe that increasing the restart value, i.e., the maximum number of basis vectors for the Krylov subspaces, results in fewer iterations to achieve convergence and CPU time, despite each iteration of the Arnoldi's method being more computationally expensive. Conversely, with a smaller restart value, each iteration is cheaper, but we need more than 1300 iterations to reach convergence and more CPU time as well.

8 Exercise 8

In this exercise we would like to investigate the effect of different drop tolerances in the ILU preconditioner. For that, we solve the linear system `Ax = b`, where `A` is the square sparse matrix with dimension $2 \cdot 10^4$ provided in `ML_laplace.mtx` and `b` is the right hand side corresponding to the exact solution vector made of all ones.

We keep `restart = 50` and change the drop tolerance for the ILU preconditioners. Table 5 displays for each of the six runs the drop tolerance, the number of iterations, the CPU time needed for computing the preconditioner (tprec), the CPU time needed for GMRES to solve the system (tsol), the total CPU (tprec+tsol), the final residual norm and the density ρ of the preconditioner, that is $\rho = \frac{\text{nnz}\,(L)+\text{nnz}\,(U)-n}{\text{nnz}\,(A)}$, where n is the order of the matrix A. Finally, Figure 7 shows the convergence profile of the six runs. We observe that decreasing the drop tolerance reduces both the number of GMRES iterations and the CPU time required to solve the linear system. However, as the ILU factorization becomes denser with a smaller drop tolerance, the CPU time required for its computation increases. As a result, it is not always convenient to arbitrarily decrease the drop tolerance. In this case, the optimal tradeoff is achieved for `droptol =` 10^{-3}.

droptol	**n° iterations**	**tprecs**	**tsols**	**cpu times**	**residual norm**	ρ
$2 \cdot 10^{-2}$	581	1.556	3.865	5.421	$3.31 \cdot 10^{-11}$	0.444
10^{-2}	258	1.578	1.739	3.318	$3.645 \cdot 10^{-11}$	0.572
$3 \cdot 10^{-3}$	129	1.496	0.620	2.116	$4.49 \cdot 10^{-11}$	0.934
10^{-3}	86	1.611	0.366	1.977	$3.86 \cdot 10^{-11}$	1.449
10^{-4}	67	2.358	0.257	2.615	$2.93 \cdot 10^{-11}$	3.533
10^{-5}	59	6.175	0.232	6.407	$2.47 \cdot 10^{-11}$	8.556

Table 5: Summary for each run with different drop tolerances.

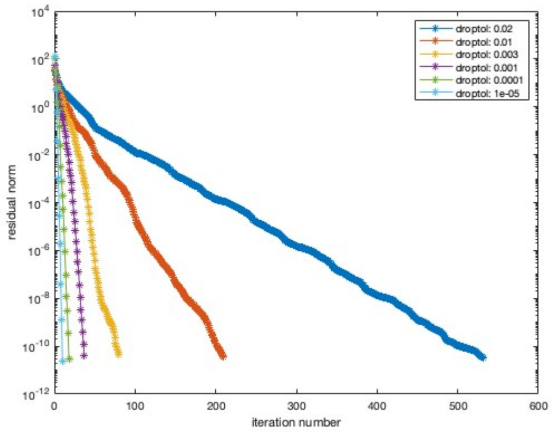


Figure 7: Convergence profile for different drop tolerances.