Numerical Algorithms - Assignment 4

Sebastiano Smaniotto 857744

February 2021

Introduction

In order to carry out the assignment we have used the programming language Python. To make sure that there are no compatibility problem, I have put the file `numalg.yaml` in the submission folder which contains the conda environment that we have used for the computation. We could not perform the fourth, optional task because we could not find a reliable way to measure the number of FLOPS when calling a function inside our script.

1 Implementation

The methods were implemented using the facilities offered by the sub-module `sparse` of the `scipy` module. We have imported the data for the sparse matrix A from MATLAB, converted the matrix into a `scipy.sparse.csr_matrix` and saved it again in the file `A.npz`. We have done the same thing for the preconditioning matrix P, obtained by applying the incomplete Cholesky factorization of A through the MATLAB function `ichol`, and stored it on file `P.npz`. As the reader can see from Figure 1 such preconditioning matrix gives rise to some kind of problems that we have not been able to determine. For such reason, we have decided to also use the Jacobi preconditioner to make further confrontations. We have used the linear solvers of packages `numpy` and `scipy` to efficiently solve linear systems $Ax = b$ when the algorithms required so, and we have taken advantage of the form of A (i.e., lower/upper triangular) whenever possible.

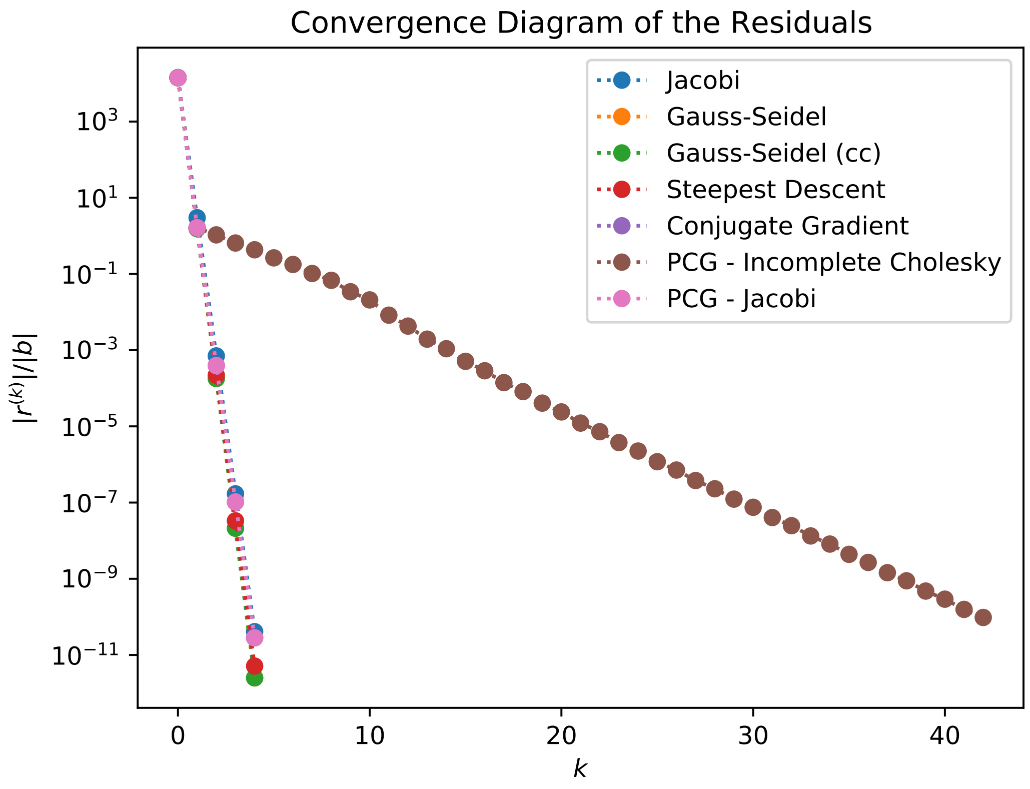

Figure 1: In this figure the convergence profile of the relative residuals is plotted. The y-axis is in log-scale.

	CPU Time Elapsed	
Method	Sparse	Dense
J	0.00e+00 ± 0.00e+00	5.12e-01 ± 4.41e-02
GS	3.94e-01 ± 3.11e-02	9.44e+00 ± 1.11e+00
GScc	3.00e+00 ± 7.05e-02	1.43e+00 ± 1.17e-01
SD	4.69e-03 ± 1.41e-02	2.06e-01 ± 5.63e-02
CG	0.00e+00 ± 0.00e+00	2.34e-01 ± 4.69e-02
PCGc	1.03e+00 ± 7.09e-02	3.44e+00 ± 2.08e-01
PCGj	2.34e-02 ± 2.34e-02	2.25e-01 ± 4.59e-02
SS	4.22e-02 ± 1.41e-02	2.59e+00 ± 2.29e-01

Table 1: Average elapsed time for each method.

2 Convergence of Relative Residuals

In this section we address the first task of this assignment and comment on the results of our experiments. In Figure 1 it is possible to see how the different methods converges as the number of iterations k increases. The y-axis is in log-scale, otherwise it would be impossible to see how the relative residuals decreases as after just one iteration they reach a magnitude on the order of 10^0, and in general the relative residuals decreases of four orders of magnitude with each iteration. There is one notable exception: for the PCG method, if we use the incomplete Cholesky factorization as preconditioning matrix the convergence is much slower. We have tried to compute its condition number, but its eigenvalues were complex. Moreover, MATLAB fails to compute its eigenvalues. As expected, the convergence is linear in log-space. Even in the case of the incomplete Cholesky preconditioner, the convergence profile of its relative residuals is clearly linear from $k = 1$ until convergence is reached.

3 Estimated Computational Time

In this section we address the second and third tasks of this assignment. We have collected the average computational time for each method in Table 1. As required in the second task, we have included the component-by-component version of the Gauss-Seidel method. Note that, for each execution of the script, the estimated computational time will vary greatly, although it remains roughly in the same order of magnitude. We have used the function `process_time` from the module `time`, which has the highest precision and does not include time elapsed during sleep. From the resulting average elapsed times, it is clear that in their simplicity the

Jacobi and Conjugate Gradient methods reach convergence much faster than any other method in the sparse case. Steepest Descent is very fast both in the sparse and in the dense cases. The Conjugate Gradient is slower in the sparse case and faster than the other method in the dense case, with the exception of PCGj. The reason for which the Jacobi method is two times slower than CG and SD is that it requires two matrix-vector products, one for updating x and another one to compute the relative residual. The other two methods require a single matrix-vector product, as the relative residual is the norm of r. In the sparse case the standard solver SS is quite slow. The function is `spsolve` from the sub-module `scipy.sparse.linalg`. In the dense case the solver is still quite slow, and the function is `solve` from the sub-module `numpy.linalg`. The Gauss-Seidel method is the slowest method of all. The reason being that it requires two matrix-vector multiplication and the use of the standard linear solver, making it extremely slow. Moreover, it requires the creation of two extra matrices of the same size of the input matrix A. It is a very expensive operation in the dense case, and in fact it takes 9.44 seconds to execute.