

EP 501 Homework 2: Numerical Linear Algebra, part II

September 24, 2020

Instructions:

- Complete all listed steps to the problems.
- Submit all source code, which must be either MATLAB or Python, and output (results printed to the screen or plotted) via Canvas.
- Results must be compiled into a single .pdf file which contains descriptions of the calculations that you have done alongside the results.
- Source codes must run and produce the same essential output presented in your documents.
- Discussing the assignment with others is fine, but you must not copy the code of another student *verbatim*; this is considered an academic integrity violation.
- During submission on the canvas website compress all of your files for a given assignment into a single .zip file, e.g. `assignment1.zip` . I should be able to run your codes by unzipping the files and then opening them and running them in the appropriate developer environment (MATLAB or Spyder).
- You may use, *verbatim* or modified, any of the example codes from one of the course repositories contained on the GitHub organization website <https://github.com/Zettergren-Courses>.
- For demonstrating that your code is correct when you turn in the assignment, you must use the test problems in the course repository found in `linear_algebra/testproblem.mat` (elimination method tests, including multiple right-hand sides), `linear_algebra/lowertriang_testproblem.mat` (lower triangular tests) and `linear_algebra/iterative_testproblem.mat` (iterative method tests requiring diagonal dominance). To load these data into your workspace use:

```
load iterative_testproblem.mat
```

or double click on the .mat file in the Matlab file browser. To load these files in Python see the example included in the Python basics section of the course repository (i.e. `./basic-python/load_matlab_file.py`).

Purpose:

- Learn principles behind numerical linear algebraic techniques, in the case of this assignment LU factorization and iterative methods.

- Develop good coding and documentation practices, so that your programs are easily run and understood by others.
- Hone skills of developing, debugging, and testing your own software
- Learn how to build more complicated programs on top of existing, simple codes that have been provided to you.

1. LU factorization and its application to solve linear systems (cf. pages 45-46 in the book)
 - (a) Create a new version of your simple forward elimination function from the first assignment so that it performs Doolittle LU factorization. Please read the book section 1.4, which explains how this can be implemented in an efficient manner. The book also gives an example fortran code you can look over and potentially use as a template for constructing your MATLAB or Python code if you wish.
 - (b) Using just the output of the factorization and a back-substitution function (provided in the repository), solve the test linear system of equations given in the `../HW1/testproblem.mat` file. Note here that we are using, for this problem, the test data from the first project/assignment.
 - (c) Use your LU factorized test matrix to set up a solution for the this system with different right hand sides, i.e. solve:

$$\underline{\underline{A}} \underline{x} = \underline{b} \tag{1}$$

$$\underline{\underline{A}} \underline{x}_2 = \underline{b}_2 \tag{2}$$

$$\underline{\underline{A}} \underline{x}_3 = \underline{b}_3 \tag{3}$$

Using only forward- and back-substitution codes (along with the matrices $\underline{\underline{L}}, \underline{\underline{U}}$ from the factorization). Multiple right-hand side test data are included in the `../HW1/testproblem.mat` file from the prior assignment as the variables (`b,b2,b3`).

- (d) Use your LU factorization function and multiple right-hand side solution (using forward and back substitution as in part c) to find a matrix inverse for the test problem defined by the $\underline{\underline{A}}$ and \underline{b}_j matrices in `testproblem.mat`.
2. Iterative methods for solving linear systems:
 - (a) Starting with the Jacobi function source code from the repository (or starting from scratch if you wish), create a new function that implements successive over-relaxation.
 - (b) Try this solver on the *iterative* test problem in this directory (`./iterative_testproblem.mat`) and show that it gives the same results as the built-in Matlab utilities.
 - (c) By repeated application of your solver, iteratively adjust the relaxation parameter until you find the approximate value that minimizes the number of iterations needed to achieve converge. Use a fairly strict convergence criteria, e.g. 10^{-6} or less.
 - (d) How many fewer iterations are needed for your optimal case vs. the standard Gauss-Seidel algorithm (relaxation parameter of one). Approximately speaking which values of relaxation parameter (if any) perform worse than Gauss-Seidel?