LU decomposition in solving linear systems of equations combining Fortran and Python

Panagiotis Karagiannis *
Department of Computer Science
University of California Santa Cruz
Santa Cruz, CA 95064
https://people.ucsc.edu/ pkaragia/

Abstract

This report explores the use of a specific *matrix factorization* technique in order to solve linear systems described by $Ax = b$ where $A \in \mathbb{R}^{n\times n}$ and $x, b \in \mathbb{R}^n$. More precisely, the factorization of matrix A is particularly useful when $A = LU$ where L is lower triangular and U is upper triangular. We use *Fortran* in order to factorize A and then use L and U to compute the solution x of the linear system of equations. We limit the use of *Python* to creating initialization files for the *Fortran* routines and also plotting our results. The reason for this implementation choice is that *Fortran* is compiled and hence faster than an interpreted language, such as *Python*, when it comes to numerical computations.

1 Method

We first compute the *LU* factorization of matrix A by using Gaussian elimination with or without partial pivoting. Then we need to solve:

$$Ax = LUx = b \tag{1}$$

To solve (1) we do the following:

- First let $y = Ux$ and then use *forward substitution* to find y by solving $Ly = b$. We calculate the i^{th} entry of $y \in \mathbb{R}^n$ by setting:

$$y_i = \frac{1}{l_{ii}}\left(b_i - \sum_{j=1}^{i-1} l_{ij}\ y_j\right) \tag{2}$$

- Then we find x, using *backward substitution*, to solve $Ux = y$. Similarly, we calculate the i^{th} entry of $x \in \mathbb{R}^n$ by setting:

$$x_i = \frac{1}{u_{ii}}\left(y_i - \sum_{j=i+1}^{n} u_{ij}\ x_j\right) \tag{3}$$

2 Fortran Modules

We use Fortran to implement the numerical part of this project. The files we use are:

*https://www.soe.ucsc.edu/people/pkaragia

1. `makefile`: Compile the necessary Fortran files using appropriate dependancies and produce an executable file `linear_solve.exe`
2. `linear_solve.f90`: Driver program that calls all the necessary subroutines to solve the linear system
3. `read_data.f90`:
   (a) Contains the subroutine `get_lines` which counts the dimension of vector `b`
   (b) Contains the subroutine `read_init_files` which uses `get_lines` to allocate enough space for the matrix A and the array b and reads these variables from a predefined external .dat file
4. `write_to_screen.f90`:
   (a) Contains the subroutine `prettyPrintMatrix` which prints a matrix of arbitrary dimension into the screen
   (b) Contains the subroutine `print_A_b` which uses `prettyPrintMatrix` to print the matrix A and vector b into the screen
5. `LU_decomp.f90`
   (a) Contains the subroutine `LU` which functions as an interface calling either `LU_with_pivot` or `LU_no_pivot`
   (b) Contains the subroutine `LU_no_pivot` which computes the LU factorization using Gaussian elimination without partial pivoting or returns an error message if a pivot has value equal to 0
   (c) Contains the subroutine `LU_with_pivot` which computes the LU factorization using Gaussian elimination with partial pivoting
   (d) Contains the subroutine `all_swaps` which is used in `LU_with_pivot` to swap two rows
   (e) Contains the subroutine `get_max_index` which is used in `all_swaps` in order to get the index of the maximum element in a given column
6. `forward_solve.f90`:
   (a) Contains the subroutine `forward` which performs forward substitution
   (b) Contains the subroutine `partial_sum` which is used in `forward` to calculate the sum involved in forward substitution (see equation 2)
7. `backward_solve.f90`:
   (a) Contains the subroutine `backward` which performs backward substitution
   (b) Contains the subroutine `partial_sum` which is used in `backward` to calculate the sum involved in backward substitution (see equation 3)
8. `write_data.f90`:
   (a) Contains the subroutine `print_solution` which prints matrices L, U as well as the solution x into the screen.
   (b) Contains the subroutine `output_write` which stores the solution vector x into an external .dat file.

3 Python Script

The python `PyRun.py` script that we use is very similar to the script used for Homework 6 . More precisely, the functions we use are the following:

1. `make_make`: Calls `make clean` if the code has already been compiled and then calls `make` to produce the Fortran executable
2. `write_runtimeParemeters`: Writes the matrix A and vector b into appropriate files. If these files already exist then it shifts the existing files to an older version before overwriting them
3. `run_lu`: Runs the executable file produced by `make_make`

4. `solution_check`: Checks if each entry of the solution vector returned by the fortran files is within 10^{-4} of the vector returned by the numpy library when solving directly the linear system i.e. $x = A^{-1}b$. This function uses an `assert` statement instead of printing to the screen.
5. `plot_data`: Performs the plotting of the matrix A and the vectors b, x

4 Results

We test our program using a set of three pairs of matrices A and corresponding vectors b. It has to be noted that our Fortran solution for all test cases are very close to the solution produced by Python for the same linear system. More precisely, every entry of the solution vector produced by Fortran is within 10^{-4} of the corresponding entry of the solution vector produced by `numpy`. Namely:

$$|\texttt{x\_fortran}[i] - \texttt{x\_python}[i]| \leq 10^{-4}$$

4.1 Test Case 1

Let:

$$A_1 = \begin{bmatrix} 1 & 1 & -1 \\ 1 & 2 & -2 \\ -2 & 1 & 1 \end{bmatrix}, b_1 = \begin{bmatrix} 1 \\ 1 \\ -2 \end{bmatrix}$$

then after running our program we get that a solution to $A_1 x_1 = b_1$ is

$$x_1 = \begin{bmatrix} 2 \\ 2 \\ 3 \end{bmatrix}$$

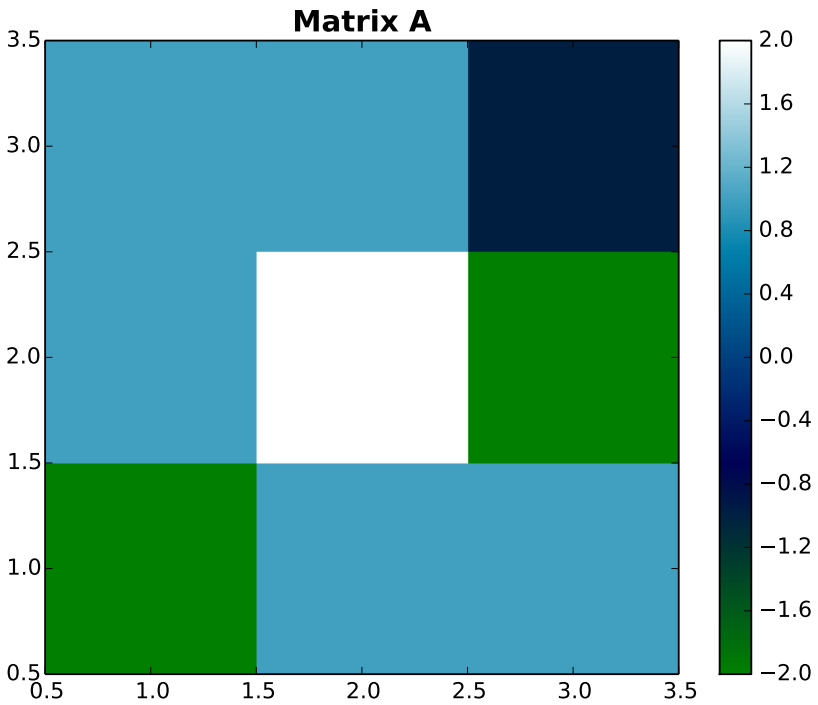

Figure 1: Plot of matrix A_1

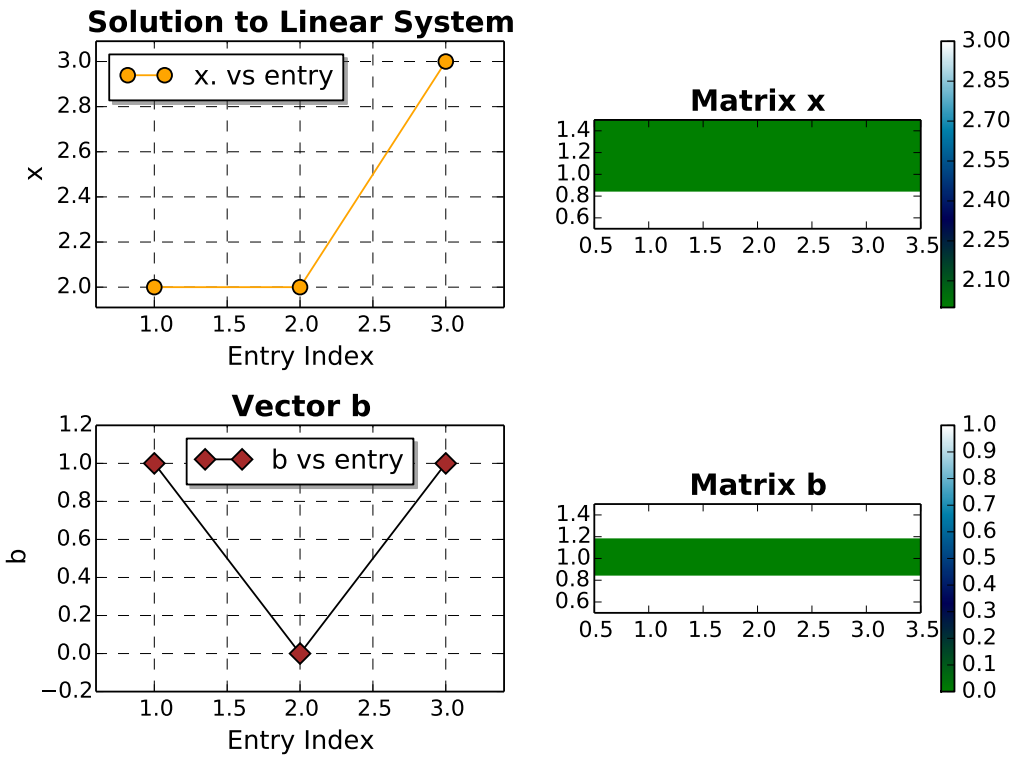

Figure 2: Plots of the values of vectors x_1, b_1 at a given index (x-axis)

4.2 Test Case 2

Let:

$$A_2 = \begin{bmatrix} 4 & 3 & 2 & 1 \\ 3 & 4 & 3 & 2 \\ 2 & 3 & 4 & 3 \\ 2 & 3 & 4 & 3 \end{bmatrix}, b_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix}$$

then after running our program we get that a solution to $A_2 x_2 = b_2$ is

$$x_2 = \begin{bmatrix} 0 \\ 1 \\ -1 \\ 0 \end{bmatrix}$$

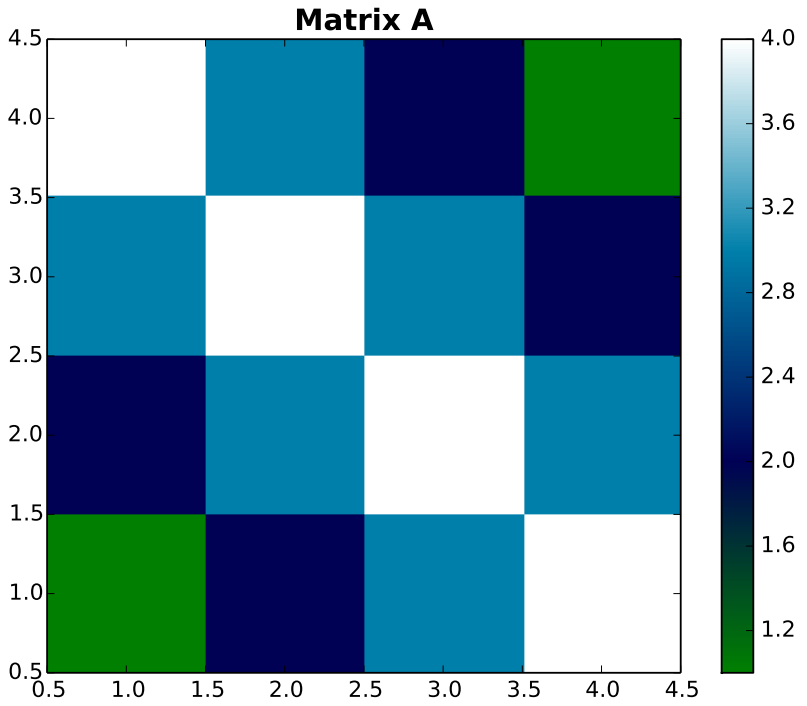

Figure 3: Plot of matrix A_2

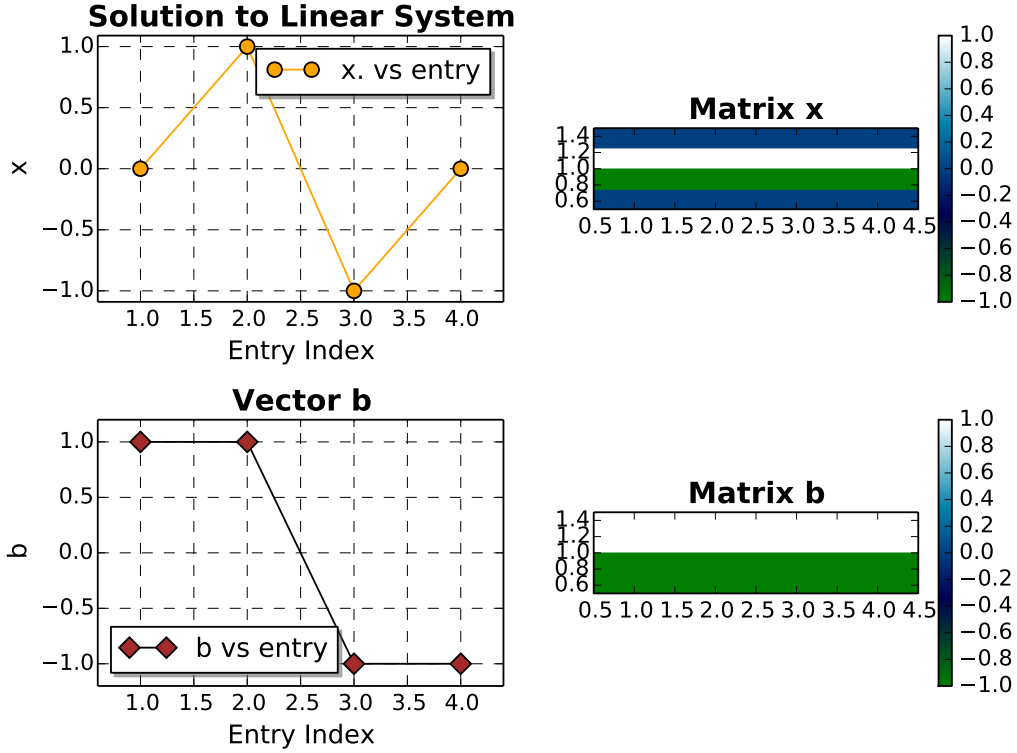

Figure 4: Plots of the values of vectors x_2, b_2 at a given index (x-axis)

4.3 Test Case 3

Let:

$$A_3 = \begin{bmatrix} 1 & -1 & 1 & -1 \\ -1 & 3 & -3 & 3 \\ 2 & -4 & 7 & -7 \\ -3 & 7 & -10 & 14 \end{bmatrix}, b_3 = \begin{bmatrix} 0 \\ 2 \\ -2 \\ -8 \end{bmatrix}$$

then after running our program we get that a solution to $A_3 x_3 = b_3$ is

$$x_3 = \begin{bmatrix} 1 \\ 1 \\ -3 \\ -3 \end{bmatrix}$$

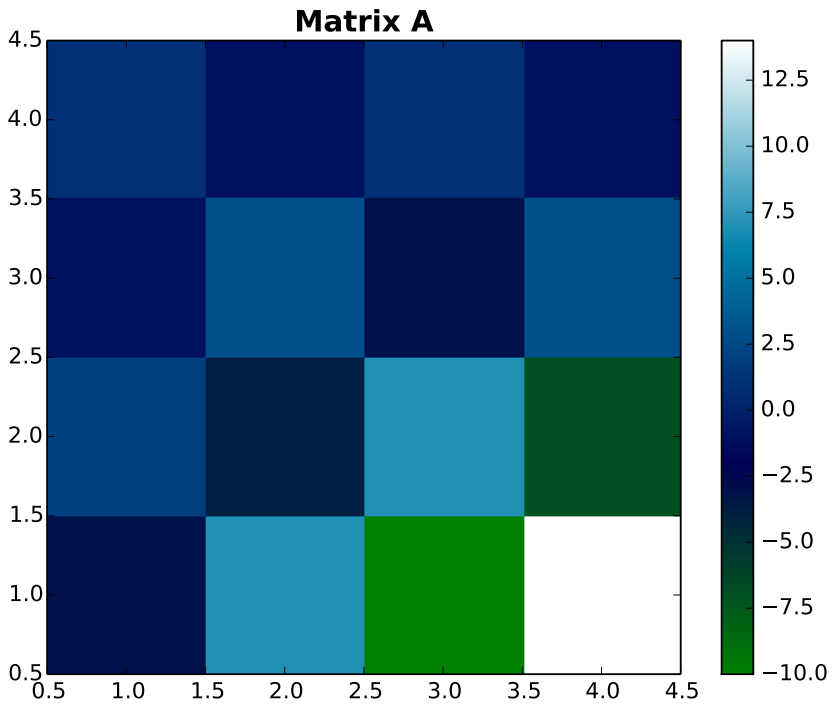

Figure 5: Plot of matrix A_2

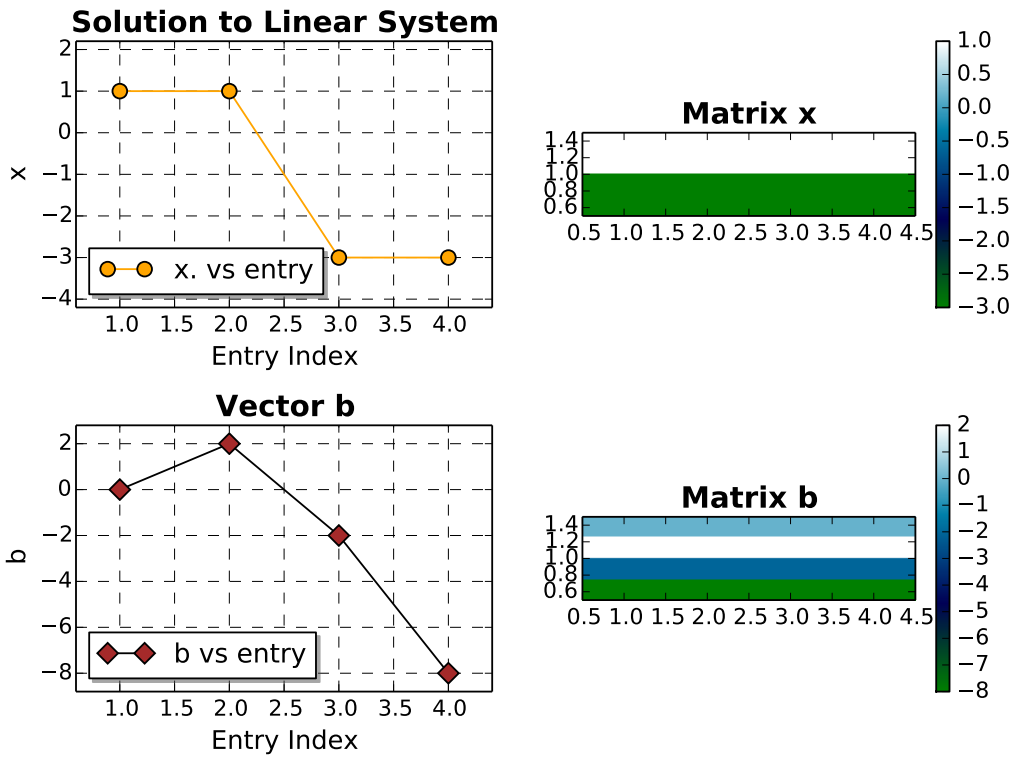

Figure 6: Plots of the values of vectors x_3, b_3 at a given index (x-axis)

5 Findings

We notice that only for the first example the `LU_with_pivot` and `LU_no_pivot` produced different L, U matrices for A. All solutions, regardless of whether we used partial pivoting or not were very close to the solution produced by Python's `numpy` library. Pivoting techniques such as partial pivoting are extremely useful tools, since not only they allow the gaussian elimination to continue, but also minimize the round-off errors in case of small pivot elements.

Moreover, all three linear systems we considered had a unique solution. Nevertheless, this should not come as a surprise since for all examples we have that A_i is invertible which is equivalent to $det(A_i) \neq 0$.

6 Comments

Producing the LU factorization of a matrix A proves to be very useful when attempting to solve linear systems of the form $Ax = b$. Solving a linear system by using Gaussian elimination requires $\mathcal{O}(n^3)$ operations, nevertheless, solving a linear with either an upper or lower triangular matrix requires $\mathcal{O}(n^2)$ arithmetic operations. Therefore, given matrices L, U we can solve the linear system in quadratic time with respect to the size of the input. As expected this runtime improvement comes at the cost: producing the LU decomposition takes $\mathcal{O}(n^3)$ time. Nonetheless, the advantage of computing the LU decomposition of a matrix, is that we only need to calculate it once. More precisely we will not have to recompute the matrix factorization for each individual value of the vector b in the right hand side of the equation. Therefore, if we expect to solve many linear systems where matrix A remains the same but vector b changes, then the proposed method reduces significantly the amount of computation we have to make.

7 Conclusion

Finally, in this project we explored the use of *LU* decomposition and its applications in solving systems of linear equations. Moreover, we utilized two very different languages in order to complete our program. We used a compile language such a s Fortran for its computational efficiency and an interpreted language such as Python for producing plots and initialization files. Combining these two languages we were able to create a program that scales efficiently with large matrices and also produces useful insights on the solutions to the linear systems.