Final Project
Numerical Linear Algebra

AMS209: Foundations of Scientific Computing

Brian Vargas

December 7, 2016

In numerical linear algebra, one of the most important problems is solving the equation $Ax = b$, where A is an $n \times n$ matrix, and x, b are $n \times 1$ column vectors. Symbolically, the solution is simply obtained as $x = A^{-1}b$ when A is invertible. However, in practice, it is not so easy to generate an inverse matrix directly.

Instead of generating the inverse matrix directly, we take an alternative, but equivalent, clever approach - via LU Decomposition. We decompose the matrix A into two matrices also of size $n \times n$ L, U so that $A = LU$, where L is a unit lower triangular matrix and U is an upper triangular matrix. By exploiting their triangular form, we can easily obtain the solution x using forward and backward substitution. In this final project we deal with both LU Decomposition with and without Partial Pivoting.

We combined the compiled language Fortran with the scripting language Python 2 in order to best handle the execution of this final project. Since compiled languages have a faster runtime than scripting languages, we used Fortran as the scientific computing engine in modular style to optimize organization. Since Python is a powerful high-level programming language, we used it for all other tasks: directory traversals, data file creation, calling the makefile and exutables, manipulating files, checking solutions, and generating plots.

1 Methods

I will explain how the code interacts as well as the mathematical reasoning that went into the algorithm design.

linear_solve.f90 is the Fortran driver file. It directs the necessary modules and subroutines in order for the entire computing sequence to take place.

Within the driver file we see the sequence for obtaining one solution are as follows:

1. Read in data from dat files and form A and b: read_data.f90
2. Print A and b to the screen for sanity check: write_to_screen.f90
3. Obtain LU Decomposition of matrix A: LU_decomp.f90
4. Applies forward substitution to solve $Ly = b$: forward_solve.f90
5. Applies backward substitution to solve $Ux = y$: backward_solve.f90
6. Save solution to file for later analysis: write_data.f90

Note that I ran steps (3)-(6) twice - once for LU Decomposition without pivoting and once for with pivoting. The solution for no pivoting gets saved as x_1.dat and the solution with pivoting gets saved as p_1.dat.

1.1 Reading Data

The inputted data from the dat files were only numerical values identifying the matrix values. I designed a way to know what size the matrices were supposed to be by counting the number of entries in the vector b file since that corresponds with the size n. Then I allocated and filled A, b accordingly. Note the clean programming style of deallocating at the end of the program!

1.2 Print and Save Data

Nothing fancy here - I printed the matrices by row and the vector by entry. I linked the print_to_screen module to the write_data module for easy access when printing the solution at the end of the sequence.

1.3 LU Decomposition

We decompose our $n \times n$ matrix A into two $n \times n$ matrices L, U where L is a unit lower triangular matrix and U is an upper triangular matrix. As an example, this is what it looks like for a 3×3 matrix:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \ell_{21} & 1 & 0 \\ \ell_{31} & \ell_{32} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

We obtain the U matrix by multiplying A by the left by a matrix that zeros out the entries below the first diagonal then multiplying that product again by the left by a matrix that now zeros out the entries below the second diagonal and continuing this process until we end up with the upper triangular matrix that is U. L is then the inverse of all these matrices multiplied together - easily obtained by just taking the opposite of each entry.

In my Fortran code, I wasted less memory space by placing both the L and U matrices

in the same LU matrix so that all the entries below the diagonal belonged to L and everything on the diagonal and above belonged to U. Thus, I stored my LU Decomposition as follows:

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ \ell_{21} & u_{22} & u_{23} \\ \ell_{31} & \ell_{32} & u_{33} \end{bmatrix}$$

1.3.1 Partial Pivoting

The only difference between the LU Decomposition with partial pivoting from without pivoting is in the step right before we eliminate the entries below the diagonal. When we eliminate the entries, we use the diagonal entry as the divisor.

For example, if we wanted to eliminate a_{31}, we would use $\ell_{31} = \dfrac{a_{31}}{a_{11}}$. However, notice how if we don't manipulate the matrix prior to using this divisor, we can potentially run into problems if we have a zero on the diagonal.

Therefore, to avoid the danger of dividing by zero, we swap rows so that the entry of largest absolute value for that particular column at or below the diagonal entry ends up on the diagonal. Thus, we swap the rows for L, b and the current matrix being manipulated $\bar{A}$. This way in the end, we end up with the right corresponding equivalencies.

Note that when partial pivoting, we technically decompose A into three $n \times n$ matrices: P^T, L, U. This way $A = P^T LU$, where P is the orthogonal permutation matrix holding the information for how b should be swapped. That is, $Ax = LUx = Pb$. However, to avoid outputting a permutation matrix which wastes a ton of memory, I simply swapped the rows of inputted b vector and outputted it at the end of the code.

1.4 Forward Substitution

This technique is used to solve a lower triangular system of equations. We solve $Ly = b$ from top to bottom for the $n \times 1$ column vector y. This is best explained by the following equation where i goes from 1 to n.

$$y_i = b_i - \sum_{j=1}^{i-1} \ell_{ij} y_j.$$

1.5 Backward Substitution

This technique is used to solve an upper triangular system of equations. We solve $Ux = y$ from bottom to top for our final solution x. This is best explained by the following equation where i goes from n to 1.

$$x_i = \frac{y_i - \sum_{j=i+1}^{n} u_{ij} x_j}{u_{ii}}.$$

2 Results

We tested our code on the following matrices.

$$A = \begin{bmatrix} 1 & 1 & -1 \\ 1 & 2 & -2 \\ -2 & 1 & 1 \end{bmatrix}, b = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \tag{1}$$

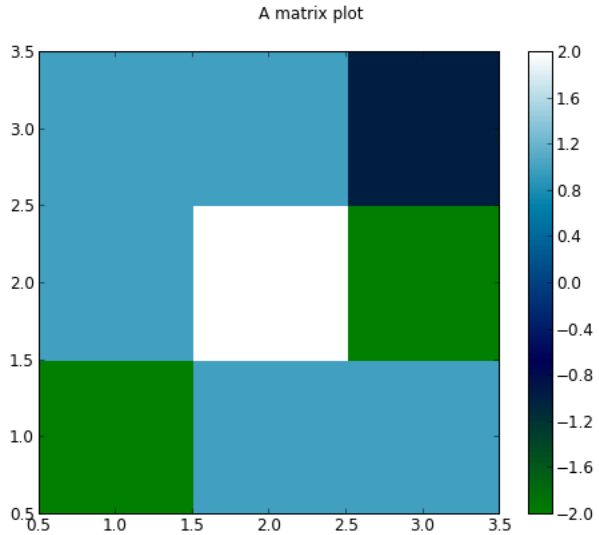
$$A = \begin{bmatrix} 4 & 3 & 2 & 1 \\ 3 & 4 & 3 & 2 \\ 2 & 3 & 4 & 3 \\ 1 & 2 & 3 & 4 \end{bmatrix}, b = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix} \tag{2}$$

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

Note that to keep the computations consistent for the general case, all data were inputted to Fortran as floating-point real numbers.

2.1 Matrix 1

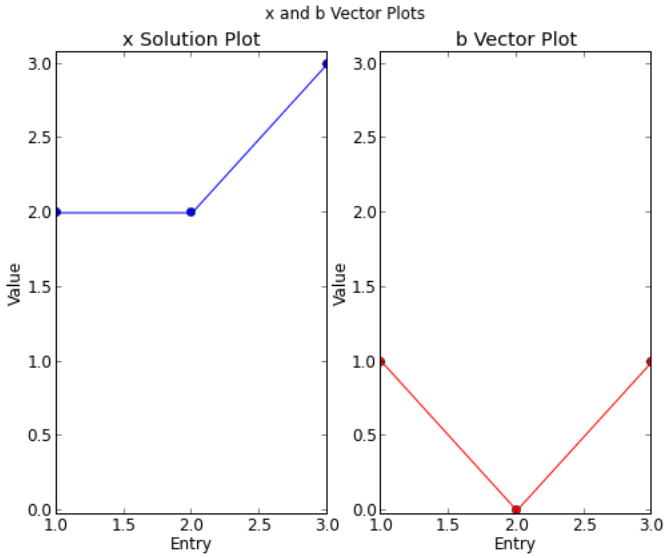
We plotted the initial matrix A to give a graphic for the values it holds.



Upon running the code we found the solution to be

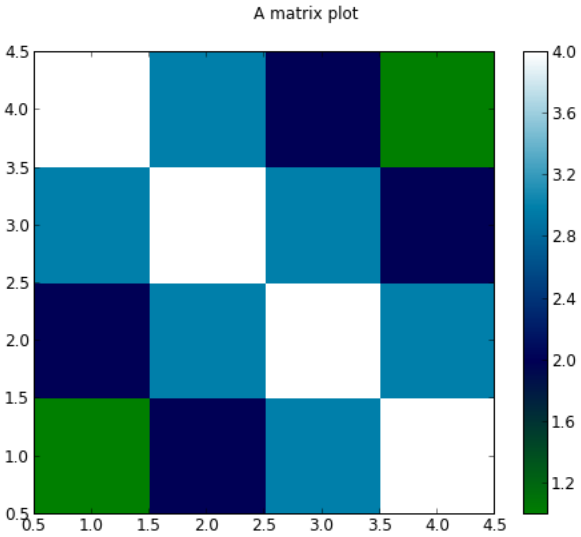
$$x = \begin{bmatrix} 2 \\ 2 \\ 3 \end{bmatrix}.$$

We plotted the solution x alongside the initial vector b.



2.2 Matrix 2

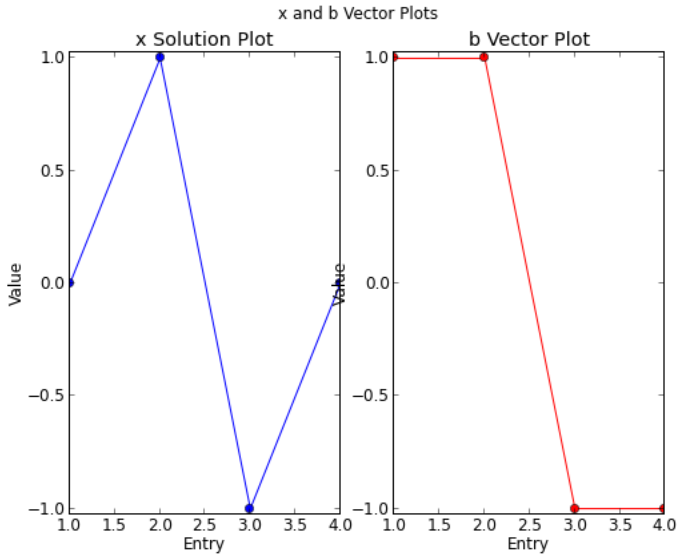
We plotted the initial matrix A to give a graphic for the values it holds.



Upon running the code we found the solution to be

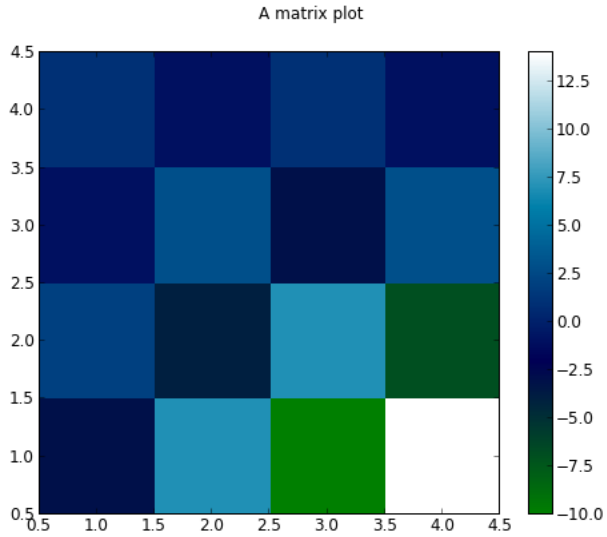
$$x = \begin{bmatrix} 0 \\ 1 \\ -1 \\ 0 \end{bmatrix}.$$

We plotted the solution x alongside the initial vector b.



2.3 Matrix 3

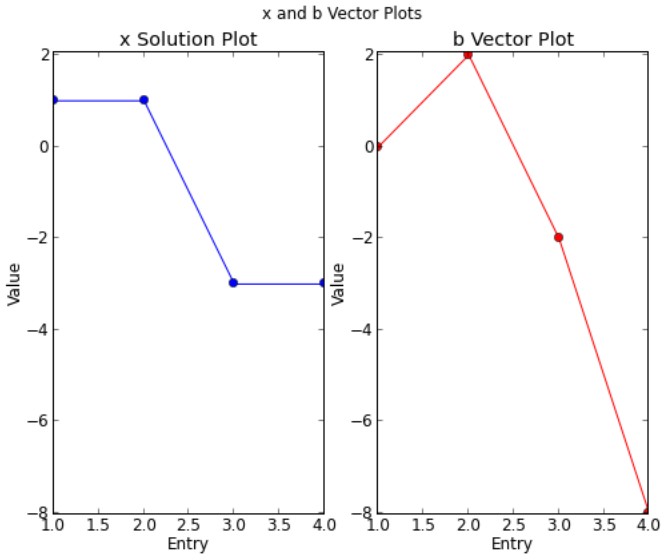
We plotted the initial matrix A to give a graphic for the values it holds.



Upon running the code we found the solution to be

$$x = \begin{bmatrix} 1 \\ 1 \\ -3 \\ -3 \end{bmatrix}.$$

We plotted the solution x alongside the initial vector b.



3 Findings

For all three test cases, we saw that we essentially obtained the same correct solutions for all three methods: pivoting, no pivoting, and Python. Each solution had an absolute error from each other no more than 10^{-15}. Note, absolute error was computed using the 2-norm:

$$E = ||x_1 - x_2||_2$$

4 Conclusion

Although we obtained correct solutions for all three test cases in this project, it is not viable to use the method without pivoting. That is solely for the reason that we don't know if we will ever end up with a zero entry on the diagonal that ends up ruining our computation. It's safer to use a method with pivoting, which is what Python uses in its matrix solve method.

Furthermore, most of the work we did in Fortran was overkill for this project. That was a bunch of modular programming in a compiled language for such small matrices. In practice, we are better off using a scripting language when the data is so small because the difference in runtime performance is minimal. Had we stuck to just using the Python method from the very beginning, we would have saved so much time dealing with Fortran. However, when we begin dealing with larger data, the Fortran code would be most handy due to its optimization benefits.

Nonetheless, using a LU Decomposition showed to be useful when solving $Ax = b$. We are aware that in practice, this is the best approach to take for a general equation. That may not be the case if the data has certain patterns we can exploit but as a starter, this would be the way to go.

5 Comments

It is interesting to note that in LU_decomp.f90, although I tried to not be wasteful by compacting my matrices, I could have wasted less memory yet. As I found more entries for L, I could have inputted those directly into the current 'A' matrix and defined a matrix multiplication using summations only for the terms in the row after the diagonal. This way, the same matrix that was inputted would be the same matrix that gets outputted - we wouldn't have to allocate for any other matrices because of this recycling.