AM 129: Final Project- Numerical Linear Algebra Deliverable

Sarosh Sopariwalla
December 18, 2020

Abstract

From the available final projects for AM 129, I have chosen to explore the one that deals with implementing basic linear algebra algorithms in FORTRAN (with program execution in Python). Our overall goal is to solve an n-dimensional system of equations that has n variables using Gaussian Elimination. We explore various ways of doing so and conclude with a numerical analysis of our methods.

1 Theoretical Motivations

It is well known that we can solve any linear system of equations through Gaussian Elimination of the augmented coefficient matrix of our system. However, there are various algorithms to use when conducting Gaussian Elimination. They all have two steps in common: first, reduce the coefficient matrix into upper triangular form and then use back substitution to get our matrix into reduced row echelon form. Our Gaussian Elimination algorithms differ in the first step.

We explore two methods of reducing a matrix into upper triangular form: the first is the one most students learn in an Elementary Linear Algebra class. The procedure states that you go row by row and scale it such that the term in the pivot position becomes a one. This works fine until there is inevitably a zero in the pivot position. Then, the algorithm fails because obviously we can't scale the row by $\frac{1}{0}$. The obvious solution is then to swap the row with a zero in the pivot position with a row below it that has a non-zero term in that column. The partial-pivoting algorithm employs a specific version of that idea.

The partial pivoting algorithm has us consider the column of values that contains the pivot position and then find the which row has the element with the largest magnitude. We then take that row and put it into the row originally in the pivot position. Ideally, this algorithm should always work and return the correct result. It's worth noting that if the first algorithm returns a matrix with well defined numbers, the answer is guaranteed to be correct. If the algorithm leads to the wrong answer we will get solutions that contains at least one NaN or "Not a Number."

2 Code Design and Structure

2.1 Approach and Cases

We test a few cases to see how both algorithms work. The first four cases are simple, we have a coefficient matrix A_i and a right-hand side vector b_i. We then solve for the solution vector x using both regular Gaussian Elimination and Gaussian Elimination with Partial Pivoting. The respective matrices and vectors can be found below:

$$\mathbf{A}_1 = \begin{bmatrix} 1 & 1 & -1 \\ 1 & 2 & -2 \\ -2 & 1 & 1 \end{bmatrix}, \mathbf{b}_1 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

$$\mathbf{A}_2 = \begin{bmatrix} 4 & 3 & 2 & 1 \\ 3 & 4 & 3 & 2 \\ 2 & 3 & 4 & 3 \\ 1 & 2 & 3 & 4 \end{bmatrix}, \mathbf{b}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix}$$

$$\mathbf{A}_3 = \begin{bmatrix} 1 & -1 & 1 & -1 \\ -1 & 3 & -3 & 3 \\ 2 & -4 & 7 & -7 \\ -3 & 7 & -10 & 14 \end{bmatrix}, \mathbf{b}_3 = \begin{bmatrix} 0 \\ 2 \\ -2 \\ -8 \end{bmatrix}$$

$$\mathbf{A}_4 = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}, \mathbf{b}_4 = \begin{bmatrix} 3 \\ 6 \\ 10 \\ 1 \end{bmatrix}$$

The fifth case is slightly more complex; we consider the coefficient matrix A_4 and augment it with with $b_2b_3b_4$. To solve this system, we use Gaussian Elimination on the whole matrix (with all three augmented vectors). In the following sections, we will show how we consider the various cases through our program.

2.2 Python Implementation

The first part of our Python program contains a function called "generate_input". This moves from the pyRun directory to the fortran directory. Once there, we put the contents of a certain matrix and vector file into A.dat and b.dat. For instance, if we call generate_input(A_1.dat , b_1.dat) it will copy the contents of A_1.dat and b_1.dat into A.dat and b.dat. This is important because our FORTRAN program is built in a way such that it always reads from the files A.dat and b.dat.

Now our executable section has a few parts. The first part is meant to solve the four cases and save their outputs into files called x_piv(i).dat and x_nopiv(i).dat where (i) is replaced by the case we are studying. The terminal/console output from this code can be found below:

Console Output

```
Welcome! We will now solve equations (1)-(5) using some Fortran code.
We will now solve the five cases in equations (1)-(5)
We solve case(1)
 Matrix A:
   1.0000000000000000        1.0000000000000000       -1.0000000000000000
   1.0000000000000000        2.0000000000000000       -2.0000000000000000
  -2.0000000000000000        1.0000000000000000        1.0000000000000000
 Vector b:
   1.0000000000000000
   0.0000000000000000
   1.0000000000000000
 ----------
 Using regular gaussian elimination, we reduce A into the following upper triangular matrix:
   1.0000000000000000        1.0000000000000000       -1.0000000000000000
   0.0000000000000000        1.0000000000000000       -1.0000000000000000
   0.0000000000000000        0.0000000000000000        2.0000000000000000
 Using back substitution, we store our solution in x_nopiv1.dat file
 ----------
 We can also use partial pivoting to reduce A into the following upper triangular matrix:
  -2.0000000000000000        1.0000000000000000        1.0000000000000000
   0.0000000000000000        2.5000000000000000       -1.5000000000000000
   0.0000000000000000        0.0000000000000000       0.39999999999999991
 Using back substitution, we store our solution in x_piv1.dat file
We solve case(2)
 Matrix A:
   4.0000000000000000        3.0000000000000000        2.0000000000000000        1.0000000000000000
   3.0000000000000000        4.0000000000000000        3.0000000000000000        2.0000000000000000
   2.0000000000000000        3.0000000000000000        4.0000000000000000        3.0000000000000000
   1.0000000000000000        2.0000000000000000        3.0000000000000000        4.0000000000000000
 Vector b:
   1.0000000000000000
   1.0000000000000000
  -1.0000000000000000
  -1.0000000000000000
 ----------
 Using regular gaussian elimination, we reduce A into the following upper triangular matrix:
   4.0000000000000000        3.0000000000000000        2.0000000000000000        1.0000000000000000
   0.0000000000000000        1.7500000000000000        1.5000000000000000        1.2500000000000000
   0.0000000000000000        0.0000000000000000        1.7142857142857144        1.4285714285714286
   0.0000000000000000        0.0000000000000000        0.0000000000000000        1.6666666666666667
 Using back substitution, we store our solution in x_nopiv2.dat file
 ----------
 We can also use partial pivoting to reduce A into the following upper triangular matrix:
   4.0000000000000000        3.0000000000000000        2.0000000000000000        1.0000000000000000
   0.0000000000000000        1.7500000000000000        1.5000000000000000        1.2500000000000000
   0.0000000000000000        0.0000000000000000        1.7142857142857144        1.4285714285714286
   0.0000000000000000        0.0000000000000000        0.0000000000000000        1.6666666666666667
 Using back substitution, we store our solution in x_piv2.dat file
We solve case(3)
 Matrix A:
   1.0000000000000000      -1.0000000000000000       1.0000000000000000      -1.0000000000000000
  -1.0000000000000000       3.0000000000000000      -3.0000000000000000       3.0000000000000000
   2.0000000000000000      -4.0000000000000000       7.0000000000000000      -7.0000000000000000
  -3.0000000000000000       7.0000000000000000      -10.000000000000000       14.000000000000000
 Vector b:
   0.0000000000000000
   2.0000000000000000
  -2.0000000000000000
  -8.0000000000000000
 ----------
 Using regular gaussian elimination, we reduce A into the following upper triangular matrix:
   1.0000000000000000      -1.0000000000000000       1.0000000000000000      -1.0000000000000000
   0.0000000000000000       2.0000000000000000      -2.0000000000000000       2.0000000000000000
   0.0000000000000000       0.0000000000000000       3.0000000000000000      -3.0000000000000000
   0.0000000000000000       0.0000000000000000       0.0000000000000000       4.0000000000000000
 Using back substitution, we store our solution in x_nopiv3.dat file
 ----------
 We can also use partial pivoting to reduce A into the following upper triangular matrix:
  -3.0000000000000000       7.0000000000000000      -10.000000000000000       14.000000000000000
   0.0000000000000000       1.3333333333333330      -2.3333333333333330       3.6666666666666661
   0.0000000000000000       0.0000000000000000       1.5000000000000002      -3.5000000000000000
   0.0000000000000000       0.0000000000000000       0.0000000000000000       3.9999999999999991
 Using back substitution, we store our solution in x_piv3.dat file
We solve case(4)
 Matrix A:
   2.0000000000000000       1.0000000000000000       1.0000000000000000       0.0000000000000000
   4.0000000000000000       3.0000000000000000       3.0000000000000000       1.0000000000000000
   8.0000000000000000       7.0000000000000000       9.0000000000000000       5.0000000000000000
   6.0000000000000000       7.0000000000000000       9.0000000000000000       8.0000000000000000
 Vector b:
   3.0000000000000000
   6.0000000000000000
   10.000000000000000
   1.0000000000000000
 ----------
 Using regular gaussian elimination, we reduce A into the following upper triangular matrix:
   2.0000000000000000       1.0000000000000000       1.0000000000000000       0.0000000000000000
   0.0000000000000000       1.0000000000000000       1.0000000000000000       1.0000000000000000
   0.0000000000000000       0.0000000000000000       2.0000000000000000       2.0000000000000000
   0.0000000000000000       0.0000000000000000       0.0000000000000000       2.0000000000000000
 Using back substitution, we store our solution in x_nopiv4.dat file
 ----------
 We can also use partial pivoting to reduce A into the following upper triangular matrix:
   8.0000000000000000       7.0000000000000000       9.0000000000000000       5.0000000000000000
   0.0000000000000000       1.7500000000000000       2.2500000000000000       4.2500000000000000
   0.0000000000000000       0.0000000000000000     -0.85714285714285721     -0.28571428571428581
   0.0000000000000000       0.0000000000000000       0.0000000000000000      0.66666666666666674
 Using back substitution, we store our solution in x_piv4.dat file
```

As you can see for each case, it prints the A matrix as well as the b vector. Then, using regular Gaussian Elimination, we print out the upper triangular matrix and get a message showing where the solution has been stored. Similarly, we see the upper triangular matrix that comes from Gaussian Elimination with Partial Pivoting and a message that shows where the solution vector is stored. (As for checking if these solutions are correct, we will do this later.)

The final part of the first section is for solving case five. (A reasonable question here is why this is not part of the for-loop that is used for cases one through four.) The reason for this is because our FORTRAN program needs to be called with an argument that specifies that our right hand side of the augment coefficient matrix has multiple columns. When calling our executable file we call it with the following format "./linear_solve # #" The first number argument should be a number 1-4 that represents which case we want to solve. The second number can be anything when we are calling cases 1-4. However, if we want to solve case 5, we must write a 5 as the second argument. Once that is done it will read in the right hand side of the augmented matrix from the file Y.dat The output for case 5 is below.

Console Output

```
We solve the final unique case (5) where A_4 X = B
 Here is matrix A_4:
   2.0000000000000000        1.0000000000000000        1.0000000000000000        0.0000000000000000
   4.0000000000000000        3.0000000000000000        3.0000000000000000        1.0000000000000000
   8.0000000000000000        7.0000000000000000        9.0000000000000000        5.0000000000000000
   6.0000000000000000        7.0000000000000000        9.0000000000000000        8.0000000000000000
 Here is our matrix B:
   1.0000000000000000        0.0000000000000000        3.0000000000000000
   1.0000000000000000        2.0000000000000000        6.0000000000000000
  -1.0000000000000000       -2.0000000000000000        10.000000000000000
  -1.0000000000000000       -8.0000000000000000        1.0000000000000000
 ________
 Using regular gaussian elimination and back substitution we see X =
                      NaN                       NaN                       NaN
   0.0000000000000000        6.0000000000000000        1.0000000000000000
  -2.0000000000000000        0.0000000000000000        2.0000000000000000
   1.0000000000000000       -4.0000000000000000       -3.0000000000000000
 Clearly this a case where basic gaussian elimination fails us.
 Trying with partial pivoting and back substitution, X =
   1.4999999999999991       -2.9999999999999991        8.7430063189231078E-016
   1.1419436824715897E-015   5.9999999999999982       0.99999999999999878
  -2.0000000000000000        0.0000000000000000        1.9999999999999996
  0.99999999999999967       -3.9999999999999991       -2.9999999999999991
 As expected, partial pivoting fixed the issue :)
 We save this solution to X.dat
```

The second part of our python code checks the accuracy of our results. Essentially, for each case, we read in the desired matrix/vector and solve the system using numpy's linalg.solve command. We then store the solution and compare it to the results we got from gaussian elimination with and without partial pivoting. Since there is a degree of mechanical "error", we compare solutions using numpy's allclose function. This checks that the corresponding terms are all within a certain tolerance of each other. We see that in all cases the results were basically the same (and our test therefore passed.) In the fifth test, we don't check the results from regular Gaussian Elimination. That's because our final solution contains NaN values, so we know it is incorrect. The output from this section can be found below.

Console Output

```
We now check the accuracy of our results
Case 1
PASS (METHOD: PIVOT)
PASS (METHOD: REGULAR)
---------------------------------
Case 2
PASS (METHOD: PIVOT)
PASS (METHOD: REGULAR)
---------------------------------
Case 3
PASS (METHOD: PIVOT)
PASS (METHOD: REGULAR)
---------------------------------
Case 4
PASS (METHOD: PIVOT)
PASS (METHOD: REGULAR)
---------------------------------
Special Case: Case 5
We have already seen regular gaussian elimination fails so we only
compare the result from partial pivoting.
PASS (METHOD: PIVOT)
```

Finally the last part of our python code explores how to plot matrices and vectors using matplotlib. For each of the cases one through three, we plot the coefficient matrix A as well as the right hand side vector b and the solution vector x. The vectors x and b are put together into one plot (with each of them being a subplot.) These plots can be seen on the last page of the report.

2.3 FORTRAN Code

Finally, we explore what exactly is happening when we call our FORTRAN executable from Python. As mention earlier, the call requires two arguments like so ./linear_solve # # For the first four cases, the first argument represents which case we are looking at; for example ./linear_solve 3 1 will tell the compiler we are solving the third case. This is primarily useful for the program when it comes to storing our output. Putting a 3 as our first argument, stores the solutions from partial pivoting and no partial pivoting into files called. x_piv3.dat and x_nopiv3.dat.

The second argument represents whether or not we are dealing with the special case five. If the second argument equals five, our code runs slightly differently since the augmented part of our matrix has multiple columns. In the earlier case, we had a subroutine created two read in the data from A.dat and b.dat where A held a matrix and b.dat stored a vector. Now, we needed a subroutine that would read in a full matrix instead of a vector. While I am sure there would have been some way to add this functionality into the subroutine we already had, it was far quicker to write another quick subroutine that read in A.dat and Y.dat where Y.dat contained the array of vectors $b_1 b_2 b_3$. Given that we now understand how our arguments work, it is easy to see the overall structure. It essentially comes down to one large if-else statement; if the second argument does not equal five, we solve our system using the regular subroutines as needed (more detail below). If the second arguemnt equals five, we will employ the same overall method but with slightly different subroutines.

2.3.1 Cases One Through Four

This part of the code is represented in the else part of our large if statement found in linear_solve.f90; we explain the overall process here. The fact that we made it to the else part of our if statement implies that the second argument was not equal to five. Thus, the first thing we do is read in the first argument and store it into a variable called arg. arg will represent what case number we are solving. We then call a subroutine called ReadInMatrix(A,b) that reads in the data from A.dat and b.dat and stores them into allocatable 2D and 1D arrays respectively. (Recall that the proper matrix for each case is copied into the file A.dat and b.dat by the python code earlier so our FORTRAN code never has to look for any other files.) We then print Matrix A and Vector b out to the console so that the user can confirm there was no errors in reading in the matrices. To print the matrix and vector we use a subroutine PrintMatrix and PrintVector that can be found in write_data.f90. Next, we reduce it into upper triangular form using the subroutine BasicGE (defined in the ge_elim.f90) module. BasicGE does gaussian elimination without partial pivoting. We then print matrix A in upper triangular form and give a message saying that the final solution can be found through back substitution and is stored in x_nopiv# where # is the number stored in arg. We do the exact same thing except we use partial pivoting and then store the solution into x_piv#.dat. Clearly, having all theses subroutines helps simplify the code. The purpose of these subroutines is fairly clear and the code for them is straightforward, so we don't delve too deeply into them, but will explain a little bit about their structure in the next section.

2.3.2 Case Five

Case five is initiated when the second argument of ./linear_solve is equal to five. Now an issue arises because our ReadInMatrix subroutine was designed to read in a matrix and a vector, but here we want to read in two matrices. To solve this issue, we used ReadInMatrix to read in the A Matrix but for the right hand side, we created a subroutine ReadJointVectors that reads in the joint vectors from Y.dat. Then, we called the BasicGE subroutine to run regular gaussian elimination on the whole augmented matrix. Of course, this subroutine also expected a matrix and a vector (not two matrices.) To fix this, I added an optional INOUT variable Y that represents another matrix. Then, I basically told the compiler if Y is inputted, do the same row operations that were done on matrix A. The same logic follows for the PartialPivot GE subroutine. Thus, in this case when calling the subroutines we needed to add the extra parameter Y, which is why we separated our code into dealing with cases one through four apart from case five.

3 Conclusion and Analysis of Results

It is clear that the basic Gaussian Elimination algorithm works in most cases until there is a zero in the pivot position. Luckily, this issue didn't actually come up until we dealt with Matrix A_4 in case 5. Thus, in general, if we want to implement a Gaussian Elimination algorithm, it would be wise to use partial pivoting just in case. The complexity of adding the partial pivot step is quite minimal compared to the benefit of having a program that works in all cases. Below we show the solutions we received for all cases as well as the plots mentioned earlier.

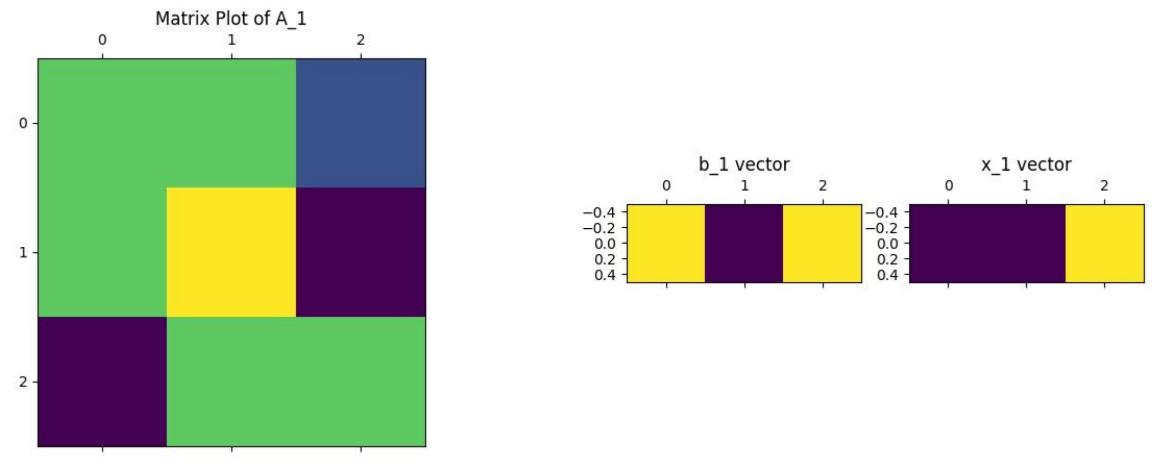
In case 1, we found that $x_1 = \begin{pmatrix} 2 \\ 2 \\ 3 \end{pmatrix}$ (with and without partial pivoting)

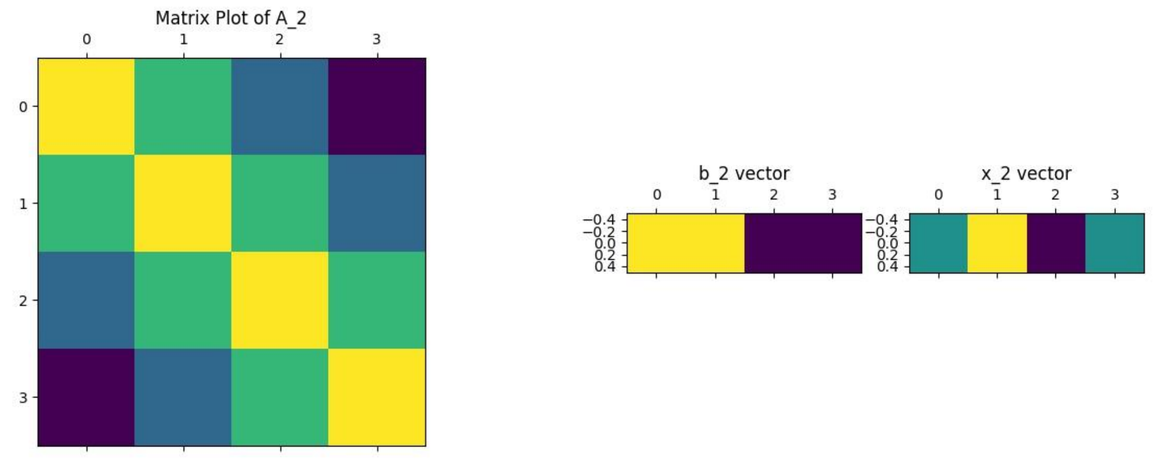
In case 2, we saw that $x_2 = \begin{pmatrix} 0 \\ 1 \\ -1 \\ 0 \end{pmatrix}$ (with and without pivoting)

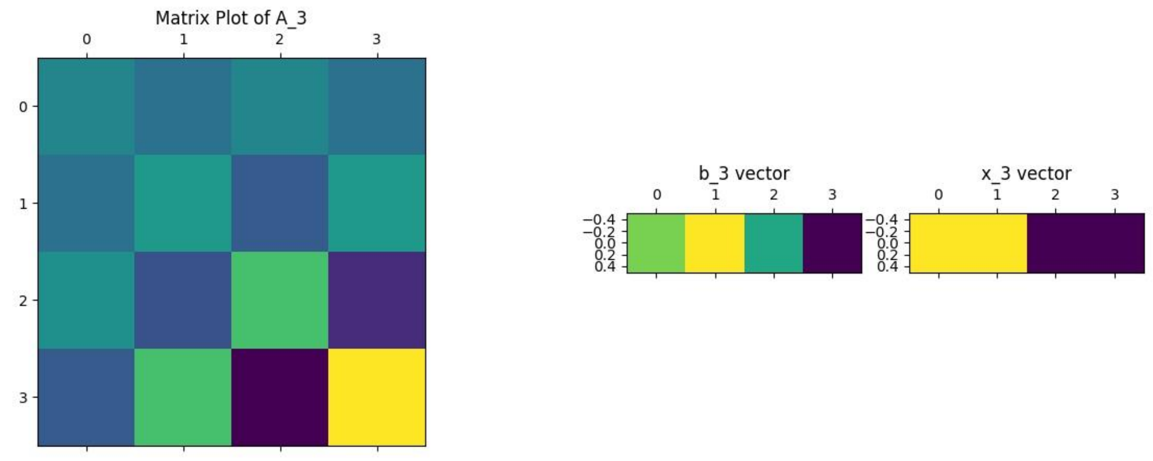
In case 3, we calculated $x_3 = \begin{pmatrix} 0 \\ 1 \\ -1 \\ 0 \end{pmatrix}$ (with and without partial pivoting)

In case 4, we discovered $x_4 = \begin{pmatrix} 0 \\ 1 \\ 2 \\ -3 \end{pmatrix}$ (with and without partial pivoting)

For special case 5, where we solve for a full matrix of solutions, we get $X = \begin{pmatrix} 1.5 & -3 & 0 \\ 0 & 6 & 1 \\ -2 & 0 & 2 \\ 1 & -4 & -3 \end{pmatrix}$ using partial pivoting since regular Gaussian Elimination resulted in NaN values.


Matrix Plot of A_1
b_1 vector
x_1 vector


Matrix Plot of A_2
b_2 vector
x_2 vector


Matrix Plot of A_3
b_3 vector
x_3 vector