LU Factorization of Linear Systems, Program 2

Colton Piper

February 14th, 2018

1 Executive Summary

This program is more programmed based than most programs as we will be discussing an algorithm to solve the Linear System $Ax = b$. The method we are using to do this is factoring the matrix A into the product of a unit lower diagonal L and an upper diagonal U. Then we will discuss stability of this and what we can do to make the stability higher for better and a few issues that pops up while doing this.

2 Statement of the Problem

We have a system $Ax = b$ where A is non-singular, which guarantees existence and uniqueness of a solution. Now we will implement three different routines to solve this system via an LU factorization. The first routine is with no pivoting applied to the Matrix A as we are going through the diagonals. The second routine is with partial pivoting which is just applying row switching permutations to A to increase stability and if we ever come across a 0 that we will divide by we can switch it out with a non-zero. The last big routine is complete pivoting which searches through the entire sub-matrix, created from the algorithm, for the highest magnitude element and then places it in the diagonal using both row switching permutations and column switching permutations.

We will also complete a lot of smaller routines which test our code correctness. This includes multiplying the LU product to show that we do indeed get our matrix back or at least our permuted matrix A back. Then to test our code for stability concerns we will create matrices that we know the LU factorization and also have a linear system $Ax = b$ where we already have the solution before we start solving the system.

3 Description of the Mathematics

The mathematics behind this program is not to deep as we really are just trying to solve a linear system via factoring the matrix into a product of two structured matrices that we can easily solve for the solution after. From highschool or undergraduate linear algebra we are using the algorithm known to most as Gauss-Jordan Elimination method. Now that though is an algorithm to solve the system thus we will discuss that in more detail in the description of algorithms and implementation.

Some other things to discuss here though is why partial pivoting and complete pivoting are more useful and stable then not using pivoting at all. First of all not pivoting at all can lead you to divide by zero and then your whole solution will be completely wrong. Pivoting can fix that by replacing the zero with a non-zero. Then complete pivoting puts the element of highest magnitude into the diagonal spot.

With no pivoting we can see that the system turns to $LUx = b$ and then we can solve $Ly = b$ and then $Ux = y$. With partial pivoting it is a little different, but similar all the same. We will just have $PAx = Pb$ to $LUx = \widetilde{b}$. So as you can see the b changes as we permuted it. Then we doe the same as with no pivoting. Lastly is complete pivoting which adds in another factor of column permutations. Thus we will have

$$\begin{aligned} PAQQ^{-1}x &= Pb \\ LU\widetilde{x} &= \widetilde{b} \\ Ly &= \widetilde{b} \\ U\widetilde{x} &= y \\ Q^{-1}x &= \widetilde{x} \\ Q\widetilde{x} &= x. \end{aligned}$$

As we will see we do not actually have to solve for the inverse of Q. Now as we saw in the class notes we need to apply those permutations in a certain order to b and then also to $\widetilde{x}$.

4 Description of the Algorithms and Implementation

4.1 LU Factoring Algorithm

The goal of this program is to solve the linear system $Ax = b$ by factoring A into the product of an lower unit diagonal matrix and an upper diagonal matrix. As we saw in the Description of Mathematics section we will also be using partial pivoting and complete pivoting. Now doing out the math is nice, but implementing these programs can be more difficult than meets the eye.

First the Gauss-Jordan elimination method has us look at the decomposing the matrix as an iterative process that we go through by using each diagonal element successfully to set the column below that diagonal to zero. Thus we will end up with an upper diagonal matrix. Now what we are doing is adding scalar multiples of each row to other rows to make the column under the diagonal we are talking about zero. Now the way to write this mathematically as we saw in the class notes is multiplying by the matrix $I - l_i e_i^T$ where the vector l_i is correlates to the diagonal we are in the i^{th} iteration. l_i is zero for the indices above and equal to i, then the coefficients needed to multiply the i^{th} row by to turn that row to zero. Now all of these matrices added up get you the L matrix which is a unit lower diagonal matrix (unit being the diagonal elements are all 1).

Now with each row addition we must transform the lower right hand sub-matrix. The cool part of this comes into the fact that because we know all the lower diagonal of the transformed A we are creating will be zero to the left of the diagonal we are on. Thus we can save our l_i columns into those spots as we only have a 2D array here. As we continue towards the last diagonal we will increasingly have to put more of L into the 2D-array which eventually hold all the information needed for L and U. Now remember we do not have to store 1 in the diagonal of U in that matrix as we know it will automatically be 1 for the algorithm we are implementing. Now to determine the coefficients in each l_i we will just have the $(l_i)_j = a_{ji}/a_{ii}$. There lies the problem though. What if our diagonal element in check is 0? In that case we cannot continue because we cannot divide by zero. This is where pivoting comes in handy. The code itself is not difficult to implement as we can see from below.

Algorithm 1 LU Factoring Without Pivoting

```
for int $j = 0$; $j < n - 1$; $j + +$ do
    for int $i = j + 1$; $i <= n - 1$; $i + +$ do
        $A[i][j] = A[i][j]/A[j][j]$;
    end for
    for int $k = j + 1$; $k <= n - 1$; $k + +$ do
        for int $i = j + 1$; $i <= n - 1$; $i + +$ do
            $A[i][k] = A[i][k] - A[i][j] * A[j][k]$;
        end for
    end for
end for
```

It is very simple to code this up the first outer iterates down the diagonal. Then the first inner loop fills in the column of A that equates to the i^{th} column of L. Then the next two loops one which is nested in the other computes the new sub-matrix. Then if you come across a zero in the diagonal element you can simply set all the matrix to zero and return it.

In this program we are always assuming that our matrix in question A is non-singular and thus a unique solution to the linear system $Ax = b$ does exist. But the LU factorization may not always exist for a matrix. Thus that is where pivoting strategies come in and the LU factorization always exist when we can pivot rows and columns thus placing a non-zero element in the location of the zero diagonal should it occur. Our algorithm does not only pivot if the element is zero. We implemented the code to pivot every single step. Partial pivoting we only check the column under the diagonal element to switch with that row, and complete pivoting we search the entire sub-matrix. Now our algorithm finds the biggest magnitude of the column or sub-matrix, then uses row permutation and column permutations to put those max magnitudes into the diagonal element. Adding this into the code is a simple if statement with switch the rows and columns which are just a for loop and temporary variables.

Now as we saw in the description of math we need to know what those permutations are and the order that we applied those to the matrix. Even this though we can add into the code as we need to know the columns and rows which are going to be switch. Then we can add those to the matrix as one or two more rows of A. We can see from the code above that the complexity for all three will be of order $O(n^3)$. Pivoting does not add that much complexity in the grand scheme of things as without pivoting we can occasionally not complete the problem at all. Also because when pivoting we are not actually doing much work. We are only searching through the Schur complement and then switching elements. Thus the amount of flops added to the complexity is minimal. We will add n or $2n$ storage for the permutation vectors which describe which row and column got switched on each step of the algorithm.

4.2 Solve Ax = b via LU Decomposition

The next algorithm we have is the linear system solver. That first algorithm was just factoring the a matrix into its LU or a permuted matrix into an LU product. Now the next function takes in the LU computed from the first algorithm and takes b and solves the system $Ax = b$. Remember with pivoting $A = LU$, $PA = LU$, or $PAQ = LU$. The way the code works without pivoting is a two step process first you solve $Ly = b$, then $Ux = y$. It is very simple to add for partial or complete pivoting. First before you solve the $Ly = b$ you need to find $\widetilde{b} = Pb$, then you can finish the steps. Lastly with complete pivoting after step three you need to change back to the original x as we have only solved for $\widetilde{x}$. This is a simple use of the Q vector saved from we can solve for x using $Q\widetilde{x} = x$.

Algorithm 2 Solving Ax=b Algorithm

```
for int $i = 1$; $i <= n - 1$; $i++$ do
    for int $j = 0$; $j <= i - 1$; $j++$ do
        $b[i] = b[i] - LU[i][j] * b[j]$;
    end for
end for
for int $i = n - 1$; $i >= 0$; $i--$ do
    for int $j = n - 1$; $j >= i + 1$; $j--$ do
        $b[i] = b[i] - LU[i][j] * b[j]$;
    end for
    $b[i] = b[i]/LU[i][i]$;
end for
```

That is the solver without pivoting but each pivoting scheme only adds another low multiple of linear complexity. To add partial pivoting we would need to add for loop in the front and that would be to permute b to make it $\widetilde{b}$ according to how we permuted the Schur Complement in each step. Then with complete pivoting we have what partial pivoting does and another for loop at the end to permute the $\widetilde{x}$ into x. Thus solving the system after factoring only takes quadratic time to complete. As in the LU Factoring Algorithm, the permutations do not add much because only the for loops themselves actually participate in flops. What happens inside the for loops is only reading variables and writing to variables.

4.3 LU Factor Checking

Another routine that took more thought that initially thought was checking that multiplying the $L * U$ gets you back to A or $\widetilde{A}$. As we are only storing LU in one nxn matrix we cannot simply write a generic matrix multiplication product. It will still be of order $O(n^3)$, but the two matrices are structured thus we can cut down the costs a lot. We know that the first row of the product will be the same as the upper matrix because L is a unit lower diagonal matrix.

As this is modeled for a cubic complexity algorithm we will have three nested for loops. Our outer loop will iterate through the rows of the product, then the first inner will iterate through the columns of the product while the second nested will be the sum for each element of the product of L and U. First things to notice is that the i^{th} row of the product will only need $i - 1$ additions and multiplications maximum to calculate that element of the product. That is because of the structure of L. Then next we know that on the j^{th} column of the product we will have $j - 1$ additions and multiplications. Thus we only need to multiply and add up to the max of $(i - 1, j - 1)$ for our iteration starting from 0. Then lastly for we need to notice that for $i > j$ we will need to add the last multiplication, but if $i = j$ or $i < j$ then we only need to add the element of U because it will be multiplying that element by 1 which is just the element.

Those are the the more complex codes which are not that complex. We also are coding some metrics and norms to test our code and test the error and how large of matrices we can use before we start seeing problems. Those routines are basic adding rows or columns of matrices and also finding max magnitudes of vectors. Nothing to difficult to put here, but I will be using the 1-Norm for matrices, the ∞-Norm for matrices, then also P-Norms, and ∞-Norm for vectors.

5 Description of the Experimental Design and Results

My first example of code correctness is only for no pivoting. The experiment design was like this. First we use a random generator to generate a diagonal dominant matrix of size n. Then we generate a random solution vector. Next we will find b by solving the matrix-vector product Ax. Now we use our systems to first factor $A = LU$, then solve then system $Ax = b$. Now we use norms to see how they are different. We will be find these norms

$$\frac{||PAQ-LU||}{||A||}$$
$$\frac{||x-\widetilde{x}||}{||x||}$$
$$\frac{||b-A\widetilde{x}||}{||b||}$$

The norms we will be using for vectors will be the 2-norm, 1-norm, and the ∞-norm while for matrices we will be using the 1-norm, ∞-norm, and Frobenius-norm. We can see from the table below that even when we get to large matrices of size $1000x1000$ matrix.

n	$A-1$	$A-\infty$	$A-F$	$x-1$	$x-2$	$x-\infty$	$b-1$	$b-2$	$b-\infty$
5	2e-16	1.8e-16	1.8e-16	1e-15	1.2e-15	1.1e-15	5.5e-16	7.4e-16	1e-15
10	2.9e-16	4e-16	2.8e-16	7.1e-15	7.6e-15	9.1e-15	3.1e-16	4.8e-16	7.2e-16
25	1.2e-15	1.1e-15	8.8e-16	3.9e-15	4.4e-15	6.3e-15	1.8e-15	2e-15	2.7e-15
50	4.7e-15	2.4e-15	3.2e-15	5.7e-13	6e-13	9.1e-13	1e-14	1.2e-14	1.6e-14
100	5.2e-14	2.6e-14	3.2e-14	2e-12	2.3e-12	4e-12	6.3e-14	7.7e-14	9.8e-14
150	2.1e-13	1.4e-14	1.2e-13	8.4e-12	9.2e-12	1.5e-11	3e-13	3.7e-13	5.2e-13
200	7.2e-14	2.5e-14	4.8e-14	1.1e-12	1.2e-12	2.5e-12	2.3e-13	2.6e-13	4e-13
300	5.9e-13	1.8e-13	3e-13	6e-12	6.4e-12	1.2e-11	3.1e-12	3.3e-12	3.6e-12
500	1.2e-12	5.7e-13	6.4e-13	1.8e-11	1.9e-11	3e-11	6.7e-12	7.7e-12	8.6e-12
750	1.1e-12	6.5e-13	5.2e-13	3e-10	3.2e-10	5.4e-10	4.6e-12	1e-11	3.3e-11
1000	4e-12	2e-12	1.9e-12	4.8e-11	5.2e-11	1e-10	9.8e-12	1.2e-11	1.9e-11

Table 1: This shows the relative errors of using different norms and checking different values computed while solving the system.

We can see that as n gets larger our relative errors do increase, but even with large matrices of 1000 and above we still get relatively slow growing error accumulation. Granted we made A diagonally dominant for this to hopefully happen. As making it not diagonally dominant can cause problems for solving the system and the LU factoring. Next we have different norms with partial pivoting

n	$A-1$	$A-\infty$	$A-F$	$x-1$	$x-2$	$x-\infty$	$b-1$	$b-2$	$b-\infty$
5	45	86	66	9.2e-16	7.6e-16	4.9e-16	1.4e-16	1.3e-16	1.1e-16
10	13	77	26	1.2e-15	1.4e-15	2.2e-15	4.9e-16	6.5e-16	1.1e-15
25	21	2.5e+02	56	6.9e-15	7.6e-15	1.3e-14	1.5e-15	1.8e-15	2.9e-15
50	84	1.5e+03	2.1e+02	6.5e-14	7.2e-14	1.4e-13	3.9e-15	4e-15	3.1e-15
100	6.5e+02	2.7e+03	5e+02	3.5e-14	3.8e-14	5.4e-14	1e-14	1.1e-14	8.7e-15
150	3.7e+03	1.7e+04	5.3e+03	4.6e-14	4.9e-14	8.4e-14	7.9e-15	8.3e-15	1.5e-14
200	7.5e+02	2.5e+04	2.4e+03	7.2e-13	7.9e-13	1.4e-12	3.1e-14	4.4e-14	1e-13
300	5.2e+02	3.3e+04	2.2e+03	2.2e-13	2.3e-13	3.8e-13	3.2e-14	4.2e-14	5.9e-14
500	8.6e+02	6.2e+04	3.7e+03	3.6e-13	3.9e-13	7.5e-13	3.7e-14	3.9e-14	4.3e-14
750	2.7e+03	3.6e+05	1.8e+04	7.3e-12	8e-12	1.6e-11	9.7e-14	1.2e-13	1.8e-13
1000	2.3e+03	2.9e+05	1.2e+04	6.4e-13	6.9e-13	1.5e-12	1.8e-13	2e-13	2.3e-13

Table 2: This shows the relative errors of using different norms and checking different values computed while solving the system with partial pivoting.

Now this is without diagonal dominance and we can see that we get much better errors for our relative errors. I ran out of time and noticed that my A norms are wrong which I think comes from me doing the $||PA - LU||$ norm incorrectly.

n	$A-1$	$A-\infty$	$A-F$	$x-1$	$x-2$	$x-\infty$	$b-1$	$b-2$	$b-\infty$
5	3e+02	2.8e+02	2.1e+02	7.2e-15	7.2e-15	7.2e-15	1e-14	7.1e-15	4.8e-15
10	13	31	17	1.6e-15	1.4e-15	1.4e-15	1.1e-15	1.2e-15	1.5e-15
25	17	92	31	2.8e-14	2.8e-14	3e-14	2.6e-14	2.9e-14	4.1e-14
50	1.1e+02	4.1e+02	1.1e+02	3.8e-15	4.1e-15	6.1e-15	3.7e-15	3.9e-15	4.7e-15
100	94	1.1e+03	1.7e+02	6e-15	6.6e-15	1.1e-14	6.6e-15	7e-15	7.6e-15
150	3.9e+02	6.3e+03	7.9e+02	7.8e-14	8.5e-14	1.4e-13	8.3e-14	8.5e-14	8.3e-14
200	3.6e+02	4.1e+03	5.7e+02	1.6e-14	1.7e-14	2.8e-14	1.7e-14	1.7e-14	2e-14
300	3.6e+02	9.6e+03	1e+03	3e-14	3.4e-14	7.4e-14	3.6e-14	3.4e-14	3.2e-14
500	6.4e+02	1.9e+04	1.7e+03	5.2e-14	5.6e-14	9e-14	5.3e-14	5.4e-14	4e-14
750	1e+04	1.7e+05	1.7e+04	1e-12	1.1e-12	1.9e-12	1.1e-12	1.1e-12	1.1e-12
1000	8.2e+02	7.2e+04	4.1e+03	4.2e-13	4.6e-13	8e-13	4.7e-13	4.7e-13	4.3e-13

Table 3: This shows the relative errors of using different norms and checking different values computed while solving the system with complete pivoting.

This is with complete pivoting and we can see that the errors for the solution and b are of the same order as partial pivoting. Unfortunately the same problem occurs as with partial pivoting and checking to see if my LU product is on the same order as the original A. It is on the same order I assume because the solution which I generated and the computed solution with complete pivoting are the same. Unfortunately I ran out of time to figure out what was wrong with the permutations in $||PAQ - LU||$. But in general pivoting pretty much guarantees that the solution will be very very very close to the actual solution.

6 Conclusions

We can see from this report that pivoting can make solving the system and factoring much more conditional. When A is diagonal dominant no-pivoting does well. But pivoting does well for any non-singular matrix A. With double precision solving a linear system via the LU decomposition with or without pivoting takes on the order of $O(n^3)$ flops to compute the solution. But using at least partial pivoting is good for conditioning reasons and does add much complexity to the problem. If A is not diagonally dominant you should use pivoting because solving a system can be ill-conditioed in many cases if you randomly generate a matrix.