SOLVING SYSTEMS OF LINEAR EQUATIONS WITH LU FACTORIZATION

June 7, 2021

Contents

1 Introduction

The goal of LU factorization (or decomposition) is the solving of square linear systems of equations, computing the inverse of a matrix, and getting the determinant, as the determinant of a triangular matrix can be calculated by obtaining the product of its diagonal entries. One of the main interests of LU factorization in numerical computation is the fact that we can easily solve for X without factoring the matrix again, and this becomes extremely computationally advantageous in larger, more complex systems where innumerable matrices with immense dimensions could be the order of the day. In LU factorization, a matrix is factored into two triangular matrices, one a lower triangle and one and upper triangle, hence the name LU.

2 LU Factorization

2.1 Basics

LU factorization was introduced in 1938 by the Polish mathematician Tadeusz Banachiewicz, although Alan Turing is often credited with having found it himself. It is essentially a modified version of Gaussian elimination, where rows are sequentially multiplied or divided by scalars and added to each other until we arrive at a reduced row echelon form of the matrix that we can use to solve the linear system.

We take a square matrix A and factor (or decompose) it into two triangular matrices L and U, which have A as their product. If we know that AX = B, where B is a vector, and we know that LU = A, then (LU)X = B. If we declare that UX = Y, we can solve for Y in LY = B, and then follow that with solving for X in UX = Y.

On its own as a solution for a single linear system, it is not necessarily very interesting. If Gaussian elimination is possible, then it is already enough to solve the system by itself, and solving AX = B will be much faster than solving (LU)X = B. However, imagine that we have 10,000 different B vectors what we would want to solve for AX = B. Once L and U are found, the computation for (LU)X = B is quite fast, and solving for 10,000 different B vectors for a given matrix A would end up being around 10,000 times faster using LU factorization if Gaussian elimination were performed to find the solution every time.

2.2 Gaussian Elimination

Imagine we have the following matrix A :

$$\begin{bmatrix} -3 & 2 & -1 \\ 6 & -6 & 7 \\ 3 & -4 & 4 \end{bmatrix}$$

The first step in LU factorization is Gaussian elimination in order to achieve reduced row echelon form and two factor triangle matrices. We start by choosing a pivot, and this will be the uppermost and leftmost value in the matrix, the -3 in the first row and first column. To implement our row operation strategy, we need to perform a multiplication or division operation on the pivot so that when we add the pivot row to the next row, the value below the pivot becomes 0. So, in this case we see that multiplying the pivot by -2 will get us what we are looking for through subtraction, or that multiplying the pivot row by 2 will get us what we want through addition. Let us use subtraction in this example, so we'll multiply by -2.

That gives us a matrix that looks like this :

$$\begin{bmatrix} -3 & 2 & -1 \\ 6 & -6 & 7 \\ 3 & -4 & 4 \end{bmatrix}$$

And now after the subtraction :

$$\begin{bmatrix} -3 & 2 & -1 \\ 0 & -2 & 5 \\ 3 & -4 & 4 \end{bmatrix}$$

We then perform the same operation for the next row below the pivot, in this case row 3. Let us take the 3 and divide it by the pivot to get -1. We multiply the first row by 0.5 and subtract that from the third row and thus every value below the pivot will be a 9, which is the goal of Gaussian elimination.

This is our result :

$$\begin{bmatrix} -3 & 2 & -1 \\ 0 & -2 & 5 \\ 0 & -2 & 3 \end{bmatrix}$$

Ok, now that the first pivot is taken care of, we move onto the next, which is the value in the second row, second column position : -2. We perform the same operation as we did in with the first pivot, except focusing entirely on the second row. In order to make our successful subtraction, we can see that as the values are the same, we only need to multiply by 1 and subtract. This gives us the following :

U :

$$\begin{bmatrix} -3 & 3 & -1 \\ 0 & -2 & 5 \\ 0 & 0 & -2 \end{bmatrix}$$

And voila! We have U, our upper triangle matrix.

It is a very simple matter to now obtain our L. Because we have been factoring our matrix A, our lower triangle matrix L will actually be a simple identity matrix with the multipliers we used earlier in the exact spaces we found them:

L :

$$\begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1 & 1 & 1 \end{bmatrix}$$

2.3 Forward Substitution in LY = B

So now that we have L and U, we can easily solve AX = B. Because LU = A, we know that AX = B is the same as (LU)X = B. Imagine we have the following vector B :

B :

$$\begin{bmatrix} -1 \\ -7 \\ -6 \end{bmatrix}$$

Taking our matrix L, we can use forward substitution to solve for Y in LY = B. With only one element in the first row of L, and that value always being 1, we know that the first element in the vector B is equal to the first element of Y. We use forward substitution to solve for Y by progressively substituting the values as we descend rows, as we would in any typical resolution of a system of linear equations with multiple unknowns.

In this example, we find that Y is equal to :

Y :

$$\begin{bmatrix} -1 \\ -9 \\ 2 \end{bmatrix}$$

2.4 Backward Substitution in UX = Y

Taking our matrix U, we perform a similar series of row operations as we did with our matrix L, but this time we begin at the bottom and use backward substitution, as the bottom row is the row with only one non-zero value. We must first perform a division (unless the value is 1), however, because the diagonal in the U matrix is not an identity matrix.

In this example, the third row, third column value of -2 is equal to 2 in the third row of the vector Y. Essentially, this means that dividing 2 by -2 from the cofactor matrix gives us a value of -1. We then go up row by row performing substitutions along the way until we get our X :

X :

$$\begin{bmatrix} 2 \\ 2 \\ -1 \end{bmatrix}$$

We can now solve for any X using using any vector B we like, simply by performing the row operations using our matrices L and U, thereby circumventing the need to perform Gaussian elimination anymore for the same cofactor matrix.

2.5 Partial Pivoting

Because in Gaussian elimination we are looking for the linear relationship between a pivot and the values below it, this can sometimes bring about a division by zero. One method of preventing this is called partial pivoting. Essentially, when choosing the pivot, we find the largest value in the current column and put its row at the row of our current pivot. This ensures a lower probability of any division by zero.

There is also full pivoting, but it requires many more calculations and isn't typically necessary.

There are some limitations to Gaussian elimination, and it will not work if for example the determinant of a matrix is zero. More on that later.

3 Project Code

3.1 Structures

```
struct matrfloat {
  int n;
  float matrice[100][100];
};
```

Listing 1: Cofactor Matrix

As seen above, the main structure in this program is a simple one that contains and int n representing the dimension of our square matrix, and a 100x100 float array that we fill as necessary. In addition to this we have a basic float array B.

3.2 Principal Functions

```
void gaussian_pivot(mat_t *m, int sub_matrix_dim) {

  int i, j;
  float pivot, quotient;
  int pivot_coord = m->n - sub_matrix_dim;

  pivot = m->m[pivot_coord][pivot_coord];
  // filling in U matrix
  for (i = pivot_coord; i < m->n - 1; i++) {
    if (pivot == 0)
      exit(0);
    quotient = m->m[i + 1][pivot_coord] / pivot;
    for (j = pivot_coord; j < m->n; j++) {
      m->m[i + 1][j] -= m->m[pivot_coord][j] * quotient;
    }
    // filling in L matrix
    m->m[i + 1][pivot_coord] = quotient;
  }
}
```

Listing 2: Gaussian Pivot

In the previous function, we see the principal operation in LU factorization: the Gaussian pivot. For computational purposes, I decided it was simpler to perform the operations to obtain L and U on the original matrix A. Because

the diagonal of the L matrix is the identity, we do not need to store it, and this allows L and U to be effectively stored in the same matrix. Rather than create a second matrix to fill, I directly modified the original matrix. When pivoting, it is absolutely ESSENTIAL to remember you must pivot the corresponding rows of vector B as well, and also pivot the values in matrix L. In terms of programming, pivoting the L values in addition to the U values is actually simpler being that they are in the same matrix, but it is still easy to miss and something to pay attention to.

```
1  void LY_equals_B(mat_t *m, float *B) {
2
3    int i, j;
4
5    for (i = 1; i < m->n; i++) {
6      for (j = 0; j < i; j++) {
7        B[i] -= m->m[i][j] * B[j];
8      }
9    }
10 }
```

Listing 3: LY equals B

In the function above, we see the first operations after we've finished the Gaussian elimination process. In the same way that I perform the row operations directly on the original matrix, I also perform the substitution operations on the original vector. The result is that this function will change our B vector into our Y vector.

```
void UX_equals_Y(mat_t *m, float *B) {

  int i, j;

  for (i = m->n - 1; i >= 0; i--) {
    for (j = m->n - i - 2; j >= 0; j--) {
      B[i] -= m->m[i][m->n - j - 1] * B[m->n - j - 1];
    }
    B[i] /= m->m[i][i];
  }
}
```

Listing 4: UX equals Y

In the function above, we see the solving of UX = Y. The main difference with LY = B is that in addition to subtracting a product from our vector (Y in this case), we are also performing a division when we have the final cofactor remaining. Once again, as the operations are performed directly on the vector, the result is that this function will change our Y vector into our X vector, thus solving for X in AX = B.

4 Notes on Solving Linear Systems with LU Factorization

4.1 Partial Pivoting

My original version of this program had no partial pivoting. Once I realized that division by a pivot whose value was 0 could easily become a problem, I at first decided to come up with my own solution. That only lasted about five minutes, because I quickly realized people much smarted than me may have already spent lifetimes researching ways of optimizing this algorithm, and that is when I discovered the partial pivot method. I suppose my own idea to find any non-zero value in the column to use as a pivot could have worked just as well on average as finding the largest value pivot, because what we really need is at least one value that is not zero.

I wrote an algorithm that would check a matrix before running Gaussian elimination to see if and where the first division by zero occurred, as I thought this would be an interesting way to gain a little efficiency over the course of many calculations by not performing any partial pivots until it was necessary. This, however, proved futile as the algorithm ended up encountering divisions by zero later in the process. Performing partial pivoting from the beginning avoided this problem; I tested it on matrices from 2x2 to 100x100 and never encountered a division by zero. It is still possible to encounter a division by zero using partial pivoting, but it seems the algorithm used to fill the matrix does not create matrices between 2x2 and 100x100 that fall in this category.

Of course, the name "partial pivoting" obviously implies there is a "full pivoting" or "whole pivoting", but I was so deep in my work that I didn't stop to consider that fact. Indeed, a mere two days before the due date of the project, I realized that it could have been an interesting path to go down from the beginning, but I would not have the time to explore it at all, due to issues with the determinant. Speaking of which...

4.2 Determinants: Or How I Learned to Stop Worrying and Love Recursivity

Once I was confident I had an algorithm that ran smoothly, I set my sights on learning a little more about the process of LU factorization and the conditions under which it did work, but more importantly, the conditions under which it did ***not*** work. I found many interesting things, but the simplest and largest unavoidable fact was that if the determinant of a matrix is 0, Gaussian elimination cannot be performed. Well, no problem, right? I've calculated determinants by hand many times, though typically on 3x3 or 4x4 matrices.

As we all know from experience, something easily done by hand can be extremely tedious to code, but something tedious to perform by hand can be vastly simplified through code. Writing the function to find the determinant, I quickly realized that it would, in fact, need to be recursive. What was I to do? I only had a couple days left and still needed to clean up my code and write a report: in fact, the very report you are reading at this very moment. Would I have enough time to iteratively construct and test my function? Would the memory management for recursive operations on progressively smaller matrices spiral out of control? What about a 100x100 matrix? Surely my professor wasn't expecting this level of extra coding for this project, as finding the determinant of a matrix (granted, through different means for comparison) was even a different project on its own.

Then it hit me: why bother finding the determinant if I can just look for the characteristics of a matrix whose determinant is 0?

4.3 If You Can't Make It, Fake It

So what are the characteristics of a matrix whose determinant is 0? First of all, they have a name! They are called singular matrices. And, they may have some of the following traits :

- Row or column of all zeros
- Duplicate rows or duplicate columns
- Linear dependency found between rows or between columns

Hey, great! Writing a function to check if a row or column has all zeros is a piece of cake. The same for duplicate rows and columns, with some creative for loops to avoid redundant checks. Easy! And now we get to linear dependence... actually, it's not quite so simple. What are signs of linear dependence?

- Every value in one row is multiplied by the same scalar when compared to another row; the same goes for columns
- Two rows of all zeros except in the same column are also linearly dependent; the same for columns in the same row

And signs that indicate linear ***in***dependence?

- Not possible to multiply a row or column by a scalar to obtain another existing row or column
- zeros at different indices in the rows or columns being compared

Well, that shouldn't be too hard, except for maybe the ugly possibility once again of division by zero...

And that is where my careful construction and emphasis on avoiding iterative redundancy went out the window. What's the project management triangle, again? "Good, fast, cheap. Choose two." I am certainly not being compensated for this work, but I suppose when under a hard deadline, time and effort do become a cost as they take from anything else you may need to do. So, while I feel the end product of the function that checks for matrix singularity is a little messy, I do believe it works and that will have to be enough for me for the time being.

5 Conclusion

All in all, I enjoyed this project. I was in the mood to get back to some of my math or physics roots as it had been years, and I am happy I did. If I had had another week or so, or even just a couple days, I would have liked to code a recursive function to find the determinant of a matrix. I also would have liked to code a function for full pivoting, even though it turns out not to be necessary given the matrices being created. Had I known in the beginning that I would try to code an iterative function looking for matrix singularity, I would have have structured some of my functions a little differently, especially to avoid redundant for loops as much as possible.

Feel free to de-comment the interactive portion of the program found in main(), though commenting out the current block of tests and solution example could be useful in that case.