CS 513 Assignment 4

Ruochen Lin

April 5, 2018

1

1.1

In the $k+1$th step of Gaussian elimination for a $m \times m$ square matrix, the first k columns and rows are untouched; thus we are effectively performing the step on a $(n-k) \times (n-k)$ square matrix. If we call this square matrix B, and the resulting $(n-k-1) \times (n-k-1)$ matrix C, then the entries of C is given by the following equation:

$$\begin{aligned} C_{i,j} &= B_{i+1,j+1} - \frac{B_{i+1,1}}{B_{1,1}} B_{1,j+1} \\ \Leftrightarrow C_{j,i} &= B_{j+1,i+1} - \frac{B_{j+1,1}}{B_{1,1}} B_{1,i+1}. \end{aligned}$$

If B is symmetric, then we have $B_{i+1,j+1} = B_{j+1,i+1}$, $B_{i+1,1} = B_{1,i+1}$, and $B_{1,j+1} = B_{j+1,1}$, which leads to $C_{i,j} = C_{j,i}$. In other words, if we start with a symmetric matrix at the beginning of the step, then the resulting matrix will also be symmetric. Note that we do start with a symmetric matrix, A; thus the lower right sqaure matrices after each step of Gaussian elimination will all be symmetric.

1.2

We can modify the algorithm of the LU-factorization for symmetric matrices, utilizing the theorem in the preceeding part: in the kth step of the Gaussian elimination, we can only update the entries in the upper triangular part of the remaining $(m-k-1) \times (m-k-1)$ block at bottom right. The pseudocode of such algorithm is the following:

Algorithm 1 LU-factorization for a symmetric matrix A

```
L = eye(m)
U = A
for i = 1 : m - 1 do
  for j = i + 1 : m do
    L(j,i) = U(i,j) / U(i,i)
    U(j,i) = 0
    for k = j : m do
      U(j,k) = U(j,k) - L(j,i) * U(i,k)
    end for
  end for
end for
```

1.3

It's easy to see that the most executed part of the algorithm is the inner most loop with the dummy index k. In lieu of the analysis in class, if we only count multiplications, then each execution of the inner most loop contributes exactly 1 operation. The total number of multiplications, $N(m)$,

can be estimated as the following:

$$\begin{aligned}
N(m) &= \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} \sum_{k=j}^{m} 1 \\
&= \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} (m-j+1) \\
&= \sum_{i=1}^{m-1} \Big[1+2+\cdots+(m-i)\Big] \\
&= \sum_{i=1}^{m-1} \frac{(m-i)(m-i+1)}{2} \\
&= \sum_{i=1}^{m-1} \Big[\frac{(m-i)^2}{2} + O(m)\Big] \\
&= \frac{1}{2}\Big[1^2+2^2+\cdots+(m-1)^2\Big] + O(m^2) \\
&= \frac{1}{2} \cdot \frac{(m-1)m(2m-1)}{6} + O(m^2) \\
&= \frac{m^3}{6} + O(m^2).
\end{aligned}$$

Thus, our current algorithm is almost twice as fast as ordinary Gaussian elimination.

1.4

To verify our analysis in the preceeding part, we have created random symmetric matrices with dimensions ranging from 10 to 100, and see how much time is needed by the two algorithms to LU-factorize the matrix. At each dimension, the time reading is averaged over 100 runs. From Fig 1, we can see that the original algorithm does take around twice of the time required by our algorithm, corroborating our analysis. In addition, the $O(m^3)$ cost can be qualitatively verified from the fact that, when the dimension is doubled, say from 50 to 100, the time elapsed in the two algorithms increaded by factors of 7.13 and 6.25 in the two alforithms, respectively, which are close to $2^3 = 8$.

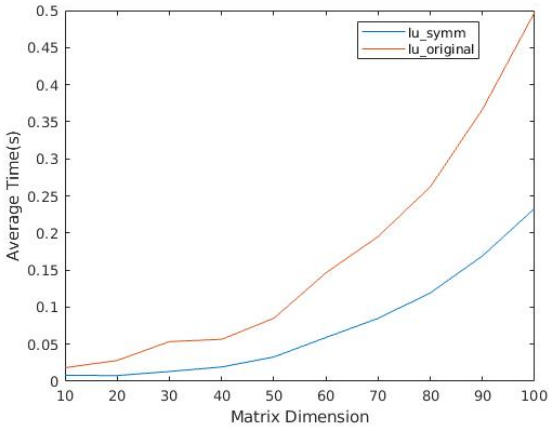

Figure 1: Plot of the time needed by LU-factorization considering symmetry and the original LU-factorization algorithms to decompose random symmetric matrices.

2

In addition to the two given matrices (labeled A_1 and A_2), I have devised the two following matrices:

$$A_{ij} = \begin{cases} 1 & i < j \\ 10^{-8} & i = j \\ 0 & i > j \end{cases},$$

labeled as A_3 and

$$A_{ij} = \begin{cases} 1 & i \neq j \\ 10^{-12} & i = j \end{cases},$$

labeled as A_4. These four cases, together with a random square matrix, are the five examples I want to show.

For each of the matrices, I have tested the three algorithms on orders 3 to 10, with scripts like the one attached. To better compare the values that can differ wildly in orders of magnitude, the outputs are in the logarithm with base 10. In addition, to compare the significance of the errors on x, we've also output $\log_{10}\|x\|_2$ in each case.

2.1

For the first matrix, ($A_{ij} = \frac{1}{i+j-1}$), we can see that the condition number explodes almost exponentially with increasing dimensions, and, as a result, the problem becomes more ill-conditioned. We can see the pivotted algorithms performs slightly better than the unpivotted one, but all of them produces exponentially increasing errors. This example shows that none of the three algorithms are stable when the problem becomes ill-conditioned enough.

2.2

Like the first example, the problem becomes more ill-conditioned as dimension increases, and all three algorithm become unstable when the matrix is ill-conditioned.

2.3

For the third matrix, it is ill-conditioned even at small dimensions; but from dimensions 4 to 10 the condition number remains more or less constant at the order of 10^{33}. However, the errors in the three algorithms still increased very fast as the dimension increased; this tells us that when the condition numbers are similar, the size of the matrix can also affect the stability of algorithms.

2.4

Despite having small elements in diagonal entries, the fourth matrix is actually well-conditioned. As what we would expect, the pivoted algorithms circumvented the small elements to produce small errors; however, for the non-pivoted algorithm, putting the small elements on the denominator makes the algorithm much less stable, compared to the pivoted ones.

2.5

For random matrices, the condition number increased slightly as we increase the dimensionality; However, they remain very well-conditioned. In this case, all three algorithms are pretty stable, despite the pivotted algorithms provide errors about one order of magnitude smaller than that generated by non-pivotted one. This means that under usual circumstances, when the problem is well-conditioned, all three algorithm should work pretty fine.

2.6 Discussions

In these experiments, I have learned that when the condition of the problem is good, usually all three algorithms are stable, but there are cases in which the unpivotted algorithm can be unstable. When the problem itself is ill-conditioned, none of the three algorithms is stable. In addition, from the analysis in class, we know that partial pivotting has comparable cost to the non-pivoted algorithm; from the observations above, it exhibits similar performance to the fully pivotted algorithm. Thus, the partial pivotted algorithm should be the standard procedure if we want to do Gaussian elimination.