Gauss-Jordan Elimination

Julian Gallo

September 2010

Overview

This is code which will take in as input a matrix and the solution vector to the equation Mx=b (where M is the matrix and b is the solution vector), solve for x, and give the determinant of the matrix M. This is accoplished by use of Gauss-Jordan elimination. Further, the user has the option of chosing whether the elimination is done with or without pivoting.

The general idea of the code is fairly simple. For elimination without pivoting, the program will start at the first element of the matrix and then look at each element below it in the same column. Then, each row below the first is added by some multiple of the first row such that the first column will become all zeroes under the first element of the matrix. The program then moves to the second element of the second row and does the same thing as before so that everything in that column below the second row would be zero. This continues for each diagonal element until the matrix is in upper triangular form. At this point, calculating the determinant is as trivial as multiplying the diagonal elements. The x vector is then found by first setting the bottom element of the vector equal to the last element of the matrix, then moving up row by row and finding the elements of the x vector one by one.

Elimination with pivoting is slightly more complicated. First of all, before the elements below a given diagonal are reduced to zero as in the non-pivoting case, the program scans every element of the column containing the relevant diagonal and finds the largest in that column. The row containing that largest element is then swapped with the row containing the currently relevant diagonal. The elements in the column below the diagonal are then reduced to zero as before. At the next diagonal, the process is repeated. Calculating the x vector and the determinant here is the same as with the non-pivoting case with one exception. The determinant switches sign with each row swap. Therefore, the number of row swaps needs to be kept track of and if there were an odd number of swaps the determinant needs to be multiplied by -1.

As the row operations are performed on the matrix, the same operations must also be performed on the solution vector so that the equation will remain unchanged. If one were to add 10 to one side of a simple equation only, the equation would not be the same. This principle holds also for matrix equations.

Complexity

One should note that the estimates given in this section are the estimates for if the program were to run with pivoting and then again without pivoting before exiting.

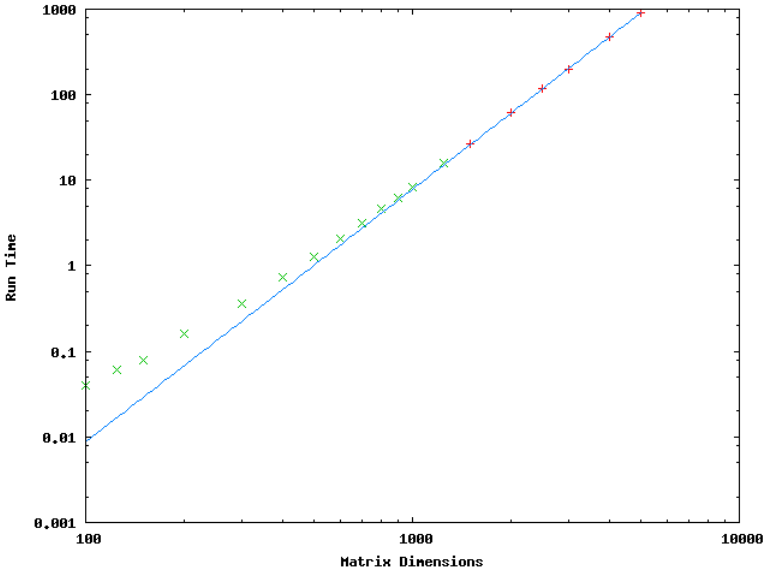
The best case scenarion in terms of complexity is when the input matrix is already in upper triangular form. In this case, only the bare minuimum number of operations needed to run the code are performed. For this particular code, the minimum number of operations performed is $n^3 + 12n^2 + 8n + c$ where n is the dimension of the matrix and c is a constant number of operations that pertains to calling functions, setting and initializing values, and calling if statements for the interface checks.

The worst case scenario is that in which every possible operation needs to be done at every step. For instance, in the worst case scenario, a row swap would need to happen each time before reducing the elements below the diagonal in a given column. For this case, the number of operations is $3n^3 + 23n^2 + 10n + c$.

To show this graphically, the program was given matrices of varying dimension (between n=100 and n=5000) and the amount of time taken to complete was timed. Since there is an n^3 term in the number of operations for both the pivoting and non-pivoting case, one would expect that the graph would go as n^3. Figure(1) is a plot of the run time of the program versus the matrix dimension. The fitted line is $x^{2.95157}$.

The trend falls away from the fit line as the dimension of the input matrix decreases. This is due to the fact that at low matrix dimension, the constant number of operations required to run the code - which is independent of any valid input - is a larger percentage of the total operations and is enough to cause run time to diverge from n^3.

Figure 1: This graph is a plot of the run time of the program versus the matrix dimension. The fit line is $x^{2.95157}$



Accuracy

Gauss-Jordan elimination without pivoting is not quite as reliable as Gauss-Jordan elimination with pivoting. However, the difference in accuracy is not terribly dramatic. I tested my program with pivoting and non pivoting for "standard" matrices with increasing dimension size and for matrices whose diagonal elements are much smaller than the off diagonal ones.

I first tested a matrix whose off-diagonal elements were much larger than the diagonal elements and noted the difference in the calculated determinant. I found that for a matrix whose off-diagonal elements were up to 8 orders of magnitude higer, the two methods agreed to at least 9 digits. For an order of magnitude difference between 9 and 15 they agreed to only 3 digits, and for an order of magnitude difference of 16, the difference was a factor of 1.5 for the particular test matrix used.

$$\begin{pmatrix} .0000000000000002 & 9 & 1 \\ 5 & .0000000000000001 & 7 \\ 7 & 2 & .0000000000000004 \end{pmatrix} \tag{1}$$

At this point, non-pivoting is no longer viable.

For matrices with very large dimensions, I tested matricies with dimensions of 10x10, 20x20, 50x50, 100x100, and 1000x1000 and noted the difference in the calculated determinants. The determinants calculated from each method differed for the 10x10 and 20x20 matrix by 1e-12, by 1e-11 for a 50x50, 1e-9 for a 100x100, and 1e-8 for a 1000x1000.

The theory is that the pivoting algorithm is a more reliable one. Thus, the conclusion here is that the non-pivoting algorithm fails more easily for matrices that are "Irregular" - ie. matrices with diagonal elements much smaller than the off-diagonal elements - than it does for very large matrices.