Numerical Computation of Inverse Matrices:

Analyzing Performance

Danny Philantrope

Queensborough Community College

Computer Science 203

Dr. Kwang-Hyun Kim

May 14, 2015

Numerical Computation of Inverse Matrices:

Analyzing Performance

Solving linear systems is an important method for many fields such as computer science, finance and engineering. Some applications include building bridges, designing aircraft, game mapping, and analysis of investments. In linear algebra, the concept of an inverse matrix is an n-by-n matrix that reduces a particular n-by-n matrix to its identity matrix when multiplied to it. An inverse matrix may be useful for solving linear equations because of the need for quotients of matrices in equations. The purpose of this report is to feature Gauss-Jordan elimination and LU decomposition (two different approaches of calculating a matrix's inverse), to implement the two into a program using C++, and to analyze the performance of the two methods when computing large sets of random matrices.

**Inversion Methods**

The two selected methods are effective because on a computer they can handle all size n-by-n matrices, and they involve minimal operations. Methods such as the Adjugate Matrix method involve increasingly higher operation numbers for large size n-by-n matrices compared to those of our methods.

Gauss-Jordan elimination involves performing operations on an augmented matrix of a system to produce the augmented matrix of an equivalent system. In terms of computing the inverse of a matrix, this system includes an n-by-n matrix and an n-by-n identity matrix. The original matrix will be reduced to an identity matrix, and as a result the identity will also be reduced – but to the inverse of the original matrix. The row operations involve interchanges between two rows, multiplying elements of a row by a nonzero constant, and subtracting a nonzero constant of a row from another row. Because a set of operations occur to complete rows of the augmented matrix for each element when traversing columns and rows, the complexity is O(n3).

The LU method for computing inverses involves using a matrix's lower triangular factor in an equation with its inverse and the identity matrix. This equation *l* x *l-1* = *I* is solved for *l-1* using forward substitution. The LU method then solves the inverse of the original matrix *A-1* by back substitution with the equation *u* x *A-1* = *l-1* where *u* is the upper triangular factor. This particular algorithm, just like the Gauss-Jordan elimination algorithm, also consists of O(n3) operations.

Our team hypothesized that the LU method would record faster computation times for large n-by-n matrices than Gauss-Jordan. Our speculation was that the row operations were more extensive than forward and back substitution. We also hypothesized that we would have errors in computation even though the algorithms were correct. Some errors that may be involved are rounding errors, LU decomposition not existing for matrices, and matrices being singular (matrix inverses not existing).

**Implementation**

The program's structure is a square matrix class that stores a matrix in a two-dimensional array. The class also holds member functions for computing a matrix inverse by Gauss-Jordan and computing by LU decomposition. The Gauss-Jordan function was improved to use scaled-pivoting. Matrices are generated with normal distribution template.

**Results and Analysis**

The overall performance of the two functions compared with each other was surprisingly similar. Tests were conducted in sets of 100 for sizes ranging from 100–1000 n-by-n square matrices. The records of each set suggested a slightly faster performance of Gauss-Jordan compared to LU, but the two were still consistent in performance. The complexity of the algorithms are O(n3) similarly, so as the matrix sizes get remarkably large, the significance of the additional operations of the LU algorithm is smaller. It is reasonable that Gauss-Jordan was actually faster because it only runs a set of operations on two matrices while traversing one, whereas the LU method runs operations on three separate matrices; and that is why the performance in our tests turns out to be roughly the same.

The next effort was recording the computational errors of the trials. For each set of trials, a separate count was taken for rounding errors, LU decomposition DNE1 and inverse matrix DNE cases. These were checked by taking the product of the inverse matrix and the original matrix and testing whether or not it matched the identity matrix, and for LU: taking the product of the lower triangular and upper triangular matrices and testing whether or not the product matched the original matrix. For 100-by-100 matrices, the results showed that the program had an error on about 3 percent of runs; this error increased to about 15 percent for 500-by-500 matrices suggesting that larger size matrices have an increase in likelihood to encounter errors.

Round off error is when the program rounds a number inaccurately because of insufficient capacity to express that number. This happens when dividing large numbers by extremely small numbers or when dividing small numbers by extremely large numbers. When one of the two occurs, the amount of digits necessary to express the number does not fit inside the scope of digits provided by a computer's memory. This problem is solved by a method called scaled-pivoting, which controls the numbers in a matrix from getting too big or too small. Scaled-pivoting checks each element in a row for maximum magnitude and then swaps the row which contains the greatest scaled numbers into the row of the pivot, where the number dividing takes places. For testing purposes and for improving the program we needed to update it with scaled-pivoting.

The case of an LU decomposition DNE may happen depending on the decomposition method implemented by a program's author. This program decomposes a matrix into LU using Gaussian elimination. Because the process does not involve row swaps: if there is a 0 as a pivot at any point of the process, the LU would not exist as a result. The case of a matrix inverse DNE, also called a singular matrix, occurs when a matrix determinant is 0. This is a result when a matrix has an entire row or column of 0's, when an entire row is the same number, or when a row is a constant multiple of another row, among other conditions. The chance of a matrix being singular is very unlikely with large n-by-n matrices, and this program minimizes the chance by generating matrices with random numbers. The chance of an LU decomposition DNE is also unlikely with large matrices, but a permutation matrix can be implemented with the LU method to correct this computation error – this is something that our team is currently researching.

**Current Improvements**

The program has been updated with scaled-pivoting which has lowered the overall occurrence of errors to lower than one percent for 100-by-100 matrices. Our team is currently implementing the permutation matrix using LU method, which is needed in this program. We are always trying to find ways to decrease the number of operations and improve the program's performance. Our next goal is to explore professional projects of linear systems, and compare them with this program for optimization purposes.

References

http://pages.pacificcoast.net/~cazelais/251/gauss-jordan.pdf

https://d1b10bmlvqabco.cloudfront.net/attach/i5cz0ovsxzq146/i94z89al5ygcu/i95lyy3pvy86sj/psc2\_1.pdf

http://mathforcollege.com/nm/simulations/nbm/04sle/nbm\_sle\_sim\_inversecomptime.pdf

https://www.math.ust.hk/~mamu/courses/231/Slides/CH06\_2A.pdf