Solving General Systems of Linear Equations with Gaussian Elimination

The following is a brief discussion of Gaussian elimination for solving a general system of n linear equations in n unknowns. This includes two sets of algorithms that can be implemented in the programming language of your choice. This is phrased in general terms and focuses on the coefficient matrix and right-hand side vector, rather than on the equations themselves, as one would do in actually solving a large system of equations either by hand or on a computer.

Suppose we have a linear system $Ax = b$, where A is an $n \times n$ matrix with ijth entry a_{ij} and x and b are n-vectors with respective ith components x_i and b_i. The first set of algorithms given below is based on "naive" Gaussian elimination. This version of Gaussian elimination is "'naive" because it breaks down if a zero pivot entry is encountered at any step of the elimination, even though the system might be perfectly solvable without breakdown if the equations and/or unknowns had been given in a different order. In practice, pivot entries that are exactly zero are rarely encountered; thus "naive" Gaussian elimination rarely suffers complete breakdown. However, it is not unusual for the algorithm to encounter relatively small pivot elements, and when this happens it becomes *unstable*.[1] In the second set of algorithms, Gaussian elimination is augmented with *pivoting*, specifically *partial pivoting*, which avoids these problems of breakdown and stability if the system is nonsingular.

These sets of algorithms are structured as most modern software library routines are. In each set, the first algorithm performs Gaussian elimination on A, *overwriting*[2] each entry in the lower triangular part of A with the "multiplier" used in that step of the elimination. The second algorithm performs the same operations on b that were performed on A. The third algorithm produces the solution using back substitution. As written here, this algorithm overwrites b with the solution to save storage; one can, of course, write the solution into a separate vector x if desired.

In library software, the first algorithm is usually coded in a routine separate from the other two. This is because it requires $O(n^3)$ arithmetic operations to execute, whereas the other two require only $O(n^2)$ arithmetic operations; thus, in actual applications, it usually requires by far most of the computation. Coding it in a separate routine allows one to re-use the output in applications that involve a number of systems with the same coefficient matrix but different right-hand sides, thereby saving significant

[1] *Stability* is an essential property of practically effective algorithms and is loosely defined as follows: An algorithm is *stable* if its execution does not contribute "unreasonable" error to the computed solution; otherwise, it is *unstable*. How much error is "unreasonable" is, of course, subjective; however, it is usually quite clear when an algorithm is unstable.

[2] This means that the current value of, say, the entry a_{ij} is replaced by that of the multiplier $-a_{ik}/a_{kk}$. In the algorithms, overwriting is denoted by "$\leftarrow$"; in an actual program, one would simply use "$=$" and write, e.g., $a_{ij} = -a_{ik}/a_{kk}$, which has the same effect.