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 \( ij \)th entry \( a_{ij} \) and \( x \) and \( b \) are \( n \)-vectors with respective \( i \)th 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 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 “\(\leftarrow\)” and write, e.g., \(a_{ij} = -a_{ik}/a_{kk}\), which has the same effect.
In practice, in order to maintain stability, it is important to avoid not only zero pivot entries but also pivot entries that are relatively small. There are two basic pivoting strategies that accomplish this: partial and complete pivoting. In partial pivoting, at the $k$th elimination step, one determines the entry of greatest magnitude among the bottom $n - k + 1$ entries in the $k$th matrix column; then that entry is brought to the pivot position by swapping rows as necessary. In complete pivoting, one determines the entry of greatest magnitude in the whole lower-right $(n - k + 1) \times (n - k + 1)$ block of entries; then that entry is brought to the pivot position by swapping rows and columns as necessary. Theoretically, complete pivoting is always stable, whereas partial pivoting is known to be unstable in certain examples. However, these examples are highly contrived, and partial pivoting can be trusted to perform as stably as complete pivoting in practice. Since partial pivoting is significantly cheaper than complete pivoting, it is virtually always used in practice.

The second set of algorithms is based on Gaussian elimination with partial pivoting. Note that the Gaussian elimination algorithm stops with $\max_{k \leq i \leq n} |a_{ik}| = 0$ for some $k < n$ if and only if $A$ is singular; in particular, it does not break down and stably computes the solution if $A$ is nonsingular. Also, note that the back-substitution algorithm is the same as before.
Gaussian Elimination with Partial Pivoting on $A$:

For $k = 1, \ldots, n - 1$

If $\max_{k \leq i \leq n} |a_{ik}| = 0$, stop; else find $i_k \geq k$ such that $|a_{i_kk}| = \max_{k \leq i \leq n} |a_{ik}|$.

If $i_k > k$, interchange $a_{kj} \leftrightarrow a_{ik}$ for $j = k, \ldots, n$.

For $i = k + 1, \ldots, n$

$a_{ik} \leftarrow -a_{ik}/a_{kk}$

For $j = k + 1, \ldots, n$

$a_{ij} \leftarrow a_{ij} + a_{ik}a_{kj}$

Row Operations on $b$:

For $k = 1, \ldots, n - 1$

If $i_k > k$, interchange $b_k \leftrightarrow b_{i_k}$.

For $i = k + 1, \ldots, n$

$b_i \leftarrow b_i + a_{ik}b_k$

Back Substitution:

$b_n \leftarrow b_n/a_{nn}$

For $i = n - 1, \ldots, 1$

$b_i \leftarrow \left(b_i - \sum_{j=i+1}^{n} a_{ij}b_j\right)/a_{ii}$