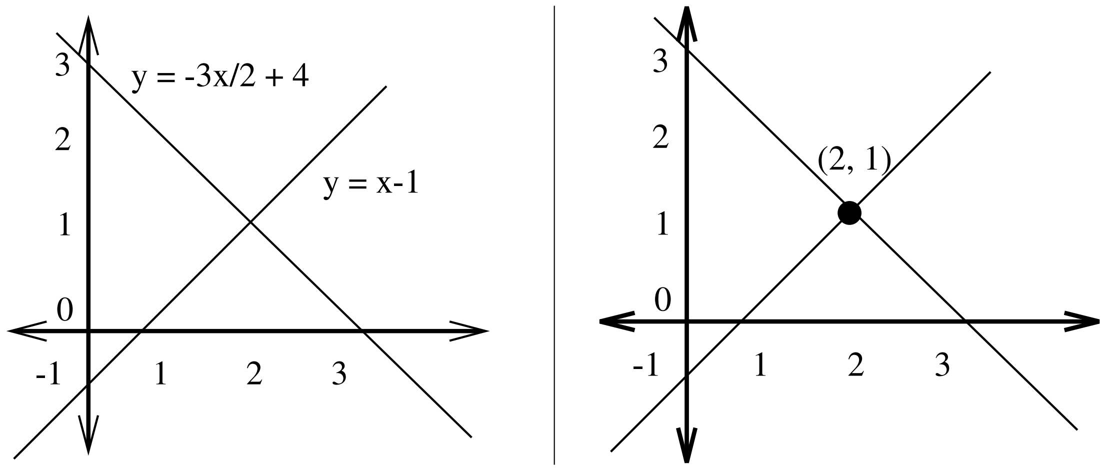

INPUT OUTPUT

13.1 Solving Linear Equations

Input description: An $m \times n$ matrix A and an $m \times 1$ vector b, together representing m linear equations on n variables.

Problem description: What is the vector x such that $A \cdot x = b$?

Discussion: The need to solve linear systems arises in an estimated 75% of all scientific computing problems [DB74]. For example, applying Kirchhoff's laws to analyze electrical circuits generates a system of equations—the solution of which puts currents through each branch of the circuit. Analysis of the forces acting on a mechanical truss generates a similar set of equations. Even finding the point of intersection between two or more lines reduces to solving a small linear system.

Not all systems of equations have solutions. Consider the equations $2x + 3y = 5$ and $2x + 3y = 6$. Some systems of equations have multiple solutions, such as $2x + 3y = 5$ and $4x + 6y = 10$. Such *degenerate* systems of equations are called *singular*, and they can be recognized by testing whether the determinant of the coefficient matrix is zero.

Solving linear systems is a problem of such scientific and commercial importance that excellent codes are readily available. There is no good reason to implement your own solver, even though the basic algorithm (Gaussian elimination) is one you learned in high school. This is especially true when working with large systems.

Gaussian elimination is based on the observation that the solution to a system of linear equations is invariant under scaling (if $x = y$, then $2x = 2y$) and adding equations (the solution to $x = y$ and $w = z$ is the same as the solution to $x = y$ and $x + w = y + z$). Gaussian elimination scales and adds equations to eliminate each variable from all but one equation, leaving the system in such a state that the solution can be read off from the equations.

The time complexity of Gaussian elimination on an $n \times n$ system of equations is $O(n^3)$, since to clear the ith variable we add a scaled copy of the n-term ith row to each of the $n - 1$ other equations. On this problem, however, constants matter. Algorithms that only partially reduce the coefficient matrix and then backsubstitute to get the answer use 50% fewer floating-point operations than the naive algorithm.

Issues to worry about include:

- *Are roundoff errors and numerical stability affecting my solution?* – Gaussian elimination would be quite straightforward to implement except for roundoff errors. These accumulate with each row operation and can quickly wreak havoc on the solution, particularly with matrices that are *almost* singular.

  To eliminate the danger of numerical errors, it pays to substitute the solution back into each of the original equations and test how close they are to the desired value. *Iterative methods* for solving linear systems refine initial solutions to obtain more accurate answers. Good linear systems packages will include such routines.

  The key to minimizing roundoff errors in Gaussian elimination is selecting the right equations and variables to pivot on, and to scale the equations to eliminate large coefficients. This is an art as much as a science, which is why you should use a well-crafted library routine as described next.

- *Which routine in the library should I use?* – Selecting the right code is also somewhat of an art. If you are taking your advice from this book, start with the general linear system solvers. Hopefully they will suffice for your needs. But search through the manual for more efficient procedures solving special types of linear systems. If your matrix happens to be one of these special types, the solution time can reduce from cubic to quadratic or even linear.

- *Is my system sparse?* – The key to recognizing that you have a special-case linear system is establishing how many matrix elements you really need to describe A. If there are only a few non-zero elements, your matrix is *sparse* and you are in luck. If these few non-zero elements are clustered near the diagonal, your matrix is *banded* and you are in even more luck. Algorithms for reducing the bandwidth of a matrix are discussed in Section 13.2. Many other regular patterns of sparse matrices can also be exploited, so consult the manual of your solver or a better book on numerical analysis for details.

- *Will I be solving many systems using the same coefficient matrix?* – In applications such as least-squares curve fitting, we must solve $A \cdot x = b$ repeatedly with different b vectors. We can preprocess A to make this easier. The lower-upper or *LU-decomposition* of A creates lower- and upper-triangular matrices

L and U such that $L{\cdot}U = A$. We can use this decomposition to solve $A{\cdot}x = b$, since

$$A \cdot x = (L \cdot U) \cdot x = L \cdot (U \cdot x) = b$$

This is efficient since backsubstitution solves a triangular system of equations in quadratic time. Solving $L \cdot y = b$ and then $U \cdot x = y$ gives the solution x using two $O(n^2)$ steps instead of one $O(n^3)$ step, after the LU-decomposition has been found in $O(n^3)$ time.

The problem of solving linear systems is equivalent to that of matrix inversion, since $Ax = B \leftrightarrow A^{-1}Ax = A^{-1}B$, where $I = A^{-1}A$ is the identity matrix. Avoid it however since matrix inversion proves to be three times slower than Gaussian elimination. LU-decomposition proves useful in inverting matrices as well as computing determinants (see Section 13.4 (page 404)).

Implementations: The library of choice for solving linear systems is apparently LAPACK—a descendant of LINPACK [DMBS79]. Both of these Fortran codes, as well as many others, are available from Netlib. See Section 19.1.5 (page 659).

Variants of LAPACK exist for other languages, like CLAPACK (C) and LAPACK++ (C++). The *Template Numerical Toolkit* is an interface to such routines in C++, and is available at *http://math.nist.gov/tnt/*.

JScience provides an extensive linear algebra package (including determinants) as part of its comprehensive scientific computing library. *JAMA* is another matrix package written in Java. Links to both and many related libraries are available at *http://math.nist.gov/javanumerics/*.

Numerical Recipes [PFTV07] (*www.nr.com*) provides guidance and routines for solving linear systems. Lack of confidence in dealing with numerical procedures is the most compelling reason to use these ahead of the free codes.

Notes: Golub and van Loan [GL96] is the standard reference on algorithms for linear systems. Good expositions on algorithms for Gaussian elimination and LU-decomposition include [CLRS01] and a host of numerical analysis texts [BT92, CK07, SK00]. Data structures for linear systems are surveyed in [PT05].

Parallel algorithms for linear systems are discussed in [Gal90, KSV97, Ort88]. Solving linear systems is one of most important applications where parallel architectures are used widely in practice.

Matrix inversion and (hence) linear systems solving can be done in matrix multiplication time using Strassen's algorithm plus a reduction. Good expositions on the equivalence of these problems include [AHU74, CLRS01].

Related Problems: Matrix multiplication (see page 401), determinant/permanent (see page 404).