Chapter 2

Linear Equations

The *linear equation* is the most elementary problem that arises in computational economic analysis. In a linear equation, an $n \times n$ matrix A and an n-vector b are given, and one must compute the n-vector x that satisfies

$$A \star x = b.$$

Linear equations arise, directly or indirectly, in most computational economic applications. For example, a linear equation may be solved when computing the steady-state distribution of a discrete-state stochastic economic process or when computing the equilibrium prices and quantities of a multicommodity market model with linear demand and supply functions. Linear equations also arise as elementary tasks in solution procedures designed to solve more complicated nonlinear economic models. For example, a nonlinear partial equilibrium market model may be solved using Newton's method, which involves solving a sequence of linear equations. And the Euler functional equation of a rational expectations model may be solved using a collocation method, which yields a nonlinear equation that in turn is solved as a sequence of linear equations.

Various practical issues arise when solving a linear equation numerically. Digital computers are capable of representing arbitrary real numbers with only limited precision. Numerical arithmetic operations, such as computer addition and multiplication, produce rounding errors that may, or may not, be negligible. Unless the rounding errors are controlled in some way, the errors can accumulate, rendering a computed solution that is far from correct. Speed and storage requirements are also important considerations in the design of a linear equation solution algorithm. In some applications, such as the stochastic simulation of a rational expectations model, linear equations may have to be solved millions of times. And in other applications, such as computing option prices using finite difference methods, linear equations of extremely high order n may be encountered.

Over the years, numerical analysts have studied linear equations extensively and have developed algorithms for solving them quickly, accurately, and with a minimum of computer storage. In most applied work, one can typically rely on Gaussian elimination, which may be implemented in various different forms depending on the structure of the linear equation. Iterative methods offer an alternative to Gaussian elimination and are especially efficient if the A matrix is large and consists mostly of zero entries.

2.1 L-U Factorization

Some linear equations $A \star x = b$ are relatively easy to solve. For example, if A is a lower triangular matrix,

$$A = \begin{bmatrix} a_{11} & 0 & 0 & \dots & 0 \\ a_{21} & a_{22} & 0 & \dots & 0 \\ a_{31} & a_{32} & a_{33} & \dots & 0 \\ & & & & \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix},$$

then the elements of x can be computed recursively using *forward-substitution*:

$$\begin{aligned} x_1 &= b_1/a_{11} \\ x_2 &= (b_2 - a_{21}x_1)/a_{22} \\ x_3 &= (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33} \\ &\vdots \\ x_n &= (b_n - a_{n1}x_1 - a_{n2}x_2 - \dots - a_{nn-1}x_{n-1})/a_{nn} \end{aligned}$$

If A is an upper triangular matrix, then the elements of x can be computed recursively using *backward-substitution*.

Most linear equations encountered in practice, however, do not have a triangular A matrix. In such cases, the linear equation is often best solved using the *L-U factorization algorithm*. The L-U algorithm is designed to decompose the A matrix into the product of lower and upper triangular matrices, allowing the linear equation to be solved using a combination of backward and forward substitution.

The L-U algorithm involves two phases. In the *factorization* phase, Gaussian elimination is used to factor the matrix A into the product

$$A = L \star U$$

of a row-permuted lower triangular matrix L and an upper triangular matrix U. A row-permuted lower triangular matrix is simply a lower triangular matrix that has had its rows rearranged. Any nonsingular square matrix can be decomposed in this way.

In the *solution* phase of the L-U algorithm, the factored linear equation

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

is solved by first solving

$$L \star y = b$$

for y using forward substitution, accounting for row permutations, and then solving

$$U \star x = y$$

for x using backward substitution.

Consider, for example, the linear equation $A \star x = b$ where

$$A = \begin{bmatrix} -3 & 2 & 3 \\ -3 & 2 & 1 \\ 3 & 0 & 0 \end{bmatrix} \text{ and } b = \begin{bmatrix} 10 \\ 8 \\ -3 \end{bmatrix}.$$

The matrix A can be decomposed into the product $A = L \star U$ where

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ -1 & 1 & 0 \end{bmatrix} \text{ and } U = \begin{bmatrix} -3 & 2 & 3 \\ 0 & 2 & 3 \\ 0 & 0 & -2 \end{bmatrix}.$$

The matrix L is row-permuted lower triangular because upon interchanging the second and third rows, a lower diagonal matrix results. The matrix U is upper triangular. Solving $L * y = b$ for y using forward substitution involves first solving for y_1, then for y_3, and finally for y_2. Given the solution $y = (10, 7, -2)'$, the linear equation $U \star x = y$ can the be solved using backward substitution, yielding the solution of the original linear equation, $x = (-1, 2, 1)$.

The L-U factorization algorithm is faster than the linear equation solution methods that are typically presented in elementary linear algebra courses. For large n, it takes approximately $n^3/3 + n^2$ long operations (multiplications and divisions) to solve an $n \times n$ linear equation using L-U factorization. Explicitly computing the inverse of A and then computing $A^{-1} \star b$ requires approximately $n^3 + n^2$ long operations. Solving the linear equation using Cramer's rule requires approximately $(n+1)!$ long operations. To solve a 10×10 linear equation, for example, L-U factorization requires exactly 430 long operations, matrix inversion and multiplication requires exactly 1100 long operations, and Cramer's rule requires nearly 40 million long operations.

Linear equations arise so frequently in numerical analysis that most numerical subroutine packages and software programs include either a basic subroutine or an intrinsic function for solving a linear equation using L-U factorization. In Matlab, the solution to the linear equation $A \star x = b$ is returned by the statement $x = A \setminus b$. The '\', or "backslash", operator is designed to solve the linear equation using L-U factorization, unless a special structure for A is detected, in which case Matlab may use another, more efficient method. In particular, if Matlab detects that A is triangular or permuted triangular, it will dispense with L-U factorization and solve the linear equation directly using forward or backward substitution. Matlab also uses special algorithms when the A matrix is positive definite or triangular.

In many numerical applications in economics, one encounters a situation in which a series of linear equations must be solved, all having the same A matrix, but different b vectors, $b_1, b_2, \ldots, b_m$. When this situation arises, it is often computationally more efficient to directly compute and store the inverse of A first and then compute the solutions $x = A^{-1} \star b_j$ by performing only simple matrix-vector multiplications. Whether explicitly computing the inverse is faster than L-U factorization depends on the size of the linear equation system n and the number of times m an equation system is to be solved. Computing $x = A \setminus b_j$ a total of m times involves $\frac{mn^3}{3} + mn^2$ long operations. Computing A^{-1} once and then computing $A^{-1} \star b_j$ a total of m times requires $n^3 + mn^2$ long operations. Thus explicit computation of the inverse should be faster than L-U factorization whenever the number of equations to be solved m is greater than three or four. The actual breakeven point will vary across numerical analysis packages, depending on the computational idiosyncrasies and overhead costs of the L-U factorization and inverse routines implemented in the package.

An alternative to the matrix inversion strategy is to compute and store the L-U factors of A first, and then repeatedly apply backward and forward substitution by computing $x = L\backslash(U\backslash b_j)$. In Matlab, the L-U factors of A are returned by the statement $[L, U] = lu(A)$. This approach requires $\frac{n^3}{3} + mn^2$ long operations, fewer than the matrix inversion strategy. Both strategies involve n^2 operations per equation after the factorization or inversion of A. In Matlab, however, computing $x = L\backslash(U\backslash b)$ can often be more expensive than computing $A^{-1} \star b$, despite the same operation count, because the operator '\' expends effort to determine that L and U are triangular. Moreover, the L-U approach requires two additional matrices to be stored, L and U, whereas the matrix inversion strategy requires only one additional matrix to be stored, A^{-1}. For these reasons, matrix inversion is preferred over single L-U factorization strategy in Matlab.

2.2 Gaussian Elimination

The L-U factors of a matrix A are computed using *Gaussian elimination.* Gaussian elimination is based on two elementary row operations: subtracting a constant multiple of one row of a linear equation from another row, and interchanging two rows of a linear equation. Either operation may be performed on a linear equation without altering its solution.

The Gaussian elimination algorithm begins with the matrices L and U initialized as $L = I$ and $U = A$, where I is the identity matrix. The algorithm then uses elementary row operations to transform U into an upper triangular matrix, while preserving the permuted lower diagonality of L and the factorization $A = L \star U$:

Consider the matrix

$$A = \begin{bmatrix} 2 & 0 & -1 & 2 \\ 4 & 2 & -1 & 4 \\ 2 & -2 & -2 & 3 \\ -2 & 2 & 7 & -3 \end{bmatrix}.$$

The first stage of Gaussian elimination is designed to nullify the subdiagonal entries of the first column of the U matrix. The U matrix is updated by subtracting 2 times first row from the second, subtracting 1 times the first row from the third, and subtracting -1 times the first row from the fourth.

The L matrix, which initially equals the identity, is updated by storing the multipliers 2, 1, and -1 as the subdiagonal entries of its first column. These operations yield updated L and U matrices:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ -1 & 0 & 0 & 1 \end{bmatrix} \qquad U = \begin{bmatrix} 2 & 0 & -1 & 2 \\ 0 & 2 & 1 & 0 \\ 0 & -2 & -1 & 1 \\ 0 & 2 & 6 & -1 \end{bmatrix}.$$

After the first stage of Gaussian elimination, $A = L \star U$ and L is lower triangular, but U still is not upper triangular.

The second stage Gaussian elimination is designed to nullify the subdiagonal entries of the second column of the U matrix. The U matrix is updated by subtracting -1 times second row from the third and subtracting 1 times the second row from the fourth. The L matrix is updated by storing the multipliers -1 and 1 as the subdiagonal elements of its second column. These operations yield updated L and U matrices:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ -1 & -1 & 0 & 1 \end{bmatrix} \qquad U = \begin{bmatrix} 2 & 0 & -1 & 2 \\ 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 5 & -1 \end{bmatrix}.$$

After the second stage of Gaussian elimination, $A = L \star U$ and L is lower triangular, but U still is not upper triangular.

In the third stage of Gaussian elimination, one encounters an apparent problem. The third diagonal element of the matrix U is zero, making it impossible to nullify the subdiagonal entry as before. This difficulty is easily remedied, however, by interchanging the third and fourth rows of U. The L matrix is updated by interchanging the previously computed multipliers residing in the third and fourth rows. These operations yield updated L and U matrices:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ -1 & -1 & 1 & 0 \end{bmatrix} \qquad U = \begin{bmatrix} 2 & 0 & -1 & 2 \\ 0 & 2 & 1 & 0 \\ 0 & 0 & 5 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The Gaussian elimination algorithm terminates with a permuted lower triangular matrix L and an upper triangular matrix U whose product is the matrix A. In theory, Gaussian elimination will compute the L-U factors of any matrix A, provided A is invertible. If A is not invertible, Gaussian elimination will detect this by encountering a zero diagonal element in the U matrix that cannot be replaced with a nonzero element below it.

2.3 Rounding Error and Pivoting

In practice, Gaussian elimination performed on a computer can sometimes render inaccurate solutions due to rounding errors. The effects of rounding errors, however, can often be controlled by *pivoting*.

Consider the linear equation

$$\begin{bmatrix} -L^{-1} & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

where L is a large positive number.

To solve this equation via Gaussian elimination, a single row operation is required: subtracting $-L$ times the first row from the second row. In principle, this operation yields the L-U factorization

$$\begin{bmatrix} -L^{-1} & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -L & 1 \end{bmatrix} \begin{bmatrix} -L^{-1} & 1 \\ 0 & L+1 \end{bmatrix}.$$

In theory, applying forward and backward substitution yields the solution $x_1 = L/(L+1)$ and $x_2 = (L+2)/(L+1)$, which are both very nearly one.

In practice, however, Gaussian elimination may yield a very different result. In performing Gaussian elimination, one encounters an operation that cannot be carried out precisely on a computer, and which should be avoided in computational work: adding or subtracting values of vastly different magnitudes. On a computer, it is not meaningful to add or subtract two values whose magnitude differ by more than the number of significant digits that the computer can represent. If one attempts such an operation, the smaller value is effectively treated as zero. For example, the sum of 0.1 and 0.0001 may be 0.1001, but on a hypothetical machine with three digit precision the result of the sum is rounded to 0.1 before it is stored.

In the linear equation above, adding 1 or 2 to a sufficiently large L on a computer simply returns the value L. Thus, in the first step of the backward substitution, x_2 is computed, not as $(L+2)/(L+1)$, but rather as L/L, which is exactly one. Then, in the second step of backward substitution,

$x_1 = -L(1 - x_2)$ is computed to be zero. Rounding error thus produces computed solution for x_1 that has a relative error of nearly 100 percent.

Fortunately, there is a partial remedy for the effects of rounding error in Gaussian elimination. Rounding error arises in the example above because the diagonal element $-L^{-1}$ is very small. Interchanging the two rows at the outset of Gaussian elimination does not alter the theoretical solution to the linear equation, but allows one to perform Gaussian elimination with a diagonal element of larger magnitude.

Consider the equivalent linear equation system after the rows have been interchanged:

$$\begin{bmatrix} 1 & 1 \\ -L^{-1} & 1 \end{bmatrix} \begin{bmatrix} x_2 \\ x_1 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}.$$

After interchanging the rows, the new A matrix may be factored as

$$\begin{bmatrix} 1 & 1 \\ -L^{-1} & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -L^{-1} & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & L^{-1}+1 \end{bmatrix}.$$

Backward and forward substitution yield the theoretical results $x_2 = 1 - L^{-1}$ and $x_1 = L^{-1} + 1 + L^{-1}(1 - L^{-1})$. In evaluating these expressions on the computer, one again encounters rounding error. Here, x_2 is numerically computed to be exactly one as before. However, x_1 is also computed to be exactly one. The computed solution, though not exactly correct, is correct to the precision available on the computer, and is certainly more accurate than the one obtained without interchanging the rows.

Interchanging rows during Gaussian elimination in order to make the magnitude of diagonal element as large as possible is called pivoting. Pivoting substantially enhances the reliability and the accuracy of a Gaussian elimination routine. For this reason, all good Gaussian elimination routines, including the ones implemented in Matlab, employ some form of pivoting.

2.4 Ill Conditioning

Pivoting cannot cure all the problems caused by rounding error. Some linear equations are inherently difficult to solve accurately on a computer, despite pivoting. This occurs when the A matrix is structured in such a way that a small perturbation δb in the data vector b induces a large change δx in the solution vector x. In such cases the linear equation or, more generally, the A matrix are said to be *ill-conditioned*.

One measure of ill-conditioning in a linear equation $A \star x = b$ is the "elasticity" of the solution vector x with respect to the data vector b

$$\epsilon = \sup_{||\delta b||>0} \frac{||\delta x||/||x||}{||\delta b||/||b||}.$$

The elasticity gives the maximum percentage change in the size of the solution vector x induced by a one percent change the size of the data vector b. If the elasticity is large, then small errors in the computer representation of the data vector b can produce large errors in the computed solution vector x. Equivalently, the computed solution x will have far fewer significant digits than the data vector b.

The elasticity of the solution is expensive to compute and thus is virtually never computed in practice. In practice, the elasticity is estimated using the condition number of the matrix A, which for invertible A is defined by

$$\kappa \equiv ||A|| \cdot ||A^{-1}||.$$

The condition number of A is an upper bound of the elasticity of the solution. The bound is tight in that for some data vector b, the condition number equals the elasticity. The condition number is always greater than or equal to one. Numerical analysts often use the rule of thumb that for each power of 10 in the condition number, one significant digit is lost in the computed solution vector x. Thus, if A has a condition number of 1000, the computed solution vector x will have about three fewer significant digits than the data vector b.

Consider the linear equation $A \star x = b$ where $A_{ij} = i^{n-j}$ and $b_i = (i^n - 1)/(i - 1)$. In theory, the solution x to this linear equation is a vector containing all ones for any n. In practice, however, if one solves the linear equation numerically using Matlab's '\' operator one can get quite different results. Below is a table that gives the supremum norm approximation error in the computed value of x and the condition number of the A matrix for different n:

n	approx error	condition number
1	0.0e+000	1.0e+000
2	0.0e+000	6.9e+000
3	8.9e-016	7.1e+001
4	6.7e-016	1.2e+003
5	2.5e-013	2.6e+004
10	5.2e-007	2.1e+012
15	1.1e+002	2.6e+021
20	9.6e+010	1.8e+031
25	8.2e+019	4.2e+040

In this example, the computed answers are accurate to seven decimal up to $n = 10$. The accuracy, however, deteriorates rapidly after that. In this example, the matrix A is a member of the a class of notoriously ill-conditioned matrices called the Vandermonde matrices, which can arise in applied numerical work if one is not careful.

Ill-conditioning ultimately can be ascribed to the limited precision of computer arithmetic. The effects of ill-conditioning can often be mitigated by performing computer arithmetic using the highest precision available on the computer. The best way to handle ill-conditioning, however, is to avoid it altogether. This is often possible when the linear equation problem is as an elementary task in a more complicated solution procedure, such as solving a nonlinear equation or approximating a function with a polynomial. In such cases one can sometimes reformulate the problem or alter the solution strategy to avoid the ill-conditioned linear equation. We will see several examples of this avoidance strategy later in the book.

2.5 Special Linear Equations

Gaussian elimination can be accelerated for A matrices possessing certain special structures. Two classes of A matrices that arise frequently in computational economic analysis and for which such an acceleration is possible are symmetric positive definite matrices and sparse matrices.

Linear equations $A \star x = b$ in which A is a symmetric positive definite arise frequently in least-squares curve-fitting and optimization applications. A special form of Gaussian elimination, the Cholesky factorization algorithm, may be applied to such linear equations. Cholesky factorization requires only half as many operations as general Gaussian elimination and has the added advantage that it is less vulnerable to rounding error and does not require pivoting.

The essential idea underlying Cholesky factorization is that any symmetric positive definite matrix A can be uniquely expressed as the product

$$A = U' \star U$$

of an upper triangular matrix U and its transpose. The matrix U is called the Cholesky factor or square root of A. Given the Cholesky factor of A, the linear equation

$$A \star x = U' \star U \star x = U' \star (U \star x) = b$$

may be solved efficiently by using forward substitution to solve

$$U' \star y = b$$

and then using backward substitution to solve

$$U \star x = y.$$

The Matlab '\' operator will automatically employ Cholesky factorization, rather than L-U factorization, to solve the linear equation if it detects that A is symmetric positive definite.

Another situation that often arises in computational practice are linear equations $A \star x = b$ in which the A matrix is sparse, that is, consists largely of zero entries. For example, in solving differential equations, one often encounters tridiagonal matrices, which are zero except on the diagonal and immediately above and below the diagonal. When the A matrix is sparse, the conventional Gaussian elimination algorithm consists largely of meaningless, but costly, operations involving either multiplication or addition with zero. The Gaussian elimination algorithm in these instances can often be dramatically increased by avoiding these useless operations.

Matlab has special routines for efficiently storing sparse matrices and operating with them. In particular, the Matlab command `S`=sparse(A) creates a version S of the matrix A stored in a sparse matrix format, in which only the nonzero elements of A and their indices are explicitly stored. Sparse matrix storage requires only a fraction of the space required to store A in standard form if A is sparse. Also, the operator '\' is designed to recognize whether a sparse matrix is involved in the operation and adapts the Gaussian elimination algorithm to exploit this property. In particular, both $x = S \setminus b$ and $x = A \setminus b$ will compute the answer to $A \star x = b$. However, the former express will be executed substantially faster by avoiding meaningless operations with zeros.

2.6 Iterative Methods

Algorithms based on Gaussian elimination are called *exact methods* because they would generate exact solutions for the linear equation $A \star x = b$ after a finite number of operations, if not for rounding error. Such methods are ideal for moderately-sized linear equations, but may be impractical for large ones. Other methods, called *iterative methods* can often be used to solve large linear equations more efficiently if the A matrix is sparse, that is, if A is composed mostly of zero entries. Iterative methods are designed to generate a sequence of increasingly better approximations to the solution of a linear equation, but generally do not yield an exact solution after a prescribed number of steps.

The most widely-used iterative methods for solving a linear equation $A \star x = b$ are developed by choosing an invertible matrix Q and writing the linear equation in the equivalent form

$$Q \star x = b + (Q - A) \star x$$

or

$$x = Q^{-1} \star b + (I - Q^{-1} \star A) \star x.$$

This form of the linear equation suggests the iteration rule

$$x^{(k+1)} \leftarrow Q^{-1} \star b + (I - Q^{-1} \star A) \star x^{(k)},$$

which, if convergent, must converge to a solution of the linear equation.

Ideally, the so-called *splitting matrix* Q will satisfy two criteria. First, $Q^{-1} \star b$ and $Q^{-1} \star A$ should be relatively easy to compute. This is true if Q is either diagonal or triangular. Second, the iterates should converge quickly to the true solution of the linear equation. The iteration rule constitutes a contraction mapping, and thus converges to the unique solution of the linear equation from any initial value, if

$$||I - Q^{-1} \star A|| < 1$$

in any matrix norm. The smaller the value of $||I - Q^{-1} \star A||$, the faster the guaranteed rate of convergence of the iterates when measured in the subordinate vector norm.

The two most popular iterative methods are the Gauss-Jacoby and Gauss-Seidel methods. The Gauss-Jacoby method lets Q be the diagonal matrix formed from the diagonal entries of A. The Gauss-Seidel method lets Q be the upper diagonal matrix formed from the upper diagonal entries of A. Using the row-sum matrix norm to test the convergence criterion, both methods are guaranteed to converge from any starting value if A is diagonally dominant, that is, if

$$|A_{ii}| > \sum_{\substack{i=1 \\ i \neq j}}^{n} |A_{ij}| \qquad \forall i.$$

Diagonally dominant matrices arise naturally in a many computational economic applications, including the solution of differential equations and the approximation of functions using cubic splines, both of which will be discussed in later sections.

The algorithms begin with a guess, typically the zero vector, for the solution x of the linear equation. Iteration continues until the norm of the change δx in the iterate falls below a specified convergence tolerance τ or until the maximum number M of allowable iterations are performed. A typical Gauss-Jacoby solution routine is given by:

```
$x = \texttt{GJACOBY}(A, b, x, n, M, \tau)$
for $k = 1 : M$
    $\delta x = (b + A \star x)\,./diag(A)$
    $x = x + \delta x$
    if $||\delta x|| < \tau$, exit
end
```

The Gauss-Seidel method is similar to the Gauss-Jacoby method, except that it updates the values of the x_i immediately after computing δx_i:

```
$x = \texttt{GSEIDEL}(A, b, x, n, M, \tau)$
for $k = 1 : M$
    for $i = 1 : n$
        $\delta x_i = \left(b_i + \sum_{j=1}^{n} A_{ij} x_j\right) / A_{ii}$
        $x_i = x_i + \delta x_i$
      end
      if $||\delta x|| < \tau$, exit
    end do
```

A general rule of thumb is that if A is large and sparse, then the linear equation is a good candidate for iterative methods. This is true, however, only if the sparse matrix storage functions are used to reduce storage requirements and computational effort. In a vector processing language such as Matlab, the Gauss-Jacoby algorithm will typically be faster than the Gauss-Seidel algorithm because the former can vectorized, that is, written compactly as a sequence of vector and matrix operations that avoid do-loops. This is not true of the Gauss-Seidel algorithm, which requires a do-loop to update individual elements of x one at a time.