Chapter 3

Solution of Linear Systems

Linear systems of equations are ubiquitous in scientific computing – they arise when solving problems in many applications, including biology, chemistry, physics, engineering and economics, and they appear in nearly every chapter of this book. A fundamental numerical problem involving linear systems is that of finding a solution (if one exists) to a set of n linear equations in n unknowns. The first digital computers (developed in the 1940's primarily for scientific computing problems) required about an hour to solve linear systems involving only 10 equations in 10 unknowns. Modern computers are substantially more powerful, and we can now solve linear systems involving thousands of equations in thousands of unknowns in a fraction of a second. Indeed, many problems in science and industry involve millions of equations in millions of unknowns. In this chapter we study the most commonly used algorithm, *Gaussian elimination with partial pivoting*, to solve these important problems.

After a brief introduction to linear systems, we discuss computational techniques for problems (diagonal, lower and upper triangular) that are simple to solve. We then describe Gaussian elimination with partial pivoting, which is an algorithm that reduces a general linear system to one that is simple to solve. Important matrix factorizations associated with Gaussian elimination are described, and issues regarding accuracy of computed solutions are discussed. The chapter ends with a section describing MATLAB implementations, as well as the main tools provided by MATLAB for solving linear systems.

3.1 Linear Systems

A linear system of order n consists of the n linear algebraic equations

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n &= b_2 \\ &\vdots \\ a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n &= b_n \end{aligned}$$

in the n unknowns $x_1, x_2, \ldots, x_n$. A solution of the linear system is a set of values $x_1, x_2, \ldots, x_n$ that satisfy all n equations simultaneously. Problems involving linear systems are typically formulated using the linear algebra language of matrices and vectors. To do this, first group together the quantities on each side of the equals sign as vectors,

$$\begin{bmatrix} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n \\ \vdots \\ a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

and recall from Section 1.2 that the vector on the left side of the equal sign can be written as a linear combination of column vectors, or equivalently as a matrix–vector product:

$$\begin{bmatrix} a_{1,1} \\ a_{2,1} \\ \vdots \\ a_{n,1} \end{bmatrix} x_1 + \begin{bmatrix} a_{1,2} \\ a_{2,2} \\ \vdots \\ a_{n,2} \end{bmatrix} x_2 + \cdots + \begin{bmatrix} a_{1,n} \\ a_{2,n} \\ \vdots \\ a_{n,n} \end{bmatrix} x_n = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Thus, in matrix–vector notation, the linear system is represented as

$$Ax = b$$

where

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} \quad \text{is the coefficient matrix of order } n,$$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \text{is the unknown, or solution vector of length } n, \text{ and}$$

$$b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \quad \text{is the right hand side vector of length } n.$$

We consider problems where the coefficients $a_{i,j}$ and the right hand side values b_i are real numbers. A **solution** of the linear system $Ax = b$ of order n is a vector x that satisfies the equation $Ax = b$. The **solution set** of a linear system is the set of all its solutions.

Example 3.1.1. The linear system of equations

$$\begin{array}{rcrcrcl} 1x_1 & + & 1x_2 & + & 1x_3 & = & 3 \\ 1x_1 & + & (-1)x_2 & + & 4x_3 & = & 4 \\ 2x_1 & + & 3x_2 & + & (-5)x_3 & = & 0 \end{array}$$

is of order $n = 3$ with unknowns x_1, x_2 and x_3. The matrix–vector form is $Ax = b$ where

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & 4 \\ 2 & 3 & -5 \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \quad b = \begin{bmatrix} 3 \\ 4 \\ 0 \end{bmatrix}$$

This linear system has precisely one solution, given by $x_1 = 1$, $x_2 = 1$ and $x_3 = 1$, so

$$x = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Linear systems arising in most realistic applications are usually much larger than the order $n = 3$ system of the previous example. However, a lot can be learned about linear systems by looking at small problems. Consider, for example, the 2×2 linear system:

$$\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \quad \Leftrightarrow \quad \begin{array}{l} a_{1,1}x_1 + a_{1,2}x_2 = b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 = b_2. \end{array}$$

If $a_{1,2} \neq 0$ and $a_{2,2} \neq 0$, then we can write these equations as

$$x_2 = -\frac{a_{1,1}}{a_{1,2}}x_1 + \frac{b_1}{a_{1,2}} \quad \text{and} \quad x_2 = -\frac{a_{2,1}}{a_{2,2}}x_1 + \frac{b_2}{a_{2,2}},$$

which are essentially the slope-intercept form equations of two lines in a plane. Solutions of this linear system consist of all values x_1 and x_2 that satisfy both equations; that is, all points (x_1, x_2) where the two lines intersect. There are three possibilities for this simple example (see Fig. 3.1):

- Unique solution – the lines intersect at only one point.
- No solution – the lines are parallel, with different intercepts.
- Infinitely many solutions – the lines are parallel with the same intercept.

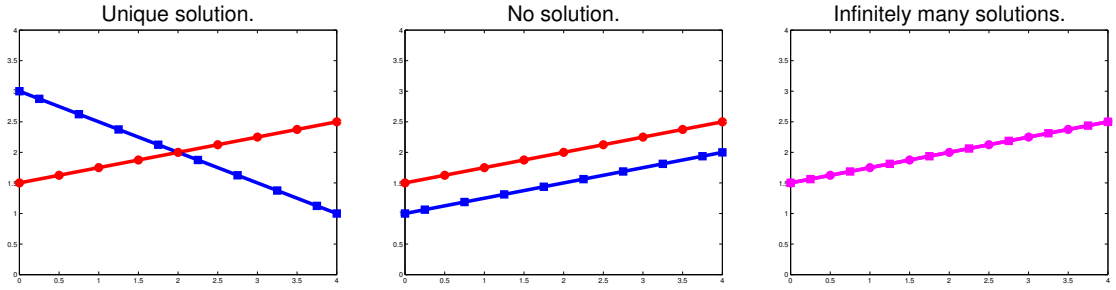

Figure 3.1: Possible solution sets for a general 2×2 linear system. The left plot shows two lines that intersect at only one point (unique solution), the middle plot shows two parallel lines that do not intersect at any points (no solution), and the right plot shows two identical lines (infinitely many solutions).

This conclusion holds for linear systems of any order; that is, **a linear system of order n has either no solution, 1 solution, or an infinite number of distinct solutions**. A linear system $Ax = b$ of order n is **nonsingular** if it has one and only one solution. A linear system is **singular** if it has either no solution or an infinite number of distinct solutions; which of these two possibilities applies depends on the relationship between the matrix A and the right hand side vector b, a matter that is considered in a first linear algebra course.

Whether a linear system $Ax = b$ is singular or nonsingular depends solely on properties of its coefficient matrix A. In particular, the linear system $Ax = b$ is nonsingular if and only if the matrix A is *invertible*; that is, if and only if there is a matrix, A^{-1}, such that $AA^{-1} = A^{-1}A = I$, where I is the identity matrix,

$$I = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{bmatrix}.$$

So, we say A is nonsingular (invertible) if the linear system $Ax = b$ is nonsingular, and A is singular (non-invertible) if the linear system $Ax = b$ is singular. It is not always easy to determine, a-priori, whether or not a matrix is singular especially in the presence of roundoff errors. This point will be addressed in Section 3.6.

Example 3.1.2. Consider the linear systems of order 2:

(a) $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \end{bmatrix} \quad \Rightarrow \quad \begin{matrix} x_2 = -\frac{1}{2}x_1 + \frac{5}{2} \\ x_2 = -\frac{3}{4}x_1 + \frac{6}{4} \end{matrix} \quad \Rightarrow \quad \begin{matrix} x_2 = -\frac{1}{2}x_1 + \frac{5}{2} \\ x_2 = -\frac{3}{4}x_1 + \frac{3}{2} \end{matrix}$

This linear system consists of two lines with unequal slopes. Thus the lines intersect at only one point, and the linear system has a unique solution.

(b) $\begin{bmatrix} 1 & 2 \\ 3 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \end{bmatrix} \Rightarrow \begin{matrix} x_2 = -\frac{1}{2}x_1 + \frac{5}{2} \\ x_2 = -\frac{3}{6}x_1 + \frac{6}{6} \end{matrix} \Rightarrow \begin{matrix} x_2 = -\frac{1}{2}x_1 + \frac{5}{2} \\ x_2 = -\frac{1}{2}x_1 + 1 \end{matrix}$

This linear system consists of two lines with equal slopes. Thus the lines are parallel. Since the intercepts are not identical, the lines do not intersect at any points, and the linear system has no solution.

(c) $\begin{bmatrix} 1 & 2 \\ 3 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 15 \end{bmatrix} \Rightarrow \begin{matrix} x_2 = -\frac{1}{2}x_1 + \frac{5}{2} \\ x_2 = -\frac{3}{6}x_1 + \frac{15}{6} \end{matrix} \Rightarrow \begin{matrix} x_2 = -\frac{1}{2}x_1 + \frac{5}{2} \\ x_2 = -\frac{1}{2}x_1 + \frac{5}{2} \end{matrix}$

This linear system consists of two lines with equal slopes. Thus the lines are parallel. Since the intercepts are also equal, the lines are identical, and the linear system has infinitely many solutions.

Problem 3.1.1. *Consider the linear system of Example 3.1.1. If the coefficient matrix A remains unchanged, then what choice of right hand side vector b would lead to the solution vector $x = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$?*

Problem 3.1.2. *Consider any linear system of equations of order 3. The solution of each equation can be portrayed as a plane in a 3-dimensional space. Describe geometrically how 3 planes can intersect in a 3-dimensional space. Why must two planes that intersect at two distinct points intersect at an infinite number of points? Explain how you would conclude that if a linear system of order 3 has 2 distinct solutions it must have an infinite number of distinct solutions.*

Problem 3.1.3. *Give an example of one equation in one unknown that has no solution. Give another example of one equation in one unknown that has precisely one solution, and another example of one equation in one unknown that has an infinite number of solutions.*

Problem 3.1.4. *The determinant of an $n \times n$ matrix, $\det(A)$, is a number that theoretically can be used computationally to indicate if a matrix is singular. Specifically, if $\det(A) \neq 0$ then A is nonsingular. The formula to compute $\det(A)$ for a general $n \times n$ matrix is complicated, but there are some special cases where it can be computed fairly easily. In the case of a 2×2 matrix,*

$$\det\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = ad - bc.$$

Compute the determinants of the 2×2 matrices in Example 3.1.2. Why do these results make sense?
Important note: The determinant is a good theoretical test for singularity, but it is not a practical test in computational problems. This is discussed in more detail in Section 3.6.

3.2 Simply Solved Linear Systems

Some linear systems are easy to solve. Consider the linear systems of order 3 displayed in Fig. 3.2. By design, the solution of each of these linear systems is $x_1 = 1$, $x_2 = 2$ and $x_3 = 3$. The structure of these linear systems makes them easy to solve; to explain this, we first name the structures exhibited.

The entries of a matrix $A = [a_{i,j}]_{i,j=1}^{n}$ are partitioned into three classes:

(a) the diagonal entries, i.e., the entries $a_{i,j}$ for which $i = j$,

(b) the strictly lower triangular entries, i.e., the entries $a_{i,j}$ for which $i > j$, and

(c) the strictly upper triangular entries, i.e., the entries $a_{i,j}$ for which $i < j$.

$$
(a)\quad \begin{array}{rcl} (-1)x_1 + 0x_2 + 0x_3 &=& -1 \\ 0x_1 + 3x_2 + 0x_3 &=& 6 \\ 0x_1 + 0x_2 + (-5)x_3 &=& -15 \end{array} \quad \Leftrightarrow \quad \begin{bmatrix} -1 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & -5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -1 \\ 6 \\ -15 \end{bmatrix}
$$

$$
(b)\quad \begin{array}{rcl} (-1)x_1 + 0x_2 + 0x_3 &=& -1 \\ 2x_1 + 3x_2 + 0x_3 &=& 8 \\ (-1)x_1 + 4x_2 + (-5)x_3 &=& -8 \end{array} \quad \Leftrightarrow \quad \begin{bmatrix} -1 & 0 & 0 \\ 2 & 3 & 0 \\ -1 & 4 & -5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -1 \\ 8 \\ -8 \end{bmatrix}
$$

$$
(c)\quad \begin{array}{rcl} (-1)x_1 + 2x_2 + (-1)x_3 &=& 0 \\ 0x_1 + 3x_2 + 6x_3 &=& 24 \\ 0x_1 + 0x_2 + (-5)x_3 &=& -15 \end{array} \quad \Leftrightarrow \quad \begin{bmatrix} -1 & 2 & -1 \\ 0 & 3 & 6 \\ 0 & 0 & -5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 24 \\ -15 \end{bmatrix}
$$

Figure 3.2: Simply Solved Linear Systems

The locations of the diagonal, strictly lower triangular, and strictly upper triangular entries of A are illustrated in Fig. 3.3. The lower triangular entries are composed of the strictly lower triangular and diagonal entries, as illustrated in Fig. 3.4. Similarly, the upper triangular entries are composed of the strictly upper triangular and diagonal entries. Using this terminology, we say that the linear system in Fig. 3.2(a) is diagonal, the linear system in Fig. 3.2(b) is lower triangular, and the linear system in Fig. 3.2(c) is upper triangular.

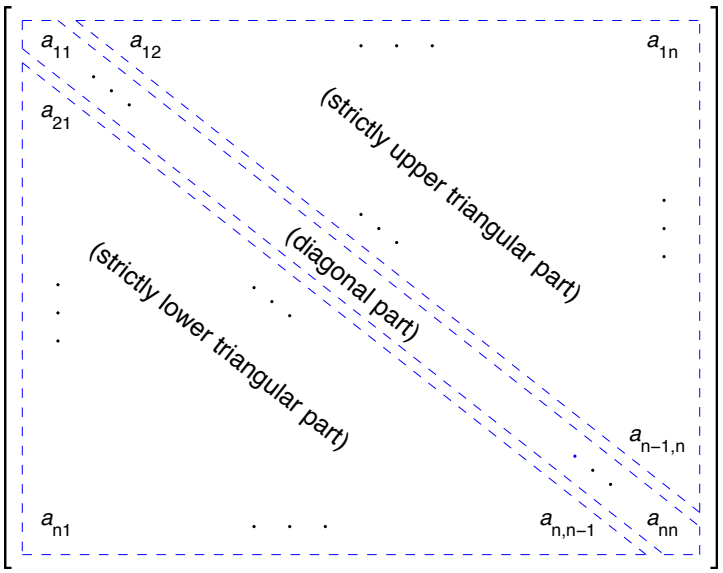

Figure 3.3: Illustration of strict triangular and diagonal matrix entries.

Problem 3.2.1. *Let A be a matrix of order n. Show that A has n^2 entries, that $n^2 - n = n(n-1)$ entries lie off the diagonal, and that each strictly triangular portion of A has $\frac{n(n-1)}{2}$ entries.*

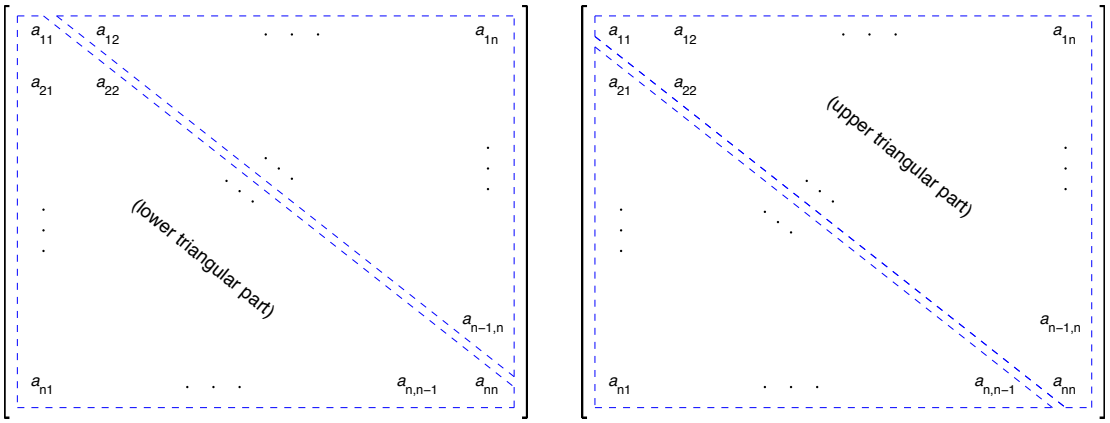

Figure 3.4: Illustration of the lower triangular and upper triangular parts of a matrix.

3.2.1 Diagonal Linear Systems

A matrix A of order n is **diagonal** if all its nonzero entries are on its diagonal. (This description of a diagonal matrix *does not state* that the entries on the diagonal are nonzero.) A **diagonal linear system** of equations of order n is one whose coefficient matrix is diagonal. Solving a diagonal linear system of order n, like that in Fig. 3.2(a), is easy because each equation determines the value of one unknown, provided that the diagonal entry is nonzero. So, the first equation determines the value of x_1, the second x_2, etc. A linear system with a diagonal coefficient matrix is singular if it contains a diagonal entry that is zero. In this case the linear system may have no solutions or it may have infinitely many solutions.

Example 3.2.1. In Fig. 3.2(a) the solution is $x_1 = \frac{-1}{(-1)} = 1$, $x_2 = \frac{6}{3} = 2$ and $x_3 = \frac{-15}{(-5)} = 3$.

Example 3.2.2. Consider the following singular diagonal matrix, A, and the vectors b and d:

$$A = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -2 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}, \quad d = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}.$$

(a) The linear system $Ax = b$ has no solution. Although we can solve the first and last equations to get $x_1 = \frac{1}{3}$ and $x_3 = 0$, it is not possible to solve the second equation:

$$0 \cdot x_1 + 0 \cdot x_2 + 0 \cdot x_3 = -1 \quad \Rightarrow \quad 0 \cdot \frac{1}{3} + 0 \cdot x_2 + 0 \cdot 0 = -1 \quad \Rightarrow \quad 0 \cdot x_2 = -1.$$

Clearly there is no value of x_2 that satisfies the equation $0 \cdot x_2 = -1$.

(b) The linear system $Ax = d$ has infinitely many solutions. From the first and last equations we obtain $x_1 = \frac{1}{3}$ and $x_3 = \frac{1}{2}$. The second equation is

$$0 \cdot x_1 + 0 \cdot x_2 + 0 \cdot x_3 = 0 \quad \Rightarrow \quad 0 \cdot \frac{1}{3} + 0 \cdot x_2 + 0 \cdot \frac{1}{2} = 0 \quad \Rightarrow \quad 0 \cdot x_2 = 0,$$

and thus x_2 can be any real number.

Problem 3.2.2. *Consider the matrix*

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Why is this matrix singular? Find a vector b so that the linear system $Ax = b$ has no solution. Find a vector b so that the linear system $Ax = b$ has infinitely many solutions.

3.2.2 Column and Row Oriented Algorithms

Normally, an algorithm that operates on a matrix must choose between a row–oriented or a column–oriented version depending on how the programming language stores matrices in memory. Typically, a computer's memory unit is designed so that the CPU can quickly access consecutive memory locations. So, consecutive entries of a matrix usually can be accessed quickly if they are stored in consecutive memory locations.

MATLAB stores matrices in column–major order; numbers in the same column of the matrix are stored in consecutive memory locations, so column–oriented algorithms generally run faster.

Other scientific programming languages behave similarly. For example, FORTRAN 77 stores matrices in column–major order, and furthermore, consecutive columns of the matrix are stored contiguously; that is, the entries in the first column are succeeded immediately by the entries in the second column, etc. Generally, Fortran 90 follows the FORTRAN 77 storage strategy where feasible, and column–oriented algorithms generally run faster. In contrast, C and C++ store matrices in row–major order; numbers in the same row are stored in consecutive memory locations, so row–oriented algorithms generally run faster. However, there is no guarantee that consecutive rows of the matrix are stored contiguously, nor even that the memory locations containing the entries of one row are placed before the memory locations containing the entries in later rows.

In the sections that follow, we present both row– and column–oriented algorithms for solving linear systems.

3.2.3 Forward Substitution for Lower Triangular Linear Systems

A matrix A of order n is **lower triangular** if all its nonzero entries are either strictly lower triangular entries or diagonal entries. A **lower triangular linear system** of order n is one whose coefficient matrix is lower triangular. Solving a lower triangular linear system, like that in Fig. 3.2(b), is usually carried out by **forward substitution**. Forward substitution determines first x_1, then x_2, and so on, until all x_i are found. For example, in Fig. 3.2(b), the first equation determines x_1. Given x_1, the second equation then determines x_2. Finally, given both x_1 and x_2, the third equation determines x_3. This process is illustrated in the following example.

Example 3.2.3. In Fig. 3.2(b) the solution is $x_1 = \frac{-1}{(-1)} = 1$, $x_2 = \frac{8-2x_1}{3} = \frac{8-2\cdot 1}{3} = 2$ and $x_3 = \frac{-8-(-1)x_1-4x_2}{(-5)} = \frac{-8-(-1)\cdot 1-4\cdot 2}{(-5)} = 3$.

To write a computer code to implement forward substitution, we must formulate the process in a systematic way. To motivate the two most "popular" approaches to implementing forward substitution, consider the following lower triangular linear system.

$$\begin{array}{lllllll} a_{1,1}x_1 & & & & & = & b_1 \\ a_{2,1}x_1 & + & a_{2,2}x_2 & & & = & b_2 \\ a_{3,1}x_1 & + & a_{3,2}x_2 & + & a_{3,3}x_3 & = & b_3 \end{array}$$

By transferring the terms involving the off-diagonal entries of A to the right hand side we obtain

$$\begin{array}{lllllll} a_{1,1}x_1 & & & = & b_1 & & \\ & a_{2,2}x_2 & & = & b_2 & - & a_{2,1}x_1 \\ & & a_{3,3}x_3 & = & b_3 & - & a_{3,1}x_1 \quad - \quad a_{3,2}x_2 \end{array}$$

The right hand sides can be divided naturally into rows and columns.

- Row–oriented forward substitution updates (modifies) the right hand side one row at a time. That is, after computing $x_1, x_2, \ldots, x_{i-1}$, we update b_i as:

$$b_i := b_i - (a_{i,1}x_1 + a_{i,2}x_2 + \cdots + a_{i,i-1}x_{i-1}) = b_i - \sum_{j=1}^{i-1} a_{i,j}x_j$$

  The symbol ":=" means assign the value computed on the right hand side to the variable on the left hand side. Here, and later, when the lower limit of the sum exceeds the upper limit the sum is considered to be "empty". In this process the variable b_i is "overwritten" with the value $b_i - \sum_{j=1}^{i-1} a_{i,j}x_j$. With this procedure to update the right hand side, an algorithm for row-oriented forward substitution could have the form:

$$\begin{array}{l} \text{for each } i = 1, 2, \ldots, n \\ \qquad \text{update } b_i := b_i - \sum_{j=1}^{i-1} a_{i,j}x_j \\ \qquad \text{compute } x_i := b_i / a_{i,i} \end{array}$$

  Notice that each update step uses elements in the ith *row* of A, $a_{i,1}, a_{i,2}, \ldots, a_{i,i-1}$.

- Column–oriented forward substitution updates (modifies) the right hand side one column at a time. That is, after computing x_j, we update $b_{j+1}, b_{j+2}, \ldots, b_n$ as:

$$\begin{aligned} b_{j+1} &:= b_{j+1} - a_{j+1,j}x_j \\ b_{j+2} &:= b_{j+2} - a_{j+2,j}x_j \\ &\vdots \\ b_n &:= b_n - a_{n,j}x_j \end{aligned}$$

  With this procedure, an algorithm for column-oriented forward substitution could have the form:

$$\begin{array}{l} \text{for each } j = 1, 2, \ldots, n \\ \qquad \text{compute } x_j := b_j / a_{j,j} \\ \qquad \text{update } b_i := b_i - a_{i,j}x_j, \quad i = j+1, j+2, \ldots, n \end{array}$$

  Notice in this case the update steps use elements in the jth *column* of A, $a_{j+1,j}, a_{j+2,j}, \ldots, a_{n,j}$.

Pseudocodes implementing row– and column–oriented forward substitution are presented in Fig. 3.5. Note that:

- We again use the symbol ":=" to assign values to variables.
- The "for" loops step in ones. So, "for $i = 1$ to n means execute the loop for each value $i = 1$, $i = 2$, until $i = n$, in turn. (Later, in Fig. 3.6 we use "downto" when we want a loop to count backwards in ones.)
- When a loop counting forward has the form, for example, "for $j = 1$ to $i - 1$" and for a given value of i we have $i - 1 < 1$ then the loop is considered *empty* and does not execute. A similar convention applies for empty loops counting backwards.
- The algorithms destroy the original entries of b. If these entries are needed for later calculations, they must be saved elsewhere. In some implementations, the entries of x are written over the corresponding entries of b.

Row–Oriented	*Column–Oriented*
Input: matrix $A = [a_{i,j}]$ vector $b = [b_i]$ Output: solution vector $x = [x_i]$ for $i = 1$ to n for $j = 1$ to $i - 1$ $b_i := b_i - a_{i,j}x_j$ next j $x_i := b_i/a_{i,i}$ next i	Input: matrix $A = [a_{i,j}]$ vector $b = [b_j]$ Output: solution vector $x = [x_j]$ for $j = 1$ to n $x_j := b_j/a_{j,j}$ for $i = j + 1$ to n $b_i := b_i - a_{i,j}x_j$ next i next j

Figure 3.5: Pseudocode for *Row– and Column–Oriented Forward Substitution. Note that the algorithm assumes that the matrix A is lower triangular.*

The forward substitution process can break down if a diagonal entry of the lower triangular matrix is zero. In this case, the lower triangular matrix is singular, and a linear system involving such a matrix may have no solution or infinitely many solutions.

Problem 3.2.3. *Why is a diagonal linear system also lower triangular?*

Problem 3.2.4. *Illustrate the operation of column–oriented forward substitution when used to solve the lower triangular linear system in Fig. 3.2(b). [Hint: Show the value of b each time it has been modified by the for-loop and the value of each entry of x as it is computed.]*

Problem 3.2.5. *Use row–oriented forward substitution to solve the linear system:*

$$\begin{aligned} 3x_1 + \quad 0x_2 + 0x_3 + \quad 0x_4 &= \quad 6 \\ 2x_1 + (-3)x_2 + 0x_3 + \quad 0x_4 &= \quad 7 \\ 1x_1 + \quad 0x_2 + 5x_3 + \quad 0x_4 &= -8 \\ 0x_1 + \quad 2x_2 + 4x_3 + (-3)x_4 &= -3 \end{aligned}$$

Problem 3.2.6. *Repeat Problem 3.2.5 but using the column–oriented version.*

Problem 3.2.7. *Modify the row– and the column–oriented pseudocodes in Fig. 3.5 so that the solution x is written over the right hand side b.*

Problem 3.2.8. *Consider a general lower triangular linear system of order n. Show that row–oriented forward substitution costs* $\frac{n(n-1)}{2}$ *multiplications,* $\frac{n(n-1)}{2}$ *subtractions, and n divisions. [Hint:* $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$*.]*

Problem 3.2.9. *Repeat Problem 3.2.8 for the column–oriented version of forward substitution.*

Problem 3.2.10. *Develop pseudocodes, analogous to those in Fig. 3.5, for row–oriented and column–oriented methods of solving the following linear system of order 3*

$$\begin{aligned} a_{1,1}x_1 \quad + \quad a_{1,2}x_2 \quad + \quad a_{1,3}x_3 \quad &= \quad b_1 \\ a_{2,1}x_1 \quad + \quad a_{2,2}x_2 \quad &= \quad b_2 \\ a_{3,1}x_1 \quad &= \quad b_3 \end{aligned}$$

Problem 3.2.11. *Consider the matrix and vector*

$$A = \begin{bmatrix} 3 & 0 & 0 \\ 1 & -2 & 0 \\ -1 & 1 & 0 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ -1 \\ c \end{bmatrix}.$$

Why is A singular? For what values c does the linear system $Ax = b$ have no solution? For what values c does the linear system $Ax = b$ have infinitely many solutions?

3.2.4 Backward Substitution for Upper Triangular Linear Systems

A matrix A of order n is **upper triangular** if its nonzero entries are either strictly upper triangular entries or diagonal entries. An **upper triangular linear system** or order n is one whose coefficient matrix is upper triangular. Solving an upper triangular linear system, like that in Fig. 3.2(c), is usually carried out by **backward substitution**. Backward substitution determines first x_n, then x_{n-1}, and so on, until all x_i are found. For example, in Fig. 3.2(c), the third equation determines the value of x_3. Given the value of x_3, the second equation then determines x_2. Finally, given x_2 and x_3, the first equation determines x_1. This process is illustrated in the following example.

Example 3.2.4. In the case in Fig. 3.2(c) the solution is $x_3 = \frac{-15}{(-5)} = 3$, $x_2 = \frac{24-6x_3}{3} = \frac{24-6\cdot 3}{3} = 2$ and $x_1 = \frac{0-(-1)x_3-2x_2}{(-1)} = \frac{0-(-1)\cdot 3-2\cdot 2}{(-1)} = 1$.

As with forward substitution, there are two popular implementations of backward substitution. To motivate these implementations, consider the following upper triangular linear system.

$$\begin{array}{rcrcrcl} a_{1,1}x_1 & + & a_{1,2}x_2 & + & a_{1,3}x_3 & = & b_1 \\ & & a_{2,2}x_2 & + & a_{2,3}x_3 & = & b_2 \\ & & & & a_{3,3}x_3 & = & b_3 \end{array}$$

By transferring the terms involving the off-diagonal entries of A to the right hand side we obtain

$$\begin{array}{rrcrcrcl} a_{1,1}x_1 & & & = & b_1 & - & a_{1,3}x_3 & - & a_{1,2}x_2 \\ & a_{2,2}x_2 & & = & b_2 & - & a_{2,3}x_3 & & \\ & & a_{3,3}x_3 & = & b_3 & & & & \end{array}$$

The right hand sides of these equations can be divided naturally into rows and columns.

- Row–oriented backward substitution updates (modifies) the right hand side one row at a time. That is, after computing $x_n, x_{n-1}, \ldots, x_{i+1}$, we update b_i as:

$$b_i := b_i - (a_{i,i+1}x_{i+1} + a_{i,i+2}x_{i+2} + \cdots + a_{i,n}x_n) = b_i - \sum_{j=i+1}^{n} a_{i,j}x_j$$

  In this case the variable b_i is "overwritten" with the value $b_i - \sum_{j=i+1}^{n} a_{i,j}x_j$. With this procedure to update the right hand side, an algorithm for row-oriented backward substitution could have the form:

$$\begin{array}{l} \text{for each } i = n, n-1, \ldots, 1 \\ \quad \text{update } b_i := b_i - \sum_{j=i+1}^{n} a_{i,j}x_j \\ \quad \text{compute } x_i := b_i / a_{i,i} \end{array}$$

  Notice that each update step uses elements in the ith *row* of A, $a_{i,i+1}, a_{i,i+2}, \ldots, a_{i,n}$.

- Column–oriented backward substitution updates (modifies) the right hand side one column at a time. That is, after computing x_j, we update $b_1, b_2, \ldots, b_{j-1}$ as:

$$\begin{aligned} b_1 &:= b_1 - a_{1,j}x_j \\ b_2 &:= b_2 - a_{2,j}x_j \\ &\vdots \\ b_{j-1} &:= b_{j-1} - a_{j-1,j}x_j \end{aligned}$$

  With this procedure, an algorithm for column-oriented backward substitution could have the form:

$$\begin{aligned} &\text{for each } j = n, n-1, \ldots, 1 \\ &\qquad \text{compute } x_j := b_j/a_{j,j} \\ &\qquad \text{update } b_i := b_i - a_{i,j}x_j, \quad i = 1, 2, \ldots, j-1 \end{aligned}$$

  Notice in this case the update steps use elements in the jth *column* of A, $a_{1,j}, a_{2,j}, \ldots, a_{j-1,j}$.

Pseudocodes implementing row– and column–oriented backward substitution are presented in Fig. 3.6.

Row–Oriented	*Column–Oriented*

Input: matrix $A = [a_{i,j}]$
vector $b = [b_i]$
Output: solution vector $x = [x_i]$
———
for $i = n$ downto 1 do
  for $j = i + 1$ to n
  $b_i := b_i - a_{i,j}x_j$
  next j
  $x_i := b_i/a_{i,i}$
next i
	Input: matrix $A = [a_{i,j}]$
	vector $b = [b_j]$
	Output: solution vector $x = [x_j]$
	———
	for $j = n$ downto 1 do
	  $x_j := b_j/a_{j,j}$
	  for $i = 1$ to $j - 1$
	  $b_i := b_i - a_{i,j}x_j$
	  next i
	next j

Figure 3.6: Pseudocode *Row– and Column–Oriented Backward Substitution.* Note that the algorithm assumes that the matrix A is upper triangular.

The backward substitution process can break down if a diagonal entry of the upper triangular matrix is zero. In this case, the upper triangular matrix is singular, and a linear system involving such a matrix may have no solution or infinitely many solutions.

Problem 3.2.12. *Why is a diagonal linear system also upper triangular?*

Problem 3.2.13. *Illustrate the operation of column–oriented backward substitution when used to solve the upper triangular linear system in Fig. 3.2(c). Hint: Show the value of b each time it has been modified by the for-loop and the value of each entry of x as it is computed.*

Problem 3.2.14. *Use row–oriented backward substitution to solve the linear system:*

$$\begin{aligned} 2x_1 + 2x_2 + 3x_3 + 4x_4 &= 20 \\ 0x_1 + 5x_2 + 6x_3 + 7x_4 &= 34 \\ 0x_1 + 0x_2 + 8x_3 + 9x_4 &= 25 \\ 0x_1 + 0x_2 + 0x_3 + 10x_4 &= 10 \end{aligned}$$

Problem 3.2.15. *Repeat Problem 3.2.14 using the column–oriented backward substitution.*

Problem 3.2.16. *Modify the row– and column–oriented pseudocodes for backward substitution so that the solution x is written over b.*

Problem 3.2.17. *Consider a general upper triangular linear system of order n. Show that row–oriented backward substitution costs $\frac{n(n-1)}{2}$ multiplications, $\frac{n(n-1)}{2}$ subtractions, and n divisions.*

Problem 3.2.18. *Repeat Problem 3.2.17 using column–oriented backward substitution.*

Problem 3.2.19. *Develop pseudocodes, analogous to those in Fig. 3.6, for row– and column–oriented methods of solving the following linear system:*

$$\begin{aligned} a_{1,3}x_3 &= b_1 \\ a_{2,2}x_2 + a_{2,3}x_3 &= b_2 \\ a_{3,1}x_1 + a_{3,2}x_2 + a_{3,3}x_3 &= b_3 \end{aligned}$$

Problem 3.2.20. *Consider the matrix and vector*

$$A = \begin{bmatrix} -1 & 2 & 3 \\ 0 & 0 & 4 \\ 0 & 0 & 2 \end{bmatrix}, \quad b = \begin{bmatrix} 2 \\ c \\ 4 \end{bmatrix}.$$

Why is A singular? For what values c does the linear system $Ax = b$ have no solution? For what values c does the linear system $Ax = b$ have infinitely many solutions?

Problem 3.2.21. *The determinant of a triangular matrix, be it diagonal, lower triangular, or upper triangular, is the product of its diagonal entries.*

(a) Show that a triangular matrix is nonsingular if and only if each of its diagonal entries is nonzero.

(b) Compute the determinant of each of the triangular matrices in Problems 3.2.5, 3.2.11, 3.2.14 and 3.2.20.

Why do these results make sense computationally?

3.3 Gaussian Elimination with Partial Pivoting

The 19^{th} century German mathematician and scientist Carl Friedrich Gauss described a process, called Gaussian elimination in his honor, that uses two elementary operations systematically to transform any given linear system into one that is easy to solve. (Actually, the process was supposedly known centuries earlier.)

The two elementary operations are

(a) exchange two equations

(b) subtract a multiple of one equation from any other equation.

Applying either type of elementary operations to a linear system does not change its solution set. So, we may apply as many of these operations as needed, in any order, and the resulting system of linear equations has the same solution set as the original system. To implement the procedure, though, we need a systematic approach in which we apply the operations. In this section we describe the most commonly implemented scheme, Gaussian elimination with partial pivoting (GEPP). Here our "running examples" will all be computed in exact arithmetic. In the next subsection, we will recompute these examples in four significant digit decimal arithmetic to provide some first insight into the effect of rounding error in the solution of linear systems.

3.3.1 Outline of the GEPP Algorithm

For linear systems of order n, GEPP uses n stages to transform the linear system into upper triangular form. At stage k we eliminate variable x_k from all but the first k equations. To achieve this, each stage uses the same two steps. We illustrate the process on the linear system:

$$\begin{aligned} 1x_1 + \quad 2x_2 + (-1)x_3 &= 0 \\ 2x_1 + (-1)x_2 + \quad 1x_3 &= 7 \\ (-3)x_1 + \quad 1x_2 + \quad 2x_3 &= 3 \end{aligned}$$

Stage 1. Eliminate x_1 from all but the first equation.

The **exchange step**, exchanges equations so that among the coefficients multiplying x_1 in all of the equations, the coefficient in the first equation has largest magnitude. If there is more than one such coefficient, choose the first. If this coefficient with largest magnitude is nonzero, then it is underlined and called the **pivot** for stage 1, otherwise stage 1 had no pivot and we terminate the elimination algorithm. In this example, the pivot occurs in the third equation, so equations 1 and 3 are exchanged.

$$\begin{aligned} 1x_1 + \quad 2x_2 + (-1)x_3 &= 0 \\ 2x_1 + (-1)x_2 + \quad 1x_3 &= 7 \\ \underline{(-3)}\, x_1 + \quad 1x_2 + \quad 2x_3 &= 3 \end{aligned}$$

exchange $\downarrow$ rows 1 and 3

$$\begin{aligned} \underline{(-3)}\, x_1 + \quad 1x_2 + \quad 2x_3 &= 3 \\ 2x_1 + (-1)x_2 + \quad 1x_3 &= 7 \\ 1x_1 + \quad 2x_2 + (-1)x_3 &= 0 \end{aligned}$$

The **elimination step**, eliminates variable x_1 from all but the first equation. For this example, this involves subtracting $m_{2,1} = \frac{2}{-3}$ times equation 1 from equation 2, and $m_{3,1} = \frac{1}{-3}$ times equation 1 from equation 3. Each of the numbers $m_{i,1}$ is a **multiplier**; the first subscript on $m_{i,1}$ indicates from which equation the multiple of the first equation is subtracted. $m_{i,1}$ is computed as the coefficient of x_1 in equation i divided by the coefficient of x_1 in equation 1 (that is, the pivot).

eliminate $\downarrow$ x_1

$$\begin{aligned} \underline{(-3)}\, x_1 + \quad 1x_2 + \quad 2x_3 &= 3 \\ 0x_1 + (-\tfrac{1}{3})x_2 + \quad \tfrac{7}{3}x_3 &= 9 \\ 0x_1 + \quad \tfrac{7}{3}x_2 + (-\tfrac{1}{3})x_3 &= 1 \end{aligned}$$

Stage 2. Eliminate x_2 from all but the first two equations.

Stage 2 repeats the above steps for the variable x_2 on a smaller linear system obtained by removing the first equation from the system at the end of stage 1. This new system involves one fewer unknown (the first stage eliminated variable x_1).

$$\begin{aligned} \underline{(-3)}\, x_1 + \quad 1x_2 + \quad 2x_3 &= 3 \\ (-\tfrac{1}{3})x_2 + \quad \tfrac{7}{3}x_3 &= 9 \\ \underline{\tfrac{7}{3}}\, x_2 + (-\tfrac{1}{3})x_3 &= 1 \end{aligned}$$

exchange $\downarrow$ rows 2 and 3

The **exchange step**, exchanges equations so that among the coefficients multiplying x_2 in all of the remaining equations, the second equation has largest magnitude. If this coefficient with largest magnitude is nonzero, then it is underlined and called the **pivot** for stage 2, otherwise stage 2 has no pivot and we terminate. In our example, the pivot occurs in the third equation, so equations 2 and 3 are exchanged.

$$\begin{aligned} \underline{(-3)}\, x_1 + \quad 1x_2 + \quad 2x_3 &= 3 \\ \underline{\tfrac{7}{3}}\, x_2 + (-\tfrac{1}{3})x_3 &= 1 \\ (-\tfrac{1}{3})x_2 + \quad \tfrac{7}{3}x_3 &= 9 \end{aligned}$$

eliminate $\downarrow$ x_2

The **elimination step**, eliminates variable x_2 from all below the second equation. For our example, this involves subtracting $m_{3,2} = \frac{-1/3}{7/3} = -\frac{1}{7}$ times equation 2 from equation 3. The **multiplier** $m_{i,2}$ is computed as the coefficient of x_2 in equation i divided by the coefficient of x_2 in equation 2 (that is, the pivot).

$$\begin{aligned} \underline{(-3)}\, x_1 + \quad 1x_2 + \quad 2x_3 &= 3 \\ \underline{\tfrac{7}{3}}\, x_2 + (-\tfrac{1}{3})x_3 &= 1 \\ 0x_2 + \quad \tfrac{16}{7}x_3 &= \tfrac{64}{7} \end{aligned}$$

This process continues until the last equation involves only one unknown variable, and the transformed linear system is in upper triangular form. The last stage of the algorithm simply involves identifying the coefficient in the last equation as the final pivot element. In our simple example, we are done at stage 3, where we identify the last pivot element by underlining the coefficient for x_3 in

the last equation, and obtain the upper triangular linear system

$$\begin{aligned} \underline{(-3)}\, x_1 + 1x_2 + \quad 2x_3 &= \ 3 \\ \underline{\tfrac{7}{3}}\, x_2 + (-\tfrac{1}{3})x_3 &= \ 1 \\ \underline{\tfrac{16}{7}}\, x_3 &= \tfrac{64}{7} \end{aligned}$$

GEPP is now finished. When the entries on the diagonal of the final upper triangular linear system are nonzero, as it is in our illustrative example, the linear system is nonsingular and its solution may be determined by backward substitution. (Recommendation: If you are determining a solution by hand in exact arithmetic, you may check that your computed solution is correct by showing that it satisfies all the equations of the original linear system. If you are determining the solution approximately this check may be unreliable as we will see later.)

The diagonal entries of the upper triangular linear system produced by GEPP play an important role. Specifically, the k^{th} diagonal entry, i.e., the coefficient of x_k in the k^{th} equation, is the pivot for the k^{th} stage of GEPP. So, *the upper triangular linear system produced by GEPP is nonsingular if and only if each stage of GEPP has a pivot.*

To summarize:

- GEPP can always be used to transform a linear system of order n into an upper triangular linear system with the same solution set.
- The k^{th} stage of GEPP begins with a linear system that involves $n-k+1$ equations in the $n-k+1$ unknowns $x_k, x_{k+1}, \cdots, x_n$. The k^{th} stage of GEPP eliminates x_k and ends with a linear system that involves $n-k$ equations in the $n-k$ unknowns $x_{k+1}, x_{k+2}, \cdots, x_n$.
- If GEPP finds a non-zero pivot at every stage, then the final upper triangular linear system is nonsingular. The solution of the original linear system can be determined by applying backward substitution to this upper triangular linear system.
- If GEPP fails to find a non-zero pivot at some stage, then the linear system is singular and the original linear system either has no solution or an infinite number of solutions.

We emphasize that **if GEPP does not find a non-zero pivot at some stage, we can conclude immediately that the original linear system is singular**. Consequently, many GEPP codes simply terminate elimination and return a message that indicates the original linear system is singular.

Example 3.3.1. The previous discussion showed that GEPP transforms the linear system to upper triangular form:

$$\begin{aligned} 1x_1 + \quad 2x_2 + (-1)x_3 &= 0 \\ 2x_1 + (-1)x_2 + \quad 1x_3 &= 7 \\ (-3)x_1 + \quad 1x_2 + \quad 2x_3 &= 3 \end{aligned} \quad \rightarrow \quad \cdots \quad \rightarrow \quad \begin{aligned} \underline{(-3)}\, x_1 + 1x_2 + \quad 2x_3 &= \ 3 \\ \underline{\tfrac{7}{3}}\, x_2 + (-\tfrac{1}{3})x_3 &= \ 1 \\ \underline{\tfrac{16}{7}}\, x_3 &= \tfrac{64}{7} \end{aligned}$$

Using backward substitution, we find that the solution of this linear system is given by:

$$x_3 = \frac{\frac{64}{7}}{\frac{16}{7}} = 4\,, \quad x_2 = \frac{1 + \frac{1}{3}\cdot 4}{\frac{7}{3}} = 1\,, \quad x_1 = \frac{3 - 1\cdot 1 - 2\cdot 4}{-3} = 2\,.$$

Example 3.3.2. Consider the linear system:

$$\begin{aligned} 4x_1 + \quad 6x_2 + (-10)x_3 &= 0 \\ 2x_1 + \quad 2x_2 + \quad 2x_3 &= 6 \\ 1x_1 + (-1)x_2 + \quad 4x_3 &= 4 \end{aligned}$$

Applying GEPP to this example we obtain:

In the first stage, the largest coefficient in magnitude of x_1 is already in the first equation, so no exchange steps are needed. The elimination steps then proceed with the multipliers $m_{2,1} = \frac{2}{4} = 0.5$ and $m_{3,1} = \frac{1}{4} = 0.25$.

$$\begin{aligned} \underline{4}\,x_1 + \quad 6x_2 + (-10)x_3 &= 0 \\ 2x_1 + \quad 2x_2 + \quad 2x_3 &= 6 \\ 1x_1 + (-1)x_2 + \quad 4x_3 &= 4 \end{aligned}$$

$$\downarrow$$

In the second stage, we observe that the largest x_2 coefficient in magnitude in the last two equations occurs in the third equation, so the second and third equations are exchanged. The elimination step then proceeds with the multiplier $m_{3,2} = \frac{-1}{-2.5} = 0.4$.

$$\begin{aligned} \underline{4}\,x_1 + \quad 6x_2 + (-10)x_3 &= 0 \\ 0x_1 + \quad (-1)x_2 + \quad 7x_3 &= 6 \\ 0x_1 + \underline{(-2.5)}\,x_2 + \quad 6.5x_3 &= 4 \end{aligned}$$

$$\downarrow$$

$$\begin{aligned} \underline{4}\,x_1 + \quad 6x_2 + (-10)x_3 &= 0 \\ 0x_1 + \underline{(-2.5)}\,x_2 + \quad 6.5x_3 &= 4 \\ 0x_1 + \quad (-1)x_2 + \quad 7x_3 &= 6 \end{aligned}$$

$$\downarrow$$

In the final stage we identify the final pivot entry, and observe that all pivots are non-zero, and thus the linear system is nonsingular and there is a unique solution.

$$\begin{aligned} \underline{4}\,x_1 + \quad 6x_2 + (-10)x_3 &= 0 \\ 0x_1 + \underline{(-2.5)}\,x_2 + \quad 6.5x_3 &= 4 \\ 0x_1 + \quad 0x_2 + \quad \underline{4.4}\,x_3 &= 4.4 \end{aligned}$$

Using backward substitution, we find the solution of the linear system:

$$x_3 = \frac{4.4}{4.4} = 1\,, \quad x_2 = \frac{4 - 6.5 \cdot 1}{-2.5} = 1\,, \quad x_1 = \frac{0 - 6 \cdot 1 + 10 \cdot 1}{4} = 1\,.$$

Example 3.3.3. Consider the linear system:

$$\begin{aligned} 1x_1 + (-2)x_2 + (-1)x_3 &= 2 \\ (-1)x_1 + \quad 2x_2 + (-1)x_3 &= 1 \\ 3x_1 + (-6)x_2 + \quad 9x_3 &= 0 \end{aligned}$$

Applying GEPP to this example we obtain:

In the first stage, the largest coefficient in magnitude of x_1 is in the third equation, so we exchange the first and third equations. The elimination steps then proceed with the multipliers $m_{2,1} = \frac{-1}{3}$ and $m_{3,1} = \frac{1}{3}$.

$$\begin{aligned} 1x_1 + (-2)x_2 + (-1)x_3 &= 2 \\ (-1)x_1 + \quad 2x_2 + (-1)x_3 &= 1 \\ \underline{3}\,x_1 + (-6)x_2 + \quad 9x_3 &= 0 \end{aligned}$$

$$\downarrow$$

$$\begin{aligned} \underline{3}\,x_1 + (-6)x_2 + \quad 9x_3 &= 0 \\ (-1)x_1 + \quad 2x_2 + (-1)x_3 &= 1 \\ 1x_1 + (-2)x_2 + (-1)x_3 &= 2 \end{aligned}$$

$$\downarrow$$

In the second stage, we observe that all coefficients multiplying x_2 in the last two equations are zero. We therefore fail to find a non-zero pivot, and conclude that the linear system is singular.

$$\begin{aligned} \underline{3}\,x_1 + (-6)x_2 + \quad 9x_3 &= 0 \\ 0x_1 + \quad 0x_2 + \quad 2x_3 &= 1 \\ 0x_1 + \quad 0x_2 + (-4)x_3 &= 2 \end{aligned}$$

Problem 3.3.1. *Use GEPP followed by backward substitution to solve the linear system of Example 3.1.1. Explicitly display the value of each pivot and each multiplier.*

Problem 3.3.2. *Use GEPP followed by backward substitution to solve the following linear system. Explicitly display the value of each pivot and multiplier.*

$$\begin{array}{rrrrr} 3x_1 + & 0x_2 + 0x_3 + & 0x_4 = & 6 \\ 2x_1 + (-3)x_2 + 0x_3 + & 0x_4 = & 7 \\ 1x_1 + & 0x_2 + 5x_3 + & 0x_4 = & -8 \\ 0x_1 + & 2x_2 + 4x_3 + (-3)x_4 = & & -3 \end{array}$$

Problem 3.3.3. *Use GEPP and backward substitution to solve the following linear system. Explicitly display the value of each pivot and multiplier.*

$$\begin{array}{rcl} 2x_1 + x_3 & = & 1 \\ x_2 + 4x_3 & = & 3 \\ x_1 + 2x_2 & = & -2 \end{array}$$

Problem 3.3.4. *GEPP provides an efficient way to compute the determinant of any matrix. Recall that the operations used by GEPP are (1) exchange two equations, and (2) subtract a multiple of one equation from another (different) equation. Of these two operations, only the exchange operation changes the value of the determinant of the matrix of coefficients, and then it only changes its sign. In particular, suppose GEPP transforms the coefficient matrix A into the upper triangular coefficient matrix U using m actual exchanges, i.e., exchange steps where an exchange of equations actually occurs. Then*

$$\det(A) = (-1)^m \det(U).$$

(a) Use GEPP to show that

$$\det\left(\begin{bmatrix} 4 & 6 & -10 \\ 2 & 2 & 2 \\ 1 & -1 & 4 \end{bmatrix}\right) = (-1)^1 \det\left(\begin{bmatrix} 4 & 6 & -10 \\ 0 & -2.5 & 6.5 \\ 0 & 0 & 4.4 \end{bmatrix}\right) = -(4)(-2.5)(4.4) = 44$$

Recall from Problem 3.2.21 that the determinant of a triangular matrix is the product of its diagonal entries.

(b) Use GEPP to calculate the determinant of each of the matrices:

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 2 \\ 4 & -3 & 0 \end{bmatrix} \quad \textit{and} \quad \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 3 & 4 & 5 \\ -1 & 2 & -2 & 1 \\ 2 & 6 & 3 & 7 \end{bmatrix}$$

3.3.2 The GEPP Algorithm in Inexact Arithmetic

The GEPP algorithm was outlined in the previous subsection using exact arithmetic throughout. In reality, the solution of linear systems is usually computed using DP standard arithmetic and the errors incurred depend on the IEEE DP (binary) representation of the original system and on the effects of rounding error in IEEE DP arithmetic. Since it is difficult to follow binary arithmetic we will resort to decimal arithmetic to simulate the effects of approximate computation. In this subsection, we will use round-to-nearest 4 significant digit decimal arithmetic to solve the problems introduced in the previous subsection.

First, we illustrate the process on the linear system in Example 3.3.1:

$$\begin{array}{rrrr} 1.000x_1 + & 2.000x_2 + (-1.000)x_3 = & 0 \\ 2.000x_1 + (-1.000)x_2 + & 1.000x_3 = & 7.000 \\ (-3.000)x_1 + & 1.000x_2 + & 2.000x_3 = 3.000 \end{array}$$

Performing the interchange, calculating the multipliers $m_{21} = -0.6667$ and $m_{31} = -0.3333$, and eliminating x_1 from the second and third equations we have

$$\begin{aligned}(-3.000)x_1 + \quad 1.000x_2 + \quad 2.000x_3 &= \ 3.000\\ 0x_1 + (-0.3333)x_2 + \quad 2.333x_3 &= \ 9.000\\ 0x_1 + \quad 2.333x_2 + (-0.3334)x_3 &= 0.9999\end{aligned}$$

Performing the interchange, calculating the multiplier $m_{32} = -0.1429$, and eliminating x_2 from the third equation we have

$$\begin{aligned}(-3.000)x_1 + 1.000x_2 + \quad 2.000x_3 &= \ 3.000\\ 0x_1 + 2.333x_2 + (-0.3334)x_3 &= 0.9999\\ 0x_1 + \quad 0x_2 + \quad 2.285x_3 &= \ 9.143\end{aligned}$$

Backsolving we have $x_3 = 4.001$, $x_2 = 1.000$ and $x_1 = 2.001$, a rounding error level perturbation of the answer with exact arithmetic.

If we consider now Example 3.3.2, we see that all the working was exact in four significant digit decimal arithmetic. So, we will get precisely the same answers as before. However, we cannot represent exactly the arithmetic in this example in binary DP arithmetic. So, computationally, in say MATLAB, we would obtain an answer that differed from the exact answer at the level of roundoff error.

Consider next the problem in Example 3.3.3:

$$\begin{aligned}1.000x_1 + (-2.000)x_2 + (-1.000)x_3 &= 2.000\\ (-1.000)x_1 + \quad 2.000x_2 + (-1.000)x_3 &= 1.000\\ 3.000x_1 + (-6.000)x_2 + \quad 9.000x_3 &= \quad 0\end{aligned}$$

Performing the interchange, calculating the multipliers $m_{21} = -0.3333$ and $m_{31} = 0.3333$, and eliminating x_1 from the second and third equations we have

$$\begin{aligned}3.000x_1 + (-6.000)x_2 + \quad 9.000x_3 &= \quad 0\\ 0x_1 + \quad 0x_2 + \quad (2.000)x_3 &= 1.000\\ 0x_1 + \quad 0x_2 + (-4.000)x_3 &= 2.000\end{aligned}$$

That is, roundoff does not affect the result and we still observe that the pivot for the next stage is zero hence we must stop.

If, instead, we scale the second equation to give

$$\begin{aligned}1x_1 + (-2)x_2 + (-1)x_3 &= 2\\ (-7)x_1 + \quad 14x_2 + (-7)x_3 &= 7\\ 3x_1 + (-6)x_2 + \quad 9x_3 &= 0\end{aligned}$$

then the first step of GEPP is to interchange the first and second equations to give

$$\begin{aligned}(-7)x_1 + \quad 14x_2 + (-7)x_3 &= 7\\ 1x_1 + (-2)x_2 + (-1)x_3 &= 2\\ 3x_1 + (-6)x_2 + \quad 9x_3 &= 0\end{aligned}$$

Next, we calculate the multipliers $m_{21} = \dfrac{1}{(-7)} = -0.1429$ and $m_{31} = \dfrac{3}{(-7)} = -0.4286$. Eliminating we compute

$$\begin{aligned}(-7)x_1 + \quad 14x_2 + (-7)x_3 &= 7\\ 0x_1 + (0.001)x_2 + (-2)x_3 &= 3\\ 0x_1 + \quad 0x_2 + \quad 6x_3 &= 3\end{aligned}$$

Observe that the elimination is now complete. The result differs from the exact arithmetic results in just the $(2,2)$ position but this small change is crucial because the resulting system is now nonsingular. When we compute a solution, we obtain, $x_3 = 0.5$, $x_2 = 4000$ and $x_1 = 8000$; the

large values possibly giving away that something is wrong. In the next subsection we describe a GEPP algorithm. This algorithm includes a test for singularity that would flag the above problem as singular even though all the pivots are nonzero.

Example 3.3.4. Here is a further example designed to reinforce what we have just seen on the effects of inexact arithmetic. Using exact arithmetic, 2 stages of GEPP transforms the singular linear system:

$$\begin{aligned} 1x_1 + \quad 1x_2 + 1x_3 &= 1 \\ 1x_1 + (-1)x_2 + 2x_3 &= 2 \\ 3x_1 + \quad 1x_2 + 4x_3 &= 4 \end{aligned}$$

into the triangular form:

$$\begin{aligned} 3x_1 + \quad 1x_2 + \quad 4x_3 &= \quad 4 \\ 0x_1 + (-4/3)x_2 + 2/3x_3 &= 2/3 \\ 0x_1 + \quad 0x_2 + \quad 0x_3 &= \quad 0 \end{aligned}$$

which has an infinite number of solutions.

Using GEPP with four significant digit decimal arithmetic, we compute multipliers $m_{21} = m_{31} = 0.3333$ and eliminating we get

$$\begin{aligned} 3.000x_1 + \quad 1.000x_2 + \quad 4.000x_3 &= \quad 4.000 \\ 0x_1 + (-1.333)x_2 + \quad 0.6670x_3 &= \quad 0.6670 \\ 0x_1 + \quad 0.6667x_2 + (-0.3330)x_3 &= -0.3330 \end{aligned}$$

So, $m_{32} = -0.5002$ and eliminating we get

$$\begin{aligned} 3.000x_1 + \quad 1.000x_2 + \quad 4.000x_3 &= \quad 4.000 \\ 0x_1 + (-1.333)x_2 + \quad 0.6670x_3 &= \quad 0.6670 \\ 0x_1 + \quad 0x_2 + 0.0006000x_3 &= 0.0006000 \end{aligned}$$

and backsolving we compute $x_3 = 1$ and $x_2 = x_1 = 0$, *one* solution of the original linear system. Here, we observe some mild effects of loss of significance due to cancellation. Later, in Section 3.3.4, we will see much greater numerical errors due to loss of significance.

Problem 3.3.5. *If the original first equation $x_1 + x_2 + x_3 = 1$ in the example above is replaced by $x_1 + x_2 + x_3 = 2$, then with exact arithmetic 2 stages of GE yields $0x_3 = 1$ as the last equation and there is no solution. In four significant digit decimal arithmetic, a suspiciously large solution is computed. Show this and explain why it happens.*

3.3.3 Implementing the GEPP Algorithm

The version of GEPP discussed in this section is properly called *Gaussian elimination with partial pivoting by rows for size*. The phrase "partial pivoting" refers to the fact that only equations are exchanged in the exchange step. An alternative is "complete pivoting" where both equations and unknowns are exchanged. The phrase "by rows for size" refers to the choice in the exchange step where a nonzero coefficient with largest magnitude is chosen as pivot. Theoretically, Gaussian elimination simply requires that any nonzero number be chosen as pivot. Partial pivoting by rows for size serves two purposes. First, it removes the theoretical problem when, prior to the k^{th} exchange step, the coefficient multiplying x_k in the k^{th} equation is zero. Second, in almost all cases, partial pivoting improves the quality of the solution computed when finite precision, rather than exact, arithmetic is used in the elimination step.

In this subsection we present detailed pseudocode that combines GEPP with backward substitution to solve linear systems of equations. The basic algorithm is:

- for stages $k = 1, 2, \ldots, n-1$
  - find the kth pivot
  - if the kth pivot is zero, quit – the linear system is singular
  - perform row interchanges, if needed
  - compute the multipliers
  - perform the elimination
- for stage n, if the final pivot is zero, quit – the linear system is singular
- perform backward substitution

Before presenting pseudocode for the entire algorithm, we consider implementation details for some of the GEPP steps. Assume that we are given the coefficient matrix A and right hand side vector b of the linear system.

- To find the kth pivot, we need to find the largest of the values $|a_{k,k}|, |a_{k+1,k}|, \ldots, |a_{n,k}|$. This can be done using a simple search:

```
p := k
for i = k + 1 to n
   if |a_{i,k}| > |a_{p,k}| then p := i
next i
```

- If the above search finds that $p > k$, then we need to exchange rows. This is fairly simple, though we have to be careful to use "temporary" storage when overwriting the coefficients. (In some languages, such as MATLAB, you can perform the interchanges directly, as we will show in Section 3.7; the temporary storage is created and used in the background and is invisible to the user.)

```
if p > k then
   for j = k to n
      temp := a_{k,j}
      a_{k,j} := a_{p,j}
      a_{p,j} :=temp
   next j
   temp := b_k
   b_k := b_p
   b_p :=temp
endif
```

  Notice that when exchanging equations, we must exchange the coefficients in the matrix A and the corresponding values in the right hand side vector b. (Another possibility is to perform the interchanges virtually; that is, to rewrite the algorithm so that we don't physically interchange elements but instead leave them in place but act as if they had been interchanged. This involves using indirect addressing which may be more efficient than physically performing interchanges.)

- Computing the multipliers, $m_{i,k}$, and performing the elimination step can be implemented as:

```
for i = k + 1 to n
   m_{i,k} := a_{i,k}/a_{k,k}
   a_{i,k} := 0
   for j = k + 1 to n
      a_{i,j} := a_{i,j} − m_{i,k} ∗ a_{k,j}
   next j
   b_i := b_i − m_{i,k} ∗ b_k
next i
```

  Since we know that $a_{i,k}$ should be zero after the elimination step, we do not perform the elimination step on these values, and instead just set them to zero using the instruction $a_{i,k} :=$

0. However, we note that since these elements are not involved in determining the solution via backward substitution, this instruction is unnecessary, and has only a "cosmetic" purpose.

Putting these steps together, and including pseudocode for backward substitution, we obtain the pseudocode shown in Fig. 3.7. Note that we implemented the elimination step to update $a_{i,j}$ in a specific order. In particular, the entries in row $k+1$ are updated first, followed by the entries in row $k+2$, and so on, until the entries in row n are updated. As with the forward and backward substitution methods, we refer to this as *row-oriented* GEPP, and to be consistent we combine it with row-oriented backward substitution. It is not difficult to modify the GEPP procedure so that it is column-oriented, and in that case we would combine it with column-oriented backward substitution.

During stage k, the pseudocode in Fig. 3.7 only modifies equations $(k+1)$ through n and the coefficients multiplying $x_k, x_{k+1}, \cdots, x_n$. Of the three fundamental operations: add (or subtract), multiply, and divide, generally add (or subtract) is fastest, multiply is intermediate, and divide is slowest. So, rather than compute the multiplier as $m_{i,k} := a_{i,k}/a_{k,k}$, one might compute the reciprocal $a_{k,k} := 1/a_{k,k}$ outside the i loop and then compute the multiplier as $m_{i,k} := a_{i,k}a_{k,k}$. In stage k, this would replace $n-k$ divisions with 1 division followed by $n-k$ multiplications, which is usually faster. Keeping the reciprocal in $a_{k,k}$ also helps in backward substitution where each divide is replaced by a multiply.

Problem 3.3.6. *In Fig. 3.7, in the second for-j loop (the elimination loop) the code could be logically simplified by recognizing that the effect of $a_{i,k} = 0$ could be achieved by removing this statement and extending the for-j loop so it starts at $j = k$ instead of $j = k+1$. Why is this NOT a good idea?*

Problem 3.3.7. *In Fig. 3.7, the elimination step is row–oriented. Change the elimination step so it is column–oriented. Hint: You must exchange the order of the for-i and for-j loops.*

Problem 3.3.8. *In Fig. 3.7, show that the elimination step of the k^{th} stage of GEPP requires $n-k$ divisions to compute the multipliers, and $(n-k)^2 + n - k$ multiplications and subtractions to perform the elimination. Conclude that the elimination steps of GEPP requires about*

$$\begin{array}{lllll} \sum_{k=1}^{n-1}(n-k) & = & \sum_{k=1}^{n-1} k & = & \frac{n(n-1)}{2} \sim \frac{n^2}{2} \text{ divisions} \\ \sum_{k=1}^{n-1}(n-k)^2 & = & \sum_{k=1}^{n-1} k^2 & = & \frac{n(n-1)(2n-1)}{6} \sim \frac{n^3}{3} \text{ multiplications and subtractions} \end{array}$$

The notation $f(n) \sim g(n)$ is used when the ratio $\frac{f(n)}{g(n)}$ approaches 1 as n becomes large.

Problem 3.3.9. *How does the result in Problem 3.3.8 change if we first compute the reciprocals of the pivots and then use these values in computing the multipliers?*

Problem 3.3.10. *How many comparisons does GEPP use to find pivot elements?*

Problem 3.3.11. *How many assignment statements are needed for all the interchanges in the GEPP algorithm, assuming that at each stage an interchange is required?*

3.3.4 The Role of Interchanges

To illustrate the effect of the choice of pivot, we apply Gaussian elimination to the linear system

$$\begin{aligned} 0.000025x_1 + 1x_2 &= 1 \\ 1x_1 + 1x_2 &= 2 \end{aligned}$$

Row-Oriented GEPP with Backward Substitution

Input: matrix $A = [a_{i,j}]$
vector $b = [b_i]$

Output: solution vector $x = [x_i]$

```
for k = 1 to n − 1
    p := k
    for i = k + 1 to n
        if |a_{i,k}| > |a_{p,k}| then p := i
    next i
    if p > k then
        for j = k to n
            temp := a_{k,j}; a_{k,j} := a_{p,j}; a_{p,j} := temp
        next j
        temp := b_k; b_k := b_p; b_p := temp
    endif
    if a_{k,k} = 0 then return (linear system is singular)
    for i = k + 1 to n
        m_{i,k} := a_{i,k}/a_{k,k}
        a_{i,k} := 0
        for j = k + 1 to n
            a_{i,j} := a_{i,j} − m_{i,k} * a_{k,j}
        next j
        b_i := b_i − m_{i,k} * b_k
    next i
next k
if a_{n,n} = 0 then return (linear system is singular)

for i = n downto 1 do
    for j = i + 1 to n
        b_i := b_i − a_{i,j} * x_j
    next j
    x_i := b_i/a_{i,i}
next i
```

Figure 3.7: Pseudocode for *row oriented GEPP and Backward Substitution*

for which the exact solution is $x_1 = \frac{40000}{39999} \approx 1$ and $x_2 = \frac{39998}{39999} \approx 1$.

What result is obtained with Gaussian elimination using floating–point arithmetic? To make pen-and-paper computation easy, we use round-to-nearest 4 significant digit decimal arithmetic; i.e., the result of every add, subtract, multiply, and divide is rounded to the nearest decimal number with 4 significant digits.

Consider what happens if the exchange step is not used. The multiplier is computed exactly because its value $\frac{1}{0.000025} = 40000$ rounds to itself. The elimination step subtracts 40000 times equation 1 from equation 2. Now 40000 times equation 1 is computed exactly. So, the only rounding error in the elimination step is at the subtraction. There, the coefficient multiplying x_2 in the second equation is $1 - 40000 = -39999$, which rounds to -40000, and the right hand side is $2 - 40000 = -39998$, which also rounds to -40000. So the result is

$$\begin{aligned} 0.000025x_1 + \quad 1x_2 &= \quad 1 \\ 0x_1 + (-40000)x_2 &= -40000 \end{aligned}$$

Backward substitution commits no further rounding errors and produces the approximate solution

$$x_2 = \frac{-40000}{-40000} = 1, \quad x_1 = \frac{1 - x_2}{0.000025} = 0$$

This computed solution differs significantly from the exact solution. Why? Observe that the computed solution has $x_2 = 1$. This is an accurate approximation of its exact value $\frac{39998}{39999}$, in error by $\frac{39998}{39999} - 1 = -\frac{1}{39999}$. When the approximate value of x_1 is computed as $\frac{1-x_2}{0.000025}$, catastrophic cancellation occurs when the approximate value $x_2 = 1$ is subtracted from 1.

If we include the exchange step, the result of the exchange step is

$$\begin{aligned} 1x_1 + 1x_2 &= 2 \\ 0.000025x_1 + 1x_2 &= 1 \end{aligned}$$

The multiplier $\frac{0.000025}{1} = 0.000025$ is computed exactly, as is 0.000025 times equation 1. The result of the elimination step is

$$\begin{aligned} 1x_1 + 1x_2 &= 2 \\ 0x_1 + 1x_2 &= 1 \end{aligned}$$

because, in the subtract operation, $1 - 0.000025 = 0.999975$ rounds to 1 and $1 - 2 \times 0.000025 = 0.99995$ rounds to 1. Backward substitution commits no further rounding errors and produces an approximate solution

$$x_2 = 1, \quad x_1 = \frac{2 - x_2}{1} = 1$$

that is correct to four significant digits.

The above computation uses round-to-nearest 4 significant digit decimal arithmetic. It leaves open the question of what impact the choice of the number of digits in the above calculation so next we repeat the calculation above using first round-to-nearest 5 significant digit decimal arithmetic then round-to-nearest 3 significant digit decimal arithmetic to highlight the differences that may arise when computing to different accuracies.

Consider what happens if we work in round-to-nearest 5 significant digit decimal arithmetic without exchanges. The multiplier is computed exactly because its value $\frac{1}{0.000025} = 40000$ rounds to itself. The elimination step subtracts 40000 times equation 1 from equation 2. Now, 40000 times equation 1 is computed exactly. So, the only possible rounding error in the elimination step is at the subtraction. There, the coefficient multiplying x_2 in the second equation is $1 - 40000 = -39999$, which is already rounded to five digits, and the right hand side is $2 - 40000 = -39998$, which is again correctly rounded. So the result is

$$\begin{aligned} 0.000025x_1 + \quad 1x_2 &= \quad 1 \\ 0x_1 + (-39999)x_2 &= -39998 \end{aligned}$$

Backward substitution produces the approximate solution

$$x_2 = \frac{-39998}{-39999} = .99997, \quad x_1 = \frac{1 - x_2}{0.000025} = 1.2$$

So, the impact of the extra precision is to compute a less inaccurate result. Next, consider what happens if we permit exchanges and use round-to-nearest 5 significant digit decimal arithmetic. The result of the exchange step is

$$\begin{aligned} 1x_1 + 1x_2 &= 2 \\ 0.000025x_1 + 1x_2 &= 1 \end{aligned}$$

The multiplier $\frac{0.000025}{1} = 0.000025$ is computed exactly, as is 0.000025 times equation 1. The result of the elimination step is

$$\begin{aligned} 1x_1 + \quad 1x_2 \quad &= \quad 2 \\ 0x_1 + 0.99998x_2 &= 0.99995 \end{aligned}$$

because, in the subtract operation, $1 - 0.000025 = 0.999975$ rounds to 0.99998 and $1 - 2 \times 0.000025 = 0.99995$ is correctly rounded. Backward substitution produces an approximate solution

$$x_2 = .99997, \quad x_1 = \frac{2 - x_2}{1} = 1$$

that is correct to five significant digits

Finally, we work in round-to-nearest 3 significant digit decimal arithmetic without using exchanges. The multiplier is computed exactly because its value $\frac{1}{0.000025} = 40000$ rounds to itself. The elimination step subtracts 40000 times equation 1 from equation 2. Now 40000 times equation 1 is computed exactly. So, the only possible rounding error in the elimination step is at the subtraction. There, the coefficient multiplying x_2 in the second equation is $1 - 40000 = -40000$, to three digits, and the right hand side is $2 - 40000 = -40000$, which is again correctly rounded. So, the result is

$$\begin{aligned} 0.000025x_1 + \quad 1x_2 &= \quad 1 \\ 0x_1 + (-40000)x_2 &= -40000 \end{aligned}$$

Backward substitution produces the approximate solution

$$x_2 = \frac{-40000}{-40000} = 1.0, \quad x_1 = \frac{1 - x_2}{0.000025} = 0.0$$

So, for this example working to three digits without exchanges produces the same result as working to four digits. The reader may verify that working to three digits with exchanges produces the correct result to three digits

Partial pivoting by rows for size is a heuristic. One explanation why this heuristic is generally successful is as follows. Given a list of candidates for the next pivot, those with smaller magnitude are more likely to have been formed by a subtract magnitude computation of larger numbers, so the resulting cancellation might make them less accurate than the other candidates for pivot. The multipliers determined by dividing by such smaller magnitude, inaccurate numbers are therefore larger magnitude, inaccurate numbers. Consequently, elimination may produce large and unexpectedly inaccurate coefficients; in extreme circumstances, these large coefficients may contain little information from the coefficients of the original equations. While the heuristic of partial pivoting by rows for size generally improves the chances that GEPP will produce an accurate answer, it is neither perfect nor always better than any other interchange strategy.

Problem 3.3.12. *Use Gaussian elimination to verify that the exact solution of the linear system*

$$\begin{aligned} 0.000025x_1 + 1x_2 &= 1 \\ 1x_1 + 1x_2 &= 2 \end{aligned}$$

is

$$x_1 = \frac{40000}{39999} \approx 1, \quad x_2 = \frac{39998}{39999} \approx 1$$

Problem 3.3.13. *(Watkins) Carry out Gaussian elimination* without *exchange steps and with row–oriented backward substitution on the linear system:*

$$\begin{aligned} 0.002x_1 + 1.231x_2 + 2.471x_3 &= 3.704 \\ 1.196x_1 + 3.165x_2 + 2.543x_3 &= 6.904 \\ 1.475x_1 + 4.271x_2 + 2.142x_3 &= 7.888 \end{aligned}$$

Use round-to-nearest 4 significant digit decimal arithmetic. Display each pivot, each multiplier, and the result of each elimination step. Does catastrophic cancellation occur, and if so where? Hint: The exact solution is $x_1 = 1$, $x_2 = 1$ and $x_3 = 1$. The computed solution with approximate arithmetic and no exchanges is $x_1 = 4.000$, $x_2 = -1.012$ and $x_3 = 2.000$.

Problem 3.3.14. *Carry out GEPP (with exchange steps) and row–oriented backward substitution on the linear system in Problem 3.3.13. Use round-to-nearest 4 significant digit decimal arithmetic. Display each pivot, each multiplier, and the result of each elimination step. Does catastrophic cancellation occur, and if so where?*

Problem 3.3.15. *Round the entries in the linear system in Problem 3.3.13 to three significant digits. Now, repeat the calculations in Problems 3.3.13 and 3.3.14 using round-to-nearest 3 significant digit decimal arithmetic. What do you observe?*

3.4 Gaussian Elimination and Matrix Factorizations

The concept of matrix factorizations is fundamentally important in the process of numerically solving linear systems, $Ax = b$. The basic idea is to decompose the matrix A into a product of *simply solved systems*, from which the solution of $Ax = b$ can be easily computed. The idea is similar to what we might do when trying to find the roots of a polynomial. For example, the equations

$$x^3 - 6x^2 + 11x - 6 = 0 \quad \text{and} \quad (x-1)(x-2)(x-3) = 0$$

are equivalent, but the factored form is clearly much easier to solve. In general, we cannot solve linear systems so easily (i.e., by inspection), but decomposing A makes solving the linear system $Ax = b$ computationally simpler. Some knowledge of matrix algebra, especially matrix multiplication, is needed to understand the concepts introduced here; a review is given in Section 1.2.

3.4.1 *LU* Factorization

Our aim here is to show that if Gaussian elimination can be used to reduce an $n \times n$ matrix A to upper triangular form, then the information computed in the elimination process can be used to write A as the product

$$A = LU$$

where L is a unit lower triangular matrix (that is, a lower triangular matrix with 1's on the diagonal) and U is an upper triangular matrix. This is called a *matrix factorization*, and we will see that the idea of matrix factorization is very powerful when solving, analyzing and understanding linear algebra problems.

To see how we can get to the LU factorization, suppose we apply Gaussian elimination to a $n \times n$ matrix A without interchanging any rows. For example, for $n = 3$,

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \longrightarrow \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ 0 & a_{2,2}^{(1)} & a_{2,3}^{(1)} \\ 0 & a_{3,2}^{(1)} & a_{3,3}^{(1)} \end{bmatrix} \longrightarrow \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ 0 & a_{2,2}^{(1)} & a_{2,3}^{(1)} \\ 0 & 0 & a_{3,3}^{(2)} \end{bmatrix}$$

$$m_{2,1} = \frac{a_{2,1}}{a_{1,1}}, \quad m_{3,1} = \frac{a_{3,1}}{a_{1,1}} \qquad m_{3,2} = \frac{a_{3,2}^{(1)}}{a_{2,2}^{(1)}}$$

Here the superscript on the entry $a_{i,j}^{(k)}$ indicates an element of the matrix modified during the kth elimination step. Recall that, in general, the multipliers are computed as

$$m_{i,j} = \frac{\text{element to be eliminated}}{\text{current pivot element}}.$$

Instead of using arrows to show the elimination steps, we can use matrix-matrix multiplication with *elimination matrices*. Specifically, consider the multipliers m_{21} and m_{31} used to eliminate the entries a_{21} and a_{31} in the first column of A. If we put these into a unit lower triangular matrix as follows,

$$M_1 = \begin{bmatrix} 1 & 0 & 0 \\ -m_{21} & 1 & 0 \\ -m_{31} & 0 & 1 \end{bmatrix},$$

then

$$\begin{aligned} M_1 A &= \begin{bmatrix} 1 & 0 & 0 \\ -m_{21} & 1 & 0 \\ -m_{31} & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \\ &= \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} - m_{21}a_{11} & a_{22} - m_{21}a_{12} & a_{23} - m_{21}a_{13} \\ a_{31} - m_{31}a_{11} & a_{32} - m_{31}a_{12} & a_{33} - m_{31}a_{13} \end{bmatrix} \\ &= \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} \\ 0 & a_{32}^{(1)} & a_{33}^{(1)} \end{bmatrix} \end{aligned}$$

Now use the multipliers for the next column to similarly define the elimination matrix M_2,

$$M_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -m_{32} & 1 \end{bmatrix}$$

and observe that

$$\begin{aligned} M_2(M_1 A) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -m_{32} & 1 \end{bmatrix} \begin{bmatrix} a_{1,1} & a_{12} & a_{13} \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} \\ 0 & a_{32}^{(1)} & a_{33}^{(1)} \end{bmatrix} \\ &= \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} \\ 0 & a_{32}^{(1)} - m_{32}a_{22}^{(1)} & a_{3,3}^{(1)} - m_{32}a_{23}^{(1)} \end{bmatrix} \\ &= \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} \\ 0 & 0 & a_{33}^{(2)} \end{bmatrix} \end{aligned}$$

Thus, for this 3×3 example, we have

$$M_2 M_1 A = U \quad \text{(upper triangular)}$$

Notice that M_1 and M_2 are nonsingular (as an exercise you should explain why this is the case), and so we can write

$$A = M_1^{-1} M_2^{-1} U$$

The next observation we make is that the inverse of an elimination matrix is very easy to compute, we just need to change the signs of $-m_{ij}$ to m_{ij}; that is,

$$M_1^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ m_{21} & 1 & 0 \\ m_{31} & 0 & 1 \end{bmatrix} \quad \text{and} \quad M_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & m_{32} & 1 \end{bmatrix}$$

This can be easily verified by showing that $M_1^{-1}M_1 = I$ and $M_2^{-1}M_2 = I$.

The final observation we need is that the product of elimination matrices (and the product of their inverses) is a unit lower triangular matrix. In particular, observe that

$$M_1^{-1}M_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ m_{21} & 1 & 0 \\ m_{31} & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & m_{32} & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ m_{21} & 1 & 0 \\ m_{31} & m_{32} & 1 \end{bmatrix}$$

Thus, for this simple 3×3 example, we have computed

$$A = (M_1^{-1}M_2^{-1})U = LU$$

where L is a unit lower triangular matrix, with multipliers below the main diagonal, and U is the upper triangular matrix obtained after the elimination is complete.

It is not difficult to generalize (e.g., by induction) to a general $n \times n$ matrix. Specifically, if A is an $n \times n$ matrix, then:

- An elimination matrix M_j is the identity matrix, except that the jth column has negative multipliers, $-m_{ij}$, below the main diagonal, $i = j+1, j+2, \ldots, n$,

$$M_i = \begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & -m_{j+1,j} & & \\ & & \vdots & \ddots & \\ & & -m_{n,j} & & 1 \end{bmatrix}$$

- The inverse of an elimination matrix, M_j^{-1}, is easy to compute by simply changing signs of the multipliers,

$$M_j^{-1} = \begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & m_{j+1,j} & & \\ & & \vdots & \ddots & \\ & & m_{n,j} & & 1 \end{bmatrix}$$

- If the process does not break down (that is, all the pivot elements, $a_{1,1}, a_{2,2}^{(1)}, a_{3,3}^{(2)}, \ldots$, are nonzero) then we can construct elimination matrices M_1, M_2, …, M_{n-2}, M_{n-1} such that

$$M_{n-1}M_{n-2}\cdots M_2M_1A = U \quad \text{(upper triangular)}$$

or

$$A = (M_1^{-1}M_2^{-1}\cdots M_{n-2}^{-1}M_{n-1}^{-1})U = LU$$

where

$$L = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ m_{2,1} & 1 & 0 & \cdots & 0 \\ m_{3,1} & m_{3,2} & 1 & & 0 \\ \vdots & \vdots & & \ddots & \vdots \\ m_{n,1} & m_{n,2} & m_{n,3} & \cdots & 1 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} \text{upper triangular matrix} \\ \text{after the elimination is} \\ \text{complete} \end{bmatrix}.$$

Example 3.4.1. Let

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

Using Gaussian elimination without row interchanges, we obtain

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & -3 & 2 \\ 3 & 1 & -1 \end{bmatrix} \xrightarrow[\substack{m_{2,1}=2 \\ m_{3,1}=3}]{} \begin{bmatrix} 1 & 2 & 3 \\ 0 & -7 & -4 \\ 0 & -5 & -10 \end{bmatrix} \xrightarrow[m_{3,2}=\frac{5}{7}]{} \begin{bmatrix} 1 & 2 & 3 \\ 0 & -7 & -4 \\ 0 & 0 & -\frac{50}{7} \end{bmatrix}$$

and thus

$$L = \begin{bmatrix} 1 & 0 & 0 \\ m_{2,1} & 1 & 0 \\ m_{3,1} & m_{3,2} & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & \frac{5}{7} & 1 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} 1 & 2 & 3 \\ 0 & -7 & -4 \\ 0 & 0 & -\frac{50}{7} \end{bmatrix}.$$

It is straightforward to verify that $A = LU$.

If we can compute the factorization $A = LU$, then

$$Ax = b \;\Rightarrow\; LUx = b \;\Rightarrow\; Ly = b, \text{ where } Ux = y.$$

So, to solve $Ax = b$:

- Compute the factorization $A = LU$.
- Solve $Ly = b$ using forward substitution.
- Solve $Ux = y$ using backward substitution.

Example 3.4.2. Consider solving the linear system $Ax = b$, where

$$A = LU = \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & -2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & -1 \\ 0 & 2 & -1 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} 2 \\ 9 \\ -1 \end{bmatrix}$$

Since the LU factorization of A is given, we need only:

- Solve $Ly = b$, or
$$\begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & -2 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 9 \\ -1 \end{bmatrix}$$
Using forward substitution, we obtain
$$y_1 = 2$$
$$3y_1 + y_2 = 9 \Rightarrow y_2 = 9 - 3(2) = 3$$
$$2y_1 - 2y_2 + y_3 = -1 \Rightarrow y_3 = -1 - 2(2) + 2(3) = 1$$
- Solve $Ux = y$, or
$$\begin{bmatrix} 1 & 2 & -1 \\ 0 & 2 & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix}$$
Using backward substitution, we obtain

$x_3 = 1.$

$2x_2 - x_3 = 3 \Rightarrow x_2 = (3+1)/2 = 2.$

$x_1 + 2x_2 - x_3 = 2 \Rightarrow x_1 = 2 - 2(2) + 1 = -1.$

Therefore, the solution of $Ax = b$ is given by

$$x = \begin{bmatrix} -1 \\ 2 \\ 1 \end{bmatrix}.$$

Problem 3.4.1. *Find the LU factorization of each of the following matrices:*

$$A = \begin{bmatrix} 4 & 2 & 1 & 0 \\ -4 & -6 & 1 & 3 \\ 8 & 16 & -3 & -4 \\ 20 & 10 & 4 & -3 \end{bmatrix}, \qquad B = \begin{bmatrix} 3 & 6 & 1 & 2 \\ -6 & -13 & 0 & 1 \\ 1 & 2 & 1 & 1 \\ -3 & -8 & 1 & 12 \end{bmatrix}$$

Problem 3.4.2. *Suppose the LU factorization of a matrix A is given by:*

$$A = LU = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1 & 1 & 1 \end{bmatrix} \begin{bmatrix} -3 & 2 & -1 \\ 0 & -2 & 5 \\ 0 & 0 & -2 \end{bmatrix}$$

For $b = \begin{bmatrix} -1 \\ -7 \\ -6 \end{bmatrix}$, *solve* $Ax = b$.

Problem 3.4.3. *Suppose*

$$A = \begin{bmatrix} 1 & 3 & -4 \\ 0 & -1 & 5 \\ 2 & 0 & 4 \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix}.$$

(a) Use Gaussian elimination without row interchanges to find the factorization $A = LU$.

(b) Use the factorization of A to solve $Ax = b$.

Problem 3.4.4. *Suppose*

$$A = \begin{bmatrix} 4 & 8 & 12 & -8 \\ -3 & -1 & 1 & -4 \\ 1 & 2 & -3 & 4 \\ 2 & 3 & 2 & 1 \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} 3 \\ 60 \\ 1 \\ 5 \end{bmatrix}.$$

(a) Use Gaussian elimination without row interchanges to find the factorization $A = LU$.

(b) Use the factorization of A to solve $Ax = b$.

3.4.2 $PA = LU$ Factorization

The practical implementation of Gaussian elimination uses partial pivoting to determine if row interchanges are needed. We show that this results in a modification of the LU factorization. Row interchanges can by represented mathematically as multiplication by a *permutation matrix*, obtained by interchanging rows of the identity matrix.

Example 3.4.3. Consider the matrix

$$A = \begin{bmatrix} 0 & 3 & 8 & -6 \\ 2 & -3 & 0 & 1 \\ -5 & 2 & -4 & 7 \\ 1 & 1 & -1 & -1 \end{bmatrix}$$

In Gaussian elimination with partial pivoting, because $|a_{13}| = 5$ is the largest (in magnitude) entry in the first column, we begin by switching rows 1 and 3. This can be represented in terms of matrix-matrix multiplication with a permutation matrix P, which is constructed by switching the first and third rows of a 4×4 identity matrix. Specifically,

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{array}{c} \text{switch} \\ \longrightarrow \\ \text{rows 1 and 3} \end{array} \quad P = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Multiplying the matrix A on the left by P switches its first and third rows. That is,

$$PA = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 3 & 8 & -6 \\ 2 & -3 & 0 & 1 \\ -5 & 2 & -4 & 7 \\ 1 & 1 & -1 & -1 \end{bmatrix} = \begin{bmatrix} -5 & 2 & -4 & 7 \\ 2 & -3 & 0 & 1 \\ 0 & 3 & 8 & -6 \\ 1 & 1 & -1 & -1 \end{bmatrix}$$

We can now combine elimination matrices with permutation matrices to describe Gaussian elimination with partial pivoting using matrix-matrix multiplications. Specifically,

$$M_{n-1}P_{n-1} \cdots M_2P_2M_1P_1A = U$$

where

- P_1 is the permutation matrix that swaps the largest, in magnitude, entry in the first column of A to the (1,1) location.
- M_1 is the elimination matrix that zeros out all entries of the first column of P_1A below the (1,1) pivot entry.
- P_2 is the permutation matrix that swaps the largest, in magnitude, entry on and below the (2,2) diagonal entry in the second column of M_1P_1A with the (2,2) location.
- M_2 is the elimination matrix that zeros out all entries of the second column of $P_2M_1P_1A$ below the (2,2) pivot entry.
- etc.

Thus, after completing Gaussian elimination with partial pivoting, we have

$$A = (M_{n-1}P_{n-1} \cdots M_2P_2M_1P_1)^{-1}U\,. \tag{3.1}$$

Unfortunately the matrix $(M_{n-1}P_{n-1} \cdots M_2P_2M_1P_1)^{-1}$ is not lower triangular, but it is sometimes called a "psychologically lower triangular matrix" because it can be transformed into a triangular matrix by permutation. Specifically, if we multiply all of the permutation matrices,

$$P = P_{n-1} \cdots P_2P_1\,,$$

then P is a permutation matrix that combines all row interchanges, and

$$L = P(M_{n-1}P_{n-1} \cdots M_2P_2M_1P_1)^{-1}$$

is a unit lower triangular matrix. Thus, if we multiply P on both sides of equation (3.1), we obtain

$$\begin{aligned} PA &= P(M_{n-1}P_{n-1}\cdots M_2P_2M_1P_1)^{-1}U \\ &= LU\,. \end{aligned}$$

Therefore, when we apply Gaussian elimination with partial pivoting by rows to reduce A to upper triangular form, we obtain an LU factorization of a *permuted* version of A. That is,

$$PA = LU$$

where P is a permutation matrix representing *all* row interchanges in the order that they are applied.

When writing code, or performing hand calculations, it is not necessary to explicitly form any of the matrices P_i or M_i, but instead we can proceed as for the LU factorization, and keep track of the row interchanges and multipliers as follows:

- Each time we switch rows of A, we switch corresponding multipliers in L. For example, if at stage 3 we switch rows 3 and 5, then we must also switch the previously computed multipliers $m_{3,k}$ and $m_{5,k}$, $k = 1, 2$.
- Begin with $P = I$. Each time we switch rows of A, we switch corresponding rows of P.

Example 3.4.4. This example illustrates the process of computing $PA = LU$ for the matrix

$$A = \begin{bmatrix} 1 & 2 & 4 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}.$$

A	P	multipliers
$\begin{bmatrix} 1 & 2 & 4 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	nothing yet
$\downarrow$	$\downarrow$	$\downarrow$
$\begin{bmatrix} 7 & 8 & 9 \\ 4 & 5 & 6 \\ 1 & 2 & 4 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$	nothing yet
$\downarrow$	$\downarrow$	$\downarrow$
$\begin{bmatrix} 7 & 8 & 9 \\ 0 & \frac{3}{7} & \frac{6}{7} \\ 0 & \frac{6}{7} & \frac{19}{7} \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$	$m_{21} = \frac{4}{7} \quad m_{31} = \frac{1}{7}$
$\downarrow$	$\downarrow$	$\downarrow$
$\begin{bmatrix} 7 & 8 & 9 \\ 0 & \frac{6}{7} & \frac{19}{7} \\ 0 & \frac{3}{7} & \frac{6}{7} \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$	$m_{21} = \frac{1}{7} \quad m_{31} = \frac{4}{7}$
$\downarrow$	$\downarrow$	$\downarrow$
$\begin{bmatrix} 7 & 8 & 9 \\ 0 & \frac{6}{7} & \frac{19}{7} \\ 0 & 0 & -\frac{1}{2} \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$	$m_{21} = \frac{1}{7} \quad m_{31} = \frac{4}{7} \quad m_{32} = \frac{1}{2}$

From the information in the final step of the process, we obtain the $PA = LU$ factorization, where

$$P = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{7} & 1 & 0 \\ \frac{4}{7} & \frac{1}{2} & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 7 & 8 & 9 \\ 0 & \frac{6}{7} & \frac{19}{7} \\ 0 & 0 & -\frac{1}{2} \end{bmatrix}$$

An efficient algorithm for computing the $PA = LU$ factorization usually does not explicitly construct P, but instead keeps an index of "pointers" to rows, but we will leave that discussion for more advanced courses on numerical linear algebra.

If we can compute the factorization $PA = LU$, then

$$Ax = b \;\Rightarrow PAx = Pb \;\Rightarrow\; LUx = Pb \;\Rightarrow\; Ly = Pb, \text{ where } Ux = y.$$

Therefore, to solve $Ax = b$:

- Compute the factorization $PA = LU$.
- Permute entries of b to obtain $d = Pb$.
- Solve $Ly = d$ using forward substitution.
- Solve $Ux = y$ using backward substitution.

It is important to emphasize the importance, and power of matrix factorizations. The cost of computing $A = LU$ or $PA = LU$ requires $O(n^3)$ FLOPS, but forward and backward solves require only $O(n^2)$ FLOPS. Thus, if we need to solve multiple linear systems where the matrix A does not change, but with different right hand side vectors, such as

$$Ax_1 = b_1\,, \quad Ax_2 = b_2 \quad \cdots$$

then we need only compute the (relatively expensive) matrix factorization once, and reuse it for all linear system solves.

Matrix factorizations might be useful for other calculations, such as computing the determinant of A. Recall the following properties of determinants:

- The determinant of a product of matrices is the product of their determinants. Thus, in particular,
$$\det(PA) = \det(P)\det(A) \quad \text{and} \quad \det(LU) = \det(L)\det(U)\,.$$
- The determinant of the permutation matrix P is ± 1; it is $+1$ if an even number of row swaps were performed, and -1 if odd number of row swaps were performed.
- The determinant of a triangular matrix is the product of its diagonal entries. In particular, $\det(L) = 1$ because it is a unit lower triangular matrix.

Using these properties, we see that

$$\begin{aligned} \det(PA) &= \det(LU) \\ \det(P)\det(A) &= \det(L)\det(U) \\ \pm\det(A) &= \det(U) \\ \det(A) &= \pm\det(U) = \pm u_{11}u_{22}\cdots u_{nn} \end{aligned}$$

Thus, once we have the $PA = LU$ factorization, it is trivial to compute the determinant of A.

Example 3.4.5. Use the $PA = LU$ factorization of Example 3.4.4 to solve $Ax = b$, where

$$b^T = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

Since the $PA = LU$ factorization is given, we need only:

- Obtain $d = Pb = \begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix}$
- Solve $Ly = d$, or

$$\begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{7} & 1 & 0 \\ \frac{4}{7} & \frac{1}{2} & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix}$$

  Using forward substitution, we obtain

  $y_1 = 3$
  $\frac{1}{7}y_1 + y_2 = 1 \Rightarrow y_2 = 1 - \frac{1}{7}(3) = \frac{4}{7}$
  $\frac{4}{7}y_1 + \frac{1}{2}y_2 + y_3 = 2 \Rightarrow y_3 = 2 - \frac{4}{7}(3) - \frac{1}{2}(\frac{4}{7}) = 0$

- Solve $Ux = y$, or

$$\begin{bmatrix} 7 & 8 & 9 \\ 0 & \frac{6}{7} & \frac{19}{7} \\ 0 & 0 & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ \frac{4}{7} \\ 0 \end{bmatrix}$$

  Using backward substitution, we obtain

  $-\frac{1}{2}x_3 = 0 \Rightarrow x_3 = 0.$
  $\frac{6}{7}x_2 + \frac{19}{7}x_3 = \frac{4}{7} \Rightarrow x_2 = \frac{7}{6}(\frac{4}{7} - \frac{19}{7}(0)) = \frac{2}{3}.$
  $7x_1 + 8x_2 + 9x_3 = 3 \Rightarrow x_1 = \frac{1}{7}(3 - 8(\frac{2}{3}) - 9(0)) = -\frac{1}{3}.$

  Therefore, the solution of $Ax = b$ is given by

$$x = \begin{bmatrix} -\frac{1}{3} \\ \frac{2}{3} \\ 0 \end{bmatrix}.$$

Problem 3.4.5. *Use Gaussian elimination with partial pivoting to find the $PA = LU$ factorization of the matrices:*

$$A = \begin{bmatrix} 1 & 3 & -4 \\ 0 & -1 & 5 \\ 2 & 0 & 4 \end{bmatrix}, \; B = \begin{bmatrix} 3 & 6 & 9 \\ 2 & 5 & 2 \\ -3 & -4 & -11 \end{bmatrix}, \; C = \begin{bmatrix} 2 & -1 & 0 & 3 \\ 0 & -\frac{3}{4} & \frac{1}{2} & 4 \\ 1 & 1 & 1 & -\frac{1}{2} \\ 2 & -\frac{5}{2} & 1 & 13 \end{bmatrix}$$

Problem 3.4.6. *Suppose that $b^T = \begin{bmatrix} 3 & 60 & 1 & 5 \end{bmatrix}$, and suppose that Gaussian elimination with partial pivoting has been used on a matrix A to obtain its $PA = LU$ factorization, where*

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3/4 & 1 & 0 & 0 \\ 1/4 & 0 & 1 & 0 \\ 1/2 & -1/5 & 1/3 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 4 & 8 & 12 & -8 \\ 0 & 5 & 10 & -10 \\ 0 & 0 & -6 & 6 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Use this factorization (do not compute the matrix A) to solve $Ax = b$.

Problem 3.4.7. *Use the $PA = LU$ factorizations of the matrices in the previous two problems to compute $det(A)$.*

3.5 Other Matrix Factorizations

As mentioned in the previous section, the idea of matrix factorization is very important and powerful when performing linear algebra computations. There are many types of matrix factorization, besides $PA = LU$, that may be preferred for certain problems. In this section we consider three other matrix factorizations: Cholesky, QR, and singular value decomposition (SVD).

3.5.1 Cholesky factorization

A matrix $A \in \mathcal{R}^{n\times n}$ is *symmetric* if $A = A^T$, which means that the entries are symmetric about the main diagonal.

Example 3.5.1. Consider the matrices

$$A = \begin{bmatrix} 3 & 0 & -1 & 5 \\ 0 & 2 & 4 & 8 \\ -1 & 4 & 1 & -2 \\ 5 & 8 & -2 & 6 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 3 & 0 & -1 & 5 \\ 1 & 2 & 4 & 8 \\ 0 & 3 & 1 & -2 \\ -2 & 1 & 1 & 6 \end{bmatrix}$$

then

$$A^T = \begin{bmatrix} 3 & 0 & -1 & 5 \\ 0 & 2 & 4 & 8 \\ -1 & 4 & 1 & -2 \\ 5 & 8 & -2 & 6 \end{bmatrix} = A \quad \text{but} \quad B^T = \begin{bmatrix} 3 & 1 & 0 & -2 \\ 0 & 2 & 3 & 1 \\ -1 & 4 & 1 & 1 \\ 5 & 8 & -2 & 6 \end{bmatrix} \neq B\,.$$

Thus A is symmetric, but B is *not* symmetric.

A matrix $A \in \mathcal{R}^{n\times n}$ is *positive definite* if $x^T Ax > 0$ for all $x \in \mathcal{R}^n$, $x \neq 0$. The special structure of a symmetric and positive definite matrix (which we will abbreviate as SPD) allows us to compute a special LU factorization, which is called the *Cholesky factorization*. Specifically, it can be shown that an $n \times n$ matrix A is SPD **if and only if** it can be factored as[1]

$$A = R^T R \quad \text{(called the Cholesky factorization)}$$

where R is an upper triangular matrix with positive entries on the diagonal. Pivoting is generally not needed to compute this factorization. The **"if and only if"** part of the above statement is important. This means that if we are given a matrix A, and the Cholesky factorization fails, then we know A is not SPD, but if it succeeds, then we know A is SPD.

To give a brief outline on how to compute a Cholesky factorization, it is perhaps easiest to begin with a small 3×3 example. That is, consider

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{12} & a_{22} & a_{23} \\ a_{13} & a_{23} & a_{33} \end{bmatrix}.$$

To find the Cholesky factorization:

- Set $R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ 0 & r_{22} & r_{23} \\ 0 & 0 & r_{33} \end{bmatrix}$
- Form the matrix $R^T R = \begin{bmatrix} r_{11}^2 & r_{11}r_{12} & r_{11}r_{13} \\ r_{11}r_{12} & r_{12}^2 + r_{22}^2 & r_{12}r_{13} + r_{22}r_{23} \\ r_{11}r_{13} & r_{12}r_{13} + r_{22}r_{23} & r_{13}^2 + r_{23}^2 + r_{33}^2 \end{bmatrix}$

[1] In some books, the Cholesky factorization is defined as $A = LL^T$ where L is lower triangular. This is the same as the notation used in this book (which better matches what is the default from computed in MATLAB), with $L = R^T$.

- Now set $A = R^T R$ and match corresponding components to solve for r_{ij}. That is,

$$\begin{aligned}
r_{11}^2 = a_{11} \quad &\Rightarrow \quad r_{11} = \sqrt{a_{11}} \\
r_{11}r_{12} = a_{12} \quad &\Rightarrow \quad r_{12} = a_{12}/r_{11} \\
r_{11}r_{13} = a_{13} \quad &\Rightarrow \quad r_{13} = a_{13}/r_{11} \\
r_{12}^2 + r_{22}^2 = a_{22} \quad &\Rightarrow \quad r_{22} = \sqrt{a_{22} - r_{12}^2} \\
r_{13}r_{12} + r_{23}r_{22} = a_{23} \quad &\Rightarrow \quad r_{23} = \left(a_{23} - r_{13}r_{12}\right)/r_{22} \\
r_{13}^2 + r_{23}^2 + r_{33}^2 = a_{33} \quad &\Rightarrow \quad r_{33} = \sqrt{a_{33} - r_{13}^2 - r_{23}^2}
\end{aligned}$$

The above process can easily be generalized for any $n \times n$ SPD matrix. We skip efficient implementation details, but make two observations. First, generally we do not need to consider pivoting when computing the Cholesky factorization of an SPD matrix, and the values inside the square root symbols are always positive. The second observation is that because the matrix is symmetric, it should not be surprising that an efficient implementation costs approximately half the number of FLOPS needed for standard $PA = LU$ factorizations.

Solving linear systems with the Cholesky factorization is essentially the same as with $PA = LU$. That is, if $A = R^T R$, then

$$Ax = b \quad \Rightarrow \quad R^T Rx = b \quad \Rightarrow \quad R^T(Rx) = b\,,$$

and so to compute x,

- use forward substitution to solve $R^T y = b$, and
- use backward substitution to solve $Rx = y$.

Problem 3.5.1. *Compute the Cholesky factorization of the matrix:*

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

Problem 3.5.2. *Compute the Cholesky factorization of the matrix:*

$$A = \begin{bmatrix} 1 & 2 & -1 \\ 2 & 8 & -4 \\ -1 & -4 & 6 \end{bmatrix}$$

Problem 3.5.3. *Compute the Cholesky factorization of*

$$A = \begin{bmatrix} 25 & 15 & -5 \\ 15 & 25 & 1 \\ -5 & 1 & 6 \end{bmatrix}$$

and use the factorization to solve $Ax = b$, where

$$b = \begin{bmatrix} -5 \\ -7 \\ 12 \end{bmatrix}.$$

3.5.2 QR factorization

A matrix $Q \in \mathcal{R}^{n\times n}$ is called *orthogonal* if the columns of Q form an orthonormal set. That is, if we write

$$Q = \left[\begin{array}{cccc} q_1 & q_2 & \cdots & q_n \end{array} \right],$$

where q_j is the jth column of Q, then

$$q_i^T q_j = \left\{ \begin{array}{lll} 1 & \text{if} & i = j \\ 0 & \text{if} & i \neq j \end{array} \right. .$$

This means that if Q is an orthogonal matrix, then

$$Q^T Q = \left[\begin{array}{c} q_1^T \\ q_2^T \\ \vdots \\ q_n^T \end{array} \right] \left[\begin{array}{cccc} q_1 & q_2 & \cdots & q_n \end{array} \right] = \left[\begin{array}{cccc} q_1^T q_1 & q_1^T q_2 & \cdots & q_1^T q_n \\ q_2^T q_1 & q_2^T q_2 & \cdots & q_2^T q_n \\ \vdots & \vdots & & \vdots \\ q_n^T q_1 & q_n^T q_2 & \cdots & q_n^T q_n \end{array} \right] = \left[\begin{array}{cccc} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 \end{array} \right].$$

That is, $Q^T Q = I$, and thus the inverse of Q is simply Q^T. This is a very nice property!

A first course on linear algebra often covers a topic called Gram-Schmidt orthonormalization, which transforms a linearly independent set of vectors into an orthonormal set. We will not review the Gram-Schmidt method in this book, but state that if it is applied to the columns of a nonsingular matrix $A \in \mathcal{R}^{n\times n}$, then it results in a matrix factorization of the form

$$A = QR,$$

where $Q \in \mathcal{R}^{n\times n}$ is an orthogonal matrix, and $R \in \mathcal{R}^{n\times n}$ is upper triangular. This is called the QR factorization of A.

We remark that if the columns of the (possibly over-determined rectangular) matrix $A \in \mathcal{R}^{m\times n}$, $m \geq n$, then it is still possible to compute a QR factorization of A. This will be discussed when we consider the topic of least squares in the chapter on curve fitting. We also remark that there are other (often better) approaches than Gram-Schmidt for computing $A = QR$ (e.g., Householder and Givens methods), but these are best left for a more advanced course on numerical linear algebra.

Computing solutions of $Ax = b$ with the QR factorization is also straight forward:

$$Ax = b \quad \Rightarrow \quad QRx = b \quad \Rightarrow \quad Rx = Q^T b,$$

and so to solve $Ax = b$,

- compute $d = Q^T b$, and
- use backward substitution to solve $Rx = d$.

Although we do not discuss algorithms for computing $A = QR$ in this book, we should note that if $A \in \mathcal{R}^{n\times n}$ is nonsingular, then an efficient implementation requires $O(n^3)$ FLOPS. But the hidden constant in the $O(\cdot)$ notation is approximately two times that for the $PA = LU$ factorization. Thus, $PA = LU$ is usually the preferred factorization for solving $n \times n$ nonsingular systems of equations. However, the QR factorization is superior for solving least squares problems.

3.5.3 Singular Value Decomposition

We end this section with arguably the most important matrix factorization. Let $A \in \mathcal{R}^{m\times n}$, $m \geq n$. Then there exist orthogonal matrices

$$\begin{array}{rcl} U & = & \left[\begin{array}{cccc} u_1 & u_2 & \cdots & u_m \end{array} \right] \in \mathcal{R}^{m\times m} \\ V & = & \left[\begin{array}{cccc} v_1 & v_2 & \cdots & v_n \end{array} \right] \in \mathcal{R}^{n\times n} \end{array}$$

and a diagonal matrix

$$\Sigma = \mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_n) = \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_n \\ & & \end{bmatrix} \in \mathcal{R}^{m \times n}$$

such that $A = U\Sigma V^T$, with $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$. The factorization, $A = U\Sigma V^T$ is called the *singular value decomposition* (SVD).

As with the QR factorization, algorithms for computing the SVD are very complicated, and best left for a more advanced course on numerical linear algebra. Here we just define the decomposition, and discuss some of its properties. We use the following notation and terminology:

- σ_i are called *singular values* of the matrix A.
- u_i, which are the columns of U, are called *left singular vectors* of the matrix A.
- v_i, which are the columns of V, are called *right singular vectors* of the matrix A.

Notice that, because V is an orthogonal matrix, we know $V^T V = I$. Thus, for the case $m \geq n$,

$$A = U\Sigma V^T \quad \Rightarrow \quad AV = U\Sigma$$

and so,

$$A \begin{bmatrix} v_1 & \cdots & v_n \end{bmatrix} = \begin{bmatrix} u_1 & \cdots & u_n & u_{n+1} & \cdots & u_m \end{bmatrix} \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_n \\ & & \end{bmatrix}.$$

The above can be written as:

$$\begin{bmatrix} Av_1 & \cdots & Av_n \end{bmatrix} = \begin{bmatrix} \sigma_1 u_1 & \cdots & \sigma_n u_n \end{bmatrix}$$

That is,

$$Av_i = \sigma_i u_i\,,\ i = 1, 2, \ldots, n$$

The SVD has the following properties:

- If $\mathrm{rank}(A) = r$, then
$$\sigma_1 \geq \cdots \geq \sigma_r > \sigma_{r+1} = \cdots = \sigma_n = 0$$
. In particular, if A is $n \times n$ and nonsingular, then all singular values are nonzero.
- If $\mathrm{rank}(A) = r$, then the nullspace of A is:
$$\mathrm{null}(A) = \mathrm{span}\{v_{r+1},\ \ldots,\ v_n\}$$
That is, $Ax = 0$ if and only if x is a linear combination of the vectors $v_{r+1},\ \ldots,\ v_n$.
- If $\mathrm{rank}(A) = r$, then the range space of A is:
$$\mathrm{range}(A) = \mathrm{span}\{u_1,\ \ldots,\ u_r\}$$

Computing solutions of $Ax = b$ with the SVD is straight forward:

$$Ax = b \quad \Rightarrow \quad U\Sigma V^T x = b \quad \Rightarrow \quad \Sigma V^T x = U^T b\,,$$

and so to solve $Ax = b$,

- compute $d = U^T b$,
- solve the diagonal system $\Sigma y = d$,
- compute $x = Vy$.

Although we do not discuss algorithms for computing the SVD in this book, we should note that if $A \in \mathcal{R}^{n\times n}$ is nonsingular, then an efficient implementation requires $O(n^3)$ FLOPS. But the hidden constant in the $O(\cdot)$ notation is approximately nine times that for the QR factorization, and eighteen times that for the $PA = LU$ factorization. Because it is so expensive, it is rarely used to solve linear systems, but it is a superior tool to analyze sensitivity of linear systems, and it finds use in important applications such as rank deficient least squares problems, principle component analysis (PCA), and even data compression.

Problem 3.5.4. *Our definition of the SVD assumes that $m \geq n$ (that is, A has at least as many rows as columns). Show that a similar definition, and hence decomposition, can be written for the case $m < n$ (that is, A has more columns than rows).*
Hint: Consider using our original definition of the SVD for A^T.

Problem 3.5.5. *Show that $A^T u_i = \sigma_i v_i$, $i = 1, 2, \ldots, n$.*

Problem 3.5.6. *Suppose $A \in \mathcal{R}^{n\times n}$. Show that $det(A) = \pm\sigma_1\sigma_2\cdots\sigma_n$.*
Hint: First show that for any orthogonal matrix U, $det(U) = \pm 1$.

Problem 3.5.7. *In this problem we consider relationships between singular values and eigenvalues. Recall from your basic linear algebra class that if $B \in \mathcal{R}^{n\times n}$, then λ is an* eigenvalue *of B if there is a nonzero vector $x \in \mathcal{R}^n$ such that*

$$Bx = \lambda x\,.$$

The vector x is called an eigenvector *of B. There are relationships between singular values/vectors and eigenvalue/vectors. To see this, assume $A \in \mathcal{R}^{m\times n}$, $m \geq n$, and $A = U\Sigma V^T$ is the SVD of A. Then from above, we know:*

$$Av_i = \sigma_i u_i \quad \text{and} \quad A^T u_i = \sigma_i v_i\,.$$

Using these relationships, show:

- *$A^T A v_i = \sigma_i^2 v_i$, and thus σ_i^2 is an eigenvalue of $A^T A$ with corresponding eigenvector v_i.*
- *$AA^T u_i = \sigma_i^2 u_i$, and thus σ_i^2 is an eigenvalue of AA^T with corresponding eigenvector u_i.*
- *If A is square and symmetric, that is $A = A^T$, then the singular values of A are the absolute values of the eigenvalues of A.*

3.6 The Accuracy of Computed Solutions

Methods for determining the accuracy of the computed solution of a linear system are discussed in more advanced courses in numerical linear algebra. However, in this section we attempt to at least qualitatively describe some of the factors affecting the accuracy.

3.6.1 Vector norms

First we need a tool to measure errors between vectors. Suppose we have a vector $\hat{x}$ that is an approximation of the vector x. How do we determine if $\hat{x}$ is a good approximation of x? It may

seem natural to consider the error vector $e = \hat{x} - x$ and determine if e is small. However, e is a vector with possibly many entries, so what does it mean to say "e is small"? To answer this question, we need the concept of *vector norm*, which uses the notation $\|\cdot\|$, and must satisfy the following properties:

1. $\|v\| \geq 0$ for all vectors $v \in \mathcal{R}^n$, and $\|v\| = 0$ if and only if $v = 0$ (that is, the vector with all zero entries),
2. $\|v + w\| \leq \|v\| + \|w\|$ for all vectors $v \in \mathcal{R}^n$ and $w \in \mathcal{R}^n$,
3. $\|cv\| = |c|\|v\|$ for all vectors $v \in \mathcal{R}^n$ and all scalars c.

There many vector norms, so sometimes we include a subscript, such as $\|\cdot\|_p$, to indicate precisely which norm we are using. Here are some examples:

- The 2-norm is the standard Euclidean length of a vector taught in multivariable calculus and linear algebra courses. Specifically, if

$$e = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{bmatrix}$$

  then we define the vector 2-norm as

$$\|e\|_2 = \sqrt{e^T e} = \sqrt{e_1^2 + e_2^2 + \cdots + e_n^2}\,.$$

- The vector 1-norm is defined as

$$\|e\|_1 = |e_1| + |e_2| + \cdots + |e_n|\,.$$

- The vector ∞-norm is defined as

$$\|e\|_\infty = \max_{1 \leq i \leq n} \{|e_i|\}\,.$$

- In general, if $1 \leq p < \infty$, then the p-norm is defined as

$$\|e\|_p = \left(\sum_{i=1}^{n} |e_i|^p \right)^{1/p}\,.$$

Although other norms are used in certain applications, we usually use the 2-norm. However, any norm gives us a single number, and if the norm of the error,

$$\|e\| = \|\hat{x} - x\|$$

is small, then we say the error is small.

We should note that "small" may be relative to the magnitude of the values in the vector x, and thus it is perhaps better to use the **relative error**,

$$\frac{\|\hat{x} - x\|}{\|x\|}$$

provided $x \neq 0$.

Example 3.6.1. Suppose

$$x = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \quad \text{and} \quad \hat{x} = \begin{bmatrix} 0.999 \\ -1.001 \end{bmatrix}$$

then

$$\hat{x} - x = \begin{bmatrix} -10^{-3} \\ -10^{-3} \end{bmatrix}$$

and so, using the 2-norm, we get

$$\|\hat{x} - x\|_2 = \sqrt{10^{-6} + 10^{-6}} = \sqrt{2} \cdot 10^{-3} \approx 0.0014142$$

and the relative error

$$\frac{\|\hat{x} - x\|_2}{\|x\|_2} = \frac{\sqrt{2} \cdot 10^{-3}}{\sqrt{2}} = 10^{-3} \, .$$

Problem 3.6.1. *Consider the previous example, and compute the relative error using the 1-norm and the ∞-norm.*

Problem 3.6.2. *If* $x = \begin{bmatrix} 1 & 2 & -3 & 0 & 1 \end{bmatrix}^T$, *compute* $\|x\|_1$, $\|x\|_2$, *and* $\|x\|_\infty$.

3.6.2 Matrix norms

The idea of vector norms can be extended to matrices. Formally, we say that $\|\cdot\|$ is a *matrix norm* if it satisfies the following properties:

1. $\|A\| \geq 0$ for all matrices $A \in \mathcal{R}^{m \times n}$, and $\|A\| = 0$ if and only if $A = 0$ (that is, the matrix with all zero entries),
2. $\|A + B\| \leq \|A\| + \|B\|$ for all matrices $A \in \mathcal{R}^{m \times n}$ and $B \in \mathcal{R}^{m \times n}$,
3. $\|cA\| = |c|\|A\|$ for all matrices $A \in \mathcal{R}^{m \times n}$ and all scalars c.

Given what we know about vector norms, it may be perhaps most natural to first consider the *Frobenius* matrix norm, which is defined as

$$\|A\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}^2} \, .$$

Other matrix norms that are often used in scientific computing are the class of *p-norms*, which are said to be *induced by* the corresponding vector norms, and are defined as

$$\|A\|_p = \max_{x \neq 0} \frac{\|Ax\|_p}{\|x\|_p} \, .$$

This is not a very useful definition for actual computations, but fortunately there are short cut formulas that can be used for three of the most popular matrix p-norms. We will not prove these, but proofs can be found in more advanced books on numerical analysis.

- The matrix 2-norm is defined as

  $$\|A\|_2 = \max_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2} \, ,$$

  but can be computed as

  $$\|A\|_2 = \sigma_1 \, ,$$

  where σ_1 is the largest singular value of A.

- The matrix 1-norm is defined as

$$\|A\|_1 = \max_{x \neq 0} \frac{\|Ax\|_1}{\|x\|_1},$$

but can be computed as

$$\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{m} |a_{ij}|,$$

that is, the maximum column sum.

- The matrix ∞-norm is defined as

$$\|A\|_\infty = \max_{x \neq 0} \frac{\|Ax\|_\infty}{\|x\|_\infty},$$

but can be computed as

$$\|A\|_\infty = \max_{1 \leq i \leq m} \sum_{j=1}^{n} |a_{ij}|,$$

that is, the maximum row sum.

The induced matrix norms satisfy two important and useful properties,

$$\|Ax\| \leq \|A\|\|x\|,$$

and

$$\|AB\| \leq \|A\|\|B\|,$$

provided the matrix multiplication of AB is defined.

Problem 3.6.3. *If*

$$A = \begin{bmatrix} 1 & -2 & 3 \\ 2 & 0 & 5 \\ -1 & 1 & -1 \\ 2 & 4 & 0 \end{bmatrix},$$

compute $\|A\|_1$, $\|A\|_\infty$, *and* $\|A\|_F$.

Problem 3.6.4. *In the case of the Frobenius norm, show that*

$$\|A\|_F^2 = \sum_{j=1}^{n} \|a_j\|_2^2 = trace(A^T A) = trace(AA^T)$$

where a_j *is the* j*-th column vector of the matrix* A*, and trace is the sum of diagonal entries of the given matrix.*

Problem 3.6.5. *In general, for an induced matrix norm, the following inequality holds:*

$$\|Ax\| \leq \|A\|\|x\|$$

In some special (important) cases, equality holds. In particular, assume Q *is an orthogonal matrix, and show that*

$$\|Qx\|_2 = \|x\|_2.$$

This property says that the Euclidean length (2-norm) of the vector x *does not change if the vector is modified by an orthogonal transformation. In this case, we say the 2-norm is invariant under orthogonal transformations.*

Problem 3.6.6. *We know that if $A = U\Sigma V^T$ then*

$$\|A\|_2 = \sigma_1 \quad \textit{(largest singular value of } A\textit{)}.$$

Show that if A is nonsingular, then

$$\|A^{-1}\|_2 = \frac{1}{\sigma_n} \quad \textit{(reciprocal of the smallest singular value of } A\textit{)}.$$

Hint: If $A = U\Sigma V^T$ is the SVD or A, what is the SVD of A^{-1}?

3.6.3 Measuring accuracy of computed solutions

Suppose we compute an approximate solution, $\hat{x}$, of $Ax = b$. How do we determine if $\hat{x}$ is a good approximation of x? If we know the exact solution, then we can simply compute the relative error,

$$\frac{\|\hat{x} - x\|}{\|x\|}$$

using any vector norm.

If we do not know the exact solution, then we may try to see if the computed solution is a good fit to the data. That is, we consider the *residual error*,

$$\|r\| = \|b - A\hat{x}\|\,,$$

or the relative residual error,

$$\frac{\|r\|}{\|b\|} = \frac{\|b - A\hat{x}\|}{\|b\|}\,.$$

We might ask the question:

> *If the relative residual error is small, does this mean $\hat{x}$ is a good approximation of the exact solution x?*

Unfortunately the answer is: Not always. Consider the following example.

Example 3.6.2. Consider the matrix

$$A = \begin{bmatrix} 0.835 & 0.667 \\ 0.333 & 0.266 \end{bmatrix}, \quad b = \begin{bmatrix} 0.168 \\ 0.067 \end{bmatrix}.$$

It is easy to verify that the exact solution to $Ax = b$ is $x = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$. Suppose we somehow compute the approximation $\hat{x} = \begin{bmatrix} 267 \\ -334 \end{bmatrix}$, which is clearly a very poor approximation of x. But the residual vector is

$$r = b - A\hat{x} \approx \begin{bmatrix} 0.001000000000019 \\ 0.000000000000003 \end{bmatrix},$$

and the relative residual error is

$$\frac{\|r\|_2}{\|b\|_2} = \frac{\|b - A\hat{x}\|_2}{\|b\|_2} \approx 0.005528913725860\,.$$

Thus in this example a small residual **does not imply** $\hat{x}$ is a good approximation of x.

Why does this happen? We can gain a little insight by examining a simple 2×2 linear system,

$$Ax = b \quad \Rightarrow \quad \begin{aligned} a_{11}x_1 + a_{12}x_2 &= b_1 \\ a_{21}x_1 + a_{22}x_2 &= b_2 \end{aligned}$$

and assume that $a_{12} \neq 0$ and $a_{22} \neq 0$. Each equation is a line, which then can be written as

$$x_2 = -\frac{a_{11}}{a_{12}}x_1 + \frac{b_1}{a_{12}} \quad \text{and} \quad x_2 = -\frac{a_{21}}{a_{22}}x_1 + \frac{b_2}{a_{22}}\,.$$

The solution of the linear system, $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ is the point where the two lines intersect. Consider the following two very different cases (it might also help to look at the plots in Figure 3.1):

> **Case 1:** The slopes, $-\frac{a_{11}}{a_{12}}$ and $-\frac{a_{21}}{a_{22}}$ are very different, e.g., the two lines are nearly perpendicular. Then small changes in b or A (e.g., due to round off error) will not dramatically change the intersection point. In this case, *the rows of A are linearly independent – very far from being linearly dependent, and A is very far from being singular.* In this case, we say the matrix A, and hence the linear system $Ax = b$, is **well-conditioned**.
>
> **Case 2:** The slopes, $-\frac{a_{11}}{a_{12}}$ and $-\frac{a_{21}}{a_{22}}$ are nearly equal, e.g., the two lines are nearly parallel. In this case, small changes in b or A (e.g., due to round off error) can cause a dramatic change the intersection point. Here, *the rows of A are nearly linearly dependent, and thus A is very close to being singular.* In this case we say that the matrix A, and hence the linear system $Ax = b$, is **ill-conditioned**.

This idea of conditioning can be extended to larger systems. In general, if the matrix is nearly singular (i.e., the columns or rows are nearly linearly dependent), then we say the matrix A is ill-conditioned.

So far our discussion of conditioning is a bit vague, and it would be nice to have a formal definition and/or way to determine if a matrix is ill-conditioned. We can do this by recalling the SVD; that is, if $A = U\Sigma V^T$, where

$$\Sigma = \text{diag}(\sigma_1, \sigma_2, \ldots, \sigma_n)\,,$$

where $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$. Recall that the rank$(A)$ is the number of nonzero singular values. That is, A is singular if the smallest singular value, $\sigma_n = 0$.

Now suppose A is nonsingular, so that $\sigma_n \neq 0$. How do we determine if A "is nearly singular"? One way is to consider the ratio of the largest and smallest singular values, $\frac{\sigma_1}{\sigma_n}$. This ratio will be ≈ 1 if the matrix is well-conditioned (e.g., as in the case of the identity matrix, I), and very large if the matrix is ill-conditioned. We know that $\|A\|_2 = \sigma_1$, and from Problem 3.6.6 we also know that $\|A^{-1}\|_2 = \frac{1}{\sigma_n}$. Thus, we can define the **condition number associated with the matrix 2-norm** to be

$$\kappa_2(A) = \|A\|_2\|A^{-1}\|_2 = \frac{\sigma_1}{\sigma_n}\,.$$

More generally, for any matrix norm, we define the condition number as

$$\kappa(A) = \|A\|\|A^{-1}\|\,.$$

Example 3.6.3. Consider the matrix

$$A = \begin{bmatrix} 0.835 & 0.667 \\ 0.333 & 0.266 \end{bmatrix}, \quad b = \begin{bmatrix} 0.168 \\ 0.067 \end{bmatrix}.$$

Using MATLAB's `svd` function, we find that $\sigma_1 \approx 1.1505e+00$ and $\sigma_2 \approx 8.6915e-07$, and hence

$$\kappa_2(A) = \|A\|_2\|A^{-1}\|_2 = \frac{\sigma_1}{\sigma_n} \approx 1.3238e+06\,.$$

This is a very large number, and so we conclude that A is ill-conditioned.

Let us now return back to the question:

If the relative residual error is small, does this mean $\hat{x}$ is a good approximation of the exact solution x?

To see how the residual relates to the relative error, suppose A is a nonsingular matrix and $\hat{x}$ is an approximate solution of $Ax = b$. Then using an induced matrix norm (e.g., 2-norm), we obtain:

- $r = b - A\hat{x} = Ax - A\hat{x} = A(x - \hat{x})$, which means

$$x - \hat{x} = A^{-1}r\,. \tag{3.2}$$

- If we take norms on both sides of (3.2), we obtain

$$\|x - \hat{x}\| = \|A^{-1}r\| \leq \|A^{-1}\|\|r\|\,. \tag{3.3}$$

- Next observe that $Ax = b$ implies $\|b\| = \|Ax\| \leq \|A\|\|x\|$, and so

$$\frac{1}{\|x\|} \leq \frac{\|A\|}{\|b\|}\,. \tag{3.4}$$

- Putting together the inequalities (3.3) and (3.4), we obtain:

$$\frac{\|x - \hat{x}\|}{\|x\|} \leq \|A\|\|A^{-1}\|\frac{\|r\|}{\|b\|} = \kappa(A)\|\frac{\|r\|}{\|b\|}\,. \tag{3.5}$$

The result in (3.5) is important! It tells us:

- If the matrix A is well conditioned, e.g. $\kappa(A) \approx 1$, and if the relative residual is small, then we can be sure to have an accurate solution.
- However, if A is ill-conditioned (e.g., $\kappa(A)$ is very large), then the relative error can be large even if the relative residual is small. Notice that we **cannot** say "will be large" because the result is given in terms of an upper bound. But the fact that it "can be large" is important to know, and should be taken into account when attempting to solve ill-conditioned linear systems.

We conclude this subsection with the following remarks. The two categories, **well-conditioned** and **ill-conditioned** are separated by a grey area. That is, while we can say that a matrix with condition number $\kappa(A)$ in the range of 1 to 100 would be considered well-conditioned, and a condition number $\kappa(A) > 10^8$ is considered ill-conditioned, there is a large "grey" area between these extremes, and it depends on machine precision. Although we cannot get rid of this grey area, we can say that if the condition number of A is about 10^p and the machine epsilon, ϵ, is about 10^{-s} then the solution of the linear system $Ax = b$ *may* have no more than about $s - p$ decimal digits accurate. Recall that in SP arithmetic, $s \approx 7$, and in DP arithmetic, $s \approx 16$.

We should keep in mind that a well-conditioned linear system has the property that *all small changes* in the matrix A and the right hand side b lead to a small change in the solution of $Ax = b$, and that an ill-conditioned linear system has the property that *some small changes* in the matrix A and/or the right hand side b can lead to a large change in the solution of $Ax = b$.

3.6.4 Backward error

It is important to emphasize that the concepts of *ill-conditioned* and *well-conditioned* are related to the problem $Ax = b$, and not to the algorithm used to solve the system. Even the very best algorithms cannot be expected to compute accurate solutions of extremely ill-conditioned problems. However, if a problem is well-conditioned, than the algorithms we use should compute accurate solutions. This topic is typically referred to as *algorithm stability*.

To understand the idea of algorithm stability, consider a nonsingular linear system $Ax = b$ of order n. The solution process involves two steps: a matrix factorization, and solves with simple methods, such as forward and backward substitution. Because of roundoff errors, we cannot expect the computed solution $\hat{x}$ to be the exact solution of $Ax = b$. However, theoretically there is a system $\hat{A}$ and $\hat{b}$ for which $\hat{x}$ is the exact solution; that is, $\hat{A}\hat{x} = \hat{b}$ (in some sense $\hat{A}$ and $\hat{b}$ are obtained by starting with $\hat{x}$ and running the steps of the algorithm in a "backwards" direction). An algorithm is called *backward stable* if it can be shown that $\hat{x}$ is the exact solution of $\hat{A}\hat{x} = \hat{b}$, where

$$\|\hat{A} - A\| \quad \text{and} \quad \|\hat{b} - b\| \tag{3.6}$$

are small.

To prove an algorithm is backward stable requires establishing bounds for the error norms in (3.6), and showing that the bounds are small. This is a relatively advanced topic, so we do not provide any results in this book, but interested readers can find the results in many excellent books on numerical linear algebra and matrix computations. However, it is important for readers of this book to understand that one of the nice properties of GEPP is that, generally, $\hat{A}$ is "close to" A and $\hat{b}$ is "close to" b, and thus GEPP is generally[2] **backward stable**. When GEPP is backward stable, the computed solution $\hat{x}$ of $Ax = b$ is the exact solution of $\hat{A}\hat{x} = \hat{b}$ where $\hat{A}$ is close to A and $\hat{b}$ is close to b.

We also mention that the best algorithms for Cholesky, QR and SVD (e.g., the ones used by MATLAB) are backward stable.

How do the concepts of well-conditioned and ill-conditioned linear systems relate to the concept of algorithm stability? *Backward error analysis* shows that the approximate solution of $Ax = b$ computed by Cholesky, QR and SVD, and usually for GEPP, are exact solutions of a related linear system $\hat{A}\hat{x} = \hat{b}$, where $\hat{A}$ is close to A and $\hat{b}$ is close to b. Thus, if $Ax = b$ is well-conditioned, it follows that the computed solution is accurate. On the other hand, if $Ax = b$ is ill-conditioned, even if $\hat{A}$ is close to A and $\hat{b}$ is close to b, these small differences *may* lead to a large difference between the exact solution of $Ax = b$ and the the computed solution $\hat{x}$. In summary, when using a backward stable algorithm to solve a well-conditioned linear system, the computed solution is accurate. On the other hand, if the linear system is ill-conditioned, then even with the most stable algorithms, the computed solution *may not* be accurate.

Problem 3.6.7. *This example illustrates* how not *to determine conditioning of a linear system. We know that* $\det(A) = 0$ *when the linear system* $Ax = b$ *is singular. So, we might assume that the magnitude of* $\det(A)$ *might be a good indicator of how close the matrix* A *is to a singular matrix. Unfortunately, this is not always the case. Consider, for example, the two linear systems:*

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad \textit{and} \quad \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 0.5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 1.0 \\ 1.5 \end{bmatrix}$$

where the second is obtained from the first by multiplying each of its equations by 0.5.

(a) Find and compare the determinant of the two coefficient matrices.

(b) Show that GEPP produces identical solutions for both linear systems (this should be trivial because each linear system is diagonal, so no exchanges or eliminations need be performed).

Thus, this problem shows that the magnitude of the determinant is not necessarily a good indicator of how close a coefficient matrix is to the nearest singular matrix.

[2] We use the term *generally* because the error bounds depend on something called a *growth factor*, which is generally small (in which case the algorithm is backward stable), but there are unusual cases where the growth factor can be large. This topic is studied in more advanced courses on numerical linear algebra.

Problem 3.6.8. *Consider the linear system of equations of order 2:*

$$\begin{aligned} 1000x_1 + 999x_2 &= 1999 \\ 999x_1 + 998x_2 &= 1997 \end{aligned}$$

(a) Show that the exact solution of this linear system is $x_1 = x_2 = 1$.

(b) For the nearby approximate solution $x_1 = 1.01$, $x_2 = 0.99$, compute the residual vector and its norm (use the 2-norm).

(c) For the very inaccurate approximate solution $x_1 = 20.97$, $x_2 = -18.99$, compute the residual vector and its norm (use the 2-norm). How does this compare to part (b)?

(d) What can you conclude about the matrix A? Can you confirm your conclusion?

Problem 3.6.9. *Consider the linear system of equation of order 2:*

$$\begin{aligned} 0.780x_1 + 0.563x_2 &= 0.217 \\ 0.913x_1 + 0.659x_2 &= 0.254 \end{aligned}$$

with exact solution $x_1 = 1$, $x_2 = -1$. Consider two approximate solutions: first $x_1 = 0.999$, $x_2 = -1.001$, and second $x_1 = 0.341$, $x_2 = -0.087$. Compute the residuals for these approximate solutions. Is the accuracy of the approximate solutions reflected in the size of the residuals? Is the linear system ill–conditioned?

Problem 3.6.10. *Consider a linear system $Ax = b$. When this linear system is placed into the computer's memory, say by reading the coefficient matrix and right hand side from a data file, the entries of A and b must be rounded to floating–point numbers. If the linear system is well–conditioned, and the solution of this "rounded" linear system is computed exactly, will this solution be accurate? Answer the same question but assuming that the linear system $Ax = b$ is ill–conditioned.*

3.7 Matlab Notes

Software is available for solving linear systems where the coefficient matrices have a variety of structures and properties. We restrict our discussion to "dense" systems of linear equations. (A "dense" system is one where all the coefficients are treated as non-zero values. So, the matrix is considered to have no special structure.) Most of today's best software for solving "dense" systems of linear equations, including that found in MATLAB, was developed in the *LAPACK* project. We describe the main tools (i.e., the backslash operator and the `linsolve` function) provided by MATLAB for solving dense linear systems. These, and other useful built-in MATLAB functions that are relevant to the topics of this chapter, and which will be discussed in this section, include:

`linsolve`	used to solve linear systems $Ax = b$
\ (backslash)	used to solve linear systems $Ax = b$
`lu`	used to compute $A = LU$ and $PA = LU$ factorizations
`cond`	used to compute condition number of a matrix
`triu`	used to get upper triangular part of a matrix
`tril`	used to get lower triangular part of a matrix
`diag`	used to get diagonal part of a matrix, or to make a diagonal matrix

First however we develop MATLAB implementations of some of the algorithms discussed in this chapter. These examples build on the introduction in Chapter 1, and are designed to introduce useful MATLAB commands and to teach proper MATLAB programming techniques.

3.7.1 Diagonal Linear Systems

Consider a simple diagonal linear system

$$\begin{array}{c} a_{11}x_1 = b_1 \\ a_{22}x_2 = b_2 \\ \vdots \\ a_{nn}x_n = b_n \end{array} \quad \Leftrightarrow \quad \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

If the diagonal entries, a_{ii}, are all nonzero, it is trivial to solve for x_i:

$$\begin{array}{c} x_1 = b_1/a_{11} \\ x_2 = b_2/a_{22} \\ \vdots \\ x_n = b_n/a_{nn} \end{array}$$

To write a MATLAB function to solve a diagonal system, we must decide what quantities should be specified as input, and what as output. For example, we could input the matrix A and right hand side vector b, and output the solution of $Ax = b$, as in the code:

```
function x = DiagSolve1(A,b)
%
%      x = DiagSolve1(A, b);
%
% Solve Ax=b, where A is an n-by-n diagonal matrix.
%
n = length(b); x = zeros(n,1);
for i = 1:n
  if A(i,i) == 0
    error('Input matrix is singular')
  end
  x(i) = b(i) / A(i,i);
end
```

We use the MATLAB function `length` to determine the dimension of the linear system, assumed the same as the length of the right hand side vector, and we use the `error` function to print an error message in the command window, and terminate the computation, if the matrix is singular. We can shorten this code, and make it more efficient by using array operations:

```
function x = DiagSolve2(A, b)
%
%     x = DiagSolve2(A, b);
%
% Solve Ax=b, where A is an n-by-n diagonal matrix.
%
d = diag(A);
if any(d == 0)
  error('Input matrix is singular')
end
x = b ./ d;
```

In `DiagSolve2`, we use MATLAB's `diag` function to extract the diagonal entries of `A`, and store them in a column vector `d`. If there is at least one 0 entry in the vector `d`, then `any(d == 0)` returns true, otherwise it returns false. If all diagonal entries are nonzero, the solution is computed using the element-wise division operation, `./` In most cases, if we know the matrix is diagonal, then we can substantially reduce memory requirements by using only a single vector (not a matrix) to store the diagonal elements, as follows:

```
function x = DiagSolve3(d, b)
%
%     x = DiagSolve3(d, b);
%
% Solve Ax=b, where A is an n-by-n diagonal matrix.
%
% Input: d = vector containing diagonal entries of A
%        b = right hand side vector
%
if any(d == 0)
  error('Diagonal matrix defined by input is singular')
end
x = b ./ d;
```

In the algorithms `DiagSolve2` and `DiagSolve3` we do not check if `length(b)` is equal to `length(d)`. If they are not equal, MATLAB will report an array dimension error.

Problem 3.7.1. *Implement the functions* `DiagSolve1`, `DiagSolve2`, *and* `DiagSolve3`, *and use them to solve the linear system in Fig. 3.2(a).*

Problem 3.7.2. *Consider the* MATLAB *commands:*

```
n = 200:200:1000;, t = zeros(length(n), 3);
for i = 1:length(n)
  d = rand(n(i),1);, A = diag(d);, x = ones(n(i),1);, b = A*x;
  tic, x1 = DiagSolve1(A,b);, t(i,1) = toc;
  tic, x2 = DiagSolve2(A,b);, t(i,2) = toc;
  tic, x3 = DiagSolve3(d,b);, t(i,3) = toc;
end
disp('-------------------------------------------')
disp(' Timings for DiagSolve functions')
disp(' n DiagSolve1 DiagSolve2 DiagSolve3')
disp('-------------------------------------------')
for i = 1:length(n)
  disp(sprintf('%4d %9.3e %9.3e %9.3e', n(i), t(i,1), t(i,2), t(i,3)))
end
```

Using the MATLAB `help` *and/or* `doc` *commands write a brief explanation of what happens when these commands are executed. Write a script M–file implementing the commands and run the script. Check for consistency by running the script sufficient times so that you have confidence in your results. Describe what you observe from the computed results.*

3.7.2 Triangular Linear Systems

We describe MATLAB implementations of the forward substitution algorithms for lower triangular linear systems. Implementations of backward substitution are left as exercises.

Consider the lower triangular linear system

$$\begin{array}{ll} a_{11}x_1 & = b_1 \\ a_{21}x_1 + a_{22}x_2 & = b_2 \\ \vdots & \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n & = b_n \end{array} \quad \Leftrightarrow \quad \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

We first develop a MATLAB implementation to solve this lower triangular system using the pseudocode for the row-oriented version of forward substitution given in Fig. 3.5:

```
function x = LowerSolve0(A, b)
%
%     x = LowerSolve0(A, b);
%
% Solve Ax=b, where A is an n-by-n lower triangular matrix,
% using row-oriented forward substitution.
%
n = length(b);, x = zeros(n,1);
for i = 1:n
  for j = 1:i-1
    b(i) = b(i) - A(i,j)*x(j);
  end
  x(i) = b(i) / A(i,i);
end
```

This implementation can be improved in several ways. As with the diagonal solve functions, we should include a statement that checks to see if `A(i,i)` is zero. Also, the innermost loop can be replaced with a single MATLAB array operation. Observe that we can write the algorithm as:

for $i = 1 : n$
  $x_i = (b_i - (a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{i,i-1}x_{i-1}))/a_{ii}$
end

Using linear algebra notation, we can write this as:

for $i = 1 : n$

$$x_i = \left(b_i - \begin{bmatrix} a_{i1} & a_{i2} & \cdots & a_{i,i-1} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{i-1} \end{bmatrix} \right) \Big/ a_{ii}$$

end

Recall that, in MATLAB, we can specify entries in a matrix or vector using colon notation. That is,

$$x(1 : i-1) = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{i-1} \end{bmatrix} \quad \text{and} \quad A(i, 1 : i-1) = \begin{bmatrix} a_{i,1} & a_{i,2} & \cdots & a_{i,i-1} \end{bmatrix}.$$

So, using MATLAB notation, the algorithm for row-oriented forward substitution can be written:

for $i = 1 : n$
  $x(i) = (b(i) - A(i, 1 : i-1) * x(1 : i-1))/ A(i, i)$
end

When $i = 1$, MATLAB considers $A(i, 1 : i-1)$ and $x(1 : i-1)$ to be "empty" matrices, and the computation $A(i, 1 : i-1) * x(1 : i-1)$ gives 0. Thus, the algorithm simply computes $x(1) = b(1)/A(1, 1)$, as it should. To summarize, a MATLAB function to solve a lower triangular system using row oriented forward substitution could be written:

```
function x = LowerSolve1(A, b)
%
%     x = LowerSolve1(A, b);
%
% Solve Ax=b, where A is an n-by-n lower triangular matrix,
% using row-oriented forward substitution.
%
if any(diag(A) == 0)
  error('Input matrix is singular')
end
n = length(b);, x = zeros(n,1);
for i = 1:n
  x(i) = (b(i) - A(i,1:i-1)*x(1:i-1)) / A(i,i);
end
```

Implementation of column-oriented forward substitution is similar. However, because x_j is computed before b_i is updated, it is not possible to combine the two steps, as in the function `LowerSolve1`. A MATLAB implementation of column-oriented forward substitution could be written:

```
function x = LowerSolve2(A, b)
%
%     x = LowerSolve2(A, b);
%
% Solve Ax=b, where A is an n-by-n lower triangular matrix,
% using column-oriented forward substitution.
%
if any(diag(A) == 0)
  error('Input matrix is singular')
end
n = length(b);, x = zeros(n,1);
for j = 1:n
  x(j) = b(j) / A(j,j);
  b(j+1:n) = b(j+1:n) - A(j+1:n,j)*x(j);
end
```

What is computed by the statement `b(j+1:n) = b(j+1:n) - A(j+1:n,j)*x(j)` when $j = n$? Because there are only n entries in the vectors, MATLAB recognizes `b(n+1:n)` and `A(n+1:n,n)` to be "empty matrices", and skips this part of the computation.

Problem 3.7.3. *Implement the functions* `LowerSolve1` *and* `LowerSolve2`*, and use them to solve the linear system in Fig. 3.2(b).*

Problem 3.7.4. *Consider the following* MATLAB *commands:*

```
n = 200:200:1000;, t = zeros(length(n), 2);
for i = 1:length(n)
  A = tril(rand(n(i)));, x = ones(n(i),1);, b = A*x;
  tic, x1 = LowerSolve1(A,b);, t(i,1) = toc;
  tic, x2 = LowerSolve2(A,b);, t(i,2) = toc;
end
disp('-----------------------------------')
disp(' Timings for LowerSolve functions')
disp(' n LowerSolve1 LowerSolve2 ')
disp('-----------------------------------')
for i = 1:length(n)
  disp(sprintf('%4d %9.3e %9.3e', n(i), t(i,1), t(i,2)))
end
```

Using the MATLAB `help` *and/or* `doc` *commands write a brief explanation of what happens when these commands are executed. Write a script M–file implementing the commands, run the script, and describe what you observe from the computed results.*

Problem 3.7.5. *Write a* MATLAB *function that solves an upper triangular linear system using row-oriented backward substitution. Test your code using the linear system in Fig. 3.2(c).*

Problem 3.7.6. *Write a* MATLAB *function that solves an upper triangular linear system using column-oriented backward substitution. Test your code using the linear system in Fig. 3.2(c).*

Problem 3.7.7. *Write a script M–file that compares timings using row-oriented and column-oriented backward substitution to solve an upper triangular linear systems. Use the code in Problem 3.7.4 as a template.*

3.7.3 Gaussian Elimination

Next, we consider a MATLAB implementation of Gaussian elimination, using the pseudocode given in Fig. 3.7. First, we explain how to implement the various steps during the k^{th} stage of the algorithm.

- Consider the search for the largest entry in the pivot column. Instead of using a loop, we can use the built-in `max` function. For example, if we use the command

  ```
  [piv, i] = max(abs(A(k:n,k)));
  ```

  then, after execution of this command, `piv` contains the largest entry (in magnitude) in the vector `A(k:n,k)`, and `i` is its location in the vector. Note that if `i = 1`, then the index `p` in Fig. 3.7 is `p = k` and, in general,

  ```
  p = i + k - 1;
  ```

- Once the pivot row, `p`, is known, then the p^{th} row of A is switched with the k^{th} row, and the corresponding entries of b must be switched. This may be implemented using MATLAB's array indexing capabilities:

  ```
  A([k,p],k:n) = A([p,k],k:n);, b([k,p]) = b([p,k]);
  ```

  We might visualize these two statements as

  $$\begin{bmatrix} a_{kk} & a_{k,k+1} & \cdots & a_{kn} \\ a_{pk} & a_{p,k+1} & \cdots & a_{pn} \end{bmatrix} := \begin{bmatrix} a_{pk} & a_{p,k+1} & \cdots & a_{pn} \\ a_{kk} & a_{k,k+1} & \cdots & a_{kn} \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} b_k \\ b_p \end{bmatrix} := \begin{bmatrix} b_p \\ b_k \end{bmatrix}$$

  That is, `b([k,p]) = b([p,k])` instructs MATLAB to replace `b(k)` with the original `b(p)`, and to replace `b(p)` with original `b(k)`. MATLAB's internal memory manager makes appropriate copies of the data so that the original entries are not overwritten before the assignments are completed. Similarly, `A([k,p],k:n) = A([p,k],k:n)` instructs MATLAB to replace the rows specified on the left with the original rows specified on the right.

- The elimination step is straightforward; compute the multipliers, which we store in the strictly lower triangular part of A:

  ```
  A(k+1:n,k) = A(k+1:n,k) / A(k,k);
  ```

  and use array operations to perform the elimination:

  ```
  for i = k+1:n
    A(i,k+1:n) = A(i,k+1:n) - A(i,k)*A(k,k+1:n);, b(i) = b(i) - A(i,k)*b(k);
  end
  ```

- Using array operations, the backward substitution step can be implemented:

  ```
  for i = n:-1:1
    x(i) = (b(i) - A(i,i+1:n)*x(i+1:n)) / A(i,i);
  end
  ```

  The statement `for i = n:-1:1` indicates that the loop runs over values $i = n, n-1, \ldots, 1$; that is, it runs from `i=n` in steps of `-1` until it reaches `i=1`.

- How do we implement a check for singularity? Due to roundoff errors, it is unlikely that any pivots a_{kk} will be exactly zero, and so a statement `if A(k,k) == 0` will usually miss detecting singularity. An alternative is to check if a_{kk} is small in magnitude compared to, say, the largest entry in magnitude in the matrix A:

```
if abs(A(k,k)) < tol
```

where `tol` is computed in an initialization step:

```
tol = eps * max(abs(A(:)));
```

Here, the command `A(:)` reshapes the matrix A into one long vector, and `max(abs(A(:)))` finds the largest entry. Of course, this test also traps matrices which are close to but not exactly singular, which is usually reasonable as by definition they tend to be ill–conditioned.

- Finally, certain other initializations are needed: the dimension, n, space for the solution vector, and for the multipliers. We use the `length`, `zeros`, and `eye` functions:

```
n = length(b);, x = zeros(n,1);, m = eye(n);
```

```
function x = gepp(A, b)
%
% Solves Ax=b using Gaussian elimination with partial pivoting
% by rows for size. Initializations:
%
n = length(b); x = zeros(n,1);
tol = eps*max(A(:));
%
% Loop for stages k = 1, 2, ..., n-1
%
for k = 1:n-1
  %
  % Search for pivot entry:
  %
  [piv, psub] = max(abs(A(k:n,k)));, p = psub + k - 1;
  %
  % Exchange current row, k, with pivot row, p:
  %
  A([k,p],k:n) = A([p,k],k:n);, b([k,p]) = b([p,k]);
  %
  % Check to see if A is singular:
  %
  if abs(A(k,k)) < tol
    error('Linear system appears to be singular')
  end
  %
  % Perform the elimination step - row-oriented:
  %
  A(k+1:n,k) = A(k+1:n,k) / A(k,k);
  for i = k+1:n
    A(i,k+1:n) = A(i,k+1:n) - A(i,k)*A(k,k+1:n);,  b(i) = b(i) - A(i,k)*b(k);
  end
end
%
% Check to see if A is singular:
%
if abs(A(n,n)) < tol
  error('Linear system appears to be singular')
end
%
% Solve the upper triangular system by row-oriented backward substitution:
%
for i = n:-1:1
  x(i) = (b(i) - A(i,i+1:n)*x(i+1:n)) / A(i,i);
end
```

The function gepp above implements a function that uses GEPP to solve $Ax = b$.

Problem 3.7.8. *Implement the function* gepp*, and use it to solve the linear systems given in problems 3.3.2 and 3.3.3.*

Problem 3.7.9. *Test* `gepp` *using the linear system*

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}.$$

Explain your results.

Problem 3.7.10. *Modify* `gepp`*, replacing the statements* `if abs(A(k,k)) < tol` *and* `if abs(A(n,n)) < tol` *with, respectively,* `if A(k,k) == 0` *and* `if A(n,n) == 0`*. Solve the linear system given in Problem 3.7.9. Why are your results different?*

Problem 3.7.11. *The built-in* MATLAB *function* `hilb` *constructs the Hilbert matrix, whose* (i, j) *entry is* $1/(i + j - 1)$*. Write a script M–file that constructs a series of test problems of the form:*

```
A = hilb(n);, x_true = ones(n,1);, b = A*x_true;
```

The script should use `gepp` *to solve the resulting linear systems, and print a table of results with the following information:*

```
-------------------------------------------------------
  n      error      residual      condition number
-------------------------------------------------------
```

where

- `error` = *relative error* = `norm(x_true - x)/norm(x_true)`
- `residual` = *relative residual error* = `norm(b - A*x)/norm(b)`
- `condition number` = *measure of conditioning* = `cond(A)`

Print a table for $n = 5, 6, \ldots, 13$*. Are the computed residual errors small? What about the relative errors? Is the size of the residual error related to the condition number? What about the relative error? Now run the script with* $n \geq 14$*. What do you observe?*

Problem 3.7.12. *Rewrite the function* `gepp` *so that it uses no array operations. Compare the efficiency (for example, using* `tic` *and* `toc`*) of your function with* `gepp` *on matrices of dimensions* $n = 100, 200, \ldots, 1000$*.*

3.7.4 Built-in Matlab Tools for Linear Systems

An advantage of using a powerful scientific computing environment like MATLAB is that we do not need to write our own implementations of standard algorithms, like Gaussian elimination and triangular solves. MATLAB provides two powerful tools for solving linear systems:

- The *backslash* operator: \
  Given a matrix A and vector b, \ can be used to solve $Ax = b$ with the single command:

  ```
  x = A \ b;
  ```

  MATLAB first checks if the matrix A has a special structure, including diagonal, upper triangular, and lower triangular. If a special structure is recognized (for example, upper triangular), then a special method (for example, column-oriented backward substitution) is used to solve $Ax = b$. If a special structure is not recognized then Gaussian elimination with partial pivoting is used to solve $Ax = b$. During the process of solving the linear system, MATLAB estimates the *reciprocal* of the condition number of A. If A is ill–conditioned, then a warning message is printed along with the estimate, `RCOND`, of the reciprocal of the condition number.

- The function `linsolve`.
  If we know a-priori that A has a special structure recognizable by MATLAB, then we can improve efficiency by avoiding checks on the matrix, and skipping directly to the special solver. This can be especially helpful if, for example, it is known that A is upper triangular. A-priori information on the structure of A can be provided to MATLAB using the `linsolve` function. For more information, see `help linsolve` or `doc linsolve`.

Because the backslash operator is so powerful, we use it almost exclusively to solve general linear systems.

We can compute explicitly the $PA = LU$ factorization using the `lu` function:

```
[L, U, P] = lu(A)
```

Given this factorization, and a vector b, we could solve the $Ax = b$ using the statements:

```
d = P * b;, y = L \ d;, x = U \ y;
```

These statements could be combined into one instruction:

```
x = U \ ( L \ (P * b) );
```

Thus, given a matrix A and vector b we could solve the linear system $Ax = b$ as follows:

```
[L, U, P] = lu(A);
x = U \ ( L \ (P * b) );
```

Note, MATLAB follows the rules of operator precedence thus backslash and `*` are of equal precedence. With operators of equal precedence MATLAB works from the left. So, without parentheses the instruction

```
x = U \ L \ P * b ;
```

would be interpreted as

```
x = ((U \ L) \ P) * b;
```

The cost and accuracy of this approach is essentially the same as using the backslash operator. So when would we prefer to explicitly compute the $PA = LU$ factorization? One situation is when we need to solve several linear systems with the same coefficient matrix, but different right hand side vectors. For large systems it is far more expensive to compute the $PA = LU$ factorization than it is to use forward and backward substitution to solve corresponding lower and upper triangular systems. Thus, if we can compute the factorization just once, and use it for the various different right hand side vectors, we can make a substantial savings. This is illustrated in problem 3.7.18, which involves a relatively small linear system.

Problem 3.7.13. *Use the backslash operator to solve the systems given in problems 3.3.2 and 3.3.3.*

Problem 3.7.14. *Use the backslash operator to solve the linear system*

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}.$$

Explain your results.

Problem 3.7.15. *Repeat problem 3.7.11 using the backslash operator in addition to* `gepp`.

Problem 3.7.16. *Consider the linear system defined by the following* MATLAB *commands:*

```
A = eye(500) + triu(rand(500));, x = ones(500,1);, b = A * x;
```

Does this matrix have a special structure? Suppose we slightly perturb the entries in A:

```
C = A + eps*rand(500);
```

Does the matrix C have a special structure? Execute the following MATLAB *commands:*

```
tic, x1 = A \ b;, toc
tic, x2 = C \ b;, toc
tic, x3 = triu(C) \ b;, toc
```

Are the solutions `x1`*,* `x2` *and* `x3` *good approximations to the exact solution,* `x = ones(500,1)`*? What do you observe about the time required to solve each of the linear systems?*

Problem 3.7.17. *Use the* MATLAB `lu` *function to compute the* $PA = LU$ *factorization of each of the matrices given in Problem 3.4.5.*

Problem 3.7.18. *Create a script M–file containing the following* MATLAB *statements:*

```
n = 50;, A = rand(n);
tic
for k = 1:n
  b = rand(n,1);, x = A \ b;
end
toc

tic
[L, U, P] = lu(A);
for k = 1:n
  b = rand(n,1);, x = U \ ( L \ (P * b) );
end
toc
```

The dimension of the problem is set to $n = 50$*. Experiment with other values of* n*, such as* $n = 100, 150, 200$*. What do you observe?*