Chapter 3

Solution of Linear Systems

In this chapter we study algorithms for possibly the most commonly occurring problem in scientific computing, the solution of linear systems of equations. We start with a review of linear systems and the conditions under which they have a solution. Then we discuss how to solve some simple linear systems before we move on to an algorithm for the general case which involves reducing the general system to solving a simple system. Finally, we discuss some of the pitfalls in linear system solution.

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 an assignment of values to $x_1, x_2, \ldots, x_n$ that satisfies all n equations simultaneously. In matrix-vector notation, the linear system is represented as

$$Ax = b$$

where

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

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \ldots \text{the unknown, or solution } n\text{-vector}$$

$$b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \quad \ldots \text{the right hand side } n\text{-vector}$$

The coefficients $a_{i,j}$ and the right hand side values b_i are real numbers. We say that a **solution** of the linear system $Ax = b$ of order n is an assignment of values to the entries of the solution vector x

that satisfies all n equations in $Ax = b$ simultaneously. 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 of this linear system is $Ax = b$ where

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

As will be verified later, this linear system has precisely one solution, given by $x_1 = 1$, $x_2 = 1$ and $x_3 = 1$, so the solution vector is

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

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 $x = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$?*

3.1.1 Singular and Nonsingular Systems

Consider the linear system of Example 3.1.1. For this linear system, the solution set of each one of the 3 equations can be portrayed as a plane in 3-space. So, the solution set of this linear system is the set of points that lie simultaneously on all 3 of these planes, i.e., the points where the 3 planes intersect. Note that 3 planes in 3-space can intersect at

- no point, e.g., when they are parallel distinct planes,
- one point, e.g., two planes intersect in a line, and this line penetrates the third plane at one point, or
- an infinite number of points, e.g., all three planes contain a common line, or the three planes coincide.

We conclude that the solution set of this linear system has either no elements, one element, or an infinite number of distinct elements. In terms of the linear system, it has either no solution, one solution, or an infinite number of distinct solutions.

This conclusion holds not only for linear systems of order 3, but for linear systems of any order. Specifically, **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 and this matter is considered in a first linear algebra course. (Commonly used synonyms in a linear algebra context: invertible for nonsingular and non-invertible for singular.)

Whether a linear system $Ax = b$ of order n is singular or nonsingular depends solely on properties of its coefficient matrix A. So, we say the matrix A is nonsingular if the linear system $Ax = b$ is nonsingular, and the matrix A is singular if the linear system $Ax = b$ is singular.

Among the many tests that can be applied to determine if a matrix is nonsingular, we mention the following. **The matrix A of order n is nonsingular if**:

- the determinant $\det(A)$ of the coefficient matrix A is nonzero; in Example 3.1.1 above, $\det(A) = 11$,
- the rows of the coefficient matrix A are linearly independent, i.e., no one row of A can be written as a sum of multiples of the other rows of A, or
- the columns of the coefficient matrix A are linearly independent, i.e., no one column of A can be written as a sum of multiples of the other columns of A.

Problem 3.1.2. *Consider any linear system of algebraic equations of order* 2. *The solution of each equation can be portrayed as a straight line in a* 2*-dimensional plane. Describe geometrically how* 2 *lines can intersect in the plane. Why must two lines that intersect at two distinct points therefore intersect at an infinite number of points? Explain how you would conclude that a linear system of order* 2 *that has* 2 *distinct solutions also has 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. Give yet another example of one equation in one unknown that has an infinite number of solutions.*

Problem 3.1.4. *Show that the matrix*

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

is singular by demonstrating that (a) the third row can be written as a sum of multiples of the first and second rows, and (b) the third column can be written as a sum of multiples of the first and second columns.

Problem 3.1.5. *Show that the matrix*

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

is singular by demonstrating that (a) the second row can be written as a sum of multiples of the first and third rows, and (b) the second column can be written as a sum of multiples of the first and third columns.

Problem 3.1.6. *Show that the matrix*

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

is singular by demonstrating that (a) the third row can be written as a sum of multiples of the first and second rows, and (b) the third column can be written as a sum of multiples of the first and second columns.

3.1.2 Simply Solved Linear Systems

Some linear systems are easy to solve. For example, consider the linear systems of order 3 displayed in Fig. 3.1. By design, the solution of each of these linear systems is $x_1 = 1$, $x_2 = 2$ and $x_3 = 3$. To explain why each of these linear systems is easy to solve we first name the structure exhibited in these linear systems.

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

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

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

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

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

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

Figure 3.1: Simply Solved Linear Systems

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

The locations of the strictly lower triangular, diagonal, and strictly upper triangular entries of A are illustrated in Fig. 3.2. The lower triangular entries are composed of the strictly lower triangular and diagonal entries, as illustrated in Fig. 3.3. Similarly, the upper triangular entries are composed of the strictly upper triangular and diagonal entries.

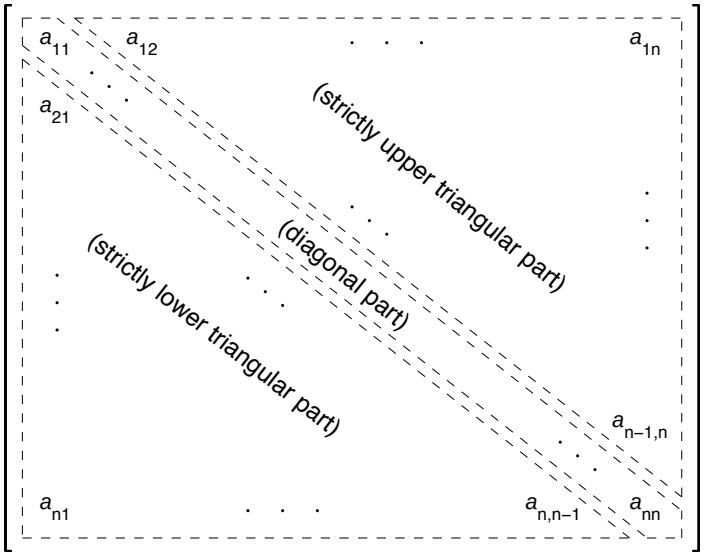

Figure 3.2: Illustration of strict triangular and diagonal matrix entries.

Problem 3.1.7. *Let A be a matrix of order n. Show that A has a total of n^2 entries. Show that $n^2 - n = n(n-1)$ entries lie off the diagonal, and show that each strictly triangular portion of A has $\dfrac{n(n-1)}{2}$ entries.*

Diagonal Linear Systems

A matrix A of order n is **diagonal** if all its nonzero entries are on its diagonal. (Note that 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.

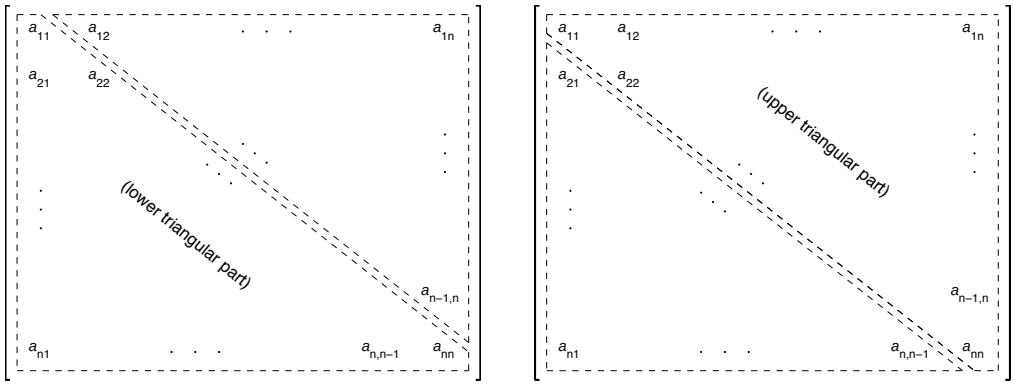

Figure 3.3: Illustration of the lower triangular and upper triangular parts of a matrix.

Solving a diagonal linear system of order n, like that in Fig. 3.1(a), is easy because each equation determines the value of one unknown, provided that each diagonal entry is nonzero. So, the first equation determines the value of x_1, the second equation determines the value of x_2, etc.

Example 3.1.2. In the case in Fig. 3.1(a) the solution is $x_1 = \frac{-1}{(-1)} = 1$, $x_2 = \frac{6}{3} = 2$ and $x_3 = \frac{-15}{(-5)} = 3$

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 of order n, like that in Fig. 3.1(b), is usually carried out by **forward substitution**. Forward substitution determines the values first of x_1, then of x_2, and finally of x_3. For the linear system in Fig. 3.1(b), observe that the first equation determines the value of x_1. Given the value of x_1, the second equation then determines the value of x_2. And finally, given the values of both x_1 and x_2, the third equation determines the value of x_3.

Example 3.1.3. In the case in Fig. 3.1(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 $x_3 = \frac{-8-(-1)x_1-4x_2}{(-5)} = \frac{-8-(-1)\cdot 1-4\cdot 2}{(-5)} = 3$.

There are two "popular" implementations of forward substitution. To motivate these implementations, consider the following lower triangular linear system.

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

Row-Oriented	*Column-Oriented*
for $i = 1$ to 3	for $j = 1$ to 3
for $j = 1$ to $i - 1$	$x_j := b_j/a_{j,j}$
$b_i := b_i - a_{i,j}x_j$	for $i = j + 1$ to 3
next j	$b_i := b_i - a_{i,j}x_j$
$x_i := b_i/a_{i,i}$	next i
next i	next j

Figure 3.4: Pseudocode *Row- and Column-Oriented Forward Substitution,* $n = 3$.

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

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

Observe that the right hand sides can be divided naturally into rows and columns. Row—oriented forward substitution evaluates the right hand side one row at a time, while column–oriented forward substitution evaluates the right hand side one column at a time. Pseudocodes implementing row- and column–oriented forward substitution are presented in Fig. 3.4. Note that:

- The "equations" in the algorithms are not intended to make mathematical sense. The symbol ":=" means assign the value of the right hand side to the variable on the left hand side.
- The "for" loops step in ones. So, "for $i = 1$ to 3 means execute the loop for each value $i = 1$, $i = 2$ and $i = 3$, in turn. (Later, in Figures 3.6 and 3.9 we use "downto" when we want a loop to count backwards.) When a loop counting forward has the form "for $j = 1$ to $i - 1$" and for a given value of i we have $i - 1 < 1$ then the loop is considered to be *empty* and does not execute for any value of j. A similar convention applies for empty loops counting backwards.
- The algorithm destroys the original entries of b. If these entries are needed for later calculations, then they must be saved elsewhere. In some implementations, the entries of x are written over the corresponding entries of b.

The corresponding algorithm for a general lower triangular system of order n is given in Figure 3.5

Problem 3.1.8. *Why is a diagonal linear system also a lower triangular linear system?*

Problem 3.1.9. *Consider a general lower triangular linear system of equations of order n. What are the conditions on the entries of its coefficient matrix so that its solution can be determined by each of the forward substitution pseudocodes?*

Problem 3.1.10. *Illustrate the operation of column–oriented forward substitution when used to solve the lower triangular linear system in Fig. 3.1(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 3.1.11. *Use the row–oriented version of forward substitution to solve the following linear system of order 4.*

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

Problem 3.1.12. *Repeat the previous problem but this time using the column–oriented version of forward substitution.*

Problem 3.1.13. *Modify the row– and the column–oriented pseudocodes in Fig. 3.5 for forward substitution so that the solution x is written over the right hand side b.*

Row-Oriented	*Column-Oriented*
for $i = 1$ to n	for $j = 1$ to n
for $j = 1$ to $i - 1$	$x_j := b_j / a_{j,j}$
$b_i := b_i - a_{i,j}x_j$	for $i = j + 1$ to n
next j	$b_i := b_i - a_{i,j}x_j$
$x_i := b_i / a_{i,i}$	next i
next i	next j

Figure 3.5: Pseudocode *Row- and Column-Oriented Forward Substitution, general n.*

Problem 3.1.14. *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: Draw a picture of A and place on the entry* $a_{i,j}$ *the symbol* $\otimes$ *every time that entry is involved in a multiply, the symbol* $\ominus$ *every time that entry is involved in a subtract, and the symbol* $\oslash$ *every time that entry is involved in a divide. How many symbols are attached to each entry of A? Hint: A more general approach observes that there are* $(i-1)$ *multiplies and subtracts in the inner loop and that this loop is executed for each value of* $i = 1, 2, \cdots, n$*. So, the total cost is*

$$0 + 1 + \cdots + (n-1) = \sum_{i=1}^{n}(i-1) = \sum_{i=0}^{n-1} i = \sum_{i=1}^{n-1} i$$

multiplies and subtracts.

Problem 3.1.15. *Repeat the previous problem but this time for the column–oriented version of forward substitution. Hint: The costs should be identical.*

Problem 3.1.16. *In the olden days (25 years ago) a* flop *referred to the operations that the computer performed when evaluating the "linked triad" assignment statement "b(i) := b(i) - a(i,j)*b(j)"; these operations include the obvious multiply and subtract operations as well as several array index computations and data transfer statements. This is an important quantity because, for many numerical linear algebra algorithms, the total number of flops dominate the amount of work performed by the algorithm. Show that this is the case for row–oriented forward substitution when* $n = 100$*, i.e., show that the number of flops is far larger than the number of divides. Remark: Today, a flop refers to a floating–point operation, generally either a multiply or an add. So, today's computers have a flop rate that is twice the flop rate they would have had in the olden days!*

Problem 3.1.17. *Develop pseudocodes, analogous to those in Fig. 3.4, for row–oriented and column–oriented methods of solving the following linear system of order* 3

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

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 of order n, like that in Fig. 3.1(c), is usually carried out by **backward substitution**. Backward substitution determines the value of x_3, then x_2, and finally x_1. For the linear system in Fig. 3.1(c), observe that the third equation determines the value of x_3. Given the value of x_3, the second equation then determines the value of x_2. And finally, given the values of x_2 and x_3, the first equation determines the value of x_1.

Problem 3.1.18. *In the case in Fig. 3.1(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

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

Row-Oriented	*Column-Oriented*
for $i = 3$ downto 1 do	for $j = 3$ downto 1 do
for $j = i + 1$ to 3	$x_j := b_j / a_{j,j}$
$b_i := b_i - a_{i,j} x_j$	for $i = 1$ to $j - 1$
next j	$b_i := b_i - a_{i,j} x_j$
$x_i := b_i / a_{i,i}$	next i
next i	next j

Figure 3.6: Pseudocode *Row- and Column-Oriented Backward Substitution*

Observe that the right hand sides of these equations can again be divided naturally into rows and columns. Row–oriented backward substitution evaluates the right hand side one row at a time, while column–oriented backward substitution evaluates the right hand side one column at a time. Pseudocodes implementing row– and column–oriented backward substitution for systems of size $n = 3$ are presented in Fig. 3.6.

Problem 3.1.19. *Why is a diagonal linear system also an upper triangular linear system?*

Problem 3.1.20. *Modify the row– and column–oriented pseudocodes in Fig. 3.6 for backward substitution so they apply to upper triangular linear systems of order n for any n.*

Problem 3.1.21. *Consider a general upper triangular linear system of order n. What are the conditions on the entries of the coefficient matrix so that its solution can be determined by each of the backward substitution pseudocodes?*

Problem 3.1.22. *Illustrate the operation of column–oriented backward substitution when used to solve the upper triangular linear system in Fig. 3.1(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 3.1.23. *Use the row–oriented version of backward substitution to solve the following linear system of order 4.*

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

Problem 3.1.24. *Repeat the previous problem but this time using the column–oriented version of backward substitution.*

Problem 3.1.25. *Modify the row– and column–oriented pseudocodes for backward substitution so that the solution x is written over b.*

Problem 3.1.26. *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. Hint: Draw a picture of A and place on the entry $a_{i,j}$ the symbol $\otimes$ every time that entry is involved in a multiply, the symbol $\ominus$ every time that entry is involved in a subtract, and the symbol $\oslash$ every time that entry is involved in a divide. How many symbols are attached to each entry of A? Hint: A more general approach observes that there are $(n - i)$ multiplies and subtracts in the inner loop and that this loop is executed for each value of $i = n, (n - 1), \cdots, 1$. So, the total cost is*

$$0 + 1 + \cdots + (n-1) = \sum_{i=1}^{n} (i-1) = \sum_{i=0}^{n-1} i = \sum_{i=1}^{n-1} i$$

multiplies and subtracts.

Problem 3.1.27. *Repeat the previous problem but this time using the column–oriented version of backward substitution.*

Problem 3.1.28. *Develop pseudocodes, analogous to those in Fig. 3.6, for row– and column–oriented methods of solving the following linear system of order* 3.

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

The Determinant of a Triangular Matrix

The determinant of a triangular matrix, be it diagonal, lower triangular, or upper triangular, **is the product of its diagonal entries**. Therefore, a triangular matrix is nonsingular if and only if each of its diagonal entries is nonzero.

This result has immediate application to solving a linear system $Ax = b$ of order n whose coefficient matrix A is triangular. Specifically, such a linear system has one and only one solution when each of the diagonal entries of A is nonzero. In this case, forward or backward substitution, as appropriate, can be used to determine its solution.

Remarks on 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 computer's 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; that is, numbers in the same column of the matrix are stored in consecutive memory locations, so column–oriented algorithms generally run faster.

We remark that 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.

C stores matrices in row-major order; that is, 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.

3.2 Gaussian Elimination

3.2.1 Outline

Gauss described a process, called Gaussian elimination (GE) in his honor, in which two elementary operations are used systematically to transform any given linear system into one that is easy to solve.

The two elementary operations used by GE are

(a) **exchange two equations**

(b) **subtract a multiple of one equation from any other equation**.[1]

[1]The algorithm described here is formally referred to as Gaussian elimination with partial pivoting by rows for size

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 we need in any order we choose and the resulting system of linear equations has the same solution set as the original system. Of course, in floating–point arithmetic each operation of type (b) introduces error due to roundoff so we do need to pay some attention to the effects of these operations.

For a linear system of order n, GE uses n stages to transform the linear system into upper triangular form. The goal of stage k is to eliminate variable x_k from all but the first k equations. To achieve this goal, each stage uses the same 3 steps. We illustrate these steps on the linear systems of order 3 presented in Examples A and B in Figs. 3.7 and 3.8, respectively.

Stage 1

The goal of stage 1 is to eliminate x_1 from all but the first equation.

The first step, called 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 with largest magnitude, 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 has no pivot and we terminate. In Example A, the pivot is already in the first equation so no exchange is necessary. In Example B, the pivot occurs in the third equation so equations 1 and 3 are exchanged.

The second step, called the **elimination step**, eliminates variable x_1 from all but the first equation. For Example A, this involves subtracting $m_{2,1} = 2/4$, times equation 1 from equation 2, and subtracting $m_{3,1} = 1/4$ times equation 1 from equation 3. For Example B, this involves subtracting $m_{2,1} = -1/2$ times equation 1 from equation 2. Each of the numbers $m_{i,1}$ is a **multiplier**; the first subscript on $m_{i,1}$ tells you from which equation the multiple of the first equation is subtracted. The multiplier $m_{i,1}$ 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).

The third step, the **removal step**, removes the first equation and makes it the 1^{st} equation of the final upper triangular linear system.

Stage 2

Stage 2 continues with the smaller linear system produced at the end of stage 1. Note that this smaller linear system involves one fewer unknown (the elimination step in the first stage eliminated variable x_1) and one fewer equation (the removal step in the first stage removed the first equation) than does the original linear system. The goal of stage 2 is to eliminate the variable x_2 from all but the first equation of this smaller linear system.

As stated before, each stage uses the same 3 steps. For stage 2, the exchange step exchanges equations so that among the coefficients multiplying x_2 in all of the remaining equations, the coefficient in the first equation has largest magnitude. If this coefficient with largest magnitude is nonzero, then it is called the pivot for stage 2, otherwise stage 2 has no pivot and we terminate. The elimination step eliminates the variable x_2 from all but the first equation. For each of Example A and B, we compute a single multiplier $m_{3,2}$ which is the coefficient of x_2 in equation 2 divided by the coefficient of x_2 in equation 1 (that is, the pivot). (Note that the numbering of the multiplier $m_{3,2}$ corresponds to the equation and variable numbering in the original linear system.) Then we subtract $m_{3,2}$ times the first equation off the second equation to eliminate x_2 from the second equation. The removal step then removes the first equation and uses it to form the 2^{nd} equation of the final upper triangular linear system.

Stage 3

Stage 3 continues with the smaller linear system produced by the last step of stage 2. Its goal is to eliminate the variable x_3 from all but the first equation. However, in this, the last stage of GE, the linear system consists of just one equation in one unknown, so the exchange and elimination steps do not change anything. The only useful work performed by this stage consists of (1) identifying

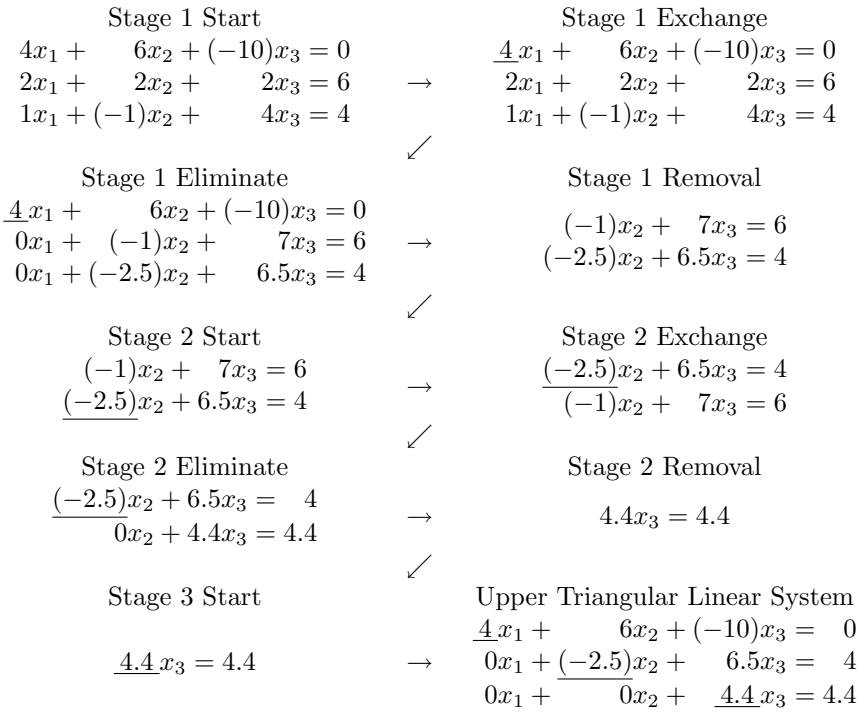

Figure 3.7: Gaussian elimination, example A.

the pivot, if one exists (if it does not we terminate), and (2) removing this equation and making it the last equation of the final upper triangular linear system.

Solving the Upper Triangular Linear System

GE is now finished. When the entries on the diagonal of the final upper triangular linear system are nonzero, as in Figs. 3.7 and 3.8, the linear system is nonsingular and its solution may be determined by backward substitution. (Recommendation: If you're determining the solution by hand, then check by substituting that your computed solution satisfies all the equations of the original linear system.)

The diagonal entries of the upper triangular linear system produced by GE 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 GE. So, *the upper triangular linear system produced by GE is nonsingular if and only if each stage of GE has a pivot*.

To summarize:

- GE 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 GE starts 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 GE ends with a linear system that involves $n-k$ equations in the $n-k$ unknowns $x_{k+1}, x_{k+2}, \cdots, x_n$, having removed its first equation and added it to the final upper triangular linear system.
- If GE finds a non–zero pivot in 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.

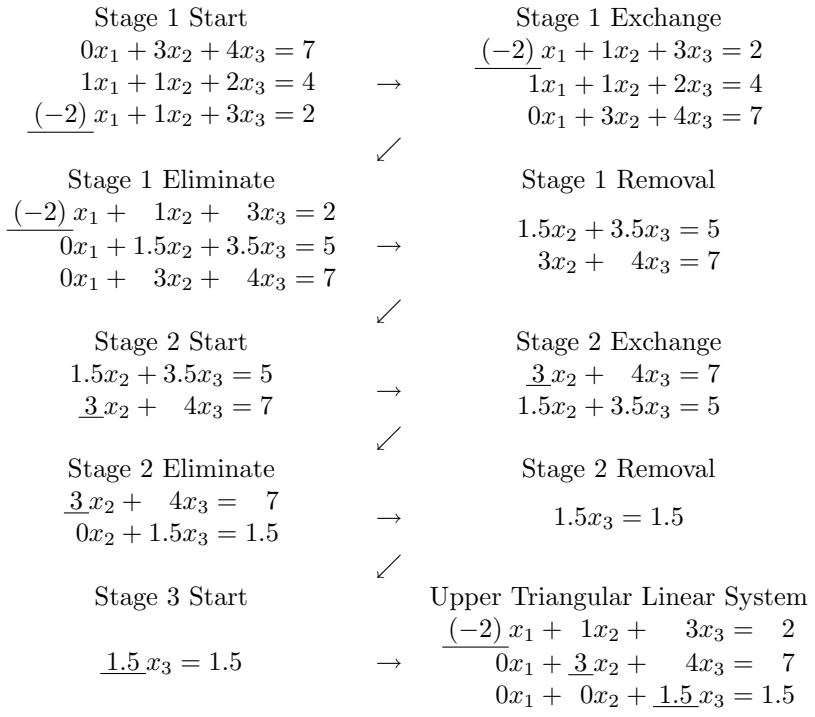

Figure 3.8: Gaussian elimination, example B.

- If GE fails to find a 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 GE does not find a pivot during some stage, then we can conclude immediately that the original linear system is singular**. Consequently, as in our algorithm, many GE codes simply terminate elimination and return a message that indicates the original linear system is singular.

Problem 3.2.1. *Use GE followed by backward substitution to solve the linear system of Example 3.1.1. Explicitly display the value of each pivot and multiplier.*

Problem 3.2.2. *Use GE followed by backward substitution to solve the following linear system of order* 4*. Explicitly display the value of each pivot and multiplier.*

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

Problem 3.2.3. *Use GE and backward substitution to solve the following linear system of order* 3*.*

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

Computing the Determinant

Recall that the two operations used by GE 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.

We therefore draw the following conclusion. Suppose GE 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

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

This relationship provides a simple and efficient way to compute the determinant of any matrix.

Example 3.2.1. Consider the initial coefficient matrix displayed in Fig. 3.7. Of the exchange steps GE uses to transform it to upper triangular form, only one performs an actual exchange. So, in the above formula we conclude that $m = 1$ and therefore

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

Problem 3.2.4. *Use GE to compute the determinant of the initial coefficient matrix displayed in Fig. 3.8.*

Problem 3.2.5. *Use GE to calculate the determinants of each of the matrices:* $\begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 2 \\ 4 & -3 & 0 \end{bmatrix}$ *and*

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

3.2.2 Implementing Gaussian Elimination

A pseudocode for GE is presented in Fig. 3.9. This version of GE 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 to partial pivoting 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, GE 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 most cases, partial pivoting improves the quality of the solution computed when finite precision, rather than exact, arithmetic is used in the elimination step. This fact will be illustrated later.

During stage k, this pseudocode only modifies equations k through n and the coefficients multiplying $x_k, x_{k+1}, \cdots, x_n$. So, the task of "removing equations" is carried out implicitly. Also, note that

- The instruction $a_{i,k} := 0$ that sets $a_{i,k}$ to zero is unnecessary in most implementations. Usually, it is implicitly assumed that the user is aware that, after the elimination of the k^{th} variable x_k, the coefficients $a_{i,k} = 0$ for $i > k$. These elements are not involved in determining the solution via backward substitution.
- Of the three fundamental operations: add, multiply, and divide, generally add 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 1, this would replace $n-1$ divisions with 1 division followed by $n-1$ multiplications. Generally, this is faster from a computer arithmetic perspective. Keeping the reciprocal in $a_{k,k}$ also helps in backward substitution where each divide is replaced by a multiply.

```
! Gaussian elimination – row oriented
for k = 1 to n − 1
    !... Exchange step
    !... Find p such that |a_{p,k}| is the first of the
    !... largest values among {|a_{q,k}|}_{q=k}^{n}
    p := k
    for i = k + 1 to n
        if |a_{i,k}| > |a_{p,k}| then p := i
    next i
    ! exchange equations p and k if necessary
    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)
    !... elimination step - row oriented
    for i = k + 1 to n
        m_{i,k} := a_{i,k}/a_{k,k}   !... compute multiplier
        a_{i,k} := 0   !... start of elimination
        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)

! Backward substitution – row oriented
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 3.9: Pseudocode *row oriented GE and Backward Substitution*

Example 3.2.2. Using exact arithmetic, 2 stages of GE transforms the following singular linear system of order 3:

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

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. When floating–point arithmetic is used, the zeros in the last equation will be very small numbers due to rounding, and the resulting computed upper triangular matrix will usually be nonsingular. Particularly, the pivot value $-4/3$ cannot be represented exactly in IEEE standard arithmetic, so operations involving it, like the computation of the final version of the third equation, are subject to rounding errors. Still, backward substitution would give a good approximation to *one* (essentially arbitrary choice of) solution of the original linear system.

If the original first equation $x_1 + x_2 + x_3 = 1$ was replaced by $x_1 + x_2 + x_3 = 2$, then with exact arithmetic 2 stages of GE would yield $0x_3 = 1$ as the last equation and there would be no solution. In floating–point arithmetic, a suspiciously large solution may be computed because in this resulting last equation the coefficient 0 multiplying x_3 would usually be computed as a non-zero, but very small, pivot and the right hand side 1 would be changed only slightly.

Problem 3.2.6. *In Fig. 3.9, in the 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? Hint: Think of the effects of floating–point arithmetic.*

Problem 3.2.7. *In Fig. 3.9, the elimination step is row–oriented. Change the elimination step so it is column–oriented. Hint: In effect, you must exchange the order of the for-i and for-j loops.*

Problem 3.2.8. *In Fig. 3.9, show that the elimination step of the k^{th} stage of GE requires $n - k$ divisions to compute the multipliers, and $(n-k)^2 + n - k$ multiplications and $(n-k)^2 + n - k$ subtractions to perform the eliminations. Conclude that the elimination steps of GE requires about*

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

We say $f(n) \sim g(n)$ when the ratio $\dfrac{f(n)}{g(n)}$ approaches 1 as n becomes large.

Problem 3.2.9. *How many comparisons does GE use to find pivot elements?*

Problem 3.2.10. *How many assignments would you use to accomplish all the interchanges in the GE algorithm? Hint: Assume that at each stage an interchange is required.*

3.2.3 The Role of Interchanges

When floating–point, rather than exact, arithmetic is used in Gaussian elimination, it is important to carefully choose each pivot. To illustrate this fact, apply Gaussian elimination to the linear system

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

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 if Gaussian elimination is applied to this linear system using floating–point arithmetic? To make pen-and-pencil computation easy, let us use round-to-nearest 4 significant digit decimal arithmetic. This means 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 the exact value of the multiplier $\frac{1}{0.000025} = 40000$ rounds to itself; that is, the portion rounded-off is 0. So, the elimination step of GE subtracts 40000 times equation 1 from equation 2. Now 40000 times equation 1 is computed exactly simply because multiplying by 1 is exact. So, the only rounding error in the elimination step is that produced when its subtraction is performed. In this subtraction 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 of the elimination step is

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

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

$$\begin{aligned} x_2 &= \frac{40000}{40000} = 1 \\ x_1 &= \frac{1 - x_2}{0.000025} = 0 \end{aligned}$$

Recall, the exact solution of the linear system has $x_1 \approx 1$, so this computed solution differs significantly from the exact solution.

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

Now consider what happens if we include the exchange step. The result of the exchange step is the linear system

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

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 of the elimination step, $1 - 0.000025 = 0.999975$ rounds to 1 and $1 - 2 \times 0.000025 = 0.99995$ rounds to 1. Even with this approximate arithmetic, backwards substitution commits no further rounding errors and produces an accurate, but not exact, approximate solution

$$\begin{aligned} x_2 &= 1 \\ x_1 &= \frac{2 - x_2}{1} = 1 \end{aligned}$$

Partial pivoting by rows for size is a heuristic that serves only as a guide to choosing the pivot. 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 will therefore be 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 GE will produce an accurate answer, it is neither perfect nor is it always better than any other (or even no) interchange strategy.

Problem 3.2.11. *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

$$\begin{aligned} x_1 &= \frac{40000}{39999} \approx 1 \\ x_2 &= \frac{39998}{39999} \approx 1 \end{aligned}$$

Problem 3.2.12. *(Watkins) Carry out GE without exchange steps and with row–oriented backwards substitution on the following linear system of order* 3*.*

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

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 3.2.13. *Carry out GE with exchange steps and row–oriented backwards substitution on the linear system of order* 3 *in Problem 3.2.12. 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?*

3.3 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*, for 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

$$x^3 - 6x^2 - 7x - 6 = 0 \quad \text{and} \quad (x-1)(x-2)(x-3) = 0$$

are equivalent, but the factored form of the equation is clearly much easier to solve. In general, we will not be able to solve linear systems so easily (i.e., by inspection), but decomposing A will make solving the linear system $Ax = b$ computationally less difficult.

We remark that some knowledge of matrix algebra, especially matrix multiplication, is needed to understand the concepts introduced in this section. A review of basic matrix algebra can be found in Section 1.2.

3.3.1 LU Factorization

Suppose we apply Gaussian elimination to an $n \times n$ matrix A without interchanging any rows. For example,

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \longrightarrow \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} \\ 0 & a_{32}^{(1)} & a_{33}^{(1)} \end{bmatrix} \longrightarrow \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} \\ 0 & 0 & a_{33}^{(2)} \end{bmatrix}$$

$$m_{21} = \tfrac{a_{21}}{a_{11}}, \;\; m_{31} = \tfrac{a_{31}}{a_{11}} \qquad\qquad m_{32} = \tfrac{a_{32}^{(1)}}{a_{22}^{(1)}}$$

Here the superscripts on the entries $a_{ij}^{(k)}$ are used to indicate entries of the matrix that are modified during the kth elimination step. Recall that, in general, the multipliers are computed as

$$m_{ij} = \frac{\text{element to be eliminated}}{\text{current pivot element}}.$$

If the process does not break down (that is, all pivot elements, $a_{11}, a_{22}^{(1)}, a_{33}^{(2)}, \ldots$, are nonzero) then the matrix A can be factored as $A = LU$ where

$$L = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ m_{21} & 1 & 0 & \cdots & 0 \\ m_{31} & m_{32} & 1 & & 0 \\ \vdots & \vdots & & \ddots & \vdots \\ m_{n1} & m_{n2} & m_{n3} & \cdots & 1 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} \text{upper triangular matrix} \\ \text{after elimination is} \\ \text{completed} \end{bmatrix}.$$

The following example illustrates the process.

Example 3.3.1. Consider the matrix

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

Using Gaussian elimination without row interchanges, we obtain

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & -3 & 2 \\ 3 & 1 & -1 \end{bmatrix} \xrightarrow[m_{21}=2,\ m_{31}=3]{} \begin{bmatrix} 1 & 2 & 3 \\ 0 & -7 & -4 \\ 0 & -5 & -10 \end{bmatrix} \xrightarrow[m_{32}=\frac{5}{7}]{} \begin{bmatrix} 1 & 2 & 3 \\ 0 & -7 & -4 \\ 0 & 0 & -\frac{50}{7} \end{bmatrix}$$

and thus

$$L = \begin{bmatrix} 1 & 0 & 0 \\ m_{21} & 1 & 0 \\ m_{31} & m_{32} & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & \frac{5}{7} & 1 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} 1 & 2 & 3 \\ 0 & -7 & -4 \\ 0 & 0 & -\frac{50}{7} \end{bmatrix}.$$

It is straightforward to verify[2] that $A = LU$; that is,

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & -3 & 2 \\ 3 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & \frac{5}{7} & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & -7 & -4 \\ 0 & 0 & -\frac{50}{7} \end{bmatrix}.$$

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

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

Therefore, the following steps can be used to solve $Ax = b$:

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

[2]Recall from Section 1.2 that if $A = LU$, then the i, j entry of A can be obtained by multiplying the ith row of L times the jth column of U.

Example 3.3.2. 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 backard 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 3.3.1. *Find the LU factorization of the following matrices:*

(a) $A = \begin{bmatrix} 4 & 2 & 1 & 0 \\ -4 & -6 & 1 & 3 \\ 8 & 16 & -3 & -4 \\ 20 & 10 & 4 & -3 \end{bmatrix}$

(b) $A = \begin{bmatrix} 3 & 6 & 1 & 2 \\ -6 & -13 & 0 & 1 \\ 1 & 2 & 1 & 1 \\ -3 & -8 & 1 & 12 \end{bmatrix}$

Problem 3.3.2. *Suppose the LU factorization of a matrix A is given by:*

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

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

Problem 3.3.3. *Suppose*

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

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

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

Problem 3.3.4. *Suppose*

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

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

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

3.3.2 $PA = LU$ Factorization

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

Example 3.3.3. Consider the matrix P obtained by switching the first and third rows of a 4×4 identity matrix:

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

Then notice that multiplying any 4×4 matrix on the left by P has the effect of switching its first and third rows. For example,

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

Suppose we apply Gaussian elimination with partial pivoting by rows to reduce a matrix A to upper triangular form. Then the matrix factorization that is computed is not an LU decomposition of A, but rather an LU decomposition of a *permuted* version of A. That is,

$$PA = LU$$

where P is a permutation matrix representing *all* row interchanges. It is nontrivial to derive this factorization, which is typically done in more advanced treatments of numerical linear algebra. However, finding P, L and U is not difficult. In general, we proceed as in the previous subsection for the LU factorization, but we keep track of the row swaps as follows:

- Each time we switch rows of A, we must switch corresponding multipliers. For example, if we switch rows 3 and 5, then we must also switch m_{3k} and m_{5k}, $k = 1, 2, 3$.
- To find P, begin with $P = I$, and each time we switch rows of A, switch corresponding rows of P.

We remark that it is possible to implement the computation of the factorization without explicitly constructing the matrix P. For example, an index of "pointers" to rows could be used. However, the above outline can be used to compute the $PA = LU$ factorization by hand, as is illustrated in the following example.

Example 3.3.4. Consider the matrix

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

To find the $PA = LU$ factorization, we proceed as follows:

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

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

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

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

Therefore, the following steps can be used 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.

Example 3.3.5. Use the $PA = LU$ factorization of the previous example to solve $Ax = b$, where

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

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 backard 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 3.3.5. *Use Gaussian elimination with partial pivoting to find the $PA = LU$ factorization of the following matrices:*

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

(b) $A = \begin{bmatrix} 3 & 6 & 9 \\ 2 & 5 & 2 \\ -3 & -4 & -11 \end{bmatrix}$

(b) $A = \begin{bmatrix} 2 & -1 & 0 & 3 \\ 0 & -\frac{3}{4} & \frac{1}{2} & 4 \\ 1 & 1 & 1 & -\frac{1}{2} \\ 2 & -\frac{5}{2} & 1 & 13 \end{bmatrix}$

Problem 3.3.6. *Suppose* $A = \begin{bmatrix} 1 & 2 & -3 & 4 \\ 4 & 8 & 12 & -8 \\ 2 & 3 & 2 & 1 \\ -3 & -1 & 1 & -4 \end{bmatrix}$ *and* $b = \begin{bmatrix} 3 \\ 60 \\ 1 \\ 5 \end{bmatrix}$.
Suppose Gaussian elimination with partial pivoting was used to obtain the $PA = LU$ *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 (not the matrix A) to solve $Ax = b$.

3.4 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, the following material qualitatively describes some of the factors affecting the accuracy.

Consider a nonsingular linear system $Ax = b$ of order n. The solution process involves two steps: GE is used to transform $Ax = b$ into an upper triangular linear system $Ux = c$, and $Ux = c$ is solved by backward substitution (say, row–oriented). We use the phrase "the computed solution" to refer to the solution obtained when these two steps are performed using floating–point arithmetic. Generally, the solution of the upper triangular linear system $Ux = c$ is determined accurately, so we focus on comparing the solutions of $Ax = b$ and $Ux = c$. Remember, the solutions of $Ax = b$ and $Ux = c$ are different because GE was performed using floating–point arithmetic.

The key to comparing the solutions of $Ax = b$ and $Ux = c$ is to reverse the steps used in GE. Specifically, consider the sequence of elementary operations used to transform $Ax = b$ into $Ux = c$. Using exact arithmetic, apply the inverse of each of these elementary operations, in the reverse order, to $Ux = c$. The result is a linear system $A'x = b'$ and, because exact arithmetic was used, it has the same solution as $Ux = c$. One of the nice properties of GE is that, generally, A' is "close to" A and b' is "close to" b. This observation is described by saying that GE is generally **stable**.

To summarize: GE is generally stable. When GE is stable, the computed solution of $Ax = b$ is the exact solution of $A'x = b'$ where A' is close to A and b' is close to b.

The next step is to consider what happens to the solution of $Ax = b$ when A and b are changed a little. Linear systems can be placed into two categories. One category, consisting of so-called **well–conditioned** linear systems, 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 x of the linear system $Ax = b$. If a linear system is not well-conditioned then it is said to be **ill–conditioned**. So, an **ill–conditioned** linear system has the property that *some small changes* in the matrix A and/or the right hand side b lead

to a large change in the solution x of the linear system $Ax = b$. An example of an ill–conditioned linear system is presented at the end of this section.

What factors determine whether a linear system is well–conditioned or ill–conditioned? Theory tells us that an important factor is the distance from the matrix A to the "closest" singular matrix. In particular, *the linear system $Ax = b$ is ill–conditioned when the matrix A is close to a singular matrix.*

How do the concepts of well–conditioned and ill–conditioned linear systems apply to the computed solution of $Ax = b$? *Backward error analysis* shows that the approximate solution computed by GE of the linear system $Ax = b$ is the exact solution of a related linear system $A'x = b'$. Usually, A' is close to A and b' is close to b. In this case, if $Ax = b$ is well–conditioned, then it follows that the computed solution is accurate. On the other hand, if $Ax = b$ is ill–conditioned, then even if A' is close to A and b' is close to b, these small differences *may* lead to a large difference between the solution of $Ax = b$ and the solution of $A'x = b'$, and the computed solution *may not* be accurate.

In summary, we draw the following conclusions about the computed solution of the linear system $Ax = b$. Suppose GE, when applied to this linear system, is stable. If the linear system is well–conditioned, then the computed solution is accurate. On the other hand, if this linear system ill–conditioned, then the computed solution *may not* be accurate.

A measure of the conditioning of a linear system is the *condition number* of the matrix A of the system. This measure, which is defined more precisely in advanced numerical analysis textbooks, is the product of the "size" of the matrix A and the "size" of its inverse A^{-1}. This quantity is always greater than one; it is one for the identity matrix (that is, a diagonal matrix with ones on the main diagonal). In general the larger the condition number, the more ill-conditioned the matrix and the more likely that the solution of the linear system will be inaccurate. To be more precise, 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 $Ax = b$ is unlikely to be more than about $s - p$ decimal digits accurate. Recall that in SP arithmetic, $s \approx 7$, and in DP arithmetic, $s \approx 15$. Functions for estimating the condition number of any matrix A fairly inexpensively are available in most high quality numerical software libraries; some remarks on this are given in Section 3.5.

We conclude this section by describing two ways of *how not* to attempt to estimate the accuracy of a computed solution. 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 the case.

Consider, for example, the two linear systems:

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

where the second is obtained from the first by multiplying each of its equations by 0.5. The determinant of the coefficient matrix of the first linear system is 1 and for the second linear system it is 0.125. So, by comparing the magnitude of these determinants, we might believe that GE would produce a less accurate solution of the second linear system. However, GE produces identical solutions for both linear systems because each linear system is diagonal, so no exchanges or eliminations need be performed. We conclude, that *the magnitude of the determinant is not necessarily a good indicator of how close a coefficient matrix is to the nearest singular matrix.*

Another test you might consider is to check how well the approximate solution x determined by GE satisfies the original linear system $Ax = b$. The *residuals* $r_1, r_2, \cdots, r_n$ associated with the approximate solution x are defined as

$$\begin{aligned} r_1 &= b_1 - a_{1,1}x_1 - a_{1,2}x_2 - \cdots - a_{1,n}x_n \\ r_2 &= b_2 - a_{2,1}x_1 - a_{2,2}x_2 - \cdots - a_{2,n}x_n \\ &\vdots \\ r_n &= b_n - a_{n,1}x_1 - a_{n,2}x_2 - \cdots - a_{n,n}x_n \end{aligned}$$

Observe that $r_k = 0$ when the approximate solution x satisfies the k^{th} equation, so the magnitudes

of the residuals indicate by how much the approximate solution x fails to satisfy the linear system $Ax = b$.

Example 3.4.1. (Watkins) Consider the linear system of equations of order 2:

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

with exact solution $x_1 = 1$ and $x_2 = 1$ for which the residuals are $r_1 = 0.0$ and $r_2 = 0.0$. For the close by approximate solution $x_1 = 1.01$ and $x_2 = 0.99$ the residuals are $r_1 = 0.01$ and $r_2 = 0.01$, a predictable size. Clearly, the choice $x_1 = 20.97$ and $x_2 = -18.99$ is very inaccurate when considered as an approximate solution of this linear system. Yet, the residuals, $r_1 = 0.01$ and $r_2 = -0.01$, associated with this approximate solution are of very small magnitude when compared to the entries of the approximate solution, to the coefficient matrix or to the right hand side. Indeed they are of the same size as for the approximate solution $x_1 = 1.01$ and $x_2 = 0.99$.

As this example shows, even very inaccurate solutions can give residuals of small magnitude. Indeed, GE generally returns an approximate solution x whose associated residuals are of small magnitude, even when the solution is inaccurate! We conclude that *the magnitudes of the residuals associated with an approximate solution are not a good indicator of the accuracy of that solution.*

By rearranging the equations for the residuals, they may be viewed as perturbations of the right hand side. If the residuals have small magnitude then they are small perturbations of the right hand side. In this case, if the solution corresponding to these residuals is very different from the exact solution (with zero residuals) then the linear system is ill–conditioned, because small changes in the right hand side have led to large changes in the solution.

Problem 3.4.1. *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, does it follow that this solution is accurate? Answer the same question but assuming that the linear system $Ax = b$ is ill–conditioned. Justify your answers.*

Problem 3.4.2. *Consider the linear system of equation of order 2:*

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

whose exact solution is $x_1 = 1$ and $x_2 = -1$. Consider two approximate solutions: first $x_1 = 0.999$ and $x_2 = -1.001$, and second $x_1 = 0.341$ and $x_2 = -0.087$. Given your knowledge of the exact solution, you might agree that among these two approximate solutions, the first seems more accurate than the second. Now compute the residuals associated with each of these approximate solutions. Discuss whether the accuracy of the approximate solutions is reflected in the size of the residuals.

3.5 Matlab Notes

A large amount of software is available for solving linear systems where the coefficient matrices may have a variety of structures and properties. We restrict our discussion to solving "dense" systems of linear equations. (A "dense" system is one where all the coefficients are considered to be non-zero. So, the matrix is considered to have no special structure.) Most of today's best software for solving "dense" systems of linear equations, including those found in MATLAB, was developed in the *LAPACK* project.

In this section we describe the main tools (i.e., the backslash operator and the `linsolve` function) provided by MATLAB for solving dense linear systems of equations. Before doing this, though, we first develop MATLAB implementations of some of the algorithms discussed in this chapter. These examples build on the brief introduction given in Chapter 1, and are designed to introduce some useful MATLAB commands as well as to teach proper MATLAB programming techniques.

3.5.1 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:

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

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

Notice that we make use of the MATLAB function `length` to determine the dimension of the linear system, assumed to be 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. Otherwise the implementation is a direct translation of the algorithm.

We can shorten this code, and make it more efficient by using array operations, as illustrated in the following code:

```
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 the function `DiagSolve2`, we use MATLAB's `diag` function to extract the diagonal entries of `A`, and store them in a column vector `d`. The purpose of the MATLAB function `any` is clear; 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, `./`

We remark that 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. In this case, our function to solve a diagonal linear system can have the form:

```
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;
```

Notice that `DiagSolve3` contains some additional comments explaining the input parameters, since they may be less obvious than was the case for `DiagSolve1` and `DiagSolve2`.

Problem 3.5.1. *Implement the functions* `DiagSolve1`*,* `DiagSolve2`*, and* `DiagSolve3`*, and use them to solve the linear system in Fig. 3.1(a).*

Problem 3.5.2. *Consider the following set of* 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
```

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

3.5.2 Triangular Linear Systems

In this subsection we describe MATLAB implementations of the forward substitution algorithms for lower triangular linear systems. Implementations of backward substitution for upper triangular linear systems 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. 3.4. A direct translation of the pseudocode into MATLAB code results in the following implementation:

```
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. First, as with the diagonal solve functions, we should include a statement that checks to see if `A(i,i)` is zero. In addition, the inner most loop can be replaced with a single MATLAB array operation. To see how this can be done, first observe that we can write the algorithm as:

$$\begin{array}{l}\text{for } i = 1 : n \\ \quad x_i = \left(b_i - \left(a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{i,i-1}x_{i-1}\right)\right)/a_{ii} \\ \text{end}\end{array}$$

Using linear algebra notation, we could write this as:

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

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

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

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

$$\begin{array}{l}\text{for } i = 1 : n \\ \quad x(i) = \left(b(i) - A(i, 1:i-1) * x(1:i-1)\right) / A(i,i) \\ \text{end}\end{array}$$

When $i = 1$, MATLAB considers $A(i, 1:i-1)$ and $x(1:i-1)$ to be "empty" matrices, and by default the computation of $A(i, 1:i-1) * x(1:i-1)$ is set to 0. Thus, when $i = 1$, the algorithm 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 as:

```
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
```

An 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 we did in the function `LowerSolve1`. Therefore, a MATLAB implementation of column-oriented forward substitution could be written as:

```
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
```

An observant reader may wonder 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 essentially skips over this part of the computation, and the loop is completed, without any errors.

Problem 3.5.3. *Implement the functions* `LowerSolve1` *and* `LowerSolve2`*, and use them to solve the linear system given in Fig. 3.1(b).*

Problem 3.5.4. *Consider the following set of* 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
```

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

Problem 3.5.5. *Write a* MATLAB *function that solves an upper triangular linear system using row-oriented backward substitution. Test your code using the linear system given in Fig. 3.1(c).*

Problem 3.5.6. *Write a* MATLAB *function that solves an upper triangular linear system using column-oriented backward substitution. Test your code using the linear system given in Fig. 3.1(c).*

Problem 3.5.7. *Write a script M-file that compares timings using row-oriented and column-oriented backward substitution to solve an upper triangular linear systems. The script should be similar to that given in Problem 3.5.4.*

3.5.3 Gaussian Elimination

Next we consider a MATLAB implementation of Gaussian elimination, using the pseudocode given in Fig. 3.9. We begin by explaining how to implement each of the various steps during the k^{th} stage of the algorithm.

- First 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 `piv` contains the largest entry (in magnitude) in the vector `A(k:n,k)`, and `i` is its location. It's important to understand that if `i = 1`, then the index `p` in Fig. 3.9 is `p = k`. Similarly, if `i = 2`, then `p = k + 1`, and in general we have

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

- Once the pivot row, `p`, is known, then the p^{th} row of A must be switched with the k^{th} row. Similarly, the corresponding p^{th} entry of b must be switched with the k^{th} entry. This can be done easily using MATLAB's array indexing capabilities:

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

  To explain what happens here, 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 fairly straight forward; 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 computations:

  ```
  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 as:

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

  Note that here the statement

```
for i = n:-1:1
```

  indicates that the loop runs over values $i = n, n-1, \ldots, 1$.

- The next question we must address is how to implement a check for singularity. Due to roundoff errors, it is unlikely that any a_{kk} will be exactly zero, and so the statement

```
if A(k,k) == 0
```

  is unlikely to detect singularity. An alternative approach is to see if a_{kk} is small compared to, say, the largest entry in the matrix A. Such a test could be implemented as follows:

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

  where `tol` is computed in an initialization step, such as:

```
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. See `help max` to learn why we did not simply use `max(abs(A))`.

- Finally, we note that, in addition to `tol`, certain initialization steps are needed. For example, we need to know the dimension, n, and we should allocate space for the solution vector and for the multipliers. We can do these things using the `length`, `zeros`, and `eye` functions:

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

Putting these steps together, we obtain the following function that uses Gaussian elimination with partial pivoting to solve $Ax = b$.

```
function x = gepp(A, b)
%
%        x = gepp(A, b);
%
% Solve Ax=b using Gaussian elimination with partial pivoting
% by rows for size.
%
n = length(b);
m = eye(n);
x = zeros(n,1);
tol = max(A(:))*eps;
%
% Loop for stages k = 1, 2, ..., n-1
%
for k = 1:n-1
  %
  % First search for pivot entry:
  %
  [piv, psub] = max(abs(A(k:n,k)));
  p = psub + k - 1;
  %
  % Next, 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
  %
  % Now 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
%
% Final 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
```

Problem 3.5.8. *Implement the function* `gepp`*, and use it to solve the linear systems given in problems 3.2.2 and 3.2.3.*

Problem 3.5.9. *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 3.5.10. *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 3.5.9. Why are these results different than those computed in problem 3.5.9?*

Problem 3.5.11. *The built-in* MATLAB *function* `hilb` *can be used to construct the so-called 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 of results 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 try to run the script with* $n \geq 14$*. What do you observe?*

Problem 3.5.12. *Rewrite the function* `gepp` *so that it does not use array operations. Compare the efficiency (e.g., using* `tic` *and* `toc`*) of this function with* `gepp` *on matrices of various dimensions.*

3.5.4 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, this operator can be used to solve $Ax = b$ with the single command:

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

When this command is executed, MATLAB first checks to see if the matrix A has a special structure, including diagonal, upper triangular, and lower triangular. If a special structure is recognized (e.g., upper triangular), then a special method (e.g., backward substitution) is used to solve $Ax = b$. If, however, a special structure is not recognized, and A is an $n \times n$ matrix, then Gaussian elimination with partial pivoting is used to solve $Ax = b$. During the process of solving the linear system, MATLAB attempts to estimate the condition number of the matrix, and prints a warning message if A is ill-conditioned.

- The function `linsolve`.
  If we know a-priori that A has a special structure recognized by MATLAB, then we can improve efficiency by avoiding any checks on the matrix, and skipping directly to the special solver routine. This can be especially helpful if, for example, it is known that A is upper triangular. Certain 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 will use it exclusively to solve general $n \times n$ linear systems.

Problem 3.5.13. *Use the backslash operator to solve the linear systems given in problems 3.2.2 and 3.2.3.*

Problem 3.5.14. *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 3.5.15. *Repeat problem 3.5.11 using the backslash operator in addition to* `gepp`.

Problem 3.5.16. *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 the matrix 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?*

It is possible to compute explicitly the $PA = LU$ factorization in MATLAB using the `lu` function. For example, given a matrix A, we can compute this factorization using the basic calling syntax:

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

Problem 3.5.17. *Use the* MATLAB `lu` *function to compute the $PA = LU$ factorization of each of the matrices given in Problem 3.3.5.*

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

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

These statements could be nested into one line of code as:

```
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) );
```

The cost of doing this 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 many different right hand side vectors. It is more expensive to compute the $PA = LU$ factorization than it is to use forward and backward substitution to solve lower and upper triangular systems. Thus if we can compute the factorization only once, and use it for the various different right hand side vectors, a substantial savings can be attained. This is illustrated in the following problem, which involves a relatively small linear system.

Problem 3.5.18. *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
```

Currently the dimension of the problem is set to $n = 50$. *Experiment with other values of* n, *such as* $n = 100, 150, 200$. *Comment on your observations.*