2 Systems of Linear Algebraic Equations

Solve the simultaneous equations $\mathbf{Ax} = \mathbf{b}$.

2.1 Introduction

In this chapter we look at the solution of n linear, algebraic equations in n unknowns. It is by far the longest and arguably the most important topic in the book. There is a good reason for its importance—it is almost impossible to carry out numerical analysis of any sort without encountering simultaneous equations. Moreover, equation sets arising from physical problems are often very large, consuming a lot of computational resources. It usually possible to reduce the storage requirements and the run time by exploiting special properties of the coefficient matrix, such as sparseness (most elements of a sparse matrix are zero). Hence there are many algorithms dedicated to the solution of large sets of equations, each one being tailored to a particular form of the coefficient matrix (symmetric, banded, sparse, and so on). A well-known collection of these routines is LAPACK—Linear Algebra PACKage, originally written in Fortran77.[1]

We cannot possibly discuss all the special algorithms in the limited space available. The best we can do is to present the basic methods of solution, supplemented by a few useful algorithms for banded coefficient matrices.

Notation

A system of algebraic equations has the form

$$\begin{aligned} A_{11}x_1 + A_{12}x_2 + \cdots + A_{1n}x_n &= b_1 \\ A_{21}x_1 + A_{22}x_2 + \cdots + A_{2n}x_n &= b_2 \\ &\vdots \\ A_{n1}x_1 + A_{n2}x_2 + \cdots + A_{nn}x_n &= b_n \end{aligned} \tag{2.1}$$

[1] LAPACK is the successor of LINPACK, a 1970s and 80s collection of Fortran subroutines.

where the coefficients A_{ij} and the constants b_j are known, and x_i represents the unknowns. In matrix notation the equations are written as

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{2.2}$$

or simply

$$\mathbf{A}\mathbf{x} = \mathbf{b}. \tag{2.3}$$

A particularly useful representation of the equations for computational purposes is the *augmented coefficient matrix* obtained by adjoining the constant vector $\mathbf{b}$ to the coefficient matrix $\mathbf{A}$ in the following fashion:

$$\left[\mathbf{A} \mid \mathbf{b}\right] = \left[\begin{array}{cccc|c} A_{11} & A_{12} & \cdots & A_{1n} & b_1 \\ A_{21} & A_{22} & \cdots & A_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{n3} & b_n \end{array}\right] \tag{2.4}$$

Uniqueness of Solution

A system of n linear equations in n unknowns has a unique solution, provided that the determinant of the coefficient matrix is *nonsingular*; that is, $|\mathbf{A}| \neq 0$. The rows and columns of a nonsingular matrix are *linearly independent* in the sense that no row (or column) is a linear combination of other rows (or columns).

If the coefficient matrix is *singular*, the equations may have an infinite number of solutions or no solutions at all, depending on the constant vector. As an illustration, take the equations

$$2x + y = 3 \qquad 4x + 2y = 6$$

Since the second equation can be obtained by multiplying the first equation by two, any combination of x and y that satisfies the first equation is also a solution of the second equation. The number of such combinations is infinite. In contrast, the equations

$$2x + y = 3 \qquad 4x + 2y = 0$$

have no solution because the second equation, being equivalent to $2x + y = 0$, contradicts the first one. Therefore, any solution that satisfies one equation cannot satisfy the other one.

Ill Conditioning

The obvious question is, What happens when the coefficient matrix is almost singular (i.e., if $|\mathbf{A}|$ is very small). To determine whether the determinant of the coefficient matrix is "small," we need a reference against which the determinant can be measured. This reference is called the *norm* of the matrix and is denoted by $\|\mathbf{A}\|$. We can then say that the determinant is small if

$$|\mathbf{A}| << \|\mathbf{A}\|$$

Several norms of a matrix have been defined in existing literature, such as the *Euclidan norm*

$$\|\mathbf{A}\|_e = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} A_{ij}^2} \tag{2.5a}$$

and the *row-sum norm*, also called the *infinity norm*

$$\|\mathbf{A}\|_\infty = \max_{1\le i\le n} \sum_{j=1}^{n} |A_{ij}| \tag{2.5b}$$

A formal measure of conditioning is the *matrix condition number*, defined as

$$\text{cond}(\mathbf{A}) = \|\mathbf{A}\| \left\|\mathbf{A}^{-1}\right\| \tag{2.5c}$$

If this number is close to unity, the matrix is well conditioned. The condition number increases with the degree of ill conditioning, reaching infinity for a singular matrix. Note that the condition number is not unique, but depends on the choice of the matrix norm. Unfortunately, the condition number is expensive to compute for large matrices. In most cases it is sufficient to gauge conditioning by comparing the determinant with the magnitudes of the elements in the matrix.

If the equations are ill conditioned, small changes in the coefficient matrix result in large changes in the solution. As an illustration, take the equations

$$2x + y = 3 \qquad 2x + 1.001y = 0$$

that have the solution $x = 1501.5$, $y = -3000$. Since $|\mathbf{A}| = 2(1.001) - 2(1) = 0.002$ is much smaller than the coefficients, the equations are ill conditioned. The effect of ill conditioning can verified by changing the second equation to $2x + 1.002y = 0$ and re-solving the equations. The result is $x = 751.5$, $y = -1500$. Note that a 0.1% change in the coefficient of y produced a 100% change in the solution!

Numerical solutions of ill-conditioned equations are not to be trusted. The reason is that the inevitable roundoff errors during the solution process are equivalent to introducing small changes into the coefficient matrix. This in turn introduces large errors into the solution, the magnitude of which depends on the severity of ill conditioning. In suspect cases the determinant of the coefficient matrix should be computed so that the degree of ill conditioning can be estimated. This can be done during or after the solution with only a small computational effort.

Linear Systems

Linear, algebraic equations occur in almost all branches of numerical analysis. But their most visible application in engineering is in the analysis of linear systems (any system whose response is proportional to the input is deemed to be linear). Linear systems include structures, elastic solids, heat flow, seepage of fluids, electromagnetic fields, and electric circuits (i.e., most topics taught in an engineering curriculum).

If the system is discrete, such as a truss or an electric circuit, then its analysis leads directly to linear algebraic equations. In the case of a statically determinate truss, for example, the equations arise when the equilibrium conditions of the joints are written down. The unknowns $x_1, x_2, \ldots, x_n$ represent the forces in the members and the support reactions, and the constants $b_1, b_2, \ldots, b_n$ are the prescribed external loads.

The behavior of continuous systems is described by differential equations, rather than algebraic equations. However, because numerical analysis can deal only with discrete variables, it is first necessary to approximate a differential equation with a system of algebraic equations. The well-known finite difference, finite element, and boundary element methods of analysis work in this manner. They use different approximations to achieve the "discretization," but in each case the final task is the same: solve a system (often a very large system) of linear, algebraic equations.

In summary, the modeling of linear systems invariably gives rise to equations of the form $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{b}$ is the input and $\mathbf{x}$ represents the response of the system. The coefficient matrix $\mathbf{A}$, which reflects the characteristics of the system, is independent of the input. In other words, if the input is changed, the equations have to be solved again with a different $\mathbf{b}$, but the same $\mathbf{A}$. Therefore, it is desirable to have an equation-solving algorithm that can handle any number of constant vectors with minimal computational effort.

Methods of Solution

There are two classes of methods for solving systems of linear, algebraic equations: direct and iterative methods. The common characteristic of *direct methods* is that they transform the original equations into *equivalent equations* (equations that have the same solution) that can be solved more easily. The transformation is carried out by applying the following three operations. These so-called *elementary operations* do not change the solution, but they may affect the determinant of the coefficient matrix as indicated in parenthesis.

1. Exchanging two equations (changes sign of $|\mathbf{A}|$)
2. Multiplying an equation by a nonzero constant (multiplies $|\mathbf{A}|$ by the same constant)
3. Multiplying an equation by a nonzero constant and then subtracting it from another equation (leaves $|\mathbf{A}|$ unchanged)

Iterative, or *indirect methods*, start with a guess of the solution **x** and then repeatedly refine the solution until a certain convergence criterion is reached. Iterative methods are generally less efficient than their direct counterparts because of the large number of iterations required. Yet they do have significant computational advantages if the coefficient matrix is very large and sparsely populated (most coefficients are zero).

Overview of Direct Methods

Table 2.1 lists three popular direct methods, each of which uses elementary operations to produce its own final form of easy-to-solve equations.

Method	Initial form	Final form
Gauss elimination	$\mathbf{Ax} = \mathbf{b}$	$\mathbf{Ux} = \mathbf{c}$
LU decomposition	$\mathbf{Ax} = \mathbf{b}$	$\mathbf{LUx} = \mathbf{b}$
Gauss-Jordan elimination	$\mathbf{Ax} = \mathbf{b}$	$\mathbf{Ix} = \mathbf{c}$

Table 2.1. Three Popular Direct Methods

In Table 2.1 **U** represents an upper triangular matrix, **L** is a lower triangular matrix, and **I** denotes the identity matrix. A square matrix is called *triangular* if it contains only zero elements on one side of the leading diagonal. Thus a 3×3 upper triangular matrix has the form

$$\mathbf{U} = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix}$$

and a 3×3 lower triangular matrix appears as

$$\mathbf{L} = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{bmatrix}$$

Triangular matrices play an important role in linear algebra, because they simplify many computations. For example, consider the equations $\mathbf{Lx} = \mathbf{c}$, or

$$\begin{aligned} L_{11}x_1 &= c_1 \\ L_{21}x_1 + L_{22}x_2 &= c_2 \\ L_{31}x_1 + L_{32}x_2 + L_{33}x_3 &= c_3 \end{aligned}$$

If we solve the equations forward, starting with the first equation, the computations are very easy, because each equation contains only one unknown at a time.

The solution would thus proceed as follows:

$$x_1 = c_1/L_{11}$$
$$x_2 = (c_2 - L_{21}x_1)/L_{22}$$
$$x_3 = (c_3 - L_{31}x_1 - L_{32}x_2)/L_{33}$$

This procedure is known as *forward substitution*. In a similar way, $\mathbf{Ux} = \mathbf{c}$, encountered in Gauss elimination, can easily be solved by *back substitution*, which starts with the last equation and proceeds backward through the equations.

The equations $\mathbf{LUx} = \mathbf{b}$, which are associated with LU decomposition, can also be solved quickly if we replace them with two sets of equivalent equations: $\mathbf{Ly} = \mathbf{b}$ and $\mathbf{Ux} = \mathbf{y}$. Now $\mathbf{Ly} = \mathbf{b}$ can be solved for $\mathbf{y}$ by forward substitution, followed by the solution of $\mathbf{Ux} = \mathbf{y}$ by means of back substitution.

The equations $\mathbf{Ix} = \mathbf{c}$, which are produced by Gauss-Jordan elimination, are equivalent to $\mathbf{x} = \mathbf{c}$ (recall the identity $\mathbf{Ix} = \mathbf{x}$), so that $\mathbf{c}$ is already the solution.

EXAMPLE 2.1

Determine whether the following matrix is singular:

$$\mathbf{A} = \begin{bmatrix} 2.1 & -0.6 & 1.1 \\ 3.2 & 4.7 & -0.8 \\ 3.1 & -6.5 & 4.1 \end{bmatrix}$$

Solution. Laplace's development of the determinant (see Appendix A2) about the first row of $\mathbf{A}$ yields

$$|\mathbf{A}| = 2.1\begin{vmatrix} 4.7 & -0.8 \\ -6.5 & 4.1 \end{vmatrix} - (-0.6)\begin{vmatrix} 3.2 & -0.8 \\ 3.1 & 4.1 \end{vmatrix} + 1.1\begin{vmatrix} 3.2 & 4.7 \\ 3.1 & -6.5 \end{vmatrix}$$
$$= 2.1(14.07) + 0.6(15.60) + 1.1(35.37) = 0$$

Since the determinant is zero, the matrix is singular. It can be verified that the singularity is due to the following row dependency: (row 3) = (3 × row 1) − (row 2).

EXAMPLE 2.2

Solve the equations $\mathbf{Ax} = \mathbf{b}$, where

$$\mathbf{A} = \begin{bmatrix} 8 & -6 & 2 \\ -4 & 11 & -7 \\ 4 & -7 & 6 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 28 \\ -40 \\ 33 \end{bmatrix}$$

knowing that the LU decomposition of the coefficient matrix is (you should verify this)

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

Solution. We first solve the equations $\mathbf{Ly} = \mathbf{b}$ by forward substitution:

$$\begin{aligned} 2y_1 &= 28 & y_1 &= 28/2 = 14 \\ -y_1 + 2y_2 &= -40 & y_2 &= (-40 + y_1)/2 = (-40 + 14)/2 = -13 \\ y_1 - y_2 + y_3 &= 33 & y_3 &= 33 - y_1 + y_2 = 33 - 14 - 13 = 6 \end{aligned}$$

The solution $\mathbf{x}$ is then obtained from $\mathbf{Ux} = \mathbf{y}$ by back substitution:

$$\begin{aligned} 2x_3 &= y_3 & x_3 &= y_3/2 = 6/2 = 3 \\ 4x_2 - 3x_3 &= y_2 & x_2 &= (y_2 + 3x_3)/4 = [-13 + 3(3)]/4 = -1 \\ 4x_1 - 3x_2 + x_3 &= y_1 & x_1 &= (y_1 + 3x_2 - x_3)/4 = [14 + 3(-1) - 3]/4 = 2 \end{aligned}$$

Hence the solution is $\mathbf{x} = \begin{bmatrix} 2 & -1 & 3 \end{bmatrix}^T$.

2.2 Gauss Elimination Method

Introduction

Gauss elimination is the most familiar method for solving simultaneous equations. It consists of two parts: the elimination phase and the back substitution phase. As indicated in Table 2.1, the function of the elimination phase is to transform the equations into the form $\mathbf{Ux} = \mathbf{c}$. The equations are then solved by back substitution. To illustrate the procedure, let us solve the following equations:

$$4x_1 - 2x_2 + x_3 = 11 \quad \text{(a)}$$

$$-2x_1 + 4x_2 - 2x_3 = -16 \quad \text{(b)}$$

$$x_1 - 2x_2 + 4x_3 = 17 \quad \text{(c)}$$

Elimination phase. The elimination phase uses only one of the elementary operations listed in Table 2.1—multiplying one equation (say, equation j) by a constant λ and subtracting it from another equation (equation i). The symbolic representation of this operation is

$$\text{Eq. } (i) \leftarrow \text{Eq. } (i) - \lambda \times \text{Eq. } (j) \tag{2.6}$$

The equation being subtracted, namely Eq. (j), is called the *pivot equation*.

We start the elimination by taking Eq. (a) to be the pivot equation and choosing the multipliers λ so as to eliminate x_1 from Eqs. (b) and (c):

$$\text{Eq. (b)} \leftarrow \text{Eq. (b)} - (-0.5) \times \text{Eq. (a)}$$

$$\text{Eq. (c)} \leftarrow \text{Eq. (c)} - 0.25 \times \text{Eq. (a)}$$

After this transformation, the equations become

$$4x_1 - 2x_2 + x_3 = 11 \quad \text{(a)}$$
$$3x_2 - 1.5x_3 = -10.5 \quad \text{(b)}$$
$$-1.5x_2 + 3.75x_3 = 14.25 \quad \text{(c)}$$

This completes the first pass. Now we pick (b) as the pivot equation and eliminate x_2 from (c):

$$\text{Eq. (c)} \leftarrow \text{Eq. (c)} - (-0.5) \times \text{Eq. (b)}$$

which yields the equations

$$4x_1 - 2x_2 + x_3 = 11 \quad \text{(a)}$$
$$3x_2 - 1.5x_3 = -10.5 \quad \text{(b)}$$
$$3x_3 = 9 \quad \text{(c)}$$

The elimination phase is now complete. The original equations have been replaced by equivalent equations that can be easily solved by back substitution.

As pointed out earlier, the augmented coefficient matrix is a more convenient instrument for performing the computations. Thus the original equations would be written as

$$\left[\begin{array}{rrr|r} 4 & -2 & 1 & 11 \\ -2 & 4 & -2 & -16 \\ 1 & -2 & 4 & 17 \end{array}\right]$$

and the equivalent equations produced by the first and the second passes of Gauss elimination would appear as

$$\left[\begin{array}{rrr|r} 4 & -2 & 1 & 11.00 \\ 0 & 3 & -1.5 & -10.50 \\ 0 & -1.5 & 3.75 & 14.25 \end{array}\right]$$

$$\left[\begin{array}{rrr|r} 4 & -2 & 1 & 11.0 \\ 0 & 3 & -1.5 & -10.5 \\ 0 & 0 & 3 & 9.0 \end{array}\right]$$

It is important to note that the elementary row operation in Eq. (2.6) leaves the determinant of the coefficient matrix unchanged. This is fortunate, because the determinant of a triangular matrix is very easy to compute—it is the product of the diagonal elements (you can verify this quite easily). In other words,

$$|\mathbf{A}| = |\mathbf{U}| = U_{11} \times U_{22} \times \cdots \times U_{nn} \quad (2.7)$$

Back substitution phase. The unknowns can now be computed by back substitution in the manner described previously. Solving Eqs. (c), (b), and (a) in that order, we get

$$x_3 = 9/3 = 3$$
$$x_2 = (-10.5 + 1.5x_3)/3 = [-10.5 + 1.5(3)]/3 = -2$$
$$x_1 = (11 + 2x_2 - x_3)/4 = [11 + 2(-2) - 3]/4 = 1$$

Algorithm for Gauss Elimination Method

Elimination phase. Let us look at the equations at some instant during the elimination phase. Assume that the first k rows of $\mathbf{A}$ have already been transformed to upper triangular form. Therefore, the current pivot equation is the kth equation, and all the equations below it are still to be transformed. This situation is depicted by the following augmented coefficient matrix. Note that the components of $\mathbf{A}$ are not the coefficients of the original equations (except for the first row), because they have been altered by the elimination procedure. The same applies to the components of the constant vector $\mathbf{b}$.

$$\left[\begin{array}{ccccccccc|c}
A_{11} & A_{12} & A_{13} & \cdots & A_{1k} & \cdots & A_{1j} & \cdots & A_{1n} & b_1 \\
0 & A_{22} & A_{23} & \cdots & A_{2k} & \cdots & A_{2j} & \cdots & A_{2n} & b_2 \\
0 & 0 & A_{33} & \cdots & A_{3k} & \cdots & A_{3j} & \cdots & A_{3n} & b_3 \\
\vdots & \vdots & \vdots & & \vdots & & \vdots & & \vdots & \vdots \\
0 & 0 & 0 & \cdots & A_{kk} & \cdots & A_{kj} & \cdots & A_{kn} & b_k \\
\hline
\vdots & \vdots & \vdots & & \vdots & & \vdots & & \vdots & \vdots \\
0 & 0 & 0 & \cdots & A_{ik} & \cdots & A_{ij} & \cdots & A_{in} & b_i \\
\vdots & \vdots & \vdots & & \vdots & & \vdots & & \vdots & \vdots \\
0 & 0 & 0 & \cdots & A_{nk} & \cdots & A_{nj} & \cdots & A_{nn} & b_n
\end{array}\right]$$

$\leftarrow$ pivot row

$\leftarrow$ row being transformed

Let the ith row be a typical row below the pivot equation that is to be transformed, meaning that the element A_{ik} is to be eliminated. We can achieve this by multiplying the pivot row by $\lambda = A_{ik}/A_{kk}$ and subtracting it from the ith row. The corresponding changes in the ith row are

$$A_{ij} \leftarrow A_{ij} - \lambda A_{kj}, \quad j = k, k+1, \ldots, n \tag{2.8a}$$
$$b_i \leftarrow b_i - \lambda b_k \tag{2.8b}$$

To transform the entire coefficient matrix to upper triangular form, k and i in Eqs. (2.8) must have the ranges $k = 1, 2, \ldots, n-1$ (chooses the pivot row), $i = k+1, k+2 \ldots, n$ (chooses the row to be transformed). The algorithm for the elimination phase now almost writes itself:

```
for k in range(0,n-1):
    for i in range(k+1,n):
```

```
if a[i,k] != 0.0:
    lam = a[i,k]/a[k,k]
    a[i,k+1:n] = a[i,k+1:n] - lam*a[k,k+1:n]
    b[i] = b[i] - lam*b[k]
```

To avoid unnecessary operations, the preceding algorithm departs slightly from Eqs. (2.8) in the following ways:

- If A_{ik} happens to be zero, the transformation of row i is skipped.
- The index j in Eq. (2.8a) starts with $k+1$ rather than k. Therefore, A_{ik} is not replaced by zero, but retains its original value. As the solution phase never accesses the lower triangular portion of the coefficient matrix anyway, its contents are irrelevant.

Back substitution phase. After Gauss elimination the augmented coefficient matrix has the form

$$\left[\mathbf{A} \mid \mathbf{b}\right] = \left[\begin{array}{ccccc|c} A_{11} & A_{12} & A_{13} & \cdots & A_{1n} & b_1 \\ 0 & A_{22} & A_{23} & \cdots & A_{2n} & b_2 \\ 0 & 0 & A_{33} & \cdots & A_{3n} & b_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & A_{nn} & b_n \end{array}\right]$$

The last equation, $A_{nn}x_n = b_n$, is solved first, yielding

$$x_n = b_n/A_{nn} \tag{2.9}$$

Consider now the stage of back substitution where $x_n,\ x_{n-1}, \ldots,\ x_{k+1}$ have been already been computed (in that order), and we are about to determine x_k from the kth equation

$$A_{kk}x_k + A_{k,k+1}x_{k+1} + \cdots + A_{kn}x_n = b_k$$

The solution is

$$x_k = \left(b_k - \sum_{j=k+1}^{n} A_{kj}x_j\right)\frac{1}{A_{kk}}, \quad k = n-1, n-2, \ldots, 1 \tag{2.10}$$

The corresponding algorithm for back substitution is

```
for k in range(n-1,-1,-1):
        x[k]=(b[k] - dot(a[k,k+1:n],x[k+1:n]))/a[k,k]
```

Operation count. The execution time of an algorithm depends largely on the number of long operations (multiplications and divisions) performed. It can be shown that Gauss elimination contains approximately $n^3/3$ such operations (n is the number of equations) in the elimination phase, and $n^2/2$ operations in back substitution. These numbers show that most of the computation time goes into the elimination phase. Moreover, the time increases very rapidly with the number of equations.

■ gaussElimin

The function `gaussElimin` combines the elimination and the back substitution phases. During back substitution `b` is overwritten by the solution vector `x`, so that `b` contains the solution upon exit.

```
## module gaussElimin
''' x = gaussElimin(a,b).
    Solves [a]{b} = {x} by Gauss elimination.
'''
import numpy as np

def gaussElimin(a,b):
    n = len(b)
  # Elimination Phase
    for k in range(0,n-1):
        for i in range(k+1,n):
           if a[i,k] != 0.0:
               lam = a [i,k]/a[k,k]
               a[i,k+1:n] = a[i,k+1:n] - lam*a[k,k+1:n]
               b[i] = b[i] - lam*b[k]
  # Back substitution
    for k in range(n-1,-1,-1):
        b[k] = (b[k] - np.dot(a[k,k+1:n],b[k+1:n]))/a[k,k]
    return b
```

Multiple Sets of Equations

As mentioned earlier, it is frequently necessary to solve the equations $\mathbf{Ax} = \mathbf{b}$ for several constant vectors. Let there be m such constant vectors, denoted by $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_m$, and let the corresponding solution vectors be $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m$. We denote multiple sets of equations by $\mathbf{AX} = \mathbf{B}$, where

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_m \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{bmatrix}$$

are $n \times m$ matrices whose columns consist of solution vectors and constant vectors, respectively.

An economical way to handle such equations during the elimination phase is to include all m constant vectors in the augmented coefficient matrix, so that they are transformed simultaneously with the coefficient matrix. The solutions are then obtained by back substitution in the usual manner, one vector at a time. It would be quite easy to make the corresponding changes in `gaussElimin`. However, the LU decomposition method, described in the next section, is more versatile in handling multiple constant vectors.

EXAMPLE 2.3

Use Gauss elimination to solve the equations $\mathbf{AX} = \mathbf{B}$, where

$$\mathbf{A} = \begin{bmatrix} 6 & -4 & 1 \\ -4 & 6 & -4 \\ 1 & -4 & 6 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} -14 & 22 \\ 36 & -18 \\ 6 & 7 \end{bmatrix}$$

Solution. The augmented coefficient matrix is

$$\left[\begin{array}{ccc|cc} 6 & -4 & 1 & -14 & 22 \\ -4 & 6 & -4 & 36 & -18 \\ 1 & -4 & 6 & 6 & 7 \end{array}\right]$$

The elimination phase consists of the following two passes:

$$\text{row } 2 \leftarrow \text{row } 2 + (2/3) \times \text{row } 1$$

$$\text{row } 3 \leftarrow \text{row } 3 - (1/6) \times \text{row } 1$$

$$\left[\begin{array}{ccc|cc} 6 & -4 & 1 & -14 & 22 \\ 0 & 10/3 & -10/3 & 80/3 & -10/3 \\ 0 & -10/3 & 35/6 & 25/3 & 10/3 \end{array}\right]$$

and

$$\text{row } 3 \leftarrow \text{row } 3 + \text{row } 2$$

$$\left[\begin{array}{ccc|cc} 6 & -4 & 1 & -14 & 22 \\ 0 & 10/3 & -10/3 & 80/3 & -10/3 \\ 0 & 0 & 5/2 & 35 & 0 \end{array}\right]$$

In the solution phase, we first compute $\mathbf{x}_1$ by back substitution:

$$X_{31} = \frac{35}{5/2} = 14$$

$$X_{21} = \frac{80/3 + (10/3)X_{31}}{10/3} = \frac{80/3 + (10/3)14}{10/3} = 22$$

$$X_{11} = \frac{-14 + 4X_{21} - X_{31}}{6} = \frac{-14 + 4(22) - 14}{6} = 10$$

Thus the first solution vector is

$$\mathbf{x}_1 = \begin{bmatrix} X_{11} & X_{21} & X_{31} \end{bmatrix}^T = \begin{bmatrix} 10 & 22 & 14 \end{bmatrix}^T$$

The second solution vector is computed next, also using back substitution:

$$X_{32} = 0$$

$$X_{22} = \frac{-10/3 + (10/3)X_{32}}{10/3} = \frac{-10/3 + 0}{10/3} = -1$$

$$X_{12} = \frac{22 + 4X_{22} - X_{32}}{6} = \frac{22 + 4(-1) - 0}{6} = 3$$

Therefore,

$$\mathbf{x}_2 = \begin{bmatrix} X_{12} & X_{22} & X_{32} \end{bmatrix}^T = \begin{bmatrix} 3 & -1 & 0 \end{bmatrix}^T$$

EXAMPLE 2.4

An $n \times n$ Vandermode matrix $\mathbf{A}$ is defined by

$$A_{ij} = v_i^{n-j}, \quad i = 1, 2, \ldots, n, \quad j = 1, 2, \ldots, n$$

where $\mathbf{v}$ is a vector. Use the function `gaussElimin` to compute the solution of $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is the 6×6 Vandermode matrix generated from the vector

$$\mathbf{v} = \begin{bmatrix} 1.0 & 1.2 & 1.4 & 1.6 & 1.8 & 2.0 \end{bmatrix}^T$$

and

$$\mathbf{b} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}^T$$

Also evaluate the accuracy of the solution (Vandermode matrices tend to be ill conditioned).

Solution

```
#!/usr/bin/python
## example2_4
import numpy as np
from gaussElimin import *

def vandermode(v):
    n = len(v)
    a = np.zeros((n,n))
    for j in range(n):
        a[:,j] = v**(n-j-1)
    return a

v = np.array([1.0, 1.2, 1.4, 1.6, 1.8, 2.0])
b = np.array([0.0, 1.0, 0.0, 1.0, 0.0, 1.0])
a = vandermode(v)
aOrig = a.copy()    # Save original matrix
bOrig = b.copy()    # and the constant vector
x = gaussElimin(a,b)
det = np.prod(np.diagonal(a))
print('x =\n',x)
print('\ndet =',det)
print('\nCheck result: [a]{x} - b =\n',np.dot(aOrig,x) - bOrig)
input("\nPress return to exit")
```

The program produced the following results:

```
x =
[   416.66666667  -3125.00000004   9250.00000012 -13500.00000017
   9709.33333345  -2751.00000003]

det = -1.13246207999e-006

Check result: [a]{x} - b =
 [0.00000000e+00  3.63797881e-12  0.00000000e+00  1.45519152e-11
  0.00000000e+00  5.82076609e-11]
```

As the determinant is quite small relative to the elements of **A** (you may want to print **A** to verify this), we expect a detectable roundoff error. Inspection of **x** leads us to suspect that the exact solution is

$$\mathbf{x} = \begin{bmatrix} 1250/3 & -3125 & 9250 & -13500 & 29128/3 & -2751 \end{bmatrix}^T$$

in which case the numerical solution would be accurate to about 10 decimal places. Another way to gauge the accuracy of the solution is to compute $\mathbf{Ax} - \mathbf{b}$ (the result should be $\mathbf{0}$). The printout indicates that the solution is indeed accurate to at least 10 decimal places.

2.3 LU Decomposition Methods

Introduction

It is possible to show that any square matrix **A** can be expressed as a product of a lower triangular matrix **L** and an upper triangular matrix **U**:

$$\mathbf{A} = \mathbf{LU} \tag{2.11}$$

The process of computing **L** and **U** for a given **A** is known as *LU decomposition* or *LU factorization*. LU decomposition is not unique (the combinations of **L** and **U** for a prescribed **A** are endless), unless certain constraints are placed on **L** or **U**. These constraints distinguish one type of decomposition from another. Three commonly used decompositions are listed in Table 2.2.

Name	Constraints
Doolittle's decomposition	$L_{ii} = 1, \quad i = 1, 2, \ldots, n$
Crout's decomposition	$U_{ii} = 1, \quad i = 1, 2, \ldots, n$
Choleski's decomposition	$\mathbf{L} = \mathbf{U}^T$

Table 2.2. Three Commonly Used Decompositions

After decomposing **A**, it is easy to solve the equations $\mathbf{Ax} = \mathbf{b}$, as pointed out in Section 2.1. We first rewrite the equations as $\mathbf{LUx} = \mathbf{b}$. After using the notation

$\mathbf{Ux} = \mathbf{y}$, the equations become

$$\mathbf{Ly} = \mathbf{b},$$

which can be solved for $\mathbf{y}$ by forward substitution. Then

$$\mathbf{Ux} = \mathbf{y}$$

will yield $\mathbf{x}$ by the back substitution process.

The advantage of LU decomposition over the Gauss elimination method is that once $\mathbf{A}$ is decomposed, we can solve $\mathbf{Ax} = \mathbf{b}$ for as many constant vectors $\mathbf{b}$ as we please. The cost of each additional solution is relatively small, because the forward and back substitution operations are much less time consuming than the decomposition process.

Doolittle's Decomposition Method

Decomposition phase. Doolittle's decomposition is closely related to Gauss elimination. To illustrate the relationship, consider a 3×3 matrix $\mathbf{A}$ and assume that there exist triangular matrices

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{bmatrix} \qquad \mathbf{U} = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix}$$

such that $\mathbf{A} = \mathbf{LU}$. After completing the multiplication on the right-hand side, we get

$$\mathbf{A} = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ U_{11}L_{21} & U_{12}L_{21} + U_{22} & U_{13}L_{21} + U_{23} \\ U_{11}L_{31} & U_{12}L_{31} + U_{22}L_{32} & U_{13}L_{31} + U_{23}L_{32} + U_{33} \end{bmatrix} \tag{2.12}$$

Let us now apply Gauss elimination to Eq. (2.12). The first pass of the elimination procedure consists of choosing the first row as the pivot row and applying the elementary operations

$$\text{row } 2 \leftarrow \text{row } 2 - L_{21} \times \text{row } 1 \text{ (eliminates} A_{21})$$

$$\text{row } 3 \leftarrow \text{row } 3 - L_{31} \times \text{row } 1 \text{ (eliminates} A_{31})$$

The result is

$$\mathbf{A}' = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & U_{22}L_{32} & U_{23}L_{32} + U_{33} \end{bmatrix}$$

In the next pass we take the second row as the pivot row, and use the operation

$$\text{row } 3 \leftarrow \text{row } 3 - L_{32} \times \text{row } 2 \text{ (eliminates} A_{32})$$

ending up with

$$\mathbf{A}'' = \mathbf{U} = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix}$$

The foregoing illustration reveals two important features of Doolittle's decomposition:

1. The matrix **U** is identical to the upper triangular matrix that results from Gauss elimination.
2. The off-diagonal elements of **L** are the pivot equation multipliers used during Gauss elimination; that is, L_{ij} is the multiplier that eliminated A_{ij}.

It is usual practice to store the multipliers in the lower triangular portion of the coefficient matrix, replacing the coefficients as they are eliminated (L_{ij} replacing A_{ij}). The diagonal elements of **L** do not have to be stored, because it is understood that each of them is unity. The final form of the coefficient matrix would thus be the following mixture of **L** and **U**:

$$[\mathbf{L}\backslash\mathbf{U}] = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ L_{21} & U_{22} & U_{23} \\ L_{31} & L_{32} & U_{33} \end{bmatrix} \tag{2.13}$$

The algorithm for Doolittle's decomposition is thus identical to the Gauss elimination procedure in `gaussElimin`, except that each multiplier λ is now stored in the lower triangular portion of **A**:

```
for k in range(0,n-1):
    for i in range(k+1,n):
        if a[i,k] != 0.0:
            lam = a[i,k]/a[k,k]
            a[i,k+1:n] = a[i,k+1:n] - lam*a[k,k+1:n]
            a[i,k] = lam
```

Solution phase. Consider now the procedure for the solution of $\mathbf{Ly} = \mathbf{b}$ by forward substitution. The scalar form of the equations is (recall that $L_{ii} = 1$)

$$\begin{aligned} y_1 &= b_1 \\ L_{21}y_1 + y_2 &= b_2 \\ &\vdots \\ L_{k1}y_1 + L_{k2}y_2 + \cdots + L_{k,k-1}y_{k-1} + y_k &= b_k \\ &\vdots \end{aligned}$$

Solving the kth equation for y_k yields

$$y_k = b_k - \sum_{j=1}^{k-1} L_{kj} y_j, \quad k = 2, 3, \ldots, n \tag{2.14}$$

Therefore, the forward substitution algorithm is

```
y[0] = b[0]
for k in range(1,n):
    y[k] = b[k] - dot(a[k,0:k],y[0:k])
```

The back substitution phase for solving $\mathbf{Ux} = \mathbf{y}$ is identical to what was used in the Gauss elimination method.

■ LUdecomp

This module contains both the decomposition and solution phases. The decomposition phase returns the matrix [**L****U**] shown in Eq. (2.13). In the solution phase, the contents of **b** are replaced by **y** during forward substitution Similarly, the back substitution overwrites **y** with the solution **x**.

```
## module LUdecomp
''' a = LUdecomp(a)
   LUdecomposition: [L][U] = [a]

   x = LUsolve(a,b)
   Solution phase: solves [L][U]{x} = {b}
'''
import numpy as np

def LUdecomp(a):
    n = len(a)
    for k in range(0,n-1):
        for i in range(k+1,n):
           if a[i,k] != 0.0:
               lam = a [i,k]/a[k,k]
               a[i,k+1:n] = a[i,k+1:n] - lam*a[k,k+1:n]
               a[i,k] = lam
    return a

def LUsolve(a,b):
    n = len(a)
    for k in range(1,n):
        b[k] = b[k] - np.dot(a[k,0:k],b[0:k])
    b[n-1] = b[n-1]/a[n-1,n-1]
    for k in range(n-2,-1,-1):
       b[k] = (b[k] - np.dot(a[k,k+1:n],b[k+1:n]))/a[k,k]
    return b
```

Choleski's Decomposition Method

Choleski's decomposition $\mathbf{A} = \mathbf{L}\mathbf{L}^T$ has two limitations:

1. Since $\mathbf{L}\mathbf{L}^T$ is always a symmetric matrix, Choleski's decomposition requires $\mathbf{A}$ to be *symmetric*.
2. The decomposition process involves taking square roots of certain combinations of the elements of $\mathbf{A}$. It can be shown that to avoid square roots of negative numbers $\mathbf{A}$ must be *positive definite*.

Choleski's decomposition contains approximately $n^3/6$ long operations plus n square root computations. This is about half the number of operations required in LU decomposition. The relative efficiency of Choleski's decomposition is due to its exploitation of symmetry.

Let us start by looking at Choleski's decomposition

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T \tag{2.15}$$

of a 3×3 matrix:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{bmatrix} \begin{bmatrix} L_{11} & L_{21} & L_{31} \\ 0 & L_{22} & L_{32} \\ 0 & 0 & L_{33} \end{bmatrix}$$

After completing the matrix multiplication on the right-hand side, we get

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} L_{11}^2 & L_{11}L_{21} & L_{11}L_{31} \\ L_{11}L_{21} & L_{21}^2 + L_{22}^2 & L_{21}L_{31} + L_{22}L_{32} \\ L_{11}L_{31} & L_{21}L_{31} + L_{22}L_{32} & L_{31}^2 + L_{32}^2 + L_{33}^2 \end{bmatrix} \tag{2.16}$$

Note that the right-hand-side matrix is symmetric, as pointed out earlier. Equating the matrices $\mathbf{A}$ and $\mathbf{L}\mathbf{L}^T$ element by element, we obtain six equations (because of symmetry only lower or upper triangular elements have to be considered) in the six unknown components of $\mathbf{L}$. By solving these equations in a certain order, it is possible to have only one unknown in each equation.

Consider the lower triangular portion of each matrix in Eq. (2.16) (the upper triangular portion would do as well). By equating the elements in the first column, starting with the first row and proceeding downward, we can compute L_{11}, L_{21}, and L_{31} in that order:

$$\begin{aligned} A_{11} &= L_{11}^2 & L_{11} &= \sqrt{A_{11}} \\ A_{21} &= L_{11}L_{21} & L_{21} &= A_{21}/L_{11} \\ A_{31} &= L_{11}L_{31} & L_{31} &= A_{31}/L_{11} \end{aligned}$$

The second column, starting with second row, yields L_{22} and L_{32}:

$$\begin{aligned} A_{22} &= L_{21}^2 + L_{22}^2 & L_{22} &= \sqrt{A_{22} - L_{21}^2} \\ A_{32} &= L_{21}L_{31} + L_{22}L_{32} & L_{32} &= (A_{32} - L_{21}L_{31})/L_{22} \end{aligned}$$

Finally the third column, third row gives us L_{33}:

$$A_{33} = L_{31}^2 + L_{32}^2 + L_{33}^2 \qquad L_{33} = \sqrt{A_{33} - L_{31}^2 - L_{32}^2}$$

We can now extrapolate the results for an $n \times n$ matrix. We observe that a typical element in the lower triangular portion of $\mathbf{LL}^T$ is of the form

$$(\mathbf{LL}^T)_{ij} = L_{i1}L_{j1} + L_{i2}L_{j2} + \cdots + L_{ij}L_{jj} = \sum_{k=1}^{j} L_{ik}L_{jk}, \quad i \geq j$$

Equating this term to the corresponding element of $\mathbf{A}$ yields

$$A_{ij} = \sum_{k=1}^{j} L_{ik}L_{jk}, \quad i = j, j+1, \ldots, n, \quad j = 1, 2, \ldots, n \tag{2.17}$$

The range of indices shown limits the elements to the lower triangular part. For the first column ($j = 1$), we obtain from Eq. (2.17)

$$L_{11} = \sqrt{A_{11}} \qquad L_{i1} = A_{i1}/L_{11}, \quad i = 2, 3, \ldots, n \tag{2.18}$$

Proceeding to other columns, we observe that the unknown in Eq. (2.17) is L_{ij} (the other elements of $\mathbf{L}$ appearing in the equation have already been computed). Taking the term containing L_{ij} outside the summation in Eq. (2.17), we obtain

$$A_{ij} = \sum_{k=1}^{j-1} L_{ik}L_{jk} + L_{ij}L_{jj}$$

If $i = j$ (a diagonal term), the solution is

$$L_{jj} = \sqrt{A_{jj} - \sum_{k=1}^{j-1} L_{jk}^2}, \quad j = 2, 3, \ldots, n \tag{2.19}$$

For a nondiagonal term we get

$$L_{ij} = \left(A_{ij} - \sum_{k=1}^{j-1} L_{ik}L_{jk} \right) / L_{jj}, \quad j = 2, 3, \ldots, n-1, \quad i = j+1, j+2, \ldots, n. \tag{2.20}$$

■ choleski

Before presenting the algorithm for Choleski's decomposition, we make a useful observation: A_{ij} appears only in the formula for L_{ij}. Therefore, once L_{ij} has been computed, A_{ij} is no longer needed. This makes it possible to write the elements of $\mathbf{L}$ over the lower triangular portion of $\mathbf{A}$ as they are computed. The elements above the leading diagonal of $\mathbf{A}$ will remain untouched. The function listed next implements Choleski's decomposition. If a negative diagonal term is encountered during decomposition, an error message is printed and the program is terminated.

After the coefficient matrix **A** has been decomposed, the solution of $\mathbf{Ax} = \mathbf{b}$ can be obtained by the usual forward and back substitution operations. The function `choleskiSol` (given here without derivation) carries out the solution phase.

```
## module choleski
''' L = choleski(a)
    Choleski decomposition: [L][L]transpose = [a]

    x = choleskiSol(L,b)
    Solution phase of Choleski's decomposition method
'''
import numpy as np
import math
import error

def choleski(a):
    n = len(a)
    for k in range(n):
        try:
            a[k,k] = math.sqrt(a[k,k]  \
                   - np.dot(a[k,0:k],a[k,0:k]))
        except ValueError:
            error.err('Matrix is not positive definite')
        for i in range(k+1,n):
            a[i,k] = (a[i,k] - np.dot(a[i,0:k],a[k,0:k]))/a[k,k]
    for k in range(1,n): a[0:k,k] = 0.0
    return a

def choleskiSol(L,b):
    n = len(b)
  # Solution of [L]{y} = {b}
    for k in range(n):
        b[k] = (b[k] - np.dot(L[k,0:k],b[0:k]))/L[k,k]
  # Solution of [L_transpose]{x} = {y}
    for k in range(n-1,-1,-1):
        b[k] = (b[k] - np.dot(L[k+1:n,k],b[k+1:n]))/L[k,k]
    return b
```

Other Methods

Crout's decomposition. Recall that the various decompositions $\mathbf{A} = \mathbf{LU}$ are characterized by the constraints placed on the elements of **L** or **U**. In Doolittle's decomposition the diagonal elements of **L** were set to 1. An equally viable method is Crout's

decomposition, where the 1's lie on the diagonal of **U**. There is little difference in the performance of the two methods.

Gauss-Jordan Elimination. The Gauss-Jordan method is essentially Gauss elimination taken to its limit. In the Gauss elimination method only the equations that lie below the pivot equation are transformed. In the Gauss-Jordan method the elimination is also carried out on equations above the pivot equation, resulting in a diagonal coefficient matrix. The main disadvantage of Gauss-Jordan elimination is that it involves about $n^3/2$ long operations, which is 1.5 times the number required in Gauss elimination.

EXAMPLE 2.5

Use Doolittle's decomposition method to solve the equations $\mathbf{Ax} = \mathbf{b}$, where

$$\mathbf{A} = \begin{bmatrix} 1 & 4 & 1 \\ 1 & 6 & -1 \\ 2 & -1 & 2 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 7 \\ 13 \\ 5 \end{bmatrix}$$

Solution. We first decompose **A** by Gauss elimination. The first pass consists of the elementary operations

$$\text{row } 2 \leftarrow \text{row } 2 - 1 \times \text{ row } 1 \text{ (eliminates } A_{21})$$
$$\text{row } 3 \leftarrow \text{row } 3 - 2 \times \text{ row } 1 \text{ (eliminates } A_{31})$$

Storing the multipliers $L_{21} = 1$ and $L_{31} = 2$ in place of the eliminated terms, we obtain

$$\mathbf{A}' = \begin{bmatrix} 1 & 4 & 1 \\ 1 & 2 & -2 \\ 2 & -9 & 0 \end{bmatrix}$$

The second pass of Gauss elimination uses the operation

$$\text{row } 3 \leftarrow \text{row } 3 - (-4.5) \times \text{ row } 2 \text{ (eliminates } A_{32})$$

Storing the multiplier $L_{32} = -4.5$ in place of A_{32}, we get

$$\mathbf{A}'' = [\mathbf{L}\backslash\mathbf{U}] = \begin{bmatrix} 1 & 4 & 1 \\ 1 & 2 & -2 \\ 2 & -4.5 & -9 \end{bmatrix}$$

The decomposition is now complete, with

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & -4.5 & 1 \end{bmatrix} \qquad \mathbf{U} = \begin{bmatrix} 1 & 4 & 1 \\ 0 & 2 & -2 \\ 0 & 0 & -9 \end{bmatrix}$$

Solution of $\mathbf{Ly} = \mathbf{b}$ by forward substitution comes next. The augmented coefficient form of the equations is

$$\left[\begin{array}{c|c} \mathbf{L} & \mathbf{b} \end{array}\right] = \left[\begin{array}{ccc|c} 1 & 0 & 0 & 7 \\ 1 & 1 & 0 & 13 \\ 2 & -4.5 & 1 & 5 \end{array}\right]$$

The solution is

$$\begin{aligned} y_1 &= 7 \\ y_2 &= 13 - y_1 = 13 - 7 = 6 \\ y_3 &= 5 - 2y_1 + 4.5y_2 = 5 - 2(7) + 4.5(6) = 18 \end{aligned}$$

Finally, the equations $\mathbf{Ux} = \mathbf{y}$, or

$$\left[\begin{array}{c|c} \mathbf{U} & \mathbf{y} \end{array}\right] = \left[\begin{array}{ccc|c} 1 & 4 & 1 & 7 \\ 0 & 2 & -2 & 6 \\ 0 & 0 & -9 & 18 \end{array}\right]$$

are solved by back substitution. This yields

$$\begin{aligned} x_3 &= \frac{18}{-9} = -2 \\ x_2 &= \frac{6 + 2x_3}{2} = \frac{6 + 2(-2)}{2} = 1 \\ x_1 &= 7 - 4x_2 - x_3 = 7 - 4(1) - (-2) = 5 \end{aligned}$$

EXAMPLE 2.6

Compute Choleski's decomposition of the matrix

$$\mathbf{A} = \begin{bmatrix} 4 & -2 & 2 \\ -2 & 2 & -4 \\ 2 & -4 & 11 \end{bmatrix}$$

Solution. First we note that $\mathbf{A}$ is symmetric. Therefore, Choleski's decomposition is applicable, provided that the matrix is also positive definite. An a priori test for positive definiteness is not needed, because the decomposition algorithm contains its own test: If a square root of a negative number is encountered, the matrix is not positive definite and the decomposition fails.

Substituting the given matrix for $\mathbf{A}$ in Eq. (2.16) we obtain

$$\begin{bmatrix} 4 & -2 & 2 \\ -2 & 2 & -4 \\ 2 & -4 & 11 \end{bmatrix} = \begin{bmatrix} L_{11}^2 & L_{11}L_{21} & L_{11}L_{31} \\ L_{11}L_{21} & L_{21}^2 + L_{22}^2 & L_{21}L_{31} + L_{22}L_{32} \\ L_{11}L_{31} & L_{21}L_{31} + L_{22}L_{32} & L_{31}^2 + L_{32}^2 + L_{33}^2 \end{bmatrix}$$

Equating the elements in the lower (or upper) triangular portions yields

$$L_{11} = \sqrt{4} = 2$$

$$L_{21} = -2/L_{11} = -2/2 = -1$$

$$L_{31} = 2/L_{11} = 2/2 = 1$$

$$L_{22} = \sqrt{2 - L_{21}^2} = \sqrt{2 - 1^2} = 1$$

$$L_{32} = \frac{-4 - L_{21}L_{31}}{L_{22}} = \frac{-4 - (-1)(1)}{1} = -3$$

$$L_{33} = \sqrt{11 - L_{31}^2 - L_{32}^2} = \sqrt{11 - (1)^2 - (-3)^2} = 1$$

Therefore,

$$\mathbf{L} = \begin{bmatrix} 2 & 0 & 0 \\ -1 & 1 & 0 \\ 1 & -3 & 1 \end{bmatrix}$$

The result can easily be verified by performing the multiplication $\mathbf{L}\mathbf{L}^T$.

EXAMPLE 2.7

Write a program that solves $\mathbf{AX} = \mathbf{B}$ with Doolittle's decomposition method and computes $|\mathbf{A}|$. Use the functions `LUdecomp` and `LUsolve`. Test the program with

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

Solution

```
#!/usr/bin/python
## example2_7
import numpy as np
from LUdecomp import *

a = np.array([[ 3.0, -1.0,  4.0], \
              [-2.0,  0.0,  5.0], \
              [ 7.0,  2.0, -2.0]])
b = np.array([[ 6.0,  3.0,  7.0], \
              [-4.0,  2.0, -5.0]])
a = LUdecomp(a)                    # Decompose [a]
det = np.prod(np.diagonal(a))
print("\nDeterminant =",det)
for i in range(len(b)):            # Back-substitute one
   x = LUsolve(a,b[i])             # constant vector at a time
   print("x",i+1,"=",x)
input("\nPress return to exit")
```

The output of the program is

```
Determinant = -77.0
x 1 = [ 1.  1.  1.]
x 2 = [ -1.00000000e+00   1.00000000e+00   2.30695693e-17]
```

EXAMPLE 2.8

Solve the equations $\mathbf{Ax} = \mathbf{b}$ by Choleski's decomposition, where

$$\mathbf{A} = \begin{bmatrix} 1.44 & -0.36 & 5.52 & 0.00 \\ -0.36 & 10.33 & -7.78 & 0.00 \\ 5.52 & -7.78 & 28.40 & 9.00 \\ 0.00 & 0.00 & 9.00 & 61.00 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 0.04 \\ -2.15 \\ 0 \\ 0.88 \end{bmatrix}$$

Also check the solution.

Solution

```
#!/usr/bin/python
## example2_8
import numpy as np
from choleski import *

a = np.array([[ 1.44, -0.36,  5.52,  0.0], \
              [-0.36, 10.33, -7.78,  0.0], \
              [ 5.52, -7.78, 28.40,  9.0], \
              [ 0.0,   0.0,   9.0,  61.0]])
b = np.array([0.04, -2.15, 0.0, 0.88])
aOrig = a.copy()
L = choleski(a)
x = choleskiSol(L,b)
print("x =",x)
print('\nCheck: A*x =\n',np.dot(aOrig,x))
input("\nPress return to exit")
```

The output is

```
x = [ 3.09212567 -0.73871706 -0.8475723   0.13947788]

Check: A*x =
[4.00000000e-02 -2.15000000e+00 -3.55271368e-15  8.80000000e-01]
```

PROBLEM SET 2.1

1. By evaluating the determinant, classify the following matrices as singular, ill conditioned, or well conditioned:

$$\text{(a)}\quad \mathbf{A}=\begin{bmatrix}1&2&3\\2&3&4\\3&4&5\end{bmatrix} \qquad \text{(b)}\quad \mathbf{A}=\begin{bmatrix}2.11&-0.80&1.72\\-1.84&3.03&1.29\\-1.57&5.25&4.30\end{bmatrix}$$

$$\text{(c)}\quad \mathbf{A}=\begin{bmatrix}2&-1&0\\-1&2&-1\\0&-1&2\end{bmatrix} \qquad \text{(d)}\quad \mathbf{A}=\begin{bmatrix}4&3&-1\\7&-2&3\\5&-18&13\end{bmatrix}$$

2. Given the LU decomposition $\mathbf{A}=\mathbf{LU}$, determine $\mathbf{A}$ and $|\mathbf{A}|$:

$$\text{(a)}\quad \mathbf{L}=\begin{bmatrix}1&0&0\\1&1&0\\1&5/3&1\end{bmatrix} \quad \mathbf{U}=\begin{bmatrix}1&2&4\\0&3&21\\0&0&0\end{bmatrix}$$

$$\text{(b)}\quad \mathbf{L}=\begin{bmatrix}2&0&0\\-1&1&0\\1&-3&1\end{bmatrix} \quad \mathbf{U}=\begin{bmatrix}2&-1&1\\0&1&-3\\0&0&1\end{bmatrix}$$

3. Use the results of LU decomposition

$$\mathbf{A}=\mathbf{LU}=\begin{bmatrix}1&0&0\\3/2&1&0\\1/2&11/13&1\end{bmatrix}\begin{bmatrix}2&-3&-1\\0&13/2&-7/2\\0&0&32/13\end{bmatrix}$$

to solve $\mathbf{Ax}=\mathbf{b}$, where $\mathbf{b}^T=\begin{bmatrix}1&-1&2\end{bmatrix}$.

4. Use Gauss elimination to solve the equations $\mathbf{Ax}=\mathbf{b}$, where

$$\mathbf{A}=\begin{bmatrix}2&-3&-1\\3&2&-5\\2&4&-1\end{bmatrix} \qquad \mathbf{b}=\begin{bmatrix}3\\-9\\-5\end{bmatrix}$$

5. Solve the equations $\mathbf{AX}=\mathbf{B}$ by Gauss elimination, where

$$\mathbf{A}=\begin{bmatrix}2&0&-1&0\\0&1&2&0\\-1&2&0&1\\0&0&1&-2\end{bmatrix} \qquad \mathbf{B}=\begin{bmatrix}1&0\\0&0\\0&1\\0&0\end{bmatrix}$$

6. Solve the equations $\mathbf{Ax}=\mathbf{b}$ by Gauss elimination, where

$$\mathbf{A}=\begin{bmatrix}0&0&2&1&2\\0&1&0&2&-1\\1&2&0&-2&0\\0&0&0&-1&1\\0&1&-1&1&-1\end{bmatrix} \qquad \mathbf{b}=\begin{bmatrix}1\\1\\-4\\-2\\-1\end{bmatrix}$$

Hint: Reorder the equations before solving.

7. Find **L** and **U** so that

$$\mathbf{A} = \mathbf{LU} = \begin{bmatrix} 4 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 4 \end{bmatrix}$$

using (a) Doolittle's decomposition; (b) Cholesky's decomposition.

8. Use Doolittle's decomposition method to solve $\mathbf{Ax} = \mathbf{b}$, where

$$\mathbf{A} = \begin{bmatrix} -3 & 6 & -4 \\ 9 & -8 & 24 \\ -12 & 24 & -26 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} -3 \\ 65 \\ -42 \end{bmatrix}$$

9. Solve the equations $\mathbf{AX} = \mathbf{b}$ by Doolittle's decomposition method, where

$$\mathbf{A} = \begin{bmatrix} 2.34 & -4.10 & 1.78 \\ -1.98 & 3.47 & -2.22 \\ 2.36 & -15.17 & 6.18 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 0.02 \\ -0.73 \\ -6.63 \end{bmatrix}$$

10. Solve the equations $\mathbf{AX} = \mathbf{B}$ by Doolittle's decomposition method, where

$$\mathbf{A} = \begin{bmatrix} 4 & -3 & 6 \\ 8 & -3 & 10 \\ -4 & 12 & -10 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}$$

11. Solve the equations $\mathbf{Ax} = \mathbf{b}$ by Cholesky's decomposition method, where

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 1 \\ 3/2 \\ 3 \end{bmatrix}$$

12. Solve the equations

$$\begin{bmatrix} 4 & -2 & -3 \\ 12 & 4 & -10 \\ -16 & 28 & 18 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1.1 \\ 0 \\ -2.3 \end{bmatrix}$$

by Doolittle's decomposition method.

13. Determine **L** that results from Cholesky's decomposition of the diagonal matrix

$$\mathbf{A} = \begin{bmatrix} \alpha_1 & 0 & 0 & \cdots \\ 0 & \alpha_2 & 0 & \cdots \\ 0 & 0 & \alpha_3 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

14. ■ Modify the function `gaussElimin` so that it will work with m constant vectors. Test the program by solving $\mathbf{AX} = \mathbf{B}$, where

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

15. ■ A well-known example of an ill-conditioned matrix is the *Hilbert matrix*:

$$\mathbf{A} = \begin{bmatrix} 1 & 1/2 & 1/3 & \cdots \\ 1/2 & 1/3 & 1/4 & \cdots \\ 1/3 & 1/4 & 1/5 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

Write a program that specializes in solving the equations $\mathbf{Ax} = \mathbf{b}$ by Doolittle's decomposition method, where $\mathbf{A}$ is the Hilbert matrix of arbitrary size $n \times n$, and

$$b_i = \sum_{j=1}^{n} A_{ij}$$

The program should have no input apart from n. By running the program, determine the largest n for which the solution is within six significant figures of the exact solution

$$\mathbf{x} = \begin{bmatrix} 1 & 1 & 1 & \cdots \end{bmatrix}^T$$

16. Derive the forward and back substitution algorithms for the solution phase of Choleski's method. Compare them with the function `choleskiSol`.
17. ■ Determine the coefficients of the polynomial $y = a_0 + a_1 x + a_2 x^2 + a_3 x^3$ that passes through the points (0, 10), (1, 35), (3, 31), and (4, 2).
18. ■ Determine the fourth-degree polynomial $y(x)$ that passes through the points (0, −1), (1, 1), (3, 3), (5, 2), and (6, −2).
19. ■ Find the fourth-degree polynomial $y(x)$ that passes through the points (0, 1), (0.75, −0.25), and (1, 1) and has zero curvature at (0, 1) and (1, 1).
20. ■ Solve the equations $\mathbf{Ax} = \mathbf{b}$, where

$$\mathbf{A} = \begin{bmatrix} 3.50 & 2.77 & -0.76 & 1.80 \\ -1.80 & 2.68 & 3.44 & -0.09 \\ 0.27 & 5.07 & 6.90 & 1.61 \\ 1.71 & 5.45 & 2.68 & 1.71 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 7.31 \\ 4.23 \\ 13.85 \\ 11.55 \end{bmatrix}$$

By computing $|\mathbf{A}|$ and $\mathbf{Ax}$ comment on the accuracy of the solution.

21. Compute the condition number of the matrix

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

based on (a) the euclidean norm and (b) the infinity norm. You may use the function `inv(A)` in `numpy.linalg` to determine the inverse of $\mathbf{A}$.

22. ■ Write a function that returns the condition number of a matrix based on the euclidean norm. Test the function by computing the condition number of the ill-conditioned matrix:

$$\mathbf{A} = \begin{bmatrix} 1 & 4 & 9 & 16 \\ 4 & 9 & 16 & 25 \\ 9 & 16 & 25 & 36 \\ 16 & 25 & 36 & 49 \end{bmatrix}$$

Use the function `inv(A)` in `numpy.linalg` to determine the inverse of **A**.

23. ■ Test the function `gaussElimin` by solving $\mathbf{Ax} = \mathbf{b}$, where **A** is a $n \times n$ random matrix and $b_i = \sum_{j=1}^{n} A_{ij}$ (sum of the elements in the ith row of **A**). A random matrix can be generated with the `rand` function in the `numpy.random` module:

```
from numpy.random import ran
a = rand(n,n)
```

The solution should be $x = \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix}^T$. Run the program with $n = 200$ or bigger.

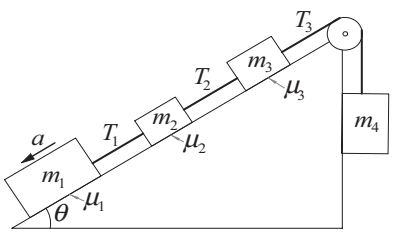
24. ■ The function `gaussElimin` also works with complex numbers. Use it to solve $\mathbf{Ax} = \mathbf{b}$, where

$$\mathbf{A} = \begin{bmatrix} 5+i & 5+2i & -5+3i & 6-3i \\ 5+2i & 7-2i & 8-i & -1+3i \\ -5+3i & 8-i & -3-3i & 2+2i \\ 6-3i & -1+3i & 2+2i & 8+14i \end{bmatrix}$$

$$\mathbf{b} = \begin{bmatrix} 15-35i & 2+10i & -2-34i & 8+14i \end{bmatrix}^T$$

Note that Python uses j to denote $\sqrt{-1}$.

25. ■



The four blocks of different masses m_i are connected by ropes of negligible mass. Three of the blocks lie on a inclined plane, the coefficients of friction between

the blocks and the plane being μ_i. The equations of motion for the blocks can be shown to be

$$\begin{aligned} T_1 + m_1 a &= m_1 g(\sin\theta - \mu_1 \cos\theta) \\ -T_1 + T_2 + m_2 a &= m_2 g(\sin\theta - \mu_2 \cos\theta) \\ -T_2 + T_3 + m_3 a &= m_3 g(\sin\theta - \mu_3 \cos\theta) \\ -T_3 + m_4 a &= -m_4 g \end{aligned}$$

where T_i denotes the tensile forces in the ropes and a is the acceleration of the system. Determine a and T_i if $\theta = 45^\circ$, $g = 9.82$ m/s^2 and

$$\mathbf{m} = \begin{bmatrix} 10 & 4 & 5 & 6 \end{bmatrix}^T \text{ kg}$$

$$\boldsymbol{\mu} = \begin{bmatrix} 0.25 & 0.3 & 0.2 \end{bmatrix}^T$$

2.4 Symmetric and Banded Coefficient Matrices

Introduction

Engineering problems often lead to coefficient matrices that are *sparsely populated*, meaning that most elements of the matrix are zero. If all the nonzero terms are clustered about the leading diagonal, then the matrix is said to be *banded*. An example of a banded matrix is

$$\mathbf{A} = \begin{bmatrix} \mathrm{X} & \mathrm{X} & 0 & 0 & 0 \\ \mathrm{X} & \mathrm{X} & \mathrm{X} & 0 & 0 \\ 0 & \mathrm{X} & \mathrm{X} & \mathrm{X} & 0 \\ 0 & 0 & \mathrm{X} & \mathrm{X} & \mathrm{X} \\ 0 & 0 & 0 & \mathrm{X} & \mathrm{X} \end{bmatrix}$$

where X's denote the nonzero elements that form the populated band (some of these elements may be zero). All the elements lying outside the band are zero. This banded matrix has a bandwidth of three, because there are at most three nonzero elements in each row (or column). Such a matrix is called *tridiagonal*.

If a banded matrix is decomposed in the form $\mathbf{A} = \mathbf{LU}$, both $\mathbf{L}$ and $\mathbf{U}$ retain the banded structure of $\mathbf{A}$. For example, if we decomposed the matrix shown above, we would get

$$\mathbf{L} = \begin{bmatrix} \mathrm{X} & 0 & 0 & 0 & 0 \\ \mathrm{X} & \mathrm{X} & 0 & 0 & 0 \\ 0 & \mathrm{X} & \mathrm{X} & 0 & 0 \\ 0 & 0 & \mathrm{X} & \mathrm{X} & 0 \\ 0 & 0 & 0 & \mathrm{X} & \mathrm{X} \end{bmatrix} \qquad \mathbf{U} = \begin{bmatrix} \mathrm{X} & \mathrm{X} & 0 & 0 & 0 \\ 0 & \mathrm{X} & \mathrm{X} & 0 & 0 \\ 0 & 0 & \mathrm{X} & \mathrm{X} & 0 \\ 0 & 0 & 0 & \mathrm{X} & \mathrm{X} \\ 0 & 0 & 0 & 0 & \mathrm{X} \end{bmatrix}$$

The banded structure of a coefficient matrix can be exploited to save storage and computation time. If the coefficient matrix is also symmetric, further economies are

possible. In this section I show how the methods of solution discussed previously can be adapted for banded and symmetric coefficient matrices.

Tridiagonal Coefficient Matrix

Consider the solution of $\mathbf{Ax} = \mathbf{b}$ by Doolittle's decomposition, where $\mathbf{A}$ is the $n \times n$ tridiagonal matrix

$$\mathbf{A} = \begin{bmatrix} d_1 & e_1 & 0 & 0 & \cdots & 0 \\ c_1 & d_2 & e_2 & 0 & \cdots & 0 \\ 0 & c_2 & d_3 & e_3 & \cdots & 0 \\ 0 & 0 & c_3 & d_4 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 0 & c_{n-1} & d_n \end{bmatrix}$$

As the notation implies, we are storing the non-zero elements of $\mathbf{A}$ in the vectors

$$\mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{bmatrix} \qquad \mathbf{d} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_{n-1} \\ d_n \end{bmatrix} \qquad \mathbf{e} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_{n-1} \end{bmatrix}$$

The resulting saving of storage can be significant. For example, a 100×100 tridiagonal matrix, containing 10,000 elements, can be stored in only $99 + 100 + 99 = 298$ locations, which represents a compression ratio of about 33:1.

Let us now apply LU decomposition to the coefficient matrix. We reduce row k by getting rid of c_{k-1} with the elementary operation

$$\text{row } k \leftarrow \text{row } k - (c_{k-1}/d_{k-1}) \times \text{row } (k-1), \quad k = 2, 3, \ldots, n$$

The corresponding change in d_k is

$$d_k \leftarrow d_k - (c_{k-1}/d_{k-1})e_{k-1} \tag{2.21}$$

whereas e_k is not affected. To finish up with Doolittle's decomposition of the form $[\mathbf{L}\backslash\mathbf{U}]$, we store the multiplier $\lambda = c_{k-1}/d_{k-1}$ in the location previously occupied by c_{k-1}:

$$c_{k-1} \leftarrow c_{k-1}/d_{k-1} \tag{2.22}$$

Thus the decomposition algorithm is

```
for k in range(1,n):
    lam = c[k-1]/d[k-1]
    d[k] = d[k] - lam*e[k-1]
    c[k-1] = lam
```

Next we look at the solution phase (i.e., solution of the $\mathbf{Ly} = \mathbf{b}$), followed by $\mathbf{Ux} = \mathbf{y}$. The equations $\mathbf{Ly} = \mathbf{b}$ can be portrayed by the augmented coefficient matrix

$$\left[\mathbf{L} \mid \mathbf{b}\right] = \left[\begin{array}{cccccc|c} 1 & 0 & 0 & 0 & \cdots & 0 & b_1 \\ c_1 & 1 & 0 & 0 & \cdots & 0 & b_2 \\ 0 & c_2 & 1 & 0 & \cdots & 0 & b_3 \\ 0 & 0 & c_3 & 1 & \ldots & 0 & b_4 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & c_{n-1} & 1 & b_n \end{array}\right]$$

Note that the original contents of **c** were destroyed and replaced by the multipliers during the decomposition. The solution algorithm for **y** by forward substitution is

```
y[0] = b[0]
for k in range(1,n):
    y[k] = b[k] - c[k-1]*y[k-1]
```

The augmented coefficient matrix representing $\mathbf{Ux} = \mathbf{y}$ is

$$\left[\mathbf{U} \mid \mathbf{y}\right] = \left[\begin{array}{cccccc|c} d_1 & e_1 & 0 & \cdots & 0 & 0 & y_1 \\ 0 & d_2 & e_2 & \cdots & 0 & 0 & y_2 \\ 0 & 0 & d_3 & \cdots & 0 & 0 & y_3 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & d_{n-1} & e_{n-1} & y_{n-1} \\ 0 & 0 & 0 & \cdots & 0 & d_n & y_n \end{array}\right]$$

Note again that the contents of **d** were altered from the original values during the decomposition phase (but **e** was unchanged). The solution for **x** is obtained by back substitution using the algorithm

```
x[n-1] = y[n-1]/d[n-1]
for k in range(n-2,-1,-1):
    x[k] = (y[k] - e[k]*x[k+1])/d[k]
end do
```

■ LUdecomp3

This module contains the functions `LUdecomp3` and `LUsolve3` for the decomposition and solution phases of a tridiagonal matrix. In `LUsolve3`, the vector **y** writes over the constant vector **b** during forward substitution. Similarly, the solution vector **x** overwrites **y** in the back substitution process. In other words, **b** contains the solution upon exit from `LUsolve3`.

```
## module LUdecomp3
''' c,d,e = LUdecomp3(c,d,e).
    LU decomposition of tridiagonal matrix [c\d\e]. On output
    {c},{d} and {e} are the diagonals of the decomposed matrix.
```

```
    x = LUsolve(c,d,e,b).
    Solves [c\d\e]{x} = {b}, where {c}, {d} and {e} are the
    vectors returned from LUdecomp3.
'''

def LUdecomp3(c,d,e):
    n = len(d)
    for k in range(1,n):
        lam = c[k-1]/d[k-1]
        d[k] = d[k] - lam*e[k-1]
        c[k-1] = lam
    return c,d,e

def LUsolve3(c,d,e,b):
    n = len(d)
    for k in range(1,n):
        b[k] = b[k] - c[k-1]*b[k-1]
    b[n-1] = b[n-1]/d[n-1]
    for k in range(n-2,-1,-1):
        b[k] = (b[k] - e[k]*b[k+1])/d[k]
    return b
```

Symmetric Coefficient Matrices

More often than not, coefficient matrices that arise in engineering problems are symmetric as well as banded. Therefore, it is worthwhile to discover special properties of such matrices and to learn how to use them in the construction of efficient algorithms.

If the matrix $\mathbf{A}$ is symmetric, then the LU decomposition can be presented in the form

$$\mathbf{A} = \mathbf{LU} = \mathbf{LDL}^T \tag{2.23}$$

where $\mathbf{D}$ is a diagonal matrix. An example is Choleski's decomposition $\mathbf{A} = \mathbf{LL}^T$ that was discussed earlier (in this case $\mathbf{D} = \mathbf{I}$).

For Doolittle's decomposition we have

$$\mathbf{U} = \mathbf{DL}^T = \begin{bmatrix} D_1 & 0 & 0 & \cdots & 0 \\ 0 & D_2 & 0 & \cdots & 0 \\ 0 & 0 & D_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & \cdots & D_n \end{bmatrix} \begin{bmatrix} 1 & L_{21} & L_{31} & \cdots & L_{n1} \\ 0 & 1 & L_{32} & \cdots & L_{n2} \\ 0 & 0 & 1 & \cdots & L_{n3} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

which gives

$$\mathbf{U} = \begin{bmatrix} D_1 & D_1 L_{21} & D_1 L_{31} & \cdots & D_1 L_{n1} \\ 0 & D_2 & D_2 L_{32} & \cdots & D_2 L_{n2} \\ 0 & 0 & D_3 & \cdots & D_3 L_{3n} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & \cdots & D_n \end{bmatrix} \tag{2.24}$$

We now see that during decomposition of a symmetric matrix only **U** has to be stored, because **D** and **L** can be easily recovered from **U**. Thus Gauss elimination, which results in an upper triangular matrix of the form shown in Eq. (2.24), is sufficient to decompose a symmetric matrix.

There is an alternative storage scheme that can be employed during **LU** decomposition. The idea is to arrive at the matrix

$$\mathbf{U}^* = \begin{bmatrix} D_1 & L_{21} & L_{31} & \cdots & L_{n1} \\ 0 & D_2 & L_{32} & \cdots & L_{n2} \\ 0 & 0 & D_3 & \cdots & L_{n3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & D_n \end{bmatrix} \tag{2.25}$$

Here **U** can be recovered from $U_{ij} = D_i L_{ji}$. It turns out that this scheme leads to a computationally more efficient solution phase; therefore, we adopt it for symmetric, banded matrices.

Symmetric, Pentadiagonal Coefficient Matrices

We encounter pentadiagonal (bandwidth = 5) coefficient matrices in the solution of fourth-order, ordinary differential equations by finite differences. Often these matrices are symmetric, in which case an $n \times n$ coefficient matrix has the form

$$\mathbf{A} = \begin{bmatrix} d_1 & e_1 & f_1 & 0 & 0 & 0 & \cdots & 0 \\ e_1 & d_2 & e_2 & f_2 & 0 & 0 & \cdots & 0 \\ f_1 & e_2 & d_3 & e_3 & f_3 & 0 & \cdots & 0 \\ 0 & f_2 & e_3 & d_4 & e_4 & f_4 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & f_{n-4} & e_{n-3} & d_{n-2} & e_{n-2} & f_{n-2} \\ 0 & \cdots & 0 & 0 & f_{n-3} & e_{n-2} & d_{n-1} & e_{n-1} \\ 0 & \cdots & 0 & 0 & 0 & f_{n-2} & e_{n-1} & d_n \end{bmatrix} \tag{2.26}$$

As in the case of tridiagonal matrices, we store the nonzero elements in the three vectors

$$\mathbf{d} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_{n-2} \\ d_{n-1} \\ d_n \end{bmatrix} \qquad \mathbf{e} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_{n-2} \\ e_{n-1} \end{bmatrix} \qquad \mathbf{f} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_{n-2} \end{bmatrix}$$

Let us now look at the solution of the equations $\mathbf{Ax} = \mathbf{b}$ by Doolittle's decomposition. The first step is to transform $\mathbf{A}$ to upper triangular form by Gauss elimination. If elimination has progressed to the stage where the kth row has become the pivot row, we have the following situation

$$\mathbf{A} = \left[\begin{array}{ccccc|ccccc} \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \\ \hline \cdots & 0 & d_k & e_k & f_k & 0 & 0 & 0 & \cdots & \leftarrow \\ \cdots & 0 & e_k & d_{k+1} & e_{k+1} & f_{k+1} & 0 & 0 & \cdots & \\ \cdots & 0 & f_k & e_{k+1} & d_{k+2} & e_{k+2} & f_{k+2} & 0 & \cdots & \\ \hline \cdots & 0 & 0 & f_{k+1} & e_{k+2} & d_{k+3} & e_{k+3} & f_{k+3} & \cdots & \\ & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \end{array}\right]$$

The elements e_k and f_k below the pivot row (the kth row) are eliminated by the operations

$$\begin{aligned} \text{row } (k+1) &\leftarrow \text{row } (k+1) - (e_k/d_k) \times \text{row } k \\ \text{row } (k+2) &\leftarrow \text{row } (k+2) - (f_k/d_k) \times \text{row } k \end{aligned}$$

The only terms (other than those being eliminated) that are changed by the operations are

$$\begin{aligned} d_{k+1} &\leftarrow d_{k+1} - (e_k/d_k)e_k \\ e_{k+1} &\leftarrow e_{k+1} - (e_k/d_k)f_k \\ d_{k+2} &\leftarrow d_{k+2} - (f_k/d_k)f_k \end{aligned} \tag{2.27a}$$

Storage of the multipliers in the *upper* triangular portion of the matrix results in

$$e_k \leftarrow e_k/d_k \qquad f_k \leftarrow f_k/d_k \tag{2.27b}$$

At the conclusion of the elimination phase the matrix has the form (do not confuse **d**, **e**, and **f** with the original contents of **A**)

$$\mathbf{U}^* = \begin{bmatrix} d_1 & e_1 & f_1 & 0 & \cdots & 0 \\ 0 & d_2 & e_2 & f_2 & \cdots & 0 \\ 0 & 0 & d_3 & e_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & 0 & d_{n-1} & e_{n-1} \\ 0 & 0 & \cdots & 0 & 0 & d_n \end{bmatrix}$$

Now comes the solution phase. The equations $\mathbf{Ly} = \mathbf{b}$ have the augmented coefficient matrix

$$\left[\mathbf{L} \mid \mathbf{b}\right] = \left[\begin{array}{cccccc|c} 1 & 0 & 0 & 0 & \cdots & 0 & b_1 \\ e_1 & 1 & 0 & 0 & \cdots & 0 & b_2 \\ f_1 & e_2 & 1 & 0 & \cdots & 0 & b_3 \\ 0 & f_2 & e_3 & 1 & \cdots & 0 & b_4 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & 0 & f_{n-2} & e_{n-1} & 1 & b_n \end{array}\right]$$

Solution by forward substitution yields

$$\begin{aligned} y_1 &= b_1 \\ y_2 &= b_2 - e_1 y_1 \\ &\vdots \\ y_k &= b_k - f_{k-2} y_{k-2} - e_{k-1} y_{k-1}, \quad k = 3, 4, \ldots, n \end{aligned} \tag{2.28}$$

The equations to be solved by back substitution, namely $\mathbf{Ux} = \mathbf{y}$, have the augmented coefficient matrix

$$\left[\mathbf{U} \mid \mathbf{y}\right] = \left[\begin{array}{cccccc|c} d_1 & d_1 e_1 & d_1 f_1 & 0 & \cdots & 0 & y_1 \\ 0 & d_2 & d_2 e_2 & d_2 f_2 & \cdots & 0 & y_2 \\ 0 & 0 & d_3 & d_3 e_3 & \cdots & 0 & y_3 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & d_{n-1} & d_{n-1} e_{n-1} & y_{n-1} \\ 0 & 0 & \cdots & 0 & 0 & d_n & y_n \end{array}\right]$$

the solution of which is obtained by back substitution:

$$\begin{aligned} x_n &= y_n / d_n \\ x_{n-1} &= y_{n-1}/d_{n-1} - e_{n-1} x_n \\ &\vdots \\ x_k &= y_k/d_k - e_k x_{k+1} - f_k x_{k+2}, \quad k = n-2, n-3, \ldots, 1 \end{aligned}$$

■ LUdecomp5

The function `LUdecomp5` decomposes a symmetric, pentadiagonal matrix **A** of the form **A** = [**f****e****d****e****f**]. The original vectors **d**, **e**, and **f** are destroyed and replaced by the vectors of the decomposed matrix. After decomposition, the solution of **Ax** = **b** can be obtained by `LUsolve5`. During forward substitution, the original **b** is replaced by **y**. Similarly, **y** is written over by **x** in the back substitution phase, so that **b** contains the solution vector upon exit from `LUsolve5`.

```
## module LUdecomp5
''' d,e,f = LUdecomp5(d,e,f).
    LU decomposition of symmetric pentadiagonal matrix [a], where
    {f}, {e} and {d} are the diagonals of [a]. On output
    {d},{e} and {f} are the diagonals of the decomposed matrix.

    x = LUsolve5(d,e,f,b).
    Solves [a]{x} = {b}, where {d}, {e} and {f} are the vectors
    returned from LUdecomp5.
    '''
def LUdecomp5(d,e,f):
    n = len(d)
    for k in range(n-2):
        lam = e[k]/d[k]
        d[k+1] = d[k+1] - lam*e[k]
        e[k+1] = e[k+1] - lam*f[k]
        e[k] = lam
        lam = f[k]/d[k]
        d[k+2] = d[k+2] - lam*f[k]
        f[k] = lam
    lam = e[n-2]/d[n-2]
    d[n-1] = d[n-1] - lam*e[n-2]
    e[n-2] = lam
    return d,e,f

def LUsolve5(d,e,f,b):
    n = len(d)
    b[1] = b[1] - e[0]*b[0]
    for k in range(2,n):
        b[k] = b[k] - e[k-1]*b[k-1] - f[k-2]*b[k-2]
    b[n-1] = b[n-1]/d[n-1]
    b[n-2] = b[n-2]/d[n-2] - e[n-2]*b[n-1]
    for k in range(n-3,-1,-1):
        b[k] = b[k]/d[k] - e[k]*b[k+1] - f[k]*b[k+2]
    return b
```

EXAMPLE 2.9

As a result of Gauss elimination, a symmetric matrix $\mathbf{A}$ was transformed to the upper triangular form

$$\mathbf{U} = \begin{bmatrix} 4 & -2 & 1 & 0 \\ 0 & 3 & -3/2 & 1 \\ 0 & 0 & 3 & -3/2 \\ 0 & 0 & 0 & 35/12 \end{bmatrix}$$

Determine the original matrix $\mathbf{A}$.

Solution. First we find $\mathbf{L}$ in the decomposition $\mathbf{A} = \mathbf{LU}$. Dividing each row of $\mathbf{U}$ by its diagonal element yields

$$\mathbf{L}^T = \begin{bmatrix} 1 & -1/2 & 1/4 & 0 \\ 0 & 1 & -1/2 & 1/3 \\ 0 & 0 & 1 & -1/2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Therefore, $\mathbf{A} = \mathbf{LU}$, or

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1/2 & 1 & 0 & 0 \\ 1/4 & -1/2 & 1 & 0 \\ 0 & 1/3 & -1/2 & 1 \end{bmatrix} \begin{bmatrix} 4 & -2 & 1 & 0 \\ 0 & 3 & -3/2 & 1 \\ 0 & 0 & 3 & -3/2 \\ 0 & 0 & 0 & 35/12 \end{bmatrix}$$

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

EXAMPLE 2.10

Determine $\mathbf{L}$ and $\mathbf{D}$ that result from Doolittle's decomposition $\mathbf{A} = \mathbf{LDL}^T$ of the symmetric matrix

$$\mathbf{A} = \begin{bmatrix} 3 & -3 & 3 \\ -3 & 5 & 1 \\ 3 & 1 & 10 \end{bmatrix}$$

Solution. We use Gauss elimination, storing the multipliers in the *upper* triangular portion of $\mathbf{A}$. At the completion of elimination, the matrix will have the form of $\mathbf{U}^*$ in Eq. (2.25).

The terms to be eliminated in the first pass are A_{21} and A_{31} using the elementary operations

$$\text{row } 2 \leftarrow \text{row } 2 - (-1) \times \text{row } 1$$

$$\text{row } 3 \leftarrow \text{row } 3 - (1) \times \text{row } 1$$

Storing the multipliers (-1 and 1) in the locations occupied by A_{12} and A_{13}, we get

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

The second pass is the operation

$$\text{row } 3 \leftarrow \text{row } 3 - 2 \times \text{row } 2$$

yields, after overwriting A_{23} with the multiplier 2

$$\mathbf{A}'' = \left[\mathbf{0}\backslash\mathbf{D}\backslash\mathbf{L}^T\right] = \begin{bmatrix} 3 & -1 & 1 \\ 0 & 2 & 2 \\ 0 & 0 & -1 \end{bmatrix}$$

Hence

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

EXAMPLE 2.11

Use the functions `LUdecmp3` and `LUsolve3` to solve $\mathbf{Ax} = \mathbf{b}$, where

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 5 \\ -5 \\ 4 \\ -5 \\ 5 \end{bmatrix}$$

Solution

```
#!/usr/bin/python
## example2_11
import numpy as np
from LUdecomp3 import *

d = np.ones((5))*2.0
c = np.ones((4))*(-1.0)
b = np.array([5.0, -5.0, 4.0, -5.0, 5.0])
e = c.copy()
c,d,e = LUdecomp3(c,d,e)
x = LUsolve3(c,d,e,b)
print("\nx =\n",x)
input("\nPress return to exit")
```

The output is

```
x =
[ 2. -1.  1. -1.  2.]
```

2.5 Pivoting

Introduction

Sometimes the order in which the equations are presented to the solution algorithm has a profound effect on the results. For example, consider these equations:

$$\begin{aligned} 2x_1 - x_2 &= 1 \\ -x_1 + 2x_2 - x_3 &= 0 \\ -x_2 + x_3 &= 0 \end{aligned}$$

The corresponding augmented coefficient matrix is

$$\left[\mathbf{A} \mid \mathbf{b}\right] = \left[\begin{array}{ccc|c} 2 & -1 & 0 & 1 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 1 & 0 \end{array}\right] \tag{a}$$

Equations (a) are in the "right order" in the sense that we would have no trouble obtaining the correct solution $x_1 = x_2 = x_3 = 1$ by Gauss elimination or LU decomposition. Now suppose that we exchange the first and third equations, so that the augmented coefficient matrix becomes

$$\left[\mathbf{A} \mid \mathbf{b}\right] = \left[\begin{array}{ccc|c} 0 & -1 & 1 & 0 \\ -1 & 2 & -1 & 0 \\ 2 & -1 & 0 & 1 \end{array}\right] \tag{b}$$

Because we did not change the equations (only their order was altered), the solution is still $x_1 = x_2 = x_3 = 1$. However, Gauss elimination fails immediately because of the presence of the zero pivot element (the element A_{11}).

This example demonstrates that it is sometimes essential to reorder the equations during the elimination phase. The reordering, or *row pivoting*, is also required if the pivot element is not zero, but is very small in comparison to other elements in the pivot row, as demonstrated by the following set of equations:

$$\left[\mathbf{A} \mid \mathbf{b}\right] = \left[\begin{array}{ccc|c} \varepsilon & -1 & 1 & 0 \\ -1 & 2 & -1 & 0 \\ 2 & -1 & 0 & 1 \end{array}\right] \tag{c}$$

These equations are the same as Eqs. (b), except that the small number ε replaces the zero element in Eq. (b). Therefore, if we let $\varepsilon \to 0$, the solutions of Eqs. (b) and (c) should become identical. After the first phase of Gauss elimination, the augmented coefficient matrix becomes

$$\left[\mathbf{A}' \mid \mathbf{b}'\right] = \left[\begin{array}{ccc|c} \varepsilon & -1 & 1 & 0 \\ 0 & 2 - 1/\varepsilon & -1 + 1/\varepsilon & 0 \\ 0 & -1 + 2/\varepsilon & -2/\varepsilon & 1 \end{array}\right] \tag{d}$$

Because the computer works with a fixed word length, all numbers are rounded off to a finite number of significant figures. If ε is very small, then $1/\varepsilon$ is huge, and an

element such as $2 - 1/\varepsilon$ is rounded to $-1/\varepsilon$. Therefore, for sufficiently small ε, Eqs. (d) are actually stored as

$$\left[\mathbf{A}' \mid \mathbf{b}'\right] = \left[\begin{array}{ccc|c} \varepsilon & -1 & 1 & 0 \\ 0 & -1/\varepsilon & 1/\varepsilon & 0 \\ 0 & 2/\varepsilon & -2/\varepsilon & 1 \end{array}\right]$$

Because the second and third equations obviously contradict each other, the solution process fails again. This problem would not arise if the first and second, or the first and the third equations were interchanged in Eqs. (c) before the elimination.

The last example illustrates the extreme case where ε was so small that roundoff errors resulted in total failure of the solution. If we were to make ε somewhat bigger so that the solution would not "bomb" any more, the roundoff errors might still be large enough to render the solution unreliable. Again, this difficulty could be avoided by pivoting.

Diagonal Dominance

An $n \times n$ matrix $\mathbf{A}$ is said to be *diagonally dominant* if each diagonal element is larger than the sum of the other elements in the same row (we are talking here about absolute values). Thus diagonal dominance requires that

$$|A_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |A_{ij}| \quad (i = 1, 2, \ldots, n) \tag{2.30}$$

For example, the matrix

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

is not diagonally dominant. However, if we rearrange the rows in the following manner

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

then we have diagonal dominance.

It can be shown that if the coefficient matrix of the equations $\mathbf{Ax} = \mathbf{b}$ is diagonally dominant, then the solution does not benefit from pivoting; that is, the equations are already arranged in the optimal order. It follows that the strategy of pivoting should be to reorder the equations so that the coefficient matrix is as close to diagonal dominance as possible. This is the principle behind scaled row pivoting, discussed next.

Gauss Elimination with Scaled Row Pivoting

Consider the solution of $\mathbf{Ax} = \mathbf{b}$ by Gauss elimination with row pivoting. Recall that pivoting aims at improving diagonal dominance of the coefficient matrix (i.e., making the pivot element as large as possible in comparison to other elements in the pivot row). The comparison is made easier if we establish an array **s** with the elements

$$s_i = \max_j \left|A_{ij}\right|, \quad i = 1, 2, \ldots, n \tag{2.31}$$

Thus s_i, called the *scale factor* of row i, contains the absolute value of the largest element in the ith row of **A**. The vector **s** can be obtained with the algorithm

```
for i in range(n):
    s[i] = max(abs(a[i,:]))
```

The *relative size* of an element A_{ij} (that is, relative to the largest element in the ith row) is defined as the ratio

$$r_{ij} = \frac{\left|A_{ij}\right|}{s_i} \tag{2.32}$$

Suppose that the elimination phase has reached the stage where the kth row has become the pivot row. The augmented coefficient matrix at this point is shown in the following matrix:

$$\left[\begin{array}{cccccc|c} A_{11} & A_{12} & A_{13} & A_{14} & \cdots & A_{1n} & b_1 \\ 0 & A_{22} & A_{23} & A_{24} & \cdots & A_{2n} & b_2 \\ 0 & 0 & A_{33} & A_{34} & \cdots & A_{3n} & b_3 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ \hline 0 & \cdots & 0 & A_{kk} & \cdots & A_{kn} & b_k \\ \vdots & \cdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & \cdots & 0 & A_{nk} & \cdots & A_{nn} & b_n \end{array}\right] \leftarrow$$

We do not automatically accept A_{kk} as the next pivot element, but look in the kth column below A_{kk} for a "better" pivot. The best choice is the element A_{pk} that has the largest relative size; that is, we choose p such that

$$r_{pk} = \max_j \left(r_{jk}\right), \quad j \geq k$$

If we find such an element, then we interchange the rows k and p, and proceed with the elimination pass as usual. Note that the corresponding row interchange must also be carried out in the scale factor array **s**. The algorithm that does all this is as follows:

```
for k in range(0,n-1):

  # Find row containing element with largest relative size
    p = argmax(abs(a[k:n,k])/s[k:n]) + k

  # If this element is very small, matrix is singular
```

```
    if abs(a[p,k]) < tol: error.err('Matrix is singular')

  # Check whether rows k and p must be interchanged
    if p != k:
      # Interchange rows if needed
        swap.swapRows(b,k,p)
        swap.swapRows(s,k,p)
        swap.swapRows(a,k,p)
  # Proceed with elimination
```

The Python statement `argmax(v)` returns the index of the largest element in the vector `v`. The algorithms for exchanging rows (and columns) are included in the module `swap` shown next.

■ swap

The function `swapRows` interchanges rows i and j of a matrix or vector **v**, whereas `swapCols` interchanges columns i and j of a matrix.

```
## module swap
''' swapRows(v,i,j).
    Swaps rows i and j of a vector or matrix [v].

    swapCols(v,i,j).
    Swaps columns of matrix [v].
'''
def swapRows(v,i,j):
    if len(v.shape) == 1:
        v[i],v[j] = v[j],v[i]
    else:
        v[[i,j],:] = v[[j,i],:]

def swapCols(v,i,j):
    v[:,[i,j]] = v[:,[j,i]]
```

■ gaussPivot

The function `gaussPivot` performs Gauss elimination with row pivoting. Apart from row swapping, the elimination and solution phases are identical to `gaussElimin` in Section 2.2.

```
## module gaussPivot
''' x = gaussPivot(a,b,tol=1.0e-12).
    Solves [a]{x} = {b} by Gauss elimination with
    scaled row pivoting
```

```
'''
import numpy as np
import swap
import error

def gaussPivot(a,b,tol=1.0e-12):
    n = len(b)

  # Set up scale factors
    s = np.zeros(n)
    for i in range(n):
        s[i] = max(np.abs(a[i,:]))

    for k in range(0,n-1):

      # Row interchange, if needed
        p = np.argmax(np.abs(a[k:n,k])/s[k:n]) + k
        if abs(a[p,k]) < tol: error.err('Matrix is singular')
        if p != k:
            swap.swapRows(b,k,p)
            swap.swapRows(s,k,p)
            swap.swapRows(a,k,p)

      # Elimination
        for i in range(k+1,n):
            if a[i,k] != 0.0:
                lam = a[i,k]/a[k,k]
                a[i,k+1:n] = a[i,k+1:n] - lam*a[k,k+1:n]
                b[i] = b[i] - lam*b[k]
    if abs(a[n-1,n-1]) < tol: error.err('Matrix is singular')

  # Back substitution
    b[n-1] = b[n-1]/a[n-1,n-1]
    for k in range(n-2,-1,-1):
        b[k] = (b[k] - np.dot(a[k,k+1:n],b[k+1:n]))/a[k,k]
    return b
```

■ LUpivot

The Gauss elimination algorithm can be changed to Doolittle's decomposition with minor changes. The most important of these changes is keeping a record of the row interchanges during the decomposition phase. In `LUdecomp` this record is kept in the array `seq`. Initially `seq` contains $[0, 1, 2, \ldots]$. Whenever two rows are interchanged, the corresponding interchange is also carried out in `seq`. Thus `seq`

shows the order in which the original rows have been rearranged. This information is passed on to the solution phase (`LUsolve`), which rearranges the elements of the constant vector in the same order before proceeding to forward and back substitutions.

```
## module LUpivot
''' a,seq = LUdecomp(a,tol=1.0e-9).
   LU decomposition of matrix [a] using scaled row pivoting.
   The returned matrix [a] = contains [U] in the upper
   triangle and the nondiagonal terms of [L] in the lower triangle.
   Note that [L][U] is a row-wise permutation of the original [a];
   the permutations are recorded in the vector {seq}.

   x = LUsolve(a,b,seq).
   Solves [L][U]{x} = {b}, where the matrix [a] = and the
   permutation vector {seq} are returned from LUdecomp.
'''
import numpy as np
import swap
import error

def LUdecomp(a,tol=1.0e-9):
   n = len(a)
   seq = np.array(range(n))

 # Set up scale factors
   s = np.zeros((n))
   for i in range(n):
       s[i] = max(abs(a[i,:]))

   for k in range(0,n-1):

     # Row interchange, if needed
       p = np.argmax(np.abs(a[k:n,k])/s[k:n]) + k
       if abs(a[p,k]) <  tol: error.err('Matrix is singular')
       if p != k:
           swap.swapRows(s,k,p)
           swap.swapRows(a,k,p)
           swap.swapRows(seq,k,p)

     # Elimination
       for i in range(k+1,n):
           if a[i,k] != 0.0:
               lam = a[i,k]/a[k,k]
```

```
                a[i,k+1:n] = a[i,k+1:n] - lam*a[k,k+1:n]
                a[i,k] = lam
    return a,seq

def LUsolve(a,b,seq):
    n = len(a)

  # Rearrange constant vector; store it in [x]
    x = b.copy()
    for i in range(n):
        x[i] = b[seq[i]]

  # Solution
    for k in range(1,n):
        x[k] = x[k] - np.dot(a[k,0:k],x[0:k])
    x[n-1] = x[n-1]/a[n-1,n-1]
    for k in range(n-2,-1,-1):
       x[k] = (x[k] - np.dot(a[k,k+1:n],x[k+1:n]))/a[k,k]
    return x
```

When to Pivot

Pivoting has two drawbacks. One is the increased cost of computation; the other is the destruction of symmetry and banded structure of the coefficient matrix. The latter is of particular concern in engineering computing, where the coefficient matrices are frequently banded and symmetric, a property that is used in the solution, as seen in the previous section. Fortunately, these matrices are often diagonally dominant as well, so that they would not benefit from pivoting anyway.

There are no infallible rules for determining when pivoting should be used. Experience indicates that pivoting is likely to be counterproductive if the coefficient matrix is banded. Positive definite and, to a lesser degree, symmetric matrices also seldom gain from pivoting. And we should not forget that pivoting is not the only means of controlling roundoff errors—there is also double precision arithmetic.

It should be strongly emphasized that these rules of thumb are only meant for equations that stem from real engineering problems. It is not difficult to concoct "textbook" examples that do not conform to these rules.

EXAMPLE 2.12

Employ Gauss elimination with scaled row pivoting to solve the equations $\mathbf{Ax} = \mathbf{b}$, where

$$\mathbf{A} = \begin{bmatrix} 2 & -2 & 6 \\ -2 & 4 & 3 \\ -1 & 8 & 4 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 16 \\ 0 \\ -1 \end{bmatrix}$$

Solution. The augmented coefficient matrix and the scale factor array are

$$\left[\mathbf{A} \mid \mathbf{b}\right] = \left[\begin{array}{ccc|c} 2 & -2 & 6 & 16 \\ -2 & 4 & 3 & 0 \\ -1 & 8 & 4 & -1 \end{array}\right] \qquad \mathbf{s} = \begin{bmatrix} 6 \\ 4 \\ 8 \end{bmatrix}$$

Note that $\mathbf{s}$ contains the absolute value of the biggest element in each row of $\mathbf{A}$. At this stage, all the elements in the first column of $\mathbf{A}$ are potential pivots. To determine the best pivot element, we calculate the relative sizes of the elements in the first column:

$$\begin{bmatrix} r_{11} \\ r_{21} \\ r_{31} \end{bmatrix} = \begin{bmatrix} |A_{11}|/s_1 \\ |A_{21}|/s_2 \\ |A_{31}|/s_3 \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/2 \\ 1/8 \end{bmatrix}$$

Since r_{21} is the biggest element, we conclude that A_{21} makes the best pivot element. Therefore, we exchange rows 1 and 2 of the augmented coefficient matrix and the scale factor array, obtaining

$$\left[\mathbf{A} \mid \mathbf{b}\right] = \left[\begin{array}{ccc|c} -2 & 4 & 3 & 0 \\ 2 & -2 & 6 & 16 \\ -1 & 8 & 4 & -1 \end{array}\right] \leftarrow \qquad \mathbf{s} = \begin{bmatrix} 4 \\ 6 \\ 8 \end{bmatrix}$$

Now the first pass of Gauss elimination is carried out (the arrow points to the pivot row), yielding

$$\left[\mathbf{A}' \mid \mathbf{b}'\right] = \left[\begin{array}{ccc|c} -2 & 4 & 3 & 0 \\ 0 & 2 & 9 & 16 \\ 0 & 6 & 5/2 & -1 \end{array}\right] \qquad \mathbf{s} = \begin{bmatrix} 4 \\ 6 \\ 8 \end{bmatrix}$$

The potential pivot elements for the next elimination pass are A'_{22} and A'_{32}. We determine the "winner" from

$$\begin{bmatrix} * \\ r_{22} \\ r_{32} \end{bmatrix} = \begin{bmatrix} * \\ |A_{22}|/s_2 \\ |A_{32}|/s_3 \end{bmatrix} = \begin{bmatrix} * \\ 1/3 \\ 3/4 \end{bmatrix}$$

Note that r_{12} is irrelevant, because row 1 already acted as the pivot row. Therefore, it is excluded from further consideration. Because r_{32} is bigger than r_{22}, the third row is the better pivot row. After interchanging rows 2 and 3, we have

$$\left[\mathbf{A}' \mid \mathbf{b}'\right] = \left[\begin{array}{ccc|c} -2 & 4 & 3 & 0 \\ 0 & 6 & 5/2 & -1 \\ 0 & 2 & 9 & 16 \end{array}\right] \leftarrow \qquad \mathbf{s} = \begin{bmatrix} 4 \\ 8 \\ 6 \end{bmatrix}$$

The second elimination pass now yields

$$\left[\mathbf{A}'' \mid \mathbf{b}''\right] = \left[\mathbf{U} \mid \mathbf{c}\right] = \left[\begin{array}{ccc|c} -2 & 4 & 3 & 0 \\ 0 & 6 & 5/2 & -1 \\ 0 & 0 & 49/6 & 49/3 \end{array}\right]$$

This completes the elimination phase. It should be noted that **U** is the matrix that would result from LU decomposition of the following row-wise permutation of **A** (the ordering of rows is the same as achieved by pivoting):

$$\begin{bmatrix} -2 & 4 & 3 \\ -1 & 8 & 4 \\ 2 & -2 & 6 \end{bmatrix}$$

Because the solution of $\mathbf{Ux} = \mathbf{c}$ by back substitution is not affected by pivoting, we skip the details computation. The result is $\mathbf{x}^T = \begin{bmatrix} 1 & -1 & 2 \end{bmatrix}$.

Alternate Solution. It is not necessary to physically exchange equations during pivoting. We could accomplish Gauss elimination just as well by keeping the equations in place. The elimination would then proceed as follows (for the sake of brevity, the details of choosing the pivot equation are not repeated):

$$\left[\mathbf{A} \mid \mathbf{b}\right] = \left[\begin{array}{ccc|c} 2 & -2 & 6 & 16 \\ -2 & 4 & 3 & 0 \\ -1 & 8 & 4 & -1 \end{array}\right] \leftarrow$$

$$\left[\mathbf{A}' \mid \mathbf{b}'\right] = \left[\begin{array}{ccc|c} 0 & 2 & 9 & 16 \\ -2 & 4 & 3 & 0 \\ 0 & 6 & 5/2 & -1 \end{array}\right] \leftarrow$$

$$\left[\mathbf{A}'' \mid \mathbf{b}''\right] = \left[\begin{array}{ccc|c} 0 & 0 & 49/6 & 49/3 \\ -2 & 4 & 3 & 0 \\ 0 & 6 & 5/2 & -1 \end{array}\right]$$

Yet now the back substitution phase is a little more involved, because the order in which the equations must be solved has become scrambled. In hand computations this is not a problem, because we can determine the order by inspection. Unfortunately, "by inspection" does not work on a computer. To overcome this difficulty, we have to maintain an integer array **p** that keeps track of the row permutations during the elimination phase. The contents of **p** indicate the order in which the pivot rows were chosen. In this example, we would have at the end of Gauss elimination

$$\mathbf{p} = \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix}$$

showing that row 2 was the pivot row in the first elimination pass, followed by row 3 in the second pass. The equations are solved by back substitution in the reverse order: Equation 1 is solved first for x_3, then equation 3 is solved for x_2, and finally equation 2 yields x_1.

By dispensing with swapping of equations, the scheme outlined here is claimed to result in a faster algorithm than `gaussPivot`. This may be true if the programs are written in Fortran or C, but our tests show that in Python `gaussPivot` is about 30% faster than the in-place elimination scheme.

PROBLEM SET 2.2

1. Solve the equations $\mathbf{Ax} = \mathbf{b}$ by utilizing Doolittle's decomposition, where

$$\mathbf{A} = \begin{bmatrix} 3 & -3 & 3 \\ -3 & 5 & 1 \\ 3 & 1 & 5 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 9 \\ -7 \\ 12 \end{bmatrix}$$

2. Use Doolittle's decomposition to solve $\mathbf{Ax} = \mathbf{b}$, where

$$\mathbf{A} = \begin{bmatrix} 4 & 8 & 20 \\ 8 & 13 & 16 \\ 20 & 16 & -91 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 24 \\ 18 \\ -119 \end{bmatrix}$$

3. Determine $\mathbf{L}$ and $\mathbf{D}$ that result from Doolittle's decomposition of the symmetric matrix

$$\mathbf{A} = \begin{bmatrix} 2 & -2 & 0 & 0 & 0 \\ -2 & 5 & -6 & 0 & 0 \\ 0 & -6 & 16 & 12 & 0 \\ 0 & 0 & 12 & 39 & -6 \\ 0 & 0 & 0 & -6 & 14 \end{bmatrix}$$

4. Solve the tridiagonal equations $\mathbf{Ax} = \mathbf{b}$ by Doolittle's decomposition method, where

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

5. Use Gauss elimination with scaled row pivoting to solve

$$\begin{bmatrix} 4 & -2 & 1 \\ -2 & 1 & -1 \\ -2 & 3 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \\ 0 \end{bmatrix}$$

6. Solve $\mathbf{Ax} = \mathbf{b}$ by Gauss elimination with scaled row pivoting, where

$$\mathbf{A} = \begin{bmatrix} 2.34 & -4.10 & 1.78 \\ 1.98 & 3.47 & -2.22 \\ 2.36 & -15.17 & 6.81 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 0.02 \\ -0.73 \\ -6.63 \end{bmatrix}$$

7. Solve the equations

$$\begin{bmatrix} 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & -1 & 2 & -1 \\ -1 & 2 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

by Gauss elimination with scaled row pivoting.

8. ■ Solve the equations

$$\begin{bmatrix} 0 & 2 & 5 & -1 \\ 2 & 1 & 3 & 0 \\ -2 & -1 & 3 & 1 \\ 3 & 3 & -1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -3 \\ 3 \\ -2 \\ 5 \end{bmatrix}$$

9. ■ Solve the symmetric, tridiagonal equations

$$\begin{aligned} 4x_1 - x_2 &= 9 \\ -x_{i-1} + 4x_i - x_{i+1} &= 5, \quad i = 2, \ldots, n-1 \\ -x_{n-1} + 4x_n &= 5 \end{aligned}$$

with $n = 10$.

10. ■ Solve the equations $\mathbf{Ax} = \mathbf{b}$, where

$$\mathbf{A} = \begin{bmatrix} 1.3174 & 2.7250 & 2.7250 & 1.7181 \\ 0.4002 & 0.8278 & 1.2272 & 2.5322 \\ 0.8218 & 1.5608 & 0.3629 & 2.9210 \\ 1.9664 & 2.0011 & 0.6532 & 1.9945 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 8.4855 \\ 4.9874 \\ 5.6665 \\ 6.6152 \end{bmatrix}$$

11. ■ Solve the following equations:

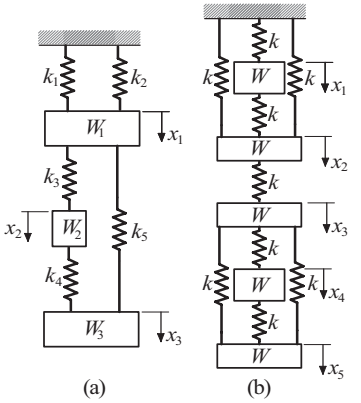
$$\begin{bmatrix} 10 & -2 & -1 & 2 & 3 & 1 & -4 & 7 \\ 5 & 11 & 3 & 10 & -3 & 3 & 3 & -4 \\ 7 & 12 & 1 & 5 & 3 & -12 & 2 & 3 \\ 8 & 7 & -2 & 1 & 3 & 2 & 2 & 4 \\ 2 & -15 & -1 & 1 & 4 & -1 & 8 & 3 \\ 4 & 2 & 9 & 1 & 12 & -1 & 4 & 1 \\ -1 & 4 & -7 & -1 & 1 & 1 & -1 & -3 \\ -1 & 3 & 4 & 1 & 3 & -4 & 7 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix} = \begin{bmatrix} 0 \\ 12 \\ -5 \\ 3 \\ -25 \\ -26 \\ 9 \\ -7 \end{bmatrix}$$

12. ■ The displacement formulation for the mass-spring system shown in Fig. (a) results in the following equilibrium equations of the masses

$$\begin{bmatrix} k_1 + k_2 + k_3 + k_5 & -k_3 & -k_5 \\ -k_3 & k_3 + k_4 & -k_4 \\ -k_5 & -k_4 & k_4 + k_5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} W_1 \\ W_2 \\ W_3 \end{bmatrix}$$

where k_i are the spring stiffnesses, W_i represent the weights of the masses, and x_i are the displacements of the masses from the undeformed configuration of the system. Write a program that solves these equations for given $\mathbf{k}$ and $\mathbf{W}$. Use the program to find the displacements if

$$\begin{aligned} k_1 &= k_3 = k_4 = k & \qquad k_2 &= k_5 = 2k \\ W_1 &= W_3 = 2W & \qquad W_2 &= W \end{aligned}$$

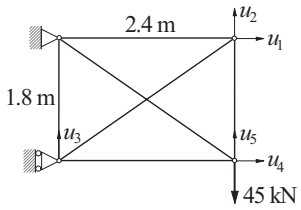


13. ■ The equlibrium equations of the mass-spring system in Fig. (b) are

$$\begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & 0 \\ 0 & -1 & 4 & -1 & -2 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & -2 & -1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} W/k \\ W/k \\ W/k \\ W/k \\ W/k \end{bmatrix}$$

where k are the spring stiffnesses, W represent the weights of the masses, and x_i are the displacements of the masses from the undeformed configuration of the system. Determine the displacements.

14. ■



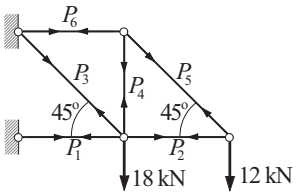
The displacement formulation for a plane truss is similar to that of a mass-spring system. The differences are (1) the stiffnesses of the members are $k_i = (EA/L)_i$, where E is the modulus of elasticity, A represents the cross-sectional area, and L is the length of the member; and (2) there are two components of displacement at each joint. For the statically indeterminate truss shown, the displacement formulation yields the symmetric equations $\mathbf{Ku} = \mathbf{p}$, where

$$\mathbf{K} = \begin{bmatrix} 27.58 & 7.004 & -7.004 & 0 & 0 \\ 7.004 & 29.57 & -5.253 & 0 & -24.32 \\ -7.004 & -5.253 & 29.57 & 0 & 0 \\ 0 & 0 & 0 & 27.58 & -7.004 \\ 0 & -24.32 & 0 & -7.004 & 29.57 \end{bmatrix} \text{MN/m}$$

$$\mathbf{p} = \begin{bmatrix} 0 & 0 & 0 & 0 & -45 \end{bmatrix}^T \text{kN}$$

Determine the displacements u_i of the joints.

15. ■

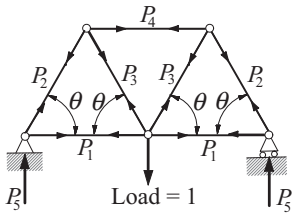


In the *force formulation* of a truss, the unknowns are the member forces P_i. For the statically determinate truss shown, force formulation is obtained by writing down the equilibrium equations of the joints

$$\begin{bmatrix} -1 & 1 & -1/\sqrt{2} & 0 & 0 & 0 \\ 0 & 0 & 1/\sqrt{2} & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1/\sqrt{2} & 0 \\ 0 & 0 & 0 & 0 & 1/\sqrt{2} & 0 \\ 0 & 0 & 0 & 0 & 1/\sqrt{2} & 1 \\ 0 & 0 & 0 & -1 & -1/\sqrt{2} & 0 \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{bmatrix} = \begin{bmatrix} 0 \\ 18 \\ 0 \\ 12 \\ 0 \\ 0 \end{bmatrix}$$

where the units of P_i are kN. (a) Solve the equations as they are with a computer program. (b) Rearrange the rows and columns so as to obtain a lower triangular coefficient matrix, and then solve the equations by back substitution using a calculator.

16. ■

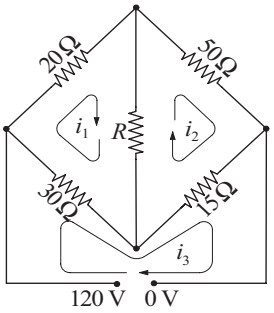


The force formulation of the symmetric truss shown results in the joint equilibrium equations

$$\begin{bmatrix} c & 1 & 0 & 0 & 0 \\ 0 & s & 0 & 0 & 1 \\ 0 & 0 & 2s & 0 & 0 \\ 0 & -c & c & 1 & 0 \\ 0 & s & s & 0 & 0 \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

where $s = \sin\theta$, $c = \cos\theta$, and P_i are the unknown forces. Write a program that computes the forces, given the angle θ. Run the program with $\theta = 53^\circ$.

17. ■

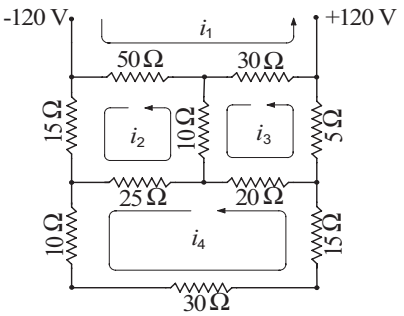


The electrical network shown can be viewed as consisting of three loops. Applying Kirchoff's law (∑voltage drops = ∑voltage sources) to each loop yields the following equations for the loop currents i_1, i_2 and i_3:

$$\begin{aligned}(50 + R)i_1 - Ri_2 - 30i_3 &= 0\\ -Ri_1 + (65 + R)i_2 - 15i_3 &= 0\\ -30i_2 - 15i_2 + 45i_3 &= 120\end{aligned}$$

Compute the three loop currents for $R = 5\ \Omega$, $10\ \Omega$, and $20\ \Omega$.

18. ■



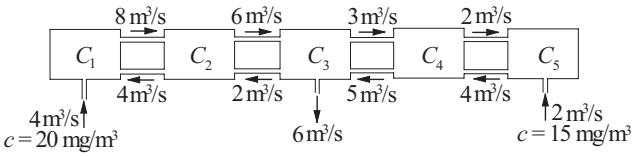
Determine the loop currents i_1 to i_4 in the electrical network shown.

19. ■ Consider the n simultaneous equations $\mathbf{Ax} = \mathbf{b}$, where

$$A_{ij} = (i + j)^2 \qquad b_i = \sum_{j=0}^{n-1} A_{ij}, \quad i = 0, 1, \ldots, n-1, \quad j = 0, 1, \ldots, n-1$$

Clearly, the solution is $\mathbf{x} = \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix}^T$. Write a program that solves these equations for any given n (pivoting is recommended). Run the program with $n = 2$, 3 and 4, and comment on the results.

20. ■



The diagram shows five mixing vessels connected by pipes. Water is pumped through the pipes at the steady rates shown on the diagram. The incoming water contains a chemical, the amount of which is specified by its concentration c (mg/m^3). Applying the principle of conservation of mass

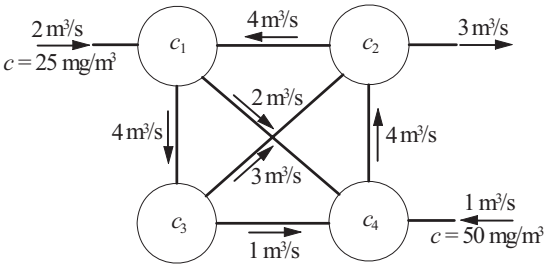
$$\text{mass of chemical flowing in} = \text{mass of chemical flowing out}$$

to each vessel, we obtain the following simultaneous equations for the concentrations c_i within the vessels:

$$\begin{aligned} -8c_1 + 4c_2 &= -80 \\ 8c_1 - 10c_2 + 2c_3 &= 0 \\ 6c_2 - 11c_3 + 5c_4 &= 0 \\ 3c_3 - 7c_4 + 4c_5 &= 0 \\ 2c_4 - 4c_5 &= -30 \end{aligned}$$

Note that the mass flow rate of the chemical is obtained by multiplying the volume flow rate of the water by the concentration. Verify the equations and determine the concentrations.

21. ■



Four mixing tanks are connected by pipes. The fluid in the system is pumped through the pipes at the rates shown in the figure. The fluid entering the system contains a chemical of concentration c as indicated. Determine the concentration of the chemical in the four tanks, assuming a steady state.

22. ■ Solve the following equations:

$$\begin{aligned} 7x_1 - 4x_2 + x_3 &= 1 \\ -4x_1 + 6x_2 - 4x_3 + x_4 &= 1 \\ x_{i-2} - 4x_{i-1} + 6x_i - 4x_{i+1} + x_{i+2} &= 1 \quad (i = 3, 4, \ldots, 8) \\ x_7 - 4x_8 + 6x_9 - 4x_{10} &= 1 \\ x_8 - 4x_9 + 7x_{10} &= 1 \end{aligned}$$

23. ■ Write a program that solves the complex tridiagonal equations $\mathbf{Ax} = \mathbf{b}$, where

$$\mathbf{A} = \begin{bmatrix} 2 & -i & 0 & 0 & \cdots & 0 \\ -i & 2 & -i & 0 & \cdots & 0 \\ 0 & -i & 2 & -i & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & -i & 2 & -i \\ 0 & 0 & 0 & \cdots & -i & 1 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 100 + 100i \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

The program should accommodate n equations, where n is arbitrary. Test it with $n = 10$.

*2.6 Matrix Inversion

Computing the inverse of a matrix and solving simultaneous equations are related tasks. The most economical way to invert an $n \times n$ matrix $\mathbf{A}$ is to solve the equations

$$\mathbf{AX} = \mathbf{I} \tag{2.33}$$

where $\mathbf{I}$ is the $n \times n$ identity matrix. The solution $\mathbf{X}$, also of size $n \times n$, will be the inverse of $\mathbf{A}$. The proof is simple: After we pre-multiply both sides of Eq. (2.33) by $\mathbf{A}^{-1}$ we have $\mathbf{A}^{-1}\mathbf{AX} = \mathbf{A}^{-1}\mathbf{I}$, which reduces to $\mathbf{X} = \mathbf{A}^{-1}$.

Inversion of large matrices should be avoided whenever possible because of its high cost. As seen from Eq. (2.33), inversion of $\mathbf{A}$ is equivalent to solving $\mathbf{Ax}_i = \mathbf{b}_i$ with $i = 1, 2, \ldots, n$, where $\mathbf{b}_i$ is the ith column of $\mathbf{I}$. Assuming that LU decomposition is employed in the solution, the solution phase (forward and back substitution) must be repeated n times, once for each $\mathbf{b}_i$. Since the cost of computation is proportional to n^3 for the decomposition phase and n^2 for each vector of the solution phase, the cost of inversion is considerably more expensive than the solution of $\mathbf{Ax} = \mathbf{b}$ (single constant vector $\mathbf{b}$).

Matrix inversion has another serious drawback—a banded matrix loses its structure during inversion. In other words, if $\mathbf{A}$ is banded or otherwise sparse, then $\mathbf{A}^{-1}$ is fully populated.

EXAMPLE 2.13

Write a function that inverts a matrix using LU decomposition with pivoting. Test the function by inverting

$$\mathbf{A} = \begin{bmatrix} 0.6 & -0.4 & 1.0 \\ -0.3 & 0.2 & 0.5 \\ 0.6 & -1.0 & 0.5 \end{bmatrix}$$

Solution. The following function `matInv` uses the decomposition and solution procedures in the module `LUpivot`.

```
#!/usr/bin/python
## example2_13
import numpy as np
from LUpivot import *

def matInv(a):
    n = len(a[0])
    aInv = np.identity(n)
    a,seq = LUdecomp(a)
    for i in range(n):
        aInv[:,i] = LUsolve(a,aInv[:,i],seq)
    return aInv

a = np.array([[ 0.6, -0.4,  1.0],\
              [-0.3,  0.2,  0.5],\
              [ 0.6, -1.0,  0.5]])
aOrig = a.copy()  # Save original [a]
aInv = matInv(a)  # Invert [a] (original [a] is destroyed)
print("\naInv =\n",aInv)
print("\nCheck: a*aInv =\n", np.dot(aOrig,aInv))
input("\nPress return to exit")
```

The output is

```
aInv =
[[ 1.66666667 -2.22222222 -1.11111111]
 [ 1.25       -0.83333333 -1.66666667]
 [ 0.5         1.          0.        ]]

Check: a*aInv =
[[  1.00000000e+00  -4.44089210e-16  -1.11022302e-16]
 [  0.00000000e+00   1.00000000e+00   5.55111512e-17]
 [  0.00000000e+00  -3.33066907e-16   1.00000000e+00]]
```

EXAMPLE 2.14
Invert the matrix

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

Solution. Because the matrix is tridiagonal, we solve $\mathbf{AX} = \mathbf{I}$ using the functions in the module `LUdecomp3` (LU decomposition of tridiagonal matrices):

```
#!/usr/bin/python
## example2_14
import numpy as np
from LUdecomp3 import *

n = 6
d = np.ones((n))*2.0
e = np.ones((n-1))*(-1.0)
c = e.copy()
d[n-1] = 5.0
aInv = np.identity(n)
c,d,e = LUdecomp3(c,d,e)
for i in range(n):
    aInv[:,i] = LUsolve3(c,d,e,aInv[:,i])
print("\nThe inverse matrix is:\n",aInv)
input("\nPress return to exit")
```

Running the program results in the following output:

```
The inverse matrix is:
[[ 0.84  0.68  0.52  0.36  0.2   0.04]
 [ 0.68  1.36  1.04  0.72  0.4   0.08]
 [ 0.52  1.04  1.56  1.08  0.6   0.12]
 [ 0.36  0.72  1.08  1.44  0.8   0.16]
 [ 0.2   0.4   0.6   0.8   1.    0.2 ]
 [ 0.04  0.08  0.12  0.16  0.2   0.24]]]
```

Note that $\mathbf{A}$ is tridiagonal, whereas $\mathbf{A}^{-1}$ is fully populated.

*2.7 Iterative Methods

Introduction

So far, we have discussed only direct methods of solution. The common characteristic of these methods is that they compute the solution with a finite number of operations. Moreover, if the computer were capable of infinite precision (no roundoff errors), the solution would be exact.

Iterative, or *indirect methods*, start with an initial guess of the solution $\mathbf{x}$ and then repeatedly improve the solution until the change in $\mathbf{x}$ becomes negligible. Because the required number of iterations can be large, the indirect methods are, in general, slower than their direct counterparts. However, iterative methods do have the following two advantages that make them attractive for certain problems:

1. It is feasible to store only the nonzero elements of the coefficient matrix. This makes it possible to deal with very large matrices that are sparse, but not necessarily banded. In many problems, there is no need to store the coefficient matrix at all.
2. Iterative procedures are self-correcting, meaning that roundoff errors (or even arithmetic mistakes) in one iterative cycle are corrected in subsequent cycles.

A serious drawback of iterative methods is that they do not always converge to the solution. It can be shown that convergence is guaranteed only if the coefficient matrix is diagonally dominant. The initial guess for $\mathbf{x}$ plays no role in determining whether convergence takes place—if the procedure converges for one starting vector, it would do so for any starting vector. The initial guess affects only the number of iterations that are required for convergence.

Gauss-Seidel Method

The equations $\mathbf{Ax} = \mathbf{b}$ are in scalar notation

$$\sum_{j=1}^{n} A_{ij}x_j = b_i, \quad i = 1, 2, \ldots, n$$

Extracting the term containing x_i from the summation sign yields

$$A_{ii}x_i + \sum_{\substack{j=1 \\ j\neq i}}^{n} A_{ij}x_j = b_i, \quad i = 1, 2, \ldots, n$$

Solving for x_i, we get

$$x_i = \frac{1}{A_{ii}}\left(b_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} A_{ij}x_j\right), \quad i = 1, 2, \ldots, n$$

The last equation suggests the following iterative scheme:

$$x_i \leftarrow \frac{1}{A_{ii}}\left(b_i - \sum_{\substack{j=1\\ j\neq i}}^{n} A_{ij}x_j\right), \quad i = 1, 2, \ldots, n \tag{2.34}$$

We start by choosing the starting vector $\mathbf{x}$. If a good guess for the solution is not available, $\mathbf{x}$ can be chosen randomly. Equation (2.34) is then used to recompute each element of $\mathbf{x}$, always using the latest available values of x_j. This completes one iteration cycle. The procedure is repeated until the changes in $\mathbf{x}$ between successive iteration cycles become sufficiently small.

Convergence of the Gauss-Seidel method can be improved by a technique known as *relaxation*. The idea is to take the new value of x_i as a weighted average of its previous value and the value predicted by Eq. (2.34). The corresponding iterative formula is

$$x_i \leftarrow \frac{\omega}{A_{ii}}\left(b_i - \sum_{\substack{j=1\\ j\neq i}}^{n} A_{ij}x_j\right) + (1-\omega)x_i, \quad i = 1, 2, \ldots, n \tag{2.35}$$

where the weight ω is called the *relaxation factor*. It can be seen that if $\omega = 1$, no relaxation takes place, because Eqs. (2.34) and (2.35) produce the same result. If $\omega < 1$, Eq. (2.35) represents interpolation between the old x_i and the value given by Eq. (2.34). This is called *under-relaxation*. In cases where $\omega > 1$, we have extrapolation, or *over-relaxation*.

There is no practical method of determining the optimal value of ω beforehand; however, an estimate can be computed during run time. Let $\Delta x^{(k)} = \left|\mathbf{x}^{(k-1)} - \mathbf{x}^{(k)}\right|$ be the magnitude of the change in $\mathbf{x}$ during the kth iteration (carried out without relaxation [i.e., with $\omega = 1$]). If k is sufficiently large (say $k \geq 5$), it can be shown[2] that an approximation of the optimal value of ω is

$$\omega_{\text{opt}} \approx \frac{2}{1 + \sqrt{1 - \left(\Delta x^{(k+p)}/\Delta x^{(k)}\right)^{1/p}}} \tag{2.36}$$

where p is a positive integer.

The essential elements of a Gauss-Seidel algorithm with relaxation are as follows:

Carry out k iterations with $\omega = 1$ ($k = 10$ is reasonable).
Record $\Delta x^{(k)}$.
Perform additional p iterations.
Record $\Delta x^{(k+p)}$.
Compute ω_{opt} from Eq. (2.36).
Perform all subsequent iterations with $\omega = \omega_{\text{opt}}$.

[2] See, for example, Terrence J. Akai, *Applied Numerical Methods for Engineers*, John Wiley & Sons (1994), p. 100.

■ gaussSeidel

The function `gaussSeidel` is an implementation of the Gauss-Seidel method with relaxation. It automatically computes ω_{opt} from Eq. (2.36) using $k = 10$ and $p = 1$. The user must provide the function `iterEqs` that computes the improved **x** from the iterative formulas in Eq. (2.35)—see Example 2.17. The function `gaussSeidel` returns the solution vector **x**, the number of iterations carried out, and the value of ω_{opt} used.

```
## module gaussSeidel
''' x,numIter,omega = gaussSeidel(iterEqs,x,tol = 1.0e-9)
    Gauss-Seidel method for solving [A]{x} = {b}.
    The matrix [A] should be sparse. User must supply the
    function iterEqs(x,omega) that returns the improved {x},
    given the current {x} ('omega' is the relaxation factor).
'''
import numpy as np
import math

def gaussSeidel(iterEqs,x,tol = 1.0e-9):
    omega = 1.0
    k = 10
    p = 1
    for i in range(1,501):
        xOld = x.copy()
        x = iterEqs(x,omega)
        dx = math.sqrt(np.dot(x-xOld,x-xOld))
        if dx < tol: return x,i,omega
      # Compute relaxation factor after k+p iterations
        if i == k: dx1 = dx
        if i == k + p:
            dx2 = dx
            omega = 2.0/(1.0 + math.sqrt(1.0  \
                  - (dx2/dx1)**(1.0/p)))
    print('Gauss-Seidel failed to converge')
```

Conjugate Gradient Method

Consider the problem of finding the vector **x** that minimizes the scalar function

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x} \tag{2.37}$$

where the matrix $\mathbf{A}$ is *symmetric* and *positive definite*. Because $f(\mathbf{x})$ is minimized when its gradient $\nabla f = \mathbf{A}\mathbf{x} - \mathbf{b}$ is zero, we see that minimization is equivalent to solving

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{2.38}$$

Gradient methods accomplish the minimization by iteration, starting with an initial vector $\mathbf{x}_0$. Each iterative cycle k computes a refined solution

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{s}_k \tag{2.39}$$

The *step length* α_k is chosen so that $\mathbf{x}_{k+1}$ minimizes $f(\mathbf{x}_{k+1})$ in the *search direction* $\mathbf{s}_k$. That is, $\mathbf{x}_{k+1}$ must satisfy Eq. (2.38):

$$\mathbf{A}(\mathbf{x}_k + \alpha_k \mathbf{s}_k) = \mathbf{b} \tag{a}$$

Introducing the *residual*

$$\mathbf{r}_k = \mathbf{b} - \mathbf{A}\mathbf{x}_k \tag{2.40}$$

Eq. (a) becomes $\alpha \mathbf{A}\mathbf{s}_k = \mathbf{r}_k$. Pre-multiplying both sides by $\mathbf{s}_k^T$ and solving for α_k, we obtain

$$\alpha_k = \frac{\mathbf{s}_k^T \mathbf{r}_k}{\mathbf{s}_k^T \mathbf{A}\mathbf{s}_k} \tag{2.41}$$

We are still left with the problem of determining the search direction $\mathbf{s}_k$. Intuition tells us to choose $\mathbf{s}_k = -\nabla f = \mathbf{r}_k$, because this is the direction of the largest negative change in $f(\mathbf{x})$. The resulting procedure is known as the *method of steepest descent*. It is not a popular algorithm because its convergence can be slow. The more efficient conjugate gradient method uses the search direction

$$\mathbf{s}_{k+1} = \mathbf{r}_{k+1} + \beta_k \mathbf{s}_k \tag{2.42}$$

The constant β_k is chosen so that the two successive search directions are *conjugate* to each other, meaning

$$\mathbf{s}_{k+1}^T \mathbf{A}\mathbf{s}_k = 0 \tag{b}$$

The great attraction of conjugate gradients is that minimization in one conjugate direction does not undo previous minimizations (minimizations do not interfere with one another).

Substituting $\mathbf{s}_{k+1}$ from Eq. (2.42) into Eq. (b), we get

$$\left(\mathbf{r}_{k+1}^T + \beta_k \mathbf{s}_k^T\right) \mathbf{A}\mathbf{s}_k = 0$$

which yields

$$\beta_k = -\frac{\mathbf{r}_{k+1}^T \mathbf{A}\mathbf{s}_k}{\mathbf{s}_k^T \mathbf{A}\mathbf{s}_k} \tag{2.43}$$

Here is the outline of the conjugate gradient algorithm:

Choose $\mathbf{x}_0$ (any vector will do).
Let $\mathbf{r}_0 \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}_0$.
Let $\mathbf{s}_0 \leftarrow \mathbf{r}_0$ (lacking a previous search direction, choose the direction of steepest descent).
do with $k = 0, 1, 2, \ldots$:
$\alpha_k \leftarrow \dfrac{\mathbf{s}_k^T \mathbf{r}_k}{\mathbf{s}_k^T \mathbf{A}\mathbf{s}_k}$.
$\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{s}_k$.
$\mathbf{r}_{k+1} \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}_{k+1}$.
if $|\mathbf{r}_{k+1}| \leq \varepsilon$ exit loop (ε is the error tolerance).
$\beta_k \leftarrow -\dfrac{\mathbf{r}_{k+1}^T \mathbf{A}\mathbf{s}_k}{\mathbf{s}_k^T \mathbf{A}\mathbf{s}_k}$.
$\mathbf{s}_{k+1} \leftarrow \mathbf{r}_{k+1} + \beta_k \mathbf{s}_k$.

It can be shown that the residual vectors $\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3, \ldots$ produced by the algorithm are mutually orthogonal; that is, $\mathbf{r}_i \cdot \mathbf{r}_j = 0$, $i \neq j$. Now suppose that we have carried out enough iterations to have computed the whole set of n residual vectors. The residual resulting from the next iteration must be a null vector ($\mathbf{r}_{n+1} = \mathbf{0}$), indicating that the solution has been obtained. It thus appears that the conjugate gradient algorithm is not an iterative method at all, because it reaches the exact solution after n computational cycles. In practice, however, convergence is usually achieved in less than n iterations.

The conjugate gradient method is not competitive with direct methods in the solution of small sets of equations. Its strength lies in the handling of large, sparse systems (where most elements of **A** are zero). It is important to note that **A** enters the algorithm only through its multiplication by a vector; that is, in the form **Av**, where **v** is a vector (either $\mathbf{x}_{k+1}$ or $\mathbf{s}_k$). If **A** is sparse, it is possible to write an efficient subroutine for the multiplication and pass it, rather than **A** itself, to the conjugate gradient algorithm.

■ conjGrad

The function `conjGrad` shown next implements the conjugate gradient algorithm. The maximum allowable number of iterations is set to n (the number of unknowns). Note that `conjGrad` calls the function `Av` that returns the product **Av**. This function must be supplied by the user (see Example 2.18). The user must also supply the starting vector $\mathbf{x}_0$ and the constant (right-hand-side) vector **b**. The function returns the solution vector **x** and the number of iterations.

```
## module conjGrad
''' x, numIter = conjGrad(Av,x,b,tol=1.0e-9)
    Conjugate gradient method for solving [A]{x} = {b}.
```

```
    The matrix [A] should be sparse. User must supply
    the function Av(v) that returns the vector [A]{v}.
'''
import numpy as np
import math

def conjGrad(Av,x,b,tol=1.0e-9):
    n = len(b)
    r = b - Av(x)
    s = r.copy()
    for i in range(n):
        u = Av(s)
        alpha = np.dot(s,r)/np.dot(s,u)
        x = x + alpha*s
        r = b - Av(x)
        if(math.sqrt(np.dot(r,r))) < tol:
            break
        else:
            beta = -np.dot(r,u)/np.dot(s,u)
            s = r + beta*s
    return x,i
```

EXAMPLE 2.15

Solve the equations

$$\begin{bmatrix} 4 & -1 & 1 \\ -1 & 4 & -2 \\ 1 & -2 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 12 \\ -1 \\ 5 \end{bmatrix}$$

by the Gauss-Seidel method without relaxation.

Solution. With the given data, the iteration formulas in Eq. (2.34) become

$$x_1 = \frac{1}{4}(12 + x_2 - x_3)$$

$$x_2 = \frac{1}{4}(-1 + x_1 + 2x_3)$$

$$x_3 = \frac{1}{4}(5 - x_1 + 2x_2)$$

Choosing the starting values $x_1 = x_2 = x_3 = 0$, the first iteration gives us

$$x_1 = \frac{1}{4}(12 + 0 - 0) = 3$$

$$x_2 = \frac{1}{4}[-1 + 3 + 2(0)] = 0.5$$

$$x_3 = \frac{1}{4}[5 - 3 + 2(0.5)] = 0.75$$

The second iteration yields

$$x_1 = \frac{1}{4}(12 + 0.5 - 0.75) = 2.9375$$

$$x_2 = \frac{1}{4}[-1 + 2.9375 + 2(0.75)] = 0.859\,38$$

$$x_3 = \frac{1}{4}[5 - 2.9375 + 2(0.859\,38)] = 0\,.945\,31$$

and the third iteration results in

$$x_1 = \frac{1}{4}(12 + 0.85938 - 0\,.94531) = 2.978\,52$$

$$x_2 = \frac{1}{4}[-1 + 2.97852 + 2(0\,.94531)] = 0.967\,29$$

$$x_3 = \frac{1}{4}[5 - 2.97852 + 2(0.96729)] = 0.989\,02$$

After five more iterations the results would agree with the exact solution $x_1 = 3$, $x_2 = x_3 = 1$ within five decimal places.

EXAMPLE 2.16

Solve the equations in Example 2.15 by the conjugate gradient method.

Solution. The conjugate gradient method should converge after three iterations. Choosing again for the starting vector

$$\mathbf{x}_0 = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T,$$

the computations outlined in the text proceed as follows:

First Iteration

$$\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0 = \begin{bmatrix} 12 \\ -1 \\ 5 \end{bmatrix} - \begin{bmatrix} 4 & -1 & 1 \\ -1 & 4 & -2 \\ 1 & -2 & 4 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 12 \\ -1 \\ 5 \end{bmatrix}$$

$$\mathbf{s}_0 = \mathbf{r}_0 = \begin{bmatrix} 12 \\ -1 \\ 5 \end{bmatrix}$$

$$\mathbf{A}\mathbf{s}_0 = \begin{bmatrix} 4 & -1 & 1 \\ -1 & 4 & -2 \\ 1 & -2 & 4 \end{bmatrix} \begin{bmatrix} 12 \\ -1 \\ 5 \end{bmatrix} = \begin{bmatrix} 54 \\ -26 \\ 34 \end{bmatrix}$$

$$\alpha_0 = \frac{\mathbf{s}_0^T \mathbf{r}_0}{\mathbf{s}_0^T \mathbf{A}\mathbf{s}_0} = \frac{12^2 + (-1)^2 + 5^2}{12(54) + (-1)(-26) + 5(34)} = 0.201\,42$$

$$\mathbf{x}_1 = \mathbf{x}_0 + \alpha_0 \mathbf{s}_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + 0.201\,42 \begin{bmatrix} 12 \\ -1 \\ 5 \end{bmatrix} = \begin{bmatrix} 2.41\,704 \\ -0.\,201\,42 \\ 1.007\,10 \end{bmatrix}$$

Second Iteration

$$\mathbf{r}_1 = \mathbf{b} - \mathbf{A}\mathbf{x}_1 = \begin{bmatrix} 12 \\ -1 \\ 5 \end{bmatrix} - \begin{bmatrix} 4 & -1 & 1 \\ -1 & 4 & -2 \\ 1 & -2 & 4 \end{bmatrix} \begin{bmatrix} 2.417\,04 \\ -0.201\,42 \\ 1.007\,10 \end{bmatrix} = \begin{bmatrix} 1.123\,32 \\ 4.236\,92 \\ -1.848\,28 \end{bmatrix}$$

$$\beta_0 = -\frac{\mathbf{r}_1^T \mathbf{A}\mathbf{s}_0}{\mathbf{s}_0^T \mathbf{A}\mathbf{s}_0} = -\frac{1.123\,32(54) + 4.236\,92(-26) - 1.848\,28(34)}{12(54) + (-1)(-26) + 5(34)} = 0.133\,107$$

$$\mathbf{s}_1 = \mathbf{r}_1 + \beta_0 \mathbf{s}_0 = \begin{bmatrix} 1.123\,32 \\ 4.236\,92 \\ -1.848\,28 \end{bmatrix} + 0.133\,107 \begin{bmatrix} 12 \\ -1 \\ 5 \end{bmatrix} = \begin{bmatrix} 2.720\,76 \\ 4.103\,80 \\ -1.182\,68 \end{bmatrix}$$

$$\mathbf{A}\mathbf{s}_1 = \begin{bmatrix} 4 & -1 & 1 \\ -1 & 4 & -2 \\ 1 & -2 & 4 \end{bmatrix} \begin{bmatrix} 2.720\,76 \\ 4.103\,80 \\ -1.182\,68 \end{bmatrix} = \begin{bmatrix} 5.596\,56 \\ 16.059\,80 \\ -10.217\,60 \end{bmatrix}$$

$$\alpha_1 = \frac{\mathbf{s}_1^T \mathbf{r}_1}{\mathbf{s}_1^T \mathbf{A}\mathbf{s}_1}$$

$$= \frac{2.720\,76(1.123\,32) + 4.103\,80(4.236\,92) + (-1.182\,68)(-1.848\,28)}{2.720\,76(5.596\,56) + 4.103\,80(16.059\,80) + (-1.182\,68)(-10.217\,60)}$$

$$= 0.24276$$

$$\mathbf{x}_2 = \mathbf{x}_1 + \alpha_1 \mathbf{s}_1 = \begin{bmatrix} 2.417\,04 \\ -0.201\,42 \\ 1.007\,10 \end{bmatrix} + 0.24276 \begin{bmatrix} 2.720\,76 \\ 4.103\,80 \\ -1.182\,68 \end{bmatrix} = \begin{bmatrix} 3.077\,53 \\ 0.794\,82 \\ 0.719\,99 \end{bmatrix}$$

Third Iteration

$$\mathbf{r}_2 = \mathbf{b} - \mathbf{A}\mathbf{x}_2 = \begin{bmatrix} 12 \\ -1 \\ 5 \end{bmatrix} - \begin{bmatrix} 4 & -1 & 1 \\ -1 & 4 & -2 \\ 1 & -2 & 4 \end{bmatrix} \begin{bmatrix} 3.077\,53 \\ 0.794\,82 \\ 0.719\,99 \end{bmatrix} = \begin{bmatrix} -0.235\,29 \\ 0.338\,23 \\ 0.632\,15 \end{bmatrix}$$

$$\beta_1 = -\frac{\mathbf{r}_2^T \mathbf{A}\mathbf{s}_1}{\mathbf{s}_1^T \mathbf{A}\mathbf{s}_1}$$

$$= -\frac{(-0.235\,29)(5.596\,56) + 0.338\,23(16.059\,80) + 0.632\,15(-10.217\,60)}{2.720\,76(5.596\,56) + 4.103\,80(16.059\,80) + (-1.182\,68)(-10.217\,60)}$$

$$= 0.0251\,452$$

$$\mathbf{s}_2 = \mathbf{r}_2 + \beta_1 \mathbf{s}_1 = \begin{bmatrix} -0.235\,29 \\ 0.338\,23 \\ 0.632\,15 \end{bmatrix} + 0.025\,1452 \begin{bmatrix} 2.720\,76 \\ 4.103\,80 \\ -1.182\,68 \end{bmatrix} = \begin{bmatrix} -0.166\,876 \\ 0.441\,421 \\ 0.602\,411 \end{bmatrix}$$

$$\mathbf{A}\mathbf{s}_2 = \begin{bmatrix} 4 & -1 & 1 \\ -1 & 4 & -2 \\ 1 & -2 & 4 \end{bmatrix} \begin{bmatrix} -0.166\,876 \\ 0.441\,421 \\ 0.602\,411 \end{bmatrix} = \begin{bmatrix} -0.506\,514 \\ 0.727\,738 \\ 1.359\,930 \end{bmatrix}$$

$$\begin{aligned} \alpha_2 &= \frac{\mathbf{r}_2^T \mathbf{s}_2}{\mathbf{s}_2^T \mathbf{A}\mathbf{s}_2} \\ &= \frac{(-0.235\,29)(-0.166\,876) + 0.338\,23(0.441\,421) + 0.632\,15(0.602\,411)}{(-0.166\,876)(-0.506\,514) + 0.441\,421(0.727\,738) + 0.602\,411(1.359\,930)} \\ &= 0.464\,80 \end{aligned}$$

$$\mathbf{x}_3 = \mathbf{x}_2 + \alpha_2 \mathbf{s}_2 = \begin{bmatrix} 3.077\,53 \\ 0.794\,82 \\ 0.719\,99 \end{bmatrix} + 0.464\,80 \begin{bmatrix} -0.166\,876 \\ 0.441\,421 \\ 0.602\,411 \end{bmatrix} = \begin{bmatrix} 2.999\,97 \\ 0.999\,99 \\ 0.999\,99 \end{bmatrix}$$

The solution $\mathbf{x}_3$ is correct to almost five decimal places. The small discrepancy may be caused by roundoff errors in the computations.

EXAMPLE 2.17

Write a computer program to solve the following n simultaneous equations by the Gauss-Seidel method with relaxation (the program should work with any value of n)[3]:

$$\begin{bmatrix} 2 & -1 & 0 & 0 & \dots & 0 & 0 & 0 & 1 \\ -1 & 2 & -1 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & \dots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & -1 & 2 & -1 \\ 1 & 0 & 0 & 0 & \dots & 0 & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-2} \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Run the program with $n = 20$. The exact solution can be shown to be $x_i = -n/4 + i/2$, $i = 1, 2, \dots, n$.

Solution. In this case the iterative formulas in Eq. (2.35) are

$$\begin{aligned} x_1 &= \omega(x_2 - x_n)/2 + (1 - \omega)x_1 \\ x_i &= \omega(x_{i-1} + x_{i+1})/2 + (1 - \omega)x_i, \quad i = 2, 3, \dots, n-1 \\ x_n &= \omega(1 - x_1 + x_{n-1})/2 + (1 - \omega)x_n \end{aligned} \tag{a}$$

These formulas are evaluated in the function `iterEqs`.

[3] Equations of this form are called *cyclic* tridiagonal. They occur in the finite difference formulation of second-order differential equations with periodic boundary conditions.

```
#!/usr/bin/python
## example2_17
import numpy as np
from gaussSeidel import *

def iterEqs(x,omega):
    n = len(x)
    x[0] = omega*(x[1] - x[n-1])/2.0 + (1.0 - omega)*x[0]
    for i in range(1,n-1):
        x[i] = omega*(x[i-1] + x[i+1])/2.0 + (1.0 - omega)*x[i]
    x[n-1] = omega*(1.0 - x[0] + x[n-2])/2.0 \
           + (1.0 - omega)*x[n-1]
    return x

n = eval(input("Number of equations ==> "))
x = np.zeros(n)
x,numIter,omega = gaussSeidel(iterEqs,x)
print("\nNumber of iterations =",numIter)
print("\nRelaxation factor =",omega)
print("\nThe solution is:\n",x)
input("\nPress return to exit")
```

The output from the program is

```
Number of equations ==> 20

Number of iterations = 259

Relaxation factor = 1.7054523107131399

The solution is:
 [ -4.50000000e+00 -4.00000000e+00 -3.50000000e+00 -3.00000000e+00
  -2.50000000e+00 -2.00000000e+00 -1.50000000e+00 -9.99999997e-01
  -4.99999998e-01  2.14047151e-09  5.00000002e-01  1.00000000e+00
   1.50000000e+00  2.00000000e+00  2.50000000e+00  3.00000000e+00
   3.50000000e+00  4.00000000e+00  4.50000000e+00  5.00000000e+00]
```

The convergence is very slow, because the coefficient matrix lacks diagonal dominance—substituting the elements of **A** into Eq. (2.30) produces an equality rather than the desired inequality. If we were to change each diagonal term of the coefficient from 2 to 4, **A** would be diagonally dominant and the solution would converge in only 17 iterations.

EXAMPLE 2.18

Solve Example 2.17 with the conjugate gradient method, also using $n = 20$.

Solution. The program shown next uses the function `conjGrad`. The solution vector **x** is initialized to zero in the program. The function `Ax(v)` returns the product $\mathbf{A} \cdot \mathbf{v}$, where **A** is the coefficient matrix and **v** is a vector. For the given **A**, the components of the vector $\mathbf{Ax}(\mathbf{v})$ are

$$(\mathbf{Ax})_1 = 2v_1 - v_2 + v_n$$

$$(\mathbf{Ax})_i = -v_{i-1} + 2v_i - v_{i+1}, \quad i = 2, 3, \ldots, n-1$$

$$(\mathbf{Ax})_n = -v_{n-1} + 2v_n + v_1$$

```
#!/usr/bin/python
## example2_18
import numpy as np
from conjGrad import *

def Ax(v):
    n = len(v)
    Ax = np.zeros(n)
    Ax[0] = 2.0*v[0] - v[1]+v[n-1]
    Ax[1:n-1] = -v[0:n-2] + 2.0*v[1:n-1] -v[2:n]
    Ax[n-1] = -v[n-2] + 2.0*v[n-1] + v[0]
    return Ax

n = eval(input("Number of equations ==> "))
b = np.zeros(n)
b[n-1] = 1.0
x = np.zeros(n)
x,numIter = conjGrad(Ax,x,b)
print("\nThe solution is:\n",x)
print("\nNumber of iterations =",numIter)
input("\nPress return to exit")
```

Running the program results in

```
Number of equations ==> 20

The solution is:
[-4.5 -4. -3.5 -3. -2.5 -2. -1.5 -1. -0.5  0.  0.5  1.  1.5
       2.  2.5  3.  3.5  4.  4.5  5. ]

Number of iterations = 9
```

Note that convergence was reached in only 9 iterations, whereas 259 iterations were required in the Gauss-Seidel method.

PROBLEM SET 2.3

1. Let

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

(note that **B** is obtained by interchanging the first two rows of **A**). Knowing that

$$\mathbf{A}^{-1} = \begin{bmatrix} 0.5 & 0 & 0.25 \\ 0.3 & 0.4 & 0.45 \\ -0.1 & 0.2 & -0.15 \end{bmatrix}$$

determine $\mathbf{B}^{-1}$.

2. Invert the triangular matrices:

$$\mathbf{A} = \begin{bmatrix} 2 & 4 & 3 \\ 0 & 6 & 5 \\ 0 & 0 & 2 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 2 & 0 & 0 \\ 3 & 4 & 0 \\ 4 & 5 & 6 \end{bmatrix}$$

3. Invert the triangular matrix:

$$\mathbf{A} = \begin{bmatrix} 1 & 1/2 & 1/4 & 1/8 \\ 0 & 1 & 1/3 & 1/9 \\ 0 & 0 & 1 & 1/4 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

4. Invert the following matrices:

$$\text{(a) } \mathbf{A} = \begin{bmatrix} 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 4 & 16 \end{bmatrix} \qquad \text{(b) } \mathbf{B} = \begin{bmatrix} 4 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 4 \end{bmatrix}$$

5. Invert this matrix:

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

6. ■ Invert the following matrices with any method:

$$\mathbf{A} = \begin{bmatrix} 5 & -3 & -1 & 0 \\ -2 & 1 & 1 & 1 \\ 3 & -5 & 1 & 2 \\ 0 & 8 & -4 & -3 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 4 & -1 & 0 & 0 \\ -1 & 4 & -1 & 0 \\ 0 & -1 & 4 & -1 \\ 0 & 0 & -1 & 4 \end{bmatrix}$$

7. ■ Invert the matrix by any method

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & -9 & 6 & 4 \\ 2 & -1 & 6 & 7 & 1 \\ 3 & 2 & -3 & 15 & 5 \\ 8 & -1 & 1 & 4 & 2 \\ 11 & 1 & -2 & 18 & 7 \end{bmatrix}$$

and comment on the reliability of the result.

8. ■ The joint displacements **u** of the plane truss in Prob. 14, Problem Set 2.2 are related to the applied joint forces **p** by

$$\mathbf{Ku} = \mathbf{p} \tag{a}$$

where

$$\mathbf{K} = \begin{bmatrix} 27.580 & 7.004 & -7.004 & 0.000 & 0.000 \\ 7.004 & 29.570 & -5.253 & 0.000 & -24.320 \\ -7.004 & -5.253 & 29.570 & 0.000 & 0.000 \\ 0.000 & 0.000 & 0.000 & 27.580 & -7.004 \\ 0.000 & -24.320 & 0.000 & -7.004 & 29.570 \end{bmatrix} \text{MN/m}$$

is called the *stiffness matrix* of the truss. If Eq. (a) is inverted by multiplying each side by $\mathbf{K}^{-1}$, we obtain $\mathbf{u} = \mathbf{K}^{-1}\mathbf{p}$, where $\mathbf{K}^{-1}$ is known as the *flexibility matrix*. The physical meaning of the elements of the flexibility matrix is K_{ij}^{-1} = displacements u_i $(i = 1, 2, \ldots 5)$ produced by the unit load $p_j = 1$. Compute (a) the flexibility matrix of the truss; (b) the displacements of the joints due to the load $p_5 = -45$ kN (the load shown in Problem 14, Problem Set 2.2).

9. ■ Invert the matrices:

$$\mathbf{A} = \begin{bmatrix} 3 & -7 & 45 & 21 \\ 12 & 11 & 10 & 17 \\ 6 & 25 & -80 & -24 \\ 17 & 55 & -9 & 7 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 2 \\ 2 & 3 & 4 & 4 \\ 4 & 5 & 6 & 7 \end{bmatrix}$$

10. ■ Write a program for inverting an $n \times n$ lower triangular matrix. The inversion procedure should contain only forward substitution. Test the program by inverting this matrix:

$$\mathbf{A} = \begin{bmatrix} 36 & 0 & 0 & 0 \\ 18 & 36 & 0 & 0 \\ 9 & 12 & 36 & 0 \\ 5 & 4 & 9 & 36 \end{bmatrix}$$

11. Use the Gauss-Seidel method to solve

$$\begin{bmatrix} -2 & 5 & 9 \\ 7 & 1 & 1 \\ -3 & 7 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 6 \\ -26 \end{bmatrix}$$

12. Solve the following equations with the Gauss-Seidel method:

$$\begin{bmatrix} 12 & -2 & 3 & 1 \\ -2 & 15 & 6 & -3 \\ 1 & 6 & 20 & -4 \\ 0 & -3 & 2 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 20 \\ 0 \end{bmatrix}$$

13. Use the Gauss-Seidel method with relaxation to solve $\mathbf{Ax} = \mathbf{b}$, where

$$\mathbf{A} = \begin{bmatrix} 4 & -1 & 0 & 0 \\ -1 & 4 & -1 & 0 \\ 0 & -1 & 4 & -1 \\ 0 & 0 & -1 & 3 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 15 \\ 10 \\ 10 \\ 10 \end{bmatrix}$$

Take $x_i = b_i/A_{ii}$ as the starting vector, and use $\omega = 1.1$ for the relaxation factor.

14. Solve the equations

$$\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

by the conjugate gradient method. Start with $\mathbf{x} = \mathbf{0}$.

15. Use the conjugate gradient method to solve

$$\begin{bmatrix} 3 & 0 & -1 \\ 0 & 4 & -2 \\ -1 & -2 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 10 \\ -10 \end{bmatrix}$$

starting with $\mathbf{x} = \mathbf{0}$.

16. ■ Write a program for solving $\mathbf{Ax} = \mathbf{b}$ by the Gauss-Seidel method based on the function `gaussSeidel`. Input should consist of the matrix $\mathbf{A}$ and the vector $\mathbf{b}$. Test the program with

$$\mathbf{A} = \begin{bmatrix} 3 & -2 & 1 & 0 & 0 & 1 \\ -2 & 4 & -2 & 1 & 0 & 0 \\ 1 & -2 & 4 & -2 & 1 & 0 \\ 0 & 1 & -2 & 4 & -2 & 1 \\ 0 & 0 & 1 & -2 & 4 & -2 \\ 1 & 0 & 0 & 1 & -2 & 3 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 10 \\ -8 \\ 10 \\ 10 \\ -8 \\ 10 \end{bmatrix}$$

Note that $\mathbf{A}$ is not diagonally dominant, but this does not necessarily preclude convergence.

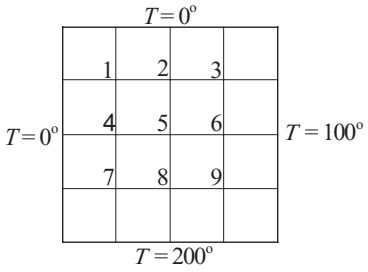
17. ■ Modify the program in Example 2.17 (Gauss-Seidel method) so that it will solve the following equations:

$$\begin{bmatrix} 4 & -1 & 0 & 0 & \cdots & 0 & 0 & 0 & 1 \\ -1 & 4 & -1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & -1 & \cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & -1 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & -1 & 4 & -1 \\ 1 & 0 & 0 & 0 & \cdots & 0 & 0 & -1 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-2} \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 100 \end{bmatrix}$$

Run the program with $n = 20$ and compare the number of iterations with Example 2.17.

18. ■ Modify the program in Example 2.18 to solve the equations in Prob. 17 by the conjugate gradient method. Run the program with $n = 20$.

19. ■



The edges of the square plate are kept at the temperatures shown. Assuming steady-state heat conduction, the differential equation governing the temperature T in the interior is

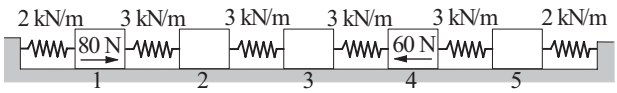
$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0.$$

If this equation is approximated by finite differences using the mesh shown, we obtain the following algebraic equations for temperatures at the mesh points:

$$\begin{bmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \\ T_5 \\ T_6 \\ T_7 \\ T_8 \\ T_9 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 100 \\ 0 \\ 0 \\ 100 \\ 200 \\ 200 \\ 300 \end{bmatrix}$$

Solve these equations with the conjugate gradient method.

20. ■



The equilibrium equations of the blocks in the spring-block system are

$$3(x_2 - x_1) - 2x_1 = -80$$

$$3(x_3 - x_2) - 3(x_2 - x_1) = 0$$

$$3(x_4 - x_3) - 3(x_3 - x_2) = 0$$

$$3(x_5 - x_4) - 3(x_4 - x_3) = 60$$

$$-2x_5 - 3(x_5 - x_4) = 0$$

where x_i are the horizontal displacements of the blocks measured in mm. (a) Write a program that solves these equations by the Gauss-Seidel method without relaxation. Start with $\mathbf{x} = \mathbf{0}$ and iterate until four-figure accuracy after the decimal point is achieved. Also print the number of iterations required. (b) Solve the equations using the function `gaussSeidel` using the same convergence criterion as in part (a). Compare the number of iterations in parts (a) and (b).

21. ■ Solve the equations in Prob. 20 with the conjugate gradient method using the function `conjGrad`. Start with $\mathbf{x} = \mathbf{0}$ and iterate until four-figure accuracy after the decimal point is achieved.

2.8 Other Methods

A matrix can be decomposed in numerous ways, some of which are generally useful, whereas others find use in special applications. The most important of the latter are the QR factorization and the singular value decomposition.

The *QR decomposition* of a matrix $\mathbf{A}$ is

$$\mathbf{A} = \mathbf{QR}$$

where $\mathbf{Q}$ is an orthogonal matrix (recall that the matrix $\mathbf{Q}$ is orthogonal if $\mathbf{Q}^{-1} = \mathbf{Q}^T$) and $\mathbf{R}$ is an upper triangular matrix. Unlike LU factorization, QR decomposition does not require pivoting to sustain stability, but it does involve about twice as many operations. Because of its relative inefficiency, the QR factorization is not used as a general-purpose tool, but finds it niche in applications that put a premium on stability (e.g., solution of eigenvalue problems). The `numpy` module includes the function `qr` that does the factorization:

```
Q,R = numpy.linalg.qr(A)
```

The *singular value decomposition* (SVD) is a useful diagnostic tool for singular or ill-conditioned matrices. Here the factorization is

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{V}^T$$

where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices and

$$\boldsymbol{\Lambda} = \begin{bmatrix} \lambda_1 & 0 & 0 & \cdots \\ 0 & \lambda_2 & 0 & \cdots \\ 0 & 0 & \lambda_3 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

is a diagonal matrix. The λ's are called the *singular values* of the matrix $\mathbf{A}$. They can be shown to be positive or zero. If $\mathbf{A}$ is symmetric and positive definite, then the

λ's are the eigenvalues of **A**. A nice characteristic of the singular value decomposition is that it works even if **A** is singular or ill conditioned. The conditioning of **A** can be diagnosed from magnitudes of λ's: The matrix is singular if one or more of the λ's are zero, and it is ill conditioned if the condition number $\lambda_{max}/\lambda_{min}$ is very large. The singular value decomposition function that comes with the `numpy` module is `svd`:

```
U,lam,V = numpy.linalg.svd(A)
```