Mathematics, rightly viewed, possesses not only truth, but supreme beauty – a beauty cold and austere, like that of sculpture.

— BERTRAND RUSSELL

Chapter 3

System of linear algebraic equation

Topics from linear algebra form the core of numerical analysis. Almost every conceivable problem, be it curve fitting, optimization, simulation of flow sheets or simulation of distributed parameter systems requiring solution of differential equations, require at some stage the solution of a system (often a large system!) of algebraic equations. MATLAB (acronym for MATrix LABoratory) was in fact conceived as a collection of tools to aid in the interactive learning and analysis of linear systems and was derived from a well known core of linear algebra routines written in FORTRAN called LINPACK.

In this chapter we provide a quick review of concepts from linear algebra. We make frequent reference to MATLAB implimentation of various concepts throughout this chapter. The reader is encouraged to try out these interactively during a MATLAB session. For a more complete treatment of topics in linear algebra see Hager (1985) and Barnett (1990). The text by Amundson (1966) is also an excellent source with specific examples drawn from Chemical Engineering. For a more rigorous, axiomatic introduction within the frame work of linear opeartor theory see Ramakrishna and Amundson (1985).

3.1 Matrix notation

We have already used the matrix notation to write a system of linear algebraic equations in a compact form in sections §1.3.1 and §1.3.2. While a matrix, as an object, is represented in bold face, its constituent elements are represented in index notation or as subscripted arrays in programming languages. For example the following are equivalent.

$$\boldsymbol{A} = [a_{ij}], \qquad i = 1, \cdots, m; \quad j = 1, \cdots, n$$

where $\boldsymbol{A}$ is an $m \times n$ matrix. a_{ij} represents an element of the matrix $\boldsymbol{A}$ in row i and column j position. A vector can be thought of as an object with a single row or column. A row vector is represented by,

$$\boldsymbol{x} = [x_1 \; x_2 \cdots x_n]$$

while a column vector can be represented by,

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

These elements can be *real* or *complex*.

Having defined objects like vectors and matrices, we can extend the notions of basic arithmetic opeartions between scalar numbers to higher dimensional objects like vectors and matrices. The reasons for doing so are many. It not only allows us to express a large system of equations in a compact symbolic form, but a study of the properties of such objects allows us to develop and codify very efficient ways of solving and analysing large linear systems. Packages like MATLAB and Mathematica present to us a vast array of such codified algorithms. As an engineer you should develop a conceptual understanding of the underlying principles and the skills to use such packages. But the most important task is to indentify each element of a vector or a matrix, which is tied closely to the physical description of the problem.

3.1.1 Review of basic operations

The arithmetic opeartions are defined both in symbolic form and using index notation. The later actually provides the algorithm for implementing the rules of operation using any programing language. The syntax of these operations in MATLAB are shown with specific examples.*

*MATLAB illustrations have been tested with Ver 5.0.0.4064

The addition operation between two matrices is defined as,

$$\text{addition:} \qquad \boldsymbol{A} = \boldsymbol{B} + \boldsymbol{C} \qquad \Rightarrow \qquad a_{ij} = b_{ij} + c_{ij}$$

This implies an element-by-element addition of the matrices $\boldsymbol{B}$ and $\boldsymbol{C}$. Clearly all the matrices involved must have the same dimension. Note that the addition operation is *commutative* as seen easily with its scalar counter part. *i.e.*,

$$\boldsymbol{A} + \boldsymbol{B} = \boldsymbol{B} + \boldsymbol{A}$$

Matrix addition is also *associative*, *i.e.*, independent of the order in which it is carried out, *e.g.*,

$$\boldsymbol{A} + \boldsymbol{B} + \boldsymbol{C} = (\boldsymbol{A} + \boldsymbol{B}) + \boldsymbol{C} = \boldsymbol{A} + (\boldsymbol{B} + \boldsymbol{C})$$

The scalar multiplication of a matrix involves multiplying each element of the matrix by the scalar, *i.e.*,

$$\text{scalar multiplication:} \qquad k\boldsymbol{A} = \boldsymbol{B} \qquad \Rightarrow \qquad k\, a_{ij} = b_{ij}$$

Subtraction operation can be handled by combining addition and sccalar multiplication rules as follwos:

$$\text{subtraction:} \qquad \boldsymbol{C} = \boldsymbol{A} + (-1)\boldsymbol{B} = \boldsymbol{A} - \boldsymbol{B} \qquad \Rightarrow \qquad c_{ij} = a_{ij} - b_{ij}$$

The product between two matrices $\boldsymbol{A}$ (of dimension $n \times m$) and $\boldsymbol{B}$ (of dimension $m \times r$) is defined as,

$$\text{multiplication:} \qquad \text{C=AB} \qquad \Rightarrow \qquad c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj} \qquad \forall \qquad i, j$$

and the resultant matrix has the dimension $n \times r$. The operation indicated in the index notation is carried out for each value of the free indices $i = 1 \cdots n$ and $j = 1 \cdots r$. The product is defined only if the dimensions of $\boldsymbol{B}$, $\boldsymbol{C}$ are compatible - *i.e.*, number of *columns* in $\boldsymbol{B}$ should equal the number of *rows* in $\boldsymbol{C}$. This implies that while the product $\boldsymbol{B}$ $\boldsymbol{C}$ may be defined, the product $\boldsymbol{C}$ $\boldsymbol{B}$ may not even be defined! Even when they are dimensionally compatible, in general

$$\boldsymbol{BC} \neq \boldsymbol{BC}$$

i.e., matrix multiplication is not *commutative*.

Example

Consider the matrices $\boldsymbol{A}$, $\boldsymbol{B}$ defined below.

$$\boldsymbol{A} = \begin{bmatrix} 2 & 3 & 4 \\ 1 & 3 & 2 \end{bmatrix}, \qquad \boldsymbol{B} = \begin{bmatrix} 1 & 2 \\ 3 & 1 \\ 4 & 1 \end{bmatrix}$$

In MATLAB they will be defined as follows:

```
»A=[2 3 4;1 3 2] % Define (2x3) matrix A. Semicolon separates rows
»B=[1 2; 3 1; 4 1]% Define (3x2) matrix B
»C=A*B           % calculate the product
C=               % Display the result
```

$$\begin{bmatrix} 27 & 11 \\ 18 & 7 \end{bmatrix}$$

Other useful products can also be defined between vectors and matrices.
A Hadamard (or Schur) product is defined as

$$C = A \circ B \qquad \Rightarrow \qquad c_{ij} = a_{ij} b_{ij} \qquad \forall \qquad i, j$$

Obviously, the dimension of $\boldsymbol{A}$ and $\boldsymbol{B}$ should be the same.

Example

The example below illustrates the Hadamard product, called the array product in MATLAB.

```
»C=A'.*B         % Note the dimensions are made the same by transpose of A
C=               % Display the result
```

$$\begin{bmatrix} \mathbf{2} & \mathbf{2} \\ \mathbf{9} & \mathbf{3} \\ \mathbf{16} & \mathbf{2} \end{bmatrix}$$

A Kronecker product is defined as

$$\boldsymbol{C} = \boldsymbol{A} \otimes \boldsymbol{B} \qquad \Rightarrow \qquad \boldsymbol{C} = \begin{bmatrix} \boldsymbol{a}_{11}\boldsymbol{B} & \boldsymbol{a}_{12}\boldsymbol{B} & \cdots & \boldsymbol{a}_{1m}\boldsymbol{B} \\ \boldsymbol{a}_{21}\boldsymbol{B} & \boldsymbol{a}_{22}\boldsymbol{B} & \cdots & \boldsymbol{a}_{2m}\boldsymbol{B} \\ \vdots & & & \\ \boldsymbol{a}_{n1}\boldsymbol{B} & \boldsymbol{a}_{n2}\boldsymbol{B} & \cdots & \boldsymbol{a}_{nm}\boldsymbol{B} \end{bmatrix}$$

Multiplying a scalar number by unity leaves it unchanged. Extension of this notion to matrices resutls in the definition of *identity* matrix,

$$\boldsymbol{I} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & 0 & 1 & 0 \\ 0 & 0 & \cdots & 1 \end{bmatrix} \qquad \Rightarrow \qquad \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

MATLAB function that generates the Kronecker product between matrices is
`C=kron(A,B)`
MATLAB function for producing an identity matrix of size N is
`I=eye(N)`

Multiplying any matrix $\boldsymbol{A}$ with an identity matrix $\boldsymbol{I}$ of appropriate dimension leaves the original matrix unchanged, *i.e.*,

$$\boldsymbol{AI} = \boldsymbol{A}$$

This allows us to generalize the notion of division with scalar numbers to matrices. Division operation can be thought of as the inverse of the multiplciation operation. For example, given a number, say 2, we can define its inverse, $\boldsymbol{x}$ in such a way that the product of the two numbers produce unity. *i.e.*, $2 \times \boldsymbol{x} = 1$ or $\boldsymbol{x} = 2^{-1}$. In a similar way, given a matrix $\boldsymbol{A}$, can we define the inverse matrix $\boldsymbol{B}$ such that

MATLAB function for finding the inverse of a matrix A is
`B=inv(A)`

$$\boldsymbol{AB} = \boldsymbol{I} \qquad \text{or} \qquad \boldsymbol{B} = \boldsymbol{A}^{-1}$$

The task of developing an algorithm for finding the inverse of a matrix will be addressed late in this chapter.

For a square matrix, *powers* of a matrix $\boldsymbol{A}$ can be defined as,

$$\boldsymbol{A}^2 = \boldsymbol{AA} \qquad \boldsymbol{A}^3 = \boldsymbol{AAA} = \boldsymbol{A}^2\boldsymbol{A} = \boldsymbol{AA}^2$$

Note that $\boldsymbol{A}^p\boldsymbol{A}^q = \boldsymbol{A}^{p+q}$ for positve integers $\boldsymbol{p}$ and $\boldsymbol{q}$. Having extended the definition of powers, we can extend the definition of *exponential* from scalars to square matrices as follows. For a scalar $\boldsymbol{\alpha}$ it is,

MATLAB operator for producing the n-th power of a matrix A is,
`A^n`
while the syntax for producing element-by-element power is,
`A.^n`.
Make sure that you understand the difference between these two operations!

$$e^{\alpha} = 1 + \alpha + \frac{\alpha^2}{2} + \cdots = \sum_{k=0}^{\infty} \frac{\alpha^k}{k!}$$

For a matrix $\boldsymbol{A}$ the *exponential matrix* can be defined as,

$$e^{\boldsymbol{A}} = \boldsymbol{I} + \boldsymbol{A} + \frac{\boldsymbol{A}^2}{2} + \cdots = \sum_{k=0}^{\infty} \frac{\boldsymbol{A}^k}{k!}$$

One operation that does not have a direct counter part in the scalar world is the *transpose* of a matrix. It is defined the result of exchanging the rows and columns of a matrix, *i.e.*,

$$\boldsymbol{B} = \boldsymbol{A}' \qquad \Rightarrow \qquad b_{ij} = a_{ji}$$

MATLAB function
`exp(A)`
evaluates the exponential element-by-element while
`expm(A)`
evaluates the true matrix exponential.

It is easy to verify that

$$(\boldsymbol{A} + \boldsymbol{B})' = \boldsymbol{A}' + \boldsymbol{B}'$$

Something that is not so easy to verify, nevertheless true, is

$$(\boldsymbol{AB})' = \boldsymbol{B}'\boldsymbol{A}'$$

3.2 Matrices with special structure

A *diagonal* matirx $\boldsymbol{D}$ has non-zero elements only along the diagonal.

$$\boldsymbol{D} = \begin{bmatrix} \boldsymbol{d}_{11} & \mathbf{o} & \cdots & \mathbf{o} \\ \mathbf{o} & \boldsymbol{d}_{22} & \cdots & \mathbf{o} \\ \vdots & \mathbf{o} & \ddots & \mathbf{o} \\ \mathbf{o} & \mathbf{o} & \cdots & \boldsymbol{d}_{nn} \end{bmatrix}$$

A *lower triangular* matrix $\boldsymbol{L}$ has non-zero elements on or below the diagonal,

$$\boldsymbol{L} = \begin{bmatrix} \boldsymbol{l}_{11} & \mathbf{o} & \cdots & \mathbf{o} \\ \boldsymbol{l}_{21} & \boldsymbol{l}_{22} & \cdots & \mathbf{o} \\ \vdots & & \ddots & \mathbf{o} \\ \boldsymbol{l}_{n1} & \boldsymbol{l}_{n2} & \cdots & \boldsymbol{l}_{nn} \end{bmatrix}$$

A *upper triangular* matrix $\boldsymbol{U}$ has non-zero elements on or above the diagonal,

$$\boldsymbol{U} = \begin{bmatrix} \boldsymbol{u}_{11} & \boldsymbol{u}_{12} & \cdots & \boldsymbol{u}_{1n} \\ \mathbf{o} & \boldsymbol{u}_{22} & \cdots & \boldsymbol{u}_{2n} \\ \mathbf{o} & \mathbf{o} & \ddots & \vdots \\ \mathbf{o} & \mathbf{o} & \cdots & \boldsymbol{u}_{nn} \end{bmatrix}$$

A *tridiagonal* matrix $\boldsymbol{T}$ has non-zero elements on the diagonal and one off diagonal row on each side of the diagonal

$$\boldsymbol{T} = \begin{bmatrix} \boldsymbol{t}_{11} & \boldsymbol{t}_{12} & \mathbf{o} & \cdots & \mathbf{o} \\ \boldsymbol{t}_{21} & \boldsymbol{t}_{22} & \boldsymbol{t}_{23} & \mathbf{o} & \mathbf{o} \\ \mathbf{o} & \ddots & \ddots & \ddots & \mathbf{o} \\ \vdots & \mathbf{o} & \boldsymbol{t}_{n-1,n-2} & \boldsymbol{t}_{n-1,n-1} & \boldsymbol{t}_{n-1,n} \\ \mathbf{o} & \cdots & \mathbf{o} & \boldsymbol{t}_{n,n-1} & \boldsymbol{t}_{n,n} \end{bmatrix}$$

A *sparse* matrix is a generic term to indicate those matrices without any specific strucutre such as above, but with a small number (typically 10 to 15 %) of non-zero elements.

3.3 Determinant

A determinant of a square matrix is defined in such a way that a scalar value is associated with the matrix that does not change with certain row or column operations on the matrix - *i.e.*, it is one of the scalar invariants of the matrix. In the context of solving a system of linear equations, the determinant is also useful in knowing whether the system of equations is solvable uniquely. The determinant is formed by summing *all* possible products formed by choosing *one and only one* element from each row and column of the matrix. The precise definition, taken from Amundson (1966), is

$$\boxed{\boldsymbol{det(A)} = |A| = \sum (-1)^h (a_{1l_1} a_{2l_2} \cdots a_{nl_n})} \tag{3.1}$$

Each term in the summation consists of a product of n elements selected such that only one element appears from each row and column. The summation involves a total of $n!$ terms accounted for as follows: for the first element l_1 in the product there are n choices, followed by $(n-1)$ choices for the second element l_2, $(n-2)$ choices for the third element l_3 *etc.* resulting in a total of $n!$ choices for a particular product. Note that in this way of counting, the set of second subscripts $\{l_1, l_2, \cdots l_n\}$ will contain all of the numbers in the range 1 to n, but they will not be in their natural order $\{1, 2, \cdots n\}$. hence, h is the number of permutations required to arrange $\{l_1, l_2, \cdots l_n\}$ in their natural order.

MATLAB function for computing the determinant of a square matrix is `det(A)`

This definition is neither intutive nor computationaly efficient. But it is instructive in understanding the following properties of determinants.

1. The determinant of a diagonal matrix D, is simply the product of all the diagonal elements, *i.e.*,

$$\boldsymbol{det(D)} = \prod_{k=1}^{n} d_{kk}$$

   This is the only product term that is non-zero in equation (3.1).

2. A little thought should convince you that it is the same for lower or upper triangular matrices as well, viz.

$$\boldsymbol{det(L)} = \prod_{k=1}^{n} l_{kk}$$

3. It should also be clear that if all the elements of any row or column are zero, then the determinant is zero.

4. If every element of any row or column of a matrix is multiplied by a scalar, it is equivalent to multiplying the determinant of the original matrix by the same scalar, *i.e.*,

$$\begin{vmatrix} ka_{11} & ka_{12} & \cdots & ka_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix} = \begin{vmatrix} a_{11} & a_{12} & \cdots & ka_{1n} \\ a_{21} & a_{22} & \cdots & ka_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & ka_{nn} \end{vmatrix} = k\,det(A)$$

5. Replacing any row (or column) of a matrix with a linear combination of that row (or column) and another row (or column) leaves the determinant unchanged.

6. A consequence of rules 3 and 5 is that if two rows (or columns) of a matrix are indentical the determinant is zero.

7. If any two rows (or columns) are interchanged, it results in a sign change of the determinant.

3.3.1 Laplace expansion of the determinant

A definition of determinant that you might have seen in an earlier linear algebra course is

$$\boxed{det(A) = |A| = \begin{cases} \sum_{k=1}^{n} a_{ik}A_{ik} & \text{for any } i \\ \sum_{k=1}^{n} a_{kj}A_{kj} & \text{for any } j \end{cases}} \tag{3.2}$$

where A_{ik}, called the *cofactor*, is given by,

$$A_{ik} = (-1)^{i+k}M_{ik}$$

and M_{ik}, called the *minor*, is the determinant of $(n-1)\times(n-1)$ submatrix of A obtained by deleting ith row and kth column of A. Note that the expansion in equation (3.2) can be carried out along any row i or column j of the original matrix A.

Example

Consider the matrix derived in Chapter 1 for the recycle example, viz. equation (1.8). Let us calculate the determinant of the matrix using the

Laplace expansion algorithm around the first row.

$$\boldsymbol{det}(A) = \begin{vmatrix} 1 & 0 & 0.306 \\ 0 & 1 & 0.702 \\ -2 & 1 & 0 \end{vmatrix}$$

$$= 1 \begin{vmatrix} 1 & 0.702 \\ 1 & 0 \end{vmatrix} + (-1)^{1+2} \times 0 \times \begin{vmatrix} 0 & 0.702 \\ -2 & 0 \end{vmatrix} + (-1)^{1+3} \times 0.306 \times \begin{vmatrix} 0 & 1 \\ -2 & 1 \end{vmatrix}$$

$$= 1 \times (-0.702) + 0 + 0.306 \times 2 = -0.09$$

A MATLAB implementation of this will be done as follows:

```
»A=[1 0 0.306; 0 1 0.702; -2 1 0] % Define matrix A
»det(A)                           % calculate the determinant
```

3.4 Solving a system of linear equations

3.4.1 Cramers rule

Consider a 2×2 system of equations,

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Direct elimination of the variable x_2 results in

$$(a_{11}a_{22} - a_{12}a_{21})\, x_1 = a_{22}b_1 - a_{12}b_2$$

which can be written in an laternate form as,

$$\boldsymbol{det}(A)\, x_1 = \boldsymbol{det}(A(1))$$

where the matrix A(1) is obtained from A after replacing the first column with the vector b. *i.e.*,

$$A(1) = \begin{vmatrix} b_1 & a_{12} \\ b_2 & a_{22} \end{vmatrix}$$

This generalizees to $n \times n$ system as follows,

$$x_1 = \frac{\boldsymbol{det}(A(1))}{\boldsymbol{det}(A)}, \quad \cdots \quad x_k = \frac{\boldsymbol{det}(A(k))}{\boldsymbol{det}(A)}, \quad \cdots \quad x_n = \frac{\boldsymbol{det}(A(n))}{\boldsymbol{det}(A)}.$$

where A(k) is an $n \times n$ matrix obtained from A by replacing the kth column with the vector b. It should be clear from the above that, in order to have a unique solution, the determinant of A should be *non-zero*. If the determinant is zero, then such matrices are called *singular*.

Example

Continuing with the recycle problem (equation (1.8) of Chapter 1), solution using Cramer's rule can be implemented with MATLAB as follows:

$$A\,x = b \quad \Rightarrow \quad \begin{bmatrix} 1 & 0 & 0.306 \\ 0 & 1 & 0.702 \\ -2 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 101.48 \\ 225.78 \\ 0 \end{bmatrix}$$

```
»A=[1 0 0.306; 0 1 0.702; -2 1 0]; % Define matrix A
»b=[101.48 225.78 0]'             % Define right hand side vector b
»A1=[b, A(:,[2 3])]               % Define A(1)
»A2=[A(:,1),b, A(:, 3)]           % Define A(2)
»A3=[A(:,[1 2]), b ]              % Define A(3)
»x(1) = det(A1)/det(A)            % solve for coponent x(1)
»x(2) = det(A2)/det(A)            % solve for coponent x(2)
»x(3) = det(A3)/det(A)            % solve for coponent x(3)
»norm(A*x'-b)                     % Check residual
```

3.4.2 Matrix inverse

We defined the inverse of a matrix A as that matrix B which, when multiplied by A produces the identity matrix - *i.e.*, $AB = I$; but we did not develop a scheme for finding B. We can do so now by combining Cramer's rule and Laplace expansion for a determinant as follows. Using Laplace expansion of the determinant of A(k) around column k,

$$det A(k) = b_1 A_{1k} + b_2 A_{2k} + \cdots + b_n A_{nk} \qquad k = 1, 2, \cdots, n$$

where A_{ik} are the cofactors of A. The components of the solution vector, x are,

$$x_1 = (b_1 A_{11} + b_2 A_{21} + \cdots + b_n A_{n1})/det(A)$$
$$x_j = (b_1 A_{1j} + b_2 A_{2j} + \cdots + b_n A_{nj})/det(A)$$
$$x_n = (b_1 A_{1n} + b_2 A_{2n} + \cdots + b_n A_{nn})/det(A)$$

The right hand side of this system of equations can be written as a vector matrix product as follows,

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

or

$$\boldsymbol{x} = \boldsymbol{B}\,\boldsymbol{b}$$

Premultiplying the original equation $\boldsymbol{A}\,\boldsymbol{x} = \boldsymbol{b}$ by $\boldsymbol{A}^{-1}$ we get

$$\boldsymbol{A}^{-1}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b} \quad \text{or} \quad \boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$$

Comparing the last two equations, it is clear that,

$$\boldsymbol{B} = \boldsymbol{A}^{-1} = \frac{\mathbf{1}}{\boldsymbol{det}(\boldsymbol{A})} \begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{21} & \cdots & \boldsymbol{A}_{n1} \\ \boldsymbol{A}_{12} & \boldsymbol{A}_{22} & \cdots & \boldsymbol{A}_{n2} \\ \vdots & & \ddots & \vdots \\ \boldsymbol{A}_{1n} & \boldsymbol{A}_{2n} & \cdots & \boldsymbol{A}_{nn} \end{bmatrix} = \frac{\boldsymbol{adj}(\boldsymbol{A})}{\boldsymbol{det}(\boldsymbol{A})}$$

The above equation can be thought of as the definition for the *adjoint* of a matrix. It is obtained by simply replacing each element with its cofactor and then transposing the resulting matrix.

Inverse of a diagonal matrix

Inverse of a diagonal matrix, $\boldsymbol{D}$,

$$\boldsymbol{D} = \begin{bmatrix} \boldsymbol{d}_{11} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \boldsymbol{d}_{22} & \cdots & \mathbf{0} \\ \vdots & \mathbf{0} & \ddots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \cdots & \boldsymbol{d}_{nn} \end{bmatrix}$$

is given by,

$$\boldsymbol{D}^{-1} = \begin{bmatrix} \frac{1}{d_{11}} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \frac{1}{d_{22}} & \cdots & \mathbf{0} \\ \vdots & \mathbf{0} & \ddots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \cdots & \frac{1}{d_{nn}} \end{bmatrix}$$

It is quite easy to verify using the definition of matrix multiplication that $\boldsymbol{D}\boldsymbol{D}^{-1} = \boldsymbol{I}$.

Inverse of a triangular matrix

Inverse of a triangular matrix is also triangular. Suppose $\boldsymbol{U}$ is a given upper triangular matrix, then the elements of $\boldsymbol{V} = \boldsymbol{U}^{-1}$, can be found sequentially in an efficient manner by simply using the definition $\boldsymbol{U}\boldsymbol{V} =$

I. This equation, in expanded form, is

$$\begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ 0 & 0 & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix} \begin{bmatrix} v_{11} & v_{12} & \cdots & v_{1n} \\ v_{21} & v_{22} & \cdots & v_{2n} \\ & & \ddots & \vdots \\ v_{n1} & v_{n2} & \cdots & v_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ & & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

We can develop the algorithm (*i.e.*, find out the rules) by simply carrying out the matrix multiplication on the left hand side and equating it element-by-element to the right hand side. First let us convince ourself that V is also upper triangular, *i.e.*,

$$\boxed{v_{ij} = 0 \qquad i > j} \tag{3.3}$$

Consider the element $(n, 1)$ which is obtained by summing the product of each element of n-th row of U (consisting mostly of zeros!) with the corresponding element of the 1-st column of V. The only non-zero term in this product is

$$u_{nn} v_{n1} = 0$$

Since $u_{nn} \neq 0$ it is clear that $v_{n1} = 0$. Carrying out similar arguments in a sequential manner (in the order $\{i = n \cdots j - 1, j = 1 \cdots n\}$ - *i.e.*, decreasing order of i and increasing order of j) it is easy to verify equation (3.3) and thus establish that V is also upper triangular.

The non-zero elements of V can also be found in a sequential manner as follows. For each of the diagonal elements (i, i) summing the product of each element of i-th row of U with the corresponding element of the i-th column of V, the only non-zero term is,

$$\boxed{v_{ii} = \frac{1}{u_{ii}} \qquad i = 1, \cdots, n} \tag{3.4}$$

Next, for each of the upper elements (i, j) summing the product of each element of i-th row of U with the corresponding element of the j-th column of V, we get,

$$u_{ii} v_{ij} + \sum_{r=i+1}^{j} u_{ir} v_{rj} = 0$$

and hence we get,

$$\boxed{v_{ij} = -\frac{1}{u_{ii}} \sum_{r=i+1}^{j} u_{ir} v_{rj} \qquad j = 2, \cdots, n;\ j > i;\ i = j - 1, 1} \tag{3.5}$$

Note that equation (3.5) should be applied in a specific order, as otherwise, it may involve unknown elements v_{rj} on the right hand side. First, all of the diagonal elements of V (viz. v_{ii}) must be calcualted from equation (3.4) as they are needed on the right hand side of equation (3.5). Next the order indicated in equation (3.5), viz. increasing j from 2 to n and for each j decreasing i from $(j-1)$ to 1, sould be obeyed to avoid having unknowns values appearing on the right hand side of (3.5).

A MATLAB implementation of this algorithm is shown in figure 3.1 to illustrate precisely the order of the calculations. Note that the built-in, general purpose MATLAB inverse function (viz. `inv(U)`) does not take into account the special structure of a triangular matrix and hence is computationally more expensive than the `invu` funcion of figure 3.1. This is illustrated with the following example.

Example

Consider the upper triangular matrix,

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

Let us find its inverse using both the built-in MATLAB function `inv(U)` and the function `invu(U)` of figure 3.1 that is applicable sepcifically for an upper triangular matrix. You can also compare the floating point opeartion count for each of the algorithm. Work through the following example using MATLAB. Make sure that the function `invu` of figure 3.1 is in the search path of MATLAB.

```
» U=[1 2 3 4; 0 2 3 1;0 0  1 2; 0 0 0 4]

U =

     1     2     3     4
     0     2     3     1
     0     0     1     2
     0     0     0     4

» flops(0)  %initialize the flop count
» V=inv(U)
```

```
V =

    1.0000   -1.0000         0   -0.7500
         0    0.5000   -1.5000    0.6250
         0         0    1.0000   -0.5000
         0         0         0    0.2500

» flops   %print flop count

ans =

   208
» flops(0);ch3_invu(U),flops  %initialize flop count, then invert

ans =

    1.0000   -1.0000         0   -0.7500
         0    0.5000   -1.5000    0.6250
         0         0    1.0000   -0.5000
         0         0         0    0.2500


ans =

    57
```

3.4.3 Gaussian elimination

Gaussian elimination is one of the most efficient algorithms for solving a large system of linear algebraic equations. It is based on a systematic generalization of a rather intuitive elimination process that we routinely apply to a small, say, (2×2) systems. *e.g.*,

$$10x_1 + 2x_2 = 4$$

$$x_1 + 4x_2 = 3$$

From the first equation we have $x_1 = (4 - 2x_2)/10$ which is used to *eliminate* the variable x_1 from the second equation, viz. $(4 - 2x_2)/10 + 4x_2 = 3$ which is solved to get $x_2 = 0.6842$. In the second phase, the

```
function v=invu(u)
% Invert upper triangular matrix
%   u - (nxn) matrix
%   v - inv(a)

n=size(u,1);   % get number of rows in matrix a

for i=2:n
 for j=1:i-1
   v(i,j)=0;
 end
end

for i=1:n
 v(i,i)=1/u(i,i);
end

for j=2:n
 for i=j-1:-1:1
   v(i,j) = -1/u(i,i)*sum(u(i,i+1:j)*v(i+1:j,j));
 end
end
```

Figure 3.1: MATLAB implementation of inverse of an upper triangular matrix

value of x_2 is *back substituted* into the first equation and we get $x_1 = 0.2632$. We could have reversed the order and eliminated x_1 from the first equation after rearranging the second equation as $x_1 = (3 - 4x_2)$. Thus there are two phases to the algorithm: (a) forward elimination of one variable at a time until the last equation contains only one unknown; (b) back substitution of variables. Also, note that we have used two rules during the elimination process: (i) two equations (or two rows) can be interchanged as it is merely a matter of book keeping and it does not in any way alter the problem formulation, (ii) we can replace any equation with a linear combination of itself with another equation. A conceptual description of a naive Gaussian elimination algorithm is shown in figure 3.2. All of the arithmetic operations needed to eliminate one variable at a time are identified in the illustration. Study that carefully.

We call it a naive scheme as we have assumed that none of the diagonal elements are zero, although this is not a requirement for existence of a solution. The reason for avoiding zeros on the diagoanls is to avoid division by zeros in step 2 of the illustration 3.2. If there are zeros on the diagonal, we can interchange two equations in such a way the diagonals do not contain zeros. This process is called *pivoting*. Even if we organize the equations in such a way that there are no zeros on the diagonal, we may end up with a zero on the diagonal during the elimination process (likely to occur in step 3 of illustration 3.2). If that situation arises, we can continue to exchange that particular row with another one to avoid division by zero. If the matrix is *singular* we wil eventually end up with an unavoidable zero on the diagonal. This situation will arise if the original set of equations is not *linearly independent*; in other words the *rank* of the matrix A is less than n. Due to the finite precision of computers, the floating point operation in step 3 of illustration 3.2 will not result usually in an exact zero, but rather a very small number. Loss of precision due to round off errors is a common problem with direct methods involving large system of equations since any error introduced at one step corrupts all subsequent calcualtions.

A MATLAB implementation of the algorithm is given in figure 3.3 through the function `gauss.m`.

Note that it is merely for illustrating the concepts involved in the elimination process; MATLAB `backslash, \` operator provides a much more elegant solution to solve $Ax = b$ in the form $x = A\backslash b$.

Example

Let us continue with the recycle problem (equation (1.8) of Chapter 1). First we obtain solution using the built-in MATLAB linear equation solver (viz. $x = A\backslash b$ and record the floating point operations (flops). Then we solve with the Gaussian elimination function `gauss` and compare the flops. Note that in order to use the naive Gaussian elimination function,

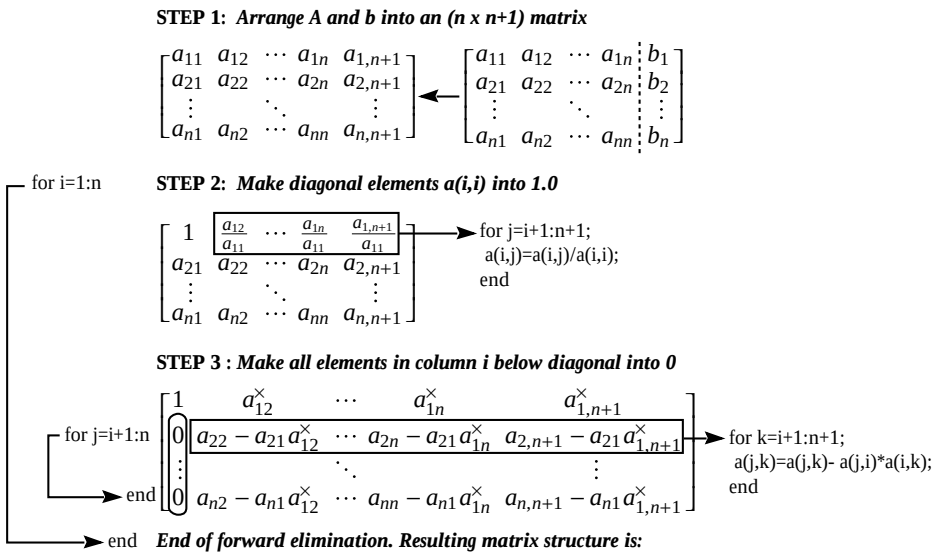

Figure 3.2: Naive Gaussian elimination shceme

```
function x=gauss(a,b)
% Naive Gaussian elimination. Cannot have zeros on diagonal
%   a - (nxn) matrix
%   b - column vector of length n

m=size(a,1);   % get number of rows in matrix a
n=length(b);   % get length of b
if (m ~= n)
  error('a and b do not have the same number of rows')
end

%Step 1: form (n,n+1) augmented matrix
a(:,n+1)=b;

for i=1:n
%Step 2: make diagonal elements into 1.0
a(i,i+1:n+1) = a(i,i+1:n+1)/a(i,i);

%Step 3: make all elements below diagonal into 0
 for j=i+1:n
   a(j,i+1:n+1) = a(j,i+1:n+1) - a(j,i)*a(i,i+1:n+1);
 end
end
%Step 4: begin back substitution
for j=n-1:-1:1
 a(j,n+1) = a(j,n+1) - a(j,j+1:n)*a(j+1:n,n+1);
end

%return solution
x=a(:,n+1)
```

Figure 3.3: MATLAB implementation of naive Gaussian elimination

we need to switch the 2nd and 3rd equations to avoid division by zero.

```
»A=[1 0 0.306;          % matrix entry continues on next two lines
»-2 1 0;
»0 1 0.702];            % Definition of matrix A complete
»b=[101.48 0 225.78]' ; % Define right hand side column vector b
»flops(0) ;             % initialize flop count
»A\b                    % solution is : [23.8920 47.7840 253.5556]
»flops                  % examine flops for MATLAB internal solver (ans: 71)
»flops(0) ;             % initialize flop count
»gauss(A,b)             % solution is, of course : [23.8920 47.7840 253.5556]
»flops                  % examine flops for gauss solver (ans: 75)
»flops(0) ;             % initialize flop count
»inv(A)*b               % obtain solution using matrix inverse
»flops                  % examine flops for MATLAB internal solver (ans: 102)
```

Example - *loss of accuracy & need for pivoting*

The need for pivoting can be illustrated with the following simple example.

$$\epsilon x_1 + x_2 = 1$$

$$x_1 + x_2 = 2$$

where ϵ is a small number. In matrix form it will be,

$$\begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

In using naive Gaussian elimination without rearranging the equations, we first make the diagonal into unity, which results in

$$x_1 + \frac{1}{\epsilon} x_2 = \frac{1}{\epsilon}$$

Next we eliminate the variable x_1 from the 2nd equation which resutls in,

$$\left(1 - \frac{1}{\epsilon}\right) x_2 = 2 - \frac{1}{\epsilon}$$

Rearranging this and using back substitution we finally get x_2 and x_1 as,

$$x_2 = \frac{2 - \frac{1}{\epsilon}}{1 - \frac{1}{\epsilon}}$$

ϵ	Naive elimination without pivoting gauss(A,b)	Built-in MATLAB $A\backslash b$
1×10^{-15}	[1 1]	[1, 1]
1×10^{-16}	[2 1]	[1, 1]
1×10^{-17}	[0 1]	[1, 1]

Table 3.1: Loss of precision and need for pivoting

$$x_1 = \frac{1}{\epsilon} - \frac{x_2}{\epsilon}$$

The problem in computing x_1 as $\epsilon \to 0$ should be clear now. As ϵ crosses the threshold of finite precision of the computation (hardware or software), taking the difference of two large numbers of comparable magnitude, can result in significant loss of precision. Let us solve the problem once again after rearranging the equations as,

$$x_1 + x_2 = 2$$

$$\epsilon x_1 + x_2 = 1$$

and apply Gaussian elimination once again. Since the diagonal element in the first equation is already unity, we can eliminate x_1 from the 2nd equation to obtain,

$$(1 - \epsilon)x_2 = 1 - 2\epsilon \qquad \text{or} \qquad x_2 = \frac{1 - 2\epsilon}{1 - \epsilon}$$

Back substitution yields,

$$x_1 = 2 - x_2$$

Both these computations are well behaved as $\epsilon \to 0$.

We can actually demonstrate this using the MATLAB function `gauss` shown in figure 3.3 and compare it with the MATLAB built-in function $A\backslash b$ which does use pivoting to rearrange the equations and minimize the loss of precision. The results are compared in table 3.1 for ϵ in the range of 10^{-15} to 10^{-17}. Since MATLAB uses double precision, this range of ϵ is the threshold for loss of precision. Observe that the naive Gaussian elimination produces incorrect results for $\epsilon < 10^{-16}$.

3.4.4 Thomas algorithm

Many problems such as the stagewise separation problem we saw in section §1.3.1 or the solution of differential equations that we will see in

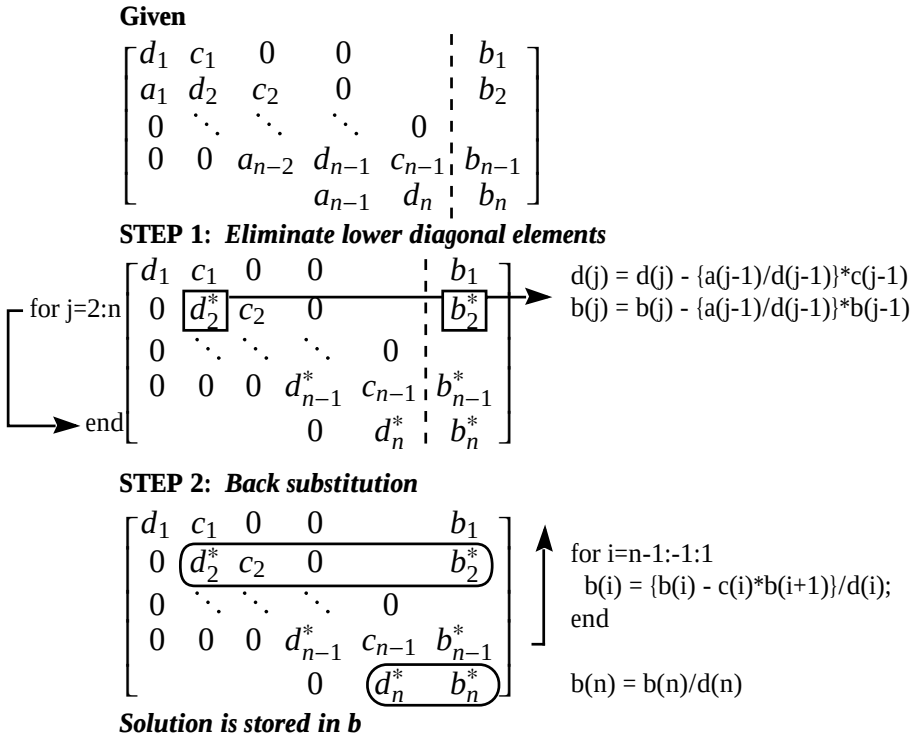

Figure 3.4: Thomas algorithm

later chapters involve solving a system of linear equations $\boldsymbol{Tx} = \boldsymbol{b}$ with a *tridiagonal* matrix structure.

$$\boldsymbol{T} = \begin{bmatrix} \boldsymbol{d}_1 & \boldsymbol{c}_1 & \mathbf{o} & \cdots & \mathbf{o} \\ \boldsymbol{a}_1 & \boldsymbol{d}_2 & \boldsymbol{c}_2 & \cdots & \mathbf{o} \\ & & \ddots & & \\ \mathbf{o} & \mathbf{o} & \boldsymbol{a}_{n-2} & \boldsymbol{d}_{n-1} & \boldsymbol{c}_{n-1} \\ \mathbf{o} & \cdots & \mathbf{o} & \boldsymbol{a}_{n-1} & \boldsymbol{d}_n \end{bmatrix} \quad \boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \vdots \\ \boldsymbol{x}_{n-1} \\ \boldsymbol{x}_n \end{bmatrix} \quad \boldsymbol{b} = \begin{bmatrix} \boldsymbol{b}_1 \\ \boldsymbol{b}_2 \\ \vdots \\ \boldsymbol{b}_{n-1} \\ \boldsymbol{b}_n \end{bmatrix}$$

Since we know where the zero elements are, we do not have to carry out the elimination steps on those entries of the matrix $\boldsymbol{T}$; but the essential steps in the algorithm remain the same as in the Gaussian elimination scheme and are illustrated in figure 3.4. MATLAB implementation is shown in figure 3.5.

```
function x=thomas(a,b,c,d)
% Thomas algorithm for tridiagonal systems
%   d - diagonal elements, n
%   b - right hand side forcing term,  n
%   a - lower diagonal elements, (n-1)
%   c - upper diagonal elements, (n-1)

na=length(a); % get length of a
nb=length(b); % get length of b
nc=length(c); % get length of c
nd=length(d); % get length of d
if (nd ~= nb | na ~= nc | (nd-1) ~= na)
  error('array dimensions not consistent')
end
n=length(d);
%Step 1: forward elimination

for i=2:n
  fctr=a(i-1)/d(i-1);
  d(i) = d(i) - fctr*c(i-1);
  b(i) = b(i) - fctr*b(i-1);
end

%Step 2: back substitution
b(n) = b(n)/d(n);
for j=n-1:-1:1
 b(j) = (b(j) - c(j)*b(j+1))/d(j);
end

%return solution
x=b;
```

Figure 3.5: MATLAB implementation of Thomas algorithm

3.4.5 Gaussian elimination - *Symbolic representaion*

Given a square matrix A of dimension $n \times n$ it is possible to write it is as the product of two matrices B and C, *i.e.*, $A = BC$. This process is called *factorization* and is in fact not at all unique - *i.e.*, there are inifnitely many possiblilities for B and C. This is clear with a simple counting of the unknowns - viz. there are $2 \times n^2$ unknown elements in B and C while only n^2 equations can be obtained by equating each element of A with the corresponding element from the product BC.

The extra degrees of freedom can be used to specify any specific structure for B and C. For example we can require $B = L$ be a lower triangular matrix and $C = U$ be an upper triangular matrix.This process is called LU factorization or decomposition. Since each triangular matrix has $n \times (n+1)/2$ unknowns, we still have a total of $n^2 + n$ unknowns. The extra n degrees of freedom is often used in one of three ways:

- *Doolitle* method assigns the diagonal elements of L to be unity.
- *Crout* method assigns the diagonal elements of U to be unity.
- *Cholesky* method assigns the diagonal elements of L to be equal to that of$U$ - *i.e.*, $l_{ii} = u_{ii}$.

While a simple degree of freedom analysis, indicates that it is possible to factorize a matrix into a product of lower and upper trinagular matrices, it does not tell us how to find out the unknown elements.

Revisiting the Gaussian elimination method from a different perspective, will show the connection between LU factorization and Gaussian elimination. Note that the algorithm outlined in section §3.4.3 is the most computationally efficient scheme for implimenting Guassian elimination. The method to be outlined below is not computationally efficient, but it is a useful conceptual aid in showing the connection between Guassian elimination and LU factorization. Steps 2 and 3 of figure 3.2 that involve making the diagonal into unity and all the elements below the diagonal into zero is equivalent to pre-multuplying A by L_1 - *i.e.*, $L_1 A = U_1$ or,

$$\begin{bmatrix} \frac{1}{a_{11}} & 0 & \cdots & 0 \\ -\frac{a_{21}}{a_{11}} & 1 & \cdots & 0 \\ \vdots & 0 & \ddots & 0 \\ -\frac{a_{n1}}{a_{11}} & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ & & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} 1 & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(1)} & \cdots & a_{2n}^{(1)} \\ & & \ddots & \vdots \\ 0 & a_{n2}^{(1)} & \cdots & a_{nn}^{(1)} \end{bmatrix}$$

Repeating this process for the 2nd row, we pre-multiply $\boldsymbol{U}_1$ by $\boldsymbol{L}_2$ - *i.e.*, $\boldsymbol{L}_2\boldsymbol{U}_1 = \boldsymbol{U}_2$ or, in expanded form,

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & \frac{1}{a_{22}^{(1)}} & 0 & \cdots & 0 \\ 0 & -\frac{a_{32}^{(1)}}{a_{22}^{(1)}} & 1 & \cdots & 0 \\ \vdots & \vdots & 0 & \ddots & 0 \\ 0 & -\frac{a_{n2}^{(1)}}{a_{22}^{(1)}} & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(1)} & \cdots & a_{2n}^{(1)} \\ & & \ddots & \vdots \\ 0 & a_{n2}^{(1)} & \cdots & a_{nn}^{(1)} \end{bmatrix} = \begin{bmatrix} 1 & a_{12}^{(1)} & a_{13}^{(1)} & \cdots & a_{1n}^{(1)} \\ 0 & 1 & a_{23}^{(2)} & \cdots & a_{2n}^{(2)} \\ 0 & 0 & a_{33}^{(2)} & \cdots & \vdots \\ 0 & 0 & \vdots & \cdots & \vdots \\ 0 & 0 & a_{n3}^{(2)} & \cdots & a_{nn}^{(2)} \end{bmatrix}$$

Continuing this process, we obtain in succession,

$$\boldsymbol{L}_1\boldsymbol{A} = \boldsymbol{U}_1$$

$$\boldsymbol{L}_2\boldsymbol{U}_1 = \boldsymbol{U}_2$$

$$\boldsymbol{L}_3\boldsymbol{U}_2 = \boldsymbol{U}_3$$

$$\boldsymbol{L}_4\boldsymbol{U}_3 = \boldsymbol{U}_4$$

$$\boldsymbol{L}_{n-1}\boldsymbol{U}_{n-2} = \boldsymbol{U}_{n-1}$$

Note that each $\boldsymbol{L}_j$ is a lower triangular matrix with non-zero elements on the $\boldsymbol{j}$-th column and unity on other diagonal elements. Eliminating all of the intermediate $\boldsymbol{U}_j$ we obtain,

$$(\boldsymbol{L}_{n-1}\boldsymbol{L}_{n-2}\cdots\boldsymbol{L}_1)\boldsymbol{A} = \boldsymbol{U}_{n-1}$$

Since the product of all lower triangular matrices is yet another lower triangular matrix, we can write the above equation as,

$$\boldsymbol{L}\boldsymbol{A} = \boldsymbol{U}$$

Also, the inverse of a lower triangular matrix is also lower triangular - *i.e.*, $\boldsymbol{Ł} = \boldsymbol{L}^{-1}$. Hence a given square matrix $\boldsymbol{A}$ can be factored into a product of a lower and upper triangular matrix as,

$$\boldsymbol{A} = \boldsymbol{L}^{-1}\boldsymbol{U} = \boldsymbol{Ł}\boldsymbol{U}$$

Although the development in this section provides us with an algorithm for constructing both $\boldsymbol{Ł}$ and $\boldsymbol{U}$, it is quite inefficient. A more direct and efficient algorithm is developed next in section §3.4.6.

3.4.6 LU decomposition

Consider the product of L and U as shown in the expanded form below. All of the elements of L and U are unkonwn. By carrying out the matrix product on the left hand side and equating element-by-element to the right hand side, we can develop sufficient number of equations to find out all of the unkown elements on the left hand side. The trick, however is, (as we did with inverting a triangular matrix) to carry out the calculations in a particualr sequence so that no more than one unknown appears in each equation.

$$\begin{bmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & \cdots & 0 \\ l_{31} & l_{32} & l_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ l_{n1} & l_{n2} & l_{n3} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} 1 & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & 1 & u_{23} & \cdots & u_{2n} \\ 0 & 0 & 1 & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ a_{31} & a_{32} & \cdots & a_{3n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

Let us first consider elements in column 1 of L. Caryying out the multiplication and equating we obtain,

$$\boxed{l_{i1} = a_{i1} \qquad i = 1, \cdots, n} \tag{3.6}$$

Next focusing on the elements in the first row of U we get,

$$\boxed{u_{1j} = a_{1j}/l_{11} \qquad j = 2, \cdots, n} \tag{3.7}$$

It would be inefficient to proceed to the 2nd column of L. Why?

Next we alternate between a column of L and a row of U. The general expression for any element i in column j of L is,

$$\boxed{l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \qquad j = 2, \cdots, n \quad i = j, \cdots, n} \tag{3.8}$$

Similarly the general expression for any element i in row j of U is,

$$\boxed{u_{ji} = \frac{\left[a_{ji} - \sum_{k=1}^{j-1} l_{jk} u_{ki}\right]}{l_{jj}} \qquad j = 2, \cdots, n \quad i = j+1, \cdots, n} \tag{3.9}$$

Equations (3.6-3.9) form the basic algorithm for LU decomposition. In order to illustrate the implementation of equations (3.6-3.9) as an algorithm, a MATLAB function called `LU.m` is shown in figure 3.6. Note that MATLAB provides a built-in function for LU decomposition called `lu(A)`.

```
function [L,U]=LU(a)
% Naive LU decomposition
%   a    - (nxn) matrix
%   L,U  - are (nxn) factored matrices
% Usage [L,U]=LU(A)

n=size(a,1);   % get number of rows in matrix a

%Step 1: first column of L
L(:,1)=a(:,1);

%Step 2: first row of U
U(1,:)=a(1,:)/L(1,1);

%Step 3: Alternate between column of L and row of U
for j=2:n
  for i = j:n
    L(i,j) = a(i,j) - sum(L(i,1:j-1)'.*U(1:j-1,j));
  end
  U(j,j) = 1;
  for i=j+1:n
    U(j,i)=(a(j,i) - sum(L(j,1:j-1)'.*U(1:j-1,i) ) )/L(j,j);
  end
end
```

Figure 3.6: MATLAB implementation of LU decomposition algorithm

Recognizing that $\boldsymbol{A}$ can be factored into the product $\boldsymbol{LU}$, one can implement an efficient scheme for solving a system of linear algebraic equations $\boldsymbol{Ax} = \boldsymbol{b}$ repeatedly, particularly when the matrix $\boldsymbol{A}$ remains unchanged, but different solutions are required for different forcing terms on the right hand side, $\boldsymbol{b}$. The equation

$$\boldsymbol{Ax} = \boldsymbol{b}$$

can be written as

$$\boldsymbol{LUx} = \boldsymbol{b} \qquad \Rightarrow \qquad \boldsymbol{Ux} = \boldsymbol{L}^{-1}\boldsymbol{b} = \boldsymbol{b}'$$

and hence

$$\boldsymbol{x} = \boldsymbol{U}^{-1}\boldsymbol{b}'$$

The operations required for forward elimination and back substitution are stored in the LU factored matrix and as we saw earlier it is relatively efficient to invert triangular matrices. Hence two additional vector-matrix products provide a solution for each new value of $\boldsymbol{b}$.

Example

Work through the following exercise in MATLAB to get a feel for the built-in MATLAB implementation of LU factorization with that given in figure 3.6. Before you work through the exercise make sure that the file `LU.m` that contains the function illustrated in figure 3.6 is in the MATLAB path. Also, be aware of the upper case function `LU` of figure 3.6 and the lower case `lu` which is the built-in function.

```
»A=[1 0 0.306; 0 1 0.702; -2 1 0]; % Define matrix A
»b=[101.48 225.78 0]'              % Define right hand vector b
»flops(0)                          % initialize flop count
»x=A\b                             % solve using built-in solver
»flops                             % flops = 74
»flops(0)                          % re-initialize flop count
»[l,u]=LU(A)                       %Use algorithm in figure 3.6
»flops                             % flops = 24
»x=inv(u)*(inv(l)*b)               % Solve linear system
»flops                             % Cumulative flops = 213
»flops(0)                          % re-initialize flop count
»[L,U]=lu(A)                       %use built-in function
»flops                             % flops = 9
»x=inv(U)*(inv(L)*b)               % Solve linear system
»flops                             % Cumulative flops = 183
```

3.5 Iterative algorithms for systems of linear equations

The direct methods discussed in section §3.4 have the advantage of producing the solution in a finite number of calculations. They suffer, however, from loss of precision due to accumulated round off errors. This problem is particulalry sever in large dimensional systems (more than 10,000 equations). Iterative methods, on the other hand, produce the result in an asymptotic manner by repeated application of a simple algorithm. Hence the number of floating point operations required to produce the final result cannot be known *a priori*. But they have the natural ability to eliminate errors at every step of the iteration. For an authoritative account of iterative methods for large linear systems see Young (1971).

Iterative methods rely on the concepts developed in Chapter 2. They are extended naturally from a single equation (one-dimensional system) to a system of equations (n-dimensional system). The development parallels that of section §2.7 on fixed point iterations schemes. Given an equation of the form, $A\,x = b$ we can rearrange it into a form,

$$\boxed{x^{(p+1)} = G(x^{(p)}) \qquad p = 0, 1, \cdots} \tag{3.10}$$

Here we can view the vector x as a point in a n-dimensional vector space and the above equation as an iterative map that maps a point $x^{(p)}$ into another point $x^{(p+1)}$ in the n-dimensional vector space. Starting with an initial guess $x^{(0)}$ we calculate successive iterates $x^{(1)}, x^{(2)} \cdots$ until the sequence converges. The only difference from chapter 2 is that the above iteration is applied to a higher dimensional system of (n) equations. Note that $G(x)$ is also vector. Since we are dealing with a linear system, G will be a linear function of x which is constructed from the given A matrix. G can typically be represented as

$$x^{(p+1)} = G(x^{(p)}) = Tx^{(p)} + c. \tag{3.11}$$

In section §2.7 we saw that a given equation $f(x) = 0$ can be rearranged into the form $x = g(x)$ in several different ways. In a similar manner, a given equation $Ax = b$ can be rearranged into the form $x^{(p+1)} = G(x^{(p)})$ in more than one way. Different choices of G results in different iterative methods. In section §2.7 we also saw that the condition for convergence of the seuqence $x_{i+1} = g(x_i)$ is $g'(r) < 1$. Recognizing that the derivative of $G(x^{(p)})$ with respect to $x^{(p)}$ is a matrix, $G' = T$ a convergence condition similar to that found for the scalar case must depend on the properties of the matrix T. Another way to demonstrate

this is as follows. Once the sequence $x^{(1)}, x^{(2)} \cdots$ converges to, say, r equation (3.11) becomes,

$$r = Tr + c.$$

Subtracting equation (3.11) from the above,

$$(x^{(p+1)} - r) = T(x^{(p)} - r).$$

Now, recognizing that $(x^{(p)} - r) = \epsilon^{(p)}$ is a measure of the error at iteration level p, we have

$$\epsilon^{(p+1)} = T\epsilon^{(p)}.$$

Thus, the error at step $(p+1)$ depend on the error at step (p). If the matrix T has the property of amplifying the error at any step, then the iterative sequence will diverge. The property of the matrix T that determines this feature is called the *spectral radius*. The *spectral radius* is defined as the largest eigenvalue in magnitude of T. For converence of the iterative sequence the *spectral radius* of T should be less than unity,

$$\boxed{\rho(T) < 1} \tag{3.12}$$

3.5.1 Jacobi iteration

The Jacobi iteration rearranges the given equations in the form,

$$\begin{aligned}
x_1^{(p+1)} &= (b_1 - a_{12}x_2^{(p)} - a_{13}x_3^{(p)} - \cdots - a_{1n}x_n^{(p)})/a_{11} \\
x_j^{(p+1)} &= \left[b_j - \sum_{k=1}^{j-1} a_{jk}x_k^{(p)} - \sum_{k=j+1}^{n} a_{jk}x_k^{(p)} \right] / a_{jj} \\
x_n^{(p+1)} &= (b_n - a_{n1}x_1^{(p)} - a_{n2}x_2^{(p)} - \cdots - a_{n,n-1}x_{n-1}^{(p)})/a_{nn}
\end{aligned} \tag{3.13}$$

where the variable x_j has been extracted form the $j-th$ equation and expressed as a function of the remaining variables. The above set of equations can be applied repetitively to update each component of the unknown vector x=$(x_1, x_2, \cdots, x_n)$ provided an inital guess is known for x. The above equation can be written in matrix form as,

$$Lx^{(p)} + Dx^{(p+1)} + Ux^{(p)} = b$$

where the matrices D, L, U are defined in term of components of A as follows.

$$D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & 0 & \ddots & 0 \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

Note that MATLAB functions
`diag`
`tril`
`triu`
are useful in extracting parts of a given matrix A

$$L = \begin{bmatrix} \mathbf{o} & \mathbf{o} & \cdots & \mathbf{o} \\ \boldsymbol{a}_{21} & \mathbf{o} & \cdots & \mathbf{o} \\ \vdots & & \ddots & \mathbf{o} \\ \boldsymbol{a}_{n1} & \boldsymbol{a}_{n2} & \cdots & \mathbf{o} \end{bmatrix} \quad U = \begin{bmatrix} \mathbf{o} & \boldsymbol{a}_{12} & \cdots & \boldsymbol{a}_{1n} \\ \mathbf{o} & \mathbf{o} & \cdots & \boldsymbol{a}_{2n} \\ \mathbf{o} & \mathbf{o} & \ddots & \vdots \\ \mathbf{o} & \mathbf{o} & \cdots & \mathbf{o} \end{bmatrix}$$

which can be rearranged as,

$$\boldsymbol{x}^{(p+1)} = \boldsymbol{D}^{-1}(\boldsymbol{b} - (\boldsymbol{L} + \boldsymbol{U})\boldsymbol{x}^{(p)}) \tag{3.14}$$

and hence $\boldsymbol{G}(\boldsymbol{x}^{(p)}) = -\boldsymbol{D}^{-1}(\boldsymbol{L}+\boldsymbol{U})\boldsymbol{x}^{(p)} + \boldsymbol{D}^{-1}\boldsymbol{b}$ and $\boldsymbol{G}' = \boldsymbol{T} = -\boldsymbol{D}^{-1}(\boldsymbol{L}+\boldsymbol{U})$. This method has been shown to be convergent as long as the original matrix $\boldsymbol{A}$ is diagonally dominant, *i.e.*,

An examination of equation (3.13) reveals that none of the diagonal elements can be zero. If any is found to be zero, one can easily exchange the positions of any two equations to avoid this problem. Equation (3.13) is used in actual computational implementation, while the matrix form of the equation (3.14) is useful for conceptual description and convergence analysis. Note that each element in the equation set (3.13) can be updated independent of the others in any order because the right hand side of equation (3.13) is evaluated at the $\boldsymbol{p}$-th level of iteration. This method requires that $\boldsymbol{x}^{(p)}$ and $\boldsymbol{x}^{(p+1)}$ be stored as two separate vectors until all elements of $\boldsymbol{x}^{(p+1)}$ have been updated using equation (3.13). A minor variation of the algorithm which uses a new value of the element in $\boldsymbol{x}^{(p+1)}$ as soon as it is available is called the Gauss-Seidel method. It has the dual advantage of faster convergence than the Jacobi iteration as well as reduced storage requirement for only one array $\boldsymbol{x}$.

3.5.2 Gauss-Seidel iteration

In the Gauss-Seidel iteration we rearrange the given equations in the form,

$$\begin{aligned} \boldsymbol{x}_1^{(p+1)} &= (\boldsymbol{b}_1 - \boldsymbol{a}_{12}\boldsymbol{x}_2^{(p)} - \boldsymbol{a}_{13}\boldsymbol{x}_3^{(p)} - \cdots - \boldsymbol{a}_{1n}\boldsymbol{x}_n^{(p)})/\boldsymbol{a}_{11} \\ \boldsymbol{x}_j^{(p+1)} &= \left[\boldsymbol{b}_j - \sum_{k=1}^{j-1} \boldsymbol{a}_{jk}\boldsymbol{x}_k^{(p+1)} - \sum_{k=j+1}^{n} \boldsymbol{a}_{jk}\boldsymbol{x}_k^{(p)}\right]/\boldsymbol{a}_{jj} \\ \boldsymbol{x}_n^{(p+1)} &= (\boldsymbol{b}_n - \boldsymbol{a}_{n1}\boldsymbol{x}_1^{(p+1)} - \boldsymbol{a}_{n2}\boldsymbol{x}_2^{(p+1)} - \cdots - \boldsymbol{a}_{n,n-1}\boldsymbol{x}_{n-1}^{(p+1)})/\boldsymbol{a}_{nn} \end{aligned} \tag{3.15}$$

Observe that known values of the elements in $\boldsymbol{x}^{(p+1)}$ are used on the right hand side of the above equations (3.15) as soon as they are available within the same iteration. We have used the superscripts $\boldsymbol{p}$ and $\boldsymbol{(p+1)}$ explicitly in equation (3.15) to indicate where the newest values occur. In a computer program there is no need to assign separate arrays for

p and $(p+1)$ levels of iteration. Using just a single array for x will automatically propagate the newest values as soon as they are updated.

The above equation can be written symbolically in matrix form as,

$$Lx^{(p+1)} + Dx^{(p+1)} + Ux^{(p)} = b$$

where the matrices D, L, U are defined as before. Factoring $x^{(p+1)}$ we get,

$$x^{(p+1)} = (L+D)^{-1}(b - Ux^{(p)}) \tag{3.16}$$

and hence $G(x^{(p)}) = -(L+D)^{-1}Ux^{(p)} + (L+D)^{-1}b$ and $G' = T = -(L+D)^{-1}U$. Thus the convergence of this scheme depends on the spectral radius of the matrix, $T = -(L+D)^{-1}U$. This method has also been shown to be convergent as long as the original matrix A is diagonally dominant.

MATLAB implementation of the Gauss-Seidel algorithm is shown in figure 3.7.

3.5.3 Successive over-relaxation (SOR) scheme

The relaxation scheme can be thought of as a convergence acceleration scheme that can be applied to any of the basic iterative methods like Jacobi or Gauss-Seidel schemes. We introduce an extra parameter, ω often called the *relaxation parameter* and choose its value in such a way that we can either speed up convergence by using $\omega > 1$ (called *over-relaxation*) or in some difficult problems with poor initial guess we can attempt to enlarge the region of convergence using $\omega < 1$ (called *under-relaxation*). Let us illustrate the implementation with the Gauss-Seidel scheme. The basic Gauss-Seidel scheme is:

$$t := x_j^{(p+1)} = \left[b_j - \sum_{k=1}^{j-1} a_{jk}x_k^{(p+1)} - \sum_{k=j+1}^{n} a_{jk}x_k^{(p)} \right] / a_{jj} \tag{3.17}$$

Instead of accepting the value of $x_j^{(p+1)}$ computed from the above formula as the current value, we store it in a temporary variable t and form a better (or accelerated) estimate of $x_j^{(p+1)}$ from,

$$x_j^{(p+1)} = x_j^{(p)} + \omega\,[t - x_j^{(p)}]$$

Observe that if $\omega = 1$, the method remains the same as Gauss-Seidel scheme. For $\omega > 1$, then the difference between two successive iterates (the term in the square bracketts) is amplified and added to the current value $x_j^{(p)}$.

```
function x=GS(a,b,x,tol,max)
% Gauss-Seidel iteration
%   a  - (nxn) matrix
%   b  - column vector of length n
%   x  - initial guess vector x
%  tol - convergence tolerance
%  max - maximum number of iterations
%        Usage x=GS(A,b,x)

m=size(a,1);   % get number of rows in matrix a
n=length(b);   % get length of b
if (m ~= n)
  error('a and b do not have the same number of rows')
end
if nargin < 5, max=100; end
if nargin < 4, max=100; tol=eps; end
if nargin == 2
  error('Initial guess is required')
end
count=0;

while  (norm(a*x-b)  > tol & count < max),
  x(1) = ( b(1) - a(1,2:n)*x(2:n) )/a(1,1);
 for i=2:n-1
   x(i) = (b(i) - a(i,1:i-1)*x(1:i-1) - ...
                  a(i,i+1:n)*x(i+1:n) )/a(i,i);
 end
  x(n) = ( b(n) - a(n,1:n-1)*x(1:n-1) )/a(n,n);
  count=count+1;
end

if (count >= max)
 fprintf(1,'Maximum iteration %3i exceeded\n',count)
 fprintf(1,'Residual is %12.5e\n ',norm(a*x-b) )
end
```

Figure 3.7: MATLAB implementation of Gauss-Seidel algorithm

The above opeartions can be written in symbollic matrix form as,

$$\boldsymbol{x}^{(p+1)} = \boldsymbol{x}^{(p)} + \omega[\{\boldsymbol{D}^{-1}(\boldsymbol{b} - \boldsymbol{L}\boldsymbol{x}^{(p+1)} - \boldsymbol{U}\boldsymbol{x}^{(p)})\} - \boldsymbol{x}^{(p)}]$$

where the term in braces represent the Gauss-Seidel scheme. After extracting $\boldsymbol{x}^{(p+1)}$ from the above equation, it can be cast in the standard iterative from of equation (3.11) as,

$$\boldsymbol{x}^{(p+1)} = (\boldsymbol{D} + \omega\boldsymbol{L})^{-1}[(1-\omega)\boldsymbol{D} - \omega\boldsymbol{U}]\boldsymbol{x}^{(p)} + \omega(\boldsymbol{D} + \omega\boldsymbol{L})^{-1}\boldsymbol{b} \quad (3.18)$$

Thus the convergence of the relaxation method depends on the spectral radius of the matrix $\boldsymbol{T}(\omega) := (\boldsymbol{D} + \omega\boldsymbol{L})^{-1}[(1-\omega)\boldsymbol{D} - \omega\boldsymbol{U}]$. Since this matrix is a function of ω we have gained a measure of control over the convergence of the iterative scheme. It has been shown (Young, 1971) that the SOR method is convergent for $0 < \omega < 2$ and that there is an optimum value of ω which results in the maximum rate of convergence. The optimum value of ω is very problem dependent and often difficult to determine precisely. For linear problems, typical values in the range of $\omega \approx 1.7 \sim 1.8$ are used.

3.5.4 Iterative refinement of direct solutions

We have seen that solutions obtained with direct methods are prone to accumulation of round-off errors, while iterative methods have the natural ability to remove such errors. In an attempt to combine the best of both worlds, one might construct an algorithm that takes the error-prone solution from a direct method as an initial guess to an iterative method and thus improve the accuracy of the solution.

Let us illustrate this concept as applied to improving the accuracy of a matrix inverse. Suppose $\boldsymbol{B}$ is an approximate (error-prone) inverse of a given matrix $\boldsymbol{A}$ obtained by one of the direct methods outlined in section §3.4. If $\boldsymbol{B}_\epsilon$ is the error in $\boldsymbol{B}$ then

$$\boldsymbol{A}(\boldsymbol{B} + \boldsymbol{B}_\epsilon) = \boldsymbol{I} \quad \text{or} \quad \boldsymbol{A}\boldsymbol{B}_\epsilon = (\boldsymbol{I} - \boldsymbol{A}\boldsymbol{B})$$

We do not, of course, know $\boldsymbol{B}_\epsilon$ and our task is to attempt to estimate it approximately. Premultiplying above equation by $\boldsymbol{B}$, and recognizing $\boldsymbol{B}\boldsymbol{A} \approx \boldsymbol{I}$, we have

$$\boldsymbol{B}_\epsilon = \boldsymbol{B}(\boldsymbol{I} - \boldsymbol{A}\boldsymbol{B})$$

Observe carefully that we have used the approximation $\boldsymbol{B}\boldsymbol{A} \approx \boldsymbol{I}$ on the left hand side where products of order unity are involved and *not* on the right hand side where difference between numbers of order unity are

involved. Now we have an estimate of the error B_ϵ which can be added to the approximate result B to obtain,

$$B + B_\epsilon = B + B(I - AB) = B(2I - AB)$$

Hence the iterative sequence should be,

$$\boxed{B^{(p+1)} = B^{(p)}(2I - AB^{(p)}) \qquad p = 0, 1 \cdots} \tag{3.19}$$

3.6 Gram-Schmidt orthogonalization procedure

Given a set of n linearly independent vectors, $\{x_i \mid i = 1, \cdots, n\}$ that are not necessarily orthonormal, it is possible to produce an orthonormal set of vectors $\{u_i \mid i = 1, \cdots, n\}$.

We begin by normalizing the x_1 vector using the norm $||x_1|| = \sqrt{x_1 \cdot x_1}$ and call it u_1.

$$u_1 = \frac{x_1}{||x_1||}$$

Subsequently we construct other vectors orthogonal to u_1 and normalize each one. For example we construct u_2' by subtracting u_1 from x_2 in such a way that u_2' contains no components of u_1 - *i.e.*,

$$u_2' = x_2 - c_0\, u_1$$

In the above c_0 is to be found in such a way that u_2' is orthogonal to u_1.

$$u_1^T \cdot u_2' = 0 = u_1^T \cdot x_2 - c_0 \quad \text{or} \quad c_0 = u_1^T \cdot x_2$$

Similarly we have,

$$u_3' = x_3 - c_1\, u_1 - c_2\, u_2$$

Requiring orthogonality with respect to both u_1 and u_2

$$u_1^T \cdot u_3' = 0 = u_1^T \cdot x_3 - c_1 \quad \text{or} \quad c_1 = u_1^T \cdot x_3$$
$$u_2^T \cdot u_3' = 0 = u_2^T \cdot x_3 - c_2 \quad \text{or} \quad c_2 = u_2^T \cdot x_3$$
$$u_3' = x_3 - (u_1^T \cdot x_3)\, u_1 - (u_2^T \cdot x_3)\, u_2$$

In general we have,

$$\boxed{u_s' = x_s - \sum_{j=1}^{s-1} (u_j^T \cdot x_s)\, u_j \qquad u_s = \frac{u_s'}{||u_s'||} \qquad s = 2, \cdots n;} \tag{3.20}$$

The Gram-Schmidt algorithm is illustrated in figure 3.8. Note that MATLAB has a built-in function `Q=orth(A)` which produces an orthonormal set from A. Q spans the same space as A and the number of columns in Q is the rank of A.

```
function u=GS_orth(x)
%Gram-Schmidt procedure applied on input x
%   x is a matrix of nxn.
%   Each x(:,j) represents a vector
n=length(x);
if rank(x) ~= n
 error('vectors in x are not linearly independent')
end
u(:,1) = x(:,1)/norm(x(:,1));
for s=2:n
   sum = zeros(n,1);
   for j=1:s-1
      sum = sum + (u(:,j)'*x(:,s)) * u(:,j);
   end
   uprime = x(:,s) - sum;
   u(:,s) = uprime/norm(uprime);
end
```

Figure 3.8: Illustration of Gram-Schmidt algorithm

3.7 The eigenvalue problem

A square matrix, $\boldsymbol{A}$, when operated on certain vectors, called eigenvectors, $\boldsymbol{x}$, leaves the vector unchanged excpet for a scaling factor, $\boldsymbol{\lambda}$. This fact can be represented as,

$$\boldsymbol{Ax} = \boldsymbol{\lambda x} \tag{3.21}$$

The problem of finding such eignvectors and eigenvalues is addressed by rewritting equation (3.21) as,

$$(\boldsymbol{A} - \boldsymbol{\lambda I})\ \boldsymbol{x} = \mathbf{0}$$

which represents a set of homogeneous equations that admit non-trivial solutions only if

$$\boldsymbol{det}(\boldsymbol{A} - \boldsymbol{\lambda I}) = \mathbf{0}.$$

i.e., only certain values of $\boldsymbol{\lambda}$ will make the above determinant zero. This requirement produces an n-th degree polynomial in $\boldsymbol{\lambda}$, called the *characteristic polynomial*. Fundamental results from algebra tell us that this polynomial will have exactly $\boldsymbol{n}$ roots, $\{\boldsymbol{\lambda}_j | \boldsymbol{j} = \mathbf{1} \cdots \boldsymbol{n}\}$ and corresponding to each root, $\boldsymbol{\lambda}_j$ we can determine an eigenvector, $\boldsymbol{x}_j$ from

$$(\boldsymbol{A} - \boldsymbol{\lambda}_j \boldsymbol{I})\ \boldsymbol{x}_j = \mathbf{0}.$$

Note that if, $\boldsymbol{x}_j$ satisfies the above equation, then $\boldsymbol{ax}_j$ will also satsify the same equation. Viewed alternately, since the $\boldsymbol{det}(\boldsymbol{A} - \boldsymbol{\lambda}_j \boldsymbol{I})$ is zero, $\boldsymbol{x}_j$ can be determined only up to an unknown constant - *i.e.*, only the direction of the eignvector can be determined, its magnitude being arbitrary.

The MATLAB built-in function `[V,D]=eig(A)` computes all of the eigenvalues and the associated eigenvectors.

Example

Determine the eigenvalues and eigenvectors of the matrix $\boldsymbol{A}$ defined below.

$$\boldsymbol{A} = \begin{bmatrix} \mathbf{2} & \mathbf{4} & \mathbf{3} \\ \mathbf{4} & \mathbf{4} & \mathbf{3} \\ \mathbf{3} & \mathbf{3} & \mathbf{5} \end{bmatrix}$$

A typical MATLAB session follows.

```
» A=[2 4 3;4  4 3;3 3 5]  %Define the matrix

A =

     2     4     3
     4     4     3
     3     3     5
%Compute the eigenvalues (in D) and the eigenvectors (in V)
» [V,D]=eig(A)

V =

    0.8197   -0.2674    0.5066
   -0.5587   -0.5685    0.6039
   -0.1265    0.7780    0.6154

D =

   -1.1894         0         0
         0    1.7764         0
         0         0   10.4130
%compute the coefficients of the characteric polynomial (in c)
» c=poly(A)

c =

    1.0000  -11.0000    4.0000   22.0000

%compute the eigenvalues (in r) and compare against the marix D
» r=roots(c)

r =

   10.4130
    1.7764
   -1.1894
```

Note that in the above example, the coefficiets of the characteristic polynomial, contained in the vector c, are defined as follows.

$$p_n(\lambda) = c_1\lambda^n + c_2\lambda^{n-1} + \cdots + c_{n+1}$$

3.7.1 Left and right eigenvectors

The eigenvector $\boldsymbol{x}$, defined in equation (3.21), is called the right eigenvector, since the matrix product operation is performed from the right. Consider the operation with a vector $\boldsymbol{y}$

$$\boldsymbol{y}'\boldsymbol{A} = \boldsymbol{y}'\boldsymbol{\lambda} \tag{3.22}$$

Since equation (3.22) can be written as

$$\boldsymbol{y}'(\boldsymbol{A} - \boldsymbol{\lambda}\boldsymbol{I}) = \mathbf{o}$$

the criterion for admitting nontrivial solutions is still the same, viz.

$$\boldsymbol{det}(\boldsymbol{A} - \boldsymbol{\lambda}\boldsymbol{I}) = \mathbf{o}.$$

Thus the eigenvalues are the same as those defined for the right eigenvector. If $\boldsymbol{A}$ is symmetric, then taking the transpose of equation (3.22) leads to equation (3.21) pointing out that the distinction between the left and right eigenvectors disappear. However for a nonsymmetric matrix, $\boldsymbol{A}$ there is a set of left $\{\boldsymbol{x}_i | \boldsymbol{i} = \mathbf{1} \cdots \boldsymbol{n}\}$ and right $\{\boldsymbol{y}_j | \boldsymbol{j} = \mathbf{1} \cdots \boldsymbol{n}\}$ eigenvectors that are distinct and form a *bi-orthogonal* set as shown below.

3.7.2 Bi-orthogonality

As long as the eigenvalues are distinct, the left and right eigenvectors form a bi-orthogonal set. To demonstrate this, take the left product of equation (3.21) with $\boldsymbol{y}'_j$ and the right product of equation (3.22) with $\boldsymbol{x}_i$. These lead respectively to

$$\boldsymbol{y}'_j \boldsymbol{A} \boldsymbol{x}_i = \boldsymbol{\lambda}_i \boldsymbol{y}'_j \boldsymbol{x}_i$$

$$\boldsymbol{y}'_j \boldsymbol{A} \boldsymbol{x}_i = \boldsymbol{\lambda}_j \boldsymbol{y}'_j \boldsymbol{x}_i$$

Subtracting one from the other, we get,

$$\mathbf{o} = (\boldsymbol{\lambda}_i - \boldsymbol{\lambda}_j) \boldsymbol{y}'_j \boldsymbol{x}_i.$$

Since $\boldsymbol{\lambda}_i$ and $\boldsymbol{\lambda}_j$ are distinct, we get

$$\boxed{\boldsymbol{y}'_j \boldsymbol{x}_i = \mathbf{o} \qquad \forall \boldsymbol{i} \neq \boldsymbol{j}}$$

which is the condition for bi-orthogonality.

3.7.3 Power iteration

A simple algorithm to find the largest eigenvalue and its associated eigenvector of a matrix A is the power iteration scheme. Conceptually the algorithm works as follows. Starting with any arbitrary vector, $z^{(0)}$, produce a sequence of vectors from

$$y^{(p+1)} = Az^{(p)}$$

$$y^{(p+1)} = k^{(p+1)}z^{(p+1)} \qquad p = 0, 1, \cdots$$

The second step above is merely a scaling operation to keep the vector length $z^{(p)}$ bounded. The vector $z^{(p)}$ converges to the eigenvector x_1 corresponding to the largest eigenvalue as $p \to \infty$.

The reason why it works can be understood, by begining with a spectral representation of the arbitrary vector, $z^{(0)}$ and tracing the effect of each operation, *i.e.*,

$$z^{(0)} = \sum_i \alpha_i x_i$$

$$y^{(1)} = Az^{(0)} = \sum_i \alpha_i A x_i = \sum_i \alpha_i \lambda_i x_i$$

$$y^{(2)} = Az^{(1)} = A\frac{y^{(1)}}{k^{(1)}} = \frac{1}{k^{(1)}} \sum_i \alpha_i \lambda_i A x_i = \frac{1}{k^{(1)}} \sum_i \alpha_i \lambda_i^2 x_i$$

Repeating this process we get,

$$y^{(p+1)} = Az^{(p)} = \frac{1}{\prod_j^p k^{(j)}} \sum_i \alpha_i \lambda_i^{p+1} x_i$$

Factoring the largest eigenvalue, λ_1,

$$y^{(p+1)} = Az^{(p)} = \frac{\lambda_1^{p+1}}{\prod_j^p k^{(j)}} \sum_i \alpha_i \left(\frac{\lambda_i}{\lambda_1}\right)^{p+1} x_i$$

Since $(\lambda_i/\lambda_1) < 1$ for $i > 1$ only the first term in the summation survives as $p \to \infty$. Thus $y^{(p)} \to x_1$, the eigenvector corresponding to the largest eigenvalue.

Several other features also become evident from the above analysis.

- The convergence rate will be faster, if the largest eigenvalue is well separated from the remaining eigenvalues.
- If the largest eigenvalue occurs with a multiplicity of two, then the above sequence will not converge, or rather converge to a subspace spaned by the eigenvectors corresponding to the eigenvalues that occur with multiplicity.

```
function [Rayleigh_Q,V] = Power(T,MaxIt)
%Power iteration to find the largest e.value of T

if nargin < 2,
   MaxIt = 100;
end

n=length(T);              %Find the size of T
z_old=rand(n,1);          %Generate random vector, z
z_new=z_old/norm(z_old); %Scale it
Check_sign=1; count = 0; %Initialize
while (norm(z_old-z_new) > 1.e-10 & count < MaxIt),
   count = count + 1;
   z_old=z_new*Check_sign;
   z_new=T*z_new;
   z_new=z_new/norm(z_new);
   Check_sign=sign((z_new'*(T*z_new))/(z_new'*z_new));
end
if (count >= MaxIt)
   error('Power iteration failed to converge')
end
%Compute the Rayliegh quotient
V=z_new;
Rayleigh_Q=(z_new'*(T*z_new))/(z_new'*z_new);
```

Figure 3.9: MATLAB implementation of power iteration

- If the initial guess vector does not contain any component in the direction of $\boldsymbol{x}_1$, then the sequence will converge to the next largest eigenvalue. This can be acheived by making the guess vector, $\boldsymbol{z}^{(0)}$, orthogonal to the known eigenvector, $\boldsymbol{x}_1$.

3.7.4 Inverse iteration

To find the smallest eigenvalue of a given a matrix $\boldsymbol{A}$, the power iteration could still be applied, but on the inverse of matrix $\boldsymbol{A}$. Consider the original eigenvalue problem

$$\boldsymbol{A}\boldsymbol{x}_i = \boldsymbol{\lambda}_i\boldsymbol{x}_i.$$

Premultiply by A^{-1} to get,

$$A^{-1}Ax_i = \lambda_i A^{-1}x_i$$

which can be rewritten as,

$$\frac{1}{\lambda_i}x_i = A^{-1}x_i.$$

Hence it is clear that the eigenvalues, μ_i of A^{-1} viz.

$$\mu_i x_i = A^{-1}x_i$$

are related to eigenvalues, λ_i of A through,

$$\lambda_i = \frac{1}{\mu_i}.$$

Although the illustration below uses the inverse of the matrix, in reality there is no need to find A^{-1} since power iteration only requires computation of a new vector y from a given vector z using,

$$y^{(p+1)} = A^{-1}z^{(p)}.$$

This can be done most effectively by solving the linear system

$$Ay^{(p+1)} = z^{(p)}$$

using LU factorization, which needs to be done only once.

3.7.5 Shift-Inverse iteration

To find the eigenvalue closest to a selected point, σ, the power iteration could be applied to the matrix $(A - \sigma I)^{-1}$, which is equivalent to solving the eigenvalue problem

$$(A - \sigma I)^{-1}x_i = \mu_i x_i.$$

This can be rewritten as,

$$\frac{1}{\mu_i}x_i = (A - \sigma I)x_i = Ax_i - \sigma x_i = (\lambda_i - \sigma)x_i.$$

Hence it is clear that the eigenvalues, μ_i of $(A - \sigma I)^{-1}$ are related to eigenvalues, λ_i of A through,

$$\lambda_i = \frac{1}{\mu_i} + \sigma.$$

As in the previous section, there is no need to find the inverse since power iteration only requires computation of a new vector $\boldsymbol{y}$ from a given vector $\boldsymbol{z}$ using,

$$\boldsymbol{y}^{(p+1)} = (\boldsymbol{A} - \sigma \boldsymbol{I})^{-1} \boldsymbol{z}^{(p)}.$$

This can be done most effectively by solving the linear system

$$(\boldsymbol{A} - \sigma \boldsymbol{I}) \boldsymbol{y}^{(p+1)} = \boldsymbol{z}^{(p)}$$

using $\boldsymbol{LU}$ factorization of $(\boldsymbol{A} - \sigma \boldsymbol{I})$, which needs to be done only once.

Example

Determine the largest and smallest eigenvalues of the matrix $\boldsymbol{T}$ defined below using the power iteration.

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

A typical MATLAB session follows.

```
» T=trid(2,-1,-2,4)  %generate the matrix

T =

     2    -1     0     0
    -2     2    -1     0
     0    -2     2    -1
     0     0    -2     2

»[d,v]=ch3_poweri(T) %apply power iteration

d =

    4.2882

v =

   -0.2012
    0.4603
   -0.6510
```

```
    0.5690

»max(eig(T)) %Find the largest e-value from ‘‘eig’’ function

ans =

    4.2882

%Find the smallest e. value
»mu=ch3_poweri(inv(T));
»lambda=1/mu

lambda =

   -0.2882
»min(eig(T))

ans =

   -0.2882

»%Find the e.value closest to 2.5
»sigma=2.5;
»mu=ch3_poweri(inv(T-sigma*eye(4)))

mu =

    2.6736

»lambda=sigma+1/mu

lambda =

    2.8740
%verfiy using ‘‘eig’’
»eig(T)

ans =

   -0.2882
    4.2882
    2.8740
```

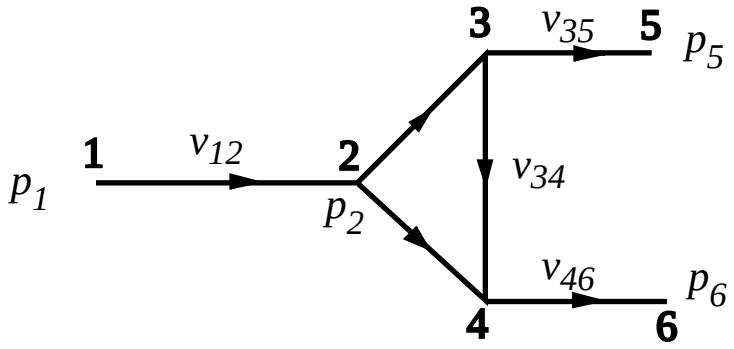

Figure 3.10: Laminar flow in a pipe network

```
1.1260
```

3.7.6 Wei-Prater analysis of a reaction system

3.8 Singular value decomposition

3.9 Genaralized inverse

3.10 Software tools

3.10.1 Lapack, Eispack library

3.10.2 MATLAB

3.10.3 Mathematica

3.11 Exercise problems

3.11.1 Laminar flow through a pipeline network

Consider laminar flow through the network shown in figure 3.10. The governing equations are the pressure drop equations for each pipe element $i - j$ and the mass balance equation at each node.
The pressure drop between nodes i and j is given by,

$$p_i - p_j = \alpha_{ij} v_{ij} \qquad \text{where} \qquad \alpha_{ij} = \frac{32 \mu l_{ij}}{d_{ij}^2} \tag{3.23}$$

The mass balance at node 2 is given, for example by,

$$d_{12}^2 v_{12} = d_{23}^2 v_{23} + d_{24}^2 v_{24} \tag{3.24}$$

Similar equations apply at nodes **3** and **4**. Let the unknown vector be

$$\underline{x} = [p_2\ p_3\ p_4\ v_{12}\ v_{23}\ v_{24}\ v_{34}\ v_{35}\ v_{46}]$$

There will be six momentum balance equations, one for each pipe element, and three mass balance (for incompressible fluids volume balance) equations, one at each node. Arrange them as a system of nine equations in nine unknowns and solve the resulting set of equations. Take the viscosity of the fluid, $\mu = 0.1 Pa \cdot s$. The dimensions of the pipes are given below.

Table 1

Element no	12	23	24	34	35	46
d_{ij} (m)	0.1	0.08	0.08	0.10	0.09	0.09
l_{ij} (m)	1000	800	800	900	1000	1000

a) Use MATLAB to solve this problem for the specified pressures of $p_1 = 300 kPa$ and $p_5 = p_6 = 100 kPa$. You need to assemble the system of equations in the form $A\,x = b$. Report flops. When reporting flops, report only for that particular operation - *i.e.*, initialize the counter using `flops(0)` before every operation.

- Compute the determinant of A. Report Flops.
- Compute the LU factor of A using built-in function `lu`. Report flops. What is the structure of L? Explain. The function `LU` provided in the lecture notes will fail on this matrix. Why?
- Compute the solution using `inv(A)*b`. Report flops.
- Compute the rank of A. Report Flops.
- Since A is sparse (*i.e.*, mostly zeros) we can avoid unnecessary operations, by using sparse matrix solvers. MATLAB Ver4.0 (not 3.5) provides such a facility. Sparse matrices are stored using triplets (i, j, s) where (i, j) identifies the non-zero entry in the matrix and s its corresponding value. The MATLAB function `find(A)` examines A and returns the triplets. Use,

» **[ii,jj,s]=find(A)**

Then construct the sparse matrix and store it in $\boldsymbol{S}$ using

» **S=sparse(ii,jj,s)**

Then to solve using the sparse solver and keep flops count, use

» **flops(0);** $\boldsymbol{x = S\backslash b}$**; flops**

Compare flops for solution by full and sparse matrix solution. To graphically view the structure of the sparse matrix, use

» **spy(S)**

Remember that you should have started MATLAB under X-windows for any graphics display of results!

- Compute the determinant of the sparse matrix, $\boldsymbol{S}$ (should be the same as the full matrix!). Report and compare flops.

b) Find out the new velocity and pressure distributions when $\boldsymbol{p}_6$ is changed to $\mathbf{150}\boldsymbol{kPa}$.

c) Suppose the forcing (column) vector in part (a) is $\boldsymbol{b1}$ and that in part (b) is $\boldsymbol{b2}$, report and explain the difference in flops for the following two ways of obtaining the two solutions using the sparse matrix. Note that in the first case both solutions are obtained simultaneously.

» $\boldsymbol{b = [b1, b2]}$**;flops(0);**$\boldsymbol{x = S\backslash b}$**,flops**
» **flops(0);**$\boldsymbol{x1 = S\backslash b1, x2 = S\backslash b2}$**,flops**

Repeat the above experiment with the full matrix, $\boldsymbol{A}$ and report flops.

d) Comment on how you would adopt the above problem formulation if a valve on line **34** is shut so that there is no flow in that line **34**.

3.11.2 Grahm-Schmidt procedure

Consider the vectors

$$\boldsymbol{x}_1 = [1234] \tag{3.25}$$

$$\boldsymbol{x}_2 = [1212] \tag{3.26}$$

$$\boldsymbol{x}_3 = [2131] \tag{3.27}$$
$$\boldsymbol{x}_4 = [2154] \tag{3.28}$$

- Check if they form a linearly independent set.
- Construct an orthonormal set using the Grahm-Schmidt procedure

3.11.3 Power iteration

Consider the matrix, $\boldsymbol{A}$ given below.

```
A =

   2.0 + 4.0i  -1.0 - 2.0i           0           0
  -2.0 - 4.0i   2.0 + 4.0i      -1.0 - 2.0i      0
        0      -2.0 - 4.0i       2.0 + 4.0i  -1.0 - 2.0i
        0           0           -2.0 - 4.0i   2.0 + 4.0i
```

Use the power iteration algorithmn given in Fig. 3.9 to solve the following problems.

- Find the largest eigenvalue. [Ans: 4.2882 + 8.5765i]
- Find the smallest eigenvalue. [Ans: -0.2882 - 0.5765i]
- Find the eigenvalue closest to $\boldsymbol{\sigma = (1+2i)}$. [Ans: 1.1260 + 2.2519i]