JAAN KIUSALAAS

Numerical Methods in Engineering with MATLAB®

CAMBRIDGE

CAMBRIDGE more information - www.cambridge.org/9780521852883

This page intentionally left blank

Numerical Methods in Engineering with MATLAB®

Numerical Methods in Engineering with MATLAB® is a text for engineering students and a reference for practicing engineers, especially those who wish to explore the power and efficiency of MATLAB. The choice of numerical methods was based on their relevance to engineering problems. Every method is discussed thoroughly and illustrated with problems involving both hand computation and programming. MATLAB M-files accompany each method and are available on the book web site. This code is made simple and easy to understand by avoiding complex book-keeping schemes, while maintaining the essential features of the method. MATLAB, was chosen as the example language because of its ubiquitous use in engineering studies and practice. Moreover, it is widely available to students on school networks and through inexpensive educational versions. MATLAB a popular tool for teaching scientific computation.

Jaan Kiusalaas is a Professor Emeritus in the Department of Engineering Science and Mechanics at the Pennsylvania State University. He has taught numerical methods, including finite element and boundary element methods for over 30 years. He is also the co-author of four other Books—*Engineering Mechanics: Statics*, *Engineering Mechanics: Dynamics*, *Mechanics of Materials*, and an alternate version of this work with Python code.

NUMERICAL METHODS IN ENGINEERING WITH MATLAB®

Jaan Kiusalaas
The Pennsylvania State University

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 2RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521852883

First published in print format 2005

ISBN-13 978-0-511-12676-5 eBook (Adobe Reader)
ISBN-10 0-511-12676-X eBook (Adobe Reader)

ISBN-13 978-0-521-85288-3 hardback
ISBN-10 0-521-85288-9 hardback

Contents

Preface

This book is targeted primarily toward engineers and engineering students of advanced standing (sophomores, seniors and graduate students). Familiarity with a computer language is required; knowledge of basic engineering subjects is useful, but not essential.

The text attempts to place emphasis on numerical methods, not programming. Most engineers are not programmers, but problem solvers. They want to know what methods can be applied to a given problem, what are their strengths and pitfalls and how to implement them. Engineers are not expected to write computer code for basic tasks from scratch; they are more likely to utilize functions and subroutines that have been already written and tested. Thus programming by engineers is largely confined to assembling existing pieces of code into a coherent package that solves the problem at hand.

The "piece" of code is usually a function that implements a specific task. For the user the details of the code are unimportant. What matters is the interface (what goes in and what comes out) and an understanding of the method on which the algorithm is based. Since no numerical algorithm is infallible, the importance of understanding the underlying method cannot be overemphasized; it is, in fact, the rationale behind learning numerical methods.

This book attempts to conform to the views outlined above. Each numerical method is explained in detail and its shortcomings are pointed out. The examples that follow individual topics fall into two categories: hand computations that illustrate the inner workings of the method, and small programs that show how the computer code is utilized in solving a problem. Problems that require programming are marked with ■.

The material consists of the usual topics covered in an engineering course on numerical methods: solution of equations, interpolation and data fitting, numerical differentiation and integration, solution of ordinary differential equations and eigenvalue problems. The choice of methods within each topic is tilted toward relevance

to engineering problems. For example, there is an extensive discussion of symmetric, sparsely populated coefficient matrices in the solution of simultaneous equations. In the same vein, the solution of eigenvalue problems concentrates on methods that efficiently extract specific eigenvalues from banded matrices.

An important criterion used in the selection of methods was clarity. Algorithms requiring overly complex bookkeeping were rejected regardless of their efficiency and robustness. This decision, which was taken with great reluctance, is in keeping with the intent to avoid emphasis on programming.

The selection of algorithms was also influenced by current practice. This disqualified several well-known historical methods that have been overtaken by more recent developments. For example, the secant method for finding roots of equations was omitted as having no advantages over Brent's method. For the same reason, the multistep methods used to solve differential equations (e.g., Milne and Adams methods) were left out in favor of the adaptive Runge–Kutta and Bulirsch–Stoer methods.

Notably absent is a chapter on partial differential equations. It was felt that this topic is best treated by finite element or boundary element methods, which are outside the scope of this book. The finite difference model, which is commonly introduced in numerical methods texts, is just too impractical in handling multidimensional boundary value problems.

As usual, the book contains more material than can be covered in a three-credit course. The topics that can be skipped without loss of continuity are tagged with an asterisk (*).

The programs listed in this book were tested with MATLAB® 6.5.0 and under Windows® XP. The source code can be downloaded from the book's website at

www.cambridge.org/0521852889

The author wishes to express his gratitude to the anonymous reviewers and Professor Andrew Pytel for their suggestions for improving the manuscript. Credit is also due to the authors of *Numerical Recipes* (Cambridge University Press) whose presentation of numerical methods was inspirational in writing this book.

1 Introduction to MATLAB

1.1 General Information

Quick Overview

This chapter is not intended to be a comprehensive manual of MATLAB®. Our sole aim is to provide sufficient information to give you a good start. If you are familiar with another computer language, and we assume that you are, it is not difficult to pick up the rest as you go.

MATLAB is a high-level computer language for scientific computing and data visualization built around an interactive programming environment. It is becoming the premiere platform for scientific computing at educational institutions and research establishments. The great advantage of an interactive system is that programs can be tested and debugged quickly, allowing the user to concentrate more on the principles behind the program and less on programming itself. Since there is no need to compile, link and execute after each correction, MATLAB programs can be developed in much shorter time than equivalent FORTRAN or C programs. On the negative side, MATLAB does not produce stand-alone applications—the programs can be run only on computers that have MATLAB installed.

MATLAB has other advantages over mainstream languages that contribute to rapid program development:

- MATLAB contains a large number of functions that access proven numerical libraries, such as LINPACK and EISPACK. This means that many common tasks (e.g., solution of simultaneous equations) can be accomplished with a single function call.
- There is extensive graphics support that allows the results of computations to be plotted with a few statements.
- All numerical objects are treated as double-precision arrays. Thus there is no need to declare data types and carry out type conversions.

The syntax of MATLAB resembles that of FORTRAN. To get an idea of the similarities, let us compare the codes written in the two languages for solution of simultaneous equations $\mathbf{Ax} = \mathbf{b}$ by Gauss elimination. Here is the subroutine in FORTRAN 90:

```
subroutine gauss(A,b,n)
   use prec_mod
   implicit none
   real(DP), dimension(:,:), intent(in out) :: A
   real(DP), dimension(:),   intent(in out) :: b
   integer, intent(in)                      :: n
   real(DP) :: lambda
   integer  :: i,k
 ! --------------Elimination phase--------------
   do k = 1,n-1
      do i = k+1,n
         if(A(i,k) /= 0) then
            lambda = A(i,k)/A(k,k)
            A(i,k+1:n) = A(i,k+1:n) - lambda*A(k,k+1:n)
            b(i) = b(i) - lambda*b(k)
         end if
      end do
   end do
 ! ------------Back substitution phase----------
   do k = n,1,-1
      b(k) = (b(k) - sum(A(k,k+1:n)*b(k+1:n)))/A(k,k)
   end do
   return
end subroutine gauss
```

The statement `use prec_mod` tells the compiler to load the module `prec_mod` (not shown here), which defines the word length `DP` for floating-point numbers. Also note the use of array sections, such as `a(k,k+1:n)`, a feature that was not available in previous versions of FORTRAN.

The equivalent MATLAB function is (MATLAB does not have subroutines):

```
function b = gauss(A,b)
n = length(b);
%-----------------Elimination phase-------------
for k = 1:n-1
    for i = k+1:n
```

```
        if A(i,k) ~= 0
            lambda = A(i,k)/A(k,k);
            A(i,k+1:n) = A(i,k+1:n) - lambda*A(k,k+1:n);
            b(i)= b(i) - lambda*b(k);
         end
     end
end
%--------------Back substitution phase-----------
for k = n:-1:1
    b(k) = (b(k) - A(k,k+1:n)*b(k+1:n))/A(k,k);
end
```

Simultaneous equations can also be solved in MATLAB with the simple command `A\b` (see below).

MATLAB can be operated in the interactive mode through its command window, where each command is executed immediately upon its entry. In this mode MATLAB acts like an electronic calculator. Here is an example of an interactive session for the solution of simultaneous equations:

```
>> A = [2 1 0; -1 2 2; 0 1 4]; % Input 3 x 3 matrix
>> b = [1; 2; 3];              % Input column vector
>> soln = A\b                  % Solve A*x = b by left division
soln =
    0.2500
    0.5000
    0.6250
```

The symbol >> is MATLAB's prompt for input. The percent sign (%) marks the beginning of a comment. A semicolon (;) has two functions: it suppresses printout of intermediate results and separates the rows of a matrix. Without a terminating semicolon, the result of a command would be displayed. For example, omission of the last semicolon in the line defining the matrix A would result in

```
>> A = [2 1 0; -1 2 2; 0 1 4]
A =
     2     1     0
    -1     2     2
     0     1     4
```

Functions and programs can be created with the MATLAB editor/debugger and saved with the `.m` extension (MATLAB calls them M-files). The file name of a saved function should be identical to the name of the function. For example, if the function for Gauss elimination listed above is saved as `gauss.m`, it can be called just like any MATLAB function:

```
>> A = [2 1 0; -1 2 2; 0 1 4];
>> b = [1; 2; 3];
>> soln = gauss(A,b)
soln =
    0.2500
    0.5000
    0.6250
```

1.2 Data Types and Variables

Data Types

The most commonly used MATLAB data types, or *classes*, are `double`, `char` and `logical`, all of which are considered by MATLAB as arrays. Numerical objects belong to the class `double`, which represents double-precision arrays; a scalar is treated as a 1×1 array. The elements of a `char` type array are strings (sequences of characters), whereas a `logical` type array element may contain only 1 (true) or 0 (false).

Another important class is `function_handle`, which is unique to MATLAB. It contains information required to find and execute a function. The name of a function handle consists of the character @, followed by the name of the function; e.g., `@sin`. Function handles are used as input arguments in function calls. For example, suppose that we have a MATLAB function `plot(func,x1,x2)` that plots any user-specified function `func` from `x1` to `x2`. The function call to plot $\sin x$ from 0 to π would be `plot(@sin,0,pi)`.

There are other data types, but we seldom come across them in this text. Additional classes can be defined by the user. The class of an object can be displayed with the `class` command. For example,

```
>> x = 1 + 3i  % Complex number
>> class(x)
ans =
double
```

Variables

Variable names, which must start with a letter, are *case sensitive*. Hence `xstart` and `xStart` represent two different variables. The length of the name is unlimited, but only the first *N* characters are significant. To find *N* for your installation of MATLAB, use the command `namelengthmax`:

```
>> namelengthmax
ans =
     63
```

Variables that are defined within a MATLAB function are local in their scope. They are not available to other parts of the program and do not remain in memory after exiting the function (this applies to most programming languages). However, variables can be shared between a function and the calling program if they are declared `global`. For example, by placing the statement `global X Y` in a function as well as the calling program, the variables `X` and `Y` are shared between the two program units. The recommended practice is to use capital letters for global variables.

MATLAB contains several built-in constants and special variables, most important of which are

`ans`	Default name for results
`eps`	Smallest number for which `1 + eps > 1`
`inf`	Infinity
`NaN`	Not a number
`i or j`	$\sqrt{-1}$
`pi`	π
`realmin`	Smallest usable positive number
`realmax`	Largest usable positive number

Here are a few of examples:

```
>> warning off  % Suppresses print of warning messages
>> 5/0
ans =
   Inf

>> 0/0
```

```
ans =
   NaN

>> 5*NaN        % Most operations with NaN result in NaN
ans =
   NaN

>> NaN == NaN   % Different NaN's are not equal!
ans =
     0

>> eps
ans =
  2.2204e-016
```

Arrays

Arrays can be created in several ways. One of them is to type the elements of the array between brackets. The elements in each row must be separated by blanks or commas. Here is an example of generating a 3×3 matrix:

```
>> A = [ 2 -1  0
        -1  2 -1
         0 -1  1]
A =
     2    -1     0
    -1     2    -1
     0    -1     1
```

The elements can also be typed on a single line, separating the rows with semicolons:

```
>> A = [2 -1 0; -1 2 -1; 0 -1 1]
A =
     2    -1     0
    -1     2    -1
     0    -1     1
```

Unlike most computer languages, MATLAB differentiates between row and column vectors (this peculiarity is a frequent source of programming and input errors). For example,

```
>> b = [1 2 3]        % Row vector
b =
     1     2     3

>> b = [1; 2; 3]      % Column vector
b =
     1
     2
     3

>> b = [1 2 3]'       % Transpose of row vector
b =
     1
     2
     3
```

The single quote (') is the *transpose operator* in MATLAB; thus b' is the transpose of b.

The elements of a matrix, such as

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

can be accessed with the statement A(i,j), where i and j are the row and column numbers, respectively. A section of an array can be extracted by the use of colon notation. Here is an illustration:

```
>> A = [8 1 6; 3 5 7; 4 9 2]
A =
     8     1     6
     3     5     7
     4     9     2

>> A(2,3)     % Element in row 2, column 3
ans =
     7

>> A(:,2)     % Second column
```

```
ans =
     1
     5
     9

>> A(2:3,2:3)  % The 2 x 2 submatrix in lower right corner
ans =
     5     7
     9     2
```

Array elements can also be accessed with a single index. Thus `A(i)` extracts the `i`th element of `A`, counting the elements down the columns. For example, `A(7)` and `A(1,3)` would extract the same element from a 3×3 matrix.

Cells

A cell array is a sequence of arbitrary objects. Cell arrays can be created by enclosing their contents between braces {}. For example, a cell array `c` consisting of three cells can be created by

```
>> c = {[1 2 3], 'one two three', 6 + 7i}
c =
    [1x3 double]    'one two three'    [6.0000+ 7.0000i]
```

As seen above, the contents of some cells are not printed in order to save space. If all contents are to be displayed, use the `celldisp` command:

```
>> celldisp(c)
c{1} =
     1     2     3
c{2} =
one two three
c{3} =
   6.0000 + 7.0000i
```

Braces are also used to extract the contents of the cells:

```
>> c{1}              % First cell
ans =
     1     2     3
```

```
>> c{1}(2)           % Second element of first cell
ans =
     2
>> c{2}              % Second cell
ans =
one two three
```

Strings

A string is a sequence of characters; it is treated by MATLAB as a character array. Strings are created by enclosing the characters between single quotes. They are concatenated with the function `strcat`, whereas a colon operator `(:)` is used to extract a portion of the string. For example,

```
>> s1 = 'Press return to exit';   % Create a string
>> s2 = ' the program';           % Create another string
>> s3 = strcat(s1,s2)             % Concatenate s1 and s2
s3 =
Press return to exit the program
>> s4 = s1(1:12)                  % Extract chars. 1-12 of s1
s4 =
Press return
```

1.3 Operators

Arithmetic Operators

MATLAB supports the usual arithmetic operators:

+	Addition
−	Subtraction
*	Multiplication
^	Exponentiation

When applied to matrices, they perform the familiar matrix operations, as illustrated below.

```
>> A = [1 2 3; 4 5 6]; B = [7 8 9; 0 1 2];

>> A + B                % Matrix addition
```

```
ans =
     8    10    12
     4     6     8

>> A*B'                 % Matrix multiplication
ans =
    50     8
   122    17

>> A*B                  % Matrix multiplication fails
??? Error using ==> *   % due to incompatible dimensions
Inner matrix dimensions must agree.
```

There are two division operators in MATLAB:

/	Right division
\	Left division

If a and b are scalars, the right division a/b results in a divided by b, whereas the left division is equivalent to b/a. In the case where A and B are matrices, A/B returns the solution of X*A = B and A\B yields the solution of A*X = B.

Often we need to apply the *, / and ^ operations to matrices in an element-by-element fashion. This can be done by preceding the operator with a period (.) as follows:

.*	Element-wise multiplication
./	Element-wise division
.^	Element-wise exponentiation

For example, the computation $C_{ij} = A_{ij}\, B_{ij}$ can be accomplished with

```
>> A = [1 2 3; 4 5 6]; B = [7 8 9; 0 1 2];
>> C = A.*B
C =
     7    16    27
     0     5    12
```

Comparison Operators

The comparison (relational) operators return 1 for true and 0 for false. These operators are

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
~=	Not equal to

The comparison operators always act element-wise on matrices; hence they result in a matrix of `logical` type. For example,

```
>> A = [1 2 3; 4 5 6]; B = [7 8 9; 0 1 2];
>> A > B
ans =
     0     0     0
     1     1     1
```

Logical Operators

The logical operators in MATLAB are

&	AND
|	OR
~	NOT

They are used to build compound relational expressions, an example of which is shown below.

```
>> A = [1 2 3; 4 5 6]; B = [7 8 9; 0 1 2];
>> (A > B) | (B > 5)
ans =
     1     1     1
     1     1     1
```

1.4 Flow Control

Conditionals

if, else, elseif
The `if` construct

```
if condition
    block
end
```

executes the *block* of statements if the *condition* is true. If the condition is false, the block skipped. The `if` conditional can be followed by any number of `elseif` constructs:

```
if condition
    block
elseif condition
    block
⋮
end
```

which work in the same manner. The `else` clause

```
⋮
else
    block
end
```

can be used to define the block of statements which are to be executed if none of the `if-elseif` clauses are true. The function `signum` below illustrates the use of the conditionals.

```
function sgn = signum(a)
if a > 0
    sgn = 1;
elseif a < 0
    sgn = -1;
else
```

```
    sgn = 0;
end

>> signum (-1.5)
ans =
    -1
```

switch

The `switch` construct is

```
switch expression
    case value1
        block
    case value2
        block
⋮
    otherwise
        block
end
```

Here the *expression* is evaluated and the control is passed to the `case` that matches the value. For instance, if the value of *expression* is equal to *value2*, the *block* of statements following `case` *value2* is executed. If the value of *expression* does not match any of the `case` values, the control passes to the optional `otherwise` block. Here is an example:

```
function y = trig(func,x)
switch func
    case 'sin'
        y = sin(x);
    case 'cos'
        y = cos(x);
    case 'tan'
        y = tan(x);
    otherwise
        error('No such function defined')
end

>> trig('tan',pi/3)
ans =
    1.7321
```

Loops

while

The `while` construct

```
while condition:
    block
end
```

executes a *block* of statements if the *condition* is true. After execution of the block, *condition* is evaluated again. If it is still true, the block is executed again. This process is continued until the *condition* becomes false.

The following example computes the number of years it takes for a $1000 principal to grow to $10,000 at 6% annual interest.

```
>> p = 1000; years = 0;
>> while p < 10000
      years = years + 1;
      p = p*(1 + 0.06);
   end
>> years
years =
    40
```

for

The `for` loop requires a *target* and a *sequence* over which the target loops. The form of the construct is

```
for target = sequence
    block
end
```

For example, to compute $\cos x$ from $x = 0$ to $\pi/2$ at increments of $\pi/10$ we could use

```
>> for n = 0:5  % n loops over the sequence 0 1 2 3 4 5
      y(n+1) = cos(n*pi/10);
   end
>> y
y =
   1.0000    0.9511    0.8090    0.5878    0.3090    0.0000
```

Loops should be avoided whenever possible in favor of *vectorized* expressions, which execute much faster. A vectorized solution to the last computation would be

```
>> n = 0:5;
>> y = cos(n*pi/10)
y =
   1.0000    0.9511    0.8090    0.5878    0.3090    0.0000
```

break

Any loop can be terminated by the `break` statement. Upon encountering a `break` statement, the control is passed to the first statement outside the loop. In the following example the function `buildvec` constructs a row vector of arbitrary length by prompting for its elements. The process is terminated when an empty element is encountered.

```
function x = buildvec
for i = 1:1000
    elem = input('==> '); % Prompts for input of element
    if isempty(elem)      % Check for empty element
        break
    end
    x(i) = elem;
end

>> x = buildvec
==> 3
==> 5
==> 7
==> 2
==>
x =
     3     5     7     2
```

continue

When the `continue` statement is encountered in a loop, the control is passed to the next iteration without executing the statements in the current iteration. As an illustration, consider the following function that strips all the blanks from the string `s1`:

```
function s2 = strip(s1)
s2 = '';                        % Create an empty string
for i = 1:length(s1)
```

```
    if s1(i) == ' '
        continue
    else
        s2 = strcat(s2,s1(i)); % Concatenation
    end
end

>> s2 = strip('This is too bad')
s2 =
Thisistoobad
```

return

A function normally returns to the calling program when it runs out of statements. However, the function can be forced to exit with the `return` command. In the example below, the function `solve` uses the Newton–Raphson method to find the zero of $f(x) = \sin x - 0.5x$. The input `x` (guess of the solution) is refined in successive iterations using the formula $x \leftarrow x + \Delta x$, where $\Delta x = -f(x)/f'(x)$, until the change Δx becomes sufficiently small. The procedure is then terminated with the `return` statement. The `for` loop assures that the number of iterations does not exceed 30, which should be more than enough for convergence.

```
function x = solve(x)
for numIter = 1:30
    dx = -(sin(x) - 0.5*x)/(cos(x) - 0.5); % -f(x)/f'(x)
    x = x + dx;
    if abs(dx) < 1.0e-6          % Check for convergence
        return
    end
end
error('Too many iterations')

>> x = solve(2)
x =
    1.8955
```

error

Execution of a program can be terminated and a message displayed with the `error` function

```
error('message')
```

For example, the following program lines determine the dimensions of a matrix and aborts the program if the dimensions are not equal.

```
[m,n] = size(A);  % m = no. of rows; n = no. of cols.
if m ~= n
    error('Matrix must be square')
end
```

1.5 Functions

Function Definition

The body of a function must be preceded by the function definition line

`function` [*output_args*] = *function_name*(*input_arguments*)

The input and output arguments must be separated by commas. The number of arguments may be zero. If there is only one output argument, the enclosing brackets may be omitted.

To make the function accessible to other programs units, it must be saved under the file name *function_name*.`m`. This file may contain other functions, called *subfunctions*. The subfunctions can be called only by the primary function *function_name* or other subfunctions in the file; they are not accessible to other program units.

Calling Functions

A function may be called with fewer arguments than appear in the function definition. The number of input and output arguments used in the function call can be determined by the functions `nargin` and `nargout`, respectively. The following example shows a modified version of the function `solve` that involves two input and two output arguments. The error tolerance `epsilon` is an optional input that may be used to override the default value `1.0e-6`. The output argument `numIter`, which contains the number of iterations, may also be omitted from the function call.

```
function [x,numIter] = solve(x,epsilon)
if nargin == 1              % Specify default value if
    epsilon = 1.0e-6;       % second input argument is
end                         % omitted in function call
for numIter = 1:100
    dx = -(sin(x) - 0.5*x)/(cos(x) - 0.5);
    x = x + dx;
    if abs(dx) < epsilon    % Converged; return to
        return              % calling program
    end
```

```
end
error('Too many iterations')

>> x = solve(2)            % numIter not printed
x =
    1.8955

>> [x,numIter] = solve(2)  % numIter is printed
x =
    1.8955
numIter =
     4

>> format long
>> x = solve(2,1.0e-12)    % Solving with extra precision
x =
   1.89549426703398
>>
```

Evaluating Functions

Let us consider a slightly different version of the function `solve` shown below. The expression for `dx`, namely $\Delta x = -f(x)/f'(x)$, is now coded in the function `myfunc`, so that `solve` contains a call to `myfunc`. This will work fine, provided that `myfunc` is stored under the file name `myfunc.m` so that MATLAB can find it.

```
function [x,numIter] = solve(x,epsilon)
if nargin == 1; epsilon = 1.0e-6; end
for numIter = 1:30
    dx = myfunc(x);
    x = x + dx;
    if abs(dx) < epsilon; return; end
end
error('Too many iterations')

function y = myfunc(x)
y = -(sin(x) - 0.5*x)/(cos(x) - 0.5);

>> x = solve(2)
x =
    1.8955
```

In the above version of `solve` the function returning `dx` is stuck with the name `myfunc`. If `myfunc` is replaced with another function name, `solve` will not work unless the corresponding change is made in its code. In general, it is not a good idea to alter computer code that has been tested and debugged; all data should be communicated to a function through its arguments. MATLAB makes this possible by passing the function handle of `myfunc` to `solve` as an argument, as illustrated below.

```
function [x,numIter] = solve(func,x,epsilon)
if nargin == 2; epsilon = 1.0e-6; end
for numIter = 1:30
    dx = feval(func,x);   % feval is a MATLAB function for
    x = x + dx;           % evaluating a passed function
    if abs(dx) < epsilon; return; end
end
error('Too many iterations')

>> x = solve(@myfunc,2)   % @myfunc is the function handle
x =
    1.8955
```

The call `solve(@myfunc,2)`creates a function handle to `myfunc` and passes it to `solve` as an argument. Hence the variable `func` in `solve` contains the handle to `myfunc`. A function passed to another function by its handle is evaluated by the MATLAB function

`feval(`*function_handle, arguments*`)`

It is now possible to use `solve` to find a zero of any $f(x)$ by coding the function $\Delta x = -f(x)/f'(x)$ and passing its handle to `solve`.

In-Line Functions

If the function is not overly complicated, it can also be represented as an *inline* object:

function_name = `inline('`*expression*`','`*var1*`','`*var2*`',...)`

where *expression* specifies the function and *var1, var2,* ... are the names of the independent variables. Here is an example:

```
>> myfunc = inline ('x^2 + y^2','x','y');
>> myfunc (3,5)
ans =
    34
```

The advantage of an in-line function is that it can be embedded in the body of the code; it does not have to reside in an M-file.

1.6 Input/Output

Reading Input

The MATLAB function for receiving user input is

```
value = input('prompt')
```

It displays a prompt and then waits for input. If the input is an expression, it is evaluated and returned in *value*. The following two samples illustrate the use of `input`:

```
>> a = input('Enter expression: ')
Enter expression: tan(0.15)
a =
    0.1511

>> s = input('Enter string: ')
Enter string: 'Black sheep'
s =
Black sheep
```

Printing Output

As mentioned before, the result of a statement is printed if the statement does not end with a semicolon. This is the easiest way of displaying results in MATLAB. Normally MATLAB displays numerical results with about five digits, but this can be changed with the format command:

`format long`	switches to 16-digit display
`format short`	switches to 5-digit display

To print formatted output, use the `fprintf` function:

```
fprintf('format', list)
```

where *format* contains formatting specifications and *list* is the list of items to be printed, separated by commas. Typically used formatting specifications are

%*w*.*d*f	Floating point notation
%*w*.*d*e	Exponential notation
\n	Newline character

where *w* is the width of the field and *d* is the number of digits after the decimal point. Line break is forced by the newline character. The following example prints a formatted table of sin *x* vs. *x* at intervals of 0.2:

```
>> x = 0:0.2:1;
>> for i = 1:length(x)
       fprintf('%4.1f %11.6f\n',x(i),sin(x(i)))
   end
 0.0    0.000000
 0.2    0.198669
 0.4    0.389418
 0.6    0.564642
 0.8    0.717356
 1.0    0.841471
```

1.7 Array Manipulation

Creating Arrays

We learned before that an array can be created by typing its elements between brackets:

```
>> x = [0 0.25 0.5 0.75 1]
x =
         0    0.2500    0.5000    0.7500    1.0000
```

Colon Operator

Arrays with equally spaced elements can also be constructed with the colon operator.

$$x = first\_elem : increment : last\_elem$$

For example,

```
>> x = 0:0.25:1
x =
         0    0.2500    0.5000    0.7500    1.0000
```

linspace

Another means of creating an array with equally spaced elements is the `linspace` function. The statement

```
x = linspace(xfirst,xlast,n)
```

creates an array of n elements starting with *xfirst* and ending with *xlast*. Here is an illustration:

```
>> x = linspace(0,1,5)
x =
         0    0.2500    0.5000    0.7500    1.0000
```

logspace

The function `logspace` is the logarithmic counterpart of `linspace`. The call

```
x = logspace(zfirst,zlast,n)
```

creates n logarithmically spaced elements starting with $x = 10^{z\ first}$ and ending with $x = 10^{z\ last}$. Here is an example:

```
>> x = logspace(0,1,5)
x =
    1.0000    1.7783    3.1623    5.6234   10.0000
```

zeros

The function call

```
X = zeros(m,n)
```

returns a matrix of m rows and n columns that is filled with zeroes. When the function is called with a single argument, e.g., `zeros(n)`, a $n \times n$ matrix is created.

ones

```
X = ones(m,n)
```

The function `ones` works in the manner as `zeros`, but fills the matrix with ones.

rand

```
X = rand(m,n)
```

This function returns a matrix filled with random numbers between 0 and 1.

eye
The function `eye`

```
X = eye(n)
```

creates an $n \times n$ identity matrix.

Array Functions

There are numerous array functions in MATLAB that perform matrix operations and other useful tasks. Here are a few basic functions:

length
The length n (number of elements) of a vector x can be determined with the function `length`:

```
n = length(x)
```

size
If the function `size` is called with a single input argument:

```
[m,n] = size(X)
```

it determines the number of rows m and number of columns n in the matrix X. If called with two input arguments:

```
m = size(X,dim)
```

it returns the length of X in the specified dimension (dim = `1` yields the number of rows, and $dim = 2$ gives the number of columns).

reshape
The reshape function is used to rearrange the elements of a matrix. The call

```
Y = reshape(X,m,n)
```

returns a $m \times n$ matrix the elements of which are taken from matrix X in the column-wise order. The total number of elements in X must be equal to $m \times n$. Here is an example:

```
>> a = 1:2:11
a =
     1     3     5     7     9    11
>> A = reshape(a,2,3)

A =
     1     5     9
     3     7    11
```

dot

$$a = \texttt{dot}(x,y)$$

This function returns the dot product of two vectors x and y which must be of the same length.

prod

$$a = \texttt{prod}(x)$$

For a vector x, `prod(`x`)` returns the product of its elements. If x is a matrix, then a is a row vector containing the products over each column. For example,

```
>> a = [1 2 3 4 5 6];
>> A = reshape(a,2,3)
A =
     1     3     5
     2     4     6

>> prod(a)
ans =
   720

>> prod(A)
ans =
     2    12    30
```

sum

$$a = \texttt{sum}(x)$$

This function is similar to `prod`, except that it returns the sum of the elements.

cross

```
c = cross(a,b)
```

The function `cross` computes the cross product: $c = a \times b$, where vectors a and b must be of length 3.

1.8 Writing and Running Programs

MATLAB has two windows available for typing program lines: the *command window* and the *editor/debugger*. The command window is always in the interactive mode, so that any statement entered into the window is immediately processed. The interactive mode is a good way to experiment with the language and try out programming ideas.

MATLAB opens the editor window when a new M-file is created, or an existing file is opened. The editor window is used to type and save programs (called *script files* in MATLAB) and functions. One could also use a text editor to enter program lines, but the MATLAB editor has MATLAB-specific features, such as color coding and automatic indentation, that make work easier. Before a program or function can be executed, it must be saved as a MATLAB M-file (recall that these files have the `.m` extension). A program can be run by invoking the *run* command from the editor's *debug* menu.

When a function is called for the first time during a program run, it is compiled into P-code (pseudo-code) to speed up execution in subsequent calls to the function. One can also create the P-code of a function and save it on disk by issuing the command

```
pcode function_name
```

MATLAB will then load the P-code (which has the `.p` extension) into the memory rather than the text file.

The variables created during a MATLAB session are saved in the MATLAB *workspace* until they are cleared. Listing of the saved variables can be displayed by the command `who`. If greater detail about the variables is required, type `whos`. Variables can be cleared from the workspace with the command

```
clear a b ...
```

which clears the variables *a*, *b*, ... If the list of variables is omitted, all variables are cleared.

Assistance on any MATLAB function is available by typing

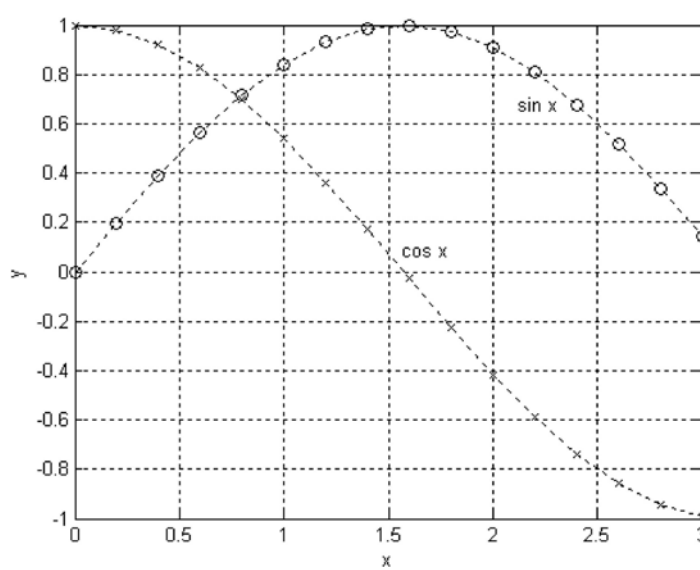
```
help function_name
```

in the command window.

1.9 Plotting

MATLAB has extensive plotting capabilities. Here we illustrate some basic commands for two-dimensional plots. The example below plots sin x and cos x on the same plot.

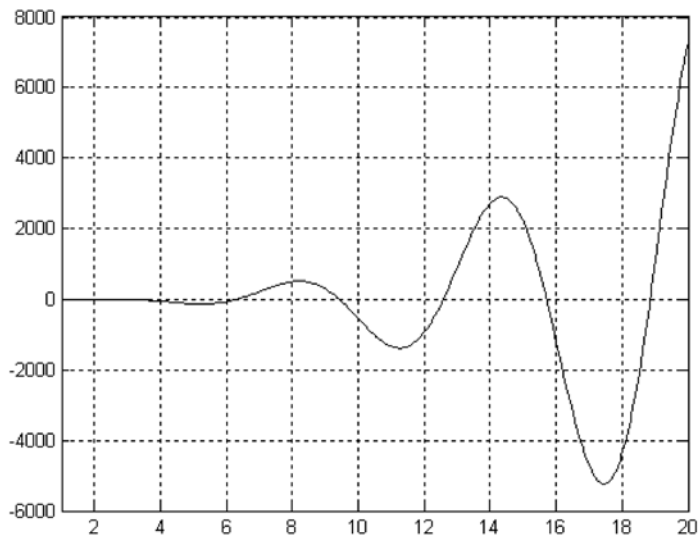
```
>> x = 0:0.2:pi;   % Create x-array
>> y = sin(x);     % Create y-array
>> plot(x,y,'k:o') % Plot x-y points with specified color
                   % and symbol ('k' = black, 'o' = circles)
>> hold on         % Allow overwriting of current plot
>> z = cos(x);     % Create z-array
>> plot(x,z,'k:x') % Plot x-z points ('x' = crosses)
>> grid on         % Display coordinate grid
>> xlabel('x')     % Display label for x-axis
>> ylabel('y')     % Display label for y-axis
>> gtext('sin x')  % Create mouse-movable text
>> gtext('cos x')
```



A function stored in a M-file can be plotted with a single command, as shown below.

```
function y = testfunc(x)      % Stored function
y = (x.^3).*sin(x) - 1./x;

>> fplot(@testfunc,[1 20])    % Plot from x = 1 to 20
>> grid on
```



The plots appearing in this book from here on were not produced by MATLAB. We used the copy/paste operation to transfer the numerical data to a spreadsheet and then let the spreadsheet create the plot. This resulted in plots more suited for publication.

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 this—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, etc.). 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 and sparse coefficient matrices.

Notation

A system of algebraic equations has the form

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

$$\begin{aligned} A_{11}x_1 + A_{12}x_2 + \cdots + A_{1n}x_n &= b_1 \\ A_{21}x_1 + A_{22}x_2 + \cdots + A_{2n}x_n &= b_2 \\ A_{31}x_1 + A_{32}x_2 + \cdots + A_{3n}x_n &= b_3 \\ &\vdots \\ A_{n1}x_1 + A_{n2}x_2 + \cdots + A_{nn}x_n &= b_n \end{aligned} \tag{2.1}$$

where the coefficients A_{ij} and the constants b_j are known, and x_i represent 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:

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

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*, i.e., if $|\mathbf{A}| \neq 0$. 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. On the other hand, 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

An obvious question is: what happens when the coefficient matrix is almost singular; i.e., if $|\mathbf{A}|$ is very small? In order 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, 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

$$\|\mathbf{A}\| = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} A_{ij}^2} \qquad \|\mathbf{A}\| = \max_{1\le i\le n} \sum_{j=1}^{n} \left|A_{ij}\right| \tag{2.5a}$$

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

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

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, consider 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 be 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 three operations listed below. These so-called *elementary operations* do not change the solution, but they may affect the determinant of the coefficient matrix as indicated in parentheses.

- Exchanging two equations (changes sign of $|\mathbf{A}|$).
- Multiplying an equation by a nonzero constant (multiplies $|\mathbf{A}|$ by the same constant).
- 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 $\mathbf{x}$, and then repeatedly refine the solution until a certain convergence criterion is reached. Iterative methods are generally less efficient than their direct counterparts due to the large number of iterations required. But 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

In the above table $\mathbf{U}$ represents an upper triangular matrix, $\mathbf{L}$ is a lower triangular matrix and $\mathbf{I}$ 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, since they simplify many computations. For example, consider the equations $\mathbf{Lx} = \mathbf{c}$, or

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

If we solve the equations forward, starting with the first equation, the computations are very easy, since each equation would contain only one unknown at a time. The solution would thus proceed as follows:

$$\begin{aligned} x_1 &= c_1/L_{11} \\ x_2 &= (c_2 - L_{21}x_1)/L_{22} \\ x_3 &= (c_3 - L_{31}x_1 - L_{32}x_2)/L_{33} \\ &\vdots \end{aligned}$$

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 (see Appendix A2) of the determinant about the first row of **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 **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 $\mathbf{x} = [\,2 \quad -1 \quad 3\,]^T$

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 solution 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. In order to illustrate