

Jaán Kiusalaas

Numerical Methods in Engineering WITH Python

CAMBRIDGE

This page intentionally left blank

Numerical Methods in Engineering with Python

Numerical Methods in Engineering with Python is a text for engineering students and a reference for practicing engineers, especially those who wish to explore the power and efficiency of Python. 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. Computer code accompanies each method and is 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. Python was chosen as the example language because it is elegant, easy to learn and debug, and its facilities for handling arrays are unsurpassed. Moreover, it is an open-source software package that can be downloaded freely on the web. Python is a great language 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 computer 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 MATLAB[®] code.

**NUMERICAL METHODS IN
ENGINEERING WITH
Python**

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/9780521852876

© Jaan Kiusalaas 2005

This publication is in copyright. Subject to statutory exception and to the provision of relevant collective licensing agreements, no reproduction of any part may take place without the written permission of Cambridge University Press.

First published in print format 2005

ISBN-13 978-0-511-12675-8 eBook (Adobe Reader)

ISBN-10 0-511-12675-1 eBook (Adobe Reader)

ISBN-13 978-0-521-85287-6 hardback

ISBN-10 0-521-85287-0 hardback

Cambridge University Press has no responsibility for the persistence or accuracy of URLs for external or third-party internet websites referred to in this publication, and does not guarantee that any content on such websites is, or will remain, accurate or appropriate.

Contents

Preface vii

1. Introduction to Python	1
2. Systems of Linear Algebraic Equations	27
3. Interpolation and Curve Fitting	103
4. Roots of Equations	142
5. Numerical Differentiation	181
6. Numerical Integration	198
7. Initial Value Problems	248
8. Two-Point Boundary Value Problems	295
9. Symmetric Matrix Eigenvalue Problems	324
10. Introduction to Optimization	381

Appendices 409

Index.....419

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 mechanics 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 Python 2.2.2 and 2.3.4 under Windows XP and Red Hat Linux. The source code can be downloaded from the book's website at

www.cambridge.org/0521852870

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 Python

1.1 General Information

Quick Overview

This chapter is not a comprehensive manual of Python. Its sole aim is to provide sufficient information to give you a good start if you are unfamiliar with Python. If you know another computer language, and presumably you do, it is not difficult to pick up the rest as you go.

Python is an object-oriented language that was developed in late 1980s as a scripting language (the name is derived from the British television show Monty Python's Flying Circus). Although Python is not as well known in engineering circles as some other languages, it has a considerable following in the programming community—in fact, Python is considerably more widespread than Fortran. Python may be viewed as an emerging language, since it is still being developed and refined. In the current state, it is an excellent language for developing engineering applications—it possesses a simple elegance that other programming languages cannot match.

Python programs are not compiled into machine code, but are run by an *interpreter*¹. The great advantage of an interpreted language 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, Python programs can be developed in a much shorter time than equivalent Fortran or C programs. On the negative side, interpreted programs do not produce stand-alone applications. Thus a Python program can be run only on computers that have the Python interpreter installed.

¹ The Python interpreter also compiles *byte code*, which helps to speed up execution somewhat.

Python has other advantages over mainstream languages that are important in a learning environment:

- Python is open-source software, which means that it is *free*; it is included in most Linux distributions.
- Python is available for all major operating systems (Linux, Unix, Windows, Mac OS etc.). A program written on one system runs without modification on all systems.
- Python is easier to learn and produces more readable code than other languages.
- Python and its extensions are easy to install.

Development of Python was clearly influenced by Java and C++, but there is also a remarkable similarity to MATLAB[®] (another interpreted language, very popular in scientific computing). Python implements the usual concepts of object-oriented languages such as classes, methods, inheritance etc. We will forego these concepts and use Python strictly as a procedural language.

To get an idea of the similarities between MATLAB and Python, let us look at the codes written in the two languages for solution of simultaneous equations $Ax = b$ by Gauss elimination. Here is the function written in MATLAB:

```
function [x,det] = gaussElimin(a,b)
n = length(b);
for k = 1:n-1
    for i = k+1:n
        if a(i,k) ~= 0
            lam = a(i,k)/a(k,k);
            a(i,k+1:n) = a(i,k+1:n) - lam*a(k,k+1:n);
            b(i) = b(i) - lam*b(k);
        end
    end
end
det = prod(diag(a));
for k = n:-1:1
    b(k) = (b(k) - a(k,k+1:n)*b(k+1:n))/a(k,k);
end
x = b;
```

The equivalent Python function is:

```
from numpy import dot
def gaussElimin(a,b):
    n = len(b)
```

```

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

```

The command `from numarray import dot` instructs the interpreter to load the function `dot` (which computes the dot product of two vectors) from the module `numarray`. The colon (`:`) operator, known as the *slicing operator* in Python, works the same way it does in MATLAB and Fortran90—it defines a section of an array.

The statement `for k = 1:n-1` in MATLAB creates a loop that is executed with $k = 1, 2, \dots, n-1$. The same loop appears in Python as `for k in range(n-1)`. Here the function `range(n-1)` creates the list $[0, 1, \dots, n-2]$; k then loops over the elements of the list. The differences in the ranges of k reflect the native offsets used for arrays. In Python all sequences have *zero offset*, meaning that the index of the first element of the sequence is always 0. In contrast, the native offset in MATLAB is 1.

Also note that Python has no end statements to terminate blocks of code (loops, conditionals, subroutines etc.). The body of a block is defined by its *indentation*; hence indentation is an integral part of Python syntax.

Like MATLAB, Python is *case sensitive*. Thus the names n and N would represent different objects.

Obtaining Python

Python interpreter can be downloaded from the Python Language Website www.python.org. It normally comes with a nice code editor called *Idle* that allows you to run programs directly from the editor. For scientific programming we also need the *Numarray* module which contains various tools for array operations. It is obtainable from the Numarray Home Page http://www.stsci.edu/resources/software_hardware/numarray. Both sites also provide documentation for downloading. If you use Linux or Mac OS, it is very likely that Python is already installed on your machine (but you must still download Numarray).

You should acquire other printed material to supplement the on-line documentation. A commendable teaching guide is *Python* by Chris Fehly, Peachpit Press, CA (2002). As a reference, *Python Essential Reference* by David M. Beazley, New Riders

Publishing (2001) is recommended. By the time you read this, newer editions may be available.

1.2 Core Python

Variables

In most computer languages the name of a variable represents a value of a given type stored in a fixed memory location. The value may be changed, but not the type. This is not so in Python, where variables are *typed dynamically*. The following interactive session with the Python interpreter illustrates this (>>> is the Python prompt):

```
>>> b = 2          # b is integer type
>>> print b
2
>>> b = b * 2.0    # Now b is float type
>>> print b
4.0
```

The assignment `b = 2` creates an association between the name `b` and the *integer* value 2. The next statement evaluates the expression `b * 2.0` and associates the result with `b`; the original association with the integer 2 is destroyed. Now `b` refers to the *floating point* value 4.0.

The pound sign (`#`) denotes the beginning of a *comment*—all characters between `#` and the end of the line are ignored by the interpreter.

Strings

A string is a sequence of characters enclosed in single or double quotes. Strings are *concatenated* with the plus (+) operator, whereas *slicing* (:) is used to extract a portion of the string. Here is an example:

```
>>> string1 = 'Press return to exit'
>>> string2 = 'the program'
>>> print string1 + ' ' + string2  # Concatenation
Press return to exit the program
>>> print string1[0:12]           # Slicing
Press return
```

A string is an *immutable* object—its individual characters cannot be modified with an assignment statement and it has a fixed length. An attempt to violate immutability will result in `TypeError`, as shown below.

```
>>> s = 'Press return to exit'
>>> s[0] = 'p'
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in ?
    s[0] = 'p'
TypeError: object doesn't support item assignment
```

Tuples

A *tuple* is a sequence of *arbitrary objects* separated by commas and enclosed in parentheses. If the tuple contains a single object, the parentheses may be omitted. Tuples support the same operations as strings; they are also immutable. Here is an example where the tuple `rec` contains another tuple `(6, 23, 68)`:

```
>>> rec = ('Smith', 'John', (6, 23, 68))    # This is a tuple
>>> lastName, firstName, birthdate = rec    # Unpacking the tuple
>>> print firstName
John
>>> birthYear = birthdate[2]
>>> print birthYear
68
>>> name = rec[1] + ' ' + rec[0]
>>> print name
John Smith
>>> print rec[0:2]
('Smith', 'John')
```

Lists

A list is similar to a tuple, but it is *mutable*, so that its elements and length can be changed. A list is identified by enclosing it in brackets. Here is a sampling of operations that can be performed on lists:

```
>>> a = [1.0, 2.0, 3.0]                # Create a list
>>> a.append(4.0)                      # Append 4.0 to list
>>> print a
[1.0, 2.0, 3.0, 4.0]
```

```
>>> a.insert(0,0.0)           # Insert 0.0 in position 0
>>> print a
[0.0, 1.0, 2.0, 3.0, 4.0]
>>> print len(a)             # Determine length of list
5
>>> a[2:4] = [1.0, 1.0] # Modify selected elements
>>> print a
[0.0, 1.0, 1.0, 1.0, 1.0, 4.0]
```

If *a* is a mutable object, such as a list, the assignment statement *b* = *a* does not result in a new object *b*, but simply creates a new reference to *a*. Thus any changes made to *b* will be reflected in *a*. To create an independent copy of a list *a*, use the statement *c* = *a*[:], as illustrated below.

```
>>> a = [1.0, 2.0, 3.0]
>>> b = a                # 'b' is an alias of 'a'
>>> b[0] = 5.0           # Change 'b'
>>> print a
[5.0, 2.0, 3.0]          # The change is reflected in 'a'
>>> c = a[:]              # 'c' is an independent copy of 'a'
>>> c[0] = 1.0           # Change 'c'
>>> print a
[5.0, 2.0, 3.0]          # 'a' is not affected by the change
```

Matrices can be represented as nested lists with each row being an element of the list. Here is a 3×3 matrix *a* in the form of a list:

```
>>> a = [[1, 2, 3], \
          [4, 5, 6], \
          [7, 8, 9]]
>>> print a[1]           # Print second row (element 1)
[4, 5, 6]
>>> print a[1][2]        # Print third element of second row
6
```

The backslash (\) is Python's *continuation character*. Recall that Python sequences have zero offset, so that *a*[0] represents the first row, *a*[1] the second row, etc. With very few exceptions we do not use lists for numerical arrays. It is much more convenient

to employ *array objects* provided by the `numarray` module, (an extension of Python language). Array objects will be discussed later.

Arithmetic Operators

Python supports the usual arithmetic operators:

+	Addition
−	Subtraction
*	Multiplication
/	Division
**	Exponentiation
%	Modular division

Some of these operators are also defined for strings and sequences as illustrated below.

```
>>> s = 'Hello '
>>> t = 'to you'
>>> a = [1, 2, 3]
>>> print 3*s                # Repetition
Hello Hello Hello
>>> print 3*a                # Repetition
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> print a + [4, 5]         # Append elements
[1, 2, 3, 4, 5]
>>> print s + t              # Concatenation
Hello to you
>>> print 3 + s              # This addition makes no sense
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in ?
    print n + s
TypeError: unsupported operand types for +: 'int' and 'str'
```

Python 2.0 and later versions also have *augmented assignment operators*, such as $a += b$, that are familiar to the users of C. The augmented operators and the equivalent arithmetic expressions are shown in the following table.

<code>a += b</code>	<code>a = a + b</code>
<code>a -= b</code>	<code>a = a - b</code>
<code>a *= b</code>	<code>a = a*b</code>
<code>a /= b</code>	<code>a = a/b</code>
<code>a **= b</code>	<code>a = a**b</code>
<code>a %= b</code>	<code>a = a%b</code>

Comparison Operators

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

<code><</code>	Less than
<code>></code>	Greater than
<code><=</code>	Less than or equal to
<code>>=</code>	Greater than or equal to
<code>==</code>	Equal to
<code>!=</code>	Not equal to

Numbers of different type (integer, floating point etc.) are converted to a common type before the comparison is made. Otherwise, objects of different type are considered to be unequal. Here are a few examples:

```
>>> a = 2           # Integer
>>> b = 1.99        # Floating point
>>> c = '2'         # String
>>> print a > b
1
>>> print a == c
0
>>> print (a > b) and (a != c)
1
>>> print (a > b) or (a == b)
1
```

Conditionals

The `if` construct

```
if condition:  
    block
```

executes a block of statements (which must be indented) if the condition returns true. If the condition returns false, the block is skipped. The `if` conditional can be followed by any number of `elif` (short for “else if”) constructs

```
elif condition:  
    block
```

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

```
else:  
    block
```

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

```
def sign_of_a(a):  
    if a < 0.0:  
        sign = 'negative'  
    elif a > 0.0:  
        sign = 'positive'  
    else:  
        sign = 'zero'  
    return sign
```

```
a = 1.5  
print 'a is ' + sign_of_a(a)
```

Running the program results in the output

```
a is positive
```

Loops

The `while` construct

```
while condition:  
    block
```

executes a block of (indented) statements if the condition is true. After execution of the block, the condition is evaluated again. If it is still true, the block is executed again. This process is continued until the condition becomes false. The `else` clause

```
else:  
    block
```

can be used to define the block of statements which are to be executed if condition is false. Here is an example that creates the list `[1, 1/2, 1/3, ...]`:

```
nMax = 5  
n = 1  
a = []          # Create empty list  
while n < nMax:  
    a.append(1.0/n) # Append element to list  
    n = n + 1  
print a
```

The output of the program is

```
[1.0, 0.5, 0.33333333333333331, 0.25]
```

We met the `for` statement before in Art. 1.1. This statement requires a target and a sequence (usually a list) over which the target loops. The form of the construct is

```
for target in sequence:  
    block
```

You may add an `else` clause which is executed after the `for` loop has finished. The previous program could be written with the `for` construct as

```
nMax = 5
a = []
for n in range(1,nMax):
    a.append(1.0/n)
print a
```

Here n is the target and the list $[1, 2, \dots, nMax-1]$, created by calling the range function, is the sequence.

Any loop can be terminated by the break statement. If there is an else cause associated with the loop, it is not executed. The following program, which searches for a name in a list, illustrates the use of break and else in conjunction with a for loop:

```
list = ['Jack', 'Jill', 'Tim', 'Dave']
name = eval(raw_input('Type a name: ')) # Python input prompt
for i in range(len(list)):
    if list[i] == name:
        print name, 'is number', i + 1, 'on the list'
        break
else:
    print name, 'is not on the list'
```

Here are the results of two searches:

```
Type a name: 'Tim'
Tim is number 3 on the list
```

```
Type a name: 'June'
June is not on the list
```

Type Conversion

If an arithmetic operation involves numbers of mixed types, the numbers are automatically converted to a common type before the operation is carried out. Type conversions can also be achieved by the following functions:

<code>int(a)</code>	Converts a to integer
<code>long(a)</code>	Converts a to long integer
<code>float(a)</code>	Converts a to floating point
<code>complex(a)</code>	Converts to complex $a + 0j$
<code>complex(a,b)</code>	Converts to complex $a + bj$

The above functions also work for converting strings to numbers as long as the literal in the string represents a valid number. Conversion from float to an integer is carried out by truncation, not by rounding off. Here are a few examples:

```
>>> a = 5
>>> b = -3.6
>>> d = '4.0'
>>> print a + b
1.4
>>> print int(b)
-3
>>> print complex(a,b)
(5-3.6j)
>>> print float(d)
4.0
>>> print int(d) # This fails: d is not Int type
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in ?
    print int(d)
ValueError: invalid literal for int(): 4.0
```

Mathematical Functions

Core Python supports only a few mathematical functions. They are:

<code>abs(a)</code>	Absolute value of a
<code>max(sequence)</code>	Largest element of <i>sequence</i>
<code>min(sequence)</code>	Smallest element of <i>sequence</i>
<code>round(a,n)</code>	Round a to n decimal places
<code>cmp(a,b)</code>	Returns $\begin{cases} -1 & \text{if } a < b \\ 0 & \text{if } a = b \\ 1 & \text{if } a > b \end{cases}$

The majority of mathematical functions are available in the `math` module.

Reading Input

The intrinsic function for accepting user input is

```
raw_input(prompt)
```

It displays the prompt and then reads a line of input which is converted to a *string*. To convert the string into a numerical value use the function

```
eval(string)
```

The following program illustrates the use of these functions:

```
a = raw_input('Input a: ')
print a, type(a)           # Print a and its type
b = eval(a)
print b, type(b)           # Print b and its type
```

The function `type(a)` returns the type of the object *a*; it is a very useful tool in debugging. The program was run twice with the following results:

```
Input a: 10.0
10.0 <type 'str'>
10.0 <type 'float'>
```

```
Input a: 11**2
11**2 <type 'str'>
121 <type 'int'>
```

A convenient way to input a number and assign it to the variable *a* is

```
a = eval(raw_input(prompt))
```

Printing Output

Output can be displayed with the print statement:

```
print object1, object2, ...
```

which converts *object1*, *object2* etc. to strings and prints them on the same line, separated by spaces. The *newline* character '`\n`' can be used to force a new line. For example,

```
>>> a = 1234.56789
>>> b = [2, 4, 6, 8]
>>> print a,b
1234.56789 [2, 4, 6, 8]
>>> print 'a =',a, '\nb =',b
a = 1234.56789
b = [2, 4, 6, 8]
```

The *modulo operator* (%) can be used to format a tuple. The form of the conversion statement is

```
'%format1 %format2 ...' % tuple
```

where *format1*, *format2* ... are the format specifications for each object in the tuple. Typically used format specifications are

<i>wd</i>	Integer
<i>w.df</i>	Floating point notation
<i>w.de</i>	Exponential notation

where *w* is the width of the field and *d* is the number of digits after the decimal point. The output is right-justified in the specified field and padded with blank spaces (there are provisions for changing the justification and padding). Here are a couple of examples:

```
>>> a = 1234.56789
>>> n = 9876
>>> print '%7.2f' % a
1234.57
>>> print 'n = %6d' % n # Pad with 2 spaces
n = 9876
```



```
>>> print 'n = %06d' %n # Pad with 2 zeroes
n = 009876
>>> print '%12.4e %6d' % (a,n)
1.2346e+003 9876
```

Error Control

When an error occurs during execution of a program an *exception* is raised and the program stops. Exceptions can be caught with `try` and `except` statements:

```
try:
    do something
except error:
    do something else
```

where *error* is the name of a built-in Python exception. If the exception *error* is not raised, the `try` block is executed; otherwise the execution passes to the `except` block. All exceptions can be caught by omitting *error* from the `except` statement.

Here is a statement that raises the exception `ZeroDivisionError`:

```
>>> c = 12.0/0.0
Traceback (most recent call last):
  File '<pyshell#0>', line 1, in ?
    c = 12.0/0.0
ZeroDivisionError: float division
```

This error can be caught by

```
try:
    c = 12.0/0.0
except ZeroDivisionError:
    print 'Division by zero'
```

1.3 Functions and Modules

Functions

The structure of a Python function is

```
def func_name(param1, param2,...):
    statements
    return return_values
```

where *param1*, *param2*,... are the parameters. A parameter can be any Python object, including a function. Parameters may be given default values, in which case the parameter in the function call is optional. If the `return` statement or *return-values* are omitted, the function returns the null object.

The following example computes the first two derivatives of $\arctan(x)$ by finite differences:

```
from math import arctan
def finite_diff(f,x,h=0.0001):    # h has a default value
    df =(f(x+h) - f(x-h))/(2.0*h)
    ddf =(f(x+h) - 2.0*f(x) + f(x-h))/h**2
    return df,ddf
x = 0.5
df,ddf = finite_diff(arctan,x)    # Uses default value of h
print 'First derivative  =',df
print 'Second derivative =',ddf
```

Note that `arctan` is passed to `finite_diff` as a parameter. The output from the program is

```
First derivative  = 0.799999999573
Second derivative = -0.6399999991892
```

If a mutable object, such as a list, is passed to a function where it is modified, the changes will also appear in the calling program. Here is an example:

```
def squares(a):
    for i in range(len(a)):
        a[i] = a[i]**2

a = [1, 2, 3, 4]
squares(a)
print a
```

The output is

```
[1, 4, 9, 16]
```

Modules

It is sound practice to store useful functions in modules. A module is simply a file where the functions reside; the name of the module is the name of the file. A module can be loaded into a program by the statement

```
from module_name import *
```

Python comes with a large number of modules containing functions and methods for various tasks. Two of the modules are described briefly in the next section. Additional modules, including graphics packages, are available for downloading on the Web.

1.4 Mathematics Modules

`math` Module

Most mathematical functions are not built into core Python, but are available by loading the `math` module. There are three ways of accessing the functions in a module. The statement

```
from math import *
```

loads *all* the function definitions in the `math` module into the current function or module. The use of this method is discouraged because it is not only wasteful, but can also lead to conflicts with definitions loaded from other modules.

You can load selected definitions by

```
from math import func1, func2, ...
```

as illustrated below.

```
>>> from math import log,sin
>>> print log(sin(0.5))
-0.735166686385
```

The third method, which is used by the majority of programmers, is to make the module available by

```
import math
```

The functions in the module can then be accessed by using the module name as a prefix:

```
>>> import math
>>> print math.log(math.sin(0.5))
-0.735166686385
```

The contents of a module can be printed by calling `dir(module)`. Here is how to obtain a list of the functions in the `math` module:

```
>>> import math
>>> dir(math)
['__doc__', '__name__', 'acos', 'asin', 'atan',
 'atan2', 'ceil', 'cos', 'cosh', 'e', 'exp', 'fabs',
 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
 'log10', 'modf', 'pi', 'pow', 'sin', 'sinh', 'sqrt',
 'tan', 'tanh']
```

Most of these functions are familiar to programmers. Note that the module includes two constants: π and e .

cmath Module

The `cmath` module provides many of the functions found in the `math` module, but these accept complex numbers. The functions in the module are:

```
['__doc__', '__name__', 'acos', 'acosh', 'asin', 'asinh',
 'atan', 'atanh', 'cos', 'cosh', 'e', 'exp', 'log',
 'log10', 'pi', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
```

Here are examples of complex arithmetic:

```
>>> from cmath import sin
>>> x = 3.0 -4.5j
>>> y = 1.2 + 0.8j
>>> z = 0.8
```

```
>>> print x/y
(-2.56205313375e-016-3.75j)
>>> print sin(x)
(6.35239299817+44.5526433649j)
>>> print sin(z)
(0.7173560909+0j)
```

1.5 numarray Module

General Information

The numarray module² is not a part of the standard Python release. As pointed out before, it must be obtained separately and installed (the installation is very easy). The module introduces *array objects* which are similar to lists, but can be manipulated by numerous functions contained in the module. The size of the array is immutable and no empty elements are allowed.

The complete set of functions in numarray is too long to be printed in its entirety. The list below is limited to the most commonly used functions.

```
['Complex', 'Complex32', 'Complex64', 'Float',
 'Float32', 'Float64', 'abs', 'arccos',
 'arccosh', 'arcsin', 'arcsinh', 'arctan',
 'arctan2', 'arctanh', 'argmax', 'argmin',
 'cos', 'cosh', 'diagonal', 'dot', 'e', 'exp',
 'floor', 'identity', 'innerproduct', 'log',
 'log10', 'matrixmultiply', 'maximum', 'minimum',
 'numarray', 'ones', 'pi', 'product', 'sin', 'sinh',
 'size', 'sqrt', 'sum', 'tan', 'tanh', 'trace',
 'transpose', 'zeros']
```

Creating an Array

Arrays can be created in several ways. One of them is to use the array function to turn a list into an array:

```
array(list, type = type-specification)
```

² Numarray is based on an older Python array module called *Numeric*. Their interfaces and capabilities are very similar and they are largely compatible. Although *Numeric* is still available, it is no longer supported.

Here are two examples of creating a 2×2 array with floating-point elements:

```
>>> from numarray import array,Float
>>> a = array([[2.0, -1.0],[-1.0, 3.0]])
>>> print a
[[ 2. -1.]
 [-1.  3.]]
>>> b = array([[2, -1],[-1, 3]],type = Float)
>>> print b
[[ 2. -1.]
 [-1.  3.]]
```

Other available functions are

```
zeros((dim1,dim2),type = type_specification)
```

which creates a $dim1 \times dim2$ array and fills it with zeroes, and

```
ones((dim1,dim2),type = type_specification)
```

which fills the array with ones. The default type in both cases is Int.

Finally, there is the function

```
arange(from,to,increment)
```

which works just like the range function, but returns an array rather than a list. Here are examples of creating arrays:

```
>>> from numarray import arange,zeros,ones,Float
>>> a = arange(2,10,2)
>>> print a
[2 4 6 8]
>>> b = arange(2.0,10.0,2.0)
>>> print b
[ 2.  4.  6.  8.]
>>> z = zeros((4))
>>> print z
[0 0 0 0]
```

```
>>> y = ones((3,3),type= Float)
>>> print y
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
```

Accessing and Changing Array Elements

If a is a rank-2 array, then $a[i, j]$ accesses the element in row i and column j , whereas $a[i]$ refers to row i . The elements of an array can be changed by assignment as shown below.

```
>>> from numpy import *
>>> a = zeros((3,3),type=Float)
>>> a[0] = [2.0, 3.1, 1.8] # Change a row
>>> a[1,1] = 5.2           # Change an element
>>> a[2,0:2] = [8.0, -3.3] # Change part of a row
>>> print a
[[ 2.   3.1  1.8]
 [ 0.   5.2  0. ]
 [ 8.  -3.3  0. ]]
```

Operations on Arrays

Arithmetic operators work differently on arrays than they do on tuples and lists—the operation is *broadcast* to all the elements of the array; that is, the operation is applied to each element in the array. Here are examples:

```
>>> from numpy import array
>>> a = array([0.0, 4.0, 9.0, 16.0])
>>> print a/16.0
[ 0.   0.25  0.5625  1.   ]
>>> print a - 4.0
[ -4.   0.   5.  12.]
```

The mathematical functions available in `numpy` are also broadcast, as illustrated below

```
>>> from numpy import array,sqrt,sin
>>> a = array([1.0, 4.0, 9.0, 16.0])
```

```
>>> print sqrt(a)
[ 1.  2.  3.  4.]
>>> print sin(a)
[ 0.84147098 -0.7568025  0.41211849 -0.28790332]
```

Functions imported from the `math` module will work on the individual elements, of course, but not on the array itself. Here is an example:

```
>>> from numarray import array
>>> from math import sqrt
>>> a = array([1.0, 4.0, 9.0, 16.0])
>>> print sqrt(a[1])
2.0
>>> print sqrt(a)
Traceback (most recent call last):
    :
    :
```

`TypeError: Only rank-0 arrays can be cast to floats.`

Array Functions

There are numerous array functions in `numarray` that perform matrix operations and other useful tasks. Here are a few examples:

```
>>> from numarray import *
>>> a = array([[ 4.0, -2.0,  1.0], \
               [-2.0,  4.0, -2.0], \
               [ 1.0, -2.0,  3.0]])
>>> b = array([1.0, 4.0, 2.0])
>>> print dot(b,b)           # Dot product
21.0
>>> print matrixmultiply(a,b) # Matrix multiplication
[-2. 10. -1.]
>>> print diagonal(a)        # Principal diagonal
[ 4.  4.  3.]
>>> print diagonal(a,1)       # First subdiagonal
[-2. -2.]
>>> print trace(a)           # Sum of diagonal elements
11.0
```



```
>>> print argmax(b)           # Index of largest element
1
>>> print identity(3)         # Identity matrix
[[1 0 0]
 [0 1 0]
 [0 0 1]]
```

Copying Arrays

We explained before that if *a* is a mutable object, such as a list, the assignment statement `b = a` does not result in a new object *b*, but simply creates a new reference to *a*, called a *deep copy*. This also applies to arrays. To make an independent copy of an array *a*, use the `copy` method in the `numarray` module:

```
b = a.copy()
```

1.6 Scoping of Variables

Namespace is a dictionary that contains the names of the variables and their values. The namespaces are automatically created and updated as a program runs. There are three levels of namespaces in Python:

- Local namespace, which is created when a function is called. It contains the variables passed to the function as arguments and the variables created within the function. The namespace is deleted when the function terminates. If a variable is created inside a function, its scope is the function's local namespace. It is not visible outside the function.
- A global namespace is created when a module is loaded. Each module has its own namespace. Variables assigned in a global namespace are visible to any function within the module.
- Built-in namespace is created when the interpreter starts. It contains the functions that come with the Python interpreter. These functions can be accessed by any program unit.

When a name is encountered during execution of a function, the interpreter tries to resolve it by searching the following in the order shown: (1) local namespace, (2) global namespace, and (3) built-in namespace. If the name cannot be resolved, Python raises a `NameError` exception.

Since the variables residing in a global namespace are visible to functions within the module, it is not necessary to pass them to the functions as arguments (although is good programming practice to do so), as the following program illustrates.

```
def divide():
    c = a/b
    print 'a/b =',c

a = 100.0
b = 5.0
divide()
>>>
a/b = 20.0
```

Note that the variable `c` is created inside the function `divide` and is thus not accessible to statements outside the function. Hence an attempt to move the `print` statement out of the function fails:

```
def divide():
    c = a/b

a = 100.0
b = 5.0
divide()
print 'a/b =',c

>>>
Traceback (most recent call last):
  File "'C:\Python22\scope.py'", line 8, in ?
    print c
NameError: name 'c' is not defined
```

1.7 Writing and Running Programs

When the Python editor *Idle* is opened, the user is faced with the prompt `>>>`, indicating that the editor is in interactive mode. Any statement typed into the editor is immediately processed upon pressing the enter key. The interactive mode is a good way to learn the language by experimentation and to try out new programming ideas.

Opening a new window places *Idle* in the batch mode, which allows typing and saving of programs. One can also use a text editor to enter program lines, but *Idle* has Python-specific features, such as color coding of keywords and automatic indentation, that make work easier. Before a program can be run, it must be saved as a Python file with the `.py` extension, e.g., `myprog.py`. The program can then be executed by typing

`python myprog.py`; in Windows, double-clicking on the program icon will also work. But beware: the program window closes immediately after execution, before you get a chance to read the output. To prevent this from happening, conclude the program with the line

```
raw_input('press return')
```

Double-clicking the program icon also works in Unix and Linux if the first line of the program specifies the path to the Python interpreter (or a shell script that provides a link to Python). The path name must be preceded by the symbols `#!`. On my computer the path is `/usr/bin/python`, so that all my programs start with the line

```
#!/usr/bin/python
```

On multiuser systems the path is usually `/usr/local/bin/python`.

When a module is loaded into a program for the first time with the `import` statement, it is compiled into bytecode and written in a file with the extension `.pyc`. The next time the program is run, the interpreter loads the bytecode rather than the original Python file. If in the meantime changes have been made to the module, the module is automatically recompiled. A program can also be run from Idle using *edit/run script* menu, but automatic recompilation of modules will not take place, unless the existing bytecode file is deleted and the program window is closed and reopened.

Python's error messages can sometimes be confusing, as seen in the following example:

```
from numpy import array
a = array([1.0, 2.0, 3.0])
print a
raw_input('press return')
```

The output is

```
File 'C:\Python22\test_module.py', line 3
    print a
    ^
SyntaxError: invalid syntax
```

What could possibly be wrong with the line `print a`? The answer is nothing. The problem is actually in the preceding line, where the closing parenthesis is missing,

making the statement incomplete. Consequently, the interpreter views the third line as continuation of the second line, so that it tries to interpret the statement

```
a = array([1.0, 2.0, 3.0])print a
```

The lesson is this: when faced with a `SyntaxError`, look at the line preceding the alleged offender. It can save a lot of frustration.

It is a good idea to document your modules by adding a *docstring* the beginning of each module. The docstring, which is enclosed in triple quotes, should explain what the module does. Here is an example that documents the module `error` (we use this module in several of our programs):

```
## module error
''' err(string).
    Prints 'string' and terminates program.
'''
import sys
def err(string):
    print string
    raw_input('Press return to exit')
    sys.exit()
```

The docstring of a module can be printed with the statement

```
print module_name.__doc__
```

For example, the docstring of `error` is displayed by

```
>>> import error
>>> print error.__doc__
err(string).
    Prints 'string' and terminates program.
```

2 Systems of Linear Algebraic Equations

Solve the simultaneous equations $Ax = b$

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 is 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 Fortran⁷⁷³.

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

$$A_{11}x_1 + A_{12}x_2 + \cdots + A_{1n}x_n = b_1$$

³ LAPACK is the successor of LINPACK, a 1970s and 80s collection of Fortran subroutines.

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

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

$$\mathbf{Ax} = \mathbf{b} \tag{2.3}$$

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:

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

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*; that is, $|\mathbf{A}| \neq 0$. The rows and columns of a nonsingular matrix are *linearly independent* in the sense that no row (or column) is a linear combination of other rows (or columns).

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

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

Since the second equation can be obtained by multiplying the first equation by two, any combination of x and y that satisfies the first equation is also a solution of the

second equation. The number of such combinations is infinite. On the other hand, the equations

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

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.

III-Conditioning

An obvious question is: what happens when the coefficient matrix is almost singular; i.e., if $|A|$ 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 and is denoted by $\|A\|$. We can then say that the determinant is small if

$$|A| \ll \|A\|$$

Several norms of a matrix have been defined in existing literature, such as

$$\|A\| = \sqrt{\sum_{i=1}^n \sum_{j=1}^n A_{ij}^2} \quad \|A\| = \max_{1 \leq i \leq n} \sum_{j=1}^n |A_{ij}| \quad (2.5a)$$

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

$$\text{cond}(A) = \|A\| \|A^{-1}\| \quad (2.5b)$$

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

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

that have the solution $x = 1501.5$, $y = -3000$. Since $|A| = 2(1.001) - 2(1) = 0.002$ 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, \dots, x_n represent the forces in the members and the support reactions, and the constants b_1, b_2, \dots, 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 parenthesis.

- Exchanging two equations (changes sign of $|A|$).
- Multiplying an equation by a nonzero constant (multiplies $|A|$ by the same constant).
- Multiplying an equation by a nonzero constant and then subtracting it from another equation (leaves $|A|$ 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 contains 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 of the determinant (see Appendix A2) about the first row of A yields

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

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 $Ax = b$, where

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

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

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

Solution We first solve the equations $Ly = b$ 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 $Ux = y$ 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 $x = \begin{bmatrix} 2 & -1 & 3 \end{bmatrix}^T$

2.2 Gauss Elimination Method

Introduction

Gauss elimination is the most familiar method for solving simultaneous equations. It consists of two parts: the elimination phase and the solution phase. As indicated in Table 2.1, the function of the elimination phase is to transform the equations into the

form $\mathbf{U}\mathbf{x} = \mathbf{c}$. The equations are then solved by back substitution. In order to illustrate the procedure, let us solve the equations

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

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

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

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

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

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

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

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

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

After this transformation, the equations become

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

$$3x_2 - 1.5x_3 = -10.5 \quad (\text{b})$$

$$-1.5x_2 + 3.75x_3 = 14.25 \quad (\text{c})$$

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

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

which yields the equations

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

$$3x_2 - 1.5x_3 = -10.5 \quad (\text{b})$$

$$3x_3 = 9 \quad (\text{c})$$

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

As pointed out before, the augmented coefficient matrix is a more convenient instrument for performing the computations. Thus the original equations