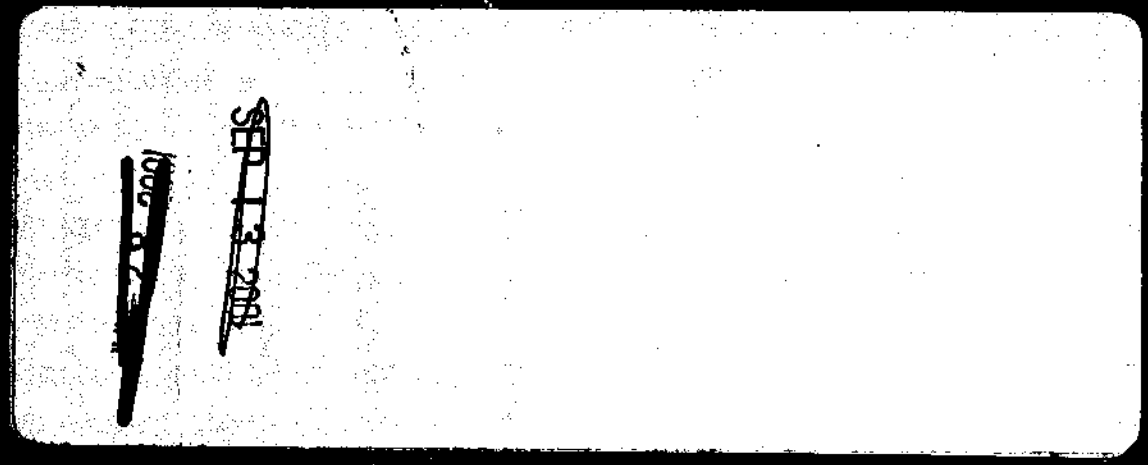
ADVANCED SCIENTIFIC FORTRAN



Advanced Scientific Fortran

David R. Willé

Mathematical Applications, Ciba-Geigy AG,
CH-4002 Basel, Switzerland

JOHN WILEY & SONS
Chichester · New York · Brisbane · Toronto · Singapore

Baffins Lane, Chichester
West Sussex PO19 1UD, England

Telephone *National* Chichester (01243) 779777
International (+44) 1243 779777

Other Wiley Editorial Offices

John Wiley & Sons, Inc., 605 Third Avenue, New York, NY 10158-0012, USA

Jacaranda Wiley Ltd, 33 Park Road, Milton, Queensland 4064, Australia

John Wiley & Sons (Canada) Ltd, 22 Worcester Road, Rexdale, Ontario M9W 1L1, Canada

John Wiley & Sons (SEA) Pte Ltd, 37 Jalan Pemimpin #05-04, Block B, Union Industrial Building, Singapore 2057

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN 0 471 95383 0

Produced from camera-ready copy supplied by the author using emTeX
Printed and bound in Great Britain by Redwood Books, Trowbridge, Wilts.

This book is printed on acid-free paper responsibly manufactured from sustainable forestation, for which at least two trees are planted for each one used for paper production.

Contents

Introduction xv
Advanced Scientific Fortran xv
Fortran 90 xv
Who should read it? xvi
Layout and solutions xvi
Notation xvi
Acknowledgements xvii

I Scientific Computing **1**

1 Double and Single Precision 3
1.1 Machine dependence 3
1.2 Type promotion 4
1.3 Portability 4
1.4 Double precision and software libraries 5
1.5 Machine representation 5
1.6 Fortran 90 5
1.7 Exercises 5

2 Portability and Standards 7
2.1 Program portability 7
2.2 The FORTRAN standard 7
2.3 Portability testers and pretty writers 8
2.4 Strong typing 8
2.5 Machine accuracy 8
2.6 Saving and file handling 10
2.7 The US military standard 10
2.8 BLAS and LAPACK 11
2.9 Comments and documentation 11

3 The NAG Library 13
3.1 Mathematical software 13
3.2 Why use software libraries? 13
3.3 Other libraries 14

3.4 Linear algebra . . . 14
3.5 Netlib . . . 15
3.6 Internet . . . 15
3.7 The NAG library . . . 15
3.7.1 An overview . . . 15
3.7.2 Using the NAG library – two examples . . . 16
3.8 Subroutine references . . . 21
3.8.1 D01AHF – integration of a definite integral . . . 21
3.8.2 C05ADF – a forward communication root finder . . . 22
3.8.3 C05AZF – a reverse communication root finder . . . 22
3.9 Exercises . . . 22

4 Save, Reverse Communication and Arrays . . . 25
4.1 Static and dynamic storage: FORTRAN and PASCAL . . . 25
4.2 Subroutine calls . . . 26
4.3 Recursion . . . 26
4.4 Disadvantages of dynamic storage . . . 27
4.5 The SAVE statement . . . 27
4.5.1 SAVE as the default . . . 28
4.5.2 Dangers with unwanted saving . . . 28
4.5.3 Efficiency . . . 28
4.5.4 Initialisation . . . 29
4.6 Reverse communication – an example . . . 29
4.7 Exercises . . . 32

5 Internal Files . . . 35
5.1 READ and WRITE statements . . . 35
5.2 Pull – an example program using internal files . . . 35
5.3 Exercises . . . 38

6 Arrays in FORTRAN . . . 39
6.1 'Row' and 'column' storage . . . 39
6.2 Calling arrays in FORTRAN . . . 40
6.3 The leading dimension . . . 42
6.4 Higher dimensional arrays and dummy dimensions . . . 42
6.5 A note of caution . . . 43
6.6 A harder example . . . 43
6.7 Exercises . . . 44

7 Work Arrays and Array Partitioning . . . 47
7.1 Passing storage . . . 47
7.2 Partitioning work arrays . . . 47

8 Memory Efficiency . . . 51
8.1 The memory problem . . . 51
8.2 Memory hierarchy . . . 51
8.3 Cache memory . . . 52

8.4 Data locality . . . 53
8.5 Memory stride . . . 54
8.6 Paging . . . 54
8.7 Other architectures . . . 54
8.8 Exercises . . . 55

9 Basic Linear Algebra Subprograms (BLAS) . . . 57
9.1 Introduction . . . 57
9.2 An overview . . . 58
9.3 Level one BLAS – SCOPY . . . 58
9.4 Negative strides – a caution . . . 59
9.5 Accessing rows and columns . . . 61
9.6 Exercises . . . 62

10 Linear Algebra and Vectorisation . . . 63
10.1 Higher level BLAS . . . 63
10.1.1 Why level three BLAS? . . . 63
10.1.2 Block algorithms . . . 64
10.1.3 High performance machines, parallel computing and sparse systems . . . 66
10.2 Vectorisation . . . 67
10.2.1 Pipelining . . . 67
10.2.2 Chaining . . . 69
10.2.3 Command pipelining and Risc architectures . . . 69
10.3 Using vectorisation . . . 70
10.3.1 When is it desirable? . . . 70
10.3.2 How to vectorise a loop . . . 70
10.4 Level one BLAS . . . 70
10.5 Exercises . . . 71

11 Gaussian Elimination . . . 73
11.1 A simple example . . . 73
11.2 A harder example . . . 75
11.3 Interpretation . . . 77
11.4 Conclusion . . . 77
11.5 Subroutine references . . . 78
11.5.1 DGER – a one-rank update of a general matrix . . . 78
11.5.2 DSCAL – scales a vector by a constant . . . 78

12 LAPACK . . . 81
12.1 Introduction . . . 81
12.2 Driver routines, computational routines and auxiliaries . . . 81
12.3 Driver routines . . . 82
12.4 Computational routines . . . 84
12.5 Other problems . . . 85

13 Programming Considerations . . . 87

13.1 FORTRAN and other languages . . . 87
13.2 Strong typing . . . 87
13.3 Modules and packages . . . 88
13.4 'Goto's considered harmful' . . . 88
13.4.1 Repeaters and completers . . . 89
13.4.2 Goto's in FORTRAN . . . 89
13.5 RETURN and ENTRY statements . . . 90
13.6 On and computed goto's . . . 90
13.7 DO-loops and CONTINUE statements . . . 91
13.7.1 DO-END DO . . . 91
13.7.2 Loop variables . . . 91
13.8 Passing constant parameters – an important note . . . 92

14 Code Optimisation . . . 93
14.1 Simple techniques . . . 93
14.1.1 Polynomials . . . 93
14.1.2 Statement functions . . . 93
14.1.3 In-line expansion . . . 94
14.1.4 Program substitution . . . 95
14.1.5 Type conversions and common subexpression enhancement . . . 97
14.1.6 Character variables . . . 97
14.1.7 DATA and parameters . . . 98
14.1.8 COMMON . . . 98
14.1.9 Unformatted input/output . . . 98
14.1.10 Input/output calls . . . 98
14.1.11 IF statements . . . 99
14.2 DO-loops . . . 99
14.2.1 Invariant code . . . 99
14.2.2 Loop unrolling and nesting . . . 100
14.2.3 Loop fusion . . . 101
14.2.4 Avoiding type conversion . . . 101
14.2.5 Loop collapse . . . 102
14.3 Calling the compiler . . . 103

15 Input/Output . . . 105
15.1 A generic system . . . 105
15.2 Unix systems – pipelines . . . 108
15.3 Exercises . . . 112

II Fortran 90 and High Performance Fortran **115**

16 Fortran 90 – Introduction . . . 117
16.1 What is Fortran 90? . . . 117
16.2 Language continuity . . . 117
16.3 Other dialects . . . 118
16.4 Free-form source codes . . . 118

16.5 Implicit none . . . 119

17 Subroutines and Modules . . . 121
17.1 Internal routines . . . 121
17.2 Modules . . . 122
17.2.1 Controlling access . . . 123
17.2.2 Data modules . . . 126
17.3 Interfaces . . . 126
17.3.1 Interface blocks and overloading . . . 127
17.3.2 Interfaces and INTENT . . . 128
17.3.3 Optional and keyword parameters . . . 129
17.4 Recursion . . . 130
17.5 Exercises . . . 131

18 Types and Pointers . . . 133
18.1 Intrinsic types . . . 133
18.1.1 Real numbers . . . 133
18.1.2 Integers, characters and logical variables . . . 134
18.1.3 Constants . . . 134
18.1.4 Short notation . . . 135
18.2 Derived types . . . 135
18.3 Operator overloading . . . 136
18.4 Pointers . . . 137
18.4.1 Targets and ALLOCATE . . . 137
18.4.2 A linked list . . . 139
18.5 Exercises . . . 141

19 Arrays . . . 143
19.1 Dynamic storage allocation . . . 143
19.2 Elemental operations and array constructors . . . 145
19.2.1 Arithmetic operators . . . 145
19.2.2 Intrinsic functions and logical relations . . . 146
19.2.3 Array constructors . . . 147
19.2.4 RESHAPE . . . 148
19.3 Array indices . . . 149
19.4 Array sections . . . 149
19.5 An example . . . 150
19.6 Fortran 90 and the BLAS . . . 151
19.7 Intrinsic routines . . . 152
19.8 Exercises . . . 152

20 Control Structures . . . 155
20.1 Labels and DO-loops . . . 155
20.2 The select-case construct . . . 157
20.3 Mask arrays . . . 158
20.4 WHERE . . . 158

21 Input/Output in Fortran 90 . . . 161
21.1 Records and character streams . . . 161
21.2 NAMELIST . . . 162
21.3 Internal files . . . 163
21.4 OPEN and INQUIRE . . . 163
21.5 New format editors . . . 163

22 High Performance Fortran . . . 165
22.1 Motivation . . . 165
22.2 The data model . . . 166
22.2.1 Defining the processors . . . 166
22.2.2 Distributing the data . . . 166
22.2.3 Communication . . . 167
22.2.4 Alignment . . . 167
22.3 Identifying parallelism . . . 168
22.3.1 FORALL . . . 168
22.3.2 Independence . . . 170
22.4 Library and extrinsic routines . . . 170
22.4.1 The HPF_LIBRARY . . . 170
22.4.2 Extrinsic routines . . . 170

Appendix A The BLAS . . . 173
A.1 The Basic Linear Algebra Subprograms . . . 175
A.2 Level 1 BLAS . . . 175
A.3 Level 2 BLAS . . . 176
A.3.1 Matrix-vector operations . . . 176
A.3.2 Vector-matrix operations . . . 177
A.4 Level 3 BLAS . . . 177

Appendix B Intrinsic Functions in Fortran 90 . . . 179
B.1 Numerical and array routines . . . 181
B.1.1 Basic mathematical functions . . . 181
B.1.2 Functions for use with the KIND-attribute . . . 181
B.1.3 Array operations . . . 182
B.1.4 Numerical accuracy and type enquiry functions . . . 183
B.1.5 Enquiry function for pointer variables . . . 183
B.2 Non-numerical routines . . . 183
B.2.1 Character handling . . . 183
B.2.2 Bit manipulation and data conversion . . . 183
B.2.3 Time and random numbers . . . 184

Appendix C Solutions to Exercises . . . 185
C.1 **Part I – Scientific Computing** . . . 187
Chapter 1 – Double and Single Precision . . . 187
Chapter 3 – The NAG Library . . . 190
Chapter 4 – Save, Reverse Communication and Arrays . . . 195
Chapter 5 – Internal Files . . . 201

Chapter 6 – Arrays in FORTRAN 203
Chapter 8 – Memory Efficiency 205
Chapter 9 – Basic Linear Algebra Subprograms (BLAS) 207
Chapter 10 – Linear Algebra and Vectorisation 208
Chapter 15 – Input/Output 211
C.2 **Part II – Fortran 90 and High Performance Fortran** 214
Chapter 17 – Subroutines and Modules 214
Chapter 18 – Types and Pointers 218
Chapter 19 – Arrays 224

References 227

Index 229

Introduction

Advanced Scientific Fortran

Fortran, through its large body of accumulated software and built up experience, is one of the principal languages for scientific and numerical computation. Indeed, in many areas, notably where high performance is required, it is the language of choice leaving others like *C* and *C++* a poor second. This book, which assumes a prior basic knowledge of Fortran programming, sets out to introduce an ordinary programmer to a selection of the some the more advanced techniques and facilities which have allowed Fortran to become such an important and successful language. Covering topics and issues usually left untouched in standard introductory texts, it sets out to provide a stepping stone to better, more portable and more efficient programming.

This is, in particular, a book about scientific and numerical computing. After a brief discussion of programming techniques and mathematical software, it thus gives an introduction to numerically intensive or high performance computing. The issue of memory management (perhaps one of the most important determining factors for program efficiency) is presented and is illustrated by extensive references to linear algebra, the BLAS and LAPACK. Special topics such as *cache*, *paging* and *vectorisation* are also included in this discussion. Part I then concludes with an overview of programming style, optimisation and a selection of novel tricks for input–output.

The second and final part then continues with a brief introduction to the new standard, *Fortran 90*, and an overview of *High Performance Fortran*, informally known as HPF. Building on the discussions from part I, it is hoped that these chapters will provide a useful first introduction for FORTRAN 77 programmers.

Fortran 90

Clearly, this is not a book on Fortran 90. Readers looking for a such a reference should supplement this text with one of the many handbooks now becoming available on the new standard. This is a text on scientific computing. Thus, whereas no text on Fortran could now be complete without an introduction to the new standard, a discussion of Fortran 90 alone is not sufficient for a full understanding of Fortran programming. Details on how programs are implemented, programming style and the range of existing software and techniques are of equal interest to modern programmers.

even when Fortran 90 becomes fully established (not the case at the time of writing) workers will still have to understand, recompile and rewrite[1] FORTRAN 77 code for many years. Thus this book should be seen to complement and not replace standard works on Fortran 90. Finally, it should be stressed that many of the issues raised for FORTRAN 77 are also directly relevant to Fortran 90. At the time of writing, both standards were in force.

Who should read it?

This is a book written for anyone who uses Fortran to do mathematical or numerical computation on a computer. It is expected to be of particular interest to science, engineering or mathematics postgraduate students and to programmers and researchers in universities or industry. Starting from a basic knowledge of the language it could be used, for example, as a second level course in scientific computing to make students more aware of efficiency, architecture and software issues. The text is also suitable for self study (for which it was actually written) covering a range of topics for which little comprehensive secondary literature exists.

The material covered by this course was originally presented by the author in a series of courses to postgraduate students at the Interdisciplinary Center for Scientific Computing (IWR) at the University of Heidelberg, Germany.

Layout and solutions

Before turning to the first chapter, the reader is encouraged to quickly flip through the solutions to the exercises given in appendix C. In all, 40 pages of worked out solutions are given and, although the reader is free to work through them by themselves, this is not essential. Alongside the questions they address, the solutions provide supplementary illustrations and extensions to many of the topics discussed in the main text. For teachers, this may form the basis of a short one day course in its own right. Whether and how to make use of them is however left open to the reader's choice.

The material in this book is presented in a sequence of 22 chapters. Although these chapters are sometimes short, this number is a reflection on the diverse range of topics addressed. The chapters are intended to be read in the order in which they were written although the second part, on Fortran 90 and High Performance Fortran, is largely self contained.

Notation

No special notation is used in this book. Programs are written in lower case (bold in the text) and language keywords in the text are written in capitals. The name

[1] Owing to the major and fundamental differences between the two standards, this change may well turn out to be more difficult than from FORTRAN 66 to 77.

FORTRAN 77 is, by convention, also written in capitals, whereas Fortran 90 is taken as a normal proper noun.

Acknowledgements

I would like to conclude by thanking all those people who through their help also made an active contribution to this book. These include colleagues in Ciba-Geigy, Heidelberg and Manchester. Special thanks should go to Nick Higham and Len Freeman (University of Manchester) for their constant support and careful comments, David Sayers (Numerical Algorithms Group, Oxford) for his continual technical support and assistance and to Wolfgang Seewald (Ciba, Basel) for his constructive and useful suggestions. Thanks also to John Crow (Cray Research), Mike Boucher and Douglas Priest (Dakota Scientific Software and SunSoft) and Salvatore Filippone (IBM, Rome) for their provision of up-to-date performance figures. Finally, I would like to thank Roslyn Meredith (John Wiley) and her colleagues for their help and encouragement and assistance in the final stages of the preparation of this book. This document was written in LaTeX.

Part I
Scientific Computing

1

Double and Single Precision

1.1 Machine dependence

The exact meaning of REAL (single precision) and DOUBLE PRECISION is not standard and varies widely between different systems. On some, for example those by IBM, the word-length (that is the number of bits used to represent for example real numbers) is so restricted that the type REAL gives only 5 or 6 significant figures accuracy instead of the 10 or 12 you might expect. For numerical algorithms this may often not be adequate. On such systems DOUBLE PRECISION should be used where REAL might otherwise have been expected. To investigate REAL on your computer, try out the following program. Compile and run it first as it is, and then after changing the REAL statement to DOUBLE PRECISION.

```
      program round
c
c     This program searches for the machine precision. That is the
c     smallest number which when added to one, gives, to within the
c     machines arithmetic, a number distinct from one. Since floating
c     point numbers are represented on the machine in hexadecimal or
c     binary, this is presumed to be some inverse power of two.
c
      real x
c
      x = 1.0
c
    1 continue
      print*,x
      if (1.0 .lt. (1.0 + x)) then
         x = x/2.0
         goto 1
c
      else
         print*,'machine precision = ',2.0*x
      endif
c
      stop
      end
```

On most systems a short word-length convention is followed and thus for most numerical applications, DOUBLE PRECISION is normal. This may sometimes be called REAL*8, the notation REAL*4 used for normal REAL. This notation does not, however, conform to the FORTRAN standard. Where extra precision is required, the non-standard type, QUADRUPLE PRECISION, otherwise known as REAL*16 may also be used.

The use of a GOTO-statement in the above program is justified since it repeats a logical block of code. A full discussion of this and other stylistic practices is given later in chapter 13.

1.2 Type promotion

Ideally, before writing a FORTRAN program you should first think about which type to choose. For numerical applications, people often write their codes directly in double precision. Other alternatives are, however, possible. On your system, for example, you may be able to write a code in initially single precision but then compile it with an option which produces a double precision binary. This is known as *type promotion.* Alternatively there are a number of software tools that can take a FORTRAN code and convert all REAL statements to DOUBLE PRECISION (and possibly DOUBLE PRECISION to QUADRUPLE PRECISION where supported) automatically. An example of such a program is discussed in the next chapter, section 2.3.

1.3 Portability

Differences between REAL and DOUBLE PRECISION, and long and short word lengths, can cause serious portability problems moving between different computer systems. They are difficult because they are hidden, the word length for which a program was written not being immediately visible from the source code alone. We return to this subject in more depth in the next chapter. To keep within the language standard, non-standard extensions such REAL*4, REAL*8 and QUADRUPLE PRECISION should, however, be avoided. This is unfortunate because these are precisely the features that would be of use in assisting otherwise difficult program migrations.

1.4 Double precision and software libraries

Even though the distinctions between single and double precision are straightforward, problems can arise often when double and single precision routines are mixed within the same program. This is sometimes attempted when, for example, using a single precision graphics library with a double precision numerical code. Such practices can be very dangerous and should strongly be discouraged. Explicit type conversions are usually necessary and programming errors are very easy to make. In such cases it is often better to write two separate programs communicating data through data files or, where supported, operating system pipelines[1]. Finally, when using type promotion, a careful note should be made of whether binaries were compiled under single or double precision since such things are easily forgotten.

1.5 Machine representation

How floating point numbers are represented on a given machine is not only important in determining numerical precision: it also affects the numeric range and the handling of error and exception conditions. Details vary widely between machines but some examples for the now widespread IEEE standard, see [12], for floating point arithmetic are given in the exercises at the end of this chapter.

1.6 Fortran 90

A more powerful approach to machine precision is provided by the new standard Fortran 90. Besides allowing the programmer to declare variables explicitly in terms of the precision they require, a comprehensive range of inquiry functions are supplied to return implementation dependent parameters. We return to Fortran 90 in part II.

1.7 Exercises

Details of machine precision are often technical and vary widely between systems. Rather than discussing them directly in the text, a number of informative examples are presented in the exercises below. As the following chapters, full explanatory solutions are given at the end of the book.

1. (*machine precision*) Is it correct to assume that the machine precision takes the form $1/2^n$?

2. (*real comparisons*) How safe are equality comparisons between real numbers? What about .LE. and .GE.? What special precautions should be taken with DO-loops? How many times, for example, would you expect the loop

[1] A number of suggestions for UNIX users are given in section 15.2.

```
      do 1 x=a,a+1.0,0.2
         write(*,*) x
1     continue
```

to execute? Further comments on the use of DO-loops are given in section 13.7.

3. (*optimised inequalities?*) Are inequalities always safe under optimisation? Consider, for example, the test used to find the machine precision in the program listed in the text. Would you expect a compiler to make such an error?

4. (*IEEE arithmetic*) Does your system use IEEE arithmetic? If so, what values are returned in response to each of the following expressions:

   (a) **1.0/2.0**,

   (b) **7.87/0.0**,

   (c) **0.0/0.0**,

   (d) **-4.78/0.0**,

   (e) **sqrt(-1.0)**?

   Give an example of how such values can be used to good effect.

5. (*arithmetic identities*) In normal life, addition is associative. Is this however always true for floating point arithmetic? Is $(x + y) + z$ always the same as $x + (y + z)$? Further, does the statement $x = y \Leftrightarrow x - y = 0$ always hold true?

6. (*over- and underflow*) Evaluating long products of the form

$$p_N = \prod_{i=1}^{N} s_i$$

   over- or underflows can sometimes occur in the intermediate products $p_r = \prod_{i=1}^{r} s_r$ even when the terms $\{s_i\}$ and the end result p_N lie within the machine range. Using the notation $p_r = \bar{p}_r \alpha^{k_r}$ for some fixed α and sequence $(k_i)_{i \geq 1}$, suggest a simple algorithm which overcomes this problem.

7. (*type promotion*) Do you know how to promote types using your compiler? Do you know of any tools which can translate source codes automatically?

2

Portability and Standards

2.1 Program portability

Given modern communications and advancing technology, a program written on one system may not necessarily be run on the same system for all its working life. At some point it may be moved to, and possibly recompiled, under a new environment. Alternatively, it may be prepared for commercial use, sold, and distributed to a large number of different sites. Where such uses are likely, to try to ensure trouble free program migration, we often seek to remove implementation dependent features as far as possible. That is, we try to ensure *program portability*. Although, as illustrated by our discussion of machine precision, the construction of truly portable software is non-trivial, a number of basic steps can be taken. These are the subject of this chapter.

2.2 The FORTRAN standard

FORTRAN is a language agreed by international standard[1]. Most compilers, however, support a wide range of extensions over and above this basic core. Whereas such facilities are often powerful, frequently resulting in more efficient and easier to read codes, they are not standardised and vary widely between systems. Where portability is an issue, they should thus be avoided. Usually, the simplest way to test for conformance to the FORTRAN standard is to flag for language extensions with an appropriate compiler or preprocessor option. Selecting this as the default (possibly using *make* under UNIX, a suitable configuration file or some suitable command macro), programs can be checked automatically throughout their development cycle. Where possible, the programmer can also try to run the program on a number of differing computer platforms and compare the results. So doing may throw light onto more difficult problems such as machine precision (discussed above) and file handling (discussed below) which can also affect program portability.

[1] FORTRAN ANSI X3.9-1978 for FORTRAN 77 and ISO/IEC 1539 : 1991 for Fortran 90. At the time of writing, both standards were in force.

2.3 Portability testers and pretty writers

Besides using compiler options to check for the language standard, there are also a number of programming tools which can be used to aid program development and aid portability. Some, for example TOOLPACK [23], can parse (that is can analyse) the source code like a compiler and flag any programming extensions or possible errors. Blocks of unreachable code, often corresponding to programming errors can, for example, be so identified. Such packages can usually also automatically re-write or re-format the program, tidying declaration and executable statements, and adjusting indentation to reflect the logical structure of the code. This can often show up program errors in itself. Such tools are sometimes referred to as 'pretty writers'. Where many people contribute to a single code, they can be specially useful to help standardise the appearance and layout[2]. This often makes codes easier to read and to edit and can thus reduce the risk of errors. TOOLPACK also supports functions to detect and replace long variable names, translate source codes automatically from single to double precision (type promotion - section 1.2) and enforce strong typing (see section 2.4 below). Where separate programming tools are not available, relevant options may be supported by a compiler or a compiler preprocessor. Examples of the latter are given in chapter 14 whereas sample output from a pretty writer is shown in figure 1.

2.4 Strong typing

FORTRAN 77 and Fortran 90 are *weakly typed* languages. Variables not explicitly declared in a type statement are typed automatically according to arbitrary default rules. To detect mistyped variables and increase program readability, it may thus be desirable to force the compiler to require that all variables are explicitly typed. We refer to this as *strong typing*. Strong typing may usually be enforced by using an appropriate compiler option, or, where supported, by the non-standard IMPLICIT NONE statement. In the author's experience, such practices are to be strongly recommended. In the absence of an appropriate compiler option or IMPLICIT NONE, the FORTRAN 77 statement

```
IMPLICIT LOGICAL (A-Z)
```

may be considered. This types all non-explicitly declared variables as *logical*, thus increasing the chance of detecting typing errors. As we see in part II, however, IMPLICIT NONE is now included in the Fortran 90 standard. An illustration of the need for strong typing is given in section 13.2.

2.5 Machine accuracy

Particular portability problems arise from machine precision. As we have already seen, an identical program may give rise to completely different results depending on which system it is run on. This is potentially a serious problem. In writing numerical

[2] LAPACK and NAG for example are prepared using such tools.

```
      SUBROUTINE C05AZF(X,Y,FX,TOLX,IR,C,IND,IFAIL)
C     MARK 8 RE-ISSUE. NAG COPYRIGHT 1979.
C     MARK 11.5(F77) REVISED. (SEPT 1985.)
C     MARK 12A REVISED. IER-496 (AUG 1986).
C     MARK 13 REVISED. USE OF MARK 12 X02 FUNCTIONS (APR 1988).
C     .. Parameters ..
      CHARACTER*6       SRNAME
      PARAMETER         (SRNAME='C05AZF')
C     .. Scalar Arguments ..
      DOUBLE PRECISION  FX, TOLX, X, Y
      INTEGER           IFAIL, IND, IR
C     .. Array Arguments ..
      DOUBLE PRECISION  C(17)
C     .. Local Scalars ..
      DOUBLE PRECISION  AB, DIFF, DIFF1, DIFF2, REL, RMAX, TOL, TOL1
      INTEGER           I
      LOGICAL           T
C     .. Local Arrays ..
      CHARACTER*1       P01REC(1)
C     .. External Functions ..
      DOUBLE PRECISION  X02AJF, X02AKF
      INTEGER           P01ABF
      EXTERNAL          X02AJF, X02AKF, P01ABF
C     .. Intrinsic Functions ..
      INTRINSIC         ABS, MAX, SIGN, DBLE, SQRT, INT
C     .. Executable Statements ..
      I = 0
      IF ((IND.GT.0 .AND. IND.LE.4) .OR. IND.EQ.-1) GO TO 20
C     USER NOT CHECKED IND OR CHANGED IT
      I = 2
      IND = 0
      GO TO 640
      ...
      ...
```

Figure 1. **Output of a pretty writer.**

An example of a pretty writer in action. Such tools can have an important rôle in code development. In the above case, for example, not only is the declaration part better set out and easier to read than in handwritten codes but it may also be easier to use. The type statements for the routine's parameters in the calling program could, for instance, be edited directly from the sections 'Scalar Arguments' and 'Array Arguments' in the subroutine itself. This program segment is reproduced here with the kind permission of the Numerical Algorithms Group Ltd. and was prepared with a polished version of TOOLPACK marketed as part of their NAGware range of products.

software, the programmer should first establish what its accuracy requirements are, and then ask if they can be met on the host system. Particular problems can occur with integers whose type cannot always be promoted. To automatically enquire about system parameters a number of programs and packages are often used. These include R1MACH available over *Netlib*[3], found in many publicly available codes, and chapter X02 of the NAG library[4]. The handling of this problem in Fortran 90 is discussed later in this text.

2.6 Saving and file handling

File handling, and the defaults determining implicit saving, frequently give rise to portability problems. The former is particularly problematic. Extra REWIND statements, for example, may be required under some systems since the initial status of the file pointer is not defined in FORTRAN 77. Indeed, for complicated applications involving say INQUIRE or unformatted files[5] (such files are system dependent although conversion programs may be possible), it may be better to place all file handling functions in a separate routine. Undetected saving and uninitialised variables can also cause portability problems. We discuss these in section 4.5.2 as we do improvements for file handling in Fortran 90 in chapter 21.

2.7 The US military standard

FORTRAN is a language currently undergoing rapid change. In particular, it is moving towards Fortran 90. Thus although they do not conform to the FORTRAN 77 standard, some extensions, such as those conforming to the US Military Standard MIL-STD 1753[6], have become so widespread that they merit recognition. These include

- IMPLICIT NONE (mentioned above),
- alternative DO-END DO syntax for DO-loops and
- variable names with more than 6 letters and/or underscores,

the second of which we return to in chapter 13. Although they make an important contribution to the language and are included in the new standard, Fortran 90, for portable programming, they should be avoided. This is the practice adopted in this book. Before considering the possible migration from FORTRAN 77 to Fortran 90 the reader should read chapter 16.

[3] R1MACH is part of the PORT library which may be found under 'core' in *Netlib*. *Netlib* is discussed in section 3.5.
[4] See chapter 3.
[5] See section 14.1.9.
[6] A U.S. Department of Defense supplement to the ANSI 1978 Fortran standard.

2.8 BLAS and LAPACK

Non-standard language features may be used to support novel architectures such as vector or parallel machines. Although in such applications their use is often inevitable, they may sometimes be avoided by the appropriate use of subprogram libraries such as the BLAS or LAPACK. These have the effect of removing architecture dependent issues from user written programs. We return to these to packages in chapters 9 to 12 when we discuss numerical linear algebra.

2.9 Comments and documentation

Central to any good code are informative comments and documentation. Although in-line comments are widely used, developing programs under UNIX, the commands *sccs* and *what* may also be of use[7]. In this book, program comments are principally restricted to the accompanying text. Exceptions are where example programs are complicated or are envisaged as a user usable routine.

[7] Alternative techniques include the insertion of special character sequences such as c:, c! or c>> in comment lines. These may then be searched for and printed by an appropriate operating system command.

3

The NAG Library

3.1 Mathematical software

In many applications, mathematical problems arise which cannot be solved directly by analytic methods and therefore must be treated numerically. Fortunately, such problems fall into certain generic classes for which special theories and techniques have already been developed. Examples include applications in linear algebra, the solution of differential equations, approximation theory and root finding. The area of mathematics which addresses such problems is known as Numerical Analysis. Although this is not a text on numerical methods, many of the techniques derived by numerical analysts have been collected and coded in the form of computer programs and are widely available in the form of mathematical software libraries. In this chapter we present some introductory illustrations of one widely used such library published by the Numerical Algorithms Group (NAG) Ltd., Oxford. For further details of this product see [37] and Phillips [22].

3.2 Why use software libraries?

Whereas it is often claimed that for many practical problems, the use of a software library is superfluous, such a view can often be misguided. The correct treatment of apparently normal numerical problems is frequently non-trivial requiring specialised methods known only to people specialising in a certain field. By using a prewritten library program, the user can however directly access state-of-the-art technology without specialist technical knowledge.

Furthermore, software libraries often come with extensive documentation which can in itself give a valuable insight into the range and scope of possible numerical schemes available. Indeed, some, such as that published by the Numerical Algorithms Group above, include an introduction to the relevant mathematical background, essential in many areas. Thus far from being a hindrance, the presence of large reference manuals is often a help, helping the user to think carefully about the problem and issues involved.

Using existing software packages often also results in clearer and safer codes. Writing your own code, or copying one from a book, can frequently introduce minor program errors which, especially in the case of numerical problems, can be very difficult to correct or detect. In contrast, most software libraries have been through stringent testing and verification phases and so are potentially much more robust. The

advantages of using well documented pre-written codes for modular programming are, of course, clear.

3.3 Other libraries

Although we give it special attention, the NAG library – the mathematical software library published by the Numerical Algorithms Group Ltd. – is not unique. The libraries

- **IMSL** [36], an American version of NAG – very popular but smaller,
- **LINPACK,** a general linear algebra package,
- **LAPACK,** a general linear algebra package – the successor to LINPACK,
- **HARWELL** [35], specialist routines for sparse matrices, and
- **ESSL** [32], highly tuned routines for IBM machines,

for example, also provide mathematical software. Indeed, some may be better suited for some purposes. Whereas, for example, the NAG and IMSL libraries seek to provide a wide range of products over many different machines, others, for example ESSL written by IBM for IBM machines, address only a limited range of tasks which they seek to optimise for a specific architecture. Thus, for example, for linear algebra on IBM systems, ESSL is often faster. This difference is also reflected in the languages in which the libraries are written. For NAG, for example, only standard FORTRAN is used to seek to achieve maintenance free portability across a wide range of computer environments. ESSL, in contrast, can use a mixture of FORTRAN and machine specific machine code to execute faster. It is thus, however, limited to IBM machines. For many problems well written FORTRAN can rival or even beat existing machine code. This is possible through the correct use of existing libraries such as ESSL and systems such as the BLAS (see below). One object of this book is to show how this can be done.

3.4 Linear algebra

A major application of numerical libraries is in numerical linear algebra. For dense (non sparse) problems, besides NAG and IMSL, two packages are in widespread use: LINPACK and LAPACK. Of these, LAPACK which we discuss later in chapter 12 is the newer and, in fact, supersedes LINPACK. LAPACK also makes extensive use of the Basic Linear Algebra Subprograms (BLAS) which we introduce in chapter 9. For an introduction to LINPACK see [5]. Both libraries can be obtained freely and electronically over *Netlib* (see below). Where program portability is not an issue, manufacturer supplied libraries, such as ESSL for IBM systems, may also be considered.

3.5 Netlib

As stated above, LINPACK and LAPACK can both be obtained without charge over *Netlib*. *Netlib* [8] is an automatic repository for general and mathematical software accessible directly by electronic mail. For further details, send the following message

```
send index
```

to one[1] of the two following *Internet* addresses:

```
netlib@ornl.gov
netlib@nac.no
```

A help file with index will then be returned to you automatically by electronic mail. Included in this data base are generic versions of the BLAS (suitable where no manufacturer supplied implementation is available — see chapter 9) and a wide range of software for non-standard problems. This includes, in particular, the collected algorithms of the journal 'Transactions in Mathematical Software' from the Association for Computing Machinery (ACM TOMS). Such sources can sometimes be useful for problems for which no standard library software exists.

3.6 Internet

Netlib is of course just a part of the rapidly expanding network of repositories and servers now connected as part of the *Internet* network. In addition to traditional *e-mail* and *ftp*, a wide range of alternative tools and services are now available to help navigate around the system. These, which include, for example, programs such as *gopher*, *archie* and *mosaic*, are well documented in the computer science literature. For further details see [11, 17]. Alternatively, remote login intelligent file servers can also provide automatic methods for searching for and getting programs and files. At the time of writing *e-lib* in Berlin[2] was one such example.

3.7 The NAG library

3.7.1 An overview

Of the libraries listed above, NAG is by far the most wide ranging including, besides standard numerical routines, subprograms for statistics, operations research and even, optionally, graphics. Internally it is organised into a number of different chapters, grouped according to the application area. The numbers of these chapters then contribute to the first three letters to the names of the routines they contain. Typical examples include

[1] From within Europe, the second address should preferably be used. These addresses were correct at the time of writing.

[2] Log in as *elib* at *elib.zib-berlin.de*. Other *Internet* tools are also supported.

- **A02** complex arithmetic,
- **C02** zeros of polynomials,
- **C05** roots of one or more transcendental equations,
- **C06** summation of series,
- **D01** quadrature,
- **D02** ordinary differential equations and
- **D03** partial differential equations.

A complete listing is given in table 1. For example, the routines

- **A02AAF** finds the square root of a complex number
- **A02ACF** computes the quotient of two complex numbers

belong to the **A02** chapter on complex arithmetic whereas

- **D01AHF** performs adaptive integration over a finite interval
- **D01BAF** evaluates a Gaussian quadrature rule

belong to the chapter **D01** and perform numerical quadrature. As indicated above, each chapter begins with a brief introduction to its mathematical background, the techniques used and the problems which can arise. This is often followed by a system of flowcharts designed to help you choose the right routine. The library comes in two otherwise identical versions, one for single and one for double precision, the precision being used to define the last letter of each routine's name. In the double precision version, all names end in the letter 'F', and in the single precision version 'E'. All references given in this book are to the more common double precision library.

3.7.2 Using the NAG library – two examples

We now consider two simple numerical problems: a simple definite integration and the solution of a non-linear algebraic equation. These will illustrate a number of features common also for more difficult problems. The second example, in particular, illustrates the use of *reverse communication*.

A numerical integration

The NAG library includes a sizable number of different routines to approximate definite integrals of the form $\int_a^b f(x)\,dx$. We, however, consider only the model problem

$$\int_0^1 \sin x^2 dx.$$

Although for this, and many other problems, a more specific method (such as a Gaussian type scheme) might be more appropriate, for purposes of illustration, we select a general purpose routine D01AHF[3] Full details of alternative methods, as well as the mathematical background to such problems, are given in the library

[3] We do this for simplicity. For general use, NAG recommends the routine D01AJF. This is illustrated in exercise 2 at the end of this chapter.

Table 1. **The NAG mark 16 FORTRAN library: chapter headings.**

A02	– complex arithmetic
C02	– zeros of polynomials
C05	– roots of one or more transcendental equations
C06	– summation of series
D01	– quadrature
D02	– ordinary differential equations
D02N-D02M	– integrators for stiff ordinary differential equations
D03	– partial differential equations
D04	– numerical differentiation
D05	– integral equations
E01	– interpolation
E02	– curve and surface fitting
E04	– minimising and maximising a function
F01	– matrix operations, including inversion
F02	– eigenvalues and eigenvectors
F03	– determinants
F04	– simultaneous linear equations
F05	– orthogonalisation
F06	– linear algebra support routines
F07	– linear equations (LAPACK)
F08	– least squares and eigenvalue problems (LAPACK)
G01	– simple calculations on statistical data
G02	– correlation and regression analysis
G03	– multivariate methods
G04	– analysis of variance
G05	– random number generators
G07	– univariate estimation
G08	– nonparametric statistics
G10	– smoothing in statistics
G11	– contingency table analysis
G12	– survival analysis
G13	– time series analysis
H	– operations research
M01	– sorting
P01	– error trapping
S	– approximations of special functions
X01	– mathematical constants
X02	– machine constants
X03	– inner-products
X04	– input/output utilities
X05	– date and time utilities

documentation [37]. A summary of the calling sequence to D01AHF is given at the end of this chapter in section 3.8.1. Here, as indicated above, the double precision version of the NAG library is assumed.

```
      program quad
      integer npts,nlimit,ifail
      double precision a,b,epsr,relerr,f,value,d01ahf
      external f,d01ahf
c
      a = 0
      b = 1
c
      nlimit = 0
      epsr   = .1e-6
      ifail  = 1
c
      value = d01ahf(a,b,epsr,npts,relerr,f,nlimit,ifail)
c
      write(*,*)
      write(*,*) ' npts      ', npts
      write(*,*) ' epsr      ', epsr
      write(*,*) ' relerr    ', relerr
      write(*,*) ' ifail     ', ifail
      write(*,*)
      write(*,*) ' result    ', value
      write(*,*)
      write(*,*)
c
      stop
      end
```

which calls the subprogram

```
      double precision function f(x)
      double precision x
      f = sin(x**2)
      return
      end
```

In the above example, the variable **ifail** is used to return error information about the integration once the NAG call has been completed. The value **ifail** = 0 indicates a satisfactory result, other values indicate error conditions which should be tested for by the user:

ifail	*interpretation for D01AHF*
0	Routine terminated successfully.
1	Integral did not converge to the requested accuracy. Try increasing **nlimit**.
2	Too many unsuccessful levels of subdivision were invoked.
3	Invalid accuracy request. That is, on entry, **epsr** less than or equal to zero.

This is known as *soft* error handling. Alternatively, setting **ifail** = 0 on calling (the simplest selection), such errors give rise to a so called *hard* failure. If an error is detected, an error message is produced and the program is terminated. The variable **ifail** is a standard feature of the NAG library. Its further use is discussed in exercise 4 at the end of this chapter.

A non-linear root finding problem

For further illustration, we now consider the problem

$$x^2 - 2 = 0. \tag{3.1}$$

Such a simple and trivial problem could obviously be treated by Newton iteration but we here consider instead the use of two NAG routines: C05ADF and C05AZF.

• C05ADF — a forward communication code

C05ADF is a simple, easy to use, code for finding a zero α

$$f(\alpha) = 0$$

of a function over a given interval $[a, b]$. The user is required to supply a FORTRAN function to return the value of $f(x)$ for given values of x. An example is given below:

```
      program forwd
      integer ifail
      double precision a,b,eps,eta,f,x
      external f
c
      a = 0.0
      b = 2.0
c
      eps = .1e-5
      eta = 0.0
c
      ifail = 0
c
      call c05adf(a,b,eps,eta,f,x,ifail)
c
      if (ifail .eq. 0) then
```

```
         write(*,*) '** f(',x,') = ',f(x)
      else
         write(*,*) '** code failed. IFAIL = ', ifail
      endif
c
      stop
      end
```

which calls

```
      double precision function f(x)
      double precision x
      f = x**2 - 2.0
      return
      end
```

A summary of the calling sequence of C05ADF is given in section 3.8.2. The use of a user supplied function in this way is standard for most applications. We refer to this as *forward* or *direct communication.*

• C05AZF — a reverse communication code

Instead of the direct communication used above, an alternative technique is illustrated in the following example. The routine C05AZF is mathematically equivalent to the standard code C05ADF[4], but instead of requiring a user supplied objective function to compute f, it returns control to the user along with a prompt to request the required value for $f(x)$. Once this has been provided, the user must then return control to the NAG routine and its execution is continued from where it was interrupted until a further $f(x)$ value is required or its execution is completed. We refer to this as *reverse communication.* Discussed in detail in section 4.6, reverse communication may be considered as analogous to that of *co-procedures* in *Modula-2* [28]. A calling summary of C05AZF appears in section 3.8.3.

```
      program back
      integer ir,ind,ifail
      double precision x,y,fx,tolx,c(17),f
      external f
c
      x = 0.0
      y = 2.0
c
      tolx = .1e-6
      ir = 0
      ifail = 0
c
      ind = 1
c
```

[4] C05ADF acts as a forward-communication interface for C05AZF.

```
1     continue
      call c05azf(x,y,fx,tolx,ir,c,ind,ifail)
c
      if ((ind .eq. 2) .or. (ind .eq. 3) .or. (ind .eq. 4)) then
         fx = f(x)
         goto 1
      elseif (ind .eq. 0) then
         if (ifail .eq. 0) then
            write(*,*) '** f(',x,') = ',f(x)
         else
            write(*,*) '** code failed. IFAIL = ',ifail
         endif
      else
         write(*,*) '** a coding error. IND should be 0,2,3,4 here'
      endif
c
      stop
      end
```

Reverse communication routines are typically slightly harder to use and code than those using standard forward communication but they are often more flexible. What would happen, for example, if we were to replace equation (3.1) by

$$x^2 - b = 0,$$

where b was input from the screen? Which of the two programs would be the easier to adapt?

3.8 Subroutine references

3.8.1 D01AHF – integration of a definite integral

The NAG FORTRAN function D01AHF returns an approximation of the definite integral across a finite interval $[a, b]$

$$\int_a^b f(x)dx$$

to within a relative error **epsr**. Taken from the double precision version[5] of the NAG library, it may be called by

```
d01ahf(a,b,epsr,npts,relerr,f,nlimit,ifail)
```

where **a**, **b**, **epsr** and **relerr** are double precision whereas **npts**, **nlimit** and **ifail** are of type integer. The variables **a**, **b**, **epsr**, **nlimit** (an upper limit on the number of function evaluations) and **ifail** (see main text, section 3.7.2) should be set before

[5] Using the single precision version (i.e. D01AHE), the double precision declarations for **a**, **b**, **epsr**, **relerr** and **f** should be replaced by single precision. The same follows for the following examples.

each call. On return, **npts** and **relerr** return the number of function evaluations and an estimate of the relative error respectively whereas on successful completion, the (double precision) function value **d01ahf** returns the computed estimate for the integral. The reference **f** is to the double precision function

```
double precision function f(x)
```

which returns the integrand value for a double precision point **x**. **x** should not be reset internally by **f**.

3.8.2 C05ADF – a forward communication root finder

The subroutine C05ADF locates an interval $[a, b]$ for a root α to within a tolerance $|x - \alpha| \leq$ **eps** or $|x - \alpha| <$ **eps** and $|f(\alpha)| <$ **eta**. It is invoked by a call of the form

```
call c05adf(a,b,eps,eta,f,x,ifail)
```

where, in the double precision library, **a**, **b**, **eps**, **eta** and **x** are double precision and **ifail** is an integer. **eps** and **eta** have the meanings indicated above except where **eta** is zero when no $|f(\alpha)| <$ **eta** test is enforced. **ifail** is an error parameter, zero upon successful completion, with the same usage as for D01AHF and **x** is used to return the solution, α, if obtained. **f** is a double precision function

```
double precision function f(x)
```

to return the value of the function f for the point **x** (double precision). **x** should not be reset internally by **f**.

3.8.3 C05AZF – a reverse communication root finder

The subroutine **c05azf** has the calling sequence

```
call c05azf(a,b,fx,tolx,ir,c,ind,ifail).
```

Mathematically, C05ADF is the equivalent to C05AZF. However the double precision function **f** is replaced by the double precision variable **fx** which should be set to $f(\mathbf{x})$ by the user every time the program is returned with integer **ind** equal to 2, 3 or 4. For normal use, **ind** should be initialised to one whereas it is equal to zero upon final exit. **x** is double precision. The integer **ir** controls the type of error test used to test for convergence and **tolx** (double precision) is the x-tolerance. For many problems, **ir** $= 0$ is an appropriate choice. **ifail** has a similar meaning as in previous routines and c is a double precision array of dimension at least 17 to be used internally by the routine. For further details, see the library documentation [37].

3.9 Exercises

The mark 16 edition of the NAG library provides over 1,000 fully documented routines designed to solve a wide range of mathematical and technical problems well beyond

those illustrated above. For further examples, including ordinary differential equations, sorting, statistics and mathematical and machine constants, see the examples below. As usual, full and comprehensive solutions are provided at the end of this text.

1. (*a simple example - ODE's*) The NAG library provides numerical routines for a wide range of mathematical problems. Included are ordinary differential equations. Using the NAG library, obtain a numerical solution to the Lorenz system

$$\begin{aligned} y_1'(t) &= 10(y_2(t) - y_1(t)) \\ y_2'(t) &= 28y_1(t) - y_2(t) - y_1(t)y_3(t) \\ y_3'(t) &= y_1(t)y_2(t) - 8/3y_3(t) \end{aligned}$$

over the range $t \in [0, 50]$ from a starting value of $\boldsymbol{y}(0) = (1, 1, 1)^T$. If suitable graphics are available, modify your code to produce output for a 3-D graph.

2. (*integration*) For general use, NAG recommends the more cautious routine D01AJF. Repeat the example of section 3.7.2 using this routine.

3. (*sorting*) NAG also supports a number of non-numerical operations. Using appropriate name routine(s), read in a list of names from a file or standard input and print them out in alphabetical order.

4. (*errors and file handling*) In the text, we explained the effect of setting IFAIL to 0 or 1 prior to a NAG routine call. What effect does passing a value of -1 have? How can you change the unit number to which error messages are sent?
Optional assignment: Using routines from chapter **X04**, print out a double precision 2D array.

5. (*machine and mathematical constants*) Using the NAG library, obtain numerical approximations to π, the Gamma function Γ at 1.5, $\Gamma(1.5)$, the machine precision (in DOUBLE PRECISION), the smallest representable positive DOUBLE PRECISION number and the largest expressible integer. Are these numbers exact?

6. (*random permutations*) The NAG routine G05EHF applies a random permutation to an integer array. Without using this routine, but rather the ranking routine M01DAF, write a program to shuffle a set of 52 playing cards labelled 1 to 52.

4

Save, Reverse Communication and Arrays

4.1 Static and dynamic storage: FORTRAN and PASCAL

FORTRAN77 and PASCAL are two very different languages. One fundamental difference is the way in which they allocate storage: the difference between *static* and *dynamic* storage allocation.

In FORTRAN77 storage allocation is essentially simple. Array dimensions and storage requirements are fixed at compile time. That is, given the declarations defined in the source code, the compiler allocates the required storage to each routine as the program is compiled. This is known as *static storage allocation*. Using static storage allocation there is no way to give up old or allocate new storage once the program is running since by this time its allocation has already been fixed.

In languages such as PASCAL, Ada or Fortran90 this is however not the case. Storage can then be allocated or recovered at will during *run time* after the compilation has been completed and once execution has been started. This can occur, for example, when subroutines are called (see below) or when special structures such as pointers or allocatable arrays are used. We refer to this as *dynamic storage allocation*.

Languages supporting dynamic storage allocation are attractive because they allow storage to be adjusted or defined in response to information which becomes available only when the program is running. They allow, for example, code segments of the form

```
subroutine compute(n,m)
integer n,m,A(n,m)
...
```

in which an array **A** is allocated and its dimension defined on the basis of two run-time variables **n** and **m**. In FORTRAN77 this would not be possible (**n** and **m** would have to be constants – parameters – and hence known at compile time) and the storage for **A** would have to be passed explicitly though the parameter list. The absence of dynamic storage in FORTRAN77 turns out in fact to be a serious drawback. Dynamic storage also permits more advanced structures such as the allocatable arrays and pointers described for Fortran90 in part II. The above example was in fact written in Fortran 90. Techniques for dealing with the lack of dynamic allocation in FORTRAN77 are returned to in chapter 7.

4.2 Subroutine calls

A good illustration of the difference between static and dynamic storage is the way in which procedures or subroutines are called. Once more, in FORTRAN the procedure call is essentially straightforward. Recall that a separate segment of storage is assigned to each routine at compile time. Thus to call a subroutine, the system needs merely transfer control to the appropriate storage location, copy the addresses of any required parameters, and resume execution. No additional storage allocation is required. The required allocation for each subroutine was made at *compile* time. In languages supporting dynamic allocation, such as PASCAL, however, the situation is more complicated. Storage allocation is made as and when it is required, in particular, here, when the procedure is called: during *run* time. Required variables are then set up in a new block of storage, control is transferred, and execution resumed. When the procedure is completed, control is reverted back to the calling procedure and the allocated storage can then be freed, and, in principle, re-used. Although this approach is obviously more complicated than in FORTRAN, it is in principle more flexible providing, for example, for recursion and automatic arrays (see sections 17.4 and 19.1).

4.3 Recursion

In FORTRAN 77 a program unit *cannot* call itself. This is a direct consequence of the use of static storage allocation. To explain this, consider the definition of

```
fact(n) := n * fact(n-1)
fact(1) := 1
```

to calculate the factorial $n!$ for integers $n \geq 1$. In FORTRAN, only one storage area is assigned to each routine with storage sufficient only for one set of variables. Thus in an implementation of the above example, **fact** could not call itself as this would require a second set of storage locations merely to implement the recursive call to **fact(n-1)**. Recursion in FORTRAN is therefore not possible because simultaneous calls to the same routine cannot be supported. This is a direct consequence of static storage allocation. In PASCAL, for example, it is not the case. Multiple areas of storage for the same routine are allowed, each one generated by each separate, normal, function call. In this respect, recursive calls are essentially no different from non-recursive calls, the number of duplicated blocks corresponding to the level of recursion. Using the above scheme, for example, n function calls and n corresponding storage allocations would be required to implement the simple product $n!$. This illustrates an important point: although recursion is often a very powerful programming tool, it can also be very inefficient. Care should always be taken to expand, where possible, recursive algorithms into non-recursive ones. One such example is given in exercise 5 at the end of this chapter. An implementation of the above factorial example in Fortran 90, a language which supports recursion, is given in the solutions to chapter 17.

4.4 Disadvantages of dynamic storage

Using dynamic storage allocation, particular problems can occur in managing dynamically allocated storage space which is no longer required. Whereas, in principle, after for example a procedure call, storage can be recovered and reused, this process is often costly and inefficient. Similar problems associated with pointers are mentioned in section 18.4.2. Methods for overcoming some of the deficiencies of the lack of dynamic storage in FORTRAN 77 are discussed in chapter 7.

4.5 The SAVE statement

One practical consequence of the use of static storage allocation in FORTRAN is that local variables (variables in a sub-program that do not appear in COMMON or in the parameter list) can be retained in storage after the sub-program call is completed. If you leave a subroutine and then re-enter it at a later time, the local variable values from the previous call *may* still be the same as when you last left it. This can be very useful. Although by default the FORTRAN standard does not retain such values, you can always force values to be preserved by means of a SAVE statement. If called with no arguments, all local variables will be kept. In contrast, followed by a list

```
      integer i,j,k,l,m
      save k,l,m
```

only the quoted variables are affected. One application of SAVE is shown by the following program used to count function invocations.

```
      subroutine count(action)
      integer tally
      character action*1
      save tally
c
      if (action .eq. 'i') then
         tally = 0
      elseif (action .eq. 'c') then
         tally = tally + 1
      elseif (action .eq. 'p') then
         write(*,*) '** count: tally = ', tally
      else
         write(*,*) '** count: error. Action ',action,' unknown.'
         stop
      endif
c
      return
      end
```

After an initial call

```
      call count('initialise')
```

to set the counter **tally** to zero, each call

```
call count('count')
```

has the effect of incrementing the counter by one. Finally, the current value of **tally**, that is the number of times **count** was called with the argument 'count', can be given by a call

```
call count('print')
```

The routine thus has a memory. Note that since there is only one local variable, the statements **save** and **save tally** are here identical. Passing character strings of length greater than one to **action** is, of course, legal but since **action** is itself length one, only the first character is read.

4.5.1 SAVE as the default

The treatment of local variables after RETURN differs greatly from compiler to compiler. In some environments, such as the IBM compiler FORTVS2 [39] under VM/CMS, it is practically the default, and local variables are nearly always preserved. Indeed this may be illustrated by the following program:

```
      program clear
c
      integer i
      write(*,*) 'initial value = ',i
      write(*,*) 'new value?'
      read(*,*) i
c
      stop
      end
```

Unless you specifically use a CLEAR load option, values may even be retained between different runs of the same program.

4.5.2 Dangers with unwanted saving

Although, as in our example COUNT above, SAVE can often be useful, the default saving of local variables by the compiler (or the partial saving with a certain probability) without an explicit SAVE can be very dangerous. Programs can behave in very strange and system dependent ways. Would you have expected the above IBM example to behave as it did? Such examples illustrate the need for correctly initialising all variables at the start of a program. We discuss this below in section 4.5.4.

4.5.3 Efficiency

As we will see later, the efficient allocation of memory for variables can be the determining factor in program efficiency. Thus, restricting the compiler's flexibility by issuing a SAVE statement may adversely affect performance. However, since such

memory problems are most likely when using, say, linear algebra, where SAVEs are usually used in non-numerical intensive applications (cf. **count** above), this conflict need not be serious. Other people may however disagree with this view.

4.5.4 Initialisation

A common cause of program errors is uninitialised program variables. Problems can occur when variables are referenced before their values have been defined. On many systems all variables may be automatically initialised to zero before execution but, as we have seen, this is not always so. Uninitialised variables are in general very difficult to cope with. For example it is often considered best to set variables to zero before use but this choice is completely arbitrary. But why not use say 78.4, 10, -1 or -999.999? Indeed, initialising variables can sometimes simply result in replicating the same mistake. A nice solution on one CDC compiler was to set all uninitialised variables to a machine representation of negative infinity. This improved the chance of detecting an error if values were used before being assigned. Other systems may test for the presence of uninitialised variables explicitly. Since uninitialised errors are a common cause of problems when moving programs between different computing environments, a uniform strategy would seem sensible.

4.6 Reverse communication – an example

One useful application of SAVE is in writing codes to be used by reverse communication. This is to save values between successive returns (requests) to the user for information. To illustrate this, and the principle of reverse communication in more detail, consider the following program:

```
      program add

      implicit none
      integer prompt
      double precision f,x,fx
      external f

      prompt = 0

 1    continue
      call sums(prompt,x,fx)

      if (prompt .eq. 0) then
         write(*,*) 'sum = ', fx
      else
         fx=f(x)
         goto 1
      endif
```

```
      stop
      end
```

which given a subroutine

```
      subroutine sums(prompt,x,fx)

      integer prompt
      double precision x, fx, sum
      save sum

      if (prompt .eq. 0) then
c        ... initialise sum and request f(1.0)
         sum = 0.0
         prompt = 1
         x = 1.0
         return

      elseif (prompt .eq. 1) then
c        ... increment sum and request f(2.0)
         sum = sum + fx
         prompt = 2
         x = 2.0
         return

      elseif (prompt .eq. 2) then
c        ... complete sum and return result
         sum = sum + fx
         prompt = 0
         fx = sum
         return

      endif

      write(*,*) '*** prompt should be 0,1 or 2'
      stop

      end
```

computes the sum

$$f(1.0) + f(2.0) \tag{4.1}$$

for a double precision function f. We illustrate this in figure 1. Here, as for C05AZF in chapter 3, a call to the objective function f is replaced by a return to the calling routine **sums**, alongside a request for the required function value. In particular, these are requested by the flags **prompt** $= 1$ and **prompt** $= 2$. Starting with an initial setting of **prompt** $= 0$, the user must merely test if the summation is complete, i.e. if **prompt** $= 0$, after each call and if not, return the appropriate value of **fx**, $f(x)$.

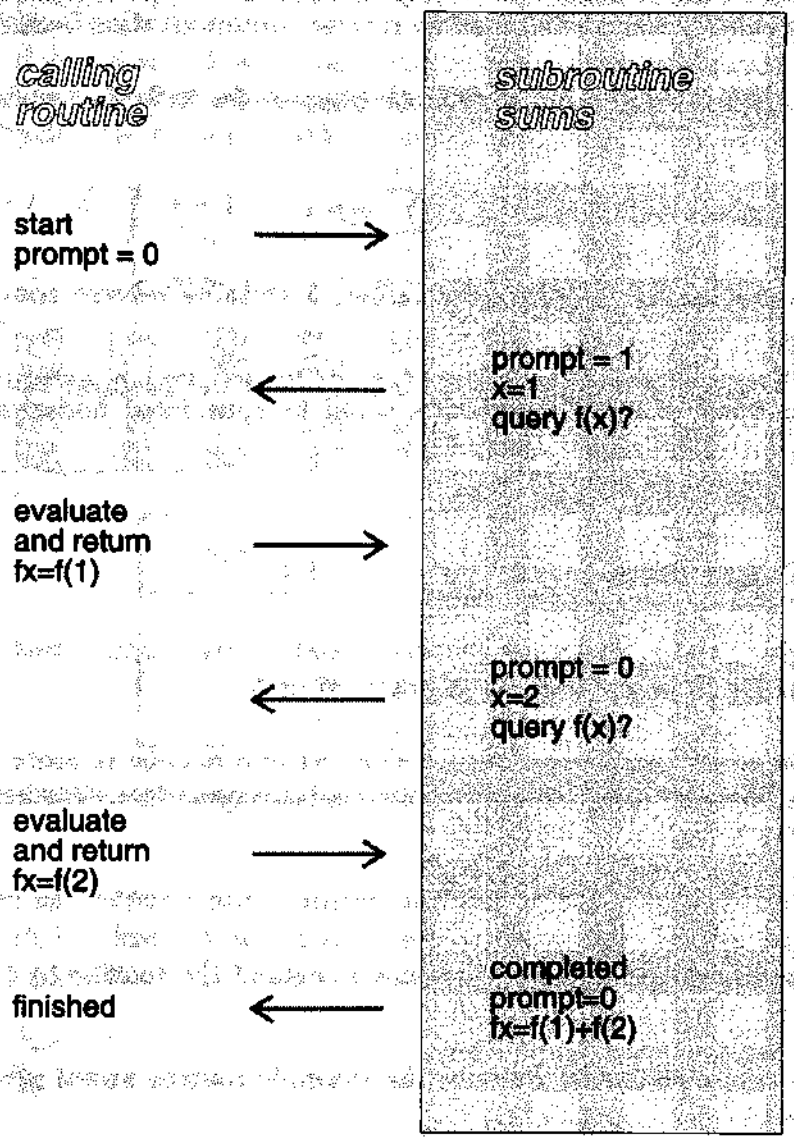


Figure 1. **Reverse communication.**
This can be thought of as replacing a functional or subroutine parameter by a dialogue between the calling and called routines. We illustrate this for program **add** *described in the text.*

The use of SAVE thus ensures that the current value for **sum** is retained between every call and hence removes the need to include it as a parameter on the parameter list. Writing subroutines to be called by reverse communication is clearly more difficult than writing standard forward communication codes, but as was pointed out in section 3.7.2, they are, in principle, more flexible. Suppose for example we were to replace the problem (4.1) by

$$\sum_{i=1}^{10} g(\boldsymbol{p}, i)$$

where $\boldsymbol{p}$ was a vector of parameter values, a suitable reverse communication code could be used without any modification – see exercise 3 below. A standard code would require additional parameters or a COMMON link to communicate the extra information. Both of these approaches could be considered undesirable[1]. In Fortran 90, an alternative technique would be to use a *data module*: see section 17.2.

4.7 Exercises

Here, we give further examples of saving and reverse communication. The routine **count** (exercise 2) might even be of practical use.

1. (*subroutines with memory*) Using SAVE, write a routine to store and recall values from an integer array. What, if any, advantages does this routine have over COMMON?

2. (*a practical example*) Following from **count**, write a routine to record and output the minimum and maximum values from a one dimensional data stream, $\{x_i\}_i$. What modifications would you require to extend the routine to a 2D data stream, $\{(x_i, y_i)\}_i$?

3. (*reverse communication*) Rewrite the example routine **sums** given in the text to compute a sum of the form

$$\sum_{i=1}^{10} f(i).$$

   Explain how this code could be extended to an expression of the form $\sum_{i=1}^{10} f(b, i)$.

4. (*Newton iteration*) A common iteration to solve problems of the form

$$f(x) = 0$$

   is Newton's method defined by

$$x_{n+1} = x_n - f(x_n)/f'(x_n).$$

[1] COMMON is often considered bad programming style. Difficulties resulting from using COMMON are mentioned in section 13.3. A possible advantage appears in section 14.1.8.

Write a subroutine to carry out the above iteration communicating f and f' by reverse communication. Apply this routine to solve equations of the form $x^2 - a = 0$ for user given a.

5. (*Ackermann's function*) Not all recursively defined functions require recursive algorithms. Consider, for example, Ackermann's function [2]:

$$A(m,n) = \begin{cases} n+1 & : \quad m = 0 \\ A(m-1,1) & : \quad m \neq 0, n = 0 \\ A(m-1, A(m,n-1)) & : \quad m \neq 0, n \neq 0. \end{cases}$$

Write a simple FORTRAN 77 program to evaluate $A(m,n)$ for given m and n. An array and loop are required but no recursion. An example of a recursive approach written in Fortran 90 is given in the solutions in appendix C. A selection of values for $A(m,n)$ is given in the table below:

$A(m,n)$	n			
	1	2	3	4
1	*3*	*4*	*5*	*6*
m 2	*5*	*7*	*9*	*11*
3	*13*	*29*	*61*	*125*
4	*7*	*61*	*61*	*125*

How might your implementation compare with a recursive algorithm?

5 Internal Files

5.1 READ and WRITE statements

In standard FORTRAN, READ and WRITE statements are not restricted to reading and writing to and from input/output units or files. They may also be used to read and write data to and from character strings. Suppose **string** is of type, say, CHARACTER*10. Thus, just as the line

```
read(1,'(i2,1x,i2)') m,n
```

reads values for reals **m** and **n** from file channel 1,

```
read(string,'(i2,1x,i2)') m,n
```

does the same from the characters in **string**,

```
string = '34 72'
read(string,'(i2,1x,i2)') m,n
```

returning values of 34 and 72. WRITE statements work in the same way and FORMAT statements can be used as you would expect. We say that the character string in each case acts as an *internal* or 'software' file. As is illustrated by the following example, such techniques can be very useful for reading parameters in from a file. List directed I/O (with a star as a format statement, FMT=*)

```
read(string,*) x,y
```

or

```
write(string,*) u,v,w
```

is, however, *not* supported as part of the FORTRAN 77 standard even if it is understood by many FORTRAN 77 compilers and Fortran 90. An example of the use of this extension is given in exercise 2 at the end of this chapter. More tricks for input/output are presented in chapter 15.

5.2 Pull – an example program using internal files

Pull is a simple program to search a file (attached to a channel **channl**) for a specified keyword. If the keyword is detected, it is then read and the remainder of the line to

its right is returned as a string **line**. This can then be read as an internal file. If the keyword is not detected the string **line** is returned as '!void'. This case should be tested against by the user. Consider

```
      subroutine pull(channl,key,line)
c
      integer channl,length
      character key*(*), line*80, inline*80
c
c     --- rewind file and measure length of search string
c
      rewind channl
      length = len(key)
c
c
c     ---- loop until you find the keyword
c
 1    continue
      read(unit=channl,fmt=1000,end=2) inline
c
      if (inline(1:length) .ne. key) then
         goto 1
         endif
c
c
c     --- if found copy the segment of the input string to the right
c         of the keyword to line. The remaining spaces are padded
c         to blank.
c
         line = inline(length+1:80)

      return
c
c
 2    continue
c
c     --- if the keyword is not detected, line is set to '!void'.
c
      line = '!void'
c
      return
c
1000  format(a80)
c
      end
```

and a simple demonstration program

```
      program Roger
c
      integer niceno
      real nicekg
      character data*80
c
      open(1,file='rogers_personal_file')
c
      call pull(1,'my cats name =',data)
      if (data .ne. '!void') then
         write(*,*) 'my cats name = ',data
      endif
c
      call pull(1,'my cats weight =',data)
      if (data .ne. '!void') then
         read(data,'(g10.3)') nicekg
         write(*,*) 'my cats mass =',nicekg
      endif
c
      call pull(1,'my cats number =',data)
      if (data .ne. '!void') then
         read(data,'(i3)') niceno
         write(*,*) 'my cats favourite integer = ',
     +              niceno
      endif
c
      stop
      end
```

For a typical data file **rogers_personal_file**,

```
      my cats weight = 100.45
      my cats number = 63
      my cats name = Roger
```

the following results were obtained:

```
      my cats name =  Roger
      my cats mass =  100.4499969
      my cats favourite integer =          63
```

Note the use of the string assignments in **pull** above. Writing for example

```
      line(1:5) = '!void'
```

sets the first five characters of the line to **'!void'** but leaves the remainder of the string, **line(6:80)**, unaffected. The assignment

```
      line = '!void'
```

assigns in contrast the first five characters **line(1:5)** but clears the remainder of the string to blanks.

5.3 Exercises

Internal files are often useful where input/output manipulations are not trivial. Where list directed I/O is supported (e.g. in Fortran 90 – see exercise 2), it can be especially powerful.

1. (*pull*) Modify the subroutine **pull** to read multiple entries from file of the form

2. (*expression parsing*) Fortran 90 and many FORTRAN 77 extensions support the use of list directed I/O with internal files. Exploiting this, write a real function **evalf** to evaluate expressions of the form $34.56 + 43.98$, $543 - 32$, $65 * 7$. You should recognise the operations $+$, $-$, $*$ and / but numbers of the form $7.89E+08$ need not be supported.

   ```
   point = 45. 67.
   point = 45. 68.
   point = 45. 69.
   line = 32. 12.4 43.9 76.0
   point = 12. 32.
   ...
   ```

3. (*printing an array*) Character arrays can also be written to as internal files. In this case, each new array element (a character string) is treated as a new line. Using this feature, write a subroutine to take an $n \times n$ array **A** and write its contents out, row by row to a character array suitable to be printed directly to standard output.

4. (*string handling*) Write a simple routine to convert arbitrary strings from lower or mixed to upper case.

6

Arrays in FORTRAN

6.1 'Row' and 'column' storage

Arrays in FORTRAN are little more than sequences of consecutive locations in storage. In machine terms they are, thus, no more than one dimensional strips. FORTRAN however supports two and higher dimensional[1] arrays accessed by two or more indices, e.g. **A(i,j)**, **A(i,j,k)** etc. To illustrate how such structures are stored, consider the following program. The final line '**write(*,*) A**' prints the elements of the array in the order they are stored in the computer.

```
      program arrays
      integer A(4,5),i,j
c
      do 1 i=1,4
         do 2 j=1,5
            A(i,j) = 10*i+j
 2          continue
 1       continue
c
      write(*,*) A
c
      stop
      end
```

Running this program, we see that an array **A(row,col)** is stored in FORTRAN column-by-column or *column-wise* in memory. In PASCAL by contrast, an equivalent array would be stored row-by-row or *row-wise*. The output of the above FORTRAN program should thus look something like

```
 11          21          31          41          12
 22          32          42          13          23
 33          43          14          24          34
 44          15          25          35          45
```

People sometimes compare the row- and column-representations of different languages claiming that one is more efficient than the other for matrix operations. Although the order in which array elements are stored and accessed is central to

[1] Up to seven in FORTRAN 77 and Fortran 90.

the efficient implementation of many codes, such comparisons are misleading. Firstly, the assignment of the array element **A(row,col)** to the matrix value $A_{row,col}$ is completely arbitrary. An association of **A(col,row)** would be equally valid and FORTRAN would then be a *row-wise* language. Secondly although many algorithms may be naturally expressed in forms better suited to, say, row operations, there are usually mathematically equivalent schemes equally efficient for column-wise storage. Thus, whereas distinctions between column- and row-wise storage can be very important for program efficiency, the differences between column- and row-wise storage between various languages amount to no more than differences between two different conventions.

6.2 Calling arrays in FORTRAN

In languages like PASCAL there are two ways to call arrays: by address or by value. Calling by value is the simplest. Each array is copied element-by-element in to a local variable in the called routine and then copied back to the calling routine on return. The second method, calling by address, is, however, potentially more efficient. Then, only the address of the first array element is passed, and, given any required information about the array's dimensions, is used by the called routine to calculate the addresses of the other elements as required. The routine can access the elements directly in storage without needing to copy them all back and forth across the parameter list. Unfortunately, the FORTRAN standard says nothing about the way in which arrays should be passed; in principle either is possible, but in practice nearly all compilers pass arrays by address. Indeed, it is usual to assume, as we do in this book, that this is always the case. Passing by address will be particularly important when we talk about BLAS (Basic Linear Algebra Subprograms) in the next chapter. To illustrate these ideas, consider the following examples. We first consider a simple code

```
      program arrays
      integer A(7,9),i,j,nrows,ncols
c
      do 1 i=1,7
         do 2 j=1,9
            A(i,j) = 10*i+j
 2       continue
 1    continue
c
      nrows = 7
      ncols = 9
c
      call one(nrows,ncols,A)
c
      stop
      end
```

which calls the subprogram

```
      subroutine one(nrows,ncols,array)
      integer nrows,ncols,array(nrows,ncols),i,j
c
      do 1 i=1,nrows
         write(*,'(9(i2,1x))') (array(i,j),j=1,ncols)
 1    continue
c
      return
      end
```

Subroutine **one** is called by program **arrays** which passes the array **A**, **nrows** and **ncols**. In particular from **A**, only the address of the first element **A(1,1)** is passed and the addresses of the remaining elements calculated using information provided by **nrows** and **ncols**. This is the standard method for passing arrays in FORTRAN. Equivalently, the call

```
      call one(nrows,ncols,A)
```

may be replaced by

```
      call one(nrows,ncols,A(1,1))
```

which has exactly the same effect, '**A**' and '**A(1,1)**' sharing the same address. Passing addresses other than this initial value is also possible. Replacing that of **A(1,1)** by **A(2,1)**, the second element in **A**, and printing out the first two columns by restricting **ncols** to two, yields the following results.

```
21 22
31 32
41 42
51 52
61 62
71 72
12 13
```

The entries have thus effectively been shifted up by one to start with the value 21 corresponding to the entry **A(2,1)**. This is what you would expect. The subroutine **one** is simply fooled into thinking that the array starts with element **A(2,1)** rather than **A(1,1)**. Since only an address is passed, it is also possible to treat a subroutine array as if it had a different number of dimensions than in the calling program. Thus in the above routine, **one** could be replaced by **two** below

```
      subroutine two(length,array)
      integer length,array(length),i
c
      write(*,'(25(i2,1x))') (array(i),i=1,length)
c
      return
      end
```

called by the lines

```
      read(*,*) start,length
      call two(length,A(start,1))
```

which refers to **A** as a one dimensional vector. In particular the first **length** elements starting from the **start**-th element are printed. Passing **A** as a simple vector does not affect its contents, only how they are accessed. Replacing **one**, for example,

```
      subroutine three(nrows,ncols,array)
      implicit none
      integer nrows,ncols,array(*),i,j
c
      do 1 i=1,nrows
         write(*,'(9(i2,1x))')
     +     (array(1 + (i-1) + (j-1)*nrows),j=1,ncols)
 1    continue
c
      return
      end
```

A two dimensional array can be reconstructed from a starting address passed across the parameter list and the appropriate dimensional information. Note the special importance of the value **nrows** to compute the addresses of the elements **array(i,j)**. We discuss this below.

6.3 The leading dimension

The above example illustrates how, for any 2D array, **A(n,m)**, given a starting address, to calculate the address of any array element **A(i,j)** in **A**, we need only the first array dimension, **n**. To calculate the location of an element in the array, the second is redundant. The address is simply that of the first element plus a displacement of

$$(i-1)+(j-1)*n.$$

Because of its special role, we refer to the first dimension as the *leading dimension*. It is always needed by the subroutine to correctly access the array. The second dimension is used merely to check the size of the array and so ensure that any called elements do not lie outside the area of storage allocated to the array by the type statement. Such 'run-time bound checking' is usually by default not enforced, being considered too expensive, and may have to be requested by a special compiler or debugging option.

6.4 Higher dimensional arrays and dummy dimensions

So far we have only talked about one and two dimensional arrays. In FORTRAN, however, up to seven are possible. Although the formulae for the displacements become a little longer, the basic principles remain the same. The final dimension is used only

for run-time array bound checking, whilst the preceding *leading dimensions* are used to calculate the displacement of each element from the starting address. In some applications, however, where it is known that there is no chance of accessing an array outside its bounds, it is often convenient to drop the final dimension completely and replace it with a suitable dummy parameter. One typical such example occurs for the function-evaluation function in an ordinary differential equation solver ODE code to return the derivative vector YP := Y′ defined by

$$Y' = f(t, Y(t)),$$

for the variables

```
double precision t, y(n), yp(n)
```

In such cases, it is usual to specify a dummy value for the (dummy) dimensions of the one dimensional Y and YP. This avoids the need to pass the true dimension as a parameter. Suitable values for the dummy parameter vary. Normally, however (the FORTRAN standard), a special symbol '*' is used which automatically suppresses run-time bound checks and the resulting error messages: **real A(7,6,*)** for **real A(7,6,3)**. Alternatively, on some systems, a value of one may be used. This, however, is not recommended. An example of array address calculations for a higher dimensional array is given in exercise 1 at the end of this chapter.

6.5 A note of caution

Dummy dimensions should be used with care. Not only do they remove the possibility of run-time array bound checks (which check for perhaps one of the most common sources of program errors) but they can drastically reduce the readability of codes. Often, the array dimensions give valuable clues to what a variable is and how it is used.

6.6 A harder example

Suppose we want to pass a rectangular subarray

$$S = \begin{pmatrix} a_{33} & a_{34} & a_{35} \\ a_{43} & a_{44} & a_{45} \end{pmatrix}$$

from the above 4×5 array A

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \end{pmatrix}$$

such that

$$\begin{pmatrix} s_{11} & s_{12} & s_{13} \\ s_{21} & s_{22} & s_{23} \end{pmatrix} = \begin{pmatrix} a_{33} & a_{34} & a_{35} \\ a_{43} & a_{44} & a_{45} \end{pmatrix}$$

Then calling

```
      call four(A(4,4))
```

we can write

```
      subroutine four(S)

      integer S(4,3), i, j

      do 1 i=1,2
         write(*,*) (S(i,j), j=1,3)
 1    continue

      return
      end
```

Note, here, that the subarray **S** has the *same* leading dimension as the original **A** in the calling program. This is so that the addresses of its elements can be correctly computed and their contents accessed. No matter how they are *referred to* within **four**, they are are still *stored* where **A** is defined, namely in the calling program. Thus, for example to move down one row (from s_{12} to s_{22} say) we need merely add 4 to the corresponding address, hence a leading dimension of 4.

To summarise, passing arrays in FORTRAN does not affect where or how their values are stored, only how they are accessed. Once storage has been defined, starting addresses and (leading) dimensions serve only to define a set of rules by which individual elements can be accessed. We return to this issue when we consider stride lengths and the BLAS in chapter 9 and skew array dimensions in the treatment of band arrays in chapter 11.

6.7 Exercises

Accessing array elements by their addresses rather than their subscripts is the key to many powerful programming techniques in Fortran. Exercises 1, 2 and 4 provide further illustrations of this.

1. (*higher dimensional arrays*) The 4D array

   ```
   logical Z(r,s,t,u)
   ```

   is passed to the subroutine **flat** as a one dimensional array

   ```
   logical ZZ(r*s*t*u)
   ```

   Write a scheme to address a general element **Z(a,b,c,d)** of **Z** stored as an element of **ZZ**.

2. (*skew dimensions*) Consider the following program

```
      program skew
      integer nrows1,ncols1,nrows2,ncols2
      parameter (nrows1=7,ncols1=9,nrows2=8,ncols2=4)
      integer a(nrows1,ncols1),i,j
c
      do 1 i=1,nrows1
         do 2 j=1,ncols1
            a(i,j) = 10*i+j
    2    continue
    1 continue
c
      call one(nrows2,ncols2,a)
c
      stop
      end
```

for various combinations of the parameters **nrows1**, **ncols1**, **nrows2** and **ncols2**. Subroutine **one** was defined in section 6.2 above. In particular, consider those values for which **nrows2** and **ncols2** do not correspond to the dimensions in the calling program. Can you predict the results? Try out this program yourself. Special techniques based on such effects can occasionally be useful. One example for sparse linear algebra is given in chapter 11.

3. (*a block algorithm*) **Mult2(A,ldA,B,ldB,C,ldC)** is a subroutine to compute the mathematical product AB of two 2×2 arrays, A and B, and add the result to a further 2×2 array, C. **ldA**, **ldB** and **ldC** are the leading dimensions of **A**, **B** and **C** respectively. By first splitting them into four 2×2 blocks, use this routine to compute the product RS of two 4×4 arrays, R and S. Would this be possible replacing **mult2** by a routine which computed only $AB \rightarrow C$?

4. (*an illegal example*) Making possible illegal out-of-bounds array violations, write a function **star5** such that calls of the form

```
star5(h,u(i,j))
```

return the five-point approximation

$$\frac{1}{h^2}\{-u_{i-1,j} + 4u_{i,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1}\}$$

to the Laplacian

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$$

at the internal points of an equally spaced mesh **real u(100,100)**.

7

Work Arrays and Array Partitioning

7.1 Passing storage

One of the most serious problems in FORTRAN77 is its inability to dynamically allocate arrays. As we saw in chapter 4, storage for arrays is allocated at compile time and so, once the program is running, cannot be added to or altered in response to problem parameters. Only parameter constants and not variables can be used to define array dimensions.

In FORTRAN77 the standard solution to this problem is to pass *work arrays*. These are fixed size arrays defined explicitly in the main program and passed to subroutines purely to supply them with the required storage from which to allocate (set up), at run time, internal array(s) in response to problem parameters as and when they are required. Using the properties of addresses and dummy variables introduced in the last chapter, it is clear that to set up such an array, the programmer need not in general know how these arrays are to be used or how much storage from them will actually be required. They must merely ensure that the passed array is of sufficient size to allocate the workspace required by the subroutine. An expression for calculating this figure (usually in terms of the problem parameters) is often supplied in the routines' documentation. An example of a work array is C in the NAG root-finder codes C05ADF and C05AZF discussed in sections 3.8.2 and 3.8.3.

7.2 Partitioning work arrays

Where two or more internal arrays are to be formed, often only one work array need be used. Suppose for example that we had written a routine

```
subroutine solve(n,A,B)
```

to solve a certain task which needed two work arrays, **real a(n)** and **real b(n)** where **n** was non-constant and known only once the program had started. Then using an intermediate routine **split** to set both arrays up, a single work array, **rwork** with dimension (at least) **2*n** could be used. That is, we could write

```
      program main
c
      integer n,m
      parameter (m=100)
      real rwork(2*m)
c
c     ... n must be .le. m
c
      call split(n,rwork)
c
      stop
      end
```

where **split** simply divides the storage

```
      subroutine split(n,rwork)
c
      integer n
      real rwork(2*n)
c
      call solve(n,rwork(1),rwork(n+1))
c
      return
      end
```

The elements **rwork(1)** to **rwork(n)** would thus be assigned to **A** and **rwork(n+1)** to **rwork(2*n)** to **B**. In more complicated programs, several different work arrays may be used for different types of variable; for example, real, double precision, integer and logical. Denoting these **rwork**, **iwork** and **lwork**, they could be passed along with integer scalar parameters, e.g. **drwork**, **diwork** and **dlwork** to give the size of the 'parent' arrays in the calling program (eg. **drwork** for **m** above). These can then be checked automatically by the called subroutine to see if they are sufficiently large from which to form the required internal arrays.

Where several internal arrays are required, a useful technique is to use the integer array (here **iwork**) to provide the starting addresses (in reality pointers) of each internal array. For example, supposed we wished to set up the following list of internal arrays where **n**, **m1** and **m2** are integer (non-constant) variables:

```
      integer k(n), a(m1), b(m2), c(m1*m2)
```

then we could write using an integer **addr**

```
      addr = 5
c
      iwork(1) = addr
      addr     = addr + n
c
      iwork(2) = addr
      addr     = addr + m1
```

```
c
      iwork(3) = addr
      addr     = addr + m2
c
      iwork(4) = addr
      addr     = addr + m1*m2
c
      if ((addr-1) .gt. diwork) then
         write(*,*) '** insufficient storage passed to iwork'
         stop
         endif
c
      call task(iwork(iwork(1)),iwork(iwork(2)),iwork(iwork(3)),
     +          iwork(iwork(4)))
```

Real and logical arrays etc. can be handled in a similar way. This technique also simplifies the access by the user of information embedded in work arrays. The elements of **b**, for example, are easily retrieved by a call

```
      call printb(n,b(iwork(3)))
```

to say

```
      subroutine printb(n,b)
      integer n
      real b(n)
c
      write(*,*) b
c
      return
      end
```

Alternative techniques are left to the reader. The importance of work arrays and array partioning in FORTRAN 77 should not be underestimated. Where storage requirements for algorithms vary with run-time parameters, they provide the only safe and flexible way to provide subroutines with the storage they need. The reader will, of course, recall that passing work arrays does not allocate storage. Total storage must be decided at compile time, and that allocated as a work array, decided and allocated in the calling routine before it is passed. Storage allocation in FORTRAN 77 is static. For complicated applications, multiple levels of work arrays and array partitioning are possible.

8
Memory Efficiency

8.1 The memory problem

In modern computers, the factors which limit peak performance are usually not the rate at which arithmetic and logical operations are performed but the efficiency with which data can be transferred to and from memory. Such issues are of particular importance for, say, linear algebra, where a basic set of simple and fast operations are repeated on a large number of data elements. Due to the central importance of linear algebra in many areas of numerical computation, such questions merit particular attention. Indeed, they turn out to be a central issue in modern computer design. As we now show, however, the speed of information flow to and from memory is also closely linked to the way in which data structures are represented and are accessed for computation. The efficient tuning of these parameters to obtain optimal data flow and performance is known as *memory management*. This is the subject of this chapter.

8.2 Memory hierarchy

Building computer memory is a compromise between capacity and access speed. Memory can either be small and fast — as for a CPU register, for example — or large and slow, like a hard disk. To accommodate large amounts of data whilst still maintaining fast access times and observing the problems outlined above, most systems employ a combination of different types of memory. A large store acts as a main albeit slow memory, whereas additional smaller and faster memory banks may be added to hold information currently being processed by the central control unit.

To increase efficiency, such intermediate memory is often layered in a hierarchical structure, smaller faster memory lying above, and closer to the CPU, than larger slower storage. Data values are then progressively passed upwards as they are required and returned downwards after being computed. Thus in this way, by accessing the top-most level of the hierarchy, the arithmetic unit can remain operational longer and need spend less time waiting for expensive main memory accesses. A schematic illustration of such a structure is given in figure 1. We refer to such a scheme as a *hierarchial memory structure*. The transfer of data between different levels is of course normally performed automatically and as such is invisible to the user. By structuring memory in this way, modern architectures seek to avoid the data bottle-necking problems outlined above. As one moves up the memory hierarchy, storage structures become progressively

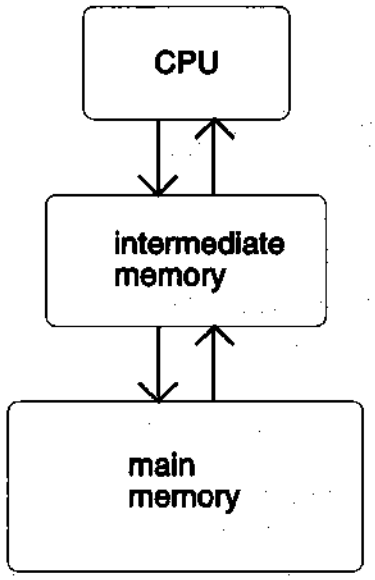


Figure 1. **A schematic representation of intermediate memory in a hierarchical storage structure.**

faster and the data values more relevant to the currently executing calculations. In writing FORTRAN programs our aim is simply to seek to ensure that the operations and constructs we use allow the memory management program to speed data up and down the memory hierarchy efficiently and thus keep the central processor fully active. This is the motivation for our discussion.

8.3 Cache memory

Although the choice of memory hierarchy can differ significantly from machine to machine, one commonly occurring feature is so-called *cache memory*. Situated between the main storage and the CPU, cache is a small high speed memory bank used to hold recently accessed or currently active data items. Since it is typically much faster than the larger main memory, it is hoped that by restricting data accesses as much as possible to values already in cache, considerable increases in memory performance can be achieved.

Technically, cache works by copying data blocks or values transferred to and from main storage. Every time a data transfer is made, a copy of the data is retained in cache in case it is needed again in the near future. Should this occur, it can be then accessed in a fraction of the time required for calls to normal main storage.

8.4 Data locality

Cache is important because it tries to exploit patterns in the order in which memory addresses are accessed. Recently called or changed locations, for example, have a much greater chance of being reaccessed than those as yet unread and consecutive accesses are more commonly applied to adjoining or nearby addresses rather than more dispersed locations. We refer to this as *temporal* (time) and *spatial* (address) data locality. For instance, where groups of variables are continuously reaccessed and updated, it makes little sense to copy them back and forth from slower main memory. A smaller, faster, holding storage like cache is obviously more efficient.

Spatial locality is exploited by replacing calls to single addresses by requests for groups or blocks of elements. Modern computers usually have special techniques for the high speed transfer of collections of elements in and out of store, and cache memory can often exploit these. By fetching blocks of consecutive elements likely to be needed in the future, total memory access times can often be reduced and comparatively expensive repeated scalar accesses avoided. Typically examples occur in, for example, linear algebra where matrices (arrays) can be loaded at once, or in sections, rather than elementwise as would otherwise be necessary.

Clearly, to make full use of the potential offered by cache, in writing FORTRAN programs it is important to try to localise array (address) calls wherever possible. This (spatial locality) is often enough to promote efficient cache use and data re-use in its own right. To illustrate the use of *spatial locality*, consider the following example. Further illustrations are given in the exercises at the end of the chapter.

```
      do 1 i=1,1024                    do 1 i=1,1024
        do 2 j=1,1024                    do 2 j=1,1024
          sum = Z(j,i) + sum               sum = Z(i,j) + sum
2       continue                 2       continue
1     continue                   1     continue
```

A **B**

Both loops perform the same task but they address the array elements in a different order. In A elements are accessed consecutively in column order as they appear in storage, whereas in B, they are accessed row-wise, every 1024th element at a time. Thus given a cache memory of size say $2^{10} = 1024$ elements[1], in principle, in A, only one cache read operation to and from main memory would be required per column whereas the loop B requires one access per element. Although this example is clearly idealised – in practice, fewer than 1024 elements would be read in at a time in A – it serves to illustrate an important point: cache failures or *cache misses* of the type illustrated by B are highly inefficient and should be avoided at all costs.

For optimal results, the programmer should try to limit the size of arrays so that they fit directly into cache. This can then ensure a minimum number of cache misses. Such methods are often possible using special *block* or *partitioned algorithms* which split an original large problem into a set of smaller subproblems with cache sized arrays. Examples of such algorithms are given in chapter 10.

[1] Real-life caches may be larger than this.

8.5 Memory stride

The above example illustrated the importance of the order in which array elements are accessed. We formalise this with the notion of a *stride length*. In loop A, the elements are accessed column-wise or consecutively in storage whereas in loop B, they are accessed row-wise, every 1024th at a time. Formally, we say that in example A, the array **Z** is accessed with *unit stride* or *stride length 1* whereas in B, it is accessed with *stride 1024* or it has *stride length 1024*. Alternatively we can say that a column in **Z** has stride 1 whereas a row has stride 1024. In general, the stride of any structure (e.g. a row or a column) in a host structure (e.g. a matrix) may be defined as the increment between the addresses of successive elements. Clearly, in the above example, accessing **Z** with a stride of 1 was much more efficient than with a stride of 1024.

8.6 Paging

In practice, a simple CPU cache – main memory hierarchy like that outlined in figure 1 may not be adequate for all applications. Where programs are too big to fit directly even into the larger main memory, special techniques may be used to back it up onto disk or other external media.

One technique for doing this is known as paging. By dividing the program's storage requirements into fixed size blocks of consecutive storage, and copying or swapping these as required back and forth from external media, paging simulates an extended or idealised memory potentially larger than that physically available on the machine itself. The whole process being managed by the operating system, this implements what is known as *virtual memory* and allows programs to run which would otherwise be too big to fit into standard memory[2].

I/O access to external media is however much slower than internal memory and, where excessive, can result in a marked loss of performance. Because of the costs they incur, in organising algorithms, care should be taken to avoid page swaps, usually referred to as *page faults* as far as possible. Fortunately, however, since page storage is consecutive, much the same principles of data locality and data reuse apply as for cache memory above. An illustrative example cited in [9] from Winter [29] is given in exercise 1 below.

8.7 Other architectures

It should be noted that not all architectures use cache memories. Some for example use special vector registers which can access main storage directly and efficiently. Although they are not discussed here in detail, the underlying issues remain the same.

[2] Virtual memory is particularly important for multiuser systems where more than one user can login at a time. Then, even if it were potentially sufficiently large, single programs are seldom given access to a the computer's entire internal memory. In such cases, virtual memory can thus be used to offer the user a *virtual machine* in which they have the illusion of having full access over the computer's resources. Here, as before, paging is under operating system control and is actually applied to the full range of addresses over which the user has access.

Memory efficiency depends strongly upon the distribution of memory accesses and this depends in turn on the way arrays are represented and the order in which their elements accessed. A comprehensive and readable introduction to memory, computer architecture and its implications is given in [14].

8.8 Exercises

The following exercises, motivated by matrix multiplication, may help illustrate some issues in cache and paging.

1. (*page faults*) Data locality can also minimise page faults. Following Winter [29], consider the following two loops:

```
(i)       do 1 i=1,n
            do 2 j=1,n
              do 3 k=1,n
                C(i,j) = A(i,k)*B(k,j) + C(i,j)
3           continue
2         continue
1       continue

(ii)      do 1 k=1,n
            do 2 j=1,n
              do 3 i=1,n
                C(i,j) = A(i,k)*B(k,j) + C(i,j)
3           continue
2         continue
1       continue
```

   to compute the sum/product $C = AB + C$ for $\mathbf{n = 1024}$. Assuming a page size of 65,536 (2^{16}) floating point elements, calculate how many different page accesses would be required to compute C in each case. If every page fault has an I/O cost of 0.5 seconds, estimate the corresponding overheads for the two loops. Can you explain which algorithm is the more efficient? How might still better results be obtained?

2. (*data reuse*) The previous exercise illustrated the costs of certain algorithms in terms of page faults. Re-address this question by considering the gains inferred by the main memory and the scope for data reuse.

9

Basic Linear Algebra Subprograms (BLAS)

9.1 Introduction

The natural language for many mathematical and scientific operations is that of linear algebra, and the natural objects matrices and vectors. Indeed, in many applications it is these operations that consume the most computing time. Many computer vendors thus provide special facilities and architectures to perform such operations very efficiently. These include the memory hierarchies discussed in the previous chapter, and vector and command pipelining which we discuss below. To help the user make effective use of these often complicated features, a range of special standardised numerical software template routines have been developed to perform a range of basic operations between scalars, vectors and matrices needed for linear algebra. These are known as the Basic Linear Algebra Subprograms, or the BLAS[1]. Computer manufacturers are requested to provide a set of FORTRAN routines conforming to these standards for each one of their machines. Often being written in low level languages, such as machine code, these are highly optimised for the machines for which they are intended. Thus, by using these routines, it is thus hoped that users may freely move codes from one system to another without having to retune basic linear algebra calls, but knowing that each one has been fully optimised (by the manufacturer written BLA routines) for the new system. The BLAS thus represent a major contribution to program portability, and together with the basic strategies outlined in the last chapter, allow users to automatically take full advantage of differing computer architectures without needing to extensively re-write numerical codes. They also make way for more structured programming. The advantages of using BLAS may be summarised as follows:

- they allow the user to call highly machine-optimised codes written by computer manufacturers and hence take full advantage of machine architectures;
- they lead to improved portability since codes written with BLAS may be moved between different systems without modification for different architectures. This also applies to system migration and maintenance. If a manufacturer upgrades their system or updates their version of their BLAS library, any user-written codes written with them will also be updated;

and finally,

[1] Alternatively, abbreviating *basic linear algebra* as *BLA*, we can refer to them as *BLA routines*.

- they lead to increased program readability. By forcing the programmer to break up the program into natural vector or matrix operations instead of nests of DO-loops, the resulting codes are often better structured and easier to read. The use of the BLAS is thus a good example of modular programming.

Currently only dense (full) matrices are supported by the BLAS but work is underway to extend this to the more difficult sparse case [24]. A listing of the BLAS is given in appendix A whereas a simple and practical introduction appears in Coleman and Van Loan [5].

It should be stressed, however, that the BLAS only address basic linear algebra operations such as multiplications, updates and triangular solves. For higher level operations such as solving linear systems, QR factorisation, eigenvalue problems or band matrices, libraries like NAG [37], LINPACK (see [5]) or LAPACK [1] should be used. We discuss LAPACK in chapter 12. The BLAS, however, have a central rôle in modern numerical software and illustrate a number of important principles. For these reasons, they are discussed in this text.

Where no optimised manufacturer or corresponding appropriate third-party supplied BLAS are available on a given machine, a set of generic FORTRAN77 routines can be obtained over *Netlib* discussed in section 3.5. These may give increase portability and legibility but may fail to give the full gains associated with architecture specific machine optimised codes.

9.2 An overview

The BLAS naturally divide into three groups, depending on the type of objects they operate upon:

- **level-1 BLAS:** *vector* and *vector-vector* operations – scaling, copying, vector norms, scalar products, sums and differences etc.,
- **level-2 BLAS:** *vector-matrix* operations – matrix multiplication, rank-one updates, triangular solves etc. and
- **level-3 BLAS:** *matrix-matrix* operations – matrix-matrix multiplication, rank-k updates and matrix triangular solves with multiple right-hand sides.

Clearly those from the first group are simpler whilst those from the second and third are more powerful. Level three BLAS are still, however, quite new and may not be supported on all machines.

9.3 Level one BLAS – SCOPY

As a simple example, consider the basic linear algebra subroutine SCOPY, designed to copy one vector to another:

```
scopy(n,x,incx,y,incy)
```

BLAS typically come in four variants, single precision real (S), double precision real (D), single precision complex (C) and double precision complex[2] (Z); their type is designated by the first letter of their name. Thus SCOPY is a copy routine for single precision reals whereas DCOPY and CCOPY perform the same operation for double precision real and single precision complex numbers respectively. Otherwise they are identical. To refer collectively to all four routines a generic name _COPY may be used or, as is more natural, the single or double precision name quoted. We shall follow the latter convention. A call to _COPY thus takes the form:

```
call scopy(n,x,incx,y,incy)
```

where **x** is the source vector with vectorstride **incx**, **y** the target vector with stride **incy** and **n** is the number of elements to be copied. **x** and **y** are here, of course, single precision real and **n**, **incx** and **incy** are integers. For normal use, **incx** and **incy** should be set to one. For other values, different results can be obtained. These can be best understood by thinking of the way the matrices are stored on the computer. Further results can be obtained by passing different starting addresses for **x** and **y**. We illustrate these effects below. The arrays **x** and **y** are assumed declared as **real x(100), y(50)**. Try these examples out for yourself.

call	*copies*	*range*
scopy(20,x,1,y,1)	x(i) to y(i),	i=1,20
scopy(15,x(10),1,y(20),1)	x(9+i) to y(19+i),	i=1,15
scopy(10,x(11),1,x(1),1)	x(10+i) to x(i),	i=1,10
scopy(50,x(1),2,y(1),1)	x(-1+2i) to y(i),	i=1,50

Further examples appear in figure 1. Zero strides are also possible:

```
call scopy(n,0.0,0,y,1)
```

copying the one dimensional length vector 0.0 (that is a scalar) **n** times to **y** starting at **y(1)**. It thus sets the first **n** elements of **y** to zero.

9.4 Negative strides – a caution

BLAS may not act as you expect for negative strides. The addresses passed as **x** and **y** are interpreted strictly as the *least* addresses in the respective arrays and *not* as their 'starting address'. Hence in the following program

```
      program minus
      integer i
      double precision x(20),y(20)
c
      do 1 i=1,20
         x(i) = i
```

[2] Where supported. Double precision complex numbers are not part of the Fortran standard.

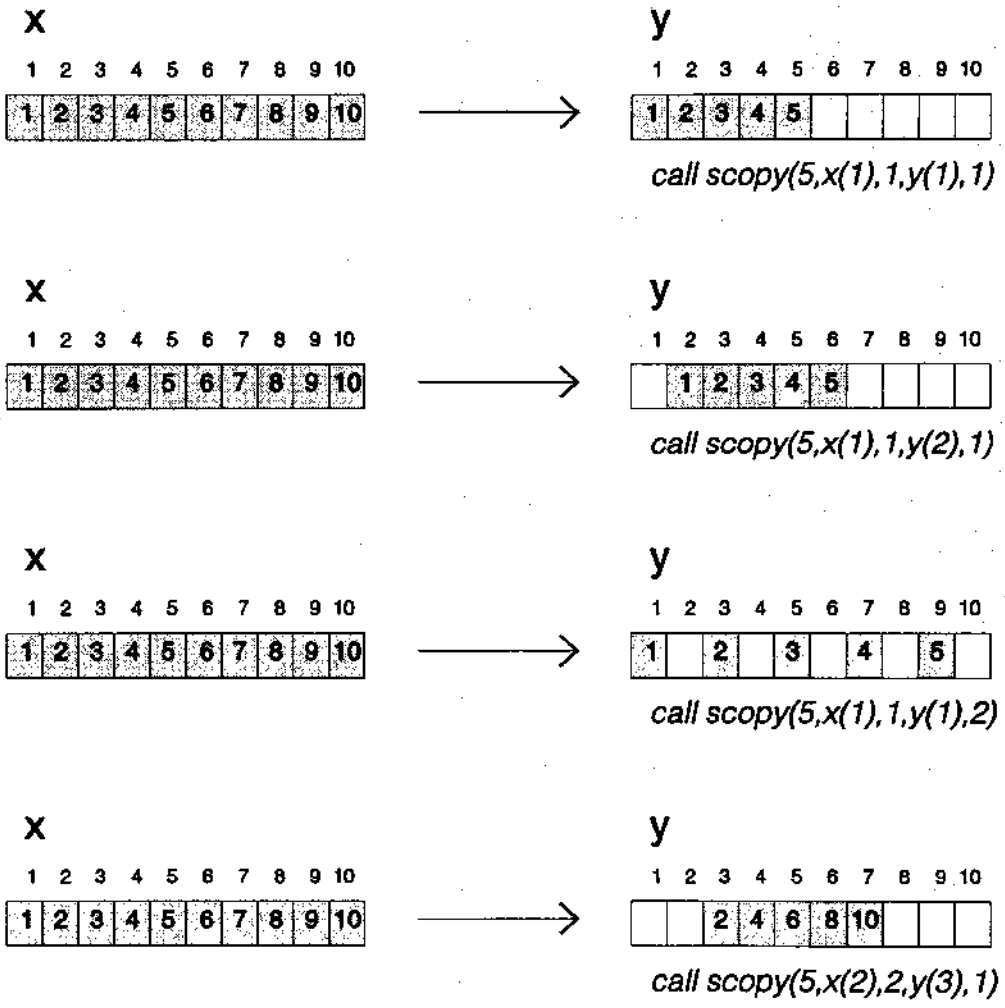


Figure 1. **SCOPY – simple examples using strides.**

SCOPY is used to copy one vector onto another but by changing the source and target vector strides and starting addresses, various results can be obtained.

```
            y(i) = -1
 1      continue
c
       call dcopy(10,x(11),-1,y(11),1)
c
       write(*,'(10(f5.1,2x))') y
c
       stop
       end
```

setting either **incx** or **incy** to -1 merely has the effect of reversing the order in which the elements are copied. Thus the output is

```
 -1.0   -1.0   -1.0   -1.0   -1.0   -1.0   -1.0   -1.0   -1.0   -1.0
 20.0   19.0   18.0   17.0   16.0   15.0   14.0   13.0   12.0   11.0
```

and the elements **x(1)** to **x(10)** are *not* read.

9.5 Accessing rows and columns

Just as columns in two dimensional arrays can be thought of as vectors with vector stride one (each element following directly after another), rows too can be considered as vectors, this time with a stride of **n**, **n** being the array's leading dimension. Hence for a matrix

```
       real a(5,6)
```

the calls

```
       call scopy(5,a(1,1),1,col1,1)
       call scopy(6,a(1,1),5,row1,1)
```

return the first column and row respectively. Equivalently

```
       call scopy(5,a(1,6),1,col6,1)
       call scopy(6,a(5,1),5,row5,1)
```

return the last column and row. Operations of this type are clearly invaluable in numerical computations. For example, using the double precision routine

```
       ddot(n,x,incx,y,incy)
```

to compute the inner product

$$x^T y = \sum_{i=1}^{n} x_i y_i$$

of two double precision arrays **x** and **y** strides **incx** and **incy** respectively[3], the sequence

$$x(i) = \text{ddot}(n, A(i,1), n, x(1), 1), \quad i = 1 \ldots n$$

[3] That is, ddot(n,x,incx,y,incy) $:= \sum_{i=1}^{n} \text{x}(1+(i-1)*\text{incx}) \cdot \text{y}(1+(i-1)*\text{incy})$.

computes the vector

$$y = Ax$$

where **A**, **x** and **y** are double precision of dimensions $n \times n$, n and n respectively. Further applications of **ddot** are given in exercise 2 below. More elegant methods for selecting 2D sections from arrays are discussed for Fortran 90 in section 19.4. Other examples of level 1 BLAS include

- **DSWAP** swaps two vectors $y \leftrightarrow x$,
- **DSCAL** premultiply by a scalar $y \leftarrow \alpha y$,
- **DAXPY** premultiply and add $y \leftarrow \alpha x + y$ and
- **DNRM2** compute the 2-norm dnrm2 $\leftarrow \|x\|_2$.

For further details see appendix A.

9.6 Exercises

The following serve as simple introductory examples to level one and two BLAS.

1. (*negative strides*) Consider once more the example in section 9.4. Can you predict the results **y** of

   ```
   call dcopy(10,x(11),incx,y(11),incy)
   ```

   for (incx, incy) = $(\pm 1, \pm 1)$? Try out each case and compare your results.

2. (*DDOT*) Using the level one BLAS DDOT, compute the product $AB = C$ for

   ```
   double precision A(n,m), B(m,r), C(n,r)
   ```

   This operation is supported directly by the level three BLAS DGEMM. Calling sequences for both these routines are given in appendix A.

3. (*Gaussian elimination*) Two examples without pivoting:

   (a) Using the level one BLAS DAXPY and DSCAL, write a routine to perform the factorisation

   $$A = LU$$

   by Gaussian elimination without pivoting. **L** (lower triangular, ones on the diagonal) and **U** (upper triangular) should be overwritten directly into **A** as the factorisation advances.

   (b) Now rewrite your routine replacing the calls to DAXPY by a call to the level two routine, DGER.

   For a description of DAXPY, see again appendix A. Calling sequences to DSCAL and DGER, together with a solution to using DGER, appear in section 11.5.1.

10

Linear Algebra and Vectorisation

10.1 Higher level BLAS

10.1.1 Why level three BLAS?

Higher level BLAS — that is level 2 and 3 BLAS — which perform *vector-array* and *array-array* operations, can often be more efficient than level 1 routines. To see this, consider the ratio of work measured in terms of arithmetic operations to the amount of data, and hence required data movements, for level 1, 2 and 3 BLAS respectively. Taking for simplicity n and $n \times n$ arrays we obtain[1]

BLAS	*amount of data*	*amount of work*
level 1	$O(n)$	$O(n)$
level 2	$O(n^2)$	$O(n^2)$
level 3	$O(n^2)$	$O(n^3)$

We thus see that for large n, this ratio is potentially much larger for level 3, thereby in principle allowing more active data to be kept closer to the top of the memory hierarchy for any given amount of computation and so minimising the ratio of memory calls to arithmetic operations. Given the observations of chapter 8, this suggests that they can lead to significant improvements in machine performance. In particular, it suggests that wherever possible, we should thus aim to write our algorithms in such a way as to make most possible use of level 3 BLAS and matrix-matrix algorithms. This is confirmed by the following megaflops speeds [9] obtained on selected high-performance cache based single processor machines [10, 3]:

machine	SUN 50 MHz SuperSPARC[2]	*IBM System/6000* *530*(POWER)	*590*(POWER2)	*Cray-2*
level 1 : $ax + y \rightarrow y$	**12**	**10**	**60**	**143**
level 2 : $by + aAx \rightarrow y$	**19**	**27**	**157**	**363**
level 3 : $bC + aAB \rightarrow C$	**41**	**46**	**250**	**455**

A *flop*, or floating point operation, denotes the scalar operation of the form $y := a * x$ or $z = x + y$, and a rate of one megaflop is a rate of one million such operations per

[1] $O(n^3)$ reads *of order* n^3. That is, for large n, $O(n^3)$ grows as n^3.
[2] Using third-party written software from Dakota Scientific Software Inc.

second[3]. The reason for the gains in the level two BLAS is explained in exercise 1 at the end of this chapter. Note that the above figures, which are rounded up to the nearest integer, may not represent the true state of the art for the above computers or their corresponding manufacturers. The values for the SUN SuperSPARC are, for example, included as a reference standard.

10.1.2 Block algorithms

In the light of these results, it is often better to solve large systems in terms of level 3 matrix-matrix operations. In doing this, however, as pointed out in section 8.3, care should be taken to ensure that these matrices are not so large as not to fit into cache memory. To avoid such problems, and thus make full use of level 3 BLAS, a special class of so-called *block* or *partitioned* algorithms have been developed. Consider, for example, the LU-factorisation

$$A = LU$$

where L is lower and U upper triangular. Given A we look for possible L and U. This 'standard' technique is widely used as a first step to solve linear systems of the form $Ax = b$. Splitting A into 2×2 blocks we can write

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ & U_{22} \end{pmatrix}$$

where L_{11} and L_{22} are lower, and U_{11} and U_{22} upper triangular. Rewriting this as

$$\begin{pmatrix} L_{11} & \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ & U_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & \\ L_{21} & I \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ & A' \end{pmatrix}$$

where $A' = L_{22}U_{22}$, it is thus clear that the system may be factorised inductively. Writing

$$\begin{aligned} A_{11} &= L_{11}U_{11} \\ A_{12} &= L_{11}U_{12} \\ A_{21} &= L_{21}U_{11} \\ A_{22} &= L_{21}U_{12} + A' \end{aligned}$$

this suggests the following so-called *right-looking* algorithm:

1. factorise $A_{11} \rightarrow L_{11}U_{11}$
2. solve $L_{11}^{-1}A_{12} \rightarrow U_{12}$
3. solve $A_{21}U_{11}^{-1} \rightarrow L_{21}$
4. update $A_{22} - L_{21}U_{12} \rightarrow A'$

This process can then be repeated inductively on A' until A' is sufficiently small to be solved efficiently by standard unblocked methods. We illustrate this by figure 1.

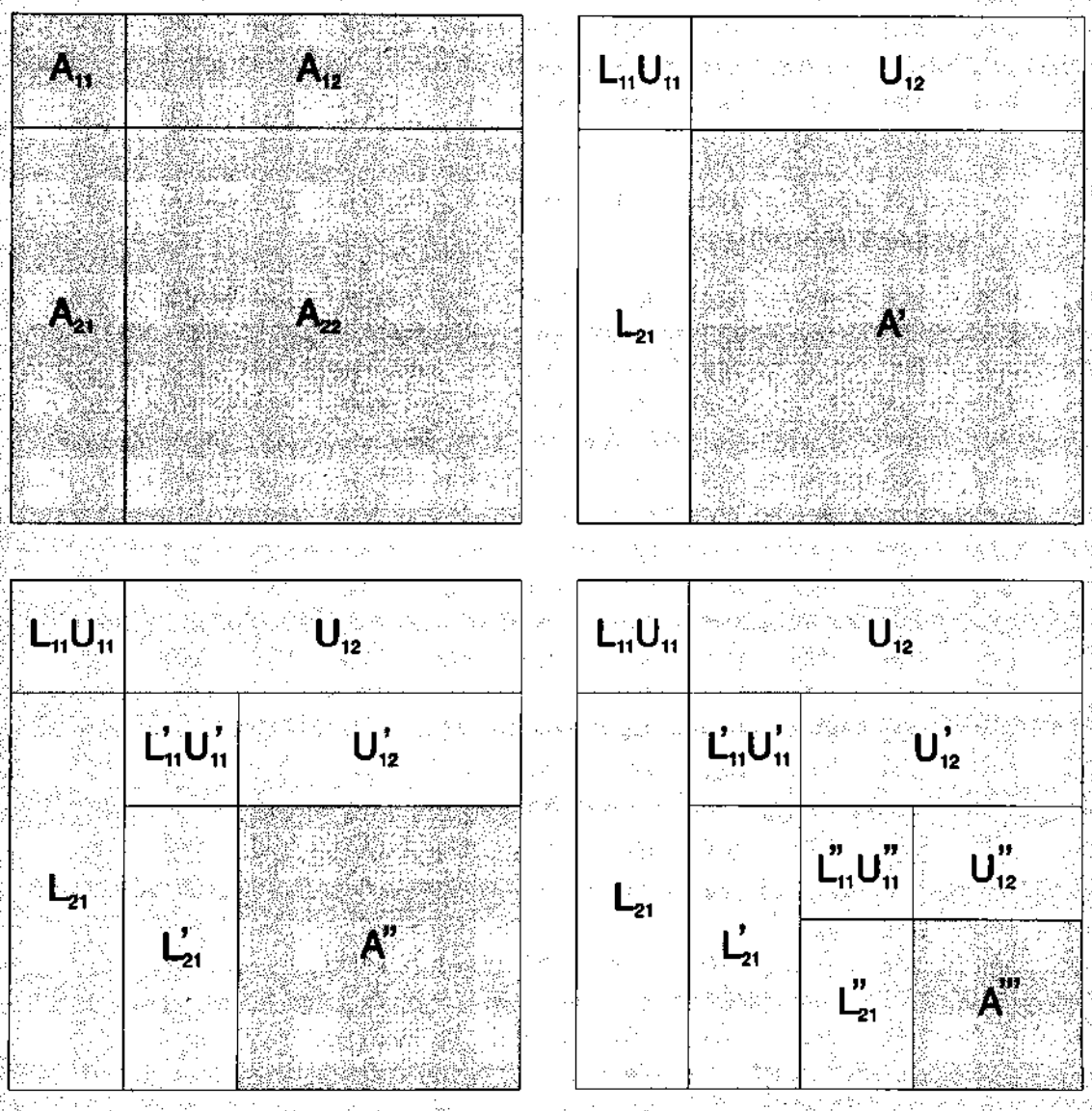


Figure 1. **Partitioning A for the right-looking algorithm.**
The contents of A after 0, 1, 2 and 3 block stages. In each case the elements still to be calculated are shaded in grey. At the end of the factorisation, the array contains the two factors L and U.

Such right-looking algorithms are however not unique and a number of other schemes have also been proposed based on similar techniques, but which do the numerical operations in different orders. These are all mathematically equivalent, as they are to standard non-block methods, but have different memory access properties. By selecting an appropriate scheme, codes can be constructed to break up large problems into manageable matrix subproblems whose size and distribution can be selected to optimise data movement to and from main memory and thus make full use of level 3 BLAS. Such techniques can also have considerable advantages for parallel computers. For further details see [9]. Block algorithms were used extensively in the LAPACK project [1]. Further examples of block algorithms for LU-factorisation and of matrix multiplication are given in the exercises at the end of this chapter.

10.1.3 High performance machines, parallel computing and sparse systems

On some high performance machines[4] which do not use cache but which have instead special high speed communication channels to and from memory, high performance rates may in fact be obtained directly even for some level one vector-vector and level two vector-array operations. At first sight, this would appear to diminish the importance of level three routines. However, for the now common multiprocessor versions of these machines, high level BLAS are still recommended, as they can serve as a powerful aid to parallelisation. BLAS are now frequently supplied and optimised for parallel architectures (parallel compilation directives are even included in standard 'generic' versions) and can, particularly in the case of level 3 operations, give considerable scope for automatic parallelisation of linear algebra kernels. A discussion of this in connection with the rôle of the BLAS in the LAPACK project is given in [1].

Although the BLAS so far discussed are provided only for full matrices, work is still being done to extend them to sparse systems. This is considerably more difficult since special storage schemes and conventions are required and these can vary widely between problems. This work is thus still at a research stage. One prototype package appears in [24].

[3] These values may in some cases exceed the clock frequencies of the corresponding processors. In such cases, where they are not dependent, the CPU can issue more than one machine-code instruction per clock cycle. Such machines are known as *superscalar*. This is closely related to pipelining, see [14].

[4] Two such examples are a CRAY Y-MP and CRAY C-90 which obtained single processor megaflop rates of (293,314,314) and (766,902,903) [6] respectively in the tests of section 10.1.1. These values, as before, do not necessarily represent the state of the art of the above two machines.

10.2 Vectorisation

To further speed up linear algebra, a special class of computers known as *vector processors* have been developed. Having nothing to do with parallel computing, vector processors seek to optimise execution by improving the way in which data and instructions are fetched into the arithmetic unit. We discuss this in more detail below. In particular they are designed for vector and matrix type operations. Vector architectures are also frequently associated with a faster clock cycle and simplified instruction sets.

10.2.1 Pipelining

Consider the following *scalar* addition:

$$c := a + b.$$

Over and above the operations required to fetch values a and b from store and return the result c, three stages are required to perform the addition itself, each of which takes one clock cycle:

1. align exponents (for a and b) ready for the addition,
2. add $c := a + b$ and finally
3. normalise the result c.

The exponent alignment is similar to lining up, say, two numbers 12345.678 and 0.123 when you add them by hand (it is easier if the tenth's and the hundredth's etc. lie over one another) and the final normalisation is required to regain the normalised form[5] $0.f_1f_2f_3\ldots f_p \times b^e$ used internally by the computer. This is clearly very inefficient. For the *vector* addition

$$\boldsymbol{z} := \boldsymbol{x} + \boldsymbol{y},$$

for example, only one component could be processed every three clock cycles and the hardware *functional units* responsible for the stages 1, 2 and 3 thus spend two thirds of their time inactive. To avoid such inefficiencies, vector processors have special architectures designed to allow different functional units to operate on different array elements at the same time. As soon as a unit has finished, its (intermediate) result can be passed, or piped, directly to the next unit which can start work immediately. The first unit can then start work on the next element. Thus there is no longer any need for any unit to wait until the final result has been obtained until it can start to process the next number. A structure of this type is referred to as a *pipeline*, a schematic representation of which is given in figure 2.

After an initial startup time needed to configure the pipeline and reach a steady state where all three elements are active (i.e. to *fill* the pipe), in this simple example, one z component is computed per cycle instead of one every three cycles as for a normal architecture. Typically such a pipeline could be set up by a single vector instruction. We refer to computer architectures which make use of such pipeline structures, often

[5] See exercise 5, chapter 1.

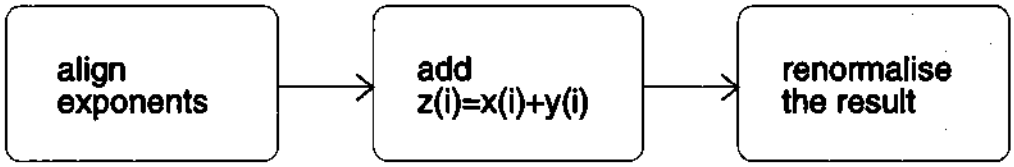


Figure 2. **A schematic representation of a pipeline for the addition of two vectors.**

After one cycle, for example, whilst the second, or 'add' unit was performing the first addition $z(1) := x(1) + y(1)$*, the first unit, 'align', could start aligning the elements* $x(2)$ *and* $y(2)$ *prior to their addition. After a further cycle, the first unit could then align* $x(3)$ *and* $y(3)$ *whilst the added unit added* $z(2) := x(2) + y(2)$ *and the final 'renormalisation' unit realigned* $z(1)$*.*

combining them with special structures to transfer data to and from main memory, as *vector architectures*.

Two analogies

Many analogies can be given to illustrate vector pipelines. The most common is a conveyer belt on a factory floor. Suppose in a car factory two workers work on a conveyer belt: first, worker A fits the door before the car is carried to worker B, who fits the windows. In a 'non-pipelined' factory worker A must wait for worker B to finish before he can start on a new car. In a 'pipelined' factory, however, worker A can start on a new car as soon as the old car has gone to worker B. He no longer needs to wait. Thus twice as many cars are produced (or doors and windows fitted), one every time the conveyer belt moves. Most factories are, of course, pipelined.

A second commonly used analogy is an escalator. Here, each step corresponds to a functional unit. Then, following the pipelined analogy, to go up the stairs you don't have to wait for the person in front of you to reach the top. You can wait just until there is a free step for you to place your feet on. The *non-pipeline* analogy would suggest that you have to wait for the person in front to reach the top before you can step on. Thus, once more, escalators are usually pipelined being, once full, n times faster than a non-pipelined analogy, n being here the number of steps.

10.2.2 Chaining

Sometimes, two different vector operations can be linked together into one long pipeline. This is true for 'SAXPY BLAS' operation[6]

$$\boldsymbol{y} \leftarrow \alpha * \boldsymbol{x} + \boldsymbol{y}$$

referred to as a *linked triad*. Once the pipeline is full, two flops, $y_i \leftarrow \alpha x_i + y_i$ are produced for every clock cycle. Linked triads such as the above are important since they occur widely in numerical computations.

10.2.3 Command pipelining and Risc architectures

Just as vector pipelining looks ahead for the next data items in a vector data stream, most computers now perform similar checks to look ahead for the instructions which are likely to be executed next and configure any required operands. This, which may be thought of as instruction or command pipelining, can often greatly improve efficiency. Indeed many modern computers now rely very heavily on such techniques combined with fast clock cycles and simplified instruction sets to achieve their high performance. Notable examples are the Risc (Reduced Instruction Set) computers which by removing unnecessarily complicated instruction-set instructions make way for smaller processing units with faster on-chip communication. For an in depth discussion, see [14] and [27].

[6] See appendix A.

10.3 Using vectorisation

10.3.1 When is it desirable?

Although vectorisation frequently enhances program performance, for small loops it can sometimes be less efficient. A certain cost (or *startup time*) is associated with setting up a vector pipeline and if this is high relative to the size of the vector(s) to be processed, then a scalar implementation may be faster. On an IBM 3090/VF, for example, the crossover or threshold point for

$$y \leftarrow \alpha * x + y$$

is about 12. Corresponding values vary for different computers and different operations but, to be cost effective, vectorisation typically requires long(ish) vectors on which to operate. At the very least, these should be longer than the cross-over or threshold point illustrated above. Furthermore on many computers, which have vector registers, there is also an *optimal vector length* for vectorisation processes. In this case you may wish to ensure that your vector length is equal to, or is a multiple of this number. For further details you should refer to your system or compiler manual.

10.3.2 How to vectorise a loop

This varies from system to system. Given an appropriate compiler flag, many compilers will however attempt to identify suitable candidate structures for vectorisation and vectorise them automatically. A vectorisation report is then usually returned back to the user detailing what vectorisation was attempted and what was achieved. Since, however, the compiler will not in general know about the likely lengths of vectors and DO-loops (these may not even be established at compile time) this process is far from fool-proof and help is normally required from the user. This is usually supplied in the form of compiler directives, specific vectorisation commands included in the source code but disguised as FORTRAN comment lines. These typically tag loops for vectorisation, mark loops not to be vectorised and give information about likely array or loop sizes. Alternatively, they can supply information about program structure or logical interdependencies which can aid the automatic vectorisation facility. For specific details, or information about the existence of any special vectorisation tools, you should once again consult your local site documentation. An example of the use of compiler directives in *High Performance Fortran*, HPF, is given in chapter 22.

10.4 Level one BLAS

Operations like those performed by level 1 BLAS are prime candidates for vectorisation. It is now, moreover, widely accepted that modern compilers are so sophisticated that if clearly presented with level one tasks, they can perform automatically all the tricks deployed by the relevant BLAS without incurring the corresponding additional subroutine call overheads. Thus, on strict efficiency grounds, for level one operations there is an argument for using straightforward DO-loops instead of BLAS. Further, for the reasons discussed in this chapter, on many systems

the principal gains in program efficiency are often only made using level 2 and level 3 BLAS. For this reason, the use of these routines is still strongly recommended. Extra support array handling in Fortran 90 is described in chapter 19.

10.5 Exercises

Exercise 1 explains why level two BLAS perform typically better than level one. Other examples include further algorithms for matrix factorisations.

1. (*level 2 BLAS*) Level two BLAS are typically much faster than level one routines because they have a lower ratio of storage accesses to floating point operations. As an example, work out the number of flops and read/write accesses for the following problems and, assuming $n = m = k$, write down their approximate ratios:

BLAS	*operation*	*routine*
level one	$y = y + \alpha x$	DAXPY
level two	$y = \beta y + \alpha Ax$	DGEMV
level three	$C = \beta C + \alpha AB$	DGEMM

   Full solutions are of course given in the solutions in appendix C. Assume here that $x \in R^n$ and $y \in R^m$ and that A, B and C are $m \times n$, $n \times k$ and $m \times k$. All arrays are double precision. How do the results for levels 1 and 2 compare with those for level 3?

2. (*memory management*) Find out how intermediate memory is structured on your computer. How are commands and data values read into memory? Is a cache used? If not, are any alternative techniques deployed?

3. (*vectorisation speed-up*) Try timing the following two operations

   (a) $\mathbf{y = a*x + z}$

   (b) $\mathbf{y = x}$

   for vectors of different lengths and plot your results as two graphs. If vector/pipeline options exist, try it with them selected and then deselected. Is there a crossover point below which vectorisation is no longer efficient? Is there an optimal vector length? Before making your comparisons, be sure that you have selected all the appropriate compiler optimisation options.

4. (*level one BLAS*) Are level one BLAS faster than standard DO-loops? Try this out for yourself on your machine with arrays of differing lengths. Once again, don't forget to select all the appropriate optimisation options. Without doing this, your comparisons may not be meaningful.

5. (*right looking Cholesky*) Where A is symmetric, $A = A^T$, the LU-factorisation $A = LU$ reduces to $A = LL^T$ and special algorithms can be used. Rewrite the right looking block algorithm given in section 10.1.2 for this case. How many operations does it save?

6. (*a left looking algorithm*) At any stage in the right looking algorithm outlined in section 10.1.2 above, the as yet unfactorised elements form a submatrix at the bottom right of the matrix under factorisation. Other schemes are however possible. Starting with a partition of the form

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ & U_{22} & U_{23} \\ & & U_{33} \end{pmatrix},$$

and assuming that the factorisation $A_{11} = L_{11}U_{11}$ and the blocks L_{21} and L_{31} are already known, derive an expression for the new block L_{12}. Then use this to compute U_{22}, L_{22} and L_{32}. Are the remaining elements of the matrix affected by this algorithm? Explain how these expressions could form the basis of a practical algorithm. Compare the memory access pattern of this scheme with that of the right looking factorisation given in the text. In both cases you should assume no pivoting.

7. (*block implementation*) Consider again the right looking algorithm of section 10.1.2. At each stage we can seek to partition the submatrix so that A_{11} remains small and the factorisation $A_{11} = L_{11}U_{11}$ can be performed efficiently in high speed memory. Given that the other blocks A_{12}, A_{21} and A_{22} are not so small, what measures can be taken for the rest of the algorithm?

11

Gaussian Elimination

11.1 A simple example

To conclude our discussion of the BLAS, we consider two algorithms for the factorisation of a square $n \times n$ matrix A

$$A = LU$$

where L is lower triangular with ones on the diagonal and U upper triangular. In particular we look at Gaussian elimination without pivoting [13]. Under certain circumstances this is stable and has various storage advantages over pivoted schemes. For example, for certain classes of sparse matrices, it can be shown to have better fill-in properties. Although we do not describe the details of the algorithm in full (see however figure 2 at the end of this chapter), we note that at every stage r, $r = 1 \ldots n$, the task of doing the row operations amounts to a single rank-one update. That is an operation of the form

$$\tilde{A}^{(r)} \leftarrow \tilde{A}^{(r)} - \alpha \boldsymbol{x} \boldsymbol{y}^T$$

where y^T is the row to be subtracted and αx the vector of scalar multipliers, α being one over the pivotal element. Numerically, this can be implemented by a single level 2 BLAS call to the routine DGER. Followed by a second BLA call, this time to the level 1 BLAS DSCAL to record the multipliers which define the lower diagonal[1] L, and one division $\alpha = 1/\tilde{A}^{(r)}_{rr}$ to compute the multiplier, these make up all the floating point operations required to advance the factorisation. Details of these two routines are given at the end of this chapter in sections 11.5.1 and 11.5.2. As usual, for full matrices, no additional storage is required beyond that to represent A, elements from L and U being overwritten onto A directly as the factorisation advances. This approach leads to a very compact albeit terse code:

[1] Alternatively, the scaling by DSCAL could *precede* the update and the entry -**mult** in DGER be replaced by −1.0. This brings no computational advantage and was here considered less clear.

```
      program gauss1
c
      integer n,row,col
      parameter (n=4)
      double precision A(n,n), mult
c
      read(*,*) ((A(row,col),col=1,n),row=1,n)
c
c
c
      do 1 row=1,n-1
c
c        ... perform rank one update (to advance U)
c
         mult = 1.0/A(row,row)
         call dger(n-row,n-row,
     +             -mult,A(row+1,row),1,A(row,row+1),n,
     +             A(row+1,row+1),n)
c
c        ... record multipliers (to advance L)
c
         call dscal(n-row,mult,A(row+1,row),1)
 1    continue
c
c
c
      do 2 row=1,n
         write(*,*) (A(row,col),col=1,n)
 2    continue
c
      stop
      end
```

This is a typical application of BLAS calls. Note how in particular the subarray $\tilde{A}^{(r)}$

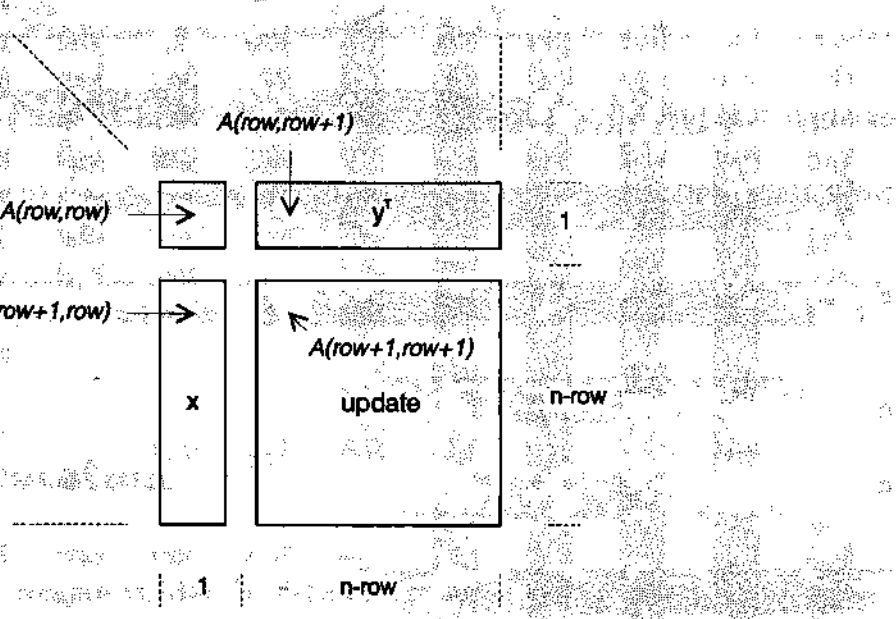


Figure 1. **Configuration of subarrays for the update $\tilde{A} \leftarrow \tilde{A} - xy^T$.**

is accessed in terms of starting addresses and array strides. We summarise this below:

name	*object*	*size*	*starting address in A*	*stride in S*
x	subcolumn	*n-row*	A(row+1,row)	*1*
y^T	subrow	*n-row*	A(row,row+1)	*n*
$\tilde{A}^{(r)} \leftarrow \tilde{A}^{(r)} - xy^T$	subarray	*(n-row)*^{2}	A(row+1,row+1)	*1 and n*

11.2 A harder example

Banded matrices commonly arise in many fields of science and engineering [13]. Despite of their different structure, they can, however, surprisingly often be solved by full matrix biased algorithms with only very minor modifications. Consider, for example, the matrix A:

$$\begin{pmatrix} 2 & 3 & & \\ 6 & 13 & 1 & \\ & 8 & 7 & 1 \\ & & 5 & 3 \end{pmatrix}$$

This is a simple tridiagonal problem. Representing it as a 'flattened' 3×4 matrix B

$$\begin{pmatrix} 0 & 3 & 1 & 1 \\ 2 & 13 & 7 & 3 \\ 6 & 8 & 5 & 0 \end{pmatrix}$$

we see how the elements from the same row in the original matrix occur as every other element in the new representation. That is to say rows in A have stride 2 in B. Columns, occur, of course consecutively. Thus passing a matrix of the form B to our

solver and a leading dimension **n-1** of 2 instead of 3, the routine can still be made to work. Care must be taken to restrict the size of the rank-one update (in this trivial example it is just a 1×1 block delineated by the variable 'size') and to offset the call to a by passing **A(2,1)** in order to avoid the leading zero in B, but it does nevertheless work. This is illustrated by the following program:

```
      program gauss2
c
      integer row,col
      double precision B(n-1,n)
c
      read(*,*) ((B(row,col),col=1,4),row=1,3)
c
      call sgauss(4,1,B(2,1))
c
      do 1 row=1,3
         write(*,*) (B(row,col),col=1,4)
 1    continue
c
      stop
      end
```

which calls

```
      subroutine sgauss(n,hwidth,A)
c
      integer n,length,row,col,size
      double precision A(n-1,*), mult
c
c
      do 1 row=1,n-1
c
c        ... perform rank one update (to advance U). Size is the
c            size of the block to be updated.
c
         mult = 1.0/A(row,row)
         size = min(n-row,hwidth)
         call dger(size,size,
     +             -mult,A(row+1,row),1,A(row,row+1),n-1,
     +              A(row+1,row+1),n-1)
c
c        ... record multipliers (to advance L)
c
         call dscal(size,mult,A(row+1,row),1)
 1    continue
c
      return
      end
```

Apart from the line

```
size = min(n-row,hwidth)
```

this routine differs only marginally from that called by **gauss1** above. Its only significant difference is the replacement of the stride n by n-1 in **A**'s type statement and the call to DGER to reflect the true stride of rows of A in the compressed matrix B. The parameter **hwidth**, the half bandwidth here passed as one, gives the number of non-zero diagonals above and below the diagonal in A. In this example, these are both assumed to be equal.

11.3 Interpretation

To move around a 2D array, from one element to another, only two numbers are required: the *row* and *column* strides. In FORTRAN these are the leading dimension and the value one respectively. They may be thought of as 'navigation rules'. To move one element down in the array, we need merely add one (the column stride) to the address. To move one element to the right, we add merely the leading dimension (or the row stride). Storing arrays in compressed form merely alters these rules, in the above example reducing the leading dimension (or row stride) by one. The arrays can however be 'navigated' as before. Care should only be taken to avoid out-of-bound, or 'incorrect' accessing: elements should only be accessed which are physically present in the new compressed representation.

11.4 Conclusion

Non-standard techniques for passing arrays can be invaluable in special applications. By correctly understanding how arrays are defined and accessed, apparently difficult problems can be solved by simple algorithms. Although they may at first appear daunting, the reader is strongly advised to try to understand such examples since a full understanding of the basic principles they employ is central to a good understanding of the use and representation of arrays in FORTRAN itself.

11.5 Subroutine references

11.5.1 DGER – a one-rank update of a general matrix

Given an $n \times m$ matrix A, a constant α (alias **alpha**) and two vectors x and y with lengths m and n, DGER performs the rank-one update

$$A \leftarrow A + \alpha x y^T.$$

This is effected by a call

```
call dger(m,n,alpha,x,incx,y,incy,A,ldA)
```

where **x**, **y**, α and **A** are all double precision whereas **m**, **n**, **incx**, **incy** and **ldA** are integers. As before for DCOPY (section 9.3), **incx** and **incy** are the respective strides of **x** and **y** whereas **ldA** is the leading dimension of **A**. The single precision version of DGER is SGER.

11.5.2 DSCAL – scales a vector by a constant

DSCAL rescales an n element double precision vector x by a double precision scalar α (alias **alpha**)

$$x \leftarrow \alpha x$$

by a call

```
call dscal(n,alpha,x,incx)
```

The integer **incx** is here the stride of **x**. The single precision version of DSCAL is SSCAL.

```
      program factor

      integer n
      parameter (n=4)

      integer row,col,i,j
      double precision A(n,n),mult
c
      read(*,*) ((A(row,col),col=1,n),row=1,n)
c
      do 1 row=1,n-1
c
      mult = 1.0/A(row,row)
c
      do 2 i=row+1,n
         do 3 j=row+1,n
            A(j,i) = A(j,i) - mult*A(row,i)*A(j,row)
3        continue
2     continue
c
      do 4 i=row+1,n
         A(i,row) = mult*A(i,row)
4     continue
c
1     continue
c
c
      do 5 row=1,n
      write(*,*) (A(row,col),col=1,n)
5     continue
c
      stop
      end
```

Figure 2. **Gaussian elimination without pivoting.**

A program to read in and factorise an array $A = LU$. At the row*-th stage, multiples of the* row*-th row are subtracted from the rows* row+1 *to* n *such that the corresponding portion of the* row*-th column is cancelled out. At the end of the factorisation, the upper half triangle of* **A** *contains the U factor. After each stage the row operations are recorded by the following DO-loop. This advances the L factor.*

Note that this code (chosen to match the following BLAS version) is not optimal. The product, **mult*A(row,i)**, *for example could be removed from the innermost loop and the L update incorporated in the previous code.*

12

LAPACK

12.1 Introduction

The BLAS provide simple, machine independent and efficient methods to perform such elementary vector and matrix operations as addition, multiplication and back substitution. For higher level problems such as matrix factorisations, eigenvalue problems or singular value decompositions where no suitable BLAS exist, the LAPACK library is recommended. This may be obtained free of charge over *Netlib*[1]. Although the source codes provided are the same for all machines, their extensive use of BLAS, combined with the use of block algorithms (see section 10.1.2), enables them to automatically take full advantage of differing machine architectures[2]. In fact, the development of LAPACK was a major incentive for the development of the BLAS. For a fuller introduction to LAPACK, the reader is referred to [1]. NAG users will note that LAPACK is currently being integrated into the F07 and F08 chapters of the NAG library.

12.2 Driver routines, computational routines and auxiliaries

For any given task or matrix type, the available LAPACK routines can be divided into three groups:

- **driver routines** which solve standard problems such as solving a linear system or computing the eigenvalues of a real symmetric matrix,
- **computational routines** which perform distinct computational tasks such as LU-factorisation or the back solution of a triangular system, and
- **auxiliary routines** which perform a range of low-level operations or subtasks required by the above routines. These may also be of use to professional software developers.

Wherever possible, driver routines are recommended for general use, but computational and auxiliary routines may be of value where additional functionality

[1] Discussed in section 3.5.

[2] This point should not be understated: LAPACK relies heavily on an efficient implementation of the BLAS on the host machine. Ideally this should be manufacturer supplied.

is required. Further, for most applications, two levels of driver are provided: a simple easy-to-use routine for normal use and an expert driver for added functionality.

12.3 Driver routines

Although general sparse structures are not supported, LAPACK provides routines for a wide range of dense and banded matrix types. These are indicated systematically by the second and third letters of the routine's name, the first letter being used to define the data type. For example the double precision (D) routines

DGESV, DPOSV, DGBSV and DSTSV

provide simple drivers[3] to solve *general* (GE) and *positive definite (symmetric)* (PO), *general banded* (GB) and *symmetric tridiagonal* (ST) systems respectively. The remaining letters (SV) are used to denote the operation to be performed, here to solve a linear system. Alternatively, single, complex and double precision complex[4] variants may be selected by the letters S, C and Z. The range of matrix types supported by LAPACK is listed in table 1. To illustrate this, consider solving the system

$$Ax = b$$

for x given b and a (double precision) matrix

$$A = \begin{pmatrix} 3 & -1 & & \\ -1 & 3 & -1 & \\ & -1 & 3 & -1 \\ & & -1 & 3 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

Storing A in full, we may do this directly, for example, using the general system solver DGESV. Since A is however *symmetric*, not all of its elements need to be stored and such a call would be somewhat inefficient. Noting that A is also positive definite, passing the elements

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ * & a_{22} & a_{23} & a_{24} \\ * & * & a_{33} & a_{34} \\ * & * & * & a_{44} \end{pmatrix}$$

as a 'full matrix', or as

$$A = (\; a_{11} \quad a_{12} \quad a_{22} \quad a_{13} \quad a_{23} \quad a_{33} \quad a_{14} \quad a_{24} \quad a_{34} \quad a_{44} \;)$$

in so-called 'packed form', it can now be solved by one of the two Cholesky solvers DPOSV or DPPSV respectively. Here the stars '$*$' indicate elements which are not accessed by the solver ('don't care') whereas the subscripts a_{ij} refer back to the original matrix written in full. Using DPOSV, for instance, we can make a call of the form

[3] The corresponding expert drivers are denoted by appending an extra 'X': DGESVX, DPOSVX, DGBSVX and DSTSVX.

[4] Where supported by the site FORTRAN compiler: complex double precision is not part of the FORTRAN77 standard.

Table 1. **Matrix types supported by LAPACK.**

code	*matrix type*
BD	bidiagonal
GB	general banded
GE	general matrix
GG	general matrix pair (generalised problem)
GT	general tridiagonal
HB	Hermitian, banded (complex)
HE	Hermitian (complex)
HG	upper Hessenberg-triangle pair (generalised problem)
HP	Hermitian, packed storage (complex)
HS	upper Hessenberg matrix
OP	orthogonal, packed storage (real)
OR	orthogonal (real)
PB	symmetric or Hermitian positive definite, banded
PO	symmetric or Hermitian positive definite
PP	symmetric or Hermitian positive definite, packed storage
PT	symmetric or Hermitian positive definite, tridiagonal
SB	symmetric banded (real)
SP	symmetric, packed storage
ST	symmetric tridiagonal (real)
SY	symmetric
TB	triangular banded
TG	tridiagonal pair (generalised problem)
TP	triangular, packed storage
TR	triangular or possibly quasi-triangular
TZ	trapezoidal
UN	unitary (complex)
UP	unitary, packed storage (complex)

Most routines in LAPACK are labelled according to a 5 or 6 letter code **xyyzz** *or* **xyyzzz** *where* **x** *is the data type,* **yy** *one of the above matrix types and* **zz** *or* **zzz** *the computational task to be performed.*

```
call dposv(uplo,n,nrhs,a,lda,b,ldb,info)
```

given suitable parameter settings. Symmetric matrices represented in lower triangular form

$$\begin{pmatrix} a_{11} & * & * & * \\ a_{21} & a_{22} & * & * \\ a_{31} & a_{32} & a_{33} & * \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

can also be handled simply by setting the above CHARACTER*1 flag, UPLO, to 'L' or 'Lower' instead of 'U' for 'Upper'. Besides requiring less storage, such special routines for symmetric arrays are more efficient, requiring only half as many operations as a normal algorithm. Further gains can be made by noting that the above A is in fact a band matrix. Using the (symmetric) compact storage scheme

$$A = \begin{pmatrix} * & -1 & -1 & -1 \\ 3 & 3 & 3 & 3 \end{pmatrix} = \begin{pmatrix} * & a_{12} & a_{23} & a_{34} \\ a_{11} & a_{22} & a_{33} & a_{44} \end{pmatrix}$$

we can then use the symmetric positive definite band solver DPBSV. Similar techniques can be used for unsymmetric matrices. Indeed, since it is in fact symmetric positive definite tridiagonal we could alternatively use the specialised tridiagonal routine DPTSV. As illustrated by this example, LAPACK leaves the user free to select an appropriate type of algorithm and storage method best suited to their problem.

12.4 Computational routines

Driver routines work by calling a sequence of computational routines to perform distinct computational tasks. For example to solve the system

$$Ax = b$$

the driver DGEVS calls the routines

- **DGETRF** to factorise[5] $A = PLU$ where L is unit lower triangular, U upper triangular and P a permutation matrix, and
- **DGETRS** to use the above result to compute x by backward and/or forward substitution.

The generic names of these routines are xyyTRF and xyyTRS where the letter 'x' denotes the data type and 'yy' the type of matrix. Further information about the factorisation can be provided, for example, by the following routines:

- **xyyCON** estimates the condition number $\kappa(A) = \|A\|\|A^{-1}\|$,
- **xyyRFS** computes forward and backward error bounds and an iterative refinement for the solution, or

[5] Or $A = LL^T$, for example, where A is symmetric positive definite.

- **xyyTRI** uses the factorisation computed by xyyTRF to form A^{-1}. This routine is not supported for band matrices since their inverses do not preserve band structure,

or by the expert solver DGESVX. Computational routines thus provide greater flexibility.

12.5 Other problems

LAPACK is not limited to solving linear systems: more general tasks are also addressed. Drivers are provided, for example, for linear least square fitting, eigenvalue problems and singular value decompositions. We list these drivers (table 2) together with the names of the matrix types which they support (table 3). More specialised operations, such as QR or LQ factorisations, and operations defined on a wider range of matrix types, are provided by the computational and auxiliary routines. Tables 2 and 3, like the rest of the discussion in this chapter, refer to release 2.0 of the LAPACK package.

Table 2. **LAPACK driver routines.**

driver	*problem*
VS	linear equations
LS, LSS	linear least square problems
LSE, GLM	generalised linear least square problems
EV, EVD	eigenvalue problems
ES	Schlur decomposition
GV	generalised eigenvalue problems
GS	generalised Schlur decomposition
SVD	singular value decompositions

For common computational tasks, a range of driver routines are provided. These are listed here, the driver names corresponding to the letters **zz** *or* **zzz** *in table 1. For each task, the list of supported matrix types is given in table 3. Where present, expert drivers may be selected by appending an additional 'X', e.g. DGESVX.*

Table 3. **Driver routines: supported types.**

driver	*supported matrices*											
VS	GB	GE		GT	PB	PO	PP	PT		SP		SY
LS, LSS		GE										
LSE, GLM			GG									
EV, EVD		GE							SB	SP	ST	SY
ES		GE										
GV		GE							SB	SP		SY
GS		GE										
SVD		GE	GG									

This table lists the range of supported matrix types for each type of driver. Where appropriate, variants for complex matrix types – HB, HE and HP – are also provided.

13

Programming Considerations

There are in general two kinds of program: those you write for yourself and those you write for others. Your own, of course, are your own affair but when writing for others, special care should be taken to ensure that they are well commented and well laid out. This is generally common sense but you may find some of the following tips useful. We motivate our discussion by comparisons with other languages.

13.1 FORTRAN and other languages

FORTRAN 77 is a relatively old language. Some people (mainly computer scientists) argue that it is also a relatively bad language. It was written, they claim, before people really knew how to write compilers and so, by implication, computer languages. Thus, it is claimed, it is fundamentally flawed. 'Nicer' modern languages designed to promote good programming techniques are often recommended. Despite their tone, such discussions are nevertheless of general interest for FORTRAN programmers. By looking at more advanced languages we can often get clues about how to write better FORTRAN programs. This is the aim of this chapter. Indeed, as we will see later, many of the features we will discuss have been introduced into the Fortran 90 standard.

13.2 Strong typing

Many modern languages are strongly typed. That is, all variables must be explicitly declared by the programmer in the program source. This not only makes codes clearer, but can also help trap errors. A famous FORTRAN 77 story concerned a DO-loop

```
      do 1 i=1,3
         call fire(i)
 1    continue
```

used to control the rockets on an American spacecraft. On data-entry, the first line was mistyped as `do 1 i=1.3`, a comma replacing the full stop. Since spaces are not significant in FORTRAN 77, and since in particular no strong typing was enforced, the statement was still valid and the error remained undetected, the code being taken as equivalent to

```
      do1i = 1.3
      call fire(i)
 1    continue
```

In the absence of a DO-statement, the variable **i** was not set, and the CONTINUE statement having no further meaning, no DO-loop was executed, the subroutine **fire** was called only once and the spacecraft went off into outer space. Regardless of the truth of such stories, errors of this type can usually be detected by strong typing. If type checking had been enforced would the variable **do1i** have been otherwise defined? If not, a compiler error would have resulted and the mission would have completed successfully. It is for this reason that strong typing is to be so highly recommended. In FORTRAN 77, programs may be strongly typed by appropriate compilation options or by the non-standard IMPLICIT NONE, now included in the Fortran 90 standard.

In more modern languages, with user defined types and operations, the effect of strong typing can be still stronger. Defining, for example, two variables, **x** and **y**, of two different user-defined types **miles** and **kilograms** with the appropriate user-defined numerical operations defined upon them, meaningless expressions such as **x+y** can be detected. Discussions of user-defined types and user-defined operations on user-defined types in Fortran 90 are given in section 18.2 and section 18.3.

13.3 Modules and packages

The standard interface in a FORTRAN 77 library is the subroutine or function. In some languages, Fortran 90 and Ada [18], for example, this idea is generalised by a *module* or a *package*. Once opened a module allows the programmer to use a specified list of functions, subroutines, data, operators and types. Other routines or values used internally by the module may be declared private and so may not be used: they remain invisible. Thus, using this idea, programmers can define the exact nature of the interface between the software library and the potential user. This situation is ideal. Contrast it for example with FORTRAN 77 where care must always be taken over hidden subroutines or COMMON BLOCKs.

In Ada, for example, complex numbers (not supported in the language standard) might be defined by such a package. It could first define the type complex as a type of 2-tuples (x, y) with meaning $x + iy$, then a set of functions to carry out the standard arithmetic operations, and finally redefine the standard symbols $+$, $-$, $/$, $*$ and $**$ in terms of the above functions when applied to complex types. In Fortran 90, such techniques are also supported through the *module* and *interface* structures. We return to these in sections 17.2, 17.3 and 18.3 where the concept of an *internal routine* is also discussed.

13.4 'Goto's considered harmful'

How and where should the GOTO statement be used in FORTRAN? Is it always bad style? To answer this, consider the following discussion.

13.4.1 Repeaters and completers

Many modern languages make extensive use of *begin-end* blocks. They act as brackets to group together several statements to act logically as one statement. This greatly simplifies command syntax. For example a command

```
do i=1..10 <statement>
```

could be applied to several statements by writing

```
do i=1..10
   begin
      <statement 1>
      <statement 2>
        ...
      <statement n>
   end
```

Given such block structures, several languages support two very useful commands: **repeat** and **complete**. These, known technically as *repeaters* and *completers*, effectively jump backward or forward to the start or the end of a logical block of code. In many languages, the command **complete** is known as **exit**[1]. For example the code

```
do forever
   begin
      read data
      print data
      if end_of_file then complete
   end
```

or alternatively

```
begin
   read data
   print data
   if more_data then repeat
end
```

would in such a language cause data to be read until the end of the file. In general, *begin-end* blocks may be named, and *repeat* and *exit* (completer) statements can be used to act on named blocks. This allows for block nesting. Such constructions are very powerful and are considered good programming style.

13.4.2 Goto's in FORTRAN

Goto's are a central feature in FORTRAN, but can often result in unreadable code. Many people therefore insist that they should be avoided at all costs. This view was

[1] This is the case for Fortran 90. See section 20.1. Named exit statements (see below) are also supported.

put forward in a famous paper *'Go to statement considered harmful'* [7]. Unfortunately such a slogan can often be misleading. Where they, for example, implement the completers and repeaters described above, goto's are examples of good programming style. Hence the counter slogan: *'Goto's considered harmful considered harmful'*. In a sentence, goto's should be avoided at all costs except when they are used to implement such good language features such as repeaters, completers or WHILE and REPEAT statements – two further structures not supported by FORTRAN 77[2]. Indeed, our objective in discussing such constructs is merely to emulate such good programming practice in FORTRAN. This is a good motivation for studying other languages. A fuller discussion of the use of goto statements in structured programming appears in [16].

13.5 RETURN and ENTRY statements

Multiple RETURN or STOP statements are also disliked by computer scientists. They maintain that if you write a program or subprogram correctly, it will only ever need one stop or return statement. Again this is related to the *begin-end* block idea. Blocks of code are entered at begin and should be exited at the end which should be the only logical exit point. This is thought to result in more modular coding. Whether or not you wish to confirm to such programming norms is, however, an open question. You may find, for example, multiple RETURN or STOP statements clearer. Multiple entry points to the same routine (a strange FORTRAN feature using the ENTRY statement) are, however, best avoided.

13.6 On and computed goto's

Like goto's, computed goto's such as

```
goto (100,102,104,106,...) i
```

which branch to the **i**-th listed statement depending on the value of **i**, and arithmetic-IF's such as

```
if (value) 100,102,104
```

which goto 100, 102 or 104 depending on the sign of **value**, are also to be discouraged. The latter form is often misunderstood by other programmers and may be hard to read. Ideally, you should use an IF-THEN-ELSE structure instead. One possible exception is for reverse communication, see chapter 4, where computed goto's are generally accepted.

[2] In fact, current thinking maintains that features such as WHILE and REPEAT are better coded directly with repeaters and completers. DO-WHILE is a new *obsolescent* feature in Fortran 90.

13.7 DO-loops and CONTINUE statements

The CONTINUE statement is a frequently under-used FORTRAN feature. Besides FORMAT statements, it should be the only statement attached to a statement number. Not only does it improve readability, but it also makes programs easier to edit. It, for example, can lessen the risk of editing away a statement number if an active statement is removed. For DO-loops for example

```
      do 1 i=1,10
        print*, i
1     continue
```

would be preferred to

```
      do 1 i=1,10
1       print*,i
```

For this reason, where two levels of DO-loop are used, two continue statements should be used. With suitable indentation this also makes the program more readable.

13.7.1 DO-END DO

On many systems the non-standard DO-END DO construct is supported. Instead of

```
      do 1 i=1,10
        print*, i
1     continue
```

we can write

```
      do i=1,10
        print*, i
      end do
```

which requires no label or CONTINUE statement. Such a notation is clearer and easier to edit. It removes, for example, the need to renumber DO-loops on copying. This, supported as part of the US Military standard cited in section 2.7, is now incorporated in the new standard, Fortran 90.

13.7.2 Loop variables

When using the DO-loop construction, the loop counter should be treated with care. For instance, in the above example, after the loop has finished execution, does i equal 10 or 11? In FORTRAN, its value is actually assigned but in other languages i would only be defined within the scope of the loop itself and it would be an error to try to access it from outside. With the exception of the case where you jump out of a loop (i.e. following a completer), a similar philosophy should be followed in FORTRAN.

13.8 Passing constant parameters – an important note

By definition, parameters should not be rewritten. Care is however needed when calling subroutines since, if it is lacking, precisely this may result. Suppose for example that a subroutine **sub** is called:

```
call sub(a,b)
```

where **a** is a parameter. Then if **sub** attempts to redefine **a** by an assignment (**a** = 8983.2 say), unexpected results may occur. If the error is not detected, since **sub** acts directly on the address of the parameter passed from the calling program, the subroutine may indeed reassign the value of the parameter constant **a**. This is clearly very dangerous[3]. An even more bizarre effect may occur during

```
call sub(1.0,b)
```

in which the 'value' of '1.0' itself can be altered. To avoid such problems, one common technique is to pass all such values through *dummy* or *temporary* variables assigned before each call. For example

```
      first = 1.0
      second = b
c
      call sub(first,second)
```

Setting out programs in this way is clearly safer and also makes them easier to read. The additional programming and runtime costs are negligible.

[3] Such effects may occur even with otherwise good compilers, e.g. XLF [30].

14

Code Optimisation

When first writing a code, care should be taken to ensure that it is well structured and easy to read. This helps, for example, with debugging and program verification. Sometimes, however, when efficiency or speed is important, steps can be taken to improve or 'optimise' performance at the expense of readability. This is the subject of this chapter. Such revisions can often be performed automatically by a suitable compiler pre-processor or a programming tool and may give some insight into how programs are actually compiled. We give here, where appropriate, examples using the IBM XLF compiler version 2.2 [30].

14.1 Simple techniques

14.1.1 Polynomials

How would you code the following polynomial?

$$a_1x^3 + a_2x^2 + a_3x + a_4$$

You might be tempted to write

```
a1*x**3 + a2*x**2 + a3*x + a4
```

which has 8 multiplies and 3 additions. A more efficient implementation of this polynomial would however be

```
((a1*x + a2)*x + a3)*x + a4
```

which requires only 3 multiplies and 3 additions. This latter representation is thus to be preferred. Note how we here assume that **x**3**, say, is implemented as **x*x*x**. This is now usual with most modern compliers, being more efficient than the older implementations of the type **log(exp(x)*3.0)**. To be completely safe, however, you should always use the longer form, **x*x*x**, and obviously always avoid statements of the type **x**3.0**.

14.1.2 Statement functions

Whenever a subroutine or a function is called, a certain cost is always incurred. Indeed when the routine is small and called many times, this can be unduly large. In such

cases it is more efficient to seek to avoid unnecessary function calls. To this end, for simple operations, *statement functions* may sometimes be used. These act like normal functions but appear as *one line* in the program unit in which they are to be called. They are placed after the declarations but before the executable statements. For example the Euclidian norm in R^2 might be coded

```
      real a,c,b,d
      real dist,x,y
c
      dist(x,y) = sqrt(x*x + y*y)
```

after which it could be referred in the relevant subroutine as a normal function. Statement functions differ from normal functions in so far as

1. they are only defined within the routine in which they are declared and have no meaning outside,

2. they are usually not compiled separately but substituted in-line[1] into the code automatically by the compiler whenever they are called. Thus, using statement functions is the same — no slower, no faster — as writing the code out in full each time. Function call overheads are thus eliminated.

It is for this second reason that they are recommended. Branching or similar statements are however not allowed in statement functions.

14.1.3 In-line expansion

As a last resort, at the expense of readability, and more significantly of length, programs can be re-written by removing calls to entire subprograms and expanding the corresponding code directly in-line into the calling program. This is usually very messy but can often be done automatically using an appropriate pre-processor or a programming tool. For example, using the IBM XLF compiler pre-processor facility, the following program:

```
      program inline
      double precision x,y,z,dist,norm
      external norm
c
      read(*,*) x,y,z
      dist = norm(x,y,z)
c
      write(*,*) dist
c
      stop
      end
```

with subroutine

[1] Although this is common practice, it is not required by the standard.

```
      double precision function norm(x,y,z)
      double precision x,y,z
c
      norm = sqrt(x*x + y*y + z*z)
c
      return
      end
```

was expanded automatically to

```
C*****************************************************************
C   Translated by IBM AIX XL FORTRAN -Pv Preprocessor/6000
C   Version 02.02.0000.0000 3.05F16 on  9/30/94 at 19:04:08
C*****************************************************************
C
      program inline
      double precision x,y,z,dist,norm
      external norm
      DOUBLEPRECISION NORM1X
c
      read(*,*) x,y,z
C*****  Code Expanded From Routine:  NORM
c
      NORM1X = SQRT(X*X + Y*Y + Z*Z)
      DIST = NORM1X
C*****  End of Code Expanded From Routine:  NORM
c
      write(*,*) dist
c
      stop
      end
```

This is faster, but harder to read. Note how additional local variables and comments were generated automatically by the pre-processor. Such changes are often only realistically feasible using automatic tools which are also likely to be less error prone.

14.1.4 Program substitution

In contrast it may sometimes be sensible to substitute blocks or in-line code by subroutine calls. For example using the same tool, this time specifying in addition an ESSL[2] preprocessor option, blocks of FORTRAN code can be replaced by appropriate manufacturer's-supplied BLAS. The routine

```
      program essl
      integer i,j,k
      double precision a(100,100),b(100,100),c(100,100)
c
```

[2] See section 3.3, here an IBM manufacturer's implementation of the BLAS.

```
      read(*,*) b,c
c
      do 1 i=1,100
         do 2 j=1,100
            a(i,j) = 0.
            do 3 k = 1,100
               a(i,j) = a(i,j) + b(i,k)*c(k,j)
 3          continue
 2       continue
 1    continue
c
      write(*,*) a
c
      stop
      end
```

was for instance automatically replaced by the code

```
C*****************************************************************
C   Translated by IBM AIX XL FORTRAN -Pv Preprocessor/6000
C   Version 02.02.0000.0000 3.05F16 on  9/30/94 at 19:05:48
C*****************************************************************
C
      program essl
      integer i,j,k
      double precision a(100,100),b(100,100),c(100,100)
c
      read(*,*) b,c
c
      CALL DGEMM ('N', 'N', 100, 100, 100, 1.D0, B, 100, C, 100, 0.D0, A
     1    , 100)
c
      write(*,*) a
c
      stop
      end
```

using the above program. The use of in-line expansion, or statement functions, and program substitution is not contradictory. Statement functions, for example, are by nature limited only to very small functions. Their use is merely to simplify coding and aid readability whilst avoiding unduly excessive and relatively expensive function calls. BLAS on the other hand are computationally and, especially in the case of higher level BLAS, data intensive. They are also very highly tuned for the job they do. Thus, for such tasks, the balance between additional calling costs and greater efficiency lies in the other direction. There is thus no contradiction between the two techniques and they may used side-by-side.

14.1.5 Type conversions and common subexpression enhancement

Care should be taken with mixed type statements. For example the code

```
      integer i,j
      real a
c
      j = a/i
```

implicitly makes two such conversions:

```
      j = int(a/float(i))
```

which are, potentially, expensive. It is thus good practice to try to group variables of the same type together. For example assuming

```
      integer i,j,k
      real a,b,c
```

i*b*j*c*k*d could be better written as **(i*j*k)*(a*b*c)**. Similar techniques should also be used with constants. Grouped together, it is easier for the compiler to identify known constant expressions at compile time. Grouping may also be used for identifying common expressions occurring more than once in the same block of code. Many compilers will also, for example, recognise the common expression **a(i)*b(i)** in the code

```
      f(i) = (a(i) * b(i)) * c(i)
      g(i) = e(i) * (a(i) * b(i))
```

and so avoid evaluating it twice in the final executable binary. Within loops, for example, this bracketing can be invoked automatically by, say, an appropriate preprocessor.

14.1.6 Character variables

The implementation of character variables can be either very efficient or very inefficient depending on the compiler used. Thus it could be argued that they are, at least for computation, best avoided. For program control, however (say specifying actions across parameter lists) they are very useful and result in generally much more readable codes. For such applications they are then thus recommended. BLAS (see the above pre-processor example), for example, uses them to specify if a matrix is upper or lower triangular etc. Note that by typing a character string parameter variable as

```
      character*(*)
```

in a subroutine, its length is defined as that of the variable passed to it from the calling program. This can then be extracted, as can the length of any character variable using the FORTRAN intrinsic function, LEN.

14.1.7 DATA and parameters

To initialise non-constant variables before execution, DATA statements can be used. These may be faster than normal assignments. Parameters (constants) should always be defined explicitly as parameters with a PARAMETER statement as this saves initialisation costs (it is 'done' at compile time) and variable look-up etc.

14.1.8 COMMON

To add to the debate on COMMON, program communication with COMMON can actually be faster than with parameters. This is because the addresses of parameters passed across parameter lists usually have to be accessed indirectly via a reference table whereas COMMON variables can be accessed directly in their relevant COMMON area. Such considerations, however, should clearly be weighed against the more general issues of program readability and style. COMMON definitions with differing array definitions in different subroutines are not recommended.

14.1.9 Unformatted input/output

Do you need to read by eye or edit program input/output? If not, you should consider using *unformatted* files rather than the standard *formatted* type. Statements of the form

```
read(channel) ...
write(channel) ...
```

(that is with no attached FORMAT statements) read and write data files directly in binary machine representation and are generally much more efficient. The resulting files are usually smaller and full machine precision is maintained. Furthermore, I/O operations are faster since no costly binary to alphanumeric conversions are required. Such unformatted files are generally not portable between different computer systems although conversion programs may be possible.

14.1.10 Input/output calls

A read or write statement typically requires at least three calls to the I/O system routines: one to initialise the transfer, another to effect it and a final to stop it. Since such I/O system calls are expensive, unnecessary calls are best avoided and the values to be output are best kept contiguous in store and transferred together *en bloc*. For example, whereas the statements

```
write(9) a
write(9) (a(i), i=1,100)
```

would each result in only three calls, the calls

```
write(9) (a(i), i=1,99,2)
write(9) ((b(i,j), j=1,100),i=1,100)
```

may require many more: two plus one for each element[3]. They are thus more expensive. Due to the high cost of such I/O calls, for large amounts of data it may be worth while preparing an output vector before writing, by copying, or even by an EQUIVALENCE statement. This is one of the few occasions where EQUIVALENCE results in more efficient code.

14.1.11 IF statements

Common sense is required with IF-statements. Something like

```
      if (i .eq. 1) ...
      if (i .eq. 2) ...
      if (i .eq. 3) ...
      ...
```

is obviously inefficient as i must be tested for 2, 3 etc. even if it is known to be 1. An IF-ELSEIF construct would be better. Better still, you could consider a computed goto (see section 13.6) or some form of nested IF-block binary search. A new construct in Fortran 90 for such cases is presented in section 20.2.

14.2 DO-loops

One of the greatest scopes for improving program efficiency lies in the re-ordering and re-structuring of DO-loops. Whereas this is normally just common sense, it often links up with the ideas on memory management introduced in chapter 8.

14.2.1 Invariant code

Clearly, invariant code should be removed from inside loops. The code

```
      factor = pi*scale(8)
      do 1 i=1,10,2
         v = v + factor*z(i)
 1    continue
```

is more efficient than

```
      do 1 i=1,10,2
         v = v + pi*scale(8)*z(i)
 1    continue
```

A related trick, where divisions are slower than multiplications, is to replace

```
      r1s = 1./s
      do 1 i=1,10
         v = v + x(i) * r1s
 1    continue
```

[3] Modern compilers are often more efficient than this.

for

```
      do 1 i=1,10
         v = v + x(i) / s
 1    continue
```

In this way only one divide is required.

14.2.2 Loop unrolling and nesting

To avoid overheads involved with implementing DO-loops it may be possible to partially expand — or *unroll* — DO-loops. For example

```
      subroutine unroll(n,a)
      integer n,i
      real a(1000)
c
      do 1 i=1,1000
         a(i) = 0.0
 1    continue
c
      return
      end
```

could be rewritten as

```
C*****************************************************************
C   Translated by IBM AIX XL FORTRAN -Pv Preprocessor/6000
C   Version 02.02.0000.0000 3.05F16 on  9/30/94 at 19:14:47
C*****************************************************************
C
      subroutine unroll(n,a)
      integer n,i
      real a(1000)
c
      DO 1 I = 1, 1000, 2
         A(I) = 0.0
         A(I+1) = 0.0
    1 CONTINUE
c
      return
      end
```

thus reducing the DO-loop's overall overhead. Because the major costs of implementing DO-loops arise during initialisation and termination, where nests of loops are coded it is often better to ensure that the highest number of iterations are carried out by the innermost loops. This reduces the total number of DO-loop invocations. Where arrays are involved, however, this should always be offset against any memory access implications related to the way in which the arrays are addressed. Suitable loop interchange can often be done automatically by an appropriate pre-processor tool.

14.2.3 Loop fusion

Two similar loops can often be more efficiently implemented as one single loop. For example

```
      do 1 i=1,n
         a(i) = b(i) + c(i)
1     continue
c
      do 2 i=1,n
         d(i) = b(i) - c(i)
2     continue
```

would be much better implemented as

```
      do 1 i=1,n
         a(i) = b(i) + c(i)
         d(i) = b(i) - c(i)
1     continue
```

which, after suitable manual or machine optimisation, can also reduce the number of accesses to b and c.

14.2.4 Avoiding type conversion

Type conversions are also expensive. Thus the loop

```
      subroutine types(n,b)
      integer n,i
      real b(1000)
c
      do 1 i=1,1000
         b(i) = i
1     continue
c
      return
      end
```

is less efficient than the longer version

```
C*****************************************************************
C   Translated by IBM AIX XL FORTRAN -Pv Preprocessor/6000
C   Version 02.02.0000.0000 3.05F16 on  9/30/94 at 19:18:53
C*****************************************************************
C
      subroutine types(n,b)
      integer n,i
      real b(1000)
      REAL R1X
c
```

```
      R1X = 1.
      DO 1 I = 1, 1000
         B(I) = R1X
         R1X = R1X + 1.
    1 CONTINUE
c
      return
      end
```

The merit of such a rewrite is, however, questionable. Again, this transformation was done automatically with the IBM tool.

14.2.5 Loop collapse

Finally, using the ideas connected with array addressing[4], a 2D or higher dimensional array can always be accessed as a 1D array, given suitable code. For example

```
      subroutine flat(r,s)
      integer i,j,k
      real r(100,100,100),s(100,100,100)
c
      do 1 k=1,100
         do 2 j=1,100
            do 3 i=1,100
               r(i,j,k) = s(i,j,k)
 3          continue
 2       continue
 1    continue
c
      return
      end
```

could be recoded as

```
C*****************************************************************
C   Translated by IBM AIX XL FORTRAN -Pv Preprocessor/6000
C   Version 02.02.0000.0000 3.05F16 on  9/30/94 at 19:20:06
C*****************************************************************
C
      subroutine flat(r,s)
      integer i,j,k
      real r(100,100,100),s(100,100,100)
c
      DO 1 K = 1, 1000000
         R(K,1,1) = S(K,1,1)
    1 CONTINUE
c
```

[4] See chapter 6.

```
      return
      end
```

which requires only one rather than 10001 DO-loop invocations. Note however that this optimisation causes array parameters to lie out of range and run time exceptions may then occur if an out-of-bound-array flag is enforced. Care should also be taken not to exceed the maximum representable machine integer within the DO-loop count.

14.3 Calling the compiler

Once a program is running correctly, and you are confident that there are no programming or logical errors, then you may often reduce its execution time by recompiling it with an appropriate compiler optimisation option. This typically results in faster running binaries but is less likely to trap program errors. It can sometimes even rearrange the order in which expressions are evaluated and so the numerical results at a machine precision level may be slightly perturbed. This can affect debugging, confusing, for example, some debugging environments. Since details vary from system to system, you should consult your appropriate system documentation. To achieve best results, you may also need to deselect other compiler options such as array subscript out-of-bound flags etc.

15

Input/Output

For production and development codes alike, data input/output is potentially difficult. If many input parameters are to be input, how should they be handled? How should they be ordered and identified? If files are used, how should they be organised? In short, how can you make a simple and friendly user interface? Here we suggest two possible schemes: one generic scheme which will run on all systems and a second motivated by UNIX-like pipelines for UNIX and other operating systems. Many alternative techniques are of course possible and this chapter is presented for illustration purposes only.

15.1 A generic system

Consider the following program to add two numbers:

```
      program add
c
      real a,b
c
      open(1,file='add_file')
c
      call search('A')
      read(1,*) a
c
      call search('B')
      read(1,*) b
c
      call search('END')
      write(1,*) a+b
c
      stop
      end
```

to be run with the data file

```
A
16

B
5

END
21.00000000
```

Here **search** (see below) scans the file attached to unit 1 for the specified keyword, and then, if found, returns control to the main program. After completion, however, the file pointer is left on the next line following the located string. This line can then be read directly by a standard READ statement. The string 'A' thus acts as a keyword to identify the value associated with the variable **a** and 'B' does the same for **b**. If either keyword is not found, an error occurs and **search** prints an appropriate error message before stopping the program. Finally, after the two data values have been input, the final call to **search('END')** scans the file for the keyword 'END' and positions the file pointer to the following line ready to output the result. Thus each time the program is run, the new sum is written at the end of the input file without overwriting the input parameters. A variation on this theme is the logical function **exist** listed below. A call

```
exist(channl,key)
```

searches the file attached to unit number **channl** for the keyword **key**. If this is found, **exist** is returned as **.true.**, if not as **.false.**. This can be very useful to trigger flags as illustrated by the following example program:

```
      program hello
      logical exist
      external exist
c
      open(1,file='greetings')
c
      if (exist(1,'How do you do?')) then
         write(*,*) 'Very well thank you.'
      else
         write(*,*) 'Goodbye'
      endif
c
      stop
      end
```

This replies with the message *'Very well thank you'* if it finds the question *'How do you do?'* in file **greetings** but otherwise responds with a curt *'Goodbye'*. **Exist** is in fact implemented as follows:

```
      logical function exist(channl,key)
c
      integer channl
```

```
      character key*(*)
      character line*80
      intrinsic len
c
      rewind channl
c
 1    read (channl,fmt=1000,end=2) line
c
      if ((line(1:len(key)).eq.key) .or.
     +          (line(2:len(key)+1).eq.key)) then
         exist = .true.
         return
      else
         go to 1
      endif
c
 2    exist = .false.
c
      return
c
 1000 format (a80)
      end
```

Note how this routine allows the keyword to start in the first or second column to allow reading from files written by other programs with list directed write statements (i.e. with FMT=*, e.g. **write**(*,*) 'A'). Using this function, **search** could then be coded as

```
      subroutine search(key)
c
      character key*(*)
      logical exist
      external exist
c
      if (.not.exist(1,key)) then
         write (*,*) ' ** keyword ', key, ' not found by search.'
         write (*,*) ' -- program stopped.--'
         stop
      endif
c
      return
      end
```

as can a generalised variant **xsrch** which we also list:

```
      subroutine xsrch(channl,key)
c
      integer channl
      character key*(*)
```

```
      logical exist
      external exist
c
      if (.not.exist(channl,key)) then
         write (*,*) ' ** keyword ', key, ' not found by xsrch',
     +               ' on channel', channl
         write (*,*) ' -- program stopped. --'
         stop
      endif
c
      return
      end
```

to be used with any input channel. These three routines, together possibly with **pull** from section 5.2, provide a very simple and effective way of managing data input and output. They can be used either to construct simple computer interfaces for full size programs or to handle temporary test parameters in test programs. Indeed under some operating systems they may be used to simulate a simple full screen interface. Given a suitable editor macro to save a control file and call and run the corresponding program, a full screen front end could be developed. Further possible refinements include refreshing the editor screen after execution to reflect any output written to the control file. Such once-and-for-all macros, together with the above FORTRAN routines, can provide a simple and effective interface for many applications.

15.2 Unix systems – pipelines

A very useful and well known feature in UNIX, as other operating systems, is the pipeline. However, pipelines are not confined merely to systems commands. They can be written using simple FORTRAN codes. Consider for example the following program:

```
      program abs_value
      real x,y
c
 1    continue
      read(unit=*,end=2,fmt=*) x,y
      write(*,*) x, abs(y)
      goto 1
c
 2    continue
      stop "(abs_value, ended OK)"
c
      end
```

This takes a stream of values $\{(x, y)\}$ as standard input and returns, as a result, a stream $\{(x, |y|)\}$ as standard output. If it is compiled as a binary **abs_value**, typing

```
cat | abs_value
1 2
2 -3
2 -6
3 5
ctrl-D
```

returns to the screen the following output:

```
1.000000000   2.000000000
2.000000000   3.000000000
2.000000000   6.000000000
3.000000000   5.000000000
```

Such techniques can also be used for graphics. If for example **image** is a program which produces a 2D plot from a standard input (x, y)-data stream and **sines** is the compiled binary of

```
      program sines
      double precision x
c
      do 1 x=0,10,.01
         write(*,*) x, sin(x)
 1    continue
c
      stop
      end
```

then

```
sines | image
```

produces a simple sine graph as we here illustrate in figure 1. Alternatively, a graph of $|\sin x|$ could be plotted:

```
sines | abs_value | image
```

We show this in figure 2. In UNIX, data streams from pipelines can be tapped using the UNIX *tee* command. For example

```
sines | tee A | abs_value | image
```

places a copy of the sine values in file **A**. The corresponding sine graph could then be plotted by either of the following commands:

```
cat A | image
image < A
```

In writing your own pipes, it is important to note the difference between standard output and standard error. Standard output may be piped to further commands whereas standard error always goes to the screen. For example, using again the IBM compiler XLF, a call to

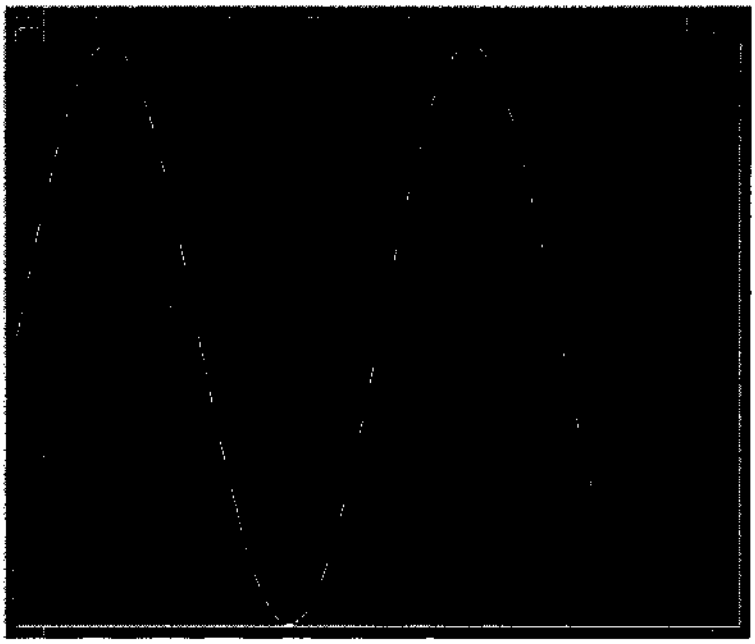


Figure 1. Graphical output for **sines | image.**

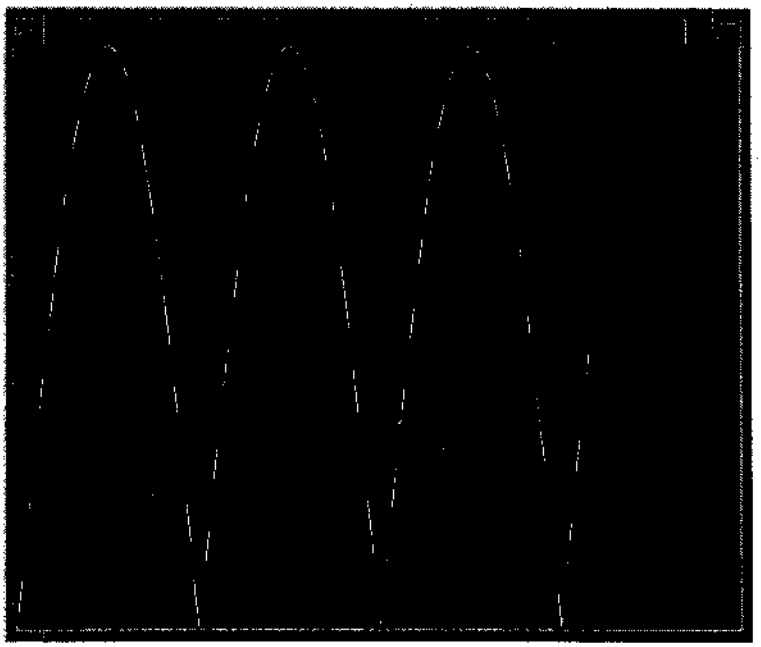


Figure 2. Graphical output for **sines | abs_value | image.**

```
      write(*,*) 'hello next pipeline'
```

writes to standard output whereas

```
      write(0,*) 'hello user'
```

or

```
      stop "(polar, ended OK)"
```

both write to standard error, that is, always to the screen[1]. Such output is not passed down the pipeline to subsequent processes. This can be useful to trace the completion of pipeline programs as they end. Note, however, that the FORTRAN standard allows only for integers to be echoed to standard output after the STOP command. Using this compiler, data can also be read directly from the screen (as distinct from standard input) using a statement of the form

```
      read(0,*) data_value
```

Such values are thus never taken from the preceding process in the pipeline. To illustrate this, consider the following two programs:

```
      program sinesn
      double precision x,n
c
      write(0,*) 'please enter n for ''sin n*x'' '
      read(*,*) n
c
      do 1 x=0,10,.01
         write(*,*) x, sin(n*x)
 1    continue
c
      stop "(sinesn, ended OK)"
      end
```

and

```
      program polar
      double precision theta,r,x,y
c
 1    continue
      read(unit=*,end=2,fmt=*) theta,r
c
         x = r * sin(theta)
         y = r * cos(theta)
         write(*,*) x,y
         goto 1
```

[1] Although widespread, this is not standard FORTRAN and may vary between systems. On some systems, for example, channel 5 may correspond to standard output and channel 6 to standard error.

```
c
 2    continue
      stop "(polar, ended OK)"
c
      end
```

which we compile as **sines_n** and **polar**. Used in a pipe

```
sines_n | polar | image
```

the user is requested to input a value for **n** and the results obtained are shown in figure 3. A variant on this theme is given by the following UNIX script **posey**:

```
echo  3 | sines_n | polar | image &
echo  9 | sines_n | polar | image &
echo 16 | sines_n | polar | image &
echo 20 | sines_n | polar | image &
```

which produces a set of plots (or a bunch of flowers) on the screen. We illustrate this in figure 4.

15.3 Exercises

The following examples illustrate and extend the techniques suggested in this chapter.

1. (*a file handling routine*) Using a keyword system, write a short routine to search a file attached to a named channel for up to 10 lines of comments, save these, and print them out on request.

2. (*data visualisation*) **ode** is a variable step ODE solver which writes the numerical solution of a one dimensional ordinary differential equation to standard output as a data stream $\{(x_i, y_i)\}$. Given the graphics tool **image** from above, write a suite of pipeline programs to create the following graphs:

   (a) the stepsize $h_i = t_i - t_{i-1}$ as a function of the time, t_i, and

   (b) the stepsize h_i as a function of the step number, i.

   Supply a further tool to represent both of these xy-graphs as a 'staircase graph', consisting solely of horizontal and vertical lines.

3. (*multiple data sets*) For many problems, we need to compare numerical solutions not just at a single point, but across an entire range. Using UNIX pipelines, suggest a technique to compare the solutions stored in two files, **A** and **B**, outputting the result to standard output.

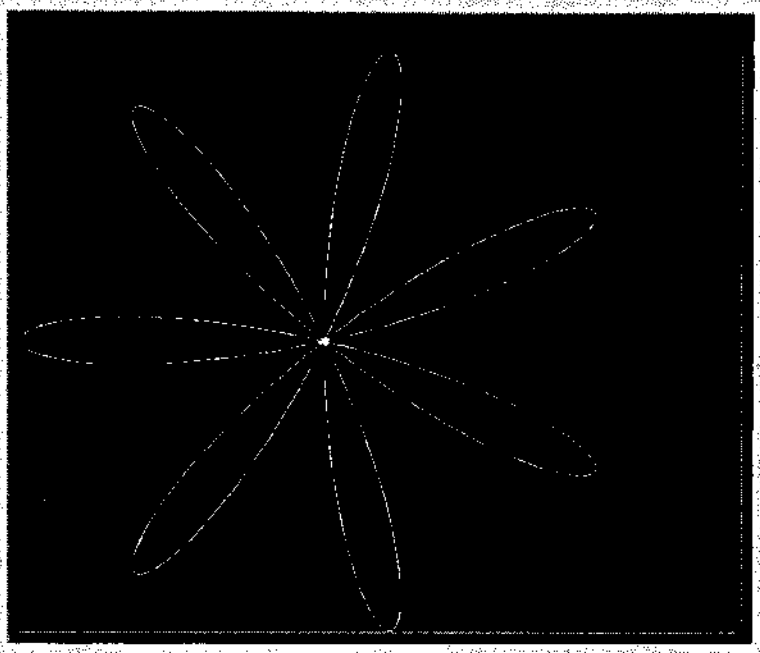


Figure 3. Graphical output for **sines_n | polar | image**.

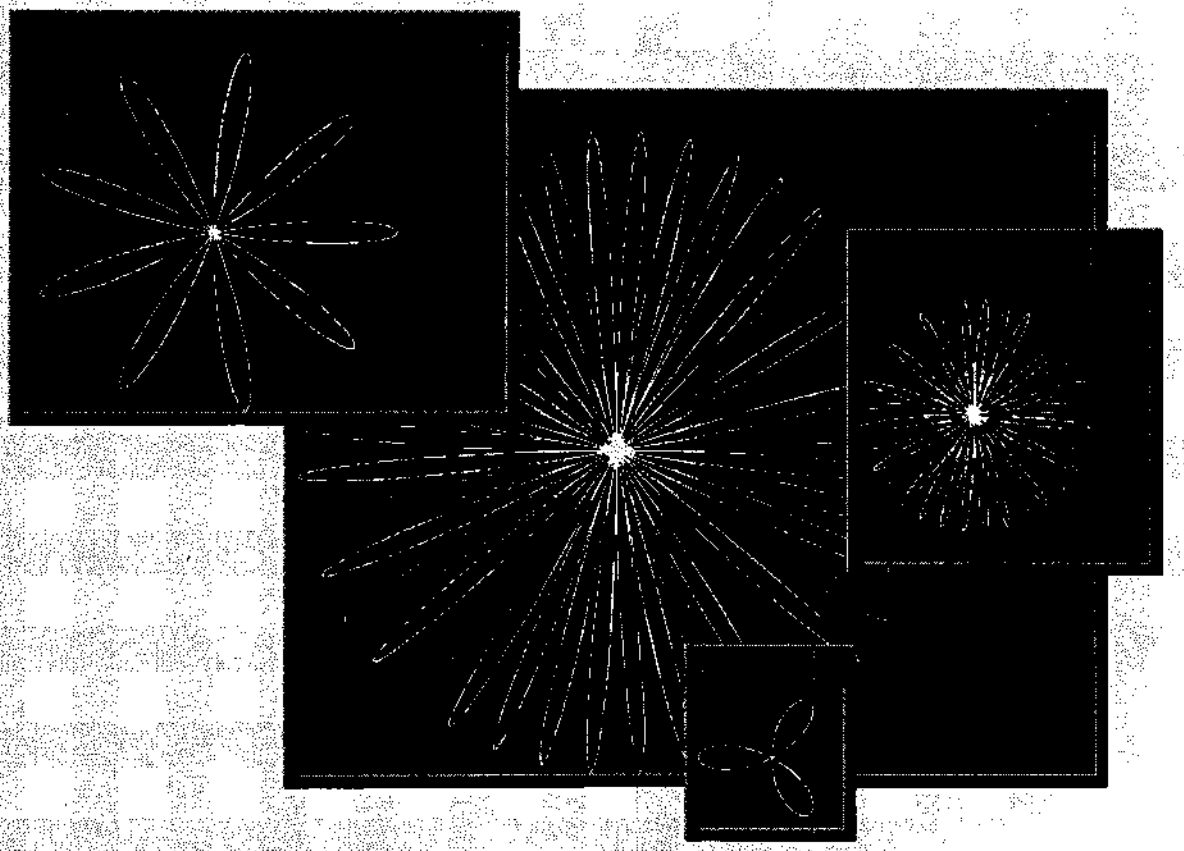


Figure 4. Graphical output generated by **posey**.

Part II

Fortran90 and High Performance Fortran

16

Fortran90 – Introduction

16.1 What is Fortran90?

FORTRAN is an old language and as such has changed greatly since it was first invented. Previous versions have names such as FORTRAN IV, FORTRAN 66 and FORTRAN 77. Fortran 90[1], however marks a significant advance over its predecessor FORTRAN 77, bringing into FORTRAN advances hereto only associated with more recent languages. Besides now common place features like pointers, derived types and recursion, new programming structures such as internal routines and modules are introduced, together with a wide range of new array operations and constructs for scientific applications. These features make for safer, more portable and more structured programming as well as making way for more efficient compiler optimisation.

Owing to the wide range of additions in Fortran 90, we here seek only to give a brief outline of the new standard. Fortran 90 may be considered as a new language in its own right. For a thorough and more detailed introduction, the reader is referred to [19].

16.2 Language continuity

Just as programs written in FORTRAN 66 will always run under FORTRAN 77[2], the FORTRAN 77 standard is supported as a subset of Fortran 90[3]. However, a number of so-called *obsolescent* features are due to be removed at the next revision. These include

- arithmetic IF,
- real and double precision DO-variables,
- the ASSIGN statement and
- alternate RETURNs.

Further, the following *deprecated* features may be removed one update later:

[1] The name *Fortran 90* is by convention written in lower case.

[2] Care should be taken with DO-loops as in FORTRAN 66 it was unclear whether they should always be executed at least once. Most FORTRAN 77 compilers have flags to accommodate this.

[3] This is the intention. Potential difficulties may for example arise with blanks which are now syntactically significant.

- BLOCK DATA,
- COMMON,
- EQUIVALENCE,
- computed GOTO,
- DOUBLE PRECISION,
- DIMENSION and
- the CHARACTER*len construct.

Some of these features, obsolescent and deprecated, are, as we will see, directly replaced by new Fortran 90 constructs.

16.3 Other dialects

Although the standard to Fortran 90 has now been agreed[4], its development took longer than expected. A number of proprietary Fortran 90-like dialects have thus been developed and proposed under different names alongside the new standard. Examples include HPF (High Performance Fortran, [34]) and CMF (Connection Machine Fortran, [33, 31]). High Performance and Connection Machine Fortran provide, in addition, various facilities for parallel architectures. CMF is otherwise a subset of Fortran 90 whereas HPF, still at proposal stage, is a superset. Following hopes that a new standard might have been achieved in the 1980's, references in the literature may still be found to the prototype standard, Fortran 8x [20]. High Performance Fortran, HPF, is described in further detail in chapter 22.

16.4 Free-form source codes

In addition to the previous static format in which statements are restricted to columns 7 to 72, the columns 1 to 6 being reserved for comments, labels and continuation characters, Fortran 90 also accepts an alternative free-form input format in which this restriction is removed. For example the code

```
.......     .
x = y +
$          a(1) + a(2) + a(3)
.......
```

can in Fortran 90 also be rewritten, starting from column one, as

```
.......
x = y + &
& a(1) + a(2) + a(3)
.......
```

or

[4] International standard ISO/IEC 1539 : 1991.

```
.........
x = y + &
a(1) + a(2) + a(3)
.........
```

or even

```
.........
x = y +
& a(1) + a(2) + a(3)
.........
```

the ampersand '&' acting as a continuation character. Semicolons also can be used to separate different commands on the same line:

```
d=1; e=5.0; i=5   ! the use of semicolons
```

and anything after a '!' is regarded as a comment. In free-form input, however, FORTRAN 77-style comment lines starting with 'c' or '*', as the columns 1 to 6, have no special meaning. The naming and declaration of variables also changes in Fortran 90. For example, optional attributes are allowed. Writing

```
integer, parameter :: system_dimension = 8
```

for instance, the *attribute* PARAMETER identifies the *integer* **system_dimension** as a parameter which is then assigned directly (at compile time) the value of 8. Here, as always, a double colon '::' is required to separate the attributes from the following variable names in the declaration list. Further examples include

```
real, dimension(3,25) :: mu,nu,pi
```

for **real mu(3,25), nu(3,25), pi(3,25)**. Full details of such type specifications are described below. Fortran 90 also permits long variable names – up to 31 characters with underscores. FORTRAN 77 type declaration statements are supported for continuity.

16.5 Implicit none

Since IMPLICIT NONE is now supported as part of the Fortran 90 standard, we use it selectively throughout the following examples. As pointed out previously, its effect is to require that all variables are explicitly typed. It should be placed after the program unit header, but before any other declaration or executable statement.

17

Subroutines and Modules

Besides the main program unit headed by the PROGRAM statement, there are only two basic types of program unit in FORTRAN 77: subroutines and functions. For large programs or complicated packages, this can be both restrictive and clumsy. Data, operations and functions frequently fall naturally into well defined higher level groups and computer languages should aim to reflect this. By introducing the concepts of the *module* and *internal routine*, Fortran 90 moves towards this goal. This will be the starting point of our discussion. Also discussed is the *interface block* which, giving a formal specification of a routine's external interface, makes way for array-valued routines, recursion, optional and key-word parameters and operator overloading – further features not supported in FORTRAN 77.

Since many of the features introduced in Fortran 90 were motivated by deficiencies in FORTRAN 77, where appropriate, a short foreword is given to set out the FORTRAN 77 problems which the Fortran 90 construct seeks to address.

17.1 Internal routines

In practice, functions and subroutines identified with one specific task need often only be called by a single program unit. In FORTRAN 77, however, all routines are automatically defined globally and thus may, in principle, be called from anywhere in the program. This is frequently both unnecessary and, where routines can be called outside their intended context, potentially dangerous. Fortran 90 addresses this issue by introducing the concept of private routines, private to, and effectively owned by, another program unit. These are known as *internal routines*.

Internal routines are normal subroutines or functions contained within, and belonging to, another program unit. They share[1] all types, variables and data values defined within the host routine but may not be called from outside. The subroutine

```
subroutine currencies

  implicit none
  real :: rate
  rate = 1.5
```

[1] Local variable definitions defined within an internal routine take precedence over, and can hence mask, same-name declarations in the host routine. Internal functions may be thought of as extending the concept of the *statement function* defined in FORTRAN 77.

```
module money

   real :: rate

contains

   subroutine set_rate(value)
      rate = value
   end subroutine set_rate

   function pounds_to_dollars(pounds)
      real :: pounds_to_dollars,pounds
      pounds_to_dollars = pounds*rate
   end function pounds_to_dollars

   function dollars_to_pounds(pounds)
      real :: dollars_to_pounds,pounds
      dollars_to_pounds = pounds/rate
   end function dollars_to_pounds

end module money


program transactions

   use money
   real :: value

   call set_rate(1.56)

   read(*,*) value
   write(*,*) dollars_to_pounds(value)

end program transactions
```

Figure 1. **Positioning of modules in Fortran 90.**

This example, taken from the main text, illustrates the placing of modules in relation to other program units in Fortran 90. Modules (which may also be compiled separately) are placed before the other program units which are otherwise arranged as in FORTRAN 77. FORTRAN 77-style external routines, if present, are placed after the main program unit (here program **transactions***) and any program unit can, as explained in the text, contain private 'internal routines' placed after a CONTAINS statement.*

```
module secret

   integer, parameter :: key=4311
   real, private :: x,y,z

contains

   subroutine read_position
      read(*,*) x,y,z
   end subroutine read_position

   subroutine write_position(key0)
      integer :: key0
      if (key0 .eq. key) then
         write(*,*) x,y,z
      endif
   end subroutine write_position

end module secret
```

the variables **x**, **y** and **z**, and the subroutines **read_position** and **write_position**, are, together with the parameter **key**, by default all accessible, *public*, from any outside routine merely by issuing the USE statement,

```
use secret
```

Declaring the variables **x**, **y**, **z** and **key** within **secret** as *private*,

```
integer, private, parameter :: key
real, private :: x,y,z
```

however, hides them from a potential user. Typing **use secret** then makes available only **read_-** and **write_position** and leaves the remaining variables readable only by means of these two routines. **x**, **y** and **z** are then said to be *private*. Equivalent results are also possible using PUBLIC and PRIVATE statements. Coding

```
private
public :: read_position, write_position
```

inside **secret** has the identical effect. Using a PUBLIC or PRIVATE statement alone with no arguments sets *all* variables to public or private. In Fortran 90, the contents of modules are, by default, public. An alternative technique uses the ONLY clause of the USE statement. Writing

```
use colours, only : red, green, blue
```

is equivalent then to

```
use colours
```

but makes available only the listed items **red**, **green** and **blue**. Any remaining contents of **colours** remain hidden. With the operator **=>**, renaming is also possible, the call

```
use colours, only : rosso  => red, &
                    azzuro => blue, &
                    verde  => green
```

making available the entities **red**, **blue** and **green** by their Italian translations only: **rosso**, **azzuro** and **verde**.

For program integrity, the concept of public and private parts has many advantages over the facilities provided in FORTRAN 77. By avoiding unnecessary access, unintentional interactions between program units can be avoided[2] as the programmer can take measures to ensure that routines, data and data types (see below) are not used out of context. They make Fortran a more exact language, providing a natural framework to package and present applications and library programs. This functionality is further extended by the addition of *interface blocks* which we discuss below.

17.2.2 Data modules

Finally, besides packaging programs, modules can also be used to group just data and data types. The data module

```
module pig_farm

   integer :: number_of_animals, farm_id
   real :: land_area, net_profit

   type animal
      integer :: id
      real :: mass
   end type animal

   type (animal), dimension(200) :: pigs

end module pig_farm
```

for example, contains just variables and derived data types (discussed below) but no routines. Used like this, the data module thus provides a more elegant and secure successor to COMMON, tagging data values with identifiers and supporting more advanced data types. In practice, modules usually contain a mixture of variables and routines: a set of data and a set of routines (methods) to operate upon it.

17.3 Interfaces

To improve upon the often clumsy and error prone methods for parameter passing in FORTRAN 77, Fortran 90 introduces a number of innovations in the way in which subroutines and functions are called. These include optional and keyword parameters,

[2] A common example would include name clashes between subroutines of the same name.

array and user defined generic functions and user defined operators. To support these new functions, a new construct is required to define the interface between different program units. This is the *interface block*. Interface blocks can also be used to aid compiler optimisation and check for program correctness.

17.3.1 Interface blocks and overloading

Program units may be considered as having an external appearance, or an interface, defined in terms of their type (in the case of functions) and the number and nature of their parameters. In Fortran 90 such information is needed to implement many of the new features listed above. Frequently, as in the case of internal routines or modules, this information is known automatically to the compiler without any special action. The interface is said to be *explicit*. Where it is not known it must be defined explicitly by means of an *interface block*. Given an external routine

```
subroutine clever_write_integer(i)
  integer :: i
  write(*,*) 'an integer: ',i
end subroutine clever_write_integer
```

we might write

```
interface clever_write_integer

  subroutine clever_write_integer(i)
    integer :: i
  end subroutine clever_write_integer

end interface
```

to indicate that it is a function with a single, integer, argument. Including, however, a second routine

```
subroutine clever_write_real(x)
  real :: x
  write(*,*) 'a real:    ',x
end subroutine clever_write_real
```

and a corresponding interface

```
interface clever_write

  subroutine clever_write_integer(i)
    integer :: i
  end subroutine clever_write_integer

  subroutine clever_write_real(x)
    real :: x
  end subroutine clever_write_real

end interface
```

more interesting results can be obtained. On encountering the statement

```
call clever_write(3.124)
```

the compiler simply scans the parameter list definitions given in the interface block for one matching the subroutine call. Once found, the corresponding routine is then called (here **clever_write_real**) to produce the required result:

```
a real:        3.1240000
```

In the language of FORTRAN 77, **clever_write_print** is a 'user defined *generic* function'. In Fortran 90 we refer to this possibility as *overloading*. A more advanced illustration of this feature is given in section 18.3.

In the case of routines defined as parts of modules, overloading is even simpler. Since the interface is then explicit, the body of interface block reduces to the MODULE PROCEDURE construction:

```
interface send
   module procedure send_integer, send_real, send_array
end interface
```

full details of each routine's interface being already known to the compiler. Finally, more advanced applications of interfaces can arise for example, for array valued functions where the size of the returned result (the function's value) may not be known until run-time, after compilation has been completed. An interface block may then be needed to make information about the routine's interface explicit. Examples of such cases are exercises 4 and 6 at the end of this chapter and exercise 1 at the end of the next.

17.3.2 Interfaces and INTENT

A frequent cause of run-time errors is the passing of incorrectly typed parameters across function or subroutine parameter lists. Although such practices are clearly illegal, the resulting errors often go undetected. Compiler checks across parameter lists are notoriously difficult since the required information is usually not available. Even if the required called routine has already been compiled, its binary may well be located in a separate file whose content and location is not known until after compile time when the program is loaded.

Different systems may offer different approaches to this problem. One option, for example, such as that used in the IBM FORTVS2 compiler, allows for the provision of a central *inter-compilation analysis* file to pool interface information about each routine's calling list as and when it is compiled. This can be then be accessed by later compilations to carry out inter-routine (i.e. inter-compilation) type checking. In Fortran 90, the rôle of such a file is played by the interface block. By placing a description of the called program unit in the calling routine, any required information can be referenced directly and checks made possible even when the two subprograms are stored in physically separate files. A generalisation of this idea, allowing the automatic checking of all routines, is to collect interface blocks into a single module and call this from each program unit as and when it is required. Not only can this

detect potential programming errors and aid documentation, but it may also improve program compilation making more information available at compilation time.

For tighter program security, Fortran 90 also provides an INTENT attribute used to specify whether selected parameters are to be used for data input, output or both. For example, in the code

```
subroutine add(a,b,sum)
   real, intent(in) :: a,b
   real, intent(out) :: sum
   sum = a + b
end subroutine add
```

used to evaluate the sum $a+b$, **a** and **b** are both input parameters whereas **sum** is used solely for output. Using INTENT above, for example, any attempt to redefine **a** or **b** internally within **add** would be flagged as incorrect. Joint input-output parameters may also be declared by INTENT(INOUT). INTENT is also useful for improving program readability.

17.3.3 Optional and keyword parameters

Fortran 90 also supports *optional* and *keyword parameters*. Given an interface[3]

```
interface tides
   function tides(day,month,year)
      real :: tides
      integer :: day,month,year
   end function tides
end interface
```

the function

```
function tides(day,month,year)
   integer :: day, month, year
   real :: tides
   ...
   ...
end function tides
```

may now be called by a statement of the form

```
write(*,*) tides(month=6,day=23,year=1945)
```

the order of parameters being disregarded. Where parameters are declared with the OPTIONAL attribute, they may be legally dropped from the calling parameter list. Thus including

```
integer, optional :: day,month,year
```

in **tides** and the corresponding interface block, the call

[3] Not required for module and internal routines. In such cases the interface is explicit.

```
write(*,*) tides(month=6,day=12)
```

is now allowed as is

```
write(*,*) tides(6,year=1994)
```

Here, normal parameter handling is performed until the first keyword is reached. Default actions may be handled using the logical intrinsic function PRESENT which tests for the presence of a parameter in the parameter list. Thus in

```
if (.not.present(year)) then
   this_year = 1984
endif
```

if **year** is not passed as a parameter a default value of '1984' is used.

17.4 Recursion

Fortran 90 introduces dynamic storage allocation. In particular it supports recursion. For example the routine

```
recursive function fibonacci(n) result(fibbi)
   integer :: n,fibbi

   if (n <= 2) then
      fibbi = n
   else
      fibbi = fibonacci(n-1)+fibonacci(n-2)
   endif

end function fibonacci
```

computes the Fibonacci sequence $\{f_i\}$ defined by

$$\begin{aligned} f(n) &= f(n-1) + f(n-2) \\ f(2) &= 2 \\ f(1) &= 1 \end{aligned}$$

for $n \geq 3$. Note the use of the heading RECURSIVE FUNCTION to identify the function as a recursive routine and the RESULT clause to identify the variable used to return the result. Here, the variable **fibonacci** could not be used as this name is already used by the function to call itself. Recursive algorithms are, however, often inefficient. To calculate $f(10)$ required 109 function calls and $f(20)$, 13529: the algorithm repeatedly recalculates the same values. Recursive subroutines are also supported.

17.5 Exercises

Examples of modules and interfaces here include *overloading* and *array valued functions*.

1. (*overloading*) Write a module containing a generic routine **subroutine xprint(x)** to print out a real scalar, vector or two dimensional array in a readable form depending on its type. How could you extend the routine to other data types? The intrinsic routines LBOUND and UBOUND may be used to give information about array bounds.

2. (*keyword arguments*) Using plain Fortran 90 only calls of the form

```
call machine(action='go')
call machine(action='go', speed=10.0)
call machine(action='reverse',speed=100.0)
...
```

   are supported. Using standard Fortran 90 structures, provide a module to allow calls of the form

```
call machine(action=go)
call machine(action=go, speed=slow)
call machine(action=reverse,speed=fast)
...
```

3. (*optional parameters*) Write a routine to return mathematical constants to appropriately named optional parameters. How might you provide a similar routine to provide conversions between different measure systems (e.g. *miles* to *kilometres*, *pounds* to *kilograms* etc.)?

4. (*SUM and array valued functions*) Write a logical function **ident** to return the rank two logical array

```
logical, dimension(n,m) :: ident
```

   with the property that $\mathbf{ident}_{i,j}$ is true if and only if $i = j$. The intrinsic function call **pack(array,mask)**, where **mask** has the same shape as array, returns (in column order) as a one dimensional vector all the elements of **array** for which the corresponding values of **mask** are true. Use this, and the above logical function, to compute the diagonal sums and products for a rank two array '**real, dimension(n,m) :: A**' for arbitrary **n** and **m**. SUM and the use of mask arrays are described in more detail in chapters 19 and 20.

5. (*recursion*) Write a recursive function to calculate the factorial $n!$.

6. (*recursion – a harder example*) Write a recursive algorithm to reverse the order of the elements in an arbitrary length character string.

7. (*Ackermann's function*) Repeat exercise 5 of chapter 4 by writing a recursive function for Ackermann's function [2]:

$$A(m,n) = \begin{cases} n+1 & : \quad m = 0 \\ A(m-1,1) & : \quad m \neq 0, n = 0 \\ A(m-1, A(m,n-1)) & : \quad m \neq 0, n \neq 0. \end{cases}$$

18 Types and Pointers

FORTRAN 77 supports only six basic types, LOGICAL, INTEGER, REAL, DOUBLE PRECISION, COMPLEX and CHARACTER. The difficulties involving portability and variable precision have already been discussed in section 2.5, whereas the absence of allocatable array and pointers, and the limitations of the six basic types, has long been a traditional criticism of Fortran. Just as internal routines, modules and interfaces provide alternative ways of packaging and presenting functions and subroutines, new Fortran 90 types and type declarations provide new methods of storing and representing data. We start our discussion with variable precision and the representation of floating point numbers.

18.1 Intrinsic types

18.1.1 Real numbers

As an alternative to FORTRAN 77 REAL and DOUBLE PRECISION statements, Fortran 90 allows variable precision to be defined directly by means of an integer attribute, KIND,

```
real(kind=1) :: normal,x,y
real(kind=2) :: fine,epsilion,small
```

which numerically selects the amount of precision to be allocated by a declaration statement. Although the appropriate values of KIND for any given precision are implementation dependent, higher values generally imply greater precision. Indeed, under some compilers, KIND may correspond to the required word-length needed to implement the type (cf. REAL*4 and REAL*8 in some FORTRAN 77 dialects). On all systems, suitable values are provided automatically, however, by the integer function SELECTED_REAL_KIND. Writing, say,

```
real(kind=selected_real_kind(P=15,R=20)) :: exact
```

or alternatively

```
integer, parameter :: long=selected_real_kind(P=15,R=20)
real(kind=long) :: exact
```

selects a KIND value sufficient to ensure a precision of 15 decimal places and an exponent range of 10^{-20} to 10^{+20} for the variable **exact**. It should however be

stressed that Fortran 90 does not support fully 'variable precision' as the range of supported P and R values is limited. If the requested value for P or R is too high, SELECT_REAL_KIND returns a value of -1, and the following type statement fails. Fortran 90 introduces no new floating point types, but merely changes the way in which existing types are selected. The new notation is however more portable, providing a more flexible platform for safe software migration between different machines. The programmer need merely specify the minimum accuracy requirements for their application and the compiler will attempt to allocate a type with sufficient precision to fulfil them. If it cannot, an error message is produced and the compilation stops.

For further information, Fortran 90 includes a range of enquiry routines to extract machine parameters relating to variable precisions and ranges. Based on the Brown model for computer arithmetic [4], they include, for example,

- **epsilon(x)** machine roundoff,
- **digits(x)** the number of significant digits and
- **tiny(x)** the smallest representable positive number,

returning values for a variable of the type **x**. Such information is often invaluable in the design of robust portable software. FORTRAN 77 programmers will note that DOUBLE PRECISION is still supported but is rendered redundant by the addition of the KIND attribute. Complex numbers are also supported in Fortran 90, their treatment being similar to that of real numbers. For further details, see [19].

18.1.2 Integers, characters and logical variables

KIND attributes are also provided for non-floating types. The integer intrinsic SELECTED_INTEGER_KIND function

```
integer (kind=selected_integer_kind(6)) :: count
```

for example, returns the required KIND-value for a 6-digit integer (i.e. one with range $-999{,}999$ to $999{,}999$) and declares a corresponding 6-digit precision variable, **count**. For characters and logical variables, KIND also supports various non-standard extensions such as extended foreign language character sets and special compressed bitwise storage techniques.

18.1.3 Constants

As in FORTRAN 77 (cf. 1.9E+06 and 34.09D+00), different precisions are also supported for constants. In Fortran 90, these are indicated using an underscore following the constant's values,

```
1.345_1
7.0_2
```

denoting KIND=1 and KIND=2 precision reals respectively. Similarly, using integer parameters

```
integer, parameter :: long=selected_real_kind(P=15,R=20)
```

```
...
...
x = 13.74_long
y = 43.0_long
```

similar notations are possible: the constant **long** following the underscore denotes the selected KIND value. Similar constructions are provided for integer, character and logical constants.

18.1.4 Short notation

Compared with REAL and DOUBLE PRECISION in FORTRAN77, the new statements of the form **real(kind=1)**, **real(kind=2)** may seem unwieldy and cumbersome. The new standard, however, also allows this form to be relaxed and the KIND keyword to be dropped,

```
real(1) :: matrix(100,100)
real(2) :: x,y,z
```

selecting, say, to the KIND values 1 and 2. Combined with appropriate parameter statements

```
integer, parameter :: long = selected_real_kind(P=14)
real(long) alpha(12), phi(n,16)
```

it thus provides a short, compact, readable, and above all portable method for defining machine precision. Similar techniques may be used for other types.

18.2 Derived types

In Fortran90, REAL, COMPLEX, INTEGER, CHARACTER and LOGICAL are known as *intrinsic* types. A major extension to the language is however the introduction of compound user-defined or *derived* types. Writing

```
type person
   character (len=10) :: name
    integer :: role_number
end type person
```

followed by

```
type(person) visitor
```

defines '**visitor**' as a variable of user defined type '**person**'. Variables of type person then have fields (**name** and **role_number**), each of which can be accessed by a % and the appropriate name:

```
visitor%name = 'George'
visitor%role_number = 273
```

Nested constructions are also permitted. Given

```
type point
   real x,y
end type point
```

for example, we can define

```
type triangle
   type (point) a,b,c
end type triangle

type (triangle) element
```

with which the complete co-ordinates of the triangle **element** could be output:

```
write(*,*) element%a%x, element%a%y
write(*,*) element%b%x, element%b%y
write(*,*) element%c%x, element%c%y
```

Further, to assign such types, the following shorthand is permitted:

```
a = point(1.,2.)
element = triangle(point(1.,2.),point(2.,2.),point(2.,1.))
```

having the effect

```
a%x = 1.; a%y = 2.
```

and

```
element%a%x = 1.; element%a%y = 2.
element%b%x = 2.; element%b%y = 2.
element%c%x = 2.; element%c%y = 1.
```

respectively.

18.3 Operator overloading

Derived types are particularly useful in passing information between program units. In particular, they are frequently used in conjunction with interface blocks. An interface

```
interface operator(.dist.)
   function distance(a,b)
   real :: distance
   type(point) a,b
   end function distance
end interface
```

for example, together with a type

```
type point
   real x,y
end type point
```

defined in a module **geometry**, and a function

```
real function distance(a,b)

   use geometry
   type(point) a,b

   distance = sqrt((a%x-b%x)**2 + (a%y-b%y)**2)

end function distance
```

define the function **distance** as an 'infix' operator[1] **.dist.**, the expression **x .dist. y** now being a synonym for **distance(x,y)**. For derived data types, even the elementary operations $+$, $-$, $*$, $/$ and $=$ can be redefined in this way. We refer to this as *operator overloading*. Defining

```
interface operator(-)
```

the call **x .dist. y** can be replaced by **x - y**. Applications of such type sensitive functionality are clearly wide. Using suitable interface blocks, the standard arithmetical operators can be extended at will to non-intrinsic types. Their meaning applied to numerical arrays is discussed in section 19.2.

18.4 Pointers

Besides arrays and derived types, the new standard also allows for more complicated user-defined structures through the use of pointers. A simple example illustrating the construction of a linked list is given at the end of this chapter in section 18.4.2.

18.4.1 Targets and ALLOCATE

In Fortran, as in most languages, variables are implemented in two parts: a first, or 'pointer', containing the *address* or *starting address* of a second part, a segment of *storage*, or 'target', used to store the corresponding value. In Fortran 90 this relation is placed under direct optional user control by two new attributes, POINTER and TARGET. In place of the statement

```
real :: x
```

we may define, for example, two parts – a *pointer* and a *target* – explicitly:

```
real, pointer :: x
real, target :: u
```

[1] A binary operator written between two arguments.

The contents of these two variables is initially undefined. However, writing

```
x => u
```

assigns the address of **u** to **x**: that is **x** is made to point to the storage location labelled as **u**. The pointer **x** can then be used like any other normal variable, labelling directly the contents of the target **u**. Pointers are frequently, however, more flexible than normal variables. Defining, for instance

```
real, target :: v,w
```

the pointer's address can be switched from one target to another:

```
x => v
x => w
```

without affecting their contents. That is, these two assignments have a similar effect to

```
x=v
x=w
```

except that only the *storage addresses*[2] of **v** and **w** are copied and *not* the contents of the corresponding storage locations. In more complicated applications, a target may be referred to by more than one pointer.

Storage (targets) can also be allocated to pointers directly by an ALLOCATE statement. For example, given

```
character(len=10), pointer :: name
integer, pointer, dimension(:,:) :: table
```

the statements

```
allocate(name)
allocate(table(100,500))
```

allocate (or create) targets of the types

```
character(len=10), target
integer, target, dimension(100,500)
```

for the pointers **name** and **table**. The meaning of the **dimension(100,500)** and **dimension(:,:)** is explained when we discuss arrays in section 19.1. Significant here is only that, whereas the size of the desired object need not be defined, its type, and in the case of an array, its rank, its number of dimensions, must. The effect of the above operations is thus similar to

```
character(len=10), target :: name
integer, target, dimension(100,500) :: table
```

or in FORTRAN 77

[2] This parallels the passing of arrays across subroutine parameter lists discussed in section 6.2.

```
character*10 name
integer table(100,500)
```

except that **name** and **table** may be optionally deallocated by means of a DEALLOCATE statement, and the storage hence recovered by the operating system, or that a pointer may be reassigned to a new target: **name => impostor**.

18.4.2 A linked list

Pointers are important because they allow for flexible, arbitrary data structures which can be created, destroyed or modified dynamically at run time. To illustrate this, consider the construction of a simple linked list. Defining first the type **pass_the_buck** which will be used to store each node

```
type pass_the_buck
   character(len=10) :: name
   type(pass_the_buck),pointer :: next
end type pass_the_buck
```

the statements

```
type(pass_the_buck) me
me%name = 'David'
```

declare (i.e. allocate) the first note, **me**, and assign to it a name 'David'. Initially the value of the field **me%next** is undefined. However, writing

```
allocate(me%next)
```

allocates a target, of type **pass_the_buck**, to this entry thus creating a new node **me%next**. This new node has a field **name** and **field** as before and can be assigned and/or allocated just like **me** above:

```
me%next%name = 'My Boss'
```

In particular, allocating

```
allocate(me%next%next)
```

allocates, and thus creates, a new third node ready for further assignment. Continuing this process indefinitely,

```
me%next%next%name = 'Their Boss'
allocate(me%next%next%next)
...
```

lists of arbitrary length can be created, as required, at run time. We refer to such a structure as a *linked list*. We illustrate this in figure 1. Such structures are often handled automatically. The following routine, for example, prints out the above list:

```
subroutine follow_list(start)
   type(pass_the_buck) start,node
```

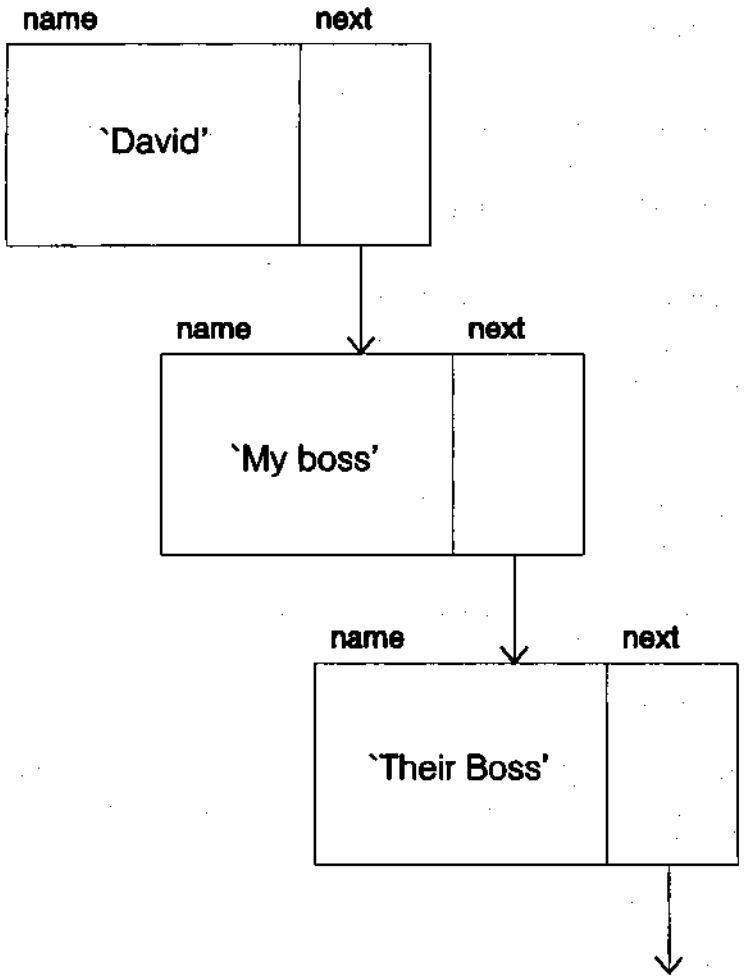


Figure 1. **Pointer structures: a linked list.**

Pointers are the most flexible of all Fortran 90 data structures. As in many other languages they allow storage objects to grow or change size whilst the program is running.

```
    node = start
 1  continue
    write(*,*) node%name
    if (associated(node%next)) then
       node = node%next
       goto 1
    else
       write(*,*) '*end*'
    endif

end subroutine follow_list
```

Here, ASSOCIATED is an intrinsic logical enquiry function which returns a value of **.true.** if and only if the named pointer has a target associated to it. A further discussion of this example is given in the solution to exercise 3 in appendix C. Despite their great flexibility, pointers are often difficult to program with efficiently. Particular attention must be paid to recovering storage after it has been used. In badly written codes, much time can be lost by the operating system in trying to recover storage from targets to which no pointer any longer points.

18.5 Exercises

Fortran 90's improved types and data structures complement its advances for subroutines and functions. Examples here include user-written support for rational and vector arithmetic.

1. (*modules and interfaces*) Write a module **vector_products** to provide two operators, **.inner.** and **.outer.**, to compute the *inner* and *outer* products

$$xy = \sum_{i=1}^{N} x_i y_i \quad \text{and} \quad xy^T = \begin{pmatrix} x_1y_1 & x_1y_2 & \cdots & x_1y_N \\ x_2y_1 & x_2y_2 & \cdots & x_2y_N \\ \vdots & \vdots & & \vdots \\ x_Ny_1 & x_Ny_2 & \cdots & x_Ny_N \end{pmatrix}$$

   for two general one-dimensional arrays **x** and **y** of equal length.

2. (*derived types and user defined operators*) Write a module to define a user defined type **fraction** suitable to represent vulgar fractions of the form a/b where a and b are integers. Provide also operators $+$, $-$, $*$ and $/$ to implement the four standard arithmetic operators on this type. By means of an additional operator **.r.**, defined such that

$$\text{a .r. b} = \text{fraction(a,b)}$$

   extend the module to support expressions of the form

$$(1.\text{r}.6) * (2.\text{r}.7) - (1.\text{r}.2)$$

Are brackets strictly necessary in this context? Explain how the above module could be extended to allow mixed expressions of the form '1+(1.r.2)'?

3. (*pointers and linked lists*) For simplicity, **follow_list** in section 18.4.2 was written using standard derived types for **start** and **node**. Rewrite this code declaring these two variables with attributes TARGET and POINTER respectively. Provide also a further routine, **add_to_list**, to *append* further entries to an existing list. To do this you should use the NULLIFY statement, **nullify(pointer_name)**, which sets a pointer's, here **pointer_name**'s, association status to 'unassociated'. By default, the association status is initially undefined.

4. (*a block algorithm*) Repeat exercise 3 of chapter 6 using a Fortran 90 routine **subroutine mult2(A,B,C)** which computes $C \leftarrow AB + C$ where A, B, C are all 2×2 matrices. How does the Fortran 90 version compare? How could this algorithm be further improved in Fortran 90?

19

Arrays

Owing to their central importance in scientific computing, Fortran 90 provides a wide range of new techniques and facilities for working with arrays. They alleviate many of the well known difficulties in FORTRAN 77 and make way for simpler, syntactically more compact codes. Further, by taking more low level operations away from the programmer into the language's syntax, better and more advanced compiler optimisation can also be performed.

19.1 Dynamic storage allocation

Defined in FORTRAN 77 as

```
real a(30,40), b(30,40), c(4:50,7)
```

or by means of the DIMENSION attribute

```
real, dimension(30,40) :: a,b
real, dimension(4:50,7) :: c
```

in Fortran 90, storage for arrays may now be allocated dynamically at run time. The subroutine

```
subroutine tasks(n,m)

   integer :: n,m
   real, dimension(n,m) :: x,y
   ....

end subroutine tasks
```

for example, is now legal, allocating on entry the two arrays **x** and **y** with dimensions $n \times m$ defined by variables **n** and **m**[1]. This, known in Fortran 90 as an *automatic array*, thus removes any need to pass extra workspace in the form of work arrays – one of the major drawbacks in FORTRAN 77, see chapter 7. Still greater flexibility in storage allocation is supported by the ALLOCATE statement. Specifying an array as *allocatable*:

[1] That is `real x(n,m), y(n,m)` in FORTRAN 77. In FORTRAN 77 this would only be legal if **n** and **m** were parameters or constants.

```
integer, dimension(:,:), allocatable :: tasks
```

when it is declared storage allocation can be deferred at will to an ALLOCATE statement

```
allocate(tasks(100*n+j,200))
```

which defines its bounds once suitable values are known. That is, decisions on array dimensions can be postponed until the program is running. For example, the following program:

```
program election

   integer :: number_of_candidates, vote
   integer, allocatable, dimension(:) :: ballot

   read(*,*) number_of_candidates
   allocate(ballot(number_of_candidates))

   ballot = 0

1  continue
   read(*,*,end=2) vote
   ballot(vote) = ballot(vote) + 1
   goto 1

2  continue
   write(*,*) ballot

end program election
```

postpones a decision on the size of the array **ballot** until after the value **no_of_candidates** has been read from standard input. Note how in both examples — **ballot** and **tasks** above — the rank (number of dimensions) is still required in the declaration statement. This is done by means of the ':' notation, specifying a dimension with a dummy (undefined) range. The use of **ballot = 0**, which sets the entire array to zero, is clarified in section 19.2 below.

As for pointers, array storage allocated by an ALLOCATE statement may be released by means of a DEALLOCATE statement once it is no longer required. Thus, the two code segments

```
integer, dimension(5,4,7:9) :: results
```

and

```
integer, dimension(:,:,:) :: results
allocate results(5,4,7:9)
```

differ only in that the second storage allocation is *temporary* and may be reversed by a DEALLOCATE statement:

```
deallocate(results)
```

Once an array has been deallocated, it may, of course, be reallocated, possibly with different bounds, by a further ALLOCATE statement.

19.2 Elemental operations and array constructors

19.2.1 Arithmetic operators

Arrays are more fully integrated into Fortran 90 than in FORTRAN 77. In particular, they are supported by the operators =, +, −, * and /. Given two equally dimensioned arrays

```
logical, dimension(10) :: lies,truth
```

for example, the assignment

```
lies = truth
```

copies the entire array, element-by-element: **lies(i)** = **truth(i)** for $i = 1$ to 10. For numeric types, such *elemental* arithmetic operations are defined similarly,

```
A = B + C
```

returning **A** as the matrix sum of **B** and **C**. Note however that the element-wise multiplication and division performed by Fortran 90 differ from the normal mathematical convention,

```
E = D * F
```

returning the 'pointwise product'

$$E(i,j) = D(i,j)F(i,j)$$

rather than the

$$E(i,j) = \sum_k D(i,k)F(k,j)$$

standard for linear algebra. Where scalars are included in matrix expressions, they are expanded automatically to match the dimensions of the corresponding matrices. For example

```
X = Z + 1
```

adds one to every element of **Z** and writes the result to **X** whilst

```
COSTS = COSTS/2.0
```

divides each element of **COSTS** by 2.0. Used alone, a scalar can thus assign a whole matrix in a single command,

```
ballot = 0
```

as in program **election** above.

19.2.2 Intrinsic functions and logical relations

Along with the operators =, +, −, * and /, many intrinsic functions also support element-wise actions. In this case they too are said to be *elemental*. The statements

```
F = sin(D)*cos(E) + 7.0*exp(E)
H = G*log(G)
```

for instance, using the normally scalar functions SIN, COS, EXP and LOG, implement the component-wise operations[2]

$$\begin{aligned} f_{ij} &= \sin d_{ij} \cos e_{ij} + 7 \exp f_{ij} \\ h_i &= g_i \ln g_i \end{aligned}$$

for suitably dimensioned arrays **D** to **H**. More significantly, elemental comparisons are also supported:

FORTRAN 77 notation	*Fortran 90 synonym*
.LT.	<
.LE.	<=
.GT.	>
.GE.	>=
.EQ.	==
.NE.	/ =

The synonyms <, <= and / = etc. are now standard for Fortran 90. Defining for example

$$A = [5, 7, 8, 3, 1]$$

the comparison

```
A < 6
```

yields a logical array of length five

```
T F F T T
```

This could then be used with an intrinsic function such as COUNT (which counts the number of true elements in a logical array) or ALL (true if and only if all the elements are true). Thus, the calls

```
count(B==5)
all(B<10)
```

return the number of elements in **B** equal to 5 and whether or not the elements of **B** are all less than 10.

[2] These again differ from normal mathematical conventions. For A square we might expect $\exp(A) = I + A + (1/2!)A^2 + (1/3!)A^3 + \cdots$.

19.2.3 Array constructors

For one dimensional arrays a powerful notation for array constants is introduced. In

```
A = (/ 5.0, 7.0, 4.0, 1.0, 7.0 /)
```

the *array constructor* **(/ 5.0, 7.0, 4.0, 1.0, 7.0 /)** acts as an array constant, the above statement having the effect of assigning

```
A(1) = 5.0;  A(2) = 7.0;  A(3) = 4.0
A(4) = 1.0;  A(5) = 7.0
```

For larger arrays implied DO-loops may be used,

```
(/ (i, i=1,100) /)
```

returning, for example, an integer array containing the numbers 1 to 100 in ascending order. Repeaters (cf. format descriptors in FORMAT statements) are also supported,

```
(/ 66, 3(1,2), 77 /)
```

being a shorthand notation for

```
(/ 66, 1, 2, 1, 2, 1, 2, 77 /)
```

Combined with such intrinsic functions as SUM (which returns the sum of all the elements of an array) or PRODUCT (computes the product), array constructors can result in highly compact readable code:

```
read(*,*) n
write(*,*) sum( (/ (j,j=1,n) /) )
write(*,*) product( (/ (j,j=1,n) /) )
```

printing for example the sum and the product, **n!**, of the first **n** integers. Note how the two array constructors are here dynamically or statically dimensioned, depending on whether the corresponding DO-loop bounds are variables or constants. Real implied DO-loops are however not allowed[3]. In

```
program spiral

   integer i
   real, dimension(101) :: theta, r, x, y

   theta = (/ (0.1*i,i=0,100) /)

   r = exp(-theta/10.0)
   x = r*cos(theta)
   y = r*sin(theta)
```

[3] This is in keeping with the currently accepted idea that loops indexed by floating point numbers should be discouraged. See the solution (in appendix C) of exercise 2 of chapter 1 for details. In Fortran 90, normal floating point DO-loops have also been declared obsolete and may be removed at a further revision.

```
      do i=1,101
         write(*,*) x(i),y(i)
      end do

   end program spiral
```

for example, we must write

```
   theta = (/ (0.1*i,i=0,100) /)
```

instead of the superficially simpler

```
   theta = (/ (x,x=0.0,10.0,0.1) /)
```

for `x` real. This program plots the spiral

$$r = e^{-\theta/10}$$

in Cartesian co-ordinates $(x, y) = (r\cos\theta, r\sin\theta)$. The results are displayed in figure 1.

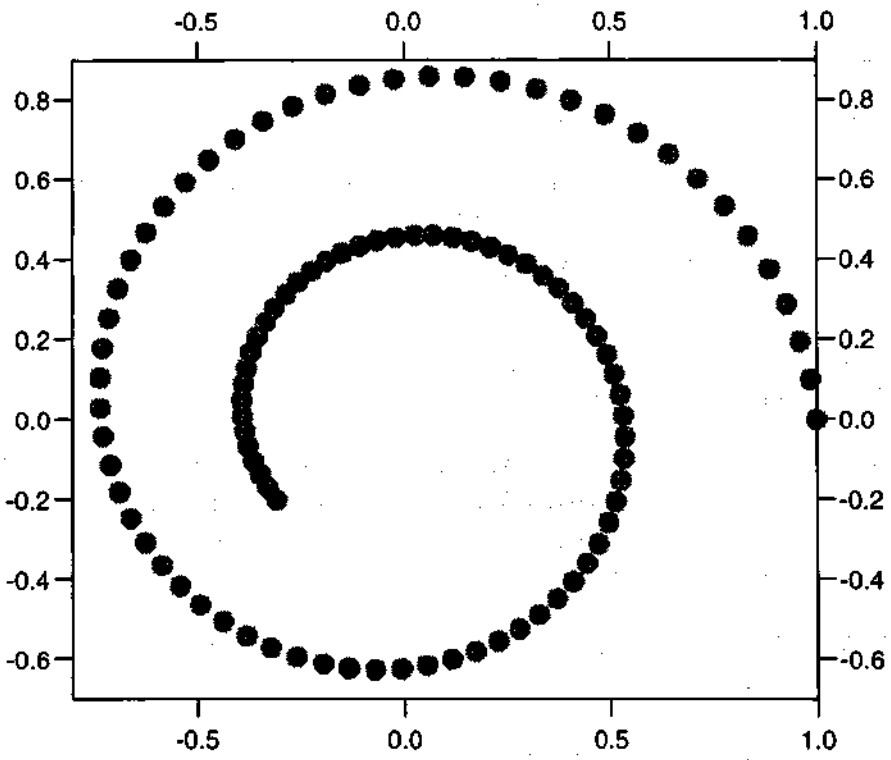


Figure 1. **Array constructors and array operations.**

Array constructors and elemental array operations can provide powerful programming techniques. This example illustrates the visualisation of a mathematical function.

19.2.4 RESHAPE

Although array constructors are only supported for one dimensional arrays, to create two and higher dimensional structures, the intrinsic function RESHAPE may be used.

Writing

```
A = (/ (i,i=1,12) /)
```

for example, a call **reshape(source=A,shape=(/3,4/))** returns a two dimensional array of the form

$$\begin{pmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{pmatrix}$$

the dimensions (3×4) and rank being defined by the parameter SHAPE. Note that the order of elements in storage, however, is not changed[4], standard column-wise ordering being assumed.

19.3 Array indices

For array indexing and operations involving permutations, (elemental) array subscripts may also now be used. Given

```
A = (/ 7,5,3,2,6 /)
I = (/ 2,1,3,5 /)
```

for instance **A(I)** returns the length four array

$$A(I) \equiv [5,7,3,6]$$

defined by **A(I(i))**, $i = 1,4$. An illustration of this technique is given in exercise 2 at the end of this chapter.

19.4 Array sections

To access subarrays, say a section of a list or a column or row of an array, the notation of array sections is introduced. Given for example

```
A = (/ 101,102,103,104,105,106,107,108,109 /)
```

the references **A(8:9)** and **A(2:5)**, for example, select[5] the length two and four subarrays **(/ 108,109 /)** and **(/ 102,103,104,105 /)**. These may be used exactly as normal arrays,

```
C = A(2:5)
```

for instance, performing the assignments

```
C(1) = A(2);   C(2) = A(3)
C(3) = A(4);   C(4) = A(5)
```

[4] This may optionally be altered by a further option **ORDER** which interchanges the order of the dimensions.

[5] Formally, we may write **A(n:m) := A((/ (i,i=n,m) /))** or in general (see below) **A(n:m:r) := A((/ (i,i=n,m,r) /))** for integer **i, n, m** and **r**.

Higher dimensional arrays can be handled similarly, **D(16:32,41:81)** representing the subarray

$$[d_{ij} \ : \ i = 16, 32, \ j = 41, 81].$$

Such examples are particularly useful in linear algebra where rectangular subarrays are frequently required. To select strides, the above notation **A(n:m)** can be extended to **A(n:m:r)** yielding for **A** above

$$\begin{aligned} A(2:6:1) &\equiv A(2:6), \\ A(2:6:2) &\equiv (/102, 104, 106/), \\ A(8:4:-1) &\equiv (/108, 107, 106, 105, 104/), \end{aligned}$$

r specifying the stride length. Equivalently for two dimensions, for the array **D** above,

```
D(16:32:8,41:81:10)
```

selects the thinned 'sub'-array

$$\begin{bmatrix} d_{16,41} & d_{16,51} & d_{16,61} & d_{16,71} & d_{16,81} \\ d_{24,41} & d_{24,51} & d_{24,61} & d_{24,71} & d_{24,81} \\ d_{32,41} & d_{32,51} & d_{32,61} & d_{32,71} & d_{32,81} \end{bmatrix}.$$

Finally, where an upper or lower bound is omitted, its value is assumed by default to coincide with that of the original matrix. For example, in **A** above **A(:5)** would be taken as **A(1:5)** and **A(3:)** as **A(3:9)**. Trivially **A(:)** selects the full extent **A(1:9)** which is simply **A**. For two or higher dimensional arrays such techniques can be especially useful by providing a simple shorthand to identify rows and columns etc. For example, given

```
logical dimension(20,50) :: X
```

the references **X(:,4)** and **X(43,:)** simply select the 4^{th} column and 43^{rd} row of **X** respectively. Likewise, dropping just one bound of the first subscript range, **X(:5,:)**, selects the first five rows of **X**. In these cases, the symbols **:**, **:**, **:5**, **:** are, in order, merely shorthands for the subranges **1:20**, **1:50**, **1:5** and **1:50**.

19.5 An example

For numerical operations, combined with elemental operations, array sections can be very powerful. Given the matrices

```
real :: A(3,4), B(4,2)
```

the $(\text{row}^{th}, \text{col}^{th})$ element of the 3×2 product $C = AB$ may be obtained directly by

```
sum(A(row,:)*B(:,col))
```

where **A(row,:)** represents the **row**-th row of **A** and **B(:,col)**, the **col**-th column in **B**. The array constructor

```
(/ (( sum(A(row,:)*B(:,col)), row=1,3), col=1,2) /)
```

then computes the entire array, C, in column order, as a one dimensional strip. Using RESHAPE, this can then be folded into the required 3×2 dimensional array,

```
reshape( &
    (/ (( sum(a(row,:)*b(:,col)), &
          row=1,3), col=1,2) /), (/ 3,2 /)  &
       )
```

thus returning the product C in a single line. Besides improving readability, the use of such built-in language features eases code optimisation by making information about program structure more clearly available to the compiler. In fact matrix-matrix products are supported directly by the array-valued intrinsic routine MATMUL

```
C = MATMUL(A,B)
```

as are inner products (cf. sum(X*Y)) by the scalar valued function DOT_PRODUCT. Given suitably typed parameters, MATMUL will also perform matrix-vector and vector-matrix computations.

19.6 Fortran90 and the BLAS

Closer examination of the handling of arrays in Fortran90 shows close parallels with techniques advanced for using the BLAS in part I of this book. Given the arrays

```
real y(4), U(8,4), M(8,4), N(4,3) NM(8,3)
```

for example, the statements

```
U=M
y=5.0*M(5,:)+y
MN=matmul(M,N)+NM
```

could be replaced by the calls

```
call scopy(4*8,M,1,N,1)
call saxpy(4,5.0,M(5,1),4,y,1)
call sgemm('No','No',8,3,4,1.0,M,8,N,4,1.0,MN,8)
```

In particular, the rôle of the stride length carries over directly[6],

```
x=y(4:16:3)
```

having the same effect as

```
call scopy(5,y(1),3,x(1),1)
```

Fortran90 thus provides simpler, easier to read ways of performing many low level operations addressed by the simpler BLAS. The rôle of BLA routines for more complicated operations, and in particular for promoting level 2 and level 3 computations, however, goes largely unchallenged.

[6] At least for positive strides. The interpretation of negative values for BLAS is described in section 9.4.

19.7 Intrinsic routines

The newly included array features introduced in Fortran 90 are further complemented by a wide range of new intrinsic functions and routines. Besides elemental extensions of existing FORTRAN 77 and scalar functions, they provide methods for returning information about array sizes and dimensions, basic shifts and reorderings and searches for maxima and minima. Although an exhaustive discussion lies beyond the scope of this text, a full list is given in appendix B. For further details, see [21].

19.8 Exercises

For scientific computing, improved array operations are one of the big gains in Fortran 90. Using its techniques, novel constructions are also possible simplifying even series and integral approximations.

1. (*SUM and array constructors*) The intrinsic function SUM takes the full form SUM(A,DIM,MASK), where DIM and MASK are optional. Specifying DIM $= i$ with a two or higher dimensional array A, the i^{th} dimension is collapsed, the result being returned as the sums for all rank-one sections through it. That is, for a $6\times4\times8$ array **A**, the call **sum(A,dim=2)** returns the 6×8 array with elements **sum(A(i,:,j))**, all values in the second dimension being added together. Specifying MASK, which should have the same shape as A, only elements from A are evaluated for which the corresponding entries in MASK are true. Use these two options to

   (a) compute the L_1- and L_∞-norms

   $$\|A\|_1 = \max_j \sum_k a_{k,j} \quad \text{and} \quad \|A\|_\infty = \max_i \sum_k a_{i,k}$$

   for a matrix $A = [a_{ij}]$ and

   (b) compute the truncated sum

   $$\sum_{1/n^2>1/1000, n>0} 1/n^2.$$

   For the first example you should also use MAXVAL which returns the largest element in an array. For the second, you should consider an array constructor.

2. (*array subscripts*) In mathematics, permutations $\{i \to a(i), i = 1 \ldots n\}$ are often denoted as n-tuples $(a(1), a(2), a(3), \ldots, a(n))$. Defining the product

   $$(a(1), a(2), a(3), \ldots, a(n))(b(1), b(2), b(3), \ldots, b(n))$$

   as $(b(a(1)), b(a(2)), b(a(3)), \ldots, b(a(n)))$, it can be shown that any permutation p can be written as the product of s cyclic permutations $p = c_1c_2c_3 \ldots c_s$. Write a program to read in such a general p and print out the powers p^r for $r \geq 1$. Apply your program to the permutation $(2, 3, 1, 4, 6, 5)$. What can you deduce about the decomposition $p = c_1c_2c_3 \ldots c_s$?

3. (*array constructors and subscripts*) The array section **A(n1:n2,m1:m2)** selects specified rectangular subsections of a 2D array A. Obtain the same result using array constructors and array subscripting.

4. (*array constructors*) Using array constructors and intrinsic functions, write a simple program to compute the rectangle rule approximation

$$\int_0^1 x^2\,dx \approx \frac{1}{N}\sum_{i=0}^{N-1}(i/N)^2.$$

How could you modify the approach to more general integration rules? Could other functions be integrated using this technique?

20 Control Structures

To aid programming, a number of new control structures are included in Fortran 90. Although much improved on FORTRAN 66, FORTRAN 77 still fell short of the provision provided by more contemporary languages. Fortran 90 goes some way towards meeting this gap but retains the same basic forms as the earlier standards. In particular, *begin-end* blocks (see section 13.4.1) are not introduced.

20.1 Labels and DO-loops

To improve readability, symbolic labels are now permitted for IF-blocks and DO-loops. Limited checks may also be performed automatically by the compiler for program correctness. For example, instead of writing

```
if (i .gt. 0) then
   if (j .gt. 0) then
      n = 1
   else
      n = 2
   endif
else
   if (j .gt. 0) then
      n = 3
   else
      n = 4
   endif
endif
```

we can now write

```
outer : if (i > 0) then
   inner : if (j > 0) then
      n = 1
   else inner
      n = 2
   endif inner
else outer
   inner : if (j > 0) then
```

```
      n = 3
   else inner
      n = 4
   endif inner
endif outer
```

where **inner** and **outer** are two symbolic labels. Applied to DO-loops, labels serve an additional purpose. In conjunction with the EXIT statement they provide a *completer*[1], branching control to the end of a named loop. The code

```
search : do i=1,n
   do i=1,n
      do j=1,m
         if (grid(i,j) <> 0) exit search
      end do
   end do
end do search
```

for example, searches through **grid** until the first non-zero element is found. Note, here, the use of the now standard DO-END DO notation first introduced as a non-standard FORTRAN 77 extension in section 13.4.1. If the label **search** were omitted, then only the inner loop would be exited. Dropping the DO-loop count, indefinite 'forever' loops can be constructed,

```
x = 0.0
do
   if (x > 100.0) exit
   read(*,*) jump
   x=x+jump
end do
```

Used together with a completer, such loops provide a structured alternative to so-called 'DO-WHILE' or 'REPEAT-UNTIL' constructs featured in other languages[2]. Finally, using the CYCLE statement, a form of *repeater* can also be constructed. In the loop

```
loop : do i=1,3
   write(*,*) 'enter PIN'
   read(*,*) number
   if (number <> 4311) cycle loop
   write(*,*) 'That was correct!'
   exit
end do
```

for instance, CYCLE causes control to be skipped to the end of the current iteration before starting the next. In this example, EXIT causes the loop to be ended completely when the correct number is typed in. A maximum of 3 tries are allowed.

[1] *Repeaters* and *completers* were originally discussed in section 13.4.1.

[2] A DO-WHILE construct is in fact introduced in the new standard but as a *new redundant* feature. Its use is thus not recommended and it may be removed at a later revision.

20.2 The select-case construct

The *select-case* construct

```
print_number : select case(number)
   case(1)
      write(*,*) 'one'
   case(2)
      write(*,*) 'two'
   case default
      write(*,*) 'neither one nor two'
end select print_number
```

provides a structured alternative for some IF-THEN-ELSE statements. Here, for example, the above code-segment prints out the words '*one*', '*two*' or an informative message, conditional on an integer **number**. String types and/or ranges are also supported:

```
door_number : select case(first_letter)
   case('A':'S')
      write(*,*) 'go to Ms. F. Lang'
   case('T':'Z')
      write(*,*) 'go to Mr. D. Hilf'
end select door_number
```

selecting a civil servant on the basis of an initial letter. The exact interpretation of the ranges **'A':'S'** and **'T':'Z'** is, however, compiler dependent. On some systems for instance, **'z' > 'A'** is **.true.** in which case the above example could give rise to confusing results. Access to the standard ASCII collating sequence can always be made directly using the intrinsic functions ACHAR, IACHAR, LGE, LGT, LLE and LLT but is typically slower. Alternatively, in the above example using the lines

```
case('a':'s', 'A':'S')
case('t':'z', 'T':'Z')
```

lower case letters can be treated directly, the first line selecting the cases **'a':'s'** or **'A':'S'**. An example using numeric ranges is shown below[3].

```
rail_travel : select case(age)
   case(0:4)
      write(*,*) 'travel free'
   case(5:15)
      write(*,*) 'can travel cheaply with &
                 &a family rail card'
   case(16:24)
      write(*,*) '1/3 off with a young &
                 &persons rail card'
   case(60:)
```

[3] The use of a double ampersand && is a special facility for continuation lines inside literal strings.

```
      write(*,*) 'at last: a senior persons rail card'
   case default
      write(*,*) 'forget it: take the bus'
end select rail_travel
```

This was true for the UK at the time of first writing. Other countries may have different fare structures.

20.3 Mask arrays

Limited logical control is also possible using so-called *mask arrays*. These are used to modify actions of selected intrinsic routines on the basis of simple logical conditions. Consider, for example, a real array **A**. Writing

```
sum(A)
```

computes the sum of all its elements. Alternatively, writing

```
sum(A,MASK=A>=5)
```

or simply

```
sum(A,A>=5)
```

selects only those elements greater than or equal to 5. As explained in section 19.2.2, the elemental comparison **A** >= 5 expands into a logical array the same size and shape as **A**. Passing this to the optional parameter MASK, only those elements in **A** for which the corresponding entries in MASK are true are added, that is to say, only those greater than or equal to 5. **A** >= 5 is said to be a *mask array*. Mask arrays are supported by a number of Fortran 90 routines and can provide useful short-cut notations for many simple problems. For further details, see again the listing of intrinsic functions in appendix B.

20.4 WHERE

Similar functionality to mask arrays is provided for array assignments by the WHERE construct. Writing, for example,

```
where (x>0)
   x = +1
elsewhere
   x = -1
end where
```

sets all the elements of **x** greater than zero to +1, and all those less than or equal to zero to −1. The construct is however quite general, also supporting multiple assignments:

```
where (x > 0.0)
   log_x = log(x)
   ok = 'yes'
elsewhere
   log_x = 0.0
   ok = 'no'
end where
```

defining a real array **log_x** and a corresponding character array **ok** conditional on the contents of a real array **x**.

21

Input/Output in Fortran90

Input/output has traditionally been one of the strengths and weaknesses of Fortran. Its wealth of file operations and format editors have long been the envy of other languages whereas its reliance on record style line-by-line mode I/O rather than simple character streams have made it sometimes clumsy and difficult to integrate with modern software. Fortran 90 provides an alternative to this model as well as making a number of technical improvements to existing FORTRAN 77 features. This chapter gives a brief survey of these topics. For further details see [21].

21.1 Records and character streams

Input/output in Fortran is *record* or *line* orientated. Originally motivated by punched cards, Fortran regards files as being composed as collections of records (lines) and treats these as the basic entities for input/output. READ, PRINT and WRITE statements all read and write one line at a time.

File handling in languages such as PASCAL and C is, however, different. There, newlines, and hence lines, have no special significance and newline characters can be set, tested for and deleted like any other. Unlike in Fortran, whole files can be treated as simple character sequences – character streams – possibly (or possibly not) including new line characters. The line/record model thus no longer applies. Such an approach is often more flexible. Lines like

The cat sat on the mat

can, for example, be read character-by-character, and hence word-by-word, as required by multiple read statements. This simplifies, for example, possible input parsing. Compare this to FORTRAN 77 where the above line could only be read by a single READ statement.

In Fortran 90 character stream style I/O is however partially supported and is known as *non-advancing*. Setting the option ADVANCE to 'NO' the normal 'newline' linefeed at the end of input or output operations can be suppressed. Thus the code

```
do i=1,10
   read(*,'(a1)',advance='no') word(i:i)
   if (word(i:i) == ' ') exit
end do
```

would read from the above line character-by-character until a blank – and hence the end of the first word – was reached. Likewise, the sequence

```
write(*,*,advance='no') 'Good '
write(*,*,advance='no') 'Morning '
write(*,*,advance='no') 'Franziska '
```

would produce the result

```
Good Morning Franziska /u/david/f90/egs>
```

The UNIX prompt in italics is included to show the absence of any terminal newline. Applications of non-advancing input/output are widespread ranging from interactions with files created by other languages to implementing simple user interface prompts such as

```
enter name >  Lang
access no  >  864237
```

By default, using ADVANCE = 'YES', Fortran 90 behaves like FORTRAN 77.

21.2 NAMELIST

Another useful addition to Fortran 90 is the NAMELIST construct. Given

```
character (len=10) :: name
real :: debt
integer :: account
```

and defining

```
namelist /collect/ name,debt,account
```

a reference to the *namelist* **collect**

```
write(*,nml=collect)
```

automatically produces a record

```
&COLLECT NAME = George     , DEBT =   5.0000000E+04, ACCOUNT = 124/
```

the listed variables bracketed by a '&' to the left and a '/' to the right. If required, these values may be read back by a corresponding READ statement

```
read(*,nml=collect)
```

thus providing a simple self-documenting technique to read and write machine readable data. The reader should however note that, although supported as an extension to many FORTRAN 77 compilers, the exact form of the records read and written by NAMELIST was not standardised until Fortran 90.

21.3 Internal files

As pointed out in chapter 5, Fortran 90 now supports list directed I/O to and from internal files. Such versatile statements as

```
string='0 34 2 -12.003'
read(string,*) i,j,k,z
```

are thus now supported as part of the standard. A useful example of this facility is given in exercise 2 of chapter 5.

21.4 OPEN and INQUIRE

The OPEN and INQUIRE statements are also extended in Fortran 90. New attributes, READ, WRITE and READWRITE, are for example introduced to restrict[1] read/write actions on external files whereas PAD and RECL for sequential files tidy up record length handling. Of special importance for program portability, the initial rewind status on opening are now explicitly controlled by the POSITION attribute which can also be used, for example, to append data to the end of files. Complicated error trapping and more sophisticated techniques are supported by SIZE which gives details of how many bytes have been read. For further details see [21, 15].

21.5 New format editors

Fortran 90 also includes a number of new format editors. Of special note are B, O and Z which convert integers to and from binary, octal and hexadecimal notation. For numerical applications the 'scientific' ES and 'engineering' EN descriptors may also be of use, producing output of the form

```
 2.12343E+13   2.76E-3   4.665E-7
```

and

```
 21.23343E+12  2.76E-3    .4665E-6
```

respectively. ES simply ensures that the decimal point is placed after the first digit whereas EN adjusts the output format such that the exponent is a multiple of 3. Finally, the new standard also extends the G edit descriptor to include integer, character and logical types. For operations on binary values, a range of intrinsic routines for bit manipulation are provided. These too are listed in appendix B.

[1] These restrictions have no effect on the operating system properties of files, but merely the range of operations that Fortran can do on them.

22

High Performance Fortran

22.1 Motivation

Implicit in our discussion so far has been the assumption that all programs are executed sequentially on a single processor. Although this still remains by far the most common case, it is by no means universal. Much attention is now being paid to so called *parallel architectures* in which computationally expensive tasks are divided, distributed and executed concurrently 'in parallel' on more than one processing unit. By combining the power of many processors, it is thus hoped to achieve a considerable reduction in overall computing time.

High Performance Fortran (HPF) is an extension of Fortran 90[1] designed to give direct support to high performance architectures and parallel systems. By giving the programmer a framework to specify (where necessary) how data should be distributed across different processors and to identify what parts of the algorithm can potentially be performed concurrently, it aims to provide the compiler with all the required information needed to generate efficient high performance or parallel object code automatically. As such, HPF may be contrasted with alternative systems such as PVM [38] which give the programmer more direct control over parallel architectures. In the HPF standard, details of individual processor configuration and communication are not addressed directly but are left to the discretion of the compiler. In this way, HPF seeks to be easier to use and less architecture dependent, being in principle portable across a wide range of high performance platforms.

For convenience we will divide our discussion into three distinct areas:

- the distribution of data across processors,
- identifying potential parallelism and
- intrinsic and extrinsic routines.

Our treatment of these topics is, however, only superficial and for a full introduction the reader is directed to [34].

[1] The draft [34] also defines a subset language, *subset HPF*, including only a part of Fortran 90 and limited elements from HPF.

22.2 The data model

Computation is the process of applying operations to data. Thus where computation is to be performed in parallel, the data must also be distributed. In **HPF** this is a two stage process: first a PROCESSORS directive is used to define a *virtual* processor configuration (see below) before one or more DISTRIBUTE directives are used to define rules by which individual named arrays are mapped onto individual processors.

22.2.1 Defining the processors

Since HPF sets out to be machine independent, the constructs it uses do not refer directly to physical processors themselves but to a model arrangement of 'virtual' or *abstract* devices defined by the user. How closely these correspond to the host's actual architecture, or how the communication between them is actually performed, is not addressed by the standard. It is implementation dependent[2].

However implemented, HPF (abstract) processors are assumed to be organised into one or more linear or rectilinear arrays defined by PROCESSORS directives. The lines

```
!HPF$ processors, dimension(128,64) :: grid
!HPF$ processors, dimension(2,2,2,2) :: cube
```

define, for example, a 128×64 processor grid and a 2^4 element hypercube. Here, as throughout the language, the characters **!HPF$**, nominally a Fortran 90 comment, act as an escape sequence to identify a HPF-directive.

22.2.2 Distributing the data

Once a processor configuration has been defined, data (array elements) can be distributed across it by the DISTRIBUTE directive. This is typically performed *blockwise* – the array is first split into blocks of consecutive elements which are then allocated *in blocks* across the processor array – or *cyclically* – where consecutive elements are mapped on to adjacent neighbouring processors on the above defined processor grid. Given a simple processor array

```
!HPF$ processors, dimension(10) :: linear
```

and two arrays **x** and **y** of size 1000, two simple examples might be

```
!HPF$ distribute (cyclic) onto linear :: x
!HPF$ distribute (block) onto linear :: y
```

which produce the *cyclic*

[2] It is however suggested that manufacturers provide machine dependent directives capable of doing this. The standard merely requires that no more physical processors are used than the abstract devices defined in the program.

processor 1:	$x(1)$,	$x(11)$,	$x(21)$,	$\cdots$	$x(991)$
processor 2:	$x(2)$,	$x(12)$,	$x(22)$,	$\cdots$	$x(992)$
processor 3:	$x(3)$,	$x(13)$,	$x(23)$,	$\cdots$	$x(993)$
$\vdots$	$\vdots$	$\vdots$			$\vdots$
processor 10:	$x(10)$,	$x(20)$,	$x(30)$,	$\cdots$	$x(1000)$

and *blockwise* element-processor distributions

processor 1:	$y(1)$,	$y(2)$,	$y(3)$,	$\cdots$	$y(100)$
processor 2:	$y(101)$,	$y(102)$,	$y(103)$,	$\cdots$	$y(200)$
processor 3:	$y(201)$,	$y(202)$,	$y(203)$,	$\cdots$	$y(300)$
$\vdots$	$\vdots$	$\vdots$			$\vdots$
processor 10:	$y(901)$,	$y(902)$,	$y(903)$,	$\cdots$	$y(1000)$

respectively. More complicated or realistic distributions are possible using suitable combinations or extensions of these two directives.

22.2.3 Communication

The careful distribution of arrays over different processors can also provide simple forms of aggregate communication. Rotating the above matrix **x** cyclically to the right[3], for example,

```
x = cshift(x,shift=1)
```

with the Fortran 90 intrinsic routine CSHIFT, has the effect of copying each element of **x** (cyclically) from each processor by one to its right-hand neighbour: a simple example of message passing. Alternatively, the loop[4]

```
do copy=1,10
   do i=1,100
      y(i+100*(copy-1))=A(i)
   enddo
enddo
```

broadcasts identical copies of the segment **A(1:100)** to every processor. Details of this and related communication techniques are left to the reader.

22.2.4 Alignment

A central goal in parallel computing is to reduce to cost of inter-processor communication. Thus, where values act on other values, they should ideally be kept on the same processor. In HPF this can be enforced by the ALIGN directive. Consider for example the following computation

$$u(i) = v(i+1) + w(i-1)$$

[3] That is **x(1):=x(1000)**, **x(2):=x(1)**, **x(3):=x(2)** ... **x(1000):=x(999)**, the operation being performed concurrently.

[4] If **y** was two dimensional, the Fortran 90 intrinsic routine SPREAD could possibly be used.

for varying i. Communication costs are then clearly minimised if u, v and w can be distributed in such a way that $u(i)$, $v(i+1)$ and $w(i-1)$ always lie on the same processing unit. In HPF, this may be requested by the following directives:

```
!HPF$ align u(i) with v(i+1)
!HPF$ align u(i) with w(i-1)
```

the use of **i** being purely symbolic. This is in contrast to the simpler case

```
!HPF$ align u(i) with v(i)
!HPF$ align u(i) with w(i)
```

otherwise written as

```
!HPF$ align u : v, w
```

where $u(i)$, $v(i)$ and $w(i)$ are requested to lie on the same processor for all i. Although in principle introducing nothing new (appropriate array-processor distributions could in theory always be coded by hand), ALIGN provides a simple and transparent shorthand to specify relationships and interdependencies between data values and so help minimise inter-processor communication.

Technically the distribution of an array over a processor grid is known as a 'template'. Through the use of the TEMPLATE directive it is in HPF possible to define pure templates in the form of 'template arrays'. These have a template (element-processor distribution) but no elements or storage. Such empty or dummy structures (template arrays are sometimes referred to as 'arrays of nothings' as distinct from 'arrays of integers', 'arrays of reals' etc.) are often useful as frameworks against which to align normal arrays. For further discussion and examples see [94].

22.3 Identifying parallelism

22.3.1 FORALL

Consider the loop:

```
do i=1,n
    do j=1,m
       y(i) = y(i) + a(i,j)*x(j)
    enddo
 enddo
```

Clearly, upon reflection, Fortran DO-loops not only define operations, but they also dictate the *order* in which they are performed. This latter restriction is often unnecessary (possibly resulting in less efficient code) and may be removed in HPF by means of the FORALL statement[5]:

```
forall(i=1:n,j=1:m) y(i)=y(i)+a(i,j)*x(j)
```

[5] This is a *statement* and *not* a directive. Directives (preceded by an !HPF$ string) are advisory and so may be removed without affecting the semantics of the program. This is not true for *statements* such as FORALL.

This has a similar effect as the above DO-loops, but leaves the order in which the assignments are to be made open to the compiler. FORALL's main use is to provide a convenient and compact notation to express such array operations. For example, the call

```
forall(i=1:n,j=1:n) h(i,j) = 1.0/(i+j)
```

performs exactly the same operations as the more cumbersome Fortran 90

```
h = 1.0 / &
          (spread( (/i,i=1,n/), dim=1, ncopies=n) + &
           spread( (/j,j=1,n/), dim=2, ncopies=n))
```

where SPREAD now generates the two rank-one matrices

$$[1,2,3,\ldots,n]^T[1,1,1,\ldots,1] \quad \text{and} \quad [1,1,1,\ldots,1]^T[1,2,3,\ldots,n].$$

Conditional statements are also allowed:

```
forall (i=1:1000, x(i) > 0.01) x(i) = 1.0/x(i)
```

as are intrinsic functions, and in certain cases [34], user defined functions.

A major difference between the FORALL statement and the standard Fortran DO-loop, lies, however, in the way their contents are evaluated. In a FORALL statement, *all the expressions on the right hand side are evaluated before any assignment is made.* It resembles thus more a generalised array assignment[6] or array valued function than a normal control structure. It would, for example, allow

```
forall (i=1:n,j=1:n) A(i,j) = A(j,i)
```

for the matrix transpose $A := A^T$ with no need for intermediate storage, as in the more cumbersome Fortran 90 direct translation:

```
do i=1,n
   do j=1,n
      temp(i,j) = a(j,i)
   enddo
enddo

a = temp
```

For more involved constructs, a FORALL *construct* is provided,

```
forall(i=1:200,j=1:200)
   x(i,j) = a(i,j) - b(i,j)
   z(i,j) = sin(x(i,j)-x(j,i))
end forall
```

being semantically equivalent to a *sequence* of *separate* FORALL statements:

```
forall(i=1:200,j=1:200) x(i,j) = a(i,j) - b(i,j)
forall(i=1:200,j=1:200) z(i,j) = sin(x(i,j)-x(j,i))
```

[6] FORALL may be thought of as a generalisation of the Fortran 90 WHERE. See [19, 34] and section 20.4.

22.3.2 Independence

Finally, potential parallelism can be flagged by means of the INDEPENDENT directive. Consider the loop

```
do i=1,199
   u(i) = a(i) + a(i+1)
   v(i) = a(i) - a(i-1)
enddo
```

Each iteration is now *independent* in that it could in principle be performed separately without requiring or affecting the results of other iterations. In particular, the iterations could be distributed and executed concurrently on different processors without requiring any inter-processor communication for the duration of the loop. This can be flagged by the directive

```
!HPF$ independent
do i=1,199
   u(i) = a(i) + a(i+1)
   v(i) = a(i) - a(i-1)
enddo
```

Such information can then be used to assist the compiler to produce parallel code. Similarly, where appropriate, FORALL constructs are also supported and nested loops can be handled through the use of the additional NEW [34] option. With these refinements INDEPENDENT clearly plays a central rôle in parallel environments by allowing the programmer to identify concurrent code directly in the Fortran source.

22.4 Library and extrinsic routines

22.4.1 The HPF_LIBRARY

Further scope for parallelism, amongst other things, is provided by the HPF-standard module HPF_LIBRARY. Lying outside the scope of this text, this includes a large number of array handling routines and routines to support parallel environments.

22.4.2 Extrinsic routines

Finally, interfaces to other languages are supported by a new routine type *extrinsic*. Specified as EXTRINSIC(ADA), EXTRINSIC(PASCAL) or EXTRINSIC(C), for example, interface blocks could be constructed for direct calls from HPF to subprograms written in *Ada*, *Pascal* or *C*. Calls to other parallel languages or machine code are possible in this way. An especially interesting possibility, however, is that of *local routines* set out in [34]. Given subprograms written in some non-parallel *serial* language, interfaces can be constructed so that a single invocation of an extrinsic procedure from HPF spawns a separate local invocation of the subprogram on each processor. Used correctly, such techniques make way for the direct programming of processors according to a *single instruction multiple data* model. An alternative

parallel-free version of HPF, known to EXTRINSIC as HPF_LOCAL, is provided for this purpose.

Appendix A

The BLAS

A.1 The Basic Linear Algebra Subprograms

We here list in brief the level 1, 2 and 3 BLAS. To save space, the first letter of each routine, the second for `I_AMAX`, has been replaced by an underscore '_'. This should be replaced by `S` (single precision), `D` (double precision), `C` (complex) or `Z` (double precision complex – COMPLEX*16), where appropriate, depending on the type required. For example, the single and double precision versions of the generic routine `_SWAP` could be called by `SSWAP` and `DSWAP` respectively. The list of available versions for each routine is given in the prefixes column. Where routines operate with mixed types, two or three prefix letters may be used, the first designating the type of the result. For example, `CSSCAL` computes the complex result of a complex vector scaled by a real scalar. For norms of complex vectors, two prefix letters are always used (eg. `SCNRM2`) since the norm of a complex vector is always real. For level 2 BLAS, a set of extended-precision routines with prefixes `ES`, `ED`, `EC` and `EZ` may also be available. Double precision complex routines (prefix `Z`) may not be supported on all machines.

A.2 Level 1 BLAS

These perform basic vector-vector and vector-scalar operations. The routines (marked '*') `_DOT`, `_DOTU`, `_DOTC`, `_NRM2`, `_ASUM` and `I_AMAX` are FORTRAN functions. All other routines in levels 1, 2 and 3 are subroutines.

prefixes	*dim scalar vector vector array*	
s,d	`_ROTG ( A, B, C, S )`	Generate plane rotation
s,d	`_ROTMG ( D1, D2, A, B, PARAM )`	Generate modified plane rotation
s,d	`_ROT ( N, X, INCX, Y, INCY, C, S )`	Apply plane rotation
s,d	`_ROTM ( N, X, INCX, Y, INCY, PARAM )`	Apply modified plane rotation
s,d,c,z	`_SWAP ( N, X, INCX, Y, INCY )`	$x \leftrightarrow y$
s,d,c,z,cs,zd	`_SCAL ( N, ALPHA, X, INCX )`	$x \leftarrow \alpha x$
s,d,c,z	`_COPY ( N, X, INCX, Y, INCY )`	$y \leftarrow x$
s,d,c,z	`_AXPY ( N, ALPHA, X, INCX, Y, INCY )`	$y \leftarrow \alpha x + y$
s,d	`_DOT* ( N, X, INCX, Y, INCY )`	$dot \leftarrow x^T y$
c,z	`_DOTU* ( N, X, INCX, Y, INCY )`	$dot \leftarrow x^T y$
c,z	`_DOTC* ( N, X, INCX, Y, INCY )`	$dot \leftarrow x^H y$
s,d,sc,dz	`_NRM2* ( N, X, INCX )`	$nrm2 \leftarrow \|x\|_2$
s,d,sc,dz	`_ASUM* ( N, X, INCX )`	$asum \leftarrow \|\Re(x)\|_1 + \|\Im(x)\|_1$
s,d,c,z	`I_AMAX* ( N, X, INCX )`	the first k such that $\|\Re(x_k)\| + \|\Im(x_k)\| = \max(\|\Re(x_i)\| + \|\Im(x_i)\|)$

A.3 Level 2 BLAS

Level 2 BLAS perform operations between vectors and matrices. As in LAPACK, to simplify notation, the first two letters following the prefix are used to denote the matrix type: `GE` – general, `SY` – symmetric, `HE` – Hermitian, `TR` – triangular, `GB` – general banded, `SB` – symmetric banded, `HB` – Hermitian banded, `TB` – triangular banded, `SP` – symmetric packed, `HP` – Hermitian packed and `TP` – triangular packed.

For both level 2 and 3 BLAS, CHARACTER*1 character strings are used to control the routine's exact function. For these, the following standard values are allowed: `TRANS_` (generic for `TRANS`, `TRANSA` or `TRANSB` where `TRANSA` and `TRANSB` refer to matrices A and B) = 'No Transpose', 'Transpose' or 'Conjugate Transpose' (meaning X, X^T and X^H), `UPLO` = 'Upper triangular' or 'Lower triangular', `DIAG` = 'Non-unit triangular', 'Unit triangular' and `SIDE` = 'Left', 'Right' (referring to the position of A relative to B). Where a character string with length greater than one is passed (as above), only the first character is read. Longer strings can, however, improve readability. Where matrices are modified by `TRANS_`, we here use the notation $X^{\square}$ where $^{\square}$ is null, T or H depending on the value of `TRANS_`. For example, `TRANS_ = 'C'` denotes $X^{\square} = X^H$. Further $X^{-\square}$ indicates $(X^{\square})^{-1}$. For the subroutine `_GEMM`, $A^{\square A}$ and $B^{\square B}$ refer to the two variables `TRANSA` and `TRANSB`.

A.3.1 Matrix-vector operations

prefixes	*options*	*dim*	*b-width*	*scalar*	*matrix*	*vector*	*scalar*	*vector*	
s,d,c,z	_GEMV(TRANS,	M,N,		ALPHA,A,	LDA,X,INCX,	BETA,Y,INCY)			$y \leftarrow \alpha A^{\square} x + \beta y, A \sim m \times n$
s,d,c,z	_GBMV(TRANS,	M,N,	KL,KU,	ALPHA,A,	LDA,X,INCX,	BETA,Y,INCY)			$y \leftarrow \alpha A^{\square} x + \beta y, A \sim m \times n$
c,z	_HEMV(UPLO,	N,		ALPHA,A,	LDA,X,INCX,	BETA,Y,INCY)			$y \leftarrow \alpha A x + \beta y$
c,z	_HBMV(UPLO,	N,K,		ALPHA,A,	LDA,X,INCX,	BETA,Y,INCY)			$y \leftarrow \alpha A x + \beta y$
c,z	_HPMV(UPLO,	N,		ALPHA,AP,	X,INCX,	BETA,Y,INCY)			$y \leftarrow \alpha A x + \beta y$
s,d	_SYMV(UPLO,	N,		ALPHA,A,	LDA,X,INCX,	BETA,Y,INCY)			$y \leftarrow \alpha A x + \beta y$
s,d	_SBMV(UPLO,	N,K,		ALPHA,A,	LDA,X,INCX,	BETA,Y,INCY)			$y \leftarrow \alpha A x + \beta y$
s,d	_SPMV(UPLO,	N,		ALPHA,AP,	X,INCX,	BETA,Y,INCY)			$y \leftarrow \alpha A^{\square} x$
s,d,c,z	_TRMV(UPLO,TRANS,DIAG,	N,		A,	LDA,X,INCX)				$y \leftarrow \alpha A^{\square} x$
s,d,c,z	_TBMV(UPLO,TRANS,DIAG,	N,K,		A,	LDA,X,INCX)				$y \leftarrow \alpha A^{\square} x$
s,d,c,z	_TPMV(UPLO,TRANS,DIAG,	N,		AP,	X,INCX)				$y \leftarrow \alpha A^{\square} x$
s,d,c,z	_TRSV(UPLO,TRANS,DIAG,	N,		A,	LDA,X,INCX)				$y \leftarrow \alpha A^{-\square} x$
s,d,c,z	_TBSV(UPLO,TRANS,DIAG,	N,K,		A,	LDA,X,INCX)				$y \leftarrow \alpha A^{-\square} x$
s,d,c,z	_TPSV(UPLO,TRANS,DIAG,	N,		AP,	X,INCX)				$y \leftarrow \alpha A^{-\square} x$

A.3.2 Vector-matrix operations

prefixes	*options*	*dim*	*scalar*	*vector*	*vector*	*matrix*		
s,d	_GER (	M, N,	ALPHA,	X, INCX,	Y, INXY,	A,	LDA)	$A \leftarrow \alpha xy^T + A, A \sim m \times n$
c,z	_GERU (	M, N,	ALPHA,	X, INCX,	Y, INXY,	A,	LDA)	$A \leftarrow \alpha xy^T + A, A \sim m \times n$
c,z	_GERC (	M, N,	ALPHA,	X, INCX,	Y, INXY,	A,	LDA)	$A \leftarrow \alpha xy^H + A, A \sim m \times n$
c,z	_HER (UPLO,	N,	ALPHA,	X, INCX,		A,	LDA)	$A \leftarrow \alpha xx^H + A$
c,z	_HPR (UPLO,	N,	ALPHA,	X, INCX,		AP)		$A \leftarrow \alpha xx^H + A$
c,z	_HER2 (UPLO,	N,	ALPHA,	X, INCX,	Y, INXY,	A,	LDA)	$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$
c,z	_HPR2 (UPLO,	N,	ALPHA,	X, INCX,	Y, INXY,	AP)		$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$
s,d	_SYR (UPLO,	N,	ALPHA,	X, INCX,		A,	LDA)	$A \leftarrow \alpha xx^T + A$
s,d	_SPR (UPLO,	N,	ALPHA,	X, INCX,		AP)		$A \leftarrow \alpha xx^T + A$
s,d	_SYR2 (UPLO,	N,	ALPHA,	X, INCX,	Y, INXY,	A,	LDA)	$A \leftarrow \alpha xy^T + \alpha yx^T + A$
s,d	_SYR3 (UPLO,	N,	ALPHA,	X, INCX,	Y, INXY,	AP)		$A \leftarrow \alpha xy^T + \alpha yx^T + A$

A.4 Level 3 BLAS

Level 3 BLAS perform matrix-matrix operations. Triangular solves are supported but full rectangular factorisations are not. Not all level 3 BLAS routines may be supported on all systems.

prefixes	*options*	*dim*	*scalar*	*matrix*	*matrix*	*scalar*	*matrix*	
s,d,c,z	_GEMM (TRANSA, TRANSB,	M, N, K,	ALPHA,	A, LDA,	B, LDB,	BETA,	C, LDC)	$C \leftarrow \alpha A^{\square_A} B^{\square_B} + \beta C$
s,d,c,z	_SYMM (SIDE, UPLO,	M, N,	ALPHA,	A, LDA,	B, LDB,	BETA,	C, LDC)	$C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$
c,z	_HEMM (SIDE, UPLO,	M, N,	ALPHA,	A, LDA,	B, LDB,	BETA,	C, LDC)	$C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$
s,d,c,z	_SYRK (UPLO, TRANS,	N, K,	ALPHA,	A, LDA,		BETA,	C, LDC)	$C \leftarrow \alpha A^{\square}(A^{\square})^T + \beta C$
c,z	_HERK (UPLO, TRANS,	N, K,	ALPHA,	A, LDA,		BETA,	C, LDC)	$C \leftarrow \alpha A^{\square}(A^{\square})^H + \beta C$
s,d,c,z	_SYRK2K (UPLO, TRANS,	N, K,	ALPHA,	A, LDA,	B, LDB,	BETA,	C, LDC)	$C \leftarrow \alpha A^{\square}(B^{\square})^T \alpha B^{\square}(A^{\square})^T + \beta C$
c,z	_HERK2K (UPLO, TRANS,	N, K,	ALPHA,	A, LDA,	B, LDB,	BETA,	C, LDC)	$C \leftarrow \alpha A^{\square}(B^{\square})^H + \alpha B^{\square}(A^{\square})^H + \beta C$
s,d,c,z	_TRMM (SIDE, UPLO, TRANSA,	DIAG, M, N,	ALPHA,	A, LDA,	B, LDB)			$B \leftarrow \alpha A^{\square} B$ or $\alpha BA^{\square}$
s,d,c,z	_TRSM (SIDE, UPLO, TRANSA,	DIAG, M, N,	ALPHA,	A, LDA,	B, LDB)			$B \leftarrow \alpha A^{-\square} B$ or $\alpha BA^{-\square}$

In the above table, C is assumed $n \times n$ except in the routines _GEMM, _SYMM and _HEMM where its shape is $m \times n$. In _TRMM and _TRSM, $B \sim m \times n$ whilst in _SYMM and _HEMM we require that $A = A^T$ and $A = A^H$ respectively.

Appendix B

Intrinsic Functions in Fortran90

Intrinsic Functions in Fortran90

The following gives a brief overview of the intrinsic routines supported in the new standard, Fortran90. For further details, and for details of language elements not covered in this book, the reader is referred to [21].

B.1 Numerical and array routines

B.1.1 Basic mathematical functions

The following functions are supported as in FORTRAN 77, execpt that they now also support arrays. With array arguments, they are applied elementwise to each element: **sin(A)**, for example, returns the result $[sin(a_i)]$ for each element a_i of one dimensional **A**.

> **sin, cos, tan, exp, log, log10, asin, acos, atan, atan2, sinh, cosh, tanh, sqrt, real, dble, int, aint, abs, ninit, anint, aimag, cmplx, mod, sign, dim, conjg, dprod, min, max**

Of these, the functions REAL, INT, AINT, NINT, ANINT and CMPLX have been extended for support types defined with the KIND-attribute. In addition, four new functions are provided:

- **CEILING** and **FLOOR** return the least and largest integers above and below a given real,
- **MODULO** provides an alternative version of MOD, potentially more useful for negative arguments, and
- **LOGICAL** is equivalent to the function REAL, but for logicals.

Note that as in FORTRAN 77, MIN and MAX can have two or more arguments.

B.1.2 Functions for use with the KIND-attribute

To define values for use with the KIND-attribute, the following functions may be used:

- **KIND** returns the KIND-value associated with a given parameter,

- **SELECTED_INT_KIND** returns the KIND-value required to implement an integer with a given range, and
- **SELECTED_REAL_KIND** returns the KIND-value required to implement a real variable with a given range or precision.

B.1.3 Array operations

For direct manipulations on arrays the following functions are introduced:

- **MATMUL** performs (mathematical) matrix-matrix, matrix-vector or vector-matrix products: AB, Ax or xA,
- **DOT_PRODUCT** computes the dot- or inner-product[1] $x.y = \sum x_i y_i$,
- **TRANSPOSE** forms the transpose of a 2D array,
- **SUM** returns the sum of all the elements in a given array, and
- **PRODUCT** returns the corresponding product.

Further for a given array

- **MIN-** and **MAXVAL** return the minimum and maximum values and
- **MIN-** and **MAXLOC** their corresponding indices.

For logical arrays, the function

- **ALL** is true if and only if all the elements are true,
- **ANY** is true if and only if at least one element is true and
- **COUNT** counts the number of true elements.

To return information about arrays the following enquiry functions may be used:

- **UBOUND** and **LBOUND** give information about upper and lower array bounds
- **SIZE** returns the array's size or information about the *extents*[2] of array dimensions
- **SHAPE** returns a rank one array listing the extents of each dimension. This is referred to as the array's *'shape'*, and
- **ALLOCATED** (for allocatable arrays) reports whether or not storage has been allocated.

More general array transformations may be effected by

- **RESHAPE** which reshapes the elements of a one dimensional vector into a 2 or higher dimensional array,

[1] That is for numeric types `sum(x*y)`.
[2] That is the *range*, the difference, between the upper and lower bounds.

- **PACK** and **UNPACK** which compresses and uncompresses a general array into a one dimensional vector under the control of a logical masking array, or
- **SPREAD, CSHIFT** and **EOSHIFT** which replicate copies of an array in a given direction, or shift its elements cyclically or acyclically.

Further, two arrays may be merged, under the control of a logical parameter, MASK, using the elemental merge function MERGE.

B.1.4 Numerical accuracy and type enquiry functions

Based upon an idealised model of machine arithmetic, the following provide machine dependent information about intrinsic types:

- **DIGITS** returns the number of significant digits representable for a given type,
- **EPSILON** gives the corresponding machine precision and
- **HUGE** yields the largest representable real or integer.

Related information can be obtained with the functions MAXEXPONENT, MINEXPONENT, PRECISION, RADIX, RANGE and TINY. For real numbers, interaction with the real arithmetic model is also possible using the routines EXPONENT, FRACTION, NEAREST, RESPACING, SCALE, SET_EXPONENT and SPACING. Note that the model on which all these routines are based is nominal, and on many systems may only be approximate.

B.1.5 Enquiry function for pointer variables

- **ASSOCIATED** returns whether a pointer is associated with a (possibly named) target.

B.2 Non-numerical routines

B.2.1 Character handling

For character variables the FORTRAN 77 commands (now elemental)

char, ichar, index, lle, lge, lgt, len

are provided together with the new commands ACHAR and IACHAR defined in terms of the standard ASCII collating sequence, and LEN_TRIM, TRIM, ADJUSTL, ADJUSTR, SCAN, REPEAT and VERIFY for more general purposes.

B.2.2 Bit manipulation and data conversion

Based on the US Military Standard MIL-STD 1753, Fortran 90 provides a number of procedures for manipulating bits held in integer storage. These are

bit_size, btest, iand, ibclr, ibits, ibset, ieor, ior, ishift, ishftc, not, mvbits

and with the exception of BIT_SIZE are all elemental. They are described in full in [21]. To convert data values between different types without changing their physical representation, a further function TRANSFER is provided.

B.2.3 Time and random numbers

Finally, to provide standardised time and random-number facilities, the following routines are introduced:

- **DATE_AND_TIME** gives the current date, time and time-zone,
- **SYSTEM_CLOCK** gives access to the system clock,
- **RANDOM_NUMBER** returns a random number from a uniform distribution defined on $[0, 1)$ and
- **RANDOM_SEED** provides access to the seed value (or array) used by RANDOM_NUMBER.

The use of these routines is simplified by the use of optional keyword parameters. If no system clock is supported, default values are assumed.

Appendix C
Solutions to Exercises

C.1 Part I – Scientific Computing

Chapter 1 – Double and Single Precision

1. (*machine precision*) Basically yes. Floating point numbers are typically represented in the form $f \times b^e$ where f is a p-digit base-b fraction $0.f_1f_2f_3 \ldots f_p$ such that $f_1 \neq 0$ and the exponent e is restricted to within certain bounds. The machine precision can then be shown to be b^{1-p} or b^{-p} depending on the type of rounding used, b is, of course, nearly always a power of 2. In some implementations, however, the true value for the machine precision ϵ may differ, very slightly, for technical reasons. For example to distinguish between $1+\epsilon$ and $1-\epsilon$ under the IEEE binary single precision standard, ϵ is set at $2^{-24}+2^{-47}$. Fine details of machine arithmetic are sometimes, thus, non-trivial. This account is based upon the comments given in the NAG library documentation [37], chapter **X02**. For a readable introduction to the IEEE floating point standard see [12]; for a more general discussion, see [14].

2. (*real comparisons*) Exact comparisons between floating point numbers are never safe. They can often give results other than you expect. Consider for example the equality

```
(0.2 + 0.2 + 0.2 + 0.2 + 0.2) .EQ. 1.0
```

This is not guaranteed to be true. Since the decimal fraction 0.2 has no direct binary representation (it is, in fact, a recurring fraction $0.0011\dot{}_2$ in a binary or binary based floating point number system – see previous question) its value can not be represented in full and must so be truncated. Thus, performed on a floating point machine in binary the sum

```
(0.2 + 0.2 + 0.2 + 0.2 + 0.2)
```

may be slightly different from one. Where real numbers are used to control DO-loops such effects may be particularly important: the number of iterations performed may depend on whether intermediate sums are rounded up or down. For example, in single precision on the author's HP 9000 series 700, the loop

```
      do 1 x= 0.0,0.2,1.0
         write(*,*) x
 1    continue
```

executed as expected 6 times since intermediate sums were rounded down whereas

```
      do 1 x= 1.0,0.2,2.0
         write(*,*) x
 1    continue
```

achieved only 5 iterations, the sum

```
1.0 + 0.2 + 0.2 + 0.2 + 0.2 + 0.2
```

narrowly overshooting 2.0. (In Fortran, DO-loops iterate until their index variable crosses their second DO-loop bound. Index variables are tested at the head of the DO-loop at the start of each iteration.) Such effects are of course arbitrary and inevitable. They are, of course, even possible for loops like

```
      do 1 x= 0.0,0.1,1.0
         write(*,*) x
 1    continue
```

since even if the number 0.1 is machine representable (as 1.0×10^{-1} on some systems), the intermediate sums 0.2, 0.3 etc. may not be.

Potential rounding effects and machine representation should therefore always be taken into account when writing numerical programs: equality comparisons between floating point values should never be trusted. Well written codes instead often ask if two floating point numbers lie within a specified distance of each other, this value being given as a user- or system-defined tolerance. Similar techniques should be used for .LE. and .GE.. Finally, particular care should be taken using DO-loops with real index variables and integer based loops are often preferred. Indeed in some languages and contexts, e.g. array constructors in Fortran 90 (section 19.2.3), REAL loop indices are not allowed. Furthermore, in Fortran 90 they have been declared an obsolescent feature due for removal in a future revision.

3. (*optimised inequalities?*) No, not always. Notice, for example, how sensitive the condition **(1.0 .lt. (1.0 + x))** is to incorrect optimisation. A compiler which spotted, and optimised, the mathematical relationship $1 < 1 + x \Leftrightarrow 0 < x$ would completely change the meaning of the program. It is hoped that modern compilers will not make such errors. A further example involving associativity arises in exercise 5.

4. (*IEEE arithmetic*) In addition to normal floating point numbers, the IEEE standard also supports a number of special values such as $-\infty$, $+\infty$ and extra '*Not-a-Number*' or '*NaN*' states. These may be substituted for normal floating point representations when appropriate. In response to the requested expressions, the following values would be produced:

   (a) 0.5 (a normal number)

   (b) $+\infty$

   (c) *NaN* (Not-a-Number)

   (d) $-\infty$

   (e) *NaN* (Not-a-Number)

   In particular, the division **c/0** yields *NaN* where $c = 0$ and $\pm\infty$ where $c \neq 0$, the sign of $\pm\infty$ being that of c.

   The special values returned by the above operations can be further used in further computations: no error exceptions are generated. For example, the identity $1/(1/x) = x$ now remains valid for all values including $x = 0$ since, in this number system, $1/0 = \infty$ and $1/\infty = 0$. The case $x = \pm\infty$ is also supported through the use of signed zeros ± 0, also included in the standard. To detect the presence of

infinities, or NaNs, a number of enquiry function and status flags are also provided. The exact form of these facilities in any implementation, however, is not defined. For further details see [12].

The automatic handling of arithmetic exceptions such as that outlined above can have many applications for numerical algorithms. The treatment of weak singularities (e.g. $\tan x$ for $x = \pi/4$), graph plotting and out-of-range function evaluations in iterations include just a few.

5. (*arithmetic identities*) Floating point arithmetic need not always be associative. Consider for example the extreme case where $x = 10^{30}$, $y = -10^{30}$ and $z = 1$ where 10^{30} lies within the machine's arithmetic range but 10^{-30} is less than the machine roundoff. Then, the two expressions $(x+y)+z$ and $x+(y+z)$ are not equal, taking values of 1 and 0 respectively. You may need to work this out on a piece of paper, applying the normal machine roundoff rules. In particular, $((x+y)+z)-(x+(y+z))$, in this floating point system, gives a value of one. Clearly, given this distinction, $((x+y)+z) \leftrightarrow (x+(y+z))$, special care should be taken during optimisation, not to assume associativity, and floating point comparisons should be treated with care.

In IEEE arithmetic [12], the statement $x = y \Leftrightarrow x - y = 0$ is however guaranteed but this is non-trivial. On, say, a three digit computer with numbers of the form[1]

$$0.f_1f_2f_3 \times 10^e \tag{C.1}$$

for $f_1 \neq 0$ and $|e| \leq 98$, the two numbers 0.459×10^{-98} and 0.452×10^{-98} are clearly not equal but the difference

$$0.459 \times 10^{-98} - 0.452 \times 10^{-98} = 0.007 \times 10^{-98}$$

is too small to be represented in the form (C.1). It should thus be rounded down to zero. This would imply that $x = y \not\Leftarrow x - y = 0$. In cases of this type, to avoid such problems, the IEEE standard allows the normalisation requirement $f_1 \neq 0$ to be dropped. The resulting number, which has fewer significant digits and less precision than normal, is then said to be *denormalised* or *subnormal*. Such *gradual underflow* is enough to guarantee the statement $x = y \Leftrightarrow x - y = 0$.

6. (*over- and underflow*) Consider the following code segment:

```
k = 0
alpha = 1.0e+15

pr = 1.0

do r=1,n
   pr = pr * s(r)
   if (abs(pr) .gt. alpha) then
      k = k + 1
```

[1] Actually a representation of $d_1.d_2d_3 \times b^e$ would be used but we follow the representation of exercise 1 and take $b = 10$.

```
      pr = pr / alpha
    elseif (abs(pr) .lt. 1.0/alpha) then
      k = k - 1
      pr = pr * alpha
    endif
  enddo

  pr = pr * alpha**k
```

Given a real number range of at least $\pm 10^{\pm 30}$, setting $\alpha = 10^{15}$ and assuming that each $|s_i| \in [10^{-15}, 10^{+15}]$, the term $\overline{p}_r s_r$ (i.e. **pr*p(r)**) remains in bounds. The multipliers α^k are then tracked by the variable k and reintroduced after the DO-loop has been completed. To test this routine, consider products of the form $(10^{10} \times 10^{10} \times 10^{10} \times 10^{10} \times 10^{-10})$ or $(10^{-10} \times 10^{-10} \times 10^{-10} \times 10^{-10} \times 10^{10})$.

More advanced variations on this algorithm using over- and underflow flags and the full floating point word are also possible. These, too, are discussed in [12].

7. (*type promotion*) Site dependent.

Chapter 3 – The NAG Library

1. (*a simple example – ODE's*) The NAG library contains a wide range of routines to solve ODE problems. For general usage – simple non-stiff problems where high accuracy is not required – the Runge-Kutta routines are usually recommended. For instance, to compute the end-point solution (here at $t = 50$) the routine D02BAF can be used. To comply with the NAG documentation, references to t have been changed to x.

```
program ode1
integer n,ifail
double precision x,xend,y(3),tol,W(3,7)
external fcn

x = 0.
xend = 50.
y(1) = 1.
y(2) = 1.
y(3) = 1.

n = 3
tol = 0.001
ifail = 0

call d02baf(x,xend,n,y,tol,fcn,w,ifail)

write(*,*) x,y

stop
```

```
      end
```

This calls a subroutine:

```
      subroutine fcn(x,y,yp)
      double precision x,y(3),yp(3)

      yp(1) = 10.*(y(2)-y(1))
      yp(2) = 28.*y(1) - y(2) - y(1)*y(3)
      yp(3) = y(1)*y(2) - 8./3. * y(3)

      return
      end
```

For 'intermediate' or 'dense' output (where solution values are output at intermediate points between the starting and ending points), the routine D02BBF should be used.

```
      program ode2
      integer n,irelab,ifail
      double precision x,xend,y(3),tol,W(3,7)
      external fcn,output

      x = 0.
      xend = 50.
      y(1) = 1.
      y(2) = 1.
      y(3) = 1.

      n = 3
      tol = 0.001
      irelab = 0
      ifail = 0

      call d02bbf(x,xend,n,y,tol,irelab,fcn,output,w,ifail)

      stop
      end
```

This calls a further subroutine **output** to output solution values:

```
      subroutine output(xsol,y)
      double precision xsol,y(3)

      write(*,*) xsol,y
      xsol = xsol + 0.01

      return
      end
```

starting with the initial point, and to advance the variable **xsol** which determines the next required output point. In the above example, the solution is thus output at all points from 0.0 to 50.0 at 0.01 intervals. Special interpolation schemes are used to do this. A 3D plot of the resulting solution is shown in figure 1.

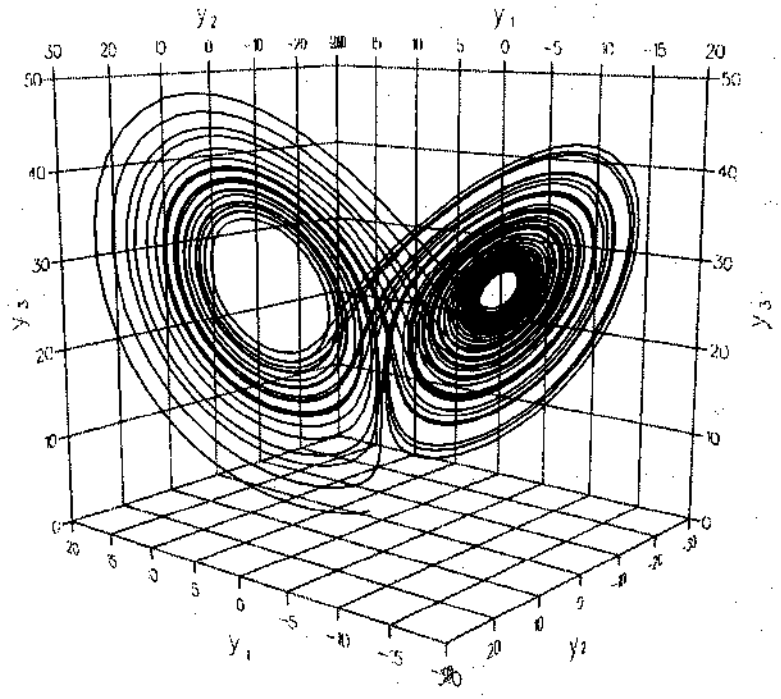


Figure 1. **Using D02BBF: a solution to Lorenz's equations.**

2. (*integration*) Using the preferred integrator D01AJF, the following code could be written:

```
      program quad2

      integer lw, liw
      parameter(lw=800, liw=lw/4)
      integer ifail, iw(liw)
      double precision a,b,epsabs,epsrel,result,
     +   abserr,w(lw),f
      external f

      a = 0
      b = 1

      epsabs = 0.0
      epsrel = 0.1e-6
```

```
      ifail  = 1

      call d01ajf(f,a,b,epsabs,epsrel,result,abserr,
     +            w,lw,iw,liw,ifail)

      write(*,*)
      write(*,*) ' ifail    ', ifail
      write(*,*) ' abserr   ', abserr
      write(*,*) ' result   ', result
      write(*,*)

      stop
      end
```

The function f is defined as for D01AHF. Work arrays, such as the parameters **w** and **iw**, are discussed in chapter 7. For vector architectures (section 10.2), the routine D01ATF may be preferred.

3. (*sorting*) One possible program would be

```
      program sort

      integer limit
      parameter(limit=10)
      integer perm(limit),ifail,i
      character*10 names(limit)

      do 1 i=1,limit
         read(*,'(a10)',end=2) names(i)
 1    continue

      write(*,*) 'program limited to ', limit, ' names'
      stop

 2    continue

      ifail = 0
      call m01dcf(names,1,i-1,1,10,'ascending',perm,ifail)
      call m01zaf(perm,1,i-1,ifail)

      do 3 i=1,i-1
         write(*,*) names(perm(i))
 3    continue

      stop
      end
```

The sorting is here split into two parts. M01DCF first computes the *rank* (that is the index of each entry from the input list in the final sorted list) of each input element. This list is returned as **perm**. Then the second routine, M01ZAF, inverts this array to compute the corresponding *indices* to be used directly by the second DO-loop to print out the results. Differences between *ranks* and *indices* may be confusing when sorting lists. The above list could also have been sorted automatically, and more simply, with a single call to the routine M01CCF.

4. (*errors and file handling*) Selecting IFAIL $= -1$ does not terminate the program on the detection of a NAG error but prints out a message detailing the error ('noisy exit'). After return, the value of IFAIL can then be tested by the calling program. To control the unit number to which the message is written, the user can use the optional routine X04AAF. To print out a double precision array, the routines X04ABF, X04CAF, and optionally, X04CBF, may be used.

5. (*machine and mathematical constants*) The required values may be obtained by the NAG routines X01AAF (π), S14AAF ($\Gamma(1.5)$), X02AJF (machine precision), X02AKF (smallest positive number) and X02BBF (largest integer) respectively. These numbers are of course only approximate, the last three being based on an *artificial model* of the machine's arithmetic. As illustrated by the answer to exercise 1 of chapter 1, the exact implementation of machine arithmetic is often non-trivial.

6. (*random permutations*) One simple way of producing a random permutation is by ranking the elements of a random array:

```
      program cards

      integer n
      parameter (n=52)

      integer seed,rank(n),m1,m2,ifail
      double precision a,b,x(n)
      character*1 order

      external g05cbf, g05faf,m01daf

c     ... read in and set seed

      write(*,*) 'enter seed'
      read(*,*) seed

      call g05cbf(seed)

c     ... generate random vector

      a=0.0
      b=1.0
      call g05faf(a,b,n,x)
```

```
c     ... and rank its elements

      m1=1
      m2=n
      order='ascending'
      ifail=0
      call m01daf(x,m1,m2,order,rank,ifail)

      write(*,*) rank

      stop
      end
```

The routine G05FAF first fills the double precision array **x** with random numbers in the range $[0, 1]$ and these are then ranked by M01DAF, the result being written to **rank**. After completion, **rank(1)** contains the index of the least element of **x**, **rank(2)** the second smallest and so on. Since the elements of **x** are randomly distributed, so must be the resulting permutation **rank**. The routine G05CBF is used to define the *seed*, an integer number used to initialise the random number generator, G05FAF. For each different seed, a different random number sequence can be generated.

Chapter 4 – Save, Reverse Communication and Arrays

1. (*subroutines with memory*) Given the routine

```
      subroutine store(action,index,value)

      integer index,value,array(1000)
      character*1 action

      save array

      if (action .eq. 's') then
         array(index) = value
      elseif (action .eq. 'r') then
         value = array(index)
      else
         write(*,*) 'store: action unknown'
         stop
      endif

      return
      end
```

integers can be stored and retrieved with calls of the form **call store('store', index,value)** and **call store('read',index,value)** where **index** is the number of

the array location and **value** the value to be copied. A working copy of this routine should also check that **index** does not lie out of range. Note also that although **action** has length one, general strings may be passed to this routine but only the first character is used. The routine has the advantage that it has no hidden COMMON blocks and that, if it is accessed by several different routines, changes to the size of the array **array** are localised. It is however complicated and slow (passing through COMMON can be even faster than passing parameters). It is thus not recommended.

2. (*a practical example*)

```
subroutine range(action,value)

double precision value,lower,upper
character*1 action
logical first
save first,upper,lower

if (action .eq. 'i') then
   first = .true.
elseif (action .eq. 'r') then
   if (first) then
      upper = value
      lower = value
      first = .false.
   else
      upper = max(value,upper)
      lower = min(value,lower)
   endif
elseif (action .eq. 'p') then
   write(*,*) 'the range is :',lower, upper
else
   write(*,*) 'range: action unknown'
endif

return
end
```

After first initialising with a call with **action = 'initialise'**, data values can be read in setting **action = 'read'**. The selection **action = 'print'** then prints out the current range. For two or higher dimensional input streams, the scalars **upper** and **lower** would have to be replaced by the corresponding arrays. As above, this routine can be called with length one character strings for **action**.

3. (*reverse communication*) Simply replace the routine sums by

```
      subroutine sums(prompt,x,fx)

      integer prompt
      double precision x, fx, sum
      save sum
      if (prompt .eq. 0) then
         sum = 0.0
         prompt = 1
         x = 1.0
         return
      elseif ((prompt .ge. 1)
     +    .and. (prompt .le. 9)) then
         sum = sum + fx
         prompt = prompt + 1
         x = x + 1.0
         return
      elseif (prompt .eq. 10) then
         sum = sum + fx
         prompt = 0
         fx = sum
         return
      endif

      write(*,*) '*** prompt should be in'
      write(*,*) '    the range 0 to 10 '
      stop

      end
```

Since the value for $f(x)$ is computed within the calling program itself, no extra communication is required to address problems of the form

$$\sum_{i=1}^{10} f(i,b).$$

We would simply need to write a call of the form

```
      ...,
      prompt = 0

    1 continue
      call sums(prompt,x,fx)

      if (prompt .eq. 0) then
         write(*,*) 'sum = ', fx
```

```
      else
         fx=f(x)
         goto 1
      endif
      ....
```

Compare this to the forward communication case where f would be evaluated by a function passed to **sums**, and an extra parameter, COMMON blocks or similar techniques would be needed to pass a value of b to this routine to correctly evaluate the sum.

4. (*Newton iteration*) One routine choice for **newton** would be

```
      subroutine newton(prompt,x,fx,dfx,eps)

      integer prompt
      double precision x, fx, dfx, eps

      if (prompt .eq. 0) then
c        ... request f(x) and f'(x) values for
c            next step
         prompt = 1
         return
      elseif (prompt .eq. 1) then
         if (abs(fx) .le. eps) then
c           ... iteration completed
            prompt = 0
            return
         else
c           ... another interation
            prompt = 1
            x = x - fx/dfx
            return
         endif
      endif

      write(*,*) '** error: prompt value not known'
      end
```

called by

```
      program root

      integer prompt
      double precision a, x, fx, dfx, eps

      write(*,*) 'enter a ...'
      read(*,*) a
```

```
c     ... set x, eps and prompt to start

      x = 2.0
      eps = 0.00000001
      prompt = 0
1     continue

      call newton(prompt,x,fx,dfx,eps)

      if (prompt .eq. 0) then
         write(*,*) 'answer = ', x
      else
         fx = x*x - a
         dfx = 2.0*x
         goto 1
      endif

      stop
      end
```

newton responds to the first call **prompt** $= 0$ with a request, **prompt** $= 1$, for the function and derivative values, $f(x)$ (**fx**) and $f'(x)$ (**dfx**) given x (**x**) and continues thus until $|f(x)| \leq$ **eps** in which case the iteration is stopped, and control returned to the user with **prompt** $= 0$. Note again the attraction of reverse communication that since the function $f(x) = x^2 - a$ is computed outside **newton**, in program **root** no extra communication is needed to pass a value for **a**.

5. (*Ackermann's function*) Given a starting value (m, n), the recursive application of the given rules

$$A(m,n) = \begin{cases} n+1 & : \quad m = 0 \\ A(m-1,1) & : \quad m \neq 0, n = 0 \\ A(m-1, A(m,n-1)) & : \quad m \neq 0, n \neq 0 \end{cases} \tag{C.2}$$

to the expression $A(m, n)$ results in a sequence of expressions of the form:

$$\begin{aligned} A(2,1) &= A(1, A(2,0)) \\ &= A(1, A(1,1)) \\ &= A(1, A(0, A(1,0))) \\ &= A(1, A(0, A(0,1))) \\ &= A(1, A(0,2)) \\ &= \ldots \end{aligned}$$

Although the derivation of these terms is in terms of a recursive rule, their form is not complicated and can be characterised by simple variable-length integer lists:

$$\begin{aligned} A(1,2) &\leftrightarrow [1,2] \\ A(1,A(2,0)) &\leftrightarrow [1,2,0] \\ A(1,A(1,1)) &\leftrightarrow [1,1,1] \\ A(1,A(0,A(1,0))) &\leftrightarrow [1,0,1,0] \\ A(1,A(0,A(0,1))) &\leftrightarrow [1,0,0,1] \\ A(1,A(0,2)) &\leftrightarrow [1,0,2] \\ \ldots &\leftrightarrow \ldots \end{aligned}$$

Representing these lists by an array, e.g. by

$$[B_1, B_2, B_3, \ldots B_{k+2}, \ldots]^T$$

for a list of length $k+2$, and noting that the rules (C.2) are applied only to the innermost one or two indices of any expression, suggests the following algorithm:

```
      program acker

      integer m
      parameter (m=1000)
      integer i,k,B(m)

      read(*,*) B(1), B(2)
      k=0

 1    continue

      if (B(k+1) .eq. 0) then
         B(k+1) = B(k+2) + 1
         k=k-1
      else
         if (B(k+2) .eq. 0) then
            B(k+2) = 1
            B(k+1) = B(k+1) - 1
         else
            B(k+3) = B(k+2) - 1
            B(k+2) = B(k+1)
            B(k+1) = B(k+1) - 1
            k=k+1
         endif
      endif

      if (k+2 .gt. m) then
         write(*,*) 'error: array B too small'
         stop
      endif
```

```
      if (k .ge. 0) goto 1

      write(*,*) B(1)

      stop
      end
```

Ackermann's function is interesting because it demonstrates how an apparently complex recursive definition can be resolved by a simple 'non-recursive' algorithm. Although such simple examples are not general, they suggest that careful analysis and thought can sometimes lead to efficient codes. A recursive implementation (see the solutions to the exercises in chapter 17) would at any stage require $k+1$ simultaneous function invocations where $k+2$ is the length of the corresponding list above.

Chapter 5 – Internal Files

1. (*pull*) Simply remove the REWIND statement.

2. (*expression parsing*) One possible solution would be

```
      real function evalf(string)
      integer i
      real x,y
      character string*(*)
      character*1 op, letter

      do 1 i=1,len(string)
         letter = string(i:i)
         if ((letter .eq. '+') .or. (letter .eq. '-') .or.
     +       (letter .eq. '*') .or. (letter .eq. '/')) then
            op = letter
            string(i:i) = ' '
         endif
    1 continue

      read(string,*) x,y

      if (op .eq. '+') then
         evalf = x + y
      elseif (op .eq. '-') then
         evalf = x - y
      elseif (op .eq. '*') then
         evalf = x * y
      else
         evalf = x / y
      endif
```

```
      return
      end
```

Noting its value, the routine first removes the operand replacing it by a blank in **string**. **string** then contains just two values, namely the two operands. These are read by an internal read and the result computed. Note how this step is greatly simplified by the use of list directed I/O which allows the numbers **x** and **y** to occur anywhere in the string **string**. Without the internal read, real and integer numbers would have to be decoded by hand.

3. (*printing an array*) One suitable routine would be

```
      subroutine mprint(n,A,lines)

      integer n, row, col
      double precision A(n,n)
      character*13 form
      character*80 lines(n)

c     ... make the format statement
      write(form,'(a1,i2,a10)') '(', n, '(f7.3,1x))'

c     ... and use it to write out the matrix
      write(lines,form) ((A(row,col), col=1,n),row=1,n)

      return
      end
```

which could be called by the program

```
      program demo

      integer n
      parameter (n=8)

      integer row,col
      double precision A(n,n)
      character*80 lines(n)

c     ... set up demo matrix
      do 1 col=1,n
         do 2 row=1,n
            A(row,col) = float(10*row+col)
 2       continue
 1    continue

c     ... generate character array
```

```
      call mprint(n,A,lines)

c     ... and output the result
      write(*,'(a80)') lines

      stop
      end
```

Note the use of the first internal file, **form**, to set up the format statement needed by the second WRITE statement to write the array. Such functionality would be clearly more difficult without internal files. More sophisticated routines for printing out arrays to internal files are provided in the NAG library. See the solution to exercise 4 of chapter 3 for details.

4. (*string handling*) One might, for example, write

```
      subroutine upper(string)
      integer i,j
      character*(*) string
      character*26 small,large

      small = 'abcdefghijklmnopqrstuvwxyz'
      large = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

      do 1 i=1,len(string)
         do 2 j=1,len(small)
            if (string(i:i) .eq. small(j:j)) then
               string(i:i) = large(j:j)
               endif
 2          continue
 1       continue

      return
      end
```

This routine illustrates the use of character substrings. It is not, however, necessarily efficient.

Chapter 6 – Arrays in FORTRAN

1. (*higher dimensional arrays*) The index in **ZZ** is simply

$$1 + (a-1) + (b-1) * r + (c-1) * r * s + (d-1) * r * s * t$$

Note how only the three leading dimensions (**r**, **s**, **t**) are used. The final dimension **u** is not required.

2. (*skew dimensions*) Experimentation left to the reader

3. (*a block algorithm*) First note that

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

where all A_{ij} and B_{ij} are 2×2. Then, after first setting all the elements of **T** to zero, the following code may be used:

```
call mult2(R(1,1),4,S(1,1),4,T(1,1),4)
call mult2(R(1,3),4,S(3,1),4,T(1,1),4)

call mult2(R(3,1),4,S(1,1),4,T(3,1),4)
call mult2(R(3,3),4,S(3,1),4,T(3,1),4)

call mult2(R(1,1),4,S(1,3),4,T(1,3),4)
call mult2(R(1,3),4,S(3,3),4,T(1,3),4)

call mult2(R(3,1),4,S(1,3),4,T(3,3),4)
call mult2(R(3,3),4,S(3,3),4,T(3,3),4)
```

each block computing the 2×2 sub-matrices T_{11}, T_{21}, T_{12} and T_{22} respectively where

$$T = \begin{pmatrix} T_{11} & T_{12} \\ T_{21} & T_{22} \end{pmatrix}$$

The choice of $AB + C \mapsto C$ is clearly necessary to form sums $A_{11}B_{11} + A_{12}B_{21}$, $A_{11}B_{12} + A_{12}B_{22}$ etc. directly without the array accesses and storage needed for a temporary intermediate storage. The BLA routine for matrix multiplication, DGEMM (chapter 10) thus also adopts this form.

4. (*an illegal example*) A suitable form for **star5** would be

```
double precision function star5(h,v)
double precision h, v(0:1,0:1)

star5 = (-v(-1,0)-v(0,1)-v(1,0)-v(-1,0)+4.0*v(0,0))/h**2

return
end
```

The function is clearly illegal, however, with constant out-of-bounds element references. These should, indeed, be detected by the compiler. To see how this routine works, set up the array **u** as

```
      do 1 row=1,100
         do 2 col=1,100
            u(row,col) = row*10000+col
2        continue
1     continue
```

and trace the corresponding values of **v(0,0)**, **v(-1,0)** etc. from inside the routine using print statements. **v(0,0)** should take, for example, the value passed as **u(i,j)**.

Chapter 8 – Memory Efficiency

1. (*page faults*) As we will see, the number of potential page faults incurred for each algorithm depends on the order in which its elements are multiplied. We consider each example in turn, starting with the innermost loop and then working out.

   Consider the first example (i). This is the 'standard' ordering you might write after thinking about matrix multiplication as an inner-product:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

It is however very inefficient. For example, in the innermost loop

```
            do 3 k=1,n
               C(i,j) = A(i,k)*B(k,j) + C(i,j)
3           continue
```

although the elements of B and C are accessed column-wise with stride one, the elements of A are accessed *row-wise* with stride 1024 (2^{10}). Thus, every 64 (2^6) columns a new page boundary is crossed ($2^6 \times 2^{10} = 2^{16}$) and 16 ($2^4$) pages must be read to complete the row. This results in 16 potential page faults. Extending our analysis to the middle loop

```
         do 2 j=1,n
            do 3 k=1,n
               C(i,j) = A(i,k)*B(k,j) + C(i,j)
3           continue
2        continue
```

we note that for each loop in **j**, B and C now range across their full extents. Thus in addition to the 16 potential faults possible for each call to the inner loop, 2 times 16 faults may occur for accesses to B and C. Repeating this 1024 times for the outer loop **i=1,1024**, we arrive at a total of $1024 \times (16 \times 1024 + 32)$, in excess of 16 million potential page faults for the calculation of $C = AB + C$ for three 1024×1024 matrices. At a cost of 0.5 seconds per page fault, this would imply a total I/O time of over 97 days. On a CYBER 205, on which this example is based, this should be compared with a CPU time of about 7 minutes.

Whether or not an I/O time of 97 days would be realised in practice is however an open question. Real machines may hold several pages at any one time in their main memory and crossing a page boundary does not automatically result in a page fault. This figure of 97 is thus exaggerated but 16 million potential I/O calls is nevertheless very high and can, as we will see, by a simple loop reordering, be drastically reduced.

To explain the arithmetic used in these examples, you may find figure 2(a) of help. For each loop (in **k**, **j** or **i** – printed for clarity in capitals to the left) the accessed elements in each array are shaded together with any relevant indices. Where page faults can occur, their potential number is printed at the bottom left of each box. Example (ii) is much more efficient than example (i). Merely by permuting the

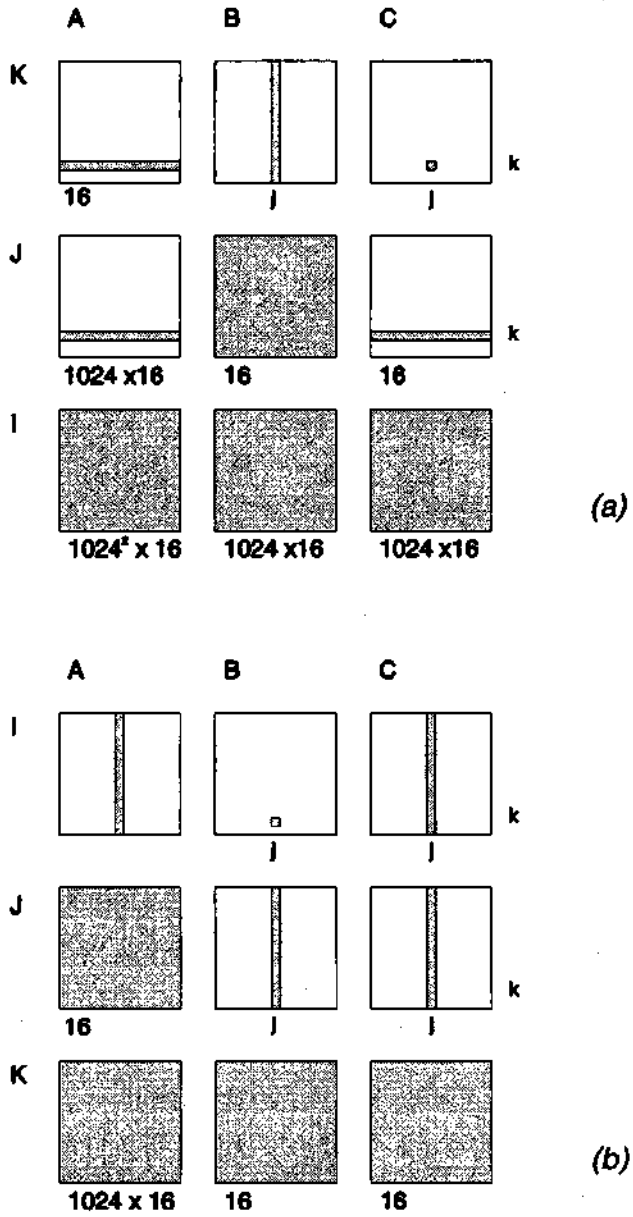


Figure 2. **Memory access patterns for the inner-product (a) and column-wise (b) matrix-matrix multiplication.**

order of the loops **i-j-k** to **j-k-i**, the number of potential page faults can be reduced by over 1,000. We illustrate this in figure 2(b). Note that now page boundaries are traversed in the innermost loop, A and C being accessed column-wise. The first chance for page faults (16 times) occurs in the middle loop where the full matrix A is accessed. Combined with 16 faults for B and C this gives a potential total of $1024 \times 16 + 32$, by the argument of (i), equivalent to about $2\frac{1}{4}$ hours. This is obviously much better and illustrates the importance of ordering and memory distribution in efficient code design.

As will be suggested in the following chapter, still better performance can be obtained using block algorithms. An example of a block algorithm for matrix-matrix multiplication was given in the solutions to chapter 6.

2. (*data reuse*) Consider a computer without any electronic memory writing all data directly to and from disk[2]. Then, even in view of the disappointing results of the last question, so soon as an element paged in from disk is accessed more than once by the CPU, a computer with memory represents a major advance on the disk-only design. This is an example of *data reuse* (or temporal data locality) and is the key to efficient programming. In writing code, our aim should be to promote it as far as possible, thus maximising the ratio of CPU accesses to memory or I/O calls.

On a smaller scale, and with smaller and faster memory, this is exactly the same principle which underlines *cache*[3]. During a program's execution, many megabytes of storage may be accessed but by structuring the code in such a way to work intensively (reusing data) on only a few kilobytes of localised store at a time, considerable savings in memory access and hence overall computing time can be achieved.

Chapter 9 – Basic Linear Algebra Subprograms (BLAS)

1. (*negative strides*) In all four cases the first 10 elements of **y** are unaffected by the DCOPY. If no or both strides are -1, the numbers $11 \ldots 20$ are copied to $y(i), i = 11 \ldots 20$ in normal (ascending) order. If only one stride is negative, the same elements are copied but in reverse order.

2. (*DDOT*)

```
      subroutine mult(n,m,r,A,B,C)

      integer n,m,r,row,col
      double precision A(n,m),B(m,r),C(n,r),ddot
      external ddot

      do 1 row=1,n
         do 2 col=1,r
            C(row,col) = ddot(m,A(row,1),n,B(1,col),1)
```

[2] By some definitions, such a device without electronic storage would not qualify as a computer at all.

[3] Similar techniques are also used by authors writing books. By only keeping the files for one or two chapters open at a time, paper-access times can be greatly decreased and a tidy office maintained.

```
 2        continue
 1     continue

       return
       end
```

3. (*Gaussian elimination*) For (a), using DAXPY and DSCAL we could write

```
       do 1 row=1,n-1

c         ... compute multipliers recording (to advance L)

          mult = 1./a(row,row)
          call dscal(n-row,mult,a(row+1,row),1)

c         ... make rank one update to (advance U)

          do 2 i=row+1,n
             call daxpy(n-row,-a(i,row),a(row,row+1),n,a(i,row+1),n)
 2        continue

 1        continue
```

DSCAL is first called to compute the multipliers $a(i, row+1)/a(row, row+1)$ used by DAXPY for the following update. A solution for (b) is given in chapter 11 while a BLAS-free version is listed in figure 2 in that chapter.

Chapter 10 – Linear Algebra and Vectorisation

1. (*level 2 BLAS*) Taking into account all operations we obtain

operation	*read/writes*	*flops*	$\approx$ *ratio*
$y = y + \alpha x$	$3n$	$2n$	$3:2$
$y = \beta y + \alpha Ax$	$mn + 2m + n$	$2mn + 3m$	$1:2$
$C = \beta C + \alpha AB$	$3mn + mk + nk$	$3mnk + 3mk$	$2:n$

ratio being the ratio of read/write calls to floating point operations. Thus, for these examples, level 2 BLAS show a threefold improvement with regard to storage and memory access over level one. This explains the strength of level 2 BLAS. The table also confirms the text's assertion that, for large n, both level 1 and 2 BLAS are outperformed by level 3.

Aside: In the above calculations, we assumed that the number of arithmetic operations required to multiply A and B was $2mnk$. Although true for most standard methods, this figure is not optimal. Theoretically, by writing down once more the multiplication as a 2-by-2 block algorithm

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} =$$

$$= \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

and then considering products of the form

$$(A_{11} + A_{22})(B_{11} + B_{22}) = A_{11}B_{11} + A_{22}B_{11} + A_{11}B_{22} + A_{22}B_{22}$$

we note that by using these terms, and a clever combination of subtractions, we can obtain our required intermediate block products for less work. Applying the process recursively to the above product, an asymptotic flop count of $O(n^{\log_2 7}) \approx O(n^{2.807})$ *can be obtained as compared to the standard* $O(n^3)$. *This is known as Strassen Multiplication: for details see [25, 13]. Moral:* $AB = C$ *is not necessarily a* $2mnk$ *operation.*

2. (*memory management*) This can differ widely between systems. Consult your local documentation or the general literature.

3. (*vectorisation speed-up*) See section 10.3.1. The speed-up rarely exceeds 2 or 3 times. *Vectorisation is not parallelisation.*

4. (*level one BLAS*) Techniques for speeding up plain code are given in chapter 14. Make sure all optimisation options are specified. The relative efficiency of the produced binaries is system dependent.

5. (*right looking Cholesky*) If $A = A^T$ the equations simplify, $A_{12} = L_{11}U_{12}$ and $A_{21} = L_{21}U_{11}$ becoming identical. Thus step (2) becomes redundant. Further the operations of factorising A_{11} and updating A_{22} become faster as, by symmetry, only half the number of elements are involved. Thus the symmetric block method uses half the number of floating point operations as the full matrix version. We refer to factorisation of this type, $A = LL^T$, as *Cholesky factorisation.*

6. (*a left looking algorithm*) We first note

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ & U_{22} & U_{23} \\ & & U_{33} \end{pmatrix}$$
$$= \begin{pmatrix} L_{11}U_{11} & L_{11}U_{12} & L_{11}U_{13} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} & L_{21}U_{13} + L_{22}U_{23} \\ L_{31}U_{11} & L_{31}U_{12} + L_{32}U_{22} & L_{31}U_{13} + L_{32}U_{23} + L_{33}U_{33} \end{pmatrix}$$

In particular, this gives us

$$U_{12} = L_{11}^{-1}A_{12}$$

and

$$\begin{aligned} L_{22}U_{22} &= A_{22} - L_{21}U_{12} \\ L_{32}U_{22} &= A_{32} - L_{31}U_{12}. \end{aligned}$$

Writing $A_{22}^* = A_{22} - L_{21}U_{12}$ and $A_{32}^* = A_{32} - L_{31}U_{12}$ this suggests the following algorithm to advance the factorisation of A by one block column to the right. Recall that we know L_{11}, U_{11}, L_{21} and L_{31} and wish to find U_{12}, U_{22}, L_{22} and L_{32}:

1. solve $U_{12} = L_{11}^{-1} A_{12}$,
2. form $A_{22}^* = A_{22} - L_{21}U_{12}$ and $A_{32}^* = A_{32} - L_{31}U_{12}$,
3. factorise $A_{22}^* \rightarrow L_{22}U_{22}$ and
4. solve $L_{32} = A_{32}U_{22}^{-1}$.

Repeating this, moving the block column $[A_{12}^T, A_{22}^T, A_{32}^T]^T$ successively to the right, this thus provides a method of advancing the factorisation to a complete factorisation $A = LU$ of the matrix A. This is called a *left looking algorithm* (see below). Where pivoting is performed, stages 3. and 4. may be combined into one single factorisation step.

The derivation of the terms *left* and *right looking algorithm* comes from the memory access properties of the two methods. Where no pivoting is performed, we summarise this for a typical block step in figure 3, the shaded areas being those

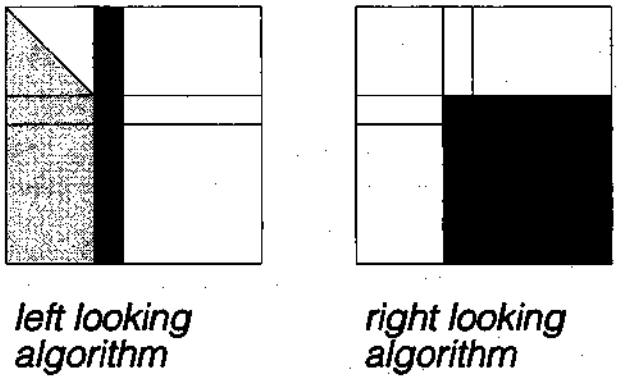


Figure 3. **Memory access for right and left looking algorithms.**

accessed by memory, the heavy shading indicating memory writes. The pivoted case is left to the reader.

The pattern of memory accesses in the left looking algorithm turns out to be very efficient on many serial machines. Interestingly, however, it can perform very badly on parallel architectures. In such cases, say, a right looking algorithm might be preferred.

7. (*block implementation*) Each of the three remaining operations could be broken up into further block stages. Writing

$$\begin{aligned} A_{12} &= \left[A_{12}^{(1)} A_{12}^{(2)} A_{12}^{(3)} \ldots A_{12}^{(r)}\right] \\ A_{21} &= \left[A_{21}^{(1)T} A_{21}^{(2)T} A_{21}^{(3)T} \ldots A_{21}^{(s)T}\right]^T \end{aligned}$$

we simply note that $L_{11}^{-1}A_{12} = U_{12}$ is equivalent to $L_{11}^{-1}A_{12}^{(i)} = U_{12}^{(i)}$ and $A_{12}U_{11}^{-1} = L_{21}$ to $A_{12}^{(j)}U_{11}^{-1} = L_{21}^{(j)}$ for suitable ranges of i and j. Further, we may write

$$L_{22}U_{22} = \left[L_{22}^{(i)}U_{22}^{(j)}\right].$$

Each of these relations is directly suitable for level 3 BLAS. Alternatively, if the arrays passed to the relevant subroutines are too large, it might be hoped that the software could partition the problem automatically.

Chapter 15 – Input/Output

1. (*a file handling routine*) The following routine

```
      subroutine header(action,chan)

      integer chan,nlines,i
      logical exist
      character*1 action
      character*80 lines(10)
      external exist

      save nlines,lines

      if (action .eq. 'r') then
         if (exist(chan,'header')) then
            read(chan,*) nlines
            if (nlines .le. 10) then
               read(chan,'(a80)') (lines(i),i=1,nlines)
            else
               write(0,*) 'header: too many lines'
            endif
         endif
      elseif (action .eq. 'w') then
         write(chan,*) 'header'
         write(chan,*) nlines
         write(chan,'(a80)') (lines(i),i=1,nlines)
      else
         write(0,*) 'header: action unknown'
         stop
      endif

      return
      end
```

may be used for copying comments between data files. The call **header(chan, 'read')** scans unit **chan** for the keyword 'header'. If it is found, **nlines** lines are read in and stored (using SAVE) in **lines**. A value for **nlines** is assumed after the keyword 'header':

```
header
4
An example of a set of comment lines to be
read and stored by the subroutine header.
```

```
Up to 10 lines are allowed, the number of lines
following the keyword 'header'.
```

Calling **header(chan,'write')** reverses this operation, writing the contents of **lines** to the channel **chan**. No check is made to see if these values have been defined. The logical function **exit** is described in the text. The statements **write(0,*)** are intended to write to standard error.

2. (*data visualisation*) To plot stepsizes as a function of t_i and i, the following two routines are suggested:

```
      program steps
      real last,new,h

      read(end=999,unit=*,fmt=*) last

 1    continue
      read(end=2,unit=*,fmt=*) new
      h = new - last
      write(*,*) new, h
      last = new
      goto 1

 2    continue
      stop "(steps, ended OK)"

 999  stop "(steps, no data)"
      end
```

and

```
      program order
      integer count
      real x,y

      count = 1

 1    continue
      read(end=2,unit=*,fmt=*) x, y
      write(*,*) count, y
      count = count + 1
      goto 1

 2    continue
      stop "(order, ended OK)"

      end
```

which could be compiled as the binaries **steps** and **order**. To produce a staircase type graph, the extra points can be inserted by the following tool:

```
      program scala
      real x,lasty,newy

      read(end=999,unit=*,fmt=*) x, lasty
      write(*,*) x, lasty

1     continue
      read(end=2,unit=*,fmt=*) x, newy
      write(*,*) x,lasty
      write(*,*) x,newy
      lasty = newy
      goto 1

2     continue
      stop "(scala, ended OK)"

999   stop "(scala, no data)"
      end
```

Compiling this as **scala**, a staircase graph of h_i against i could then be computed:

```
ode | steps | order | scala | image
```

where **image** is the graphics tool proposed in the text. As in the text, comments have been added after each STOP statement. These provide a trace of the pipeline's activity to standard error but are not standard FORTRAN.

3. (*multiple data sets*) Although there is only one standard input, to differentiate between different datasets, monotonicity may sometimes be used. Given two data sets **A** and **B**, both monotone, **A** being more accurate, constructs of the form

```
cat A B | compare | image
```

can be used where **compare** is a program which reads and stores values from **A**, before comparing them with those stored in **B**, possibly with interpolation, writing the results to standard output. The point where **A** ends and **B** begins can be checked for by monotonicity. Note, however, that the order of **A** and **B** should not normally be commuted,

```
cat B A | compare | image
```

resulting, typically, in a different graph.

C.2 Part II – Fortran90 and High Performance Fortran

Chapter 17 – Subroutines and Modules

Since IMPLICIT NONE is now permitted in Fortran 90, it is used in all of the following example solutions.

1. (*overloading*) Without using any special format statements, one possible module would be

```
module pretty_print

   interface xprint
      module procedure xprint_real_scalar, &
                       xprint_real_vector, &
                       xprint_real_2d_array
   end interface

contains

   subroutine xprint_real_scalar(x)
      implicit none
      real x
      write(*,*) x
   end subroutine xprint_real_scalar

   subroutine xprint_real_vector(x)
      implicit none
      real x(:)
      write(*,*) x
   end subroutine xprint_real_vector

   subroutine xprint_real_2d_array(x)
      implicit none
      integer row
      real x(:,:)

      do row=lbound(x,1),ubound(x,1)
         write(*,*) x(row,:)
      end do

   end subroutine xprint_real_2d_array

end module pretty_print
```

xprint could be extended to new types by adding further subroutines and entries in the interface block. Note how the use of LBOUND and UBOUND allows arrays

of the form, for example, **real, dimension(3:6,7:6)** to be output, the second range being handled automatically by the array sections. Once more, for a more refined array output format, see NAG routines **X04CAF** and **X04CBF** (exercise 4, chapter 3).

For a one dimensional array **x**, *the upper and lower bounds are returned simply by UBOUND(x) and LBOUND(x). For higher dimensional arrays,* **M**, *a syntax UBOUND(M,i) and LBOUND(M,i) should be used where UBOUND(M,i) returns, for example, the maximum extent (the upper bound) of the i-th dimension.*

2. (*keyword arguments*) Modules can contain not only subprograms, but also variables and constants. These can be used in conjunction with keywords. For example the module

```
module factory

  integer, parameter :: &
     go      =  1,  &
     stop    =  0,  &
     reverse = -1

  real, parameter :: &
     stationary =   0.0, &
     slow       =  10.0, &
     medium     =  50.0, &
     fast       = 100.0

contains

  subroutine machine(action,speed)

     implicit none
     integer, optional :: action
     real, optional :: speed

     write(*,*) '*** call to machine ***'

     if (present(action)) write(*,*) 'action = ',action
     if (present(speed))  write(*,*) 'speed  = ',speed

  end subroutine machine

end module factory
```

supports calls of the form

```
call machine(action=go)
call machine(action=go, speed=slow)
call machine(action=reverse, speed=fast)
```

```
...
```

following a **use factory** statement. Note that the routine **machine** here does nothing other than print out, if present, values from the parameter list. Additional safeguards against, say, using one of the above parameter constants for an incorrect parameter, could be incorporated by using user-defined derived types.

3. (*optional parameters*) To return mathematical constants, the following routine could be used.

```
subroutine constants(pi,e,euler)

   implicit none
   real, optional :: pi,e,euler

   if (present(pi))    pi =  3.14159265358979312
   if (present(euler)) euler =  0.577215664901532866
   if (present(e))     e =    2.71828182845904509

end subroutine constants
```

Typing

```
call constants(pi=x)
call constants(e=y,euler=z)
```

then sets `x`, `y` and `z` to π, Euler's constant, γ, and e respectively.

The use of PRESENT avoids the illegal assignment of dummy parameters which are not passed. A conversion program along the lines of

```
read(*,*) x
call convert(miles=x,km=y)
write(*,*)
```

however would not be possible because the *order* of the parameters is not known. A function based alternative however would be

```
real function convert(miles,kilometres)

   implicit none
   real, optional :: miles,kilometres

   if (present(miles)) then
      convert = 8./5. * miles
   elseif (present(kilometres)) then
      convert = 5./8. * kilometres
   else
      convert = 0.0
   endif

end function convert
```

one kilometre being approximately five-eights of a mile.

4. (*SUM and array valued functions*) Writing the logical array valued function

```
function ident(a)

   implicit none
   real, dimension(:,:) :: a
   integer :: k
   logical, dimension(size(a,1),size(a,2)) :: ident

   ident = .false.

   do k=1,min(size(a,1),size(a,2))
      ident(k,k) = .true.
   end do

end function ident
```

the sums and rows of the diagonal elements of a matrix A may be evaluated as

```
write(*,*) sum(pack(a,ident(a)))
write(*,*) product(pack(b,ident(b)))
```

where the size of **ident** is taken directly from the dimensions of **A**. Note, however, how this assumes that the lower bounds of **A** are both one. If **ident** is not defined in an internal routine or a module, the following interface block is required:

```
interface
   function ident(a)
      real, dimension(:,:) :: a
      logical, dimension(size(a,1),size(a,2)) :: ident
   end function ident
end interface
```

5. (*recursion*) The factorial $n!$ may be evaluated as follows:

```
recursive integer function factorial(n) result(answer)

   implicit none
   integer n

   if (n>1) then
      answer = n*factorial(n-1)
   else
      answer = n
   endif

end function factorial
```

6. (*recursion – a harder example*)

```
recursive function reverse_letters(string) result(answer)

   character(len=*) :: string
   character(len=len(string)) :: answer
   integer :: n

   n = len(string)

   if (n > 1) then
      answer = string(n:n)//reverse_letters(string(1:n-1))
   else
      answer = string
   endif

end function reverse_letters
```

Note, however, that this routine needs an explicit interface (i.e. one defined by an interface block or by being called as an internal routine) since the function's type (more specifically its length) is not defined until the function is called. Clearly, **reverse_letters** is another example of an inefficient recursive algorithm.

7. (*Ackermann's function*) One possible solution would be:

```
recursive integer function ackermann(m,n) result(answer)

   implicit none
   integer m,n

   if (m==0) then
      answer = n+1
   else
      if (n==0) then
         answer=acker(m-1,1)
      else
         answer=acker(m-1,acker(m,n-1))
      endif
   endif

end function ackermann
```

Note, however, that this function can also be evaluated non-recursively. For details, see exercise 5 in chapter 4.

Chapter 18 – Types and Pointers

1. (*modules and interfaces*) Assuming, again, lower bounds of one, the module

```
module vector_products

  interface operator(.inner.)
    module procedure inner_product
  end interface

  interface operator (.outer.)
    module procedure outer_product
  end interface

contains

  function inner_product(x,y)
    implicit none
    real inner_product
    real x(:),y(:)
    inner_product = dot_product(x,y)
  end function inner_product

  function outer_product(x,y)
    implicit none
    integer :: row,col
    real, dimension(:) :: x,y
    real, dimension(size(x),size(y)) :: outer_product

    do row=1,size(x)
      do col=1,size(y)
        outer_product(row,col) = x(row)*y(col)
      end do
    end do

  end function outer_product

end module vector_products
```

implements calls of the form

```
(/ 1.0, 2.0 /) .inner. (/ 4.0, 5.0 /)
(/ 2.0, 3.0 /) .outer. (/ 5.0, 7.0 /)
```

Note the use of prime numbers (2,3,5,7, etc.) to test the outer product. Restricting ourselves to such numbers it is always possible to reconstruct the original factors from any product. Hence

$$\boldsymbol{x}\boldsymbol{y}^T = \begin{pmatrix} 2 \\ 3 \end{pmatrix} \begin{pmatrix} 5 & 7 \end{pmatrix} = \begin{pmatrix} 10 & 14 \\ 15 & 21 \end{pmatrix}$$

was obviously computed as

$$\begin{pmatrix} 2\cdot 5 & 2\cdot 7 \\ 3\cdot 5 & 3\cdot 7 \end{pmatrix}$$

The prime number 1 has obviously no use in this respect.

2. (*derived types and user defined operators*) A simple module to implement the type **fractions** and the addition operator '+' could take the form

```
module rational

   type fraction
      integer :: top, bottom
   end type fraction

   interface operator(+)
      module procedure add, add_ir, add_ri
   end interface

contains

   function add(a,b)

      type (fraction) :: add,a,b

      add%top    = a%top * b%bottom + b%top * a%bottom
      add%bottom = a%bottom * b%bottom

      add = simplify(add)

   end function add


   function simplify(a)

      integer :: try

      type (fraction) :: simplify, a

      simplify = a

      do try = a%bottom,2,-1
         if ((modulo(simplify%top,try) == 0) .and. &
             (modulo(simplify%bottom,try) == 0)) then
            simplify%top = simplify%top/try
            simplify%bottom = simplify%bottom/try
         endif
```

```
        enddo

    end function simplify

end module rational
```

where the type definition **fraction** is made automatically available to **add** by host association. The remaining operators $-$, $*$ and $/$ may be defined in the same way using the corresponding additional functions and interface blocks. The function **simplify** merely cancels out common factors reducing fractions ka/kb to the standard form a/b. Expressions of the form (1.*r*.7) may be supported by a further function

```
function r(above,below)

    integer :: above, below
    type (fraction) :: r

    r%top = above
    r%bottom = below

end function r
```

with interface

```
interface operator(.r.)
    module procedure r
end interface
```

In expressions written with this operator

$$(1.\mathrm{r}.6) * (2.\mathrm{r}.7) - (1.\mathrm{r}.2)$$

brackets *are* required since the precedence of the user defined operators **.r.**, $+$, $-$, $*$ and $/$ is not defined[4]. Mixed expressions of the form

$$1 + (1.\mathrm{r}.2)$$

may be implemented with the aid of hybrid routines of the form

```
function add_ir(ia,b)

    integer :: ia
    type (fraction) :: add_ir,b

    add_ir = add(r(ia,1),b)

end function add_ir
```

[4] Or it is rather defined in terms of the precedence of the 'corresponding' intrinsic operators. That is $+$, $-$, $*$ and $/$ take precedence over .r..

```
function add_ri(a,ib)

   integer :: ib
   type (fraction) :: add_ri,a

   add_ri = add(a,r(ib,1))

end function add_ri
```

given the corresponding extensions to the relevant interface blocks:

```
interface operator(+)
   module procedure add, add_ir, add_ri
end interface
```

Note. When added security is required, operators may be declared public or private with the relevant PUBLIC and PRIVATE statements. For example, if the user is required only to see the type **fraction**, *the operator* **.r.** *and the four arithmetical operators* $+$, $-$, $*$ *and* $/$ *defined for this type, the statements*

```
PRIVATE
PUBLIC :: fraction, operator(.r.), operator(+), &
          operator(-), operator(*), operator(/)
```

may be used. For details of the use of PRIVATE and PUBLIC with operators and assignments, see [21].

3. (*pointers and linked lists*) Replacing normal '=' assignments with pointer '=>' assignments, **follow_list** may be rewritten:

```
subroutine follow_list(start)

   implicit none

   type(pass_the_buck), target  :: start
   type(pass_the_buck), pointer :: node

   node => start
1     continue
      write(*,*) node%who
      if (associated(node%next)) then
         node => node%next
         goto 1  ! a repeater
      else
         write(*,*) '*end*'
      endif

end subroutine follow_list
```

The following program will read the items from standard input and append them to a list:

```
      subroutine add_to_list(start)

      implicit none

      type(pass_the_buck), target  :: start
      type(pass_the_buck), pointer :: node
      character(len=20) name

      node => start
1     continue
        if (associated(node%next)) then
            node => node%next
            goto 1  ! a repeater
        else
3           continue
            read(*,'(a20)',end=2) name
            allocate(node%next)
            node => node%next
            node%who = name
            nullify(node%next)
            goto 3  ! a repeater
        endif

2     continue  ! a completer

      end subroutine add_to_list
```

The first loop (coded with the **goto 1** statement) follows the list to its end before the second (**goto 3**) adds the extra items. Note the use of the NULLIFY statement to ensure the pointer from the newly created node is initially *unassociated*. This is necessary because its default association status is *undefined*. Care must also be taken when initialising a list for use with this routine: if the free pointer is undefined, unpredictable results may occur.

4. (*a block algorithm*) The solution to exercise 3, chapter 6, now reduces to

```
      T = 0.0

      call mult2(R(1:2,1:2),S(1:2,1:2),T(1:2,1:2))
      call mult2(R(1:2,3:4),S(3:4,1:2),T(1:2,1:2))

      call mult2(R(3:4,1:2),S(1:2,1:2),T(3:4,1:2))
      call mult2(R(3:4,3:4),S(3:4,1:2),T(3:4,1:2))

      call mult2(R(1:2,1:2),S(1:2,3:4),T(1:2,3:4))
```

```
call mult2(R(1:2,3:4),S(3:4,3:4),T(1:2,3:4))

call mult2(R(3:4,1:2),S(1:2,3:4),T(3:4,3:4))
call mult2(R(3:4,3:4),S(3:4,3:4),T(3:4,3:4))
```

where strides need no longer be passed. Alternatively, using the intrinsic function MATMUL, **T(1:2,1:2)** for example could be calculated:

```
T(1:2,1:2) = matmul(R(1:2,1:2),S(1:2,1:2)) +
      &      matmul(R(1:2,3:4),S(3:4,1:2))
```

The advantages of block algorithms were outlined in section 10.1.2.

Chapter 19 – Arrays

1. (*SUM and array constructors*) (a) The norms $||A||_1$ and $||A||_\infty$ are returned by calls of the form

```
maxval(abs(sum(a,dim=1)))
maxval(abs(sum(a,dim=2)))
```

respectively whereas the truncated sum for (b) is given by

```
sum( &
     array = (/ (1./(i*i), i=1,100) /), &
     mask = (/ (1./(i*i), i=1,100) /) > .001 &
   )
```

since $1/100^2 < 1/1000$. The sum

$$\sum_{i=1}^{\infty} \frac{1}{n^2}$$

is, of course, convergent ('defined'), having the value $\pi^2/6$. For further details see [26].

2. (*array subscripts*) The relevant powers of p may be generated by the following code:

```
program permutation

   implicit none
   integer :: size,max_power,power,i
   integer, dimension(:), allocatable :: perm,map

   write(*,*) 'size and maximum power:'

   read(*,*) size,max_power
   allocate(perm(size),map(size))
```

```
      write(*,*) 'the permutation'
      read(*,*) perm

      map = (/ (i,i=1,size) /)

      do power=1,max_power
         map = perm(map)
         write(*,*) power,' : ',map
      end do

   end program permutation
```

which gives the results $\{p^r : r = 1, \text{max\_power}\}$. Applied to $(2,3,1,4,6,5)$ for **max_power** $= 9$, the result is

```
1  :  2 3 1 4 6 5
2  :  3 1 2 4 5 6
3  :  1 2 3 4 6 5
4  :  2 3 1 4 5 6
5  :  3 1 2 4 6 5
6  :  1 2 3 4 5 6
7  :  2 3 1 4 6 5
8  :  3 1 2 4 5 6
9  :  1 2 3 4 6 5
```

Note the cycles in the elements $\{1,2,3\}$ and $\{5,6\}$: they cycle

$$\begin{array}{ll} 1 \mapsto 2 & \\ 2 \mapsto 3 & 5 \mapsto 6 \\ 3 \mapsto 1 & 6 \mapsto 5 \end{array}$$

without affecting the other elements. We can thus write p as the product $(2,3,1,4,5,6)(1,2,3,4,6,5)$. Note further that p^6 is the identity permutation – p has *order* 6.

3. (*array constructors and subscripts*) The standard solution to this problem would be

```
A( (/ (i,i=n1,n2) /), (/ (i,i=m1,m2) /) )
```

but, accessing as a one dimensional array (and provoking the corresponding array out-of-bound exceptions) we could also write

```
A( (/ ((1+(i-1)+4*(j-1), i=n1,n2), j=m1,m2) /), 1)
```

and reshape the result with the RESHAPE function.

4. (*array constructors*) Defining **x** as a real allocatable array of rank one, we could write

```
read(*,*) N
allocate(x(0:N-1))
x = 1./N * (/ (i,i=0,N-1) /)
result = 1./N * sum(x**2)
```

where **N** is an integer. More general integration formulae could be implemented as **sum(c*x**2)** for appropriate choices of **c** and differing meshes **x**. Any valid expression may be substituted for x^2, although if user defined functions are used, they must be provided with an interface to support elemental evaluation.

References

1 **Anderson, E., Bai Z., Bischof, C., Demmel, J., Dongarra, J., Greenbaum, A., Ostrouchov, S. and Sorenson, D.** (1995) LAPACK users' guide, Second Edition. *SIAM, Philadelphia.*[1]

2 **Brassard, G. and Bratley, P.** (1988) Algorithmics: Theory and Practice. *Prentice-Hall, New Jersey.*

3 **Boucher, M. and Priest, D.** Megaflop rates for level 1, 2 and 3 BLAS. *(1995) private correspondence, Dakota Scientific Software Inc., Rapid City, South Dakota.*

4 **Brown, W. S.** (1981) A simple but realistic Model of Floating Point Computation. *ACM Trans.* 7 *pp445-480.*

5 **Coleman, F. and Van Loan, C.** (1988) Handbook for Matrix Computations. *SIAM, Philadelphia.*

6 **Crow, J. A.** Megaflop rates for level 1, 2 and 3 BLAS. *(1994) private correspondence, Cray Research, Eagan, MN.*

7 **Dijkstra, E.W.** (1968) Go To Statement considered Harmful. *Communications of the ACM,* **11** *3 pp147-148.*

8 **Dongarra, J. J. and Grosse, E.** (1987) Distribution of mathematical software via electronic mail. *Communications of the ACM,* **30** *pp403-407.*

9 **Dongarra, J. J., Duff, I. S., Sorensen, D. C. and Van der Vorst, H. A.** (1991) Solving Linear Systems on Vector Shared Memory Computers. *SIAM, Philadephia.*

10 **Filippone, S.** Megaflop rates for level 1, 2 and 3 BLAS on the IBM System/6000. *(1994) private correspondence.*

11 **Gilster, S.** (1994) Finding it on the Internet. *John Wiley & Sons, Chichester, UK.*

12 **Goldberg, D.** (1991) What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys* **23** *No. 1, pp5-48.*

13 **Golub, G. E. and Van Loan, C. F.** (1989) Matrix Computations. *2nd ed., Johns Hopkins University Press, Baltimore.*

14 **Hennessy, J. L. and Patterson, D. A.** (1990) Computer Architecture: a Quantitative Approach. *Morgan Kaufmann, California.*

15 **Kerrigan, J. F.** (1993) Migrating to Fortran 90. *O'Reilly & Associates, California.*

16 **Knuth, D. E.** (1974) Structured Programming with *go to* statements. *Computing Surveys* **6** *No. 4 pp261-301.*

17 **Krol, E.** (1992) The Whole Internet User's Guide & Catalog. *O'Reilly & Associates, California.*

18 **Ledgard, H.** (1983) ADA – an introduction. *2nd edition, Springer Verlag, Heidelberg.*

19 **Metcalf, M.** (1985) Effective FORTRAN 77. *Oxford University Press, Oxford.*

20 **Metcalf, M. and Reid, J.** (1987) Fortran 8x explained. *Oxford University Press, Oxford.*

21 **Metcalf, M. and Reid, J.** (1990) Fortran 90 explained. *Oxford University Press, Oxford.*

22 **Phillips, J.** (1985) The NAG Library – a Beginners Guide. *Oxford University Press, Oxford.*

23 **Pollicini, A. A., ed.** (1989) Using Toolpack Software Tools. *pp 235-249, Kluwer, Dordrecht.*

[1] Currently also available under *mosaic* at *www.netlib.org.*

24 **Saad, Y.** (1990) SPARSKIT: a Basic Tool for Sparse Matrix Computations. *Technical report CSRD TR 1029, University of Illinois, Urbana, Illinois.*

25 **Strassen, V.** (1969) Gaussian Elimination is Not Optimal. *Numer. Math.* **13** *pp354-356.*

26 **Tall, D. O.** (1977) Functions of a complex variable. *Routledge & Kegan Paul, London.*

27 **Weiss, S. and Smith J. E.** (1994) Power and PowerPC. *Morgan Kaufmann, California.*

28 **Welsh, J. and Elder, J.** (1987) Introduction to Modula-2. *International Series in Computer Science, Prentice Hall, New Jersey.*

29 **Winter, D. T.** (1987) Efficient use of memory and input/ouput. In *Algorithms and Applications on Vector and Parallel Computers, ed. J. J. te Riele, T. J. Dekker and H.A. van der Vorst, North-Holland, Amsterdam.*

30 **User's Guide for IBM AIX XL FORTRAN Compiler/6000 Version 2.3.** *(1992) SC09-1354, IBM.*

31 **CM Fortran Programming Guide, version 1.0.** *(1991) Thinking Machines Corporation, Cambridge, Massachussetts.*

32 **ESSL Guide and Reference** *(1994) SC23-0526, IBM.*

33 **Getting started in CM Fortran.** *(1991) Thinking Machines Corporation, Cambridge, Massachussetts.*

34 **High Performance Fortran – Language Specification, version 1.0.** *Report of the High Performance Fortran Forum, appearing in Scientific Programming,* **2**, *No. 1, 1993.*[2]

35 **Harwell Subroutine Library: Specification (release 11).** *(1994) AEA Technology, 424.4 Harwell, Didcot, England.*

36 **MATH/LIBRARY – Fortran routines for mathematical applications.** *(1987) MALB-USM-PERFECT-1.0, IMSL, Houston.*

37 **NAG FORTRAN Library Manual – mark 16.** *(1994) The Numerical Algorithms Group Ltd., Oxford.*

38 **PVM 3 – user's guide and reference manual.** *(1994) Oak Ridge National Laboratory, Tennessee.*

39 **VS FORTRAN version 2 – Programming Guide, release 4.** *(1991) SC26-4222-04, IBM.*

[2] This document is also obtainable over *netlib* using the command *send hpf-v10-final.ps from hpff* or other *Internet* connections.

Index

Where topics are raised in the exercises, the reference is given to the solutions in brackets after the index entry.

Ackermann's function
  non recursive code, (199–201)
  recursive code, (218)
ACMS TOMS, 15
ADVANCE, 161
ALIGN, in HPF, 167–168
ALL, 146
ALLOCATE
  with arrays, 143
  with pointers, 138
alternate RETURN, 117
archie, 15
arithmetic IF
  in FORTRAN 77, 90
  status in Fortran 90, 117
array(s)
  addressing model, 77
  arithmetic in Fortran 90, 145–146
  automatic, 143
  calling by address, 40–42
  column-wise storage, 39
  constructors, 147–149
  dummy dimensions, 42–43
  higher dimensional, 42–43, (203)
  in Fortran 90
    chapter on, 143–152
  in FORTRAN 77
    chapter on, 39–44
  indices, 149
  leading dimension, 42
  mask arrays, 158
  multiplication
    in Fortran 90, 145
    Strassen, (208–209)
    with constructors, 150–151
    with MATMUL, 151
  passing subarrays, 43–44
  row-wise storage, 39
  sections, 149–151
  skew dimensions, 75–77
  valued functions, 128, (217)
  work arrays
    partitioning storage, 47–49
    the need for, 47
ASSIGN, 117
ASSOCIATED, 141
associativity
  not for floating point, (189)
attributes
  dimension, 143
  optional, 129
  parameter, 119
automatic arrays, 143

bitwise storage, in Fortran 90, 134
BLAS
  and parallel computing, 66
  and sparse matrices, 66
  and stride lengths, 59
  comparative timings, 63
  introduction to, 57–58
  level 1
    argument against, 70–71
    chapter on, 57–62
    examples, 58–62
  level 2 and 3
    need for, 63–64
    operation counts, 63, (208–209)
    section on, 63–66
  negative strides, 59–61, (207)
  operation counts, (208–209)
  relation to LAPACK, 81
  routine listings, 173–177
  zero strides, 59
block algorithms
  Cholesky factorisation, (209)
  left looking factorisation, (209–210)
  multiplication, (204), (223–224)
  practical implementation, (210–211)
  right looking factorisation, 64–66
  storage access, (210)
  the need for, 53

BLOCK DATA, 118

C05ADF
  an example program, 19–20
  subroutine reference, 22
C05AZF
  an example program, 20–21
  subroutine reference, 22
cache memory
  alternatives to, 66
  and the BLAS, 63–66
  section on, 52
characters
  ASCII codes, 157
  collating sequence, 157
  comparisons of, 157
  efficiency of, 97
  non-standard, 134
  string assignments, 37
Cholesky
  right looking, (209)
column-wise storage, 39
command chaining, 69
command pipelining, 69
COMMON
  and data modules, 126
  efficiency of, 98
  replacement in Fortran 90, 126
  status in Fortran 90, 118
communication
  examples in HPF, 167
comparisons
  and Fortran 90 arrays, 146
  between reals, (187–188)
completers
  and other languages, 89
  in Fortran 90, 156
complex numbers
  in Fortran 90, 134
computed GOTO, 90
  status in Fortran 90, 118
Connection Machine Fortran, 118
constants
  in Fortran 90, 134–135
  overwriting, 92
control structures
  general comments, 88–91
  in Fortran 90, 155–159
COUNT, 146
CYCLE, 156

D01AHF
  an example program, 16–19
  subroutine reference, 21–22
data locality, 53
data modules, 126
data pipelining, 67–69
data reuse, 54, (207)
DDOT, 61
DEALLOCATE
  and pointers, 139
  with arrays, 144
denormalised numbers, (189)
DGER, subroutine reference, 78
DIMENSION
  Fortran 90 attribute, 143
  statement in Fortran 90, 118
DISTRIBUTE
  HPF directive, 166
DO WHILE
  alternative in Fortran 90, 156
  general comments, 90
DO-END DO, 91
DO-loops
  and CONTINUE, 91
  and independence in HPF, 170
  DO-END DO, 91
  in Fortran 90, 155–156
  invariant code, 99–100
  loop collapse, 102–103
  loop fusion, 101
  nesting, 100
  optimisation of, 99–103
  scope of variables, 91
  unrolling of, 100
  with reals, (187–188)
    in Fortran 90, 117, 147
DOUBLE PRECISION
  in FORTRAN 77 3–5
  replacement in Fortran 90, 133
  status in Fortran 90, 118
DSCAL, subroutine reference, 78
dynamic storage allocation
  and FORTRAN 77, 25–27
  arrays in Fortran 90, 143–145
  disadvantages of, 27

e-lib, Berlin, 15
elemental operations, 145–146
ENTRY, considered harmful, 90
EQUIVALENCE
  possible application, 99
  status in Fortran 90, 118
error handling
  IFAIL in NAG, 18, (194)
ESSL
  and other libraries, 14
  with XLF preprocessor, 95
EXIT, in Fortran 90, 156
external routines, 122
extrinsic routines, in HPF, 170–171

Fibonacci series, 130
file handling, *see* Input/Output
FORALL, in HPF, 168–170
Fortran 8x, 118
FORTRAN 77
  relation to FORTRAN 66, 117
Fortran 90
  and internal files, 163
  and the BLAS, 151
  arrays, 143–152
  control structures, 155–159
  deprecated features, 117
  derived types, 135–137
  dynamic storage, 137, 143–145
  free form input, 118–119
  Input/Output, chapter on, 161–163
  interfaces, 126–130
  internal routines, 121–122
  intrinsic functions, 152
  intrinsic types, 133–135, 137–141
  introduction to, 117–119
  modules, 122–126
  new format editors, 163
  obsolescent features, 117
  other dialects, 118
  recursion, 130
  relation to previous dialects, 117–118
FORTVS2 and SAVE, 28
free-form input, 118–119
functional units, 67

Gaussian elimination
  chapter on, 73–78
  for band matrices, 75–77
  for full matrices, 73–75
generic functions
  user defined, 127–128
gopher, 15
GOTO
  considered harmful, 88–90
  onto and computed, 90
  use in FORTRAN, 89–90

HARWELL, 14
High Performance Fortran, 118
  and parallel computing, 165
  chapter on, 165–171
  HPF_LIBRARY, 170
host association, 122

IEEE arithmetic, (188–189)
  and machine precision, (187)
IF-statements
  arithmetic IF
    in FORTRAN 77, 90
    status in Fortran 90, 117
  efficient nesting of, 99
  in Fortran 90, 155
IFAIL
  introduction, 18
  noisy exit, (194)
IMPLICIT LOGICAL, 8
IMPLICIT NONE
  in Fortran 90, 119
  in FORTRAN 77 dialects, 8
IMSL, 14
in-line expansion, 94–95
INDEPENDENT, in HPF, 170
initialisation
  DATA statements, 98
  the need for, 29
Input/Output
  and NAG routines, (194)
  and portability, 10
  chapter on, 105–112
  efficiency, 98–99
  Fortran model, 161
  generic system, 105–108
  in Fortran 90, chapter on, 161–163
  in PASCAL and C, 161
  internal files, 35–37
    in Fortran 90, 163
  non advancing, 161
  UNIX pipelines, 108–112
INQUIRE, 10
INTENT, 128–129
intercompilation analysis, 128
interfaces
  and modules, 128
  operator overloading, 136–137
  overloading, 127–128
  section on, 126–130
intermediate memory, 51–52
internal files
  chapter on, 35–37
  in Fortran 90, 163
  simple parsing, (201–202)
internal routines, 121–122
internet, 15
intrinsic functions in Fortran 90
  with arrays, 146–149
intrinsic routines
  in Fortran 90
    list of, 179–184
intrinsic types, 133–135

keyword parameters, 129–130
  and symbolic constants, (215–216)
KIND
  for non-reals, 134
  for reals, 133–135
  short notation, 135

labels, in Fortran 90, 155–156
LAPACK
  auxiliary routines, 81, 85
  band storage, 84
  chapter on, 81–85
  computational routines, 81, 84–85
  driver routines, 81–84, 86
  relation to NAG and BLAS, 81
  relation to other libraries, 14
  supported matrix types, 83
largest integer, 103, (194)
left looking factorisation, (209–210)
LEN, 97
linked lists, 139, (222–223)
linked triad, 69
LINPACK
  relation to other libraries, 14
Lorenz's equations, (191, 192)

machine precision
  a numerical experiment, 3
  and IEEE, (187)
  and portability, 4–5, 8
  model in Fortran 90, 134
  NAG routine, (194)
  representation, (187)
make, 7
mask arrays, 158
memory hierarchy, 51–52
memory management
  chapter on, 51–55
  the need for, 51
MIL-STD 1753, 10–183
military standard, US, *see* MIL-STD 1753
Modula-2
  co-procedures, 20
MODULE PROCEDURE, 128
modules, 122–126
  and internal routines, 123
  controlling access, 123–126
  in other languages, 88
  placement of, 122, 124
mosaic, 15

NAG
  chapter on, 13–22
  mark 16 listings, 17
  relation to LAPACK, 81
NAMELIST, 162
negative strides, 59–61, 207
Netlib
  access to, 15
  the PORT library, 10
NULLIFY, (223)
numerical integration
  an example, 16–19
  further example, (192–193)

operators
  infix, 137
  overloading, 136–137
  precedence, (221)
optimisation
  chapter on, 93–103
optional parameters, 129–130
  further examples, (216–217)
overloading
  and modules, 128
  further example, (214–215)
  of functions, 127–128
  of operators, 136–137

page faults, *see* paging
paging
  an example, (205–207)
  discussion, 54
parallel computing
  and HPF, 165
  and portability, 11
  and the BLAS, 66
parameters
  attributes, 119
  keywords, 129–130
  optional, 129–130
  overwriting, 92
parsing
  with internal files, (201–202)
PASCAL
  and subroutine calls, 26
  arrays in, 39
  Input/Output, 161
  storage allocation in, 25
pipelines
  command pipelining, 69
  UNIX pipelines, 108–112
  vector pipelines, 67–69
pointers
  disadvantages of, 141
  in Fortran 90, 137–141
polynomial evaluation, 93
portability
  and Fortran 90, 134
  and file handling, 10
  and machine precision, 4–5
  chapter on, 7–11
pre-processing, 93
PRESENT, 130
pretty writers
  discussion of, 8
  sample output from, 9
prime numbers
  for testing products, (219–220)

privacy
  and operators, (222)
  and modules, 123–126
PRIVATE, 125
PROCESSORS
  HPF directive, 166
PRODUCT, 147
program substitution, 95–96
PUBLIC, 125

QUADRUPLE PRECISION
  in FORTRAN 77 dialects, 4

R1MACH, 10
random permutations, (194–195)
REAL
  in Fortran 90, 133–135
  in FORTRAN 77, 3–5
REAL*4, REAL*8, 4
recursion
  n!, (217–218)
  Ackermann's function
    in Fortran 90, (218)
    non recursive code, (199–201)
  in Fortran 90, 130
  not in FORTRAN 77, 26
  string reversal, (218)
RECURSIVE, 130
REPEAT UNTIL
  alternative in Fortran 90, 156
RESHAPE, 148–149
RESULT, 130
RETURN, use of, 90
reverse communication
  advantage, 32
  discussion, 29–32
  further example, (198–199)
  with C05AZF, 20–21
REWIND, 10
  rewind status in Fortran 90, 163
right looking factorisation, 64
  Cholesky variant, (209)
  storage access, (210)
Risc machines, 69
root finding
  Newton's method, (198–199)
  two examples, 19–21
routines
  external, 122
  extrinsic, 170–171
  internal, 121–122
  modules, 122–126
  recursive in Fortran 90, 130
row-wise storage, 39

SAVE
  and FORTVS2, 28
  as the default, 28
  dangers with, 28
  definition and use, 27–29
  efficiency of, 28–29
SAXPY
  command chaining, 69
sccs, 11
SCOPY, example use, 58
screen input, 111
SELECT CASE, 157–158
SELECTED_INTEGER_KIND, 134
SELECTED_REAL_KIND, 133
sorting, (193–194)
sparse matrices, 66
spatial locality
  an example, 53
  definition, 53
standard error, 109
statement functions, 94
  and internal routines, 121
  in FORTRAN 77, 93
static storage allocation, 25–26
Strassen multiplication, (208–209)
stride lengths
  and the BLAS, 59
  column and row access, 61–62
  definition, 54
strong typing, 8
  need for, 87–88
subroutine calls, and storage, 26
SUM, 147
superscalar processors, 66

targets, of Fortran 90 pointers, 137
tee, 109
temporal locality, 53
types
  derived, 135–137
  in Fortran 90
    chapter on, 133–141
  intrinsic, 133–135
  pointers, 137–141
  type conversion, 97
  type promotion
    of integers, 10
    of reals, 4, 8

underflow
  gradual underflow, (189)
  special algorithms, (189–190)
unformatted files, 98
uninitialised variables, 29
UNIX
  pipelines, 108–112
US military standard, *see* MIL-STD 1753

USE
  normal use, 123
  with ONLY, 125

variable names
  in Fortran 90, 119
variable precision in Fortran 90
  absence of, 134
vectorisation
  and portability, 11
  compiler directives, 70
  costs and gains, 70
  section on, 67–70
  two analogies, 69
  vector registers, 54
virtual memory, 54

weak typing
  dangers of, 87–88
  in Fortran, 8
what, 11
WHERE
  in Fortran 90, 158–159
  relation to FORALL, 169
work arrays
  chapter on, 47–49
  partitioning storage, 47–49
  the need for, 47