a course on PROGRAMMING in FORTRAN

REVISED TO INCORPORATE **FORTRAN 77**

SECOND EDITION

JALERIE J. CALDERBANK

A Course on Programming in FORTRAN

Revised to Incorporate

FORTRAN 77

Second Edition

Valerie J. Calderbank
Business Development and Computer Division
UKAEA, Culham Laboratory

LONDON NEW YORK
CHAPMAN AND HALL

First published 1969 by
Chapman and Hall Ltd
11 New Fetter Lane, London EC4P 4EE
Published in the USA by
Chapman and Hall
733 Third Avenue, New York NY 10017
Second edition 1983

Printed in Great Britain at
The University Press, Cambridge

ISBN 0 412 24270 2 (cased)
ISBN 0 412 23790 3 (Science Paperback)

British Library Cataloguing in Publication Data

Calderbank, Valerie J.
A course on programming in FORTRAN. – 2nd ed.
1. FORTRAN (Computer program language)
I. Title
001.64'24 QA76.73.F25

ISBN 0-412-24270-2
ISBN 0-412-23790-3 Pbk

Library of Congress Cataloging in Publication Data

Calderbank, Valerie Joyce.

A course on programming in FORTRAN.
(Science paperbacks; 65)
Rev. ed. of: A course on programming in FORTRAN IV. 1969.
Includes index.
1. FORTRAN (Computer program languge)
2. Electronic digital computers – Programming.
I. Title. II. Series.
QA76.73.F25C35 1983 001.64'24 82-4569
AACR2

ISBN 0-412-24270-2
ISBN 0-412-23790-3 (pbk.)

Preface

During the twelve years since this book first appeared, the success of FORTRAN as a programming language has continued unabated. An enhanced definition of the language, produced by the American National Standards Institute, has become widely known as ANSI 77 FORTRAN or FORTRAN 77 (see ANSI X3.9, 1978).

FORTRAN 77 compilers are now becoming generally available from most of the major computer manufacturers and interest in the language is increasing. It seems an appropriate time, therefore, to revise the original text of my book in order to describe the new facilities.

People interested in FORTRAN 77 fall into one of two categories – those interested in learning it as a new language in its own right and those interested in converting from FORTRAN 66. This very much affects the way in which it is taught.

In order to cater for the second group, I have revised the text of my original book to describe ANSI 66 FORTRAN with the ANSI 77 enhancements introduced separately, as they arise naturally. I hope that by describing the two dialects side by side, I have provided a text which will be useful to many programmers and lecturers in both of the above categories. The two dialects can be distinguished easily in the text by the fact that they are printed in different type founts. In line with the trend in computing generally, I have introduced more examples and exercises of use in non-numerical programming applications.

I owe a continuing debt of gratitude to Professor E. J. Burge, Head of the Physics Department at Chelsea College, London, for the encouragement he gave me during my early career, and for the confidence he always displayed during the period when the original text was written. He suggested numerous improvements to the text of the original edition, and has continued to give helpful advice on the content of this second edition.

My thanks also go to Dr A. Howarth who, as an experienced lecturer in FORTRAN at North Staffordshire Polytechnic, has made many useful comments on the text, and to my colleague Dr P Kirby (of the Theoretical Physics Division, Culham Laboratory) for reading and commenting on the text so thoroughly, and for the many useful contributions he has made to programming standards at Culham.

I wish to thank my husband, Dr M. Calderbank, of the Business Development and Computer Division at Culham, not only for helping me with the text and the exercises, but also for giving me the constant support without which this book would never have appeared.

My acknowledgements would not be complete, however, without thanking Chapman and Hall for the vast amount of work they have had to put into the production of both editions, and of course my employers (UKAEA, Culham Laboratory) for giving their permission to write this book, and for providing computer time on their PRIME 500 computers without which I would have been unable to provide so many exercises and solutions.

Finally, I would like to point out that, after some deliberation, I decided to keep in this edition the two complete examples from the original text. The first may be found in Appendix Two and is a least-squares curve-fitting program which uses the Gaussian Elimination Method (originally described in R. W. Hamming, *Numerical Methods for Scientists and Engineers*, p. 360, McGraw-Hill, 1962). It is in ANSI 66 FORTRAN. The second may be found in Appendix Three and is a Kutta–Merson numerical integration program (originally described in L. Fox, *Numerical Solution of Ordinary and Partial Differential Equations*, p. 24, Pergamon Press, 1962). This has been translated into ANSI 77 FORTRAN.

I recognize that these examples will be beyond the comprehension of the average reader and may well be too out-of-date to be of use to the professional numerical analyst. I use them only as illustrations of complete FORTRAN programs which should be of interest to beginners and experts alike.

Valerie J. Calderbank
Business Development and Computer Division
Culham Laboratory
1982

Contents

1 Fundamentals of FORTRAN

1.1 Introduction

The purpose of this book is to teach the reader to program a computer in one particular language. It is not intended to discuss the way in which a modern computer is constructed or operated. However, a certain minimum knowledge of the structure of a computer system is required before any attempt can be made to program it, and therefore a brief introduction is given here.

Computer architecture can vary greatly in detailed design but the basic principle of most systems can be represented by the diagram in Fig. 1.1. Information is presented to the computer via an *input device* and stored in the computer's central memory. The memory consists of binary digits or *bits* which are grouped together into larger units called *bytes* (typically 8 bits) and *words* (typically 16, 32, 48 or 64 bits). The position of a word or byte in memory is known as its *address*.

Calculations are performed in the *Arithmetic Unit* which contains one or more *accumulators* or high-speed working registers. The *control unit* controls and coordinates the sequence of operations within the computer. It is able to access and decode the instructions held in the memory and initiate the appropriate action. The results of a computation are transferred to an *output device*. The arithmetic unit, control unit, registers and central memory together form the *central processor unit* (CPU). To this are connected input/output devices and other cheaper (but slower) storage devices known as *secondary store* or backing store. These devices are collectively known as *peripherals*.

A typical modern scientific computer installation will provide a variety of input devices such as teletypes, visual display units (VDUs), paper-tape and card readers; a variety of output devices such as lineprinters and graph plotters, and a variety of secondary storage devices such as magnetic discs, drums and tapes. During the last ten years, cards and paper tapes, as input media, have been

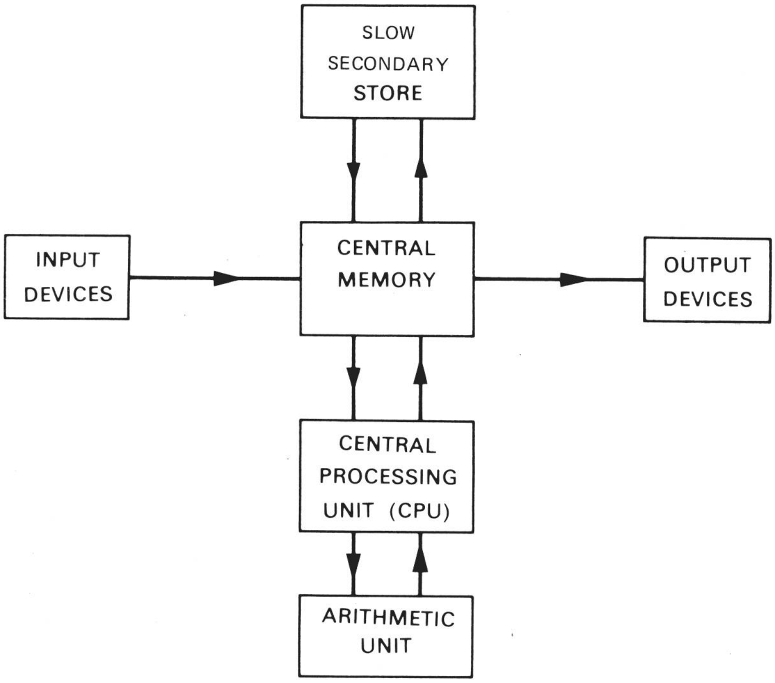

Figure 1.1 A schematic diagram showing the flow of information through a typical computer system.

superseded by on-line devices such as teletypes and VDUs which are connected directly to the computer and enable the user to type information on the keyboard into files on disc. From disc, the information may be transferred to central memory when required. Similarly, output may also be to a disc file from where it may be printed by a lineprinter or displayed on a VDU. Input and output are often autonomous processes carried out in parallel with computation in the CPU and place no load on it.

A computer is able to produce a solution to a particular problem only if it is presented with a series of simple instructions that it is able to perform and which will, when obeyed in a specific order, produce the desired result. This sequence of instructions is referred to as a *program*, and it is the responsibility of the human programmer to present the problem to the computer in this rigid form. Programs are collectively termed computer *software*. The *hardware* of any particular computer is designed to obey a limited number of basic instructions such as addition, subtraction, multiplication, division and so on. Therefore a sophisticated mathematical computation can be performed on a computer only if the computation is capable of being broken down into a logical sequence of these basic operations. Many numerical methods are concerned with the reduction of problems, such as integration, minimization, the solution of equations and so on, to this simple numerical form. Any process described in this way is generally known as an *algorithm*. It is the programmer's first task, therefore, to produce a suitable algorithm for a particular problem. A typical algorithm to find the sum of the squares of the integer numbers 1 to 100 might read as follows:

Set the value of SUM equal to zero.
Set the value of I equal to 1.

Start: If the value of I is greater than 100, end the process and print the value of SUM.

Otherwise square I and add the result to SUM.
Add 1 to I and return to the line labelled Start to continue the process.

Alternatively (or in addition), the algorithm can be produced in a diagrammatic form known as a *Flow Chart*. A flow chart for the above algorithm is shown in Fig. 1.2. Note that certain conventions are generally followed in flow charts. All commands (e.g. add 1 to the value of I) are placed in rectangular boxes. Questions to be asked are placed in diamond-shaped boxes. These are generally referred to as *decision boxes*, since a decision has to be made at this point as to whether the answer to the question is *yes* or *no*. Thus there are always at least two ways out of any decision box, one to a series of instructions to be performed if the answer is *yes*, and one to another series if the answer is *no*. Input or output instructions (e.g. print the value of SUM) are usually placed in rounded boxes. Arrows indicate the flow of the logic from box to box.

So far the programmer has been concerned only with crystallizing ideas and formulating the problem in a suitable way. Now the program must be produced – that is, a sequence of instructions must be written in a language which the computer can understand. Any computer is constructed in such a way that it can fundamentally understand only one language – the *machine language* of that computer. Since programming in this basic machine language is a tedious and error-prone process, a multitude of so-called *high-level languages* have been designed to ease the programmer's task. A program written in one of these languages cannot be understood directly by the computer, and therefore it must be translated from the high-level language into the machine language of the computer. This task is performed by a program resident in the computer and known as the *compiler* (or *translator*). The compiler takes a *source* program in the high-level language and produces a logically equivalent *object* program in the machine code. This process is known as compiling the program, and obeying the resulting sequence of machine code instructions is known as *executing* (or running) the program. From this it can be appreciated that any program written in a high-level language cannot be run on a computer which does not have a compiler for the particular language or version of that language.

This book describes the grammatical rules for writing programs in one particular

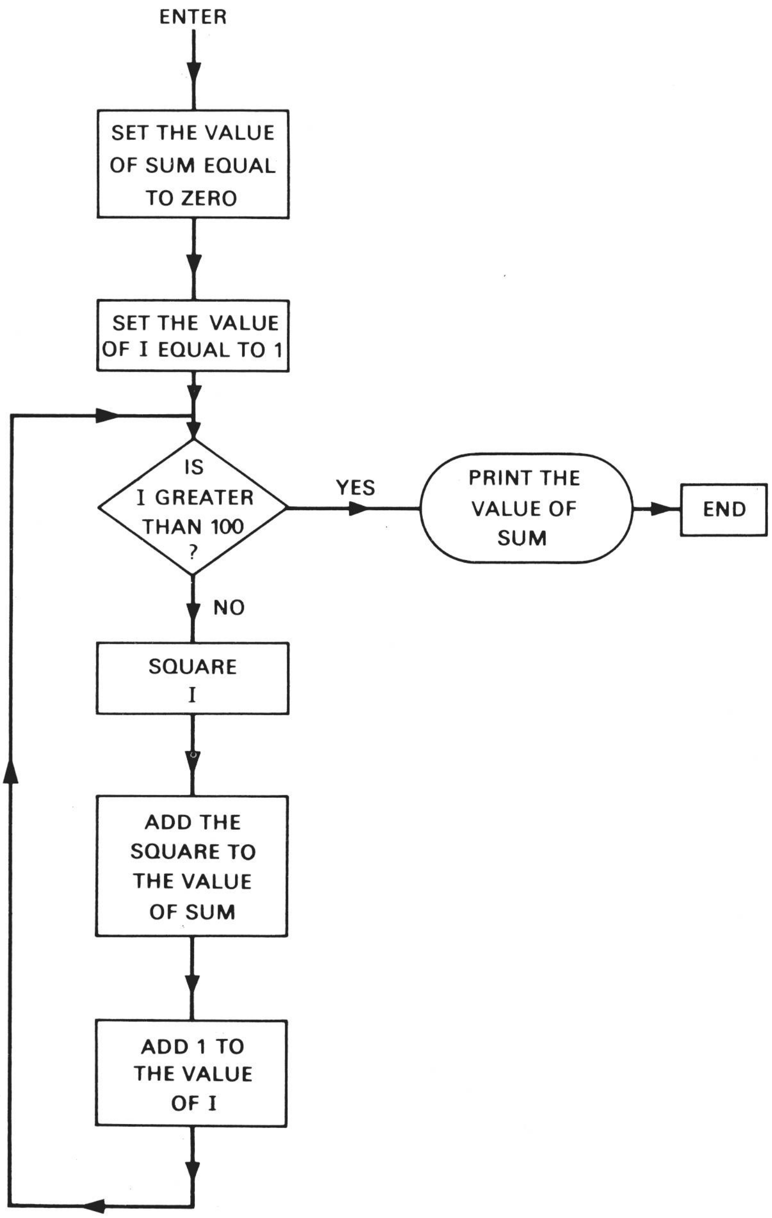
ENTER
SET THE VALUE OF SUM EQUAL TO ZERO
SET THE VALUE OF I EQUAL TO 1
IS I GREATER THAN 100 ?
YES
PRINT THE VALUE OF SUM
END
NO
SQUARE I
ADD THE SQUARE TO THE VALUE OF SUM
ADD 1 TO THE VALUE OF I

Figure 1.2 Flow chart of an algorithm to sum the squares of the integer numbers 1 to 100.

scientifically oriented high-level language – FORTRAN. The FORTRAN project was started in the early 1950s and it produced a high-level language for scientific use on IBM computers. It is very much a card-oriented language since it was designed at a time when most programmers entered their programs and data on punched cards. Since that time it has become one of the world's most widely used scientific languages and almost all computer manufacturers provide a compiler for it. This in itself presents a problem for although the language is in widespread use, there was for some time no universally recognized definition of it. The version of FORTRAN recognized by one compiler could be quite different from that recognized by another. To remedy this, the American National Standards Institute (ANSI) produced a definition of FORTRAN in 1966 and all manufacturers were encouraged to provide compilers which conformed to this standard. The result of this would be an improvement in program *portability*, i.e. a FORTRAN program written for one computer to the ANSI 66 standard should run with little or no modification on any other computer with an ANSI FORTRAN compiler. The FORTRAN compilers produced by many manufacturers provide enhancements to the ANSI definition of the language but programmers are warned that the use of such features should be avoided if possible since they may cause problems when the program is moved to a new computer. It is for this reason that the main text of this book describes ANSI 66 FORTRAN.

Since the 1966 definition, FORTRAN has been used for more and more programming applications and several weaknesses in the language became apparent. In 1977 the ANSI committee produced an upgrade to the definition known as ANSI 77 FORTRAN. This is implemented now on a wide range of machines and is becoming increasingly used. Additional sections have been added to this book to describe the ANSI 77 enhancements to FORTRAN and these are highlighted in the text. Complete beginners may see the additional FORTRAN 77 features as unnecessary complications to the language, so perhaps it should be pointed out, at this stage, that the aims of the extensions are mainly to make it easier to write correct programs by removing irritating restrictions in FORTRAN 66, and by providing better building blocks plus extra features such as character handling.

Throughout the text the term FORTRAN will be used to refer to both languages. Features specific to the ANSI 77 definition will be referred to as either ANSI 77 or FORTRAN 77. Note that, with a few important differences, ANSI 77 FORTRAN is a superset of ANSI 66 FORTRAN. This should allow earlier programs to run with only minimal changes, if any.

1.2 Layout of FORTRAN programs

Historically, FORTRAN is a card-oriented language and this early influence is still apparent in the language today. The statements of FORTRAN programs are typed on 80 character lines, one statement per line. Each character position on the line is called a column; this derives from early nomenclature when FORTRAN statements were typed on 80 column punched cards. Columns are numbered from left to right starting with column 1.

Lines are divided into four distinct regions. FORTRAN statements may be typed in columns 7 to 72 only, but may be positioned anywhere within that region. In general, spaces or blanks are ignored in FORTRAN statements and may be used to improve readability.

Any FORTRAN statement may be labelled with a numeric *label* of 1 to 5 decimal digits which may be positioned anywhere within columns 1 to 5 of the line. This region is reserved for that purpose only.

Columns 73 to 80 of the line are ignored by the compiler and may be used for any purpose. One common use for this region is to number the statements in order; it was particularly important to do this in the heyday of punched cards when large programs could be shuffled or dropped accidentally and would have to be reassembled in correct sequence.

FORTRAN allows long statements, which will not fit on to one line to be constructed by means of continuation lines. These are indicated by typing any character (except space or zero) in column 6 of the line. The very first line of the statement must contain a space or zero in column 6. Subsequent lines are often identified as a continuation of the first line by typing 1, 2, 3, etc. in column 6. The standard definition of FORTRAN allows up to 19 continuation lines (although in practice some compilers may produce errors for less than this number). Statements may be broken and continued at any point and do not necessarily have to extend up to column 72 before they are continued. Columns 1 to 5 of a continuation line must be blank.

Comment lines may appear anywhere in a FORTRAN program and are totally ignored by the compiler. They are indicated by the letter C in column 1 and columns 2 to 80 may be used for explanatory comment. **FORTRAN 77 permits the use of an asterisk in column 1 also.** The liberal use of the comment facility improves program readability and is to be encouraged (see Fig. 1.3). More hints on writing good FORTRAN programs are given throughout the book and in the Conclusion. Note that in a comment line, any character capable of representation in the computer may be used but this may vary from one computer to another. Lines which are completely blank from columns 1 to 72 (or 80) are also accepted as valid comment lines by some compilers but are not standard.

Since FORTRAN is a language, like any other language it must have an alphabet in which it can be written and typed. This consists of:

(1) The decimal digits 0 to 9
(2) The upper case letters A to Z
(3) The arithmetic operators + – * /
(4) Left and right parentheses ( )

(5) Equals =
(6) Comma, and decimal point .
(7) **Apostrophe ' and colon :**
(8) The currency symbol **$** (sometimes £)
(9) Space or blank

Note that in this book the terms 'space' and 'blank' are used synonymously.

The decimal digits and letters together are known as the *alphanumeric characters.* **The apostrophe and colon are ANSI 77 additions. Note that although the currency symbol forms part of the character set, no rules are given in the standard about how it should be used.**

Some versions of FORTRAN will permit other characters to be used and will permit more than one statement per line. However, such practices are not to be encouraged if programs are to be run with little modification on a variety of machines. Note also that the use of the currency symbol can sometimes cause portability problems.

The rules for the layout of lines apply only to FORTRAN statements in programs. They do not apply to the layout of data for the program. As we shall see later, this is under the control of the programmer who has all 80 columns available for this purpose.

Figure 1.3 shows a typical FORTRAN program to find the roots of a quadratic equation of the form:

$$ax^2 + bx + c = 0$$

These are given by the expression

$$x = \frac{-b \pm \sqrt{(b^2 - 4ac)}}{2a}$$

The statements of this program will be described in the following chapters. At present it is sufficient to note the layout of the program.

In the remaining sections of this chapter and in the following two chapters, the basic concepts of FORTRAN will be introduced so that the program example in Fig. 1.3 will be comprehensible to the reader. This should enable similar simple programs to be written, and exercises to do this will be provided at the end of Chapters 2 and 3.

1.3 Data types

All computers work, in principle, by obeying the instructions of a program stored at a particular address in memory. These instructions operate on data stored in other locations in memory. Many different types of data may be stored in a computer's memory and each may require a different unit of store, e.g. a word, several words, a bit or a byte.

```
C***********************************************************************
C                                                                      *
C A FORTRAN PROGRAM TO FIND THE ROOTS OF A QUADRATIC EQUATION          *
C                                                                      *
C***********************************************************************
C
C READ THE COEFFICIENTS OF THE EQUATION
C
      READ(5,10)A,B,C
C
C THE NEXT STATEMENT PROTECTS AGAINST A POSSIBLE DIVISION BY
C ZERO LATER ON. EXECUTION OF THE PROGRAM IS TERMINATED IF A=0.
C
      IF(A.EQ.0.0)STOP
C
C OTHERWISE CALCULATE B SQUARED MINUS 4AC
C
      B2M4AC=B**2-4.0*A*C
C
C THE NEXT STATEMENT CHECKS FOR IMAGINARY ROOTS
C
      IF(B2M4AC.LT.0.0)GO TO 1
C
C OTHERWISE CALCULATE THE ROOTS.
C
      B2M4AC=SQRT(B2M4AC)
      A2=2.0*A
      SOLN1=(-B+B2M4AC)/A2
      SOLN2=(-B-B2M4AC)/A2
C
C NOW PRINT THE RESULTS
C
      WRITE(6,20)SOLN1,SOLN2
C
C AND TERMINATE THE PROGRAM
C
      STOP
C
C IF ROOTS ARE COMPLEX THEN OUTPUT WARNING MESSAGE
C
  1   WRITE(6,30)
C
C AND TERMINATE THE PROGRAM.
C
      STOP
C
C THE FORMAT STATEMENTS REQUIRED BY THE PROGRAM
C
  10  FORMAT(3F10.5)
  20  FORMAT(14H THE ROOTS ARE,2F10.5)
  30  FORMAT(23H THE ROOTS ARE COMPLEX.)
C
C END OF PROGRAM
C
      END
```

Figure 1.3 A FORTRAN program to find the roots of a quadratic equation.

FORTRAN allows five basic data types – integer, real, double precision, complex and logical. **FORTRAN 77 has introduced an additional character type.** A more detailed discussion of types is given in Chapter 5 but for simplicity we shall consider only the types real and integer at this stage.

As a program is obeyed it may change the contents of data storage locations. Other locations hold a fixed value which must not change during program execution. The latter is a program *constant* and the former a program *variable.*

1.4 Constants

Integer and real data types in FORTRAN are each assigned one numeric storage location in memory (usually a word) but the bits within that location are used in a different way for the two types.

In a FORTRAN program, an integer constant is a sequence of decimal digits without a decimal point and may be preceded optionally by a + or – sign, e.g. 50, –25 and +3 are all valid FORTRAN constants. Unsigned constants are taken to be positive. This decimal number is converted internally to a binary representation so that each bit in the word represents a power of 2 (where the rightmost bit is 2^0). A few decimal numbers with the corresponding binary representations are:

5	101	$(1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5)$
32	100000	$(1 \times 2^5 = 32)$
10	1010	$(1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 10)$

It follows from this that the maximum sized integer which can be held in a computer is dependent on the word length of that computer and this varies from one machine to the next. The leftmost bit of the word is reserved usually for the sign bit which is 0 if the number is positive and 1 otherwise. Thus in a 16 bit word machine, the maximum sized positive integer that may be stored is represented by a leftmost bit of 0 and all other bits 1 (i.e. 32767). The maximum sized negative number is represented by a leftmost bit of 1 and all other bits 0 (i.e. –32768). Note that a word with all bits set to 1 usually represents –1.

The internal representation should not, however, concern the FORTRAN programmer unduly. Decimal numbers are used throughout FORTRAN programs. But note that integers are held *exactly* in the computer (up to the maximum allowed value that is).

In FORTRAN, neither decimal point nor commas are permitted in integer constants. Spaces within constants are allowed and are ignored by the compiler. Leading zeros may be specified, e.g. 059 and 59 are both valid and have the same value.

The following are all examples of valid integer constants:

```
237   –82   +1   000   0028   –0   +0
```