MA155A
Introduction to Computing and Programming
FORTRAN course
M. K. Warby
September 1996 – January 1997

Room: M209

Address: BICOM, Department of Mathematics, Brunel University, Uxbridge, Middlesex UB8 3PH, U.K.

Login Id.: `icstmkw`

Email: `Mike.Warby@brunel.ac.uk`

URL: `http://www.brunel.ac.uk/~icstmkw/home.html`

Tel: `++44 1895 274000 Ext. 2169`

Mathematics Department Fax: `++44 1895 203303`

Contents

Part I

The FORTRAN language

1 Introduction

As the title of this course suggests, the programming part of of MA155A is introductory and it assumes no previous programming experience. Thus these notes begin by defining a small collection of the terminology used in computing and in programming.

1.1 Some Computing Jargon

Hardware The collection of electronic and electromechanical components and devices which makes up the "physical part" of the computer.

Software The programs which control the computer.

Computer Memory The area of the computer which stores the instructions and data to be processed.

CPU This is the Central Processing Unit. This is where the "processing" is performed. The CPU consists of a **control unit** and an **arithmetic unit**.

Input Device The means for entering a program and data into a computer, typically this is the keyboard and/or a mouse.

Output Device The means for communicating the results to the outside world, typically a VDU (Visual Display Unit) or a printer.

Bit A binary digit. A binary digit has 2 possible values which are usually represented by 0 or 1. A bit is the smallest unit of storage.

Byte This is 8 bits. The unit of storage normally used.

Machine Code or Assembly Language The language understood by the computer.

High Level Languages The name for programming languages in which the instructions are in "human readable" form. For example FORTRAN, C, C++, Basic, Pascal, Algol, ADA, Cobol, etc.

FORTRAN This is short for FORmula TRANslation.

Compilers Special programs for translating high level language programs into machine code which is understood by a computer. In this course we will use the FTN77 FORTRAN compiler from Salford Software on the PC's operating under DOS (Disk Operating System). You can also use the SUNPro FORTRAN compiler f77 on the Unix system.

Editors Special programs for creating files which usually consist of ASCII characters. (ASCII is short for American Standard Code for Information Interchange.) In this course the files are usually FORTRAN programs or input files required in the running of a program.

Code Jargon for computer programs.

Bug Jargon for an error in a program. The story is that a moth short-circuited two thermionic valves in one of the earliest computers. The charcoaled moth (bug) took days to find.

1.2 Why learn programming ?

This is a natural question to ask of any subject. If you are going to be programmer then it is reasonably clear that you need to start learning programming at some stage. However the advantages of learning programming are not so evident if you think that you may rarely if ever need to write programs in the future. For example your use of computers in the future may be only to use specific packages. To answer this it should be realised that learning programming does involve learning how to logically think out and specify a problem. A skill which may be useful elsewhere, i.e. in the current jargon it is a transferable skill. Also if in the future you mainly only use specific packages then you often find tasks which you may wish to do but which is not done by the package. Having programming skills often enables you to relatively quickly overcome such difficulties. Finally, there is of course the obvious reason for learning programming now being that you are examined in it as part of your Level 1 Scheme of Studies.

1.3 Why FORTRAN ?

In deciding which programming languages to first learn there is always much debate between academics as to which language to teach and there is never 100% agreement. Of the popular languages around (e.g. FORTRAN, C, C++, Basic, Pascal) FORTRAN is still probably the most widely used scientific languages and it should also be said one of the most criticised languages. It is the oldest of these languages, the most well supported in terms of numerical and other scientific software and it probably the “most standard”. This long history of FORTRAN is used in both arguments for keeping FORTRAN and for rejecting FORTRAN in favour of a more “modern language”. Which language is best often depends on the type of programs that are to be written and it is also a matter of personal choice. On the practical side, FORTRAN is taught here as we have more expertise in this language than in other languages, the software is available and there is only about 8 weeks timetabled for it.

1.4 Some landmarks in the history of FORTRAN

1954 Researchers at IBM (International Business Machines Corporation) develop the IBM Mathematical FORmula TRANslation system, FORTRAN.

1957 The first FORTRAN compiler was generally available.

1966 The FORTRAN IV standard was approved by ANSI and ISO. (ANSI=American National Standards Institute, ISO=International Standards Organization)

1978 FORTRAN 77 is approved by ANSI and ISO.

1991 FORTRAN 90 is approved by ANSI and ISO.

As yet we do not have the FORTRAN 90 compiler at Brunel and thus we will be using FORTRAN 77 which is a subset of FORTRAN 90. Also, as has been already mentioned, there is only 8 weeks available to programming and thus we can only cover a subset of FORTRAN 77.

1.5 Reference books

There are many books which describe FORTRAN 77 and there are also several books which describe FORTRAN 90. It is difficult to recommend any single text book as "the course text book". However a list of books which I have used at some stage follows.

Fortran 77 books

[BM79] Balfour A. and Marwick D. H. (1979) *Programming in Standard FORTRAN 77.* Heinemann Educational Books, London. QA76.73.F25B34.

[Edg92] Edgar S. L. (1992) *FORTRAN for the 90's, Problem Solving for Scientists and Engineers.* Computer Science Press, Oxford. QA76.73.F25E35.

[KF93] Koffmann E. B. and Friedman F. L. (1993) *FORTRAN with Engineering Applications.* Addison-Wesley, Reading, Massachusetts, fifth edition. QA76.73.F25K63.

[Mon82] Monro D. M. (1982) *Fortran 77.* Edward Arnold, London. QA76.73.F25M66.

Fortran 90 books (not covered in this course)

[EPL94] Ellis T. M. R., Philips I. R., and Lahey T. M. (1994) *FORTRAN 90 Programming.* Addison-Wesley, Reading, Massachusetts. QA76.73.F25E435.

[Hah94] Hahn B. D. (1994) *Fortran 90 For Scientist and Engineers.* Edward Arnold, London.

[MR91] Metcalf and Reid (1991) *Fortran 90.* Oxford University Press, Oxford. QA76.73.F25M47.

[Smi95] Smith I. M. (1995) *Programming in Fortran 90, A First Course for Engineers and Scientists.* Wiley, Chichester. QA76.73.F25S34.

[Wil95] Wille D. R. (1995) *Advanced Scientific Fortran.* Wiley, Chichester. QA76.73.F25W54.

2 Complete Programs, types and constants

The following are "short" valid FORTRAN 77 programs. In each case the programs are created using an editor.

The first program just prints 1 line containing 2 words.

```
      WRITE(*,*)'hello world'
      END
```

The next program just prints approximations to π and e using single precision arithmetic and double precision arithmetic respectively.

```
      WRITE(*,*)4*ATAN(1.0),EXP(1.)
      WRITE(*,*)4D0*ATAN(1D0),EXP(1D0)
      END
```

Finally the following program is documented by the comment statements.

```
C
C ::: the volume and surface area of a sphere of radius r
      PI=4.*ATAN(1.)
      R=2.3
      V=PI*R**3
      S=(4./3.)*PI*R**2
      WRITE(*,*)'r,v,s=',R,V,S
      END
```

If the file named `test01.for` corresponds to one of the above programs then the process of compiling, linking and running the program depends on your computer system and the particular FORTRAN compiler being used. In the case of FTN77 operating under DOS this can be achieved by the command

`ftn77 test01 -undef -lgo`

You should note here that the compiler requires that the name of the FORTRAN program is of the form `*.for`.

2.1 Compulsory statements and the columns to use

In the above 3 examples the `END` statement is compulsory. Each program unit must have exactly 1 `END` statement. It denotes the end of the program unit when the program is compiled and it acts as a `STOP` statement in the main program when it is encountered. (In each of the above examples there is only 1 program unit which is known as the main program. Initially all your programs will have only 1 program unit. It is only when you start to use subprograms that you will have programs with more than 1 program unit.)

In FORTRAN 77 there are rules concerning the layout of the statements. These are as follows:

Column 1 If this contains the character '`C`', '`c`' or '`*`' then the line is a comment line. (The 3rd example program contains comment lines.) Comments help the readability of the program but do not affect in any way how it runs.

Columns 1–5 This is for statement labels which can be any positive integer which fits into these 5 columns, i.e. any number between 1 and 99999. (There are no labels in the above examples.)

Column 6 This must be left blank unless the statement on the previous line is incomplete and it needs to be extended onto the current line. Any character other than the blank character or the digit `0` means that the line continues the previous line. (There are no continuation lines in the above examples.)

Column 7–72 The main part of each statement is put here. Thus the majority of non-comment statements have their first 6 characters as blank characters. When such a statement is entered with an editor I would recommend that the `TAB` key is NOT used although it may appear to help position the cursor key in the correct position. If a `TAB` key is used it depends on the compiler how this is interpreted.

2.2 Integers, Reals and Characters

You need to distinguish between these in FORTRAN and indeed in other high level programming languages. To understand why you need to consider how things are actually stored or are represented in a computer. In its most basic form all information is stored as a series of bits where a bit is a binary digit.

2.2.1 Integers

For integers which are of length 2 bytes there are $2 \times 8 = 16$ bits which gives $2^{16} = 65536$ different possible on/off possibilities for these bits. With FORTRAN this is used to represent any integer in the range $[-2^{15}, 2^{15} - 1] = [-32768, 32767]$. Integers within this range are represented exactly but integers outside this range cannot be represented in 2 bytes. Similar reasoning applies to integers which are 4 bytes in size where the range becomes $[-2^{31}, 2^{31} - 1] = [-2147483648, 2147483647]$.

2.2.2 Reals

For real numbers the mantissa–exponent representation is used which involves numbers of the form

$$(\text{mantissa}) \times 2^{(\text{exponent})}$$

(The term floating point representation is also used here.) With reals of length 32 bits (4 bytes) a portion of these is used for the mantissa and the remainder is used for the exponent. The portions used depends on the computer and it is typically 24 bits for the mantissa and 8 bits for the exponent which leads to an accuracy in arithmetic operations of about 7 decimal digits and a number range of about $\pm(10^{-37}, 10^{37})$. Given any real number the computer approximates it by the nearest number that it can store which is of the form

$$(\text{mantissa}) \times 2^{(\text{exponent})}$$

and thus it is important to note that small errors may be incurred when numbers are stored and when arithmetic operations are performed.

A similar reasoning applies to reals which are 8 bytes in length. (These are sometimes referred to as double precision.) In this case we get about 15 decimal digit accuracy and a number range of about $\pm(10^{-307}, 10^{307})$.

2.2.3 Characters

In the case of characters, each character is stored in 1 byte. Thus there are $2^8 = 256$ different possible characters which can be represented. This is more than enough for all the printable characters (e.g. `A, B, ..., Z, a, b, ..., z, 0, 1, ... 9, +, - , *, /` etc.).

The key point about types is that FORTRAN needs to know which of these types it is dealing with in order to properly interpret the bytes.

Computers use the binary number system but humans prefer to use the decimal number system which is what is used in FORTRAN programs.

2.3 Constants

2.3.1 Integer constants

E.g. `0, 1, -1, 29, -3197`

2.3.2 Real constants

E.g. `0.0, , 1.0, -1., 29., -3197E3, 0.3197E4, 3197E0`

A constant is real if a decimal point is used or if the exponent `E` is used. Observe that the same real number can be given in more than one way as in `-3.197E3, -3.197E+03, 0.3197E4, 3197E0` and `3197.0` which corresponds to the real number `3197`.

2.3.3 Character strings

E.g. `'name', 'dog and cat', 'hello world'`

The character string is between the quote character '. It is respectively 4 bytes, 11 bytes and 11 bytes in length in the above 3 examples.

Take care not to use similar looking symbols to ' on your keyboard. You should not use the character ‘ or the double quotes character ". Some FORTRAN compilers allow both the ' and " characters and indeed this is part of FORTRAN 90. The FORTRAN 77 compiler FTN77 does not however allow this.

2.4 The FORTRAN 77 Character set

Standard FORTRAN 77 only allows the following.

The letters `A` to `Z`
The digits `0` to `9`
The 13 special characters `= + - * / ( ) , . $ ' :` and the blank character.

You can safely add the 26 lower case letters `a` to `z` to the above as this is allowed with nearly all FORTRAN 77 compilers.

3 Variables, Arithmetic Expressions and Assignment Statements

3.1 Variables

A variable in a FORTRAN 77 program is an entity possessing both a **name** and a **type**.

3.1.1 The name

In standard FORTRAN 77 the name cannot be more than 6 letters in length, it must begin with one of the 26 letters `A` to `Z` and it can only contain alphanumeric characters, i.e. `A, ..., Z, 0, 1, ..., 9`.

Many FORTRAN compilers allow names of more than 6 characters with FTN77 allowing 32 characters. 31 characters is the FORTRAN 90 standard. Thus 6 letter length restriction can be safely ignored. Also most FORTRAN compilers allow both the use of upper and lower case letters although it must be noted that in most situations the compiler is case insensitive and will thus consider, for example, the names

`A1b3` and `a1B3`

to be the same.

3.1.2 The type

By default names beginning with a letter in the range `A-H` or `O-Z` are assumed to be of type `REAL` whilst a variable beginning with a letter in the range `I-N` is assumed to be of type `INTEGER`. To override this convention or to explicitly declare your variables you can use type statements, e.g.

```
INTEGER A,B,I,J
REAL C19,L
```

which declares variables `A, B, I` and `J` to be of type `INTEGER` and variables `C19` and `L` to be of type `REAL`.

These are not the only types in FORTRAN 77. FORTRAN also allows the types `DOUBLE PRECISION` (which can also be referred to as `REAL*8` on most computers), `LOGICAL`, `COMPLEX` and `CHARACTER`.

3.2 Arithmetic Expressions and Assignment Statements

In the following we assume that all the variables have the type determined by the 1st letter of their name.

Examples

```
I=I+1
D=SQRT(B**2-4*A*C)
X1=(-B+D)/(2*A)
X2=(-B-D)/(2*A)
PI=4*ATAN(1.0)
V=(A+B)**2/4.0
B=1/2
C=1.0/2.0
F=1.0/2
```

<u>Remarks</u>

1. The statement `I=I+1` must be interpreted as take the contents of the current variable `I`, add `1` to it and store the result in `I` (which overwrites the previous contents).

2. `SQRT` and `ATAN` are the FORTRAN intrinsic functions for the square root $\surd$ and inverse tangent, i.e. $\tan^{-1}$.

3. `V=(A+B)**2/4.0` corresponds to the mathematical statement $v = \frac{1}{4}(a+b)^2$.

4. `B=1/2` and `C=1.0/2.0` give different results in FORTRAN and the FTN77 compiler will warn about the statement `B=1/2`. The statement is a valid FORTRAN statement but the compiler queries the statement as the outcome is probably not what you intend. To understand this it is important to distinguish between the types `INTEGER` and `REAL`. `1` and `2` are both written in a way which means that they are to be interpreted as integers. The arithmetic operation `1/2` is thus performed in integer arithmetic (as both quantities are integers) and the integer answer is `0`. As `B` is of type real the integer `0` is then converted to the real `0.0` which is the outcome of the statement.

5. `C=1.0/2.0` and `F=1.0/2` will give the same result. In the second case the division involves a real constant and an integer constant. In order to cope with this the integer `2` is converted to the real number `2.0` and the division is done with real arithmetic as you would hope. Integers and reals can be safely mixed in arithmetic expressions provided at each stage at least one of the quantities in each part of an expression is of type `REAL`.

3.3 The levels of precedence in arithmetic expressions

The processing of an expression is done according to the following order of precedence of the brackets and arithmetic operations that appear.

1st	`()`
2nd	Function evaluations
3rd	`**`
4th	`*` or `/`
5th	`+` or `-`

Operations of equal precedence are processed from left to right (except for the `**` operation). Thus with integers `I*J/K` is the same as `(I*J)/K` but is generally different to `I*(J/K)`. I recommend that you use brackets whenever you feel it makes the expression clearer even if they are not strictly needed.

3.4 FORTRAN Intrinsic Functions

The standard mathematical functions are in FORTRAN called **intrinsic functions**, i.e. they are builtin to the language. The tag **external** is used for functions that you create yourself. The following are some of the intrinsic functions. In each case listed the variable `X` is assumed to be of type real.

`SQRT` used as e.g. `SQRT(X)` corresponding to $\sqrt{x}$.

`EXP` used as e.g. `EXP(X)` corresponding to e^x.

`LOG` used as e.g. `LOG(X)` corresponding to $\log_e(x)$ or $\ln(x)$.

`LOG10` used as e.g. `LOG10(X)` corresponding to $\log_{10}(x)$.

`SIN` used as e.g. `SIN(X)` corresponding to $\sin(x)$.

`COS` used as e.g. `COS(X)` corresponding to $\cos(x)$.

`TAN` used as e.g. `TAN(X)` corresponding to $\tan(x)$.

`ASIN` used as e.g. `ASIN(X)` corresponding to $\sin^{-1}(x)$, the value in the range $[-\pi/2, \pi/2)$ is returned.

`ACOS` used as e.g. `ACOS(X)` corresponding to $\cos^{-1}(x)$ the value in the range $[0, \pi]$ is returned.

`ATAN` used as e.g. `ATAN(X)` corresponding to $\tan^{-1}(x)$ the value in the range $(-\pi/2, \pi/2)$ is returned.

`SINH` used as e.g. `SINH(X)` corresponding to $\sinh(x)$, i.e. $(\mathrm{e}^x - \mathrm{e}^{-x})/2$.

`COSH` used as e.g. `COSH(X)` corresponding to $\cosh(x)$ i.e. $(\mathrm{e}^x + \mathrm{e}^{-x})/2$.

`TANH` used as e.g. `TANH(X)` corresponding to $\tanh(x)$, i.e. $\sinh(x)/\cosh(x)$.

In the following list the intrinsic functions are concerned with type conversion or require more than one argument.

`INT` used as `INT(X)` which gives the integer part of `X`. Thus if `X=3.7` then the result is `3.0`. If `X=-3.7` then the result is `-3.0`.

`NINT` used as `NINT(X)` which gives the nearest integer to `X`. Thus if `X=3.7` then the result is `4.0`.

`ABS` used as `ABS(X)` and mathematically corresponds to $|x|$.

`MAX` used as `MAX(A,B)` or `MAX(X1,X2,X3)` or `MAX(A,B,X,Y)` or ... . In all cases the result returned is the largest of the variables listed. All the variables must be of the same type.

`MIN` used as `MIN(A,B)` or `MIN(X1,X2,X3)` or `MIN(A,B,X,Y)` or ... . In all cases the result returned is the smallest of the variables listed. All the variables must be of the same type.

`ATAN2` used as `ATAN2(Y,X)`. This gives the angle of the point $(x, y) \neq (0, 0)$ as measured from the x-axis in the anti-clockwise sense. The value returned is in the range $[-\pi, \pi)$.

4 Control statements

The statements in a program are executed sequentially (i.e. in the order they are written) unless there are statements to override this. Statements which override the sequential order are called control statements. Certain of the control statements involve the testing of logical expressions with the subsequent order in which statements are evaluated depending on the outcome of these tests. We thus first consider logical expressions.

4.1 Logical Expressions

Logical expressions are expresions which produce a result of type `LOGICAL` with a value of true or false. For example

```
B*B.LT.4.0*A*C
```

is the FORTRAN way of testing if $b^2 < 4ac$. The outcome is true or false depending on the contents of the variables `A`, `B` and `C` at the time the statement is executed. The relational operator that can be used are tabulated below.

`.LT.`	$<$
`.LE.`	$\leq$
`.EQ.`	$=$
`.NE.`	$\neq$
`.GT.`	$>$
`.GE.`	$\geq$

Often you wish to combine logical expressions. This can be done with the following.

`.NOT.` as in `.NOT.(B*B.LT.4.0*A*C)`.

`.AND.` as in `(B*B.LT.4.0*A*C).AND.(A.GT.0.0)`. The result is true if both conditions are true.

`.OR.` as in `(X.LT.2).OR.(X.GT.6)`. The result is true if at least one of the conditions is true. In this example it is not possible for both conditions to be true and thus it is true either when $x < 2$ or when $x > 6$ and it is false otherwise, i.e. when $2 \leq x \leq 6$.

If you want a test which evaluates to true when $i \neq 1$ and $i \neq 2$ then you can put, as part of a statement,

```
(I.NE.1).AND.(I.NE.2)
```

When both bracketed parts are true the whole expression is true. You should contrast this with the logical expression

```
(I.NE.1).OR.(I.NE.2)
```

which would probably only be entered as a mistake as it always evaluates to true. Thus care is needed when creating logical expressions. Care is also needed when you may wish to use the operator `.EQ.` with real quantities. For example if `A` and `B` are real variables then you can write `A.EQ.B`. This is correct FORTRAN but the outcome can be unreliable as it only evaluates to true if the floating point representations of `A` and `B` are identical and when `A` and `B` are as a result of floating point arithmetic this is unlikely. `A` and `B` may be very close but unless they are identical this will evaluate to false. The FTN77 compiler warns you when you use such comparisons.

4.2 IF THEN ELSE block structures

This can have any of the following forms. In the simplest case we have

```
IF( logical expression )THEN
...
... almost any statements
...
ENDIF
```

If the logical expression is true then the statements in the block are executed. If not then the block is skipped. A slightly more complicated form of the block is

```
IF(  logical expression )THEN
...
...  Statements A
...
ELSE
...
...  Statements B
...
ENDIF
```

If the logical expression is true then *Statements A* are executed and if it is false then *Statements B* are executed. As a further form you can have

```
IF(  logical expression 1 )THEN
...
...  Statements 1
...
ELSEIF(  logical expression 2 )THEN
...
...  Statements 2
...
ELSEIF(  ... )THEN
...
...
...  possibly many other ELSEIF statements
...
...
ELSEIF(  logical expression m )THEN
...
...  Statements m
...
ELSE
...
...  Statements B
...
ENDIF
```

In this case *logical expression 1* is tested and *Statements 1* are executed if it is true. Otherwise *logical expression 2* is tested and *Statements 2* are executed if it is true. This testing and possible execution of a block of statements is repeated until a logical expression is reached which is true. If none of the logical expressions are true then the *Statements B* following the `ELSE` statement are executed. In all cases when the appropriate block of statements has been executed the next statement executed is the first statement following the `ENDIF`.

As an example, if you wish to solve the quadratic $ax^2 + bx + c = 0$ then you may have the statements

```
      ...
      ... A, B and C must be assigned values
      ...
      D=B**2-4*A*C
      IF(D.GE.0.0)THEN
      DS=SQRT(D)
      X1=(-B+DS)/(2.0*A)
      X2=(-B-DS)/(2.0*A)
      WRITE(*,*)'roots are',X1,X2
      ENDIF
      ...
```

To output some information in the case $b^2 - 4ac < 0$ and to add some comments and to indent the blocks you can modify this to

```
      ...
      ... A, B and C must be assigned values
      ...
      D=B**2-4*A*C
      IF(D.GE.0.0)THEN
C
C ::: real roots case
        DS=SQRT(D)
        X1=(-B+DS)/(2.0*A)
        X2=(-B-DS)/(2.0*A)
        WRITE(*,*)'roots are',X1,X2
C
      ELSE
C
C ::: complex roots case
        WRITE(*,*)'roots are complex'
        WRITE(*,*)'real part=',B/(2.0*A)
        WRITE(*,*)'imaginary part=+/-',SQRT(-D)/(2.0*A)
      ENDIF
      ...
```

As a different example suppose that MONTH is an integer variable containing the number of the month of the year. To output the name of the month you might put

```
...
IF(MONTH.EQ.1)THEN
  WRITE(*,*)'The month is January'
ELSEIF(MONTH.EQ.2)THEN
  WRITE(*,*)'The month is February'
ELSEIF(MONTH.EQ.3)THEN
...
... missing statements
...
ELSEIF(MONTH.EQ.12)THEN
  WRITE(*,*)'The month is December'
ELSE
  WRITE(*,*)'Month is out of range 1 to 12 !!'
  STOP
ENDIF
...
```

The STOP statement in the above causes the program to stop. You can have any number of STOP statements in a program.

As a final remark about IF-THEN-ELSE block structures it should be noted that they can be nested. For example you can have statements of the form

```
...
IF(  logical expression 1 )THEN
  ...
  ...
  IF(  logical expression 2 )THEN
    ...
    ...
  ENDIF
  ...
  ...
ELSE
  ...
  ...
  IF(  logical expression 3 )THEN
    ...
    ...
  ENDIF
  ...
  ...
ENDIF
...
```

There is no limit specified in the FORTRAN language to the level of nesting that can be done but there will probably be a compiler dependent limit although this will rarely if ever be a problem for a programmer.

4.3 The Do-loop

In its simplest form this involves statements of the form

```
      DO   s i=exp1,exp2
      ...
      ...  the do-loop statements
      ...
    s terminal statement
```

where s is a statement label (a number between `1` and `99999`), i should be the name of an integer (it is known as the Do-variable) and *exp1* and *exp2* should be of type `INTEGER`. In this form *exp1* is the starting value of i, *exp2* is the final value of i with i increased in steps of 1. Thus for example you can have

```
      ...
      DO 10 I=1,29
      ...
      ...  the do-loop statements
      ...
   10 CONTINUE
      ...
```

In this case the statements in the do-loop are executed for `I=1`, then for `I=2`, then for `I=3`, ... and finally for `I=29` (this is provided there is no `GOTO` statement (to be described later) in the loop to jump out of the loop). Such a construction can be used to compute sums. For example if mathematically

$$s = \sum_{i=1}^{1000} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{1000}$$

then a complete program which computes s is as follows.

```
      S=0.0
      DO 10 I=1,1000
      S=S+1.0/I
   10 CONTINUE
      WRITE(*,*)'s=',S
      END
```

The variable `S` is first **initialised** to `0.0`. Then in the do-loop it successively stores the partial sums corresponding to 1, $1+1/2$, $1+1/2+1/3$, ..., $1+1/2+\cdots+1/1000$ with at the end of the loop S containing the (floating point approximation to the) required value.

If you wish the do-variable to increase in steps which are different from 1 then you should use the slightly more general form of the statement which is

```
      DO   s i=exp1,exp2,exp3
      ...
      ...   the do-loop statements
      ...
    s terminal statement
```

where *exp3* should be an integer (positive or negative but not 0) and it is called the incrementation parameter. Thus you can have

```
      ...
      DO 20 I=1000,1,-1
      ...
      ...   the do-loop statements
      ...
   20 CONTINUE
      ...
```

which means that I takes the values 1000, 999, 998, ..., 1 (in this order). You can also have statements of the form

```
      ...
      DO 30 I=L,M,N
      ...
      ...   the do-loop statements
      ...
   30 CONTINUE
      ...
```

which means that I takes the values L, L+N, L+2*N, ... until M is reached or passed. For this to work L, M and N must have been given values before the statement is executed and N≠0. If at the time of execution N>0 but M<L or similarly if N<0 but M>L, i.e. the values are incompatible, then the do statement block is skipped.

An alternative syntax for the DO-loop

There is another form of the syntax of the do-loop which is supported by many FORTRAN 77 compilers and which is part of FORTRAN 90 which avoids the need for the statement labels. The syntax is

```
DO i=exp1,exp2,exp3
...
...  the do-loop statements
...
ENDDO
```

Although this is not part of standard FORTRAN 77 it can be used in this course. Thus the summation program can be written as

```
      S=0.0
      DO I=1,1000
      S=S+1.0/I
      ENDDO
      WRITE(*,*)'s=',S
      END
```

As another example consider Newton's method for attempting to solve

$$f(x) = 0 \ .$$

Given a starting value x_0 it involves computing x_1, x_2, ... by using

$$x_{n+1} = x_n - f(x_n)/f'(x_n), \quad n = 0, 1, 2, \cdots$$

In the case $f(x) = x^2 - 2$, $f'(x) = 2x$ we have

$$x_n - f(x_n)/f'(x_n) = x_n - \frac{x_n^2 - 2}{2x_n} = \frac{1}{2}\left(x_n + \frac{2}{x_n}\right) \ .$$

With $x_0 = 2$ we can compute $x_1, \cdots, x_6$ with the program

```
C
C ::: A Newton's method example
C
C ::: set starting point
      X0=2.0
C
C ::: the iteration
      DO I=1,6
        X0=0.5*(X0+2.0/X0)
        WRITE(*,*)'i,x0=',I,X0
      ENDDO
      END
```

When the program is run the value stored by `X0` corresponds respectively to x_0 then x_1, then x_2, then ... and finally x_6.

As in the case of the IF-THEN-ELSE construction DO-loops can be nested.

4.4 The `GOTO` statement

This is of the form

```
GOTO  n
```

where n is a statement label. It means go to or jump to the statement labelled n. Extensive or inappropriate use of `GOTO` statements can make a program very difficult to understand. I would recommend that if it is used then it should only be used sparingly and you should always consider whether or not there is a better way of achieving the same task.

As an example of a situation where it might be used consider again the Newton iteration program. If a stage is reached when $f(x_m)$ is very small in magnitude and similarly for $f(x_{m+1})$, $f(x_{m+2})$, ... etc. then from a practical point of view we judge that the iteration has "converged" and that x_m is a sufficiently good approximation to the required solution. The subsequent computations of x_{m+1}, x_{m+2}, ... are thus unnecessary. In writing a program we would thus wish to stop the program from performing the unnecessary computation. You can achieve this as is demonstrated by the following program.

```
C
C ::: A Newton's method example
C
C ::: set starting point
      X0=2.0
C
C ::: the loop
      DO I=1,20
C
C ::: set new value and the function value
        X0=0.5*(X0+2/X0)
        FVALUE=X0**2-2.0
        WRITE(*,*)'i,x0=',I,X0
C
C ::: test for convergence and exit loop if it has
        IF(ABS(FVALUE).LT.1E-6)THEN
          WRITE(*,*)'iteration has converged'
          GOTO 20
        ENDIF
      ENDDO
C
C ::: not converged
      WRITE(*,*)'iteration has not converged after 20 iterates'
C
   20 CONTINUE
      END
```

If the logical expression `ABS(FVALUE).LT.1E-6` evaluates to true then you jump out of the loop to the labelled statement. Thus the loop is only performed whilst the logical expression remains false.

5 Arrays and Subscripted variables

There are many situations in mathematics where subscripted variables are used. For example given n numbers $x_1, x_2, \cdots, x_n$ the mean is defined by

$$\bar{x} = \frac{1}{n}\sum_1^n x_i,$$

the variance σ^2 is defined by

$$\sigma^2 = \frac{1}{n-1}\sum_1^n (x_i - \bar{x})^2$$

and $\sigma = \sqrt{\sigma^2}$ is the standard deviation. To conveniently do such computation in FORTRAN you can use arrays. To use arrays you will first have to declare this with a statement of the form

```
      REAL X(100)
```

which declares a 1-dimensional array with name `X` of type real and size `100`. Such a statement must be one of the first statements in your program unit. Within the program unit you can then refer to the entries of the array as `X(1)`, `X(2)`, ..., `X(100)`. Thus to compute the mean a skeleton program is as follows.

```
      REAL X(100)
      ...
      ...  obtain the number N.LE.100  and X(1),...,X(N)
      ...  somehow (see later)
      ...
C
C ::: get xbar=mean of x(1),...,x(n)
      SUM=0.0
      DO 10 I=1,N
        SUM=SUM+X(I)
   10 CONTINUE
      XBAR=SUM/N
C
C ::: get sigma=standard deviation
      SUM=0.0
      DO 20 I=1,N
        SUM=SUM+(X(I)-XBAR)**2
   20 CONTINUE
      SIGMA=SQRT(SUM/(N-1))
C
      WRITE(*,*)'mean,standard deviation=',XBAR,SIGMA
      ...
      ...
      END
```

In the above example the declaration `REAL X(100)` reserves space for `X(1)`, ... ,`X(100)`.

In some situations you may wish to have negative or zero subscripts. This can be achieved as in the following example.

```
      REAL A(-5:5),B(0:6)
```

which declares two 1-dimensional arrays of length 11 and 7 respectively with values `A(-5)`, ..., `A(-1)`, `A(0)`, `A(1)`, ... `A(5)` and `B(0)`, `B(1)`, ..., `B(6)` respectively.

You can declare arrays with more than 1 dimension. For example

```
      REAL A(4,4),B(2,3)
```

declares two 2-dimensional arrays. Within the computer the 16 entries of `A` which we would probably think of as being arranged as the matrix

```
A(1,1)  A(1,2)  A(1,3)  A(1,4)
A(2,1)  A(2,2)  A(2,3)  A(2,4)
A(3,1)  A(3,2)  A(3,3)  A(3,4)
A(4,1)  A(4,2)  A(4,3)  A(4,4)
```

are in fact arranged in a linear way in the order

`A(1,1)`, `A(2,1)`, `A(3,1)`, `A(4,1)`, `A(1,2)`, `A(2,2)`, `A(3,2)`, `A(4,2)`, `A(1,3)`, `A(2,3)`, `A(3,3)`, `A(4,3)`, `A(1,4)`, `A(2,4)`, `A(3,4)`, `A(4,4)`.

That is the order is column by column or with the first subscript varying most quickly. Similary the 6 entries of `B` are arranged in the linear way in the order

`B(1,1)`, `B(2,1)`, `B(3,1)`, `B(1,2)`, `B(2,2)`, `B(3,2)`.

That is the arrays are 2-dimensional in the way you must refer to the entries but they are 1-dimensional in the way they are stored and they are stored column by column. This aspect is important when arrays are passed from one program unit to another (see later).

Arrays in FORTRAN can have up to 7-dimensions and, as in the previous example, if you want the subscript to start at any integer other than 1 then you must use the colon notation. In the above the arrays have been declared with the keyword `REAL`. For an integer array you should use `INTEGER`, e.g.

```
INTEGER L(20),M(5,5)
```

You can also declare arrays with the keyword `DIMENSION`, e.g.

```
DIMENSION A(10,0:3),NN(4)
```

In this case the type of the array is determined implicitly or by other type statements.

6 Input and Output

In the examples so far there have been statements of the form

```
WRITE(*,*) 's=',S
```

where the effect is to output the 2 characters `s=` followed by the contents of `S` in some form. This is an example of **list-directed output** to the **designated output device**. The items to be output are separated by commas. It is the first `*` in `(*,*)` which indicates that the output device is the designated output device which in our case is the terminal, i.e. the result appears on the VDU. The second `*` in `(*,*)` indicates that the information is to be written in a list directed way with the computer system deciding on the precise format to be used. If you wish to send the output to a file instead of to the screen then you need to change the first `*` in `(*,*)`. Similarly if you wish to specify the format of the output then you should change the second `*` in `(*,*)`. Before this is described in more detail we first consider list directed input.

6.1 List directed input from the terminal

You have statements of the form

```
READ(*,*)   list
```

where *list* is a list of variables names, array entries or array names. For example if A, B and C are real variables then

```
READ(*,*) A,B,C
```

is the instruction for inputting A, B and C at the keyboard. You can enter these numbers on one line or spread over several lines with the numbers entered separated by blank spaces or commas. In fact the correct terminology here is not 'a line' but 'a record' although in most cases these are in effectively the same thing. The data entered at the keyboard is only processed when the return key is entered which signifies the end of a record. In this example the input statement is satisfied when the 3 numbers have been successively read. If when a record is processed there are in total more than 3 numbers then the extra numbers are ignored and if when a record is processed there are less than 3 numbers then the statement expects the additional numbers to be entered on subsequent records. Thus in a program for solving a quadratic equation with real roots you can have

```
      WRITE(*,*)'enter a,b,c'
      READ(*,*)A,B,C
      DS=SQRT(B**2-4*A*C)
      X1=(-B+DS)/(2*A)
      X2=(-B-DS)/(2*A)
      WRITE(*,*)'roots are ',X1,X2
      END
```

In the example given a WRITE statement precedes the READ statement. This is recommended as a prompt to the data you are expected to enter at the keyboard.

6.2 List directed input from a file

List directed input from the keyboard is not recommended when there are many items of data as there is a reasonable chance that you will make mistakes entering the data leading to incorrect answers or a breakdown of the program due to invalid characters being entered. Also the next time you try to run the program in this form you will again have to try to enter all the numbers correctly and you are again just as likely to make mistakes. Instead you should create a file with an editor which contains all the input items and you should write your program in such a way that it reads from the file. To do this you need to open the file and you should have appropriate read statements. In the quadratic equation program this can be done as follows.

```
OPEN(10,FILE='quadrati.dat')
READ(10,*)A,B,C
CLOSE(10)
...
...  as before
...
```

For this to work the file called `quadrati.dat` needs to already exist and it must contain at least 3 numbers in a form which can be read by the list directed input statement. In this example the number `10` is the input device number or input channel number. (The input device number should be between `0` and `99` but you should note that the numbers `0`, `1`, `2`, `5` and `6` have special significance in FORTRAN and/or DOS and thus these numbers are best avoided in the `OPEN` statement.) The `CLOSE` statement closes the connection between the program and the file. A successful completion of the run of the program also closes all the files that are open and thus in this example the `CLOSE` statement is not necessary but it is considered to be good programming practice to include it when you know that no more data will be read.

There are more complicated forms of the `OPEN` statement. However the only one that will be given here is as follows. You can replace the above `OPEN` statement with

```
OPEN(10,FILE='quadrai.dat',STATUS='OLD')
```

In this form the program will crash at this point if the file `quadrati.dat` does not already exist, i.e. if it does not have status `'OLD'`. If a status is not explicitly given then it is taken to have the status `'UNKNOWN'`.

There are also more complicated forms of the `READ` statement. In these notes only the additional keyword known as `IOSTAT` is described. Typically this is used if you wish to prevent a program crashing at a `READ` statement due to invalid data. You can use it as illustrated in the following statement

```
READ(*,*,IOSTAT=IT)A,B,C
```

In the statement it is assumed that `IT` is a variable of type `INTEGER`. If the `READ` statement is successful then `IT` is set to `0` and `A`, `B` and `C` have been assigned values. If not then it is set to a non-zero value and `A`, `B` and `C` are undefined. This then gives you the possibility of writing the program in such a way that if `IT` is set to a non-zero value then you can request that the data is entered again until 3 numbers are successfully entered.

You would use the `READ` statement in this form if you need to increase the robustness of your program.

As an example of list directed input from a file with arrays consider the following problem.

Example

Suppose the marks of 100 students are stored in a file called `marks.dat` and you wish to compute the mean and standard deviation. You will need to know the format of the file `marks.dat` in order to know how to construct the `READ` statement and so for this example suppose that the contents of `marks.dat` is

```
31.0  65.0  50.0  17.0  74.0  55.0  16.0  25.0  20.0  14.0
30.0  69.0  43.0   5.0  34.0  19.0   9.0  29.0  60.0  73.0
75.0  79.0  54.0  95.0  90.0  23.0  26.0  45.0  87.0  16.0
43.0  33.0  48.0  45.0  81.0  16.0  77.0  31.0  25.0  55.0
72.0  40.0  36.0  67.0  93.0  30.0  94.0  33.0  99.0  59.0
40.0  26.0  20.0  21.0  77.0  32.0  38.0  47.0  52.0  47.0
13.0  57.0   7.0  91.0  34.0  51.0  64.0  56.0   4.0  20.0
 5.0  22.0  22.0  71.0  12.0  23.0  24.0   7.0  23.0  19.0
49.0  17.0  33.0  50.0  59.0  34.0  84.0  78.0   3.0  78.0
10.0  39.0  60.0  44.0 100.0  13.0  52.0  15.0  27.0  66.0
```

A complete program follows.

```
C
C ::: mean and standard deviation program
      PARAMETER(N=100)
      REAL X(N)
C
C ::: open file, read data and close file
      OPEN(10,FILE='marks.dat')
      READ(10,*)(X(I),I=1,N)
      CLOSE(10)
C
C ::: get xbar=mean of x(1),...,x(n)
      SUM=0.0
      DO 10 I=1,N
        SUM=SUM+X(I)
   10 CONTINUE
      XBAR=SUM/N
C
C ::: get sigma=standard deviation
      SUM=0.0
      DO 20 I=1,N
        SUM=SUM+(X(I)-XBAR)**2
   20 CONTINUE
      SIGMA=SQRT(SUM/(N-1))
C
      WRITE(*,*)'mean,standard deviation=',XBAR,SIGMA
      END
```

There are two new features in this program. The `PARAMETER` statement gives the name `N` to the constant `100` and the statement

```
READ(10,*)(X(I),I=1,N)
```

contains an **implied do-loop**. It means read N numbers into respectively X(1), X(2), ..., X(N) where the N numbers can be on one record or spread over several records (there are 10 records in the above example). In the implied do-statement I plays the role of the do-index or the do-parameter with 1 as the starting value and N as the terminal value as in a do-loop specification. In this example the READ statement can be simplified to

```
READ(10,*)X
```

which means read as many numbers as is the size of X. Thus if X is of size 100000 then the statement is only satisfied when 100000 numbers have been read.

6.3 Output to a File

You can send output to a file instead of to the screen by first opening it as in the last subsection and then replacing the desiginated output identifier * to the channel number used in the OPEN statement. For example

```
OPEN(20,FILE='quadrati.res')
WRITE(20,*)'x1,x2=',X1,X2
CLOSE(20)
```

If in this example the file quadrati.res exists before the program containing this statement is run then its contents are overwritten by the output produced by this WRITE statement. If you wish to ensure that you do not overwrite a previously created file then you should instead put

```
OPEN(20,FILE='quadrati.res',STATUS='NEW')
```

The program will fail when it runs if the file quadrati.res already exists.

The remarks about reading arrays also apply to the writing arrays. For example in the mean–standard deviation example you can write

```
WRITE(20,*)(X(I),I=1,N)
```

which writes X(1), ..., X(N) to the file. If X is an array then the statement

```
WRITE(20,*)X
```

writes the whole array to the file.

There are a number of other specifiers that you can use with OPEN, CLOSE, READ and WRITE statements many of which are concerned with various sorts of failure scenarios. Also there is an INQUIRE statement which can be used to check the status of files which can be used to improve the robustness of FORTRAN programs. The additional features of the OPEN, CLOSE, READ and WRITE statements and the INQUIRE statement will not be considered in this course.

6.4 Formatted Output

List directed input is the simplest way of inputting data as it does not require that the data be precisely formatted, it only requires that each item is appropriately separated by one or more blanks or by a comma. The result of list directed output can however be untidy and when this is the case you need to consider making an effort to more precisely format your output. As an example consider the following statement involving list directed output.

```
      WRITE(*,*) 'a,b,c=',A,B,C
```

A formatted version of the statement can be given as

```
      WRITE(*,'(A,F10.5,2X,F10.5,2X,F10.5)')'a,b,c=',A,B,C
```

or as

```
      WRITE(*,9000) 'a,b,c=',A,B,C
 9000 FORMAT(A,F10.5,2X,F10.5,2X,F10.5)
```

The format specifier is `'(A,F10.5,2X,F10.5,2X,F10.5)'` which can be given inside the `WRITE` statement or as a separate labelled statement as shown. It is a matter of personal choice as to which you prefer to use. In the `FORMAT` specification `A` is for a character string, `F` is for a floating point number and `X` is for a space. `F10.5` means that the number has width 10 characters with 5 decimal digits after the decimal point. As the decimal point occupies a character there is only 4 decimal digits possible before the point. To use the `F` descriptor you need to know the likely size of `A`, `B` and `C` beforehand. If you do not then you should use the `E` descriptor as in

```
      (*,'(A,E13.5,2X,E13.5,2X,E13.5)')
```

`E13.5` means that the number occupies 13 character positions (the width) with 5 decimal digits after the decimal point in a mantissa-exponent representation. For example you will get numbers of the form

```
      -0.24601E+03
```

for the number `-246.01`. The `24601` occupies 5 positions and the `-0.` and the `E+03` together give a further 7 positions. The 13 positions are made up a leading blank.

To summarize some of the field descriptors that can be used we have the following.

`I`	For integers as in `I7`.
`F` and `E`	For reals as in `F10.5` or `E13.5`.
`A`	For character strings of any length.
`X` or `T`	For column spacing as in `3X` which means 3 spaces or `T30` which means move to column 30.
`/`	For a new line.

You can repeat a specifier or a set of specifiers as in

```
5E13.5
```

which is equivalent to

```
E13.5,E13.5,E13.5,E13.5,E13.5
```

or as

```
3(E13.5,5X)
```

which is equivalent to

```
E13.5,5X,E13.5,5X,E13.5,5X
```

As an example consider the times table up to 12. This can be neatly generated by the following program.

```
      WRITE(*,'((12I4))')((I*J,J=1,12),I=1,12)
      END
```

The `WRITE` statement is outputting 144 values. The `12I4` specification is for 12 integer values of width 4 character positions (`-999` to `9999` fits into this width). After the first 12 numbers have been written the format specification is repeated with the next 12 values and so on until all the data has been output. Thus if you give incomplete format specifiers for the amount of data being written then it re-uses the specification as many times as is necessary to satisfy the `WRITE` statement.

The best way to learn about format specification is by example.

6.5 Formatted Input

This will not be covered and it is not recommended when you have the option of also using list directed input. In most of my own programs I only use formatted input for reading characters.

7 Function and Subroutine Subprograms

In all the examples programs so far presented there has only been one program unit which is known as the main program. This was all that was necessary as the tasks to be performed could be done in a small number of FORTRAN statements. Also in each case there was not much scope for breaking down the problem into a series of smaller subproblems. This however will not be the case for larger tasks. In this case the usual programming strategy or algorithm is to break down the task into a number of smaller tasks, an approach known as "top down design". Hopefully each of the smaller tasks can be done with a "smaller number of statements". As we will show each of the smaller tasks or perhaps a group of the smaller tasks can often be conveniently written as a subprogram of some kind which you (or anyone else) may be able to make use of in other similar tasks. For example consider again the code computing $x_1, x_2, \cdots, x_6$ defined in Newton's method by

$$x_{n+1} = x_n - f(x_n)/f'(x_n), \quad n = 0, 1, \cdots, 5$$

given on page 18 in the case $f(x) = x^2 - 2$ and with $x_0 = 2$. Omitting for brevity comments and assuming the explicit types of variables you can write

```
      X0=2.0
      DO I=1,6
        X0=X0-F(X0)/DF(X0)
        WRITE(*,*)I,X0
      ENDDO
      END
      FUNCTION F(X)
      F=X**2-2.0
      END
      FUNCTION DF(X)
      DF=2.0*X
      END
```

The computation of the Newton iterates is written for a general function f and has the advantage that you do not need to concern yourself with the details of f or f' when you write this part of the program. You can postpone this detail until you need to write the function subprograms `F` and `DF`. The complete program shown now consists of a main program and two function subprograms.

It is generally considered to be good programming practice to include a `PROGRAM` statement naming the main program, to include some comments and to declare the variables that are used and thus a better presented version of the previous program conforming to these conventions is as follows.

```
      PROGRAM NEWTONM
      REAL X0,F
      INTEGER I
C
C ::: 6 steps of Newton's method for a particular function f
C
C ::: set starting point
      X0=2.0
C
C ::: the loop
      DO I=1,6
        X0=X0-F(X0)/DF(X0)
        WRITE(*,*)I,X0
      ENDDO
      END
      FUNCTION F(X)
      REAL F,X
C
C ::: the function f
      F=X**2-2.0
      END
      FUNCTION DF(X)
      REAL DF,X
C
C ::: the function df=derivative of f
      DF=2.0*X
      END
```

7.1 Syntax of the Function subprogram

The general form is as follows.

```
FUNCTION  name(  ..dummy arguments..)
...
... function body
...
END
```

The type of the `FUNCTION` is the implicitly determined type (as determined by the 1st letter) or it can be explicitly given in the `FUNCTION` statement, as in e.g.

```
REAL FUNCTION F(X)
```

or after the `FUNCTION` statement, as in e.g.

```
FUNCTION F(X)
REAL F
```

Within the function body the name of the function is treated as a variable and this must be assigned a value before the control is returned to the referencing program unit. Control is returned to the referencing program unit if the `END` statement is reached or if a `RETURN` statement is reached. For example you can have the following function subprogram.

```
      FUNCTION F(X,I)
      REAL F,X
      INTEGER I
C
C ::: f=function, 1st derivative, 2nd derivative (at x)
C :::   when i is respectively 0,1,2
      IF(I.EQ.0)THEN
        F=(X**5)*EXP(X)
        RETURN
      ELSEIF(I.EQ.1)THEN
        F=(X**5+5*X**4)*EXP(X)
        RETURN
      ELSEIF(I.EQ.2)THEN
        F=(X**5+10*X**4+20*X**3)*EXP(X)
        RETURN
      ELSE
        WRITE(*,'(A,E13.5,2X,I4)')'i is out of range !!  x,i=',X,I
        STOP
      ENDIF
      END
```

which returns $f(x)$, $f'(x)$ or $f''(x)$ in the case $i = 0$, $i = 1$ or $i = 2$ and in the case $f(x) = x^5 \exp(x)$. If i has a different value then a message is written and the program is stopped.

In the above example `F` has two arguments, a real argument followed by an integer. All references to `F` must have **the same number of arguments and the arguments must have matching types**. This is a very important point if the program is going to compile and run as you intend it too. Another important thing to realise is that all names used and all statement labels in the subprogram are **local** to the subprogram. The communication with the subprogram is only through the argument list and through `COMMON` blocks (there are no such blocks in this example). About the only thing which is not local to a subprogram is an input or output channel, i.e. if the input channel has been opened in one program unit then it remains open on that unit until a corresponding `CLOSE` statement is encountered or until the program terminates.

In all the examples given so far the arguments have been variables. You can also have arguments which are arrays or subprogram names. The passing of subprograms as arguments will not be considered in these notes but we will next consider the passing of arrays by means of examples.

7.2 Arrays as Arguments

Consider the following skeleton program relating to the mean-standard deviation program given previously.

```
      PARAMETER(N=1000)
      REAL X(N),MEAN
      ...
      ... obtain the N values X(1), ..., X(N) somehow
      ...
      XBAR=MEAN(X,N)
      ...
      END
      FUNCTION MEAN(A,M)
      REAL A(M),MEAN
      INTEGER M
C
C ::: compute the mean of a(1), ..., a(m)
      INTEGER I
      MEAN=0.0
      DO 10 I=1,M
        MEAN=MEAN+A(I)
   10 CONTINUE
      MEAN=MEAN/M
      END
```

The real function subprogram MEAN has an array as its first argument and it is declared within the subprogram with **adjustable dimensions**, i.e. as

```
      REAL A(M)
```

This can be done in subprograms provided the array name and the adjustable bound (M in this case) is also an argument. The association between the entries X(1), X(2), ... in the main program and A(1), A(2), ... in the subprogram are, in this case, as you would expect and correspond to

```
X(1)  X(2)  ···  X(N)
A(1)  A(2)  ···  A(N)
```

The bytes in the computer memory which correspond to the entry A(I) are the same as that which corresponds to X(I) and thus changing A(I) in the subprogram means that X(I) is correspondingly changed. The correspondence of the array elements between the referencing program unit and the subprogram is according to **the way the array is**

stored and to the starting position indicated in the reference. Thus `MEAN(X,N)` means the first entry of `X` corresponds to the first entry of `A`. In the case that `X` is declared as

```
REAL X(0:N)
```

this would mean that the alignment is

```
X(0)  X(1)  ···  X(N-1)
A(1)  A(2)  ···   A(N)
```

As another example suppose that the function subprogram is as given and the statement

```
V=MEAN(X(47),6)
```

is given in the referencing program unit. In this case the alignment is

```
X(47)  X(48)  X(49)  X(50)  X(51)  X(52)
 A(1)   A(2)   A(3)   A(4)   A(5)   A(6)
```

The value returned is thus the mean of `X(47)`, ..., `X(52)`.

As a final example suppose that `X` is declared as a 2-dimensional array with the type statement

```
REAL X(3,2)
```

and the statement

```
V=MEAN(X,6)
```

is used. It is valid FORTRAN to have different dimensions of the arrays as declared in the two program units. The correspondence is according to how the arrays are stored. In this case this is as follows.

```
X(1,1)  X(2,1)  X(3,1)  X(1,2)  X(2,2)  X(3,2)
 A(1)    A(2)    A(3)    A(4)    A(5)    A(6)
```

7.2.1 A Possible Runtime Error

With FORTRAN 77 the actual bounds of the arrays must be given somewhere, typically in the main program, with adjustable dimensions being permitted in subprograms. This can lead to the following run time error situation which can sometimes be difficult to track down. Suppose that the array `X` is declared in the main program as

```
REAL X(1000)
```

and the statement

```
Y=MEAN(X,9000)
```

is also given in the main program. The array labelled `A` in subprogram `MEAN` expects to have length `9000` but the length of `X` in the main program is only `1000`, i.e. insufficient space has been allocated. The checking options of the FTN77 compiler will detect this error but if these are not used then the program will fail without too much indication as to the reason for the failure. This will be the case with all other compilers. Thus care is needed to ensure that you never try to reference entries outside of the bounds of your arrays.

7.3 Subroutine subprograms

As so far described the function subprogram can take any number of arguments and returns a single value (the value of the function). It is possible to return other pieces of information by assigning or by altering any of the arguments or any quantities specified in common blocks (not yet covered). However it is a convention that if the task is to return several pieces of information then a subroutine subprogram is used instead. A subroutine subprogram has the form

```
SUBROUTINE  name(  ..dummy arguments..)
...
... subroutine body
...
END
```

and it is referenced by a statement of the form

```
CALL  name(   ..actual arguments..)
```

name is the name of the subroutine and it has no type associated with it. Apart from these syntax differences a subroutine subprogram is the same as a function subprogram in the association of the actual arguments with the dummy arguments.

We can now consider again the mean-standard deviation program with now a subroutine subprogram which computes the mean and the standard deviation. Such a subroutine is as follows.

```
      SUBROUTINE STATS(A,M,MEAN,STDEV)
      REAL A(M),MEAN,STDEV
C
C ::: Compute the mean (mean) and standard deviation (stdev)
C ::: of a(1), ..., a(m)
      INTEGER I
      REAL SUM
C
C ::: the mean
      MEAN=0.0
      DO 10 I=1,M
        MEAN=MEAN+A(I)
   10 CONTINUE
      MEAN=MEAN/M
C
C ::: the stdev
      SUM=0.0
      DO 20 I=1,M
        SUM=SUM+(A(I)-MEAN)**2
   20 CONTINUE
      STDEV=SQRT(SUM/(M-1))
      END
```

This can be referenced with a statement of the form

```
CALL STATS(X,100,XBAR,SIGMA)
```

7.4 Common Blocks

In the above description the only communication between a subprogram and its referencing program unit is through the argument list. There is however an alternative way of doing which involves common blocks. Common blocks are useful in situations where a long list of variables and arrays need to be available to several different program units. You can of course pass all the variables and arrays as arguments but this will give long and cumbersome argument lists. In such a case the preferred approach is to use common blocks which in its simplest case involves placing a statement of the form

COMMON *list of variables and arrays*

in each program unit that needs to share the variables and arrays. This is in fact a **blank common block** which is all that will be covered here. (You can also have named common blocks.)

Remarks

1. A variable or array cannot appear in both a `COMMON` block and as an argument to a subprogram.

2. Common blocks are frequently given using an `INCLUDE` statement which is allowed with many FORTRAN 77 compilers and it is part of FORTRAN 90 but which is not part of standard FORTRAN 77. This involves putting the `COMMON` statement (which may need to be spread over several lines) separately into a file and then referring to this file with an `INCLUDE` statement at all the places where you need the `COMMON` statement which helps to ensure consistency. If the include file is called `mycommon.inc` then the syntax of the `INCLUDE` statement is

   ```
   INCLUDE 'mycommon.inc'
   ```

3. When argument lists are used as the mechanism for communicating between program units the correspondence required that the actual arguments and the dummy arguments agreed in type. In particular this required that there was the same number of actual arguments and dummy arguments. With common blocks it is the length of the block in terms of bytes that matters and you need to have the same length in each program unit. In two program units you can have, for example,

   ```
   COMMON U,V,W,X,Y,Z
   ```

   and

   ```
   COMMON A(6)
   ```

   Assuming that all quantities are real the correspondence is

U	V	W	X	Y	Z
A(1)	A(2)	A(3)	A(4)	A(5)	A(6)

   (In this example the statement

   ```
   COMMON A(6)
   ```

   declares `A` to be array of length `6` as well as specifying the common block.)

7.5 Library Routines

These are just a collection of subprograms for performing "standard" tasks which you can reference with a `CALL` statement or by appropriately referring to the function name. If the library routine performs the task that you want then you can use this instead of investing the time and effort into devising and writing your own (possibly inferior and almost certainly less well tested) version of the code. To use the library routine you just need to know the details of the arguments in the argument list and the local site and computer dependent information about how to **link** your program to this library. This will not be covered further in this course.

8 January 1996 Examination paper with solutions

Brunel
THE UNIVERSITY OF WEST LONDON

DEPARTMENT OF MATHEMATICS AND STATISTICS

BACHELOR DEGREE EXAMINATIONS

January 1996

MA155A Introduction to Computing

Time Allowed: 2 hours

All questions carry equal marks.
Full marks can be obtained for satisfactory answers to **FIVE** *questions.*
If more than **FIVE** *questions are answered then only the best* **FIVE** *solutions will be counted.*
In some questions you are asked to give Fortran statements. When you give these statements we will assume that you are putting any statement labels in columns 1 to 5 with the main part of the statement being in columns 7 to 72 as is required in standard Fortran 77.

1. (i) Convert the following into Fortran assignment statements. In your answer you can use one statement or several statements.

(a)

$$c = 1 + \frac{b}{v} + \frac{d}{v^2}$$

(b)

$$d = \frac{38.197(r^3 - s^3)\sin(x)}{(r^2 - s^2)x}$$

(c)

$$z = e^{(\cos(x) - 4\sin(y))}$$

For the above `SIN(X)`, `COS(X)` and `EXP(X)` give respectively the values of the intrinsic Fortran functions sine, cosine and exponential at the point `X`.

(ii) Convert the following Fortran statements into algebraic form.

(a) `F=1.0/(2.0*3.14159)*(1.0/XL*C)**0.5`

(b) `R=1.0/((1.0/X1)+(2.0/X2)+(3.0/X3))`

(iii) The statements

```
FUNCTION D(X1,Y1,X2,Y2)
D=SQRT((X2-X1)**2+(Y2-Y1)**2)
END
```

define a FUNCTION subprogram for computing the distance between two points `(X1,Y1)` and `(X2,Y2)`. If `(X1,Y1)`, `(X2,Y2)` and `(X3,Y3)` are the three vertices of a triangle then give a Fortran statement which uses the subprogram to compute the length of the boundary of the triangle.

(iv) Give Fortran statements to compute

$$s = \sum_{n=1}^{20000} \frac{1}{n}$$

---

2. What will be the output to the screen as a result of the following sets of assignment statements.

(a)
```
      A=7.23
      B=9.25
      A=A-B
      B=A+B
      A=B-A
      WRITE(*,*)A,B
      END
```

(b)
```
      DO 20 I=2,3
      DO 10 J=1,I
      WRITE(*,*)I*J
   10 CONTINUE
      WRITE(*,*)(I+J,J=1,I)
   20 CONTINUE
      END
```

(c)
```
      A=1.2
      B=2.1
      C=F(A,B)-0.04
      D=F(C,C)
      WRITE(*,*)C,D
      END
      FUNCTION F(X,Y)
      F=(X-2)**2+Y
      END
```

(d)
```
      I=2
      A=I/3
      B=I/3.0
      C=A-B
C
C ::: output a,b and c with 3 digits after the decimal point
      WRITE(*,9000)A,B,C
 9000 FORMAT(1X,3F10.3)
      END
```

(e)
```
      A=1.1
      C=2.9
      CALL SUB1(A)
      WRITE(*,*)C
      END
      SUBROUTINE SUB1(A)
      C=A*A
      WRITE(*,*)C
      END
```

3. (i) Suppose a real array `X` has been declared of length 30 with entries `X(1)`, $\cdots$, `X(30)`. Give statements which will determine the maximum of the entries and the mean of these entries.

   (ii) Assuming that `X(1)`, $\cdots$, `X(30)` store 30 distinct values and the maximum `XMAX` of these numbers has already been computed, give Fortran statements which will determine the second largest value.

   (iii) Assuming that `X(1)`, $\cdots$, `X(30)` store 30 real values give statements which determine the number of these values in the interval $(\mathtt{A}, \mathtt{B})$.

---

4. (i) The solutions of the quadratic equation

$$ax^2 + bx + c = 0, \quad a \neq 0$$

are

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

In the above you may assume that a, b and c are real. If $b^2 - 4ac \geq 0$ then x_1 and x_2 are real whilst if $b^2 - 4ac < 0$ then x_1 and x_2 are complex. Write a Fortran program that will read the numbers a, b and c and will output the values x_1 and x_2 for all possible values of a, b and c with $a \neq 0$.

   (ii) Some Fortran compilers warn you about unreliable results if you use statements of the form

```
D=B**2-4*A*C
IF(D.EQ.0)THEN
X1=-B/(2*A)
WRITE(*,*)'double root at x1=',X1
ENDIF
```

where in the above `D` is a real variable. What is the problem and why do you not get the warning with such statements if `D` is of type `INTEGER` ?

   (iii) If in part (i) $b > 0$ and $|a|$ is very small compared to $|b|$ then the root

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} = \frac{-2c}{b + \sqrt{b^2 - 4ac}} \approx -\frac{c}{b} .$$

In this situation indicate why it is better to compute x_1 using the representation

$$x_1 = \frac{-2c}{b + \sqrt{b^2 - 4ac}}$$

instead of the representation given in part (i).

---

5. (i) Newton's method for numerically solving $f(x) = 0$ involves choosing a starting point x_0 and successively generating

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \quad k = 1, 2, \cdots .$$

In the case of $f(x) = e^x - 2x - 1$ and with the starting value $x_0 = 1$ give a FORTRAN program which generates x_1, x_2, $\cdots$, x_l where l is determined by the condition $l \le 10$ or by the condition $|f(x_l)| \le 10^{-5}$. When x_l is obtained the program should stop. For the above the FORTRAN intrinsic functions `EXP` and `ABS` are such that `EXP(X)` gives the exponential of the real variable `X` and `ABS(Y)` gives the absolute value of the real variable `Y`.

(ii) If a function subprogram of the form

```
FUNCTION F(X)
REAL F,X
...
F=...
END
```

has already been written for computing the value of a function f at a point `X` then write a main program which uses this in order to compute the Trapezoidal rule approximation

$$s_n = h\left(\frac{f(0) + f(1)}{2} + \sum_{1}^{n-1} f(x_i)\right), \quad h = 1/n, \quad x_i = i/n$$

to the integral

$$s = \int_0^1 f(x)\ dx$$

in the case $n = 64$.

6. A vending machine allows customers to feed in more money than is required for an item and it gives the appropriate change which can be up to $99p$ and which can involve the coins $1p$, $2p$, $5p$, $10p$, $20p$ and $50p$. When change is given the vending machine always gives the highest value coins in order to make up the appropriate amount. As an example, if the change is $73p$ then the change returned to the customer is a $50p$, $20p$, $2p$ and a $1p$. If n, $1 \le n \le 99$, is the change to be returned to the customer then give a program which determines the coins that the vending machine will give.

---

7. It can be shown that the binomial coefficients

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}$$

involved in the binomial expansion

$$(1+x)^n = \sum_{r=0}^{n} \binom{n}{r} x^r$$

satisfy the recurrence relation

$$\binom{n+1}{r} = \binom{n}{r} + \binom{n}{r-1}, \quad r = 1, \cdots, n$$

with the end values given by

$$\binom{n}{n} = \binom{n}{0} = 1 \ .$$

By first declaring an array `IBIN` by the statement

```
DIMENSION IBIN(0:20,20)
```

write a program which computes all the binomial coefficients for $n \le 20$ using the recurrence relation given above in such a way that the `N+1` numbers `IBIN(0,N)`, `IBIN(1,N)`, $\cdots$, `IBIN(N,N)`, correspond respectively to the binomial coefficients

$$\binom{\mathtt{N}}{0}, \ \binom{\mathtt{N}}{1}, \ \cdots, \binom{\mathtt{N}}{\mathtt{N}} \ .$$

Solutions

1. (i) (a)

```
      C=1+B/V+D/(V*V)
```

(b)

```
      A=38.197*(R**3-S**3)*SIN(X)
      B=(R**2-S**2)*X
      D=A/B
```

(c)

```
      V=COS(X)-4*SIN(Y)
      Z=EXP(V)
```

(ii) (a)

$$f = \frac{1}{2(3.14159)} \left(\frac{C}{XL}\right)^{1/2}$$

(b)

$$r = \frac{1}{\dfrac{1}{x1} + \dfrac{2}{x2} + \dfrac{3}{x3}}$$

(iii)

```
      D(X1,Y1,X2,Y2)+D(X2,Y2,X3,Y3)+D(X3,Y3,X1,Y1)
```

(iv)

```
      S=0
      DO 10 N=1,20000
      S=S+1.0/N
   10 CONTINUE
```

---

2. (a) `A` and `B` swap. The output to the screen is

```
9.25   7.23
```

(b)

```
2
4
3 4
3
6
9
4 5 6
```

(c) $(-0.8)^2 + 2.1 - 0.04 = 2.7$
$(0.7)^2 + 2.7 = 0.49 + 2.7 = 3.19$
The output to the screen is

```
 2.7  3.19
```

(d) `A` is `0` due to integer arithmetic being used. The output to the screen is

```
     0.000      0.667     -0.667
```

(e) In `SUB1`, `C=A`$^2 = 1.21$. In the main program, `C=2.9`. The output to the screen is

```
  1.21
  2.9
```

---

3. (i)

```
      XMAX=X(1)
      XMEAN=X(1)
      DO 10 I=2,30
      XMAX=MAX(XMAX,X(I))
      XMEAN=XMEAN+X(I)
   10 CONTINUE
      XMEAN=XMEAN/30
```

(ii)

```
      X2=MIN(X(1),X(2))
      DO 20 I=1,30
      IF(X(I).GT.X2.AND.X(I).LT.XMAX)THEN
      X2=X(I)
      ENDIF
   20 CONTINUE
```

(iii)

```
      NUM=0
      DO 30 I=1,30
      IF(A.LT.X(I).AND.X(I).LT.B)THEN
      NUM=NUM+1
      ENDIF
   30 CONTINUE
```

---

4. (i)

```
READ(*,*)A,B,C
D=B**2-4*A*C
DD=SQRT(ABS(D))
IF(D.GE.0)THEN
  X1=(-B+DD)/(2*A)
  X2=(-B-DD)/(2*A)
  WRITE(*,*)X1,X2
ELSE
  U=-B/(2*A)
  V=DD/(2*A)
  WRITE(*,*)'complex root ',U,V
  WRITE(*,*)'complex root ',U,-V
ENDIF
END
```

(ii) Since `D` is computed by floating point arithmetic it depends on the rounding errors whether or not you obtain `0` or a number very close to `0`.

If `D` is of type INTEGER then exact arithmetic is used.

(iii) The computation of `X1` by subtracting two positive numbers involves a loss of significance. The formula involving the sum of two positive numbers leads to a more accurate result.

5. (i)

```
      X0=1
      DO 10 K=1,10
      E=EXP(X0)
      F=E-2*X-1
      DF=E-2
      X0=X0-F/DF
      IF(ABS(F).LT.1E-5)STOP
   10 CONTINUE
      END
```

(ii)

```
      N=64
      H=1.0/N
      SUM=0
      DO 10 I=1,N-1
      X=I*H
      SUM=SUM+F(X)
   10 CONTINUE
      V=F(0.0)+F(1.0)
      SN=H*(0.5*V+SUM)
      END
```

---

6. $99p$ involves 50,20,20,5,2,2 (i.e. 6 coins).

   No change can involve more than 6 coins.

```
      NREM=N
      DO 100 I=1,6
      IF(NREM.GE.50)THEN
        WRITE(*,*)'give a 50p'
        NREM=NREM-50
      ELSEIF(NREM.GE.20)THEN
        WRITE(*,*)'give a 20p'
        NREM=NREM-20
      ELSEIF(NREM.GE.10)THEN
        WRITE(*,*)'give a 10p'
        NREM=NREM-10
      ELSEIF(NREM.GE.5)THEN
        WRITE(*,*)'give a 5p'
        NREM=NREM-5
      ELSEIF(NREM.GE.2)THEN
        WRITE(*,*)'give a 2p'
        NREM=NREM-2
      ELSEIF(NREM.GE.1)THEN
        WRITE(*,*)'give a 1p'
        NREM=NREM-1
      ELSEIF(NREM.EQ.0)THEN
        STOP
      ENDIF
  100 CONTINUE
      END
```

7.
```
      DIMENSION IBIN(0:20,20)
      DO 100 N=1,20
C
C ::: set the end values
      IBIN(0,N)=1
      IBIN(N,N)=1
C
C ::: set the intermediate values
      DO 10 J=1,N-1
      IBIN(J,N)=IBIN(J,N-1)+IBIN(J-1,N-1)
   10 CONTINUE
C
  100 CONTINUE
      END
```

Part II

Lab. Sessions

9 Computing Tasks for the Lab. Sessions

There should be 8 or 9 FORTRAN Lab. Sessions in this course and it is anticipated that the Lab. Sessions labelled 1, 2 3 and 4 will take at least the two hours allocated. Typically 5 Lab. Sessions may be needed to complete them. The remaining Lab. Sessions are intended for the assignment which will be given out later in the course. This counts for 30% of the total mark of MA155A.

9.1 Lab. Session 1

The following is designed to introduce you to the process of creating a FORTRAN program with an editor of your choice and then compiling, linking and running the program. Usually you will have to create the program yourself in order to solve a particular task but before you are ready for this you should do the following.

1. Create the following program and call the file `prog01.for`.

```
C23456789012345678901234567890123456789012345678901234567890123456789
C
C ::: in the following observe the columns used in the statements
C
C ::: 1st observe the difference between integer,
C ::: real and double precision arithemetic
C ::: the compiler gives a warning with the integer division !!
      WRITE(*,*)'1/3,1./3.,1D0/3D0=',1/3,1./3.,1D0/3D0
C
C ::: some loops
      DO 10 I=1,5
      WRITE(*,*)'i,i**2,i**3,i**4=',I,I**2,I**3,I**4
   10 CONTINUE
C
      DO 30 I=1,3
      DO 20 J=1,I
      WRITE(*,*)'i,j=',I,J
   20 CONTINUE
   30 CONTINUE
C
C ::: a decision with an if-then-else-endif construction
      DO 40 I=1,6
      A=I
      V=ABS(SIN(A))
      IF(V.GT.0.5D0)THEN
      WRITE(*,*)'a,abs(sin(a))=',A,V,' is greater than 0.5'
      ELSEIF(V.GT.0.25D0)THEN
      WRITE(*,*)'a,abs(sin(a))=',A,V,' is greater than 0.25'
      ELSE
      WRITE(*,*)'a,abs(sin(a))=',A,V,' is less than or equal to 0.25'
```

```
      ENDIF
 40   CONTINUE
      END
```

2. Compile, link and run the program with the command

   `ftn77 prog01 -undef -lgo`

   Although you will probably not understand all the statements yet see if you can relate some of the output created to the above statements.

3. Compile, link and run the program within the debug mode with the command

   `ftn77 prog01 -undef -lgo -br`

   You should try typing the `F4` key and then stepping through the program with the `F7` key. At various stages you should see the effect of pressing the `F5` key and to exit from the debug mode you should type the keys `SHIFT` and `F1` together.

4. Type the command

   `ftn77 -help`

   to observe all the options for the `ftn77` command.

5. If you want a printout of your program onto computer paper then you should type

   `copy prog01.for lpt2:`

   Your output will eventually appear in the output racks in the computer centre on computer paper. You will not be charged for this. You are however charged if you send output to a laser printer which corresponds to `lpt3:` . You can type

   `lpq`

   to list the jobs currently in this printer queue. Observe any notices in the computer rooms for other information concerning printing.

6. If you want a printout of the results created by the program then you need to save this information somehow into a file and then to send the file to the printer. A simple way of doing this is to use the DOS redirect character `>`. By typing

   `ftn77 prog01 -undef -lgo >res01`

   the usual screen output is sent to the file `res01`. Similarly the command

   `ftn77 prog01 -undef -lgo >>res01`

   appends the information to the file `res01` (which may involve first creating the file).

7. Create the following program and call the file `prog02.for`. This program computes factorials ($n! = 1 \cdot 2 \cdot 3 \cdots n$) using INTEGER arithmetic.

```
C2345678901234567890123456789012345678901234567890123456789012345678
C ::: factorials using integer arithmetic,
C ::: k accumulates the factorials
      K=1
      DO 10 I=1,300
      K=K*I
      WRITE(*,*)I,K
   10 CONTINUE
      END
```

8. Compile, link and run the program with the command

   `ftn77 prog02 -undef -lgo`

   Note the output and note where the program fails. Now try the command

   `ftn77 prog02 -undef -intl -lgo`

   Again note the output and note where the program fails. In the first case short integers are used whilst in the second case long integers are used.

9. Edit the file `prog02.for` to create `prog03.for` which is a version of the program which uses REAL arithmetic instead of INTEGER arithmetic.

```
C2345678901234567890123456789012345678901234567890123456789012345678
C ::: factorials using real arithemetic,
C ::: s accumulates the factorials
      S=1
      DO 10 I=1,300
      S=S*I
      WRITE(*,*)I,S
   10 CONTINUE
      END
```

10. Compile, link and run the program with the command

   `ftn77 prog03 -undef -lgo`

   Note the output and note where the program fails. Now try the command

   `ftn77 prog03 -undef -dreal -lgo`

   Again note the output and note where the program fails.

9.2 Lab. Session 2

The purpose of this session is to make use of do-loops and if-then-else structures. The first 2 questions only require you to enter specific programs and to run them. The subsequent questions all require you to write your own programs to attempt to solve the specific tasks.

1. Create the DO LOOP ILLUSTRATION program given below and compile, link and run it with a command of the form

   `ftn77 xxxx -lgo -undef`

   Make sure that you understand the output produced.

```
C
C ::: do loop illustration
C
C ::: 1st with a statement label
      DO 10 I=1,7
      WRITE(*,*)I
   10 CONTINUE
C
C ::: next with and enddo statement and a step different from 1
      DO I=13,1,-2
      WRITE(*,*)I
      ENDDO
      END
```

2. Create the IF-THEN-ELSE ILLUSTRATION program given below and compile, link and run it. Make sure that you understand the output produced.

```
C
C ::: if-then-else construction illustration --- neat version
      DO I=1,10
        WRITE(*,*)I
        IF(I.EQ.2)THEN
          WRITE(*,*)'within i=2 block'
        ELSEIF(3.LE.I.AND.I.LT.6)THEN
          WRITE(*,*)'within 3<=i<6 block'
        ELSEIF(I.GT.5.AND.I.LT.7)THEN
          WRITE(*,*)'within 5<i and i<7 (i.e. i=6) block'
        ELSEIF(I.EQ.7.OR.I.EQ.8)THEN
          WRITE(*,*)'within i=7 or i=8 block'
        ELSE
          WRITE(*,*)'within whatever is left block'
        ENDIF
      ENDDO
      END
```

3. Write your own program to compute the value of the sum

$$\sum_{n=1}^{30000} \frac{1}{n}$$

   by adding the numbers in the order 1, 1/2, 1/3, $\cdots$ and then in the order 1/30000, 1/29999, $\cdots$, 2,1 and compare the results.

4. Write your own program to compute Fibonnaci numbers. These are defined as follows. The first 2 numbers are $f_0 = f_1 = 1$ and the subsequent numbers are defined by the recurrence $f_i = f_{i-1} + f_{i-2}$, $i = 2, 3, \cdots$. Thus the beginning of the sequence is 1,1,2,3,5,8,13,21,$\cdots$. Your program should compute these numbers up to f_{40}. Verify that the numbers f_i computed using the recurrence are equal to

$$A_1 {r_1}^i + A_2 {r_2}^i,$$

where

$$r_1 = (1+\sqrt{5})/2,\ r_2 = (1-\sqrt{5})/2,\ A_1 = (1-r_2)/\sqrt{5},\ \text{and}\ A_2 = (r_1 - 1)/\sqrt{5}.$$

5. Let $s_i = \sin(i)$ for $i = 1, 2, 3, \cdots$. Determine a value of i such that

$$s_i = \max\{s_j\ : 1 \le j \le 10000\}$$

6. In the approximation of π by rational numbers $r = m/n$, where m and n are integers, we have $r = 3$ when $n = 1$ and $r = 22/7$ when $n = 7$. By considering numbers of the form m/n for $n = 1, 2, \cdots, 10000$, where m is the nearest integer to πn, determine rational approximations to π which are more accurate than 22/7.

9.3 Lab. Session 3

1. The following illustrate do-loops in FORTRAN.

   Type the short program listed below and run it on the PC with the compiler FTN77.

```
      DO 20 I=1,12
        DO 10 J=1,12
          WRITE(*,*)I,'*',J,'=',I*J
10      CONTINUE
20    CONTINUE
      END
```

   To improve the format of the output edit your program to the following and then run the new version.

```
      DO 20 I=1,12
        DO 10 J=1,12
          WRITE(*,'(I2,A,I2,A,I3)')I,'*',J,'=',I*J
10      CONTINUE
20    CONTINUE
      END
```

   Edit the program further to the following and re-run again.

```
      DO 20 I=1,12
        WRITE(*,'(12(1X,I3,1X))')(I*J,J=1,12)
20    CONTINUE
      END
```

   In the above example the part `(I*J,J=1,12)` is an implied do-loop which can be used in `READ` and `WRITE` statements. It should be interpreted in the same way as the ordinary do-loops and it is used to output several "similar" items on 1 line.

   In the format specifications `I2` and `I3` are for integers for length 2 and 3 respectively. `1X` gives 1 space. `A` is for writing a character string of any length.

   In the format specification `'(12(1X,I3,1X))'` the `12` means repeat the combination `(1X,I3,1X) 12` times.

2. In the following the task is to read in some data from a file into an array and to perform some computations on the data.

   The declaration of an array called `X` of length `30` can be achieved by a statement of the form

```
      DIMENSION X(30)
```

   This must be one of the first statements in the program unit.

   The statements for connecting a file `bearings.dat` on unit number `10` and reading from this file into an array `X` are as follows.

```
OPEN(10,FILE='bearings.dat')
N=30
READ(10,*)(X(I),I=1,N)
CLOSE(10)
```

Create a file called `bearings.dat` containing the 30 numbers

```
16.6  15.3  16.3  14.2  16.7  17.3
18.2  15.6  14.9  17.2  18.7  16.4
19.0  15.8  18.4  15.1  17.0  18.9
18.3  15.9  13.6  18.3  17.2  18.0
15.8  19.3  16.8  17.7  16.8  17.9
```

These numbers are meant to correspond to the lengths obtained when 30 different bearings were measured.

Create a program which reads this data as indicated above and then progressively extend the program to solve the following tasks.

(a) Determine the minimum length and the maximum length.

(b) The mean $\bar{x}$, the variance σ^2 and the standard deviation σ of n items $x_1, \cdots x_n$ are defined, in the usual way, by

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \quad \text{and} \quad \sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2.$$

In the case of the data given above determine the mean $\bar{x}$, the variance σ^2 and the standard deviation σ.

(c) Determine the number of entries which lie in the interval

$$(\bar{x} - \sigma, \bar{x} + \sigma)$$

.

(d) Determine the median of the 30 values which is the mean of the 15th and 16th largest values. This is the most difficult part of this question and it requires that the values are sorted. To assist you here you can use a sorting routine which is an additional routine of the FTN77 compiler. This will require that your program contains the following statements.

```
...
INTEGER*4 IX(30)
...
...
CALL RSORT@(IX,X,30)
...
```

The `INTEGER*4` array `IX` contains the order, that is `X(IX(1))`, `X(IX(2))`, ..., `X(IX(30))` are in order. You may wish to convince yourself of this adding by the statement

```
WRITE(*,'(15F5.1)')(X(IX(J)),J=1,30)
```

(You should note here is that the 3rd argument of the builtin routine `RSORT@` needs to be of type `INTEGER*4`. This can be achieved by using the `-intl` argument of FTN77.)

3. Newton's method for numerically solving $f(x) = 0$ involves choosing a starting point x_0 and successively generating

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \quad k = 1, 2, \cdots.$$

The iteration is stopped when $|f(x_k)|$ is sufficiently small. For the function $f(x) = \sin(x) - x/2$ we note that $f(\pi/2) = 1 - \pi/4 > 0$ and $f(\pi) = -\pi/2 < 0$ and thus, since f is continuous, f has a root between $x = \pi/2$ and $x = \pi$. Write a program which implements Newton's method in this case using $x_0 = 2$ as the starting point in order to accurately compute the root.

4. Create a program which uses the truncated Maclaurin approximation

$$\sinh(x) \approx \sum_{i=0}^{10} \frac{x^{2i+1}}{(2i+1)!}$$

to estimate $\sinh(0.1)$, $\sinh(1)$ and $\sinh(10)$ and compare the results with that obtained when you use the FORTRAN intrinsic function `SINH`.

9.4 Lab. Session 4

1. The following short program is designed to illustrate the use of FUNCTION subprograms and SUBROUTINE subprograms. Type the following short program listed below.

```
      PROGRAM DIFFER
C
C ::: this program compares the value of the exact derivative
C ::: of a function with a finite difference approximation
C
C ::: in the following t is the increment used
C ::: compar is a routine for doing the comparison
      T=1E-1
      CALL COMPAR(T)
      T=1E-3
      CALL COMPAR(T)
      T=1E-5
      CALL COMPAR(T)
      END
      SUBROUTINE COMPAR(H)
C
C ::: compare at the 5 points 0,0.25,0.5,0.75,1.0
C ::: h is the increment used
      WRITE(*,'(/A,E12.4/)')'h=',H
      DO 10 I=0,4
C
C ::: set x and evaluate functions at x
C ::: f=function, df=exact derivative,
C ::: dif=finite difference approximation function
      X=0.25*I
      VF=F(X)
      VDF=DF(X)
      VDIF=DIF(X,H)
C
C ::: compare and output
      ERR=VDF-VDIF
      WRITE(*,'(A,5E12.4)')'x,f,df,dif,err=',X,VF,VDF,VDIF,ERR
   10 CONTINUE
      END
      FUNCTION F(X)
      F=X**3*SIN(X)-EXP(-2*X)
      END
      FUNCTION DF(X)
      DF=X**3*COS(X)+3*X**2*SIN(X)+2*EXP(-2*X)
      END
      FUNCTION DIF(X,H)
      DIF=(F(X+0.5*H)-F(X-0.5*H))/H
      END
```

If you have named the file `subprog0.for` then compile, link and run the program with the command

```
ftn77 subprog0 -undef -lgo
```

When the above works try running the program again with the command

```
ftn77 subprog0 -undef -lgo -br
```

The `-br` option runs the program in a debug mode. The intention here is to use the debug facilities in order to illustrate how different program units (the named program named as `DIFFER`, the subroutine subprogram `COMPAR` and the function subprograms `F`, `DF` and `DIF`) interact. Once you have entered the debug mode you should press the `F4` key 2 times and you should then step through the program by repeatedly pressing the `F7` key. You should observe the flow of the program and how the various quantities change. At various stages you can try pressing the `F5` key to see the results of the `WRITE` statements. To exit this mode at any stage you should type the `SHIFT` and `F1` keys together.

2. The Trapezoidal rule and Simpson's rule are two schemes for approximating the definite integral

$$I = \int_a^b f(x)dx \ .$$

The Trapezoidal rule approximation using n strips is

$$T_n = h\left(\frac{f(a)+f(b)}{2} + \sum_{i=1}^{n-1} f(x_i)\right), \quad \text{where } h = \frac{b-a}{n} \text{ and } x_i = a + ih.$$

The Simpson's rule approximation using $n = 2m$ strips is

$$S_{2m} = \frac{h}{3}\left(f(a)+f(b)+4\sum_{i=1}^{m} f(x_{2i-1}) + 2\sum_{i=1}^{m-1} f(x_{2i})\right), \quad h = \frac{b-a}{n}, \quad x_i = a + ih.$$

Write subprograms which compute respectively the Trapezoidal rule approximation and Simpson's rule approximation for a general function f with arguments corresponding to a, b and n. Test your routines in the case $a = 1$, $f(x) = 1/x$ and using various values for b and n. You should compare your computed values with the values computed by the intrinsic function `LOG` since

$$\log_e b = \ln b = \int_1^b \frac{1}{x} dx \ .$$

3. Write a program for the task given in question 7 of the January 1996 examination paper. You should output all the binomial coefficients for $n \le 20$.

A Using the FTN77 Fortran Compiler on the networked PC's and setting the environment

A.1 Getting the FORTRAN batch file

To obtain the batch file called `startfor.bat` which is to be used for the setting up of the environment for the use of the Fortran compiler FTN77 you should type the following:

```
net use r: /d
net use r: \\topaz\vol\pcpack\ma\staff
attrib -r startfor.bat
copy r:\e\u\startfor.bat
attrib +r startfor.bat
```

In the above you may obtain error messages with the 1st and 3rd lines typed as you may not have an `r:` drive to dismount and you may not already have a file called `startfor.bat`. The error messages are harmless. The 2nd line must execute successfully for the 4th line to work. The 5th line is to partially protect the file `startfor.bat` from accidental deletion.

If the above works satisfactorily and you never require to update the DOS batch file `startfor.bat` then you should never need to type the above 5 lines again.

A.2 Setting up the FORTRAN environment

Type the following:

```
startfor
```

If the compiler is already set up on the `d:` drive of the PC then the set up should be completed quite quickly. If however the batch file detects that the compiler is not already set up on the `d:` drive of your PC, which may be the case the first time that you use it on the PC, then it invokes another batch file which attempts to do the installation. This is unlikely to take less than about 1 minute. If this happens then you will be instructed to type the command again in order to complete the installation.

If for some reason the installation on the `d:` drive is unsuccessful, for example because you do not have a `d:` drive or the `d:` drive is write protected or there is insufficient space available on the `d:` drive or ..., then you can still use the compiler by typing

```
startfor /nod
```

This enables you to use the compiler directly from the network. You can use this command even if the `d:` drive set up would be successful. However it is likely that the response of the PC to any of the FORTRAN commands will be slower.

A.3 DOS History Mechanism

DOS has a history mechanism which enables you to use the up-arrow and down-arrow keys to help you quickly re-use and edit previous commands without the need for completely re-typing them. To enable you to make use of this you should type

---

```
doskey
```

Thus to summarize the recommended lines to type when you start a FORTRAN session are as follows.

```
startfor
doskey
```

B Editors

The following notes describe briefly some of the commands of the DOS `edit` editor and the `vi` editor which is available on UNIX machines and for which there are versions which also run under DOS.

To learn about an editor you need to use it frequently and thus at best the descriptions given here can only serve as a supplement to the knowledge and experience that you will obtain from actual use. The choice of editor to use is very much a matter of personal taste and my recommendation is that it is better to try to obtain a good working knowledge of one editor than a less good knowledge of several editors. From previous experience of this course most (probably all) students used the DOS `edit` editor which is certainly easier for the beginner than the `vi` editor.

You are of course free to use any other editor which is available to you.

B.1 EDIT — the MSDOS screen editor

This is a fairly 'user-friendly' editor with certain on-line help facilities. However you tend to work faster if you already know a set of basic operations so that you do not need to use the on-line help very often.

B.1.1 Starting EDIT

If you want to create or modify a file called, for example, `zyx.for` then at the DOS prompt you type the following.

```
edit zyx.for
```

If 'special keys' are avoided then everything you type is inserted to a version of `zyx.for`. In fact the starting version, if it exists, is not actually changed until you save what you have done when exiting edit.

B.1.2 Moving around the file

The 4 arrow keys move the cursor. The `PgUp` and `PgDn` move up and down a screen full respectively. The `CONTROL` and `Home` keys together move the cursor to the start of the file. The `CONTROL` and `END` keys together move the cursor to the end of the file. The `Home` key moves the cursor to the start of the line. The `END` key moves the cursor to the end of the line.

The keys `SPACE`, `TAB` or `RETURN` should not be used for moving around the file. The `SPACE`, `TAB` or `RETURN` keys move the cursor position but also add respectively a space, a tab or a carriage return/new line feed combination to a file.)

B.1.3 Exiting edit

Using only the keyboard you should type the `ALT` key (which activates the File menu) and then move to the exit option by using the up and down arrow keys (up twice should go to exit). The highlighted pending action is then acted on when you press the `RETURN` key. You are then asked whether or not you wish to save what you have done. You select your response using the `TAB` key with the action implemented when you press the `RETURN` key.

There are other drastic ways of exiting which results in losing what you might have just done. You can switch the PC off or you can try rebooting the PC by simultaneously typing the 3 keys `CONTROL ALT DELETE`.

B.1.4 Special keys

For exiting edit and also for various editing operations EDIT uses the following keys. `ALT`, `CONTROL`, the 4 arrow keys, `TAB` for selecting various options and `RETURN` for acting on selected options. Operations involving deletion and moving lines from one place to another (cut and paste) also makes use of the keys `SHIFT`, `DELETE` and `INSERT`. Several operations require simultaneous use of 2 keys. The `ESC` key is used to remove help screens.

B.1.5 Insert and replace modes

Unless you change the default setting you are in insert mode. To change to replace mode (i.e. what you type replaces what was under the cursor position) type `CONTROL V`, i.e. both keys together. To return to the insert mode type the `INSERT` key.

B.1.6 Deleting text and moving text

Action involving one line only

`CONTROL Y` deletes the current line from the screen and puts it into the buffer. If you want to put this back somewhere then you should do this next, i.e. you should move to where you want to insert it and then do one of the actions described below.
`CONTROL Q Y` deletes the current line from the cursor position to the end of the line putting a copy in the buffer. Thus this only differs from the above if the cursor is not at the start of the line.

Action possibly involving several lines

Marking the text

You can mark the text by holding the `SHIFT` key down and then moving the cursor from your current position to another position. Any of the movement commands can be used. The part of the file between where you start it to where you end it is highlighted.

Action

`SHIFT DELETE` Deletes the highlighted text from the screen and puts it in a buffer (i.e. it moves it to the buffer). If you want to put this back somewhere then you should do this next, i.e. you should move to where you want to insert it and then do one of the actions described below.

`SHIFT DELETE` Inserts the current buffer before the current cursor position.

`DELETE` This removes the highlighted part forever !

`CONTROL INSERT` This keeps the highlighted text on the screen and puts a copy of it in the buffer.

B.1.7 Other editing operations — Substitutions (possibly globally)

Use `ALT` and the right and left arrow keys to activate the Search menu. Then use the up and down arrow keys to move to Change followed by the `RETURN` key. You are then asked to enter the text you want to change followed by what you want to change it to. After typing the first part you must enter the `TAB` key to get the next part otherwise it will be assumed that you are replacing the first part with nothing (i.e. no characters, a deletion). Continuing using `TAB` to select the type of action to be performed. The `RETURN` key activates what you have selected.

B.2 Using the Vi screen editor

This is the standard Unix screen editor, i.e. you should find implementations of it on all UNIX machines. At Brunel we also have implementations of it on PC's operating under DOS. The version of Vi which you get when running the commands used to set up the FTN77 compiler is almost equivalent to the full UNIX version. There is however a lesser PC version available on the network which you will get if you have not used the FORTRAN set up commands. You can distinguish the compilers by the colour of the screen. The full version gives you a blue screen whilst the other version gives a black screen. The lesser version lacks many of the commands in the full version and has one major difference in the key required to end an insert mode. The usual character is the `ESC` key but on the lesser version it is the `END` key. This is an important point to note and causes great frustration to people who wish to exit the editor but find that everything that they type just adds to the file. Thus if you are stuck try the `ESC` key (or possibly the `END` key) in order to return to the command mode.

The Vi editor has been around for many years now and it still must be regarded as one of the more powerful editors in common use. I have used it for many years and like it but this is not a universal opinion. The power of Vi is partly at the expense of its lack of user frendliness, i.e. you need to know what you are doing to use it and if you do not then it does not try to help you. Another feature of Vi which you need to be aware of is that it does not make a backup of the file being edited. Thus it is possible to easily lose several hours or days work if care is not taken with some of the editing commands. If the files are kept in your home directory or a subdirectory of this then the computer centre regularly makes backups of these files and thus you can probably get a version of any destroyed file which existed about 1 day previously.

B.2.1 Starting Vi

To invoke the screen editor type the command

```
vi
```

followed by the `RETURN` key or

```
vi name
```

followed by the `RETURN` key where `name` is the name of a file to be created or edited.

B.2.2 Basic commands

In the following only a few of the Vi commands are given. You can obtain more information from certain computer centre handouts.

The modes and exiting Vi

When you are using Vi you can be in **command mode** or **insert mode** (sometimes called input mode or append mode). When you first enter the editor you are in command mode and you must return to this mode in order to exit the editor. In the full version of Vi the `ESC` key returns you to the command mode but, as already indicated, the `END` key is used on the lesser PC version. If you are not sure which mode you are in then try typing `ESC`. If you are already in the command mode then on most computers the bell will sound but your file is not affected. On the full PC version it should be set up so that when you are in the insert mode the word `Input` appears in the bottom right hand corner. When you leave the insert mode this word disappears from the corner of the screen. This can also be set up on the UNIX version but it is not usually the default set up on a system.

When you are in the command mode all commands entered must be followed by pressing the `RETURN` key.

To exit Vi when at the command mode saving the file being edited you can type `ZZ` or `:x` . To exit Vi when at the command mode abandoning everything you can type `:q!` .

Entering insert mode

`a` To append text starting at the current cursor position.

`i` To insert text at the current cursor position. This is thus the same as the `a` command except that the text to be entered is translated 1 cursor postion to the left of what the `a` command would do.

`ESC` The usual key to end an append or insert command. Until this is typed all the keys typed become part of the file.

Moving around a file

You can use the 4 arrow keys and the `RETURN` or `ENTER` key to move around the file.
On the PC versions the `PgUp` and `PgDn` key usually work but not on the UNIX version.
In all cases you can move around the file by pressing the `CONTROL` key and certain characters together.

`CONTROL d` Scroll down (i.e. forward) 1/2 a screen full (about 12 lines)

`CONTROL f` Go forward 1 screen full

`CONTROL u` Scroll up (i.e. back) 1/2 a screen full

`CONTROL b` Go back 1 screen full

To determine where you are in a file type `CONTROL g`.

Other movement commands follow.

`b` go back 1 word.

`26b` to go back 26 words. Any number can be used here.

`w` go forward 1 word.

`45w` to go forward 45 words. Any number can be used here.

`1G` go to line 1.

`23G` go to line 23.

`G` go to the last line.

`0` go to the start of the current line.

`$` go to the end of the current line.

Deleting text

`x` delete character under the cursor.

`7x` delete 7 characters starting at the cursor position.

`23dw` delete the next 23 words.

`D` delete remainder of line starting at cursor position.

`dd` delete current line.

`88dd` delete 88 lines starting at current line.

`u` undo last command (useful if you have done an incorrect deletion !!).

NOTE: If you use the `x`,`dw`,`D` or `dd` commands then the deleted text is actually put in a buffer which can then be put back into the file at the current cursor position by typing `p`. The buffer only however stores the last deletion.

B.2.3 The colon(:) commands (not available in the lesser PC version of Vi)

There are numerous commands of this type which is what gives Vi some of its power. Only a few of these commands follow.

`:/string/` print the first occurrence of `string` from the current cursor position.

`:g/string/p` print every occurrence of `string` in the file.

`:v/string/p` print every line which does not contain the letters `string`

`:s/string1/string2/` substitute the first occurrence of `string1` in the current line with `string2`

`:s/string1/string2/g` substitute every occurrence of `string1` in the current line with `string2`

`:11,67s/string1/string2/` substitute every occurrence of `string1` in lines 11 to 67 with `string2`

`:1,$s/string1/string2/` substitute every occurrence of `string1` in the file. (`$` denotes the last line in the file).

`:.,$s/string1/string2/` substitute every occurrence of `string1` from the current line to the last line in the file. (`.` denotes the current line in the file).

`:2,7d` deletes lines 2 to 7.

`:7,11j` join together lines 7 to 11.

`:r prog` input the contents of file `prog` after the current line.

`:!command` execute an operating system command called `command` without leaving Vi.

`:w file99` store the current contents of the editor as the file called `file99`

`:w! file99` store the current contents of the editor as the file called `file99` overwriting any existing file of that name.

C Other Compilers and FORTRAN Tools

C.1 FORTRAN on the SUN's

If you wish to use FORTRAN on the SUN machines you will need to type

```
use sunpro
```

once per login session. You can then compile programs with the `f77` command. If the FORTRAN file is called `prog34.for` then this can be compiled with the command

```
f77 prog34.for
```

If the compilation is successful then you will be default create the file called `a.out` which you can run by typing

```
a.out
```

C.2 FTNCHEK and F2C on the PC's

FTNCHEK and F2C can run on both UNIX systems and DOS systems but in these notes only the DOS version is described.

FTNCHEK is a program for analyzing FORTRAN source files in order to determine possible inconsistencies which may not be detected by a compiler. A DOS batch file which uses it with certain command arguments set is called `f77check.bat` . If the FORTRAN file is called `prog34.for` then this can be checked by this program by typing

```
f77check prog34
```

F2C is program for creating a C-language source program from a FORTRAN language program. Thus this program together with a C-compiler enables FORTRAN programs to be effectively compiled and run on machines which only have C-compilers. To use this on the PC's and also to use the GNU `gcc` C-compiler you should first type

```
setgcc
```

Then for the file `prog34.for` the command

```
f77gcc prog34
```

runs the `f2c` command to create a C-source file and then its compiles the C-file with the `gcc` compiler. On the PC's this is much slower than using the FTN77 compiler.