The University of Sydney

Name:

Phone:

Email

Applied Mathematics 3 MATH3076/3976

Mathematical Computing Part 1

Fortran90/95 Course Notes

March 5, 2012

R.W. James and D.J. Ivers

School of Mathematics & Statistics
University of Sydney

Contents

1. INTRODUCTION / OVERVIEW

Preface

These notes are not self-contained. They are an accompaniment to information in the fortuts and lectures. Incomplete examples and notes herein will be completed in lectures.

PART 1 Fortran90 Programming

Some of Part 1 is automated so that there will be less need for standard lectures, but more time devoted to computer tutorials. An organised student of average competence should be able to complete the course in about 4-5 hours per week, composed of lecture/discussion, text reading, analysing the problems and manually preparing detailed answers, doing the tutorials and problems. The amount of time spent in the computer labs can be greatly decreased by careful planning of answers and programs, and manually checking them before typing them in and attempting execution.

Much of the work may be done in your own time, and you are welcome to use the Mathematics Computing Laboratories 610, 729-30, 705-6 outside the timetabled tutorials provided the rooms are not booked by another class. If other tutorials are in progress, but there are spare terminals, consult the tutor as to whether you might use them. During semester, the labs are open from about 9 am to 5 pm, and one lab, usually 730 remains open longer. The labs are not open on weekends or public holidays. Consult the notices on the lab doors for more details.

1.1 Course Outline and References

Outline

The aim in Part 1 is to work through a set of interactive lessons, the *fortuts*, read related sections from a standard text, and complete supplementary problems that show the potential of, and teach the basic rules of, one particular computer programming language, Fortran90 (or Fortran95 which is much the same). Scientific programming languages like Fortran enable automation of various tasks, and Fortran is especially suited for large-scale numerical computations. The references are described below, and most are available in the SciTech Library, Jane Foss Russell Building Level 1, in one case on reserve for reading in the library, or short term borrowing — see a Librarian.

One of the advantages of computer-based courses of this type, is that you may work to some extent at your own pace, subject to meeting due dates for assignments and being approximately coordinated with any lectures etc. MAKE SURE THAT YOU LOG ON REGULARLY, SEVERAL TIMES A WEEK AT LEAST, AND CHECK COURSE MESSAGES, which will recommend a minimum rate of progress to aim for, advise corrections to exercise sets, etc. You will be advised of any unread messages each time you log on, and you can inspect the messages at any time via the command

mess

entered at the UNIX % prompt.

Books

There are numerous F90 books on the market, but only a small number are recommended as listed/described below.

Those wishing to purchase a reference book, should consider the following by Chapman:-

[Ch9503] S.J. Chapman, *Fortran95/2003 for Scientists and Engineers* (McGraw-Hill 2008). This is the third edition of reference **[Ch90/95]** below.

[ChInt] S.J. Chapman, *Introduction to Fortran90/95* (McGraw-Hill B.E.S.T. series 1998). This is a cut-down relatively inexpensive version of reference **[Ch90/95]** below. [Out of print in 2005, and only available second-hand.]

[Ch90/95] S.J. Chapman, *Fortran90/95 for Scientists and Engineers* (McGraw-Hill 1998). [2 copies in the SciTech Library (1 in SciTech Reserve). A much more advanced coverage.

Alternatives to the above are:-

[NLInt] L.Nyhoff, S. Leestma, *Introduction to Fortran90* (Prentice Hall 1999). This is a cut-down version of reference **NL** below – 300 vs 1000 pages.

[NL] L.Nyhoff, S. Leestma, *Fortran90 for Engineers and Scientists* (Prentice Hall 1999). 2 copies in the SciTech Library.

Other useful F90 books:-

[EPL] T.M.R. Ellis, I.R. Philips, T.M. Lahey, *Fortran90 Programming* (Addison-Wesley 1994). 1 copy in the SciTech Library, and 1 in Fisher.

[BB] R.J. Barlow, A.R. Barnett, *Computing for Scientists: Principles of programming with Fortran90 and C++*, (Wiley 1998). 1 copy in the SciTech Library. This gives a comparison of F90 and C++, but is not a standalone introduction or reference.

Some Fortran77 books may be useful, but should not be purchased. There are very significant differences between Fortran77 and Fortran90/95, which will become apparent from the lectures and the interactive fortuts. One such useful text is:-

[Etter] D.M. Etter *Structured Fortran77 for Engineers and Scientists* (Benjamin Cummings). This out-of-print book is NOT recommended for purchase since it is based on the out-dated Fortran77 Standard. It is recommended for library use on the grounds of its readability and good selection of context problems. There are about half a dozen copies of various editions of Etter in the SciTech Library.

Web Sites/Compilers etc

- A free F95 compiler called g95 is available for various operating systems at http://www.g95.org .
- A free Silverfrost FTN95 Fortran Compiler for Windows (Salford FTN95) is available at http://www.silverfrost.com .
- A free F95 compiler called gfortran is available for various operating systems at http://gcc.gnu.org
- A great deal of Fortran related material is on the web, some links to which can be obtained through, for example, Computer Transition Systems, http://www.cts.com.au
- A very useful source for computer jargon, programming concepts, history etc, is the Free Online Dictionary of Computing, from Imperial College, http://foldoc.org/
  (For humour, see the entries under *Real Programmers*, and *Story of Mel*.)
- A great deal of free scientific subprograms is available via the Guide to Available Mathematical Software at http://gams.nist.gov

1.2 Doing the F90 Exercises and Assignment

1. You should attempt most of the exercises labelled F90 EXERCISES. For assessment, you must hand in those exercises specified in the relevant electronic course message(s). A sample, possibly all, of your attempts will be inspected by the marker, and credit given for satisfactory attempts. In addition, certain questions selected retrospectively will be marked in detail. Don't hand in any additional exercises nor those from the fortuts. On or before the due date, submit them electronically using the relevant command

   handin mc3fp fpexs.ans (MATH3076)
   handin mc3fpa fpexs.ans (MATH3976)

   where fpexs.ans (or other name of your choice) is the file in which you have stored your answers.

2. Try to do the exercises at a steady pace throughout the course. Try to proceed at least as fast as the recommended course schedule. Last minute attempts to complete assignments may be hindered by crowded labs, restricted access, and system breakdowns.

3. In the F90 EXERCISES, **questions marked by a dagger † require processing of a computer program**. For questions involving programs, use meaningful file names such as setnqm.f90 for Fortran programs, setnqm.dat for input data files, and setnqm.res for results (or other names related to each question). For assessable questions, copy these files, and other parts of the answers, into a single answer file, fpexs.ans say. Avoid the use of brackets in file names. Thus, for example, use a filename like set1q7i.f90 for Set 1 Question 7 Part (i), not set1q7(i).f90 . For non-computer parts of questions just type a brief answer directly into fpexs.ans. For marker convenience and concomitant loss of marks, wherever possible **use variable-names and other notation as specified in each question**. It is also important for marker convenience that the answers be presented in a clearly demarcated and uniform way. Therefore, obtain a copy of a suitable template answer file by using the command

   ```
   cpi  $mc3fp/fpexs.ans  .
   ```

   (Note the last **.** which stands for your current directory.) Your initial copy, called fpexs.ans, shows an appropriate layout, but does not necessarily include any correct answers. It does however include a half-baked program for the triangle-coordinates question of Set 1. To save typing, cut and paste this incomplete program into a file, e.g. triangle.f90, set1q9.f90, or similar, edit and complete it.

4. Those **questions marked by an asterisk** [A] **are Advanced** level; non [A]ed questions are Normal and Advanced.

5. **Non-computer questions or parts not marked by a † should be answered without running a program**, although you may use a program to check your answers. So it is insufficient to just write, run and hand in a program and answer for a non-computer question. **Marks may be deducted if brief but sufficient reasoning is not shown for non-computer answers. It is insufficient to just state the answer.**

6. You may find it useful to also do additional exercises chosen from the references.

7. **Feedback**: Answers to the exercise sets in these notes are not published. However you may ask for help on any question, including assignment questions, preferably during tutorials. Also, you may ask for any of your answers to be inspected prior to handing in, but due to time constraints this must be done during formal tutorials.

1.3 Getting Started (mess and fortuts on rome)

See the document of the same name, available in the first lecture, ~~or from the slots outside Room 623~~.

1.4 Notes on Windows, Editing, Key Definitions and Unix

Students unfamiliar with UNIX and/or the Mathematics computing laboratories, should use the notes below to do the exercises that follow on p.8.

1. Manipulating Windows

To **activate** or **select** a window, move the mouse pointer (i.e. 'point the mouse') onto the window and press and release the left mouse button (i.e. 'left-click'). The window frame will be highlighted to indicate active. Usually, a window must be active before you can operate in it. If the required window is fully obscured, then in the visible window select (i.e. left-click) the '**Window Manager**' button [-]

in the top left corner, then select '**Lower**', or select '**Workspace', 'Shuffle**'. The '**Workspace**' menu is also selectable by left-pressing outside any window.

To **reshape** a window, place the mouse pointer over any part of the window frame, so the pointer changes shape. Then move the mouse whilst holding down the left button (i.e. 'left-drag') in the required direction. Note that the window's corner pieces may also be left-dragged.

To **move** a window, left-drag the window's title bar, which is immediately below the top of the window frame.

To **close** a window select (i.e. left-click) it, and press **[Alt][F4]**. Alternatively, right-click on the window's pane, or left click on the window's '**File**' menu (if it exists), and select **'Close'** or similar.

To **iconise** a window, left-click the '**Minimise'** button [•], the second one in from the top right corner. To **un-iconise**, fast double-left-click the icon.

2. Editing a File using Nedit

To **open** an Nedit window for the file called name, use the command

ned name

at the UNIX % prompt. Alternatively, but much less directly, select the '**Applications**' menu in the '**Session Manager**' window, and select '**Nedit'.** Then select '**File**'. If the file exists, select **'Open',** otherwise select '**New**'. To **save** a file under its current name, select '**File', then 'Save**'; to save under a new name, select '**File**', then '**Save As'.**

To **select** text for cutting, copying or pasting, left-drag the mouse over the required text and release the mouse button. Alternatively, press the arrow keys whilst [SHIFT] is depressed. Rapid double- or triple-left-clicking selects the word or line under the mouse pointer. (Yet another way is described in the section on scrolling below.)

Most commonly used functions in Nedit are obtained by selecting the pop-down menus '**File**', '**Edit**' and '**Search**', and then selecting the required function. Editing can be significantly sped up by learning shortcut key definitions, some but not all shown in the pop-down menus, e.g.:-

[CONTROL][Z]	Undo previous command; to correct blunders.
[CONTROL][F]	Find a string pattern
[CONTROL][G]	Go to next occurrence of the pattern
[CONTROL][R]	Replace a string pattern
[CONTROL][A]	Select all the file
[CONTROL][C]	Copy selected text
[CONTROL][X]	Excise (cut) selected text, and make a copy
[CONTROL][V]	Paste in copied text
[CONTROL][S]	Save current file, leaving it open for more editing
[CONTROL][Q]	Quit editing, optionally saving the file.

3. Scrolling, Copying and Pasting In and Between Windows

Scrolling is accomplished in various ways, including:-

a) line-by-line by left-clicking to set the insertion point near the top or bottom of the window, then using [Up-arrow] or [Down-arrow] keys (e.g. in Nedit, but not in Xterms);
b) placing the mouse pointer over the scroll bar just inside the left or right window frame, depressing the middle mouse button and then dragging the scroll bar up or down as desired;
c) line by line by middle-clicking on the up or down pointers (if they exist, e.g. when in Nedit) at the top and bottom of the scroll-bar's slide panel.

When using the middle mouse button, be very careful not to press it accidentally whilst the insertion point is inside the text area of a window, since this may paste in unwanted previously selected text (see the paste section below). In Nedit, such a blunder can be reversed using [CONTROL][Z] as described earlier.

To **copy** and **paste** from one window to another, e.g. from an Xterm window to an editor window, or from one part of a window to another part of the same window, (i) select the text by left-dragging the mouse or, in Nedit, using [SHIFT][arrow] keys as described earlier, (ii) reposition the mouse pointer at a new insertion point in the same or other window and middle-click to paste in the text. Note the insertion point for Xterms is fixed by default at the bottom Unix % input prompt.

If there is a lot of text, or if you need to scroll the window to select the desired text, then (i) left-click the start of the text, (ii) move the mouse pointer to the end of the text (using the mouse and scroll bar or scroll pointers if need be), (iii) right-click to highlight and copy the text, (iv) move the mouse pointer to the new insertion point and middle-click to paste. NOTE: in some applications, including Nedit, use [SHIFT]-left-click instead of right-click to mark the end of the range.

4. Command Line Editing

The following key definitions allow editing of command lines input at the Unix % prompt, and inside some applications (e.g. MATLAB, and Splus, but not Nedit where different meanings may apply).

[Up-arrow], [Down-arrow] step through previous input commands
[Left-arrow], [Right-arrow] move cursor to new insertion point
[Backspace] Delete character to left of insertion point
[CONTROL][D] Delete character at insertion point
[CONTROL][A] Move cursor to start of the command line
[CONTROL][E] Move cursor to end of line
[CONTROL][U] Undo (delete) entire line
[CONTROL][K] Kill (delete) right to end of line
[CONTROL][W] Wipe (delete) left to start of line

5. Basic Unix file system commands

`ls`	list names of files in current directory
`ll`	(short for `ls -l`) long list; as for `ls` but with dates, sizes etc
`rmi name`	(short for `rm -i`) remove file called name
`cpi name1 name2`	(short for `cp -i`) copy file name1 to file name2
`mvi name1 name2`	(short for `mv -i`) change filename name1 to name2
`cat name`	list file called name on the screen
`lpr name`	print file called name on nearest printer
`lpr -Pc705 name`	print file name on printer in Carslaw lab xxx, etc
`lpq`	list the print queues
`lpq -Pc705`	list the queue on printer in Carlaw 705, etc
`xt name`	open a new Xterm window titled name
`passwd`	change your password
`wc name`	(word) count the number of lines, words and characters
`cd name`	change to directory name
`mkdir name`	create directory called name
`rmdir name`	remove directory called name, if it is empty
`rm -r name`	remove directory called name, and all files in it
`!xxx`	retrieve the most recent command starting with letters xxx

The wild-card characters `?` and `*` match single and multiple characters respectively. E.g. '`rmi *.o`' will remove all files with extension '`.o`'. The use of the '`i`' option in `rmi`, `cpi`, `mvi` will prompt you for confirmation to safeguard against removing or overwriting wanted files.

INTRODUCTORY EXERCISES ON WINDOWS, EDITING, AND UNIX

1. Manipulating Windows

After logging on to your computer you may see various windows with titles like**'Session Manager'**, '**Console**' and '**Xterm**'. Iconise the '**Console**' window by selecting [Dismiss]. Practise selecting the various windows one after the other by pointing the mouse and left-clicking.

a) Practise changing the shape of the '**Xterm**' window by left-dragging its frame, including left-dragging the corner handles other than horizontally or vertically.

b) Move the '**Xterm**' to the centre of the screen by left-dragging its title bar.

c) Close the '**Xterm**' by first selecting the '**File**' menu, and '**Close**'.

d) Open a new '**Xterm**' by selecting '**Applications**' in the '**Session Manager**' window, then '**Xterm**'.

e) Iconise the '**Xterm**' window by left-clicking the [•] button near the top right corner. Un-iconise it by fast double-left-clicking the icon.

1. Editing a File

a) Use Nedit to open an empty file called called 1st_edit.txt. Type in :-
Every good student deserves fruit.
Maths is beaut.
Maths Computing is even beauter.
To heck with HECS!

b) Be sure to press [RETURN] after the last line. Note the qualifier 'Modified' in the windows title, indicating changes have been made but not yet saved. Save the file without closing it, noting the removal of 'Modified'.

c) Practise moving the cursor around the window by pressing the [←], [↑], [→] and [↓] keys.

d) Change a few words, e.g. change 'fruit' to 'rest', 'beauter' to 'better'. In these cases, use the mouse and arrow keys to position the insertion point for the changes, then use [BACKSPACE] and retype. Save the file again.

e) Select the '**Xterm**' window and enter the Unix command **ls** for listing your directory. Check that 1st_edit.txt is there. Depending on how Nedit is setup, you may also see a backup copy called 1st_edit.txt.bck which would contain the version of 1st_edit.txt prior to your last save.

f) Activate the 1st_edit.txt window. Move lines 2 and 3 to the end of the file by the following steps:-
   i) Select lines 2 and 3.
   ii) Cut and copy them using the '**Edit**' menu or [CTRL][X].
   iii) Place the insertion point at the end of the file – just move the mouse pointer anywhere down below the last line and left-click.
   iv) Paste in lines 2 and 3 using the '**Edit**' menu or [CTRL][V].

g) Move the last line to the top as follows. Triple left-click on the last line to select it, move the mouse pointer to the beginning of the file, and [SHIFT]-middle-click to move (i.e. cut and paste) the line.

h) Instead of saving the edited file, select **'File'** and **'SaveAs'**. Enter the file name 2nd_edit.txt and select [OK]. Notice that the title of the Nedit window has been changed to reflect the new name.

i) Use **ls** to list your directory in the **'Xterm'**, and inspect the output.

j) Select any range of text in the **'Xterm'** by left-dragging. Move the insertion point to anywhere in the 2nd_edit.txt window, and middle-click to paste in. Inspect the result.

k) Use [CTRL][A] to select the entire file; then select the **'Search'** menu, or use [CTRL][R], and replace all occurrences of Math by Mathematic. Check that the replacements worked correctly.

l) Undo the last changes by using [CTRL][Z]. Check the undo.

m) Quit, saving the file. (In practice, if you make a huge blunder, then you might quit without saving. When editing, it is a good idea to save the file regularly, say after about every 5 lines or operations. Then, if disaster occurs, you will probably not lose a substantial amount of work.)

3. Command Line Editing

a) In the **'Xterm'** enter the command
echo me nime is gobblygook
and press [RETURN] or [ENTER]. Observe the resulting output.

b) Retrieve the previous command using [↑]. Edit the command and execute it to produce output like
My real name is whatever.

4. Unix Commands

a) Use **mkdir** to make a subdirectory called junk.

b) Use **cpi** to copy 1st_edit.txt to junk.

c) Use **mvi** to move 2nd_edit.txt to junk.

d) Use **ls** to list your directory and check that 2nd_edit.txt has gone.

e) Use **cd** to change to directory junk.

f) List the directory using **ll**, and compare the file sizes.

g) Use **wc *** to determine the number of lines in the files.

h) Use **rmi *.txt** to remove the files in junk.

i) Change directory back to the top level using just **cd** or **'cd ..'** where the two dots '..' mean go up one directory level.

j) Remove directory junk using **rmdir**.

k) List (i.e. **cat**) the file 1st_edit.txt on the screen.

2. BASICS and PERSPECTIVE

2.1 Basic Concepts

'**programming**' ≡ writing a set of instructions for a computer to interpret and execute. E.g.
Z = X + Y
might mean *add the values stored in variables X and Y, and store the result in variable Z.*

bits, bytes, words - Computers can essentially perform binary tasks (on, off; high, low; open, closed), that can be represented by a
bit i.e. a binary digit 0 or 1
Machine architecture is based on
byte 8 bits
word 8, 16, 32, 64 bits; computer-dependent
On a 32-bit computer, a character is stored in an 8-bit byte, and numbers usually in 4-byte words. E.g.
00101000 ← *the letter C*
00000000000000000000000000000101 ← *the integer 5*
01000001010111000000000000000000 ← *might represent a non-integer like 5.75*

Until mid-1950's most programming was done in *machine language (*aka *machine code),* and *assembly language.*

machine code - User required to specify the bit patterns representing machine operations, e.g. transfer a number from a memory register into a processor register ready for addition, multiplication etc. For example, to calculate Z, where Z = X + Y, given the values of X and Y, in machine code you might have to use instructions like
0010 0000 0000 1100
0100 0000 0000 0101 *PAINFUL!!*
0011 0000 0000 0110

assembly language - Like machine code, a 'low-level' language, but closer to English. Consists of acronyms, that more easily allow one to specify the basic machine operations. The earlier Z-computation might then be
MOV X, R0 ← *Move value in memory address X to processing register R0*
MOV Y, R1 ← *Move value in memory address Y to processing register R1*
ADD R0, R1, R2 ← *Add values in registers R0, R1 and place result in register R2*
MOV R2, Z ← *Move value in register R2 to memory address Z*
Still tedious !
The assembly code is then automatically converted by an *assembler* into binary machine code.

Both machine code and assembler are relatively difficult/cumbersome to use. However, they permit great control over the computer, and therefore job-execution as efficient as you can make it. (Assembler is still used today for large-scale jobs where high efficiency important, to detect inefficiencies and bugs in compilers, and to crack copy-protected codes.) But it became apparent by early 1950's that the expense of designing and debugging programs was comparable to or greater than the expense of running them. Other hindrances existed such as hardware constraints. E.g. low memory - even in 1960's most mainframes were restricted to < 100 Kb central memory. Compare this with the modern PC default of ~100 Mb, and which is rapidly rising. Also, most hardware of the early 1950's did not include floating-point arithmetic or indexing (i.e. arrays like A(i), i=1,...) -– this had to be coded by the user. Being restrained to integer arithmetic and trying to mimic floating point presented difficulties and inconvenience.
4GB

Integer arithmetic is a special case of fixed-point arithmetic, where a fixed number of digits are kept after the point. In integer arithmetic zero digits are kept after the point:-
e.g. in Fortran 3/2 returns a value of 1 - note the large round-off error 50%.

Floating-point arithmetic example: consider $12.34 + 2.1\times10^{-2}$. In f-p arithmetic this is done by:-
(i) putting larger number in some standard form, say $.1234 \times 10^2$,
(ii) aligning the smaller number by moving ('floating') the **.** and adjusting the exponent,
(iii) adding as below.

$$\begin{array}{rll} & .1234 & \times 10^2 \\ & .00021 & \times 10^2 \\ & .12361 & \times 10^2 \\ \rightarrow & .1236 & \times 10^2 \end{array}$$

on a computer that keeps 4 decimal digits.

In Fortran, floating-point numbers are indicated by either a decimal point or exponent form (e.g. 1.23 or equivalently any one of 123E-2, 12.3E-1, 1.23E0, 0.123E1, 0.123E+1, etc).

2.2 'High Level' Languages

The large cost of debugging and coding assembler and machine code prompted the development of easier-to-use languages, the first and most famous of which was:-

FORTRAN the IBM Mathematical **FOR**mula **TRAN**slating System
Aim: Originally just execution efficiency and ease of coding. (In FORTRAN the previous Z computation is programmed simply as Z = X + Y. Note the meaning of "=". (Z = Z + 1 is legitimate.) Good for general scientific numerical work.
Earlier versions not good for class/groups manipulations, character manipulations, some recursive operations, (e.g. a subprogram cannot refer to itself in pre-1990-standard FORTRANs).
For execution efficiency, software availability, and historical reasons, Fortran is still the most widely used scientific language for very large-scale numerical computations.

Began at IBM in early 1955, team of 10, 18-person years of work.
Released 1957 - didn't work properly, but was adopted by many hardware manufacturers, and gained rapid popularity.
FORTRAN II - - 1958
FORTRAN III - 1958
FORTRAN IV - 1962
ANSI (Amer. Nat. Stand. Institute) **Standard FORTRAN66** - 1966
Standard FORTRAN77
Until the mid-1990's, many FORTRANs were still just **F77 +** frills, and many FORTRAN programmers still use just that.
Immediately post 1977, an attempt was begun on a new Fortran standard. For some years labelled Fortran8X, work actually starting in 1978, hoping for X=8. It eventuated as
Fortran90
with compilers and texts starting to appear about 1992. **F90** overcomes many of the criticisms of earlier Fortrans. It borrows from C, Pascal and other languages (e.g. user-defined types, recursive functions, free-format source code), and includes much more (e.g. built-in array operations, and parallel operations with large-scale parallel processing computers in mind). Fortran is now regarded as an evolutionary language with a new standard scheduled to appear every 5 years approximately.
The immediate sequel to F90 was
Fortran95
which is a minor extension of F90, mainly aimed at introducing some HPF concepts (see below). The Applied Mathematics 3 *Mathematical Computing* course uses Fortran90/95 + frills.
~~The current standard~~ (released Novermber 2004) is
Fortran2003
originally intended to be Fortran2000. This is a major extension of F95. The numerous enhancements include greatly increased facilities for object oriented programming, and interoperability with C. The latter allows C programmers systematic access to efficient Fortran code, and Fortran programmers systematic access to low level system routines written in C. Amongst the many other new features are IEEE exception handling, and permitted use of any intrinsic functions in initialization statements.
Another extension to F95 is

High Performance Fortran (HPF)
aimed at facilitating parallel programming on multiple–cpu systems. However HPF has not gained universal acceptance as the best approach to parallel programming in Fortran.

Fortran2008
is the current standard. This is an extension of F2003. The numerous enhancements include new intrinsic functions such as BESSEL_J0, BESSEL_J1, BESSEL_JN, BESSEL_Y0, BESSEL_Y1, BESSEL_YN, GAMMA, LOG_GAMMA, and improved interoperability with C.

[ASIDE: Student terminals are PC's, connected to a multiple-cpu computer called 'rome'. For very large scale projects, e.g. simulating/predicting weather, modelling the Earth's magnetic field, designing Starfighters etc, 'supercomputers' are necessary. One type is the massively parallel processing computer - e.g. some so-called Connection Machines of the early 1990's had up to 64,000 1-bit processors. More common are parallel computers and clusters, comprising a number (up to hundreds) of fast processors. Examples exist at ANU, Sydney (**Silica**: 75 nodes×2 cpu's per node×4 cores per cpu = 150 cpu's = 600 cores) and UNSW. Such machines are useful/necessary because:-
Math models → d.e.'s → discretization → large linear system (dim=10^3 or higher common)
Solving involves many independent operations that can be performed in parallel.
F90 allows things like
DIMENSION X(100), Y(100), Z(100)
Z = X + Y
which evaluates all 100 elements of Z, i.e. Z(i) = X(i)+Y(i), for i=1,...,100 ; and essentially simultaneously on a parallel machine with a sufficient number of processors.]

2.3 Many other languages

BASIC **B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode
Introduced 1963-64 for interactive time-sharing use at Dartmouth College.
Good for simplicity of use and small problem solving.
Dreadful for large problem solving → unstructured hard-to-read programs.
Available on pocket calculators and micros.

COBOL **CO**mmon **B**usiness **O**riented **L**anguage
Together with Fortran, the most widely used language 1960–1990.
Verbose and cumbersome for non-simple scientific work.
Main applications in large input/output, record handling, sorting, payrolls, accounting tasks.

Rivals to Fortran
There have been many attempts to replace Fortran because of inadequacies in the earlier Standard FORTRANs. Amongst the most notable rivals were Algol, APL, PL/1 and PASCAL.

Algol58 **Algo**rithmic **L**anguage - 1958
Aim: Language should be as close as possible to standard mathematical notation and easily readable.
Algol60
Algol68

APL **A** **P**rogramming **L**anguage - 1962
Featured concise notation for many operations with structures such as matrices, e.g. whole arrays could be used as arguments.
Utilised a special keyboard including Greek symbols.
Many devotees in industrial scientific programming until mid 1980's.

PL/1 IBM's **P**rogramming **L**anguage **One** - 1964
Somewhat like a union of older FORTRAN (syntactic style), ALGOL (block structures), COBOL (record handling) incorporating features from each.

Aim: multipurpose language, ease-of-use for beginners, detailed control for experts.
Not very widespread because somewhat unmanageable and complex for most programmers.
A large (Swiss-Army-Knife) language.

PASCAL First designed 1968-70
Algol like but aimed to have advantages over FORTRAN, e.g. a return to simplicity with an ability to handle non-numeric data, but maintain the compilation and execution efficiency of FORTRAN. Originally intended as a language for teaching programming, but has proved more generally successful. Has advantages of

- rigidity - simplifies debugging, catches many latent errors.
- small size - easy to master it all.
- recursive and non-numeric operations.

Decreasing population of users at the present time.

C First implemented at Bell Labs 1972 as improvement to **B**.
B and assembler were used to write the first UNIX operating system. C is the backbone of UNIX systems and is the preferred language of systems programmers.
Increasingly used in general scientific work and is the main rival to Fortran post 1990. Has advantages for manipulation of non-numeric data, communicating internally with the computer and externally (e.g. with scientific instruments).

MODULA, LISP, ADA, C++, Visual Basic, **HTML**, **Java**, **Perl**, **Python**, **Ruby**, etc etc etc
These are amongst the many other 'high-level' languages. Application specific programs/languages, such as Visual Basic and Mathematica are sometimes called 'Fourth Generation' languages. Mathematica, written in C, performs symbolic, e.g. mathematical, manipulations as well as numerical, and is one example of so-called *computer algebra* programs. Visual Basic converts user operations on a screen to BASIC commands, to produce Windows interfaces and associated manipulations.

2.4 Why Fortran for this course?

Beware the 'first/one language syndrome' - i.e. don't get hung up on just one language for all occasions. Fortran is not intended for doing computer algebra, writing web pages etc. And C and related languages are the choice for systems programmers, and people who need detailed control over bits and bytes of data, and close interaction with the overall operating system. Usually each language has some advantages and disadvantages over another. For example, Fortran has built-in complex arithmetic, and array operations, whereas C and Pascal require special constructs. Fortran is easier to learn than C, and there are more numerical scientific subroutine packages available, especially free ('public domain') ones. Post 1990 an increasing number of C versions of such packages have appeared.

Linking object code from different languages is standard today.

Various reasons for choosing Fortran for this course:-
(i) Reasonably easy to learn as a first language.
(ii) Teaches good structured programming techniques that are readily applied in other languages.
(iii) Fortran is the language of choice for most high performance **numerica**l work (but is not often used outside scientific and engineering applications)
(iv) Many scientific subroutine packages available, and many of these free. Individual routines are retrievable through the Internet and World Wide Web.
(v) Numerous post-F90 goodies now available, overcoming previous deficiencies of F77, and offering parallel programming functionality.
(vi) Supplements, rather than duplicates, the languages taught in other courses.

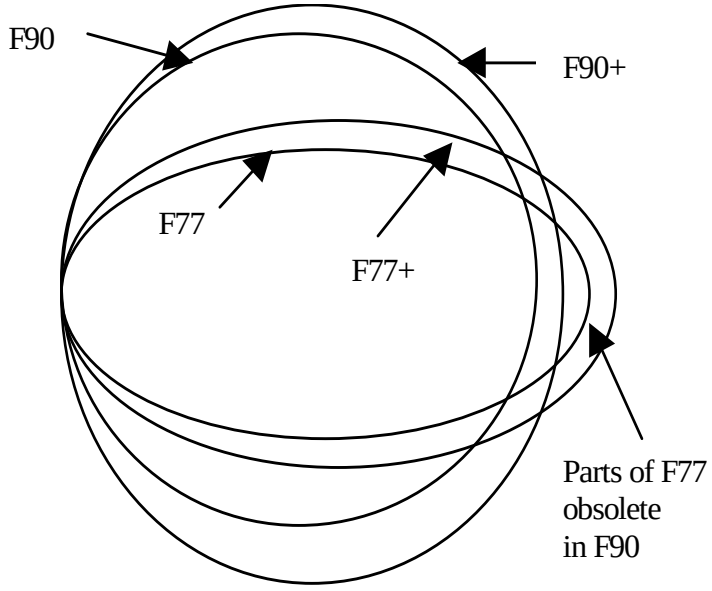
2.5 Recommendation on languages

A good background for many computing tasks in maths-based science and engineering should involve:-

- experience in a scientific programming language (Fortran or C or similar),
- an interface to scientific subroutines and graphics (MATLAB),
- a computer-algebra package (REDUCE, *Mathematic*a, Macsyma or Maple, the last being available separately or as a toolkit in MATLAB).

2.6 Different Fortran Standards

One of the historical advantages of Fortran has been the existence of Standard Fortrans, which facilitated portability of programs between different systems. The current Standard is Fortran95, and the version used on the School of Mathematics system is F95 (although called locally F90) plus frills. The overlapping standards and extensions are shown schematically below.



Beware the differences between standards still in use, and between different non-standard implementations. For example, in the F77 Standard, names of variables, programs etc could not exceed 6 alphanumeric characters (i.e. a-z, 0-9). In the F90/95 Standards 31 characters are allowed, chosen from the alphanumeric and underscore (_). In Apollo Domain F77+ (circa 1990), up to 4096 characters were allowed, alphanumeric, _ and $. In the Compaq (formerly Digital)l F90, up to 31 are permitted chosen from alphanumeric, _ and $. F2003 extends F95 by allowing names up to 63 characters long. Longer names are useful for readability, and extra-long names are useful for programs generated automatically by other programs. Always start names with a letter (rather than a digit or _) since that is standard to F77, F90 and F2003.

3. PROGRAMMING

1. 4 stage program development
2. Mechanics of creating and running programs

3.1 4 stage program development

This should be as close to the Fortran code as possible.

Basic idea is to keep the various aspects of program development separated.

I. Clearly define the problem, including the mathematical formulae to be used.

II. Lay out the coarse logical structure, as 'coarse pseudo-code' (or flowcharts for large programs).

III. Expand result of Stage II to a more detailed structure (i.e. 'refined pseudocode'). Hack it around to get details correct.

IV. Translate to Fortran or whatever, and type up into a source code.

The above staged process has advantage that one doesn't worry about language syntax and logic simultaneously. In Stages II and III, you may use your own style, although make it generally understandable. The pseudocode might, for example be a mixture of Fortran and English. For very large scale problems, flowcharts may be advantageous for Stage II, but they are generally too cumbersome to warrant use in smaller problems, or in Stage III. (For flowchart examples, see **NL**, or **Etter** 1st edition, or especially **ChInt** or **Ch** (who uses them a lot, probably due to the author's background in writing 10^5-lines Fortran programs for radar control installations.). The variables and equations in I and IV should be as close as possible.

Ch9503

Example 3.1 (done in lectures)
Conversion of minutes to hours and minutes. Creation of hrs_mins.f90 for Exercise Set 1, Q6, p.40.

3.2 Mechanics of creating and running programs

See also the course message of the same name. Basically one has to:-
(a) edit a file to contain the *source code* for the program;
(b) compile this code to produce *object code;* go back to (a) to remove *compilation errors* if any;
(c) link the object code to other object code including system input/output routines, to produce *executable code;*
(d) run the executable; go back to (a) to remove *run-time* errors if any.

The commands and procedure for doing this on the School of Maths system are shown below.

Example 3.1 continued

The basic steps for part (i) of Set1 Q6 are:-

(i) In an Xterm at the % prompt
`ned hrs_mins.f90` (*edit the source code*)
to open an Nedit window. Type in the program from Stage IV, using [CTRL][s] to save occasionally. Then, in the Xterm window, compile and run the program via the commands
`f90o hrs_mins.f90` (*compile and link the program*)
`hrs_mins` (*run the executable*)
This will produce a prompt for you to enter the data (`MINS`) at the keyboard, after which the program will print output (`HOURS` and `RMINS`) to the screen.

The modifications needed for parts (ii) and (iii) of the exercise are:-

(ii) `ned hrs_mins.f90`
Comment out the prompt statement in hrs_mins.f90. Then,
`ned hrs_mins.dat`
and place the single number (`MINS`) in the file hrs_mins.dat. Then, recompile hrs_min.f90, and run the executable taking the input from file hrs_mins.dat as follows.

```
f90o  hrs_mins.f90
hrs_mins  <  hrs_mins.dat
```

Unix only

(iii) Comment out the prompt and `READ` statements in hrs_mins.f90, replacing them by

```
OPEN(1,FILE='hrs_mins.dat')
READ(1,*)  MINS
```

then compile, link and run as in (i).

Also PRINT*, WRITE(*,*) WRITE(2,*)

Note case !

Compare with READ*, MINS

3.3 Typographic and variable-naming conventions:-

Fortran is basically case-insensitive, unlike C, MATLAB, *Mathematica* et al. A variety of typographical and naming conventions are used in programming practice, and some of these are illustrated below. Choose your own preference.

(i) Capitals for all except comments and character strings; easy to type. (E.g. used by Etter, fortuts.)

```
PROGRAM HRS_MINS
! Converts minutes to hours & minutes
   INTEGER MINS, HOURS, RMINS
   PRINT*,'Please enter number of minutes:'
```

(ii) Lower case for all except comments and strings; easy to type. (Preferred by some programmers used to case-sensitive languages with lower case keywords, e.g. C.)

```
   program hrs_mins
! Converts minutes to hours & minutes
   integer mins, hours, rmins
   print*,'Please enter number of minutes:'
```

(iii) Capitals for keywords, lower case for all else except comments and strings. (E.g. Chapman; Barlow & Barnett.)

```
   PROGRAM hrs_mins
! Converts minutes to hours & minutes
   INTEGER mins, hours, rmins
   PRINT*,'Please enter number of minutes:'
```

(iv) Variable names and sub-words starting with capitals. This avoids the need for an underscore `_` as a name separator. (E.g. Nyhoff and Leestma.)

```
   PROGRAM HrsMins
! Converts minutes to hours & minutes
   INTEGER Mins, Hours, RMins
   PRINT*,'Please enter number of minutes:'
```

(v) Variable names starting with capital but with lower case prefix to indicate variable type, `i` for integer, `f` for floating-point, etc. (A.k.a. Hungarian notation.)

```
   PROGRAM HrsMins
! Converts minutes to hours & minutes
   INTEGER iMins, iHours, iRMins
   PRINT*,'Please enter number of minutes:'
```

Example 3.2 (done in lectures)
Calculation of triangle area given vertex coordinates (Set 1 Q10 on p.42).

4. VARIABLES and FORMATS

INTEGER, REAL and CHARACTER (introductory) variables and formats are covered in fortuts 1-3. COMPLEX, DOUBLE PRECISION, CHARACTER (more fully) and LOGICAL are covered in fortut8. Various others (e.g. pointers, structures) are not covered in this course.

Type declarations can take one of 2 basic forms, a simple form, or a *double-colon* form. For example,

```
REAL   A          and        REAL :: A
```

both declare A to be a single-precision REAL variable. Similarly,

(-3,3) gives A(-3),...A(3)

```
REAL   A(3)       and        REAL :: A(3)
```

declare A to be a single-precision REAL array with 3 elements. The double-colon form allows an initial value and additional attributes of A to be included in the type declaration. For example,

```
REAL, DIMENSION(3) :: A=1      and      REAL :: A(3)=1
```

both declare A to be an array of 3 elements (all initially equal to 1). There are several other examples of the double colon **::** given in Appendix B on page 73.

The following table summarizes the essential standard-type declarations, and the corresponding formats to be used in READ and WRITE statements. Some of the many variations of these are given in Appendix B, which includes a number of non-standard variations that should be avoided. The ranges shown in the middle column are those of the *IEEE Standard* (Institute of Electronic and Electrical Engineers) for 32 bit representation of numbers. **Warning:** Exceeding those ranges results in *Underflow* and *Overflow*. Floating-point overflow usually results in program termination with an error message. Floating-point underflow usually results in the small numbers being taken as zero. Integer overflow is very dangerous since it is often not trapped by program execution and then produces wrong answers due to large truncation/roundoff. For example, if integer overflow is not trapped, then the statement `PRINT*,2**31,2**32` produces output `-2147483648 0`. The first output has right magnitude wrong sign, and the second is totally wrong. This is because the integers $+2^{31}$ and $+2^{32}$ would require IEEE-like 33 and 34 bit patterns 110⋯0, the first bit being for the + sign. These cannot be stored in 32 bits in IEEE fashion, and excess bits are truncated on the left.

TYPE	Characteristics	FORMAT
INTEGER	32 bits, approx magnitude range $0\text{-}2^{30}$ or about $0\text{-}10^{9}$	Iw where w=no. of cols
REAL REAL(KIND(1E0))	32 bits, approx 7 dec digit arithmetic, so-called single precision, approx range $10^{\pm 38}$	Fw.d (fixed), Ew.d (exponent), Gw.d (general),
COMPLEX COMPLEX(KIND(1E0))	2 REALs, with automatic complex arithmetic	2 REAL formats as above
DOUBLE PRECISION REAL(KIND(1D0))	64 bits, approx 15 dec digit arithmetic, approx range $10^{\pm 308}$	F,E,G as for REAL.
COMPLEX(KIND(1D0))	2 REALS, 64-bits each	2 REAL formats

CHARACTER	Stores strings like 'abcd1234' each character in 1 byte.	Aw – the string is printed left justified in w columns; w defaulting to length of the character variable if omitted
LOGICAL	Values .true. or .false.	Lw – T or F is printed right justified in w columns.

In Fortran, if no type declarations are used then variables starting with any of the letters A–H,O–Z are assumed REAL, otherwise (i.e. starting with one of I–N) they are assumed INTEGER. This implicit typing can be turned off by using the statement

```
IMPLICIT NONE
```

before any other declarations. Then every variable must be declared in a following type declaration. Or, IMPLICIT can be used in forms like

```
IMPLICIT COMPLEX(C,X-Z), DOUBLE PRECISION (D)
```

which declares all variables starting with any one of the letters C,X,Y,Z to be COMPLEX, and those starting with D to be DOUBLE PRECISION.

i.e. most programs

In longer programs IMPLICIT NONE should definitely be used. Forcing every variable to be explicitly declared protects against mistyping of variable names, e.g. having a variable `a1` ("a-one") and mistakenly a variable `al` ("a-el").

4.1 Output Formats for Numbers

(Note that numbers are rounded on output.)

I (Integer) Descriptor:- For w>0, **Iw** prints an integer number right justified in w columns. **I0** uses the minimum number of columns. Example:-

```
PRINT '(I10)', 1234      produces      bbbbbb1234      (b for blank)
PRINT '(I0)',  1234      produces      1234
```

F (Fixed-point) Descriptor: For w>0, **Fw.d** prints a floating-point number right justified in w columns, rounded to d digits after the decimal point. Useful if you know the magnitude of the numbers beforehand. Otherwise "*******" will be printed if you don't allow a sufficient space for any numbers. **F0.d** prints using the minimum width needed to show the number. Example:-

```
PRINT '(F12.4)', -21.12345     produces     bbbb-21.1235
PRINT '(F0.4)',  -21.12345     produces     -21.1235
```

E (Exponential) Descriptor: For w>0, **Ew.d** prints a floating-point number right justified showing d leading digits (mantissa) with a following exponent (4 characters in form Exxx). Usually w should be at least 7 more than d (4 for exponent, 1 for decimal point, 2 for possible "-0" to left of point.) E-formats are useful if you don't know the magnitude of the output numbers beforehand. Example:-

```
PRINT '(E12.5)', -1.234567     produces   -0.12346E+01 or b-.12346E+01
```

(the latter being the F90/95 standard, but many systems display a leading zero). Note the roundoff.

G (General) Descriptor: For REAL numbers **Gw.d** outputs like Fw.d followed by 4 blanks, or like Ew.d (where the 4 blanks are replaced by the exponent Exxx), depending on the magnitude of the number. Like E formats, G is useful if you don't know the magnitude of the numbers beforehand. **Gw** can also be used for INTEGER, CHARACTER and LOGICAL variables, in which case it is identical to Iw, Aw or Lw respectively.

4.2 List-directed Input and Output

The use of `FORMAT` statements and the basics of reading from and writing to files are covered in fortuts ~~13~~. Further information in respect to arrays is in Chapter ~~0~~ herein. Below we discuss 'list-directed' formatting, which is sometimes colloquially referred to as using "free-format".

In the following, `vlist` is any list of variables, e.g. `A,B,IJ`. An asterisk * is used to represent both the default input or output device, and the default format to be used. The default format is chosen to match the types of variables in `vlist`.

```
READ*, vlist          READ(*,*) vlist          READ(5,*) vlist
PRINT*, vlist         WRITE(*,*) vlist         WRITE(6,*) vlist
```

The first 2 entries on each of the above `READ` and `PRINT` lines are equivalent. The * essentially stands for 'default'. Thus, for example, `READ(*,*)` means read from the default input device in default format (i.e. list directed format where the data are just separated by spaces or commas). The default input device is the keyboard, or might be a file if "<" is used to redirect input when the executable version of the program is run. Similarly `WRITE(*,*)` means write to the default output device in list directed format. The default output device is the screen, or might be a file if ">" is used. Furthermore, whilst not part of the Fortran Standard, some implementations of Fortran treat unit numbers 5 and 6 as special. For historical reasons, it is common for 5 to represent the default `READ` device and 6 the default `WRITE` device, in which case the third entry on each line above is equivalent to the first two. You may however use 5 and 6 to point to your own files if you wish.

4.3 Input and Output Formats for Character Data

A Descriptor: For a CHARACTER input variable of length l, **Aw** inputs the next w characters and assigns the rightmost min(w,l) characters to the leftmost min(w,l) positions in the input variable, padding any remaining positions with blanks. E.g. (l=5,w=4,6)

```
CHARACTER(len=5):: C1,C2
READ(*,'(A4)') C1
READ(*,'(A6)') C2
```

If the two lines

```
ABCD
ABCDEF
```

are input, the value of C1 is `'ABCD '` (note the trailing blank) and C2 is `'BCDEF'` (no A). If w is omitted then w=l is assumed.

For a CHARACTER output variable of length l, **Aw** outputs the leftmost min(w,l) characters of the variable to the rightmost min(w,l) positions of the next w positions. E.g. (l=5,w=4,6)

```
CHARACTER(len=5):: C
C='ABCDE'
WRITE(*,'(A4)') C
WRITE(*,'(A6)') C
```

the two output lines are

```
ABCD
 ABCDE
```

Note the leading blank in the second line. If w is omitted then w=l is assumed.

G Descriptor: **Gw** can also be used for CHARACTER variables, in which case it is identical to Aw.

List-Directed: List-directed input works fine if the input is a string, i.e. enclosed in quotes. E.g.

```
CHARACTER(len=5):: C
READ(*,*) C
```

If the input line is

```
'AB DE'
```

then C is `'AB DE'` (note the trailing blank). However, if the input line is

```
AB CD
```

then C is `'AB  '` (note the three trailing blanks). The blank in the input is interpreted as a data delimiter (or separator).

If the data contains blanks or quotes, which are not delimiters, use A (no w) .

4.4 Creating Tables

Recommendations (see examples in §5, PROCESSING CONTROL STATEMENTS)**:-**

(1) Use visual alignment for headings. Often best to start with the longest line.

(2) Fill in other lines using visual alignment relative to first line.

(3) Insert a comment reference line to allow easy column counting. Just run a finger along the numeric keys 1,2,3,....,0 at the top of the keypad. Either repeat that, or more simply use a mouse or editor to repeat the pattern as needed. Make the first "1" start under the leftmost output character of the headings. Alternatively use an editor to determine the column numbers.

(4) Use T (tabular) format to print the table. Format Tn means read or print in column n. (For output on some systems the first column may be lost to carriage control of the output device. Then Tn means column n-1.) Advantage: The position of any one column can be modified without affecting the positions of any other column. This is NOT the case with X formatting, which should not be used for more than about 2 columns. (nX means leave n blank spaces.)

5. PROCESSING CONTROL STATEMENTS

GOTO, DO, EXIT, CYCLE, IF, CASE (see fortuts 4,5 for more details).

Recommendations:-

(a) Both GOTO and DO allow infinite loops. But it is almost always preferable to avoid the possibility of accidental infinite looping by using DO with a definite upper limit of loops.
(b) Often GOTO's may be used in place of other commands. Use of a single GOTO may be convenient and not affect readability, but excessive use of GOTO's reduces readability and checkability, and should be avoided. Overlapping/tangled GOTO's → *spaghetti programming*, jumpy GOTO's → *kangaroo code*. E.g.

```
      GO TO 3
2     ...
      GOTO ...
      ...
3     ...
      ...
      GOTO 2
```

Can be difficult to read, and debug.

Aim for so-called 'top-down' program logic

Example 5.1 (partly done in lectures)
Student registration classifications, Set 2 Q5, p47.

Example 5.2 (outlined in lectures)
Tabulation of series approximations for $\sin^2 x$, Set2 Q8, p.49.

Example 5.3
Comparison of BLOCK IF and SELECT CASE constructs. The following programs may be copied from $mc3fp/brackets_if.f90 and $mc3fp/brackets_case.f90, and compiled and run in the usual way. G31

```
!============================================================
   program brackets_if
   ! To read line-by-line from standard input, and count
   ! left and right parentheses(), braces{}, brackets[]

      ! Initialize counters for each type
```

```
      integer :: lpars=0,rpars=0,lbraces=0,rbraces=0,lbracks=0,rbracks=0

      ! Declare a sufficiently long character string to hold each
      ! line of the input file, and one to hold the ith character
      character line*200, ithc*1

      ! Start the read loop -- reading 1st 200 characters from each line
2     read(*,'(A)',end=1) line  ! NOT read(*,*,end=1) line
                                ! To terminate input with < 200 characters
                                ! use an EOF: [Ctrl][d] in unix; or
                                ! [Enter][Ctrl][z] windows

      ! Test each character in the input line, and keep count
      do i=1,len_trim(line)   !Inbuilt function len_trim returns length
                              !after trailing blanks removed.
                              !Less efficiently, could do i=1,200
         !ith character in line, i.e. from position i to position i is
         ithc = line(i:i)
         if(ithc=='(') then             !Left parantheses
            lpars= lpars+1
         elseif(ithc==')') then         !Right parentheses
            rpars=rpars+1
         elseif(ithc=='[') then         !Left brackets
            lbracks=lbracks+1
         elseif(ithc==']') then         !Right brackets
            rbracks=rbracks+1
         elseif(ithc=='{') then         !Left braces
            lbraces=lbraces+1
         elseif(ithc=='}') then         !Right braces
            rbraces=rbraces+1
         endif
      enddo

      ! Read another line
      goto 2

      ! Print the respective totals
1     print 100, lpars, rpars,lbraces,rbraces,lbracks,rbracks
100   format(/I5,' ( ,  ',I5,' ) ,' &
             /I5,' { ,  ',I5,' } ,' &
             /I5,' [ ,  ',I5,' ] ,' /)

   end program
!============================================================


!============================================================
   program brackets_case
```

(… same as above except IF construct replaced by CASE construct.)

```
      select case (ithc)
      case('(')                 !Left parantheses
         lpars= lpars+1
      case(')')                 !Right parentheses
         rpars=rpars+1
      case('[')                 !Left brackets
         lbracks=lbracks+1
      case(']')                 !Right brackets
         rbracks=rbracks+1
      case('{')                 !Left braces
         lbraces=lbraces+1
      case('}')                 !Right braces
         rbraces=rbraces+1
      end select
!============================================================
```

Basic rules for the general DO-loop:-

```
DO I=I1,I2,Istep
...
ENDDO
```

(a) To avoid round-off errors, the variables `I`, `I1`, `I2`, `Istep` must all be INTEGER type.

(b) The loop is executed `N` times, where, using integer arithmetic,

$$N=\max\left\{0,\ \frac{I2-I1}{Istep}+1\right\};$$

i.e. `I=I1, I1+Istep,...,I1+(N-1)*Istep.`

Note:
I2>I1 and Istep<0 imply N=0
I2<I1 and Istep>0 imply N=0,
i.e. loop is not executed in these cases

(c) If the loop is executed the full `N` times, then the value of `I` after the final time through the loop is `I=I1+N*Istep` ,i.e. one `Istep` more than the value of `I` used last time through the loop. If the loop is exited before being executed `N` times, then the value of `I` following the loop is just the last value of `I` used in the loop.

6. READING/WRITING DATA FILES

See fortuts 3,6, for more information on reading, writing and arrays. The present chapter considers various scenarios, their solutions being completed in lectures. The chapter deals mainly with reading. Writing is similar, except that a FORMAT statement should be used to control the output.

6.1 Reading versus Writing

READ	WRITE
If data is separated by blanks or commas, then it is easiest/safest to use READ*, or READ(1,*) etc Otherwise use a FORMAT.	Use FORMAT if control is required over the form of the output. Otherwise use PRINT*, or WRITE(1,*) etc

Suppose it is required to read and write a mixed data line like

```
3.1, 2   1E4   name  (2.1,-3.2)
```

```
REAL
```

```
READ
```

```
WRITE
```

6.2 Detecting End of Input Data

Basically 2 possibilities as in scenarios (1) and (2) below:-

(1) File ends with user-supplied end-of-data flag (i.e. any item of the same type, but with value distinct from the data).

What if myfile.dat looks like

```
xx    xx    xx    xx
⋮
```

```
xx    xx    xx    xx
99    yy    yy    yy
```

where xx are integers, 99 is any integer distinct from the data, yy is any integer padding, and we want to read each line of numbers into variables K,L,M,N for processing?

(2) File ends with the system supplied EOF (i.e. end-of-file) mark.

What if myfile.dat looks like

```
xx    xx    xx    xx
⋮
xx    xx    xx    xx
```

and we want to read each line into variables K,L,M,N for processing?

EOF hidden mark

6.3 Reading Data Files into 1D Arrays

(1) Reading a regularly spaced array

What if myfile.dat looks like

```
x.x  x.x  x.x  x.x
x.x  x.x  x.x  x.x
x.x  x.x  x.x  x.x
x.x  x.x  x.x  x.x
x.x  x.x  x.x  x.x
x.x  x.x  x.x  x.x
```

and we want to read into 1D REAL array A?

(2) What if lines are unequal lengths, and spacing variable.

```
x.x   x.x  x.x x.x
x.x x.x
(a blank line perhaps)
…
x.x  x.x x.x
```

(3) What if we don't know beforehand, but want to, know the number of input items? Then one approach is to (i) preset the elements of array A to some value distinct from the data, (ii) read in

values overwriting A, until the EOF is reached, (iii) test as in §6.2(1) earlier. Note that in tests comparing floating-point numbers, allowance should be made for roundoff errors.

6.4 Reading Data into 2D Arrays

Higher dimensions similarly.

(1) What if myfile.dat is as below, and we want to read into A(3,2)? (See fortut6, and Set2 Q13(i), p.51, 'aeroplane seats' problem.)

```
1  2
3  4
5  6
```

6.5 Reading Data that Contains Headings

In general, always inspect data files if possible before reading, but this is sometimes not possible.

(1) What if data file contains a heading line, not needed for the actual processing.

```
Any heading line
   1  2
   3  4
   5  6
```

(2) What if the heading is to be stored for re-use?

```
Any heading line
   1  2
   3  4
   5  6
```

(3) What if headings are embedded in a data line, and one wants to extract the numeric values, not worrying about the variable names? E.g. suppose the data line is:-

```
A = 34.2     B = 21.72,C= (13.3,-4.2)
```

(Note the "C=" with no space separator.) Then, in F90 (but not earlier standards), simply

```
REAL A,B;    COMPLEX     ;    CHARACTER

READ
```

(4) What if headings are in the data line as in (3), but one wants to store and re-use the headings, and perhaps process or write out the data in a different format?

```
      REAL A,B;

      READ

      WRITE

100 FORMAT
```

Note the equivalence of the 'array section' expression ANY(3:4) to the list of elements ANY(3),ANY(4), since ANY is an array – see fortut6.

(5) What if the data consists of a single line of blank-separated items, and one wants to determine the number of items in the line?

```
CHARACTER LINE*100          !100 being any sufficiently large number
OPEN(1,FILE='myfile.dat')
READ(1,'(A)') LINE          !Data line stored in character variable LINE
NITEMS=0
DO  I=100,2,-1
  IF(LINE(I:I) = ' '.AND.LINE(I-1:I-1) /= ' ')  NITEMS=NITEMS+1
ENDDO
```

6.6 Internal Files

In the same way that data may be read from and written to a data file, it may also be read from or written to a character variable, which is then known as an 'internal file'. This is particularly useful for converting numeric data to character and vice versa. It then allows numeric data to be edited character-by-character for example. Basically, a character variable may be used in place of a logical unit number in READ and WRITE statements. For example,

```
CHARACTER C*10
WRITE(C,'(F8.3)')  12.3
READ(C,'(I4)')  J
```

causes C to have the value 'bb12.300', where b represents a blank, and causes J to have value 12 .

6.7 Namelists

A NAMELIST associates a name with a list of variables. NAMELISTs are particularly useful for input and output. Also a NAMELIST record in an input or output file identifies the data. E.g

```
INTEGER : : I=1, J, K
REAL :: X,Y=1.5
COMPLEX :: Z
REAL, DIMENSION(3) :: W
CHARACTER (len=10) :: C
LOGICAL :: L
NAMELIST / inputdata / I, J, X, Y, W, C, L
```

The `NAMELIST` statement groups the variables `I,J,X,Y,Z,W,C,L` into a list called '`inputdata`'.
The following input is a NAMELIST record

```
&inputdata
X = 2.5
Y = 3.5
Z = (0.5,2.0)  ! Note complex data.
J = 4
W = 1., 2., 3.
C = 'string'
L = .true.
/
```

Note the first statement with the ampersand `&` and the name `inputdata` of the NAMELIST. The record is terminated with a forward slash `/`. The order of the variables in the record is irrelevant and not all NAMELIST variables need be included in the record: here `I` is omitted. Select elements of an array may also be set. Comments (with an `!`) may be included in a record. Large arrays should not be included in NAMELISTS.

If this record is contained in a data file called 'input.dat' it can be read using

```
OPEN(1,FILE='input.dat')
READ(1,NML=inputdata)
```

The values of the variables `J,X,Y,Z,W,C,L` will be set or changed to the values in the record; `I` is unchanged. A file may contain more than one NAMELIST record, which can be read by subsequent READ's. A NAMELIST can be written to a file (or to screen) as follows:

```
OPEN(2,FILE='output.dat')
WRITE(2,NML=inputdata)
```

Depending on the compiler the output will look something like:

```
&inputdata
I = 1
J = 4
X = 2.5
Y = 3.5
Z = (0.5,2.0)
W = 1., 2., 3.
C = 'string'
L = .true.
/
```

The array may be listed element by element.

7. FUNCTIONS, SUBROUTINES and MODULES

See fortut7 for additional examples and information.

(1) Statement Functions: for functions definable by a single statement.

(2) Internal Functions and **(3) Internal Subroutines**
Subprograms (2) and (3) are the same except for the manner in which they are referenced/invoked.

(4) External Functions and **(5) External Subroutines**
Subprogram types (4) and (5) are the same except for the manner in which they are referenced/invoked. Usually a Function is used to return just its value, whereas a Subroutine returns values for more than a single argument.

Subprograms (2)--(5) are also known as 'procedures'.

(6) Modules

7.1 Statement Functions

```
!====================
Program unit header line
......
Type declarations and other
specification statements
......
```

Start of the host program unit which in this illustration might be a main program or procedure (i.e. function or subroutine, internal or external). The host may be a module (see later), but then does not invoke the statement function itself.

```
F(x,y,...) = exp involving x,y
......
......
......
......
......
```

Definition - occurs after any IMPLICT or related type declarations, but before any executable statement. Dummy arguments x,y,... are simple variables. RHS is any expression involving x,y,... , including constants and other functions.

z = *exp involving* F(a,b,...)	**Invocation** - just some expression referencing F.
.....	Actual arguments a,b,... are expressions or constants
.....	
END	End of the invoking program unit
!======================	

Example 7.1
The following program is available in the file $mc3fp/stfun.f90

```
!=======================================
PROGRAM STATEMENT_FUNCTION_DEMO
!  Calculates areas of various circles

   REAL RAD(4)
!  REAL, DIMENSION(4)::RAD=[1.2,2.3,3.4,4.5]  (An alternative)
!  (F90 Standard 'array constructor' is RAD=(/1.2,2.3,3.4,4.5/)

   AREA(R)=PI*R**2    !The statement function
   PI=ACOS(-1.)
   RAD=[1.2,2.3,3.4,4.5]
   DO I=1,4
      PRINT '(A,F4.1,A,G10.3)',&
            'Area of circle, radius ',RAD(I),' is ',AREA(RAD(I))
   ENDDO
END
!=======================================
```

7.2 Internal Procedures

!======================	Start of the host program unit which in this illustration might be a main program or **external** procedure (i.e. function or subroutine). It may never be an **internal** procedure. The host may be a module (see later), but then does not invoke the internal procedure itself.
Program unit header line	
.....	
.....	
.....	
.....	
h = *exp involving* F(u,v,...)	**(2) Invocation** – any expression referencing F
.....	Actual arguments u,v,... may be variables, expressions; array names, array elements; or function names; or even constants provided they are not changed by the invoked subprogram.
.....	
.....	
.....	
CALL S(r,s,...)	**(3) Invocation** of subroutine S.
.....	Actual arguments r,s,... obey the same rules as in (2) above.
CONTAINS	←← signalling start of internal procedures
!------------------------	

FUNCTION F(t,w,...) ..... *Typically statements that use the dummy arguments and other shared variables to define* F. *These should include a statement* F = *expression* *that returns a value for* F *to the invoking program.*	**(2) Definition** of F Dummy arguments t,w,... may be variables, array names or function names. They may not be constants or array elements. Some of the dummy arguments may be changed in F, and hence the values of the actual arguments in the invoking program are also changed. Any variable not in the dummy argument list, and not declared REAL, INTEGER etc, in the internal program unit (directly or via USE see §7.4, is taken to be the variable of the same name in the host program unit (i.e. so-called 'host association').
END FUNCTION F !----------	Inclusion of the word FUNCTION in the END statement is compulsory for internal functions.
SUBROUTINE S(l,m,...) ..... *Typically statements that define or change dummy arguments and other shared variables.* S *itself does not take a value or have a type. On return to the invoking program, values of the dummy args are taken by the corresponding actual args, and similarly for other shared variables.*	**(3) Definition** of S Dummy arguments l,m,..and other variables obey the same rules as for internal functions in (2) above.
END SUBROUTINE S !---------- ..... *Other internal procedures* ..... !----------	Inclusion of the word SUBROUTINE in the END statement is compulsory for internal subroutines.
END ..... !==========	End of the host program unit

Example 7.2

The following programs are available in the files $mc3fp/intfun.f90 and intsub.f90. Note the use of a character variable to store a format.

```
!=======================================
PROGRAM INTERNAL_FUNCTION_DEMO
!Calculates areas of a circle and a square

   CHARACTER FMT*60 !Character variable to hold format spec
   PI=ACOS(-1.)

   !The circle
   R=200
   !Based on col ref numbers below, print result in cols 40-
   FMT='(          A,                F5.1,     A,       T40,G10.3)'
   PRINT FMT,'For circle, radius ',R,', area is: ',AREA('circle',R,R)
   !cols ref: 12345678901234567890123456789023456789O

   !The square
   A=100;B=100
   PRINT FMT,'For square, side ',A,', area is: ,AREA('rectangle',A,B)
```

```
CONTAINS !-----------------------------
   FUNCTION AREA(OBJECT,LENGTH1,LENGTH2)
      CHARACTER(*) OBJECT
      !Alternatively CHARACTER*(*) OBJECT  or  CHARACTER OBJECT*(*)
      REAL LENGTH1, LENGTH2
      SELECT CASE(OBJECT)
      CASE('circle')
         AREA=PI*LENGTH1**2
      CASE('rectangle')
         AREA=LENGTH1*LENGTH2
      END SELECT
   END FUNCTION
END PROGRAM
!=======================================

!=======================================
PROGRAM INTERNAL_SUBROUTINE_DEMO
!Calculates areas and perimeters of a circle and a rectangle

!The circle
      PI=ACOS(-1.)
      R=200
      CALL MEASURES('circle',R,R)
      PRINT 100, R,AREA,PERIM
100   FORMAT( 'For circle, radius ',F5.1,' :',&
              T35,' Area = ',G10.3,T55,' Perimeter = ',G10.3)

!The rectangle
      A=100;B=200
      CALL MEASURES('rectangle',A,B)
      PRINT 110, A,B,AREA,PERIM
110   FORMAT( 'For rectangle, sides ',F5.1,',',F5.1' :',&
              T35,' Area = ',G10.3,T55,' Perimeter = ',G10.3)

CONTAINS !-----------------------------
      SUBROUTINE MEASURES(OBJECT,LENGTH1,LENGTH2)
!Have omitted AREA, PERIM from CALL and SUBROUTINE
!statements, since they are not declared locally in
!Subroutine Measures, and are therefore the same variables
!in both program and internal subroutine (as is PI).
         CHARACTER(*) OBJECT
         REAL LENGTH1, LENGTH2
         SELECT CASE(OBJECT)
         CASE('circle')
            AREA=PI*LENGTH1**2
            PERIM=2*PI*LENGTH1
         CASE('rectangle')
            AREA=LENGTH1*LENGTH2
            PERIM=2*(LENGTH1+LENGTH2)
         ENDSELECT
      END SUBROUTINE
END PROGRAM
!=======================================
```

7.3 External Procedures

!====================	Start of the invoking program unit which may be a main program, internal or external procedure, but not a module.
Program unit header line	
.....	
.....	
z = *exp involving* F(u,v,...)	**(4) Invocation** of function F.
.....	
CALL S(r,s,...)	**(5) Invocation** of subroutine S.
.....	
.....	Actual arguments u,v,r,s,... obey the same rules as for internal procedures in (2) and (3).
.....	
.....	
.....	
END	End of invoking program
!====================	
FUNCTION F(t,w,...)	**(4) Definition**
.....	Dummy arguments t,w,... obey the same rules as for internal procedures in (2) and (3).
As for (2) earlier.	
.....	
.....	Any variable not in the dummy argument list, is local, i.e. distinct from a variable of the same name in any other program unit, unless an association is made via a COMMON (i.e.'storage association') or USE statement ('use association').
.....	
.....	
.....	
.....	
END FUNCTION F	End of function F. ('FUNCTION' and 'F' are optional.)
!====================	
SUBROUTINE S(l,m,...)	**(5) Definition.** Dummy arguments and local variables obey the same rules as for (4) above.
.....	
As for (3) earlier	
.....	
END SUBROUTINE S	End of subroutine S. ('SUBROUTINE' and 'S' are optional.)
!====================	

Example 7.3

Available in file $mc3fp/extsub.f90 . Illustrates external subroutine. Also, shows use of initialization statement and SAVE statement. In general, because of dynamic memory management in modern computers, values of local variables in subprograms are often forgotten between successive calls to the subprogram. A value can be forced to be remembered by including the variable name in a SAVE statement This is illustrated below for the variables PI and K (although variables like K that are initialized in a type declaration are automatically saved, making the SAVE statement redundant).

```
!=====================================
PROGRAM EXTERNAL_SUBROUTINE_DEMO
!Calculates areas and perimeters of a circle and a rectangle

      !The circle
      R=200
      CALL MEASURES('circle',R,R,AREA,PERIM)
      PRINT 100, R,AREA,PERIM
100   FORMAT( 'For circle, radius ',F5.1,' :',&
              T35,' Area = ',G10.3,T55,' Perimeter = ',G10.3)
```

```
      !The rectangle
      A=100;B=200
   CALL MEASURES('rectangle',A,B,AREA,PERIM)
      PRINT 110, A,B,AREA,PERIM
110   FORMAT( 'For rectangle, sides ',F5.1,','F5.1' :',&
              T35,' Area = ',G10.3,T55,' Perimeter = ',G10.3)

END PROGRAM
!--------------------------------------
!An external subroutine follows:-
SUBROUTINE MEASURES(OBJECT,LENGTH1,LENGTH2,AREA,PERIM)
!Here, can't omit AREA, PERIM from CALL and SUBROUTINE statements, since
!they are now local variables in both program and subroutine.

   CHARACTER(*) OBJECT
   REAL LENGTH1, LENGTH2
   INTEGER :: K=1
   SAVE PI,K        !Redundant for K since variables initialized in a type declaration
                    !are automatically SAVEd
   IF (K==1) THEN !on first call only, assign PI
      PI=ACOS(-1.)
      K=0
   ENDIF
   SELECT CASE(OBJECT)
   CASE('circle')
      AREA=PI*LENGTH1**2
      PERIM=2*PI*LENGTH1
   CASE('rectangle')
      AREA=LENGTH1*LENGTH2
      PERIM=2*(LENGTH1+LENGTH2)
   ENDSELECT
END SUBROUTINE
!=======================================
```

7.4 Modules

Modules are useful for packaging global data and subprograms. In their simplest form they contain just type declarations, data initializations and internal subprograms. They may be USEd by any other program unit after the module has been defined, and variables are then shared by 'use association'.

```
!====================
MODULE  Modname
......
Type declarations and
initializations.
......

CONTAINS                    ←←  Signals start of the internal procedures
!----------------------
  ......
Internal procedures
  ......
!----------------------
END MODULE Modname          End of the module
!--------------------------------
```

Program unit header line.....	Start of the using program unit
.....	
USE Modname	Variables and subprograms in MODULE Modname are now known to this program just as if they had been declared or defined internally. They should not be declared again if already declared in Modname.
Type declarations etc	
.....	
Other program statements	
.....	
END	End of the using program unit
!----------	

Examples:- See fortut7

7.5 General Rules

(i) Type (i.e. REAL, INTEGER, CHARACTER, ...) of dummy arguments must match actual arguments.

(ii) The number of actual arguments in the function reference or CALL statement, may be less than the number of dummy arguments in the FUNCTION or SUBROUTINE statement, but for simplicity and safety it is best to keep the numbers equal. (Compatibility note: the two numbers must be equal pre-F90.)

(iii) Dimensions of actual arguments are declared in the invoking program, and dimensions of dummy arguments must be declared in the subprograms.

(iv) For arguments that are arrays, it is safest to give both dummy and actual arguments the same dimensions. (But see also §7.7 "adjustable dimensions", on p.34, and §7.9 on p.37.)

(v) A dummy argument name may or may not be the same as the actual argument name. Corresponding dummy and actual arguments usually share the same memory location, and thus take the same value. (The COMMON statement in fortut7 achieves a similar end.) For readability it is often best to keep the actual and dummy argument names the same.

(vi) Procedures may contain multiple RETURNs and STOPs. If no such statements are present then the return to the invoking program occurs when the END statement is reached.

(vii) Subprograms may be grouped in any order with the main program in the same file, main.f say. MODULEs must go before they are USEd, i.e. even before the main program. (Pre-F90, when there were no MODULEs, the main program had to go first.) Compilation and linking is then achieved via

```
f90o  main.f90
```

Alternatively, some or all routines may be in separate files. Then

```
f90o  main.f90  myfun.f90  mysub.f90
```

or use .o files if previously compiled using f90 –c.

7.6 Advantages

The use of subprograms:-

(i) *Facilitates repetition* of same calculation with changed arguments. Avoids need to repeat large blocks of statements.

(ii) *Increases readability*: streamlines each program unit. E.g. saves the main program being cluttered by complicated details.

(iii) *Increases checkability*: larger programs are best divided into smaller units each of which can be separately compiled and checked for errors on test data.

(iv) *Increases portability*: External procedures and modules may be written independently of invoking programs units, and then readily used by other invoking programs.

(v) *Saves time* when compiling/debugging.

Example (a): Suppose we want to test some large calculation on 2 (or more) different functions. Compiling main.f90, f1.f90, f2.f90 separately, i.e.

```
f90 -c  main.f90
f90 -c  f1.f90
f90 -c  f2.f90
```

and then linking via

```
f90o  main.o  f1.o
f90o  main.o  f2.o
```

may be much faster than compiling main twice as in

```
f90o  main.f90  f1.f90
f90o  main.f90  f2.f90
```

if main.f90 takes a long time to compile.

Example (b): Saves recompiling the entire program when an error is traceable to just one program unit. Only the faulty program unit need be recompiled, and the resulting .o file linked to the other existing .o files.

(vi) Allows use of pre-compiled scientific/commercial subprogram packages: One only needs to link them with one's program. E.g. NAG, IMSL, NSWC, KMN libraries.
Programming then becomes an exercise in linking together appropriate 'black boxes' and supplying a 'driver' main program. One advantages of Fortran is the availability of many free/low-cost subprogram packages.

7.7 Adjustable dimensions

Consider the problem of inverting a number of matrices of different sizes.

Case (1): Using different names for the different matrices.

```
!============================
PROGRAM MAIN
    REAL A(2,2), B(3,3),
    ......
    ......
    ... define A
    CALL INVERT(A,2)
    ... define B
    CALL INVERT(B,3)
    ......
END
!----------------------------
SUBROUTINE INVERT(C,N)
!inverts the NxN matrix C
REAL C(N,N)                          ← "Adjustable dimensions"
    ......
END
!============================
```

Case (2): Consider using just one matrix A(3,3) to do the previous job. Then more care is needed, to avoid addressing errors as shown below.

```
!============================
PROGRAM MAIN
    REAL A(3,3)
    ......
 define elements A(1,1),
 A(1,2), A(2,1), A(2,2)
CALL INVERT(A,2)                     ← This would attempt to invert the wrong
    ......                             submatrix - see the explanation below.
 define A(1,1),...,A(3,3)
CALL INVERT(A,3)                     ← This would correctly invert A.
    ......
END
!----------------------------
SUBROUTINE INVERT(C,N)
    REAL C(N,N)
!inverts the NxN matrix C
    ......
END
!============================
```

The reason that the CALL INVERT(A,2) in Case (2) fails to invert the correct sub-matrix is that matrices are stored columnwise (i.e. in so-called array-element-order, first subscript varying fastest), and the first element of A is matched to the first element of C. The incorrect match is thus:-

$$\begin{array}{l} C(1,1) \rightarrow A(1,1) \\ C(2,1) \rightarrow A(2,1) \\ C(1,2) \rightarrow A(3,1) \\ C(2,2) \rightarrow A(1,2) \end{array} \quad \Big| \quad \text{i.e.} \begin{bmatrix} C(1,1) & C(1,2) \\ C(2,1) & C(2,2) \end{bmatrix} = \begin{bmatrix} A(1,1) & A(3,1) \\ A(2,1) & A(1,2) \end{bmatrix}$$

This problem is overcome by making the column length the same in the subprogram as in the main (or invoking) program unit, as shown below:-

```
!==============================
PROGRAM MAIN
   REAL  A(3,3)
 define A(1,1),...,A(2,2) as
 before, leaving A(1,3), A(2,3)
 possibly undefined.

CALL BETTER_INV(A,3,2)
   ......
END
!------------------------------
SUBROUTINE BETTER_INV(C,N1,N2)
   REAL  C(N1,N2)
 just invert the top left
   N2×N2 submatrix of C,
   i.e. C(1:N2,1:N2)
   ......
END
!==============================
```

← Pass the number of rows (i.e. 3) to the subprogram, as well as the dimension of the sub-matrix of interest.

← Correct submatrix is now inverted - see details below.

The match, now correct, is:-

$C(1,1) \rightarrow A(1,1)$
$C(2,1) \rightarrow A(2,1)$
$C(3,1) \rightarrow A(3,1)$ *undefined*
$C(1,2) \rightarrow A(1,2)$
$C(2,2) \rightarrow A(2,2)$
$C(3,2) \rightarrow A(3,2)$ *undefined*

I.e. $\begin{bmatrix} C(1,1) & C(1,2) \\ C(2,1) & C(2,2) \end{bmatrix} = \begin{bmatrix} A(1,1) & A(1,2) \\ A(2,1) & A(2,2) \end{bmatrix}$

7.8 Intrinsic Procedures

F90 contains more than 100 intrinsic functions and subroutines, many of which have already been experienced, e.g. SIN(X), ACOS(X) etc.

Example 7.4
The following example program (for Set3 Q6, p.57), available as $mc3fp/random_pi.f90, uses the intrinsic random-number generator in F90 to illustrate the Monte-Carlo method. (For other simple examples of Monte-Carlo simulations see **NL,** and W. Cheney & D. Kincaid *Numerical Mathematics and Computing* §9.3).

```
!================================================================
   PROGRAM MONTE_CARLO_PI
   !Estimation of pi by Monte Carlo Method
!============================
!! MAIN VARIABLES/SUBPROGRAMS
!----------------------------
!  COORDS          Array to hold randomly generated (X,Y) pair
!  PI_EST          Estimate of pi
!  RANDOM_NUMBER   Intrinsic F90 uniform random number subroutine
!============================
      INTEGER TRIALS, COUNT
      REAL COORDS(2)

      !Set up headings
      PRINT 100
```

```
100    FORMAT(' ESTIMATE OF PI BY MONTE-CARLO METHOD'//&
              '         No. of        Pi'/&
              '         trials     Estimate'/)
       !Cols   12345678901123456789023456789O

       !Initialize success counter
       COUNT=0

       !Begin loop for pointpicking
       DO TRIALS=1,100000

          CALL RANDOM_NUMBER(COORDS)
          X=2*COORDS(1)-1     !uniformly distributed over (-1,1)
          Y=2*COORDS(2)-1
          !Increment counter if point inside circle
          IF (X**2+Y**2 <= 1) COUNT=COUNT+1

          !Print area-based estimate of pi every 10^4 trials
          IF (MOD(TRIALS,10000) == 0) THEN
             PI_EST=4.*REAL(COUNT)/TRIALS   !<<---Avoid integer arithmetic
             PRINT 101, TRIALS, PI_EST
101          FORMAT(T9,I6,T20,F7.4)
          ENDIF

       ENDDO    !trials loop
   END PROGRAM
!===============================================================
!
! ESTIMATE OF PI BY MONTE-CARLO METHOD
!
!         No. of        Pi
!         trials     Estimate
!
!          10000      3.1208
!          20000      3.1268
!          30000      3.1423
!          40000      3.1435
!          50000      3.1418
!          60000      3.1421
!          70000      3.1419
!          80000      3.1410
!          90000      3.1401
!         100000      3.1410
```

7.9 Recursive, Elemental and Array-valued Procedures

F95 allows various other special types of procedures, not examinable in this course, but some of which are briefly described here for interest.

(i) **Recursive Procedures**: These are procedures that reference themselves, and are useful for defining functions like the Factorial Function, or doing iterated integrals. See fortut7 for an example.

(ii) **Elemental Functions**: These are functions with scalar dummy arguments, but which can be invoked with array actual arguments, the function operations then being performed elementwise. The intrinsic function `COS(X)` is an example. F95 (but not F90) permits user-defined elemental functions. A simple program that would take a given array `X` and return the same array but with negative elements set to zero is

```
REAL X(4)
X=[1,-2,3,-4]
PRINT*, F(X)
CONTAINS
   ELEMENTAL FUNCTION F(DX)    !F and dummy arg DX are scalars
      REAL,INTENT(IN)::DX      !Dummy args must have intent IN only
      IF (DX<0) THEN
         F=0
      ELSE
         F=DX
      ENDIF
   END FUNCTION
END PROGRAM
```

The array values output by the `PRINT` statement would be `1.0 0.0 3.0 0.0`

(iii) **Array-valued Functions:** These are functions that return an array as the function value. They require a so-called *explicit interface* which can be achieved via an `INTERFACE` command (not part of this course), or by making the function internal via a `CONTAINS` statement, or by `USE`ing a module that `CONTAINS` the function. For example, the same result as in (ii) can be achieved by an array-valued internal function as follows.

```
REAL X(4)
X=[1,-2,3,-4]
PRINT*, F(X)
CONTAINS
   FUNCTION F(DX)      !F and DX are arrays
      REAL F(4), DX(4)
      WHERE (DX<0)
         F=0
      ELSEWHERE
         F=DX
      END WHERE
   END FUNCTION
END PROGRAM
```

8. F90 Exercises

The following exercises are for assessment unlike those in the fortuts. The fortuts exercises should also be done, but not handed in. **It is important to read §1.2,** 'Doing the F90 Exercises and Assignment', on p.4. Note that questions tagged by **A** are Advanced.

8.1 F90 Exercises Set 1

NOTE: Do not use arrays, subprograms or statement functions in any question of this set. Assume Fortran default types (i.e. INTEGER, REAL), determined by leading characters of variable names, unless otherwise specified.

1. Classify the following variable names as valid or invalid under the 1990 Fortran Standard. If invalid give a brief reason, and if valid then state their Fortran default type as either REAL or INTEGER.
   (i) `LENGTH` (ii) `AREA` (iii) `2TIMER` (iv) `$AMOUNT` (v) `What_a_name!` (vi) `MASSES`
   (vii) `VELOCITY` (viii) `ThisIsTooLongIsItNot?`

2. State whether or not the 2 values in each of the following pairs are the same number. If not, indicate the difference. (Here, 'same' means that the numbers have the same bit pattern in the computer memory, so that integer 1 is not the same as real 1.0 .)
   (i) 0.000007 , 70E-6 ; (ii) 1010 , 101.0E01 ; (iii) -0.76, 7.6E-01 ; (iv) 123.4E-01 , 0.1234*E+02 .

3. Convert the following algebraic equations into valid Fortran statements. Assume default variable types, and that the RHS variables have been assigned values matching their types.. In each case give two answers: (a) without mixed mode arithmetic, and (b) with mixed mode wherever possible.
   (i) Equivalent resistance of 4 parallel resistors: use only a single statement.

$$R = \frac{1}{\frac{1}{R_1}+\frac{1}{R_2}+\frac{1}{R_3}+\frac{1}{R_4}}$$

   (ii) Mass flow rate through a nozzle: use about 6 relatively simple statements, one of which uses the intrinsic function ATAN(x) (which approximates the mathematical function $\tan^{-1}x$, also known as arctan$x$) to evaluate π.

$$X = 2\pi X_1 \sqrt{\frac{32.2\ \ G}{X_2 Y_1}} \frac{Y_2}{Y_1}\left[1+\frac{G-1.0}{2.0}\left(\frac{Y_2}{Y_1}\right)^2\right]^{\frac{G+1}{2-2G}}$$

   (iii)**A** Flow of an ideal, compressed gas: for the non-mixed-mode version (a), use precisely 2 statements, and for the mixed-mode (b) use a single statement. There should be no integer roundoff in the calculation.

$$\mathrm{FLOW} = \left[1+\frac{\mathrm{K}-1}{2\mathrm{M}^2}\right]^{\mathrm{K}/(\mathrm{K}-1)}$$

   [Hints: (α) The statement FK=K converts the INTEGER K to floating-point form and stores the result in the REAL variable FK. (β) the functions FLOAT(M) and REAL(M) are equivalent ways of converting INTEGER M to a REAL (i.e. floating-point) representation.]

4. Assume Fortran default types, and the initial values R = -2.3, J = 7, I = 2, X = 3.4 . State the value that will be stored in the variable on the left side of the following equations. Show your answers in the correct form, i.e. REAL or INTEGER type numbers.

   (i) N1 = X - 3 (ii) AJ = J/4/I + 1 (iii) BJ=J/4*I
   (iv) K = X + I/J**2 (v) A = X*J**(-I)

5. Rewrite each of the following sets of statements as a single statement that will yield the same final assignment. For example, the two statements

```
B = A + 1
A = B + 2
```

have the same final effect as `A = A + 3`

(i)
```
A = B
B = A
```

(ii)
```
Y = Y + 2
Z = X + Y
```

(iii)
```
A = B + 2
B = A + 2
A = A + B
```

(iv)
```
M = N + 1
M = M - 1
N = M
```

6.† (The basic mechanics for doing parts (i)-(iii) of this question are given on p.15.)
Write a Fortran program `HRS_MINS` (in file hrs_mins.f90) that will read in `MINS` minutes and correctly determine the corresponding number of hours, `HOURS`, and the remaining minutes, `RMINS`. For example, when `MINS` is 65, then your program should yield output similar to

```
HOURS = 1,  RMINS = 5
```

For input and output, use only READ* and PRINT*. The program should produce a single line of output like the above, being a mixture of headings and results. (Of course, PRINT* allows only limited control over the output format, so won't allow output precisely as shown above.) Use only integer variables and integer arithmetic throughout, and use the variable and file names stated above. Do not use any Fortran intrinsic functions such as INT or MOD.

In the following parts read in the value 357 for `MINS`.

(i) Compile and test run your program, by inputting `MINS` from the keyboard. Include a prompt for input.
(ii) Modify your program to comment out the input prompt, and re-run it using the UNIX input directive "<" to read `MINS` from a file hrs_mins.dat, constructed by you, and containing the value of MINS as the first (if not only) entry. So use a command like
hrs_mins < hrs_mins.dat
at the UNIX % prompt. This will run the executable file hrs_mins, taking input from hrs_mins.dat.
(iii) Modify your program from (ii), commenting out the `READ` statement, and inserting an `OPEN` statement with an associated `READ`. Re-run the program using the same file hrs_min.dat as in (ii) but without using "<" .

[If selected for handing in then handin (a) the program for (iii), and (b) for each of (i), (ii) and (iii) the commands used to compile and run, and corresponding output including prompt if any.]

7.† Write a Fortran program `COIN_DISPENSER` (in file coins.f90) that will read in a price `P` and amount tendered `A` and determine the corresponding numbers `N100, N50, N20, N10, N5`, of coins of various types in the change `C` (with a minimum number of total coins being used). Assume both `P` and `A` are integer numbers of cents, already rounded off to the nearest 5c value. Also assume `P` and `A` are no greater than $2 in value, so that `C` will be less than $2 . For example, when the price is $1.15, and the amount tendered $2, then `N100=0, N50=1, N20=1, N10=1, N5=1`, and your program should yield output like

```
CHANGE:  0 × $,  1 × 50c,  1 × 20c,  1 × 10c,  1 × 5c .
```

For input use `READ*`, and for output use a `FORMAT` statement with format `I2` for each of the coin amounts. The program should produce a single line of output like the above, being a mixture of headings and results. Use only integer variables and integer arithmetic throughout, and use the variable and file names stated above. Do not use any Fortran intrinsic functions.

Proceed as in Q6(i),(ii),(iii), but inputting `P` and `A` equivalent to 55c and $2 . For (ii) and (iii) use a file coins.dat. [If selected for handing in, then proceed analogously to the note in Q6.]

8. In (i) and (ii) below, express each given value in the specified FORMAT. Indicate blanks by the letter b.

```
            Value      Format   Output
Example     1000.3      F8.2    b1000.30
```

```
(i)     136         I4
(ii)    163.21      F8.1
```

In (iii) and (iv) below, show the output from the WRITE statements. Clearly indicate the vertical spacing as well as the horizontal spacing. Assume the variables TIME = 10.21, RESP1 = –0.12345, and RESP2 = 14.73 are stored exactly.

(iii)
```
      WRITE(6,5) TIME, RESP1, RESP2
5     FORMAT(F6.2,5X,2F7.4)
```

(iv)
```
      WRITE(6,1) TIME, RESP1, RESP2
1     FORMAT('EXPERIMENT RESULTS'/X,'TIME',2X,&
             'RESPONSE 1',2X,'RESPONSE 2'/&
              1X,F4.2,2E12.3)
```

Correction: /1X

9. (i) Suppose that the statements

```
      READ(*,7) ID, HT, WIDTH
7     FORMAT(I2,F4.1,2X,F4.2)
```

are processed with an input data line

```
123.456.78.91
```

State the values that will be stored in the variables ID, HT, WIDTH.

(ii) With the same variables and format statement as in (i), show the output line that would result from each of the following statements. (Use b to indicate blank places if any.)

(a) `WRITE(*,7) ID,HT,WIDTH`

(b)[A] `PRINT '(I3,F0.0,F0.1)', ID,HT,WIDTH`

In (iii) and (iv) below, specify how many input data lines are required for the given statements to correctly READ into the 4 variables. Indicate which variables should be on which line, and which columns they should occupy.

(iii)
```
          READ(*,14) TIME, DIST, VEL, ACCEL
   14     FORMAT(3F6.2)
```

(iv)
```
          READ(*,3) TIME, DIST
          READ(*,3) VEL, ACCEL
    3     FORMAT(F6.3)
```

(v) How many input lines would be required if the 'READ' commands in (iii) and (iv) were replaced by 'READ*,' or equivalently 'READ(*,*)' ?

10.† Suppose that the coordinates of three points (X_1, Y_1), (X_2, Y_2), (X_3, Y_3) are on one data line in a file triangle.dat, in columns as indicated below:-

columns	1-4	6-9	11-14	16-19	21-24	26-29
coordinates	X_1	Y_1	X_2	Y_2	X_3	Y_3
format	xx.x	xx.x	xx.x	xx.x	xx.x	xx.x

This file can be copied via the command

`cpi $mc3fp/triangle.dat .` (Note the second dot .)

Write a program to read the coordinates, and compute and print out the area of the triangle having the points as vertices. You may assume the formula

$$\text{AREA} = \tfrac{1}{2}\left| X_1 Y_2 - X_2 Y_1 + X_2 Y_3 - X_3 Y_2 + X_3 Y_1 - X_1 Y_3 \right|$$

[Aside: Give a 2D vector product interpretation of this formula.] Your program should print the coordinates and area strictly in the format:-

```
          <2 blank lines>
<8 blanks>Triangle vertices:
            (1) xx.x, xx.x
            (2) xx.x, xx.x
            (3) xx.x, xx.x
          <1 blank line>
          Triangle Area:
            xxxx.xx
```

(i) Test your program by reading the coordinates from the data file using the UNIX '<' directive.

(ii) Modify the program, using `OPEN` to read the data file. Compile and run the modified program.

[If selected for handing in, then for (i), hand in the (a) program, (b) data file, (c) processing commands and output. For (ii) you need not hand in the entire program, but just any lines differing from what you have already included in (i)(a), plus the new processing commands and output.]

11.† Write a program that reads a value of N and then computes a `REAL` approximation to N! using *Stirling's formula*

$$N! \sim \sqrt{2\pi N}\left(\frac{N}{e}\right)^N$$

which applies for positive integer N. [In this context, the tilde notation "~" means "is asymptotically equal to", and Stirling's formula is a so-called asymptotic approximation which is accurate as $N \to \infty$.] Your program should (a) issue a prompt and read N from the keyboard; (b) use `INTEGER` variable `N`, a `REAL` variable `FACN` say, for N! , and use mixed mode arithmetic in Stirling's formula; (c) use only list-directed `READ*` and `PRINT*`; (d) with appropriately chosen arguments, use the Fortran intrinsic functions `ACOS` (which approximates the mathematical function $\cos^{-1}$, i.e. arccos) to approximate π , and `EXP` to approximate *e*; and (e) output a single line result like:-

```
..! IS APPROXIMATELY ......
```

Run your program 5 separate times using N = 1, 7, 14, 23, 36. Compare with N! from your pocket calculator and briefly comment on the accuracy of Stirling's formula, and any error message. [Note: your program should not use loops or double precision variables.]

12. F90 has various intrinsic functions that involve integers, including `INT` (integer part of, i.e. nearest integer towards zero), `NINT` (nearest integer), `CEILING` (nearest higher integer), `FLOOR` (nearest lower integer). If `X=-3.4`, state the values of (i) `INT(X)`, (ii) `NINT(X)`, (iii) `CEILING(X)`, (iv) `FLOOR(X)`.

13.[A] The Fortran90 Standard allows for two forms of source code, namely FREE form and FIXED form. The latter is a slight extension of earlier Fortran standards, and allows near compatibility with those. The basic rules for the preferred FREE form source code are:-

(i) Statements may occur anywhere in columns 1–132. Characters after column 132 are ignored. Statement labels are numeric, up to 5 digits, and are placed to the left of the statement.
(ii) Comments must be preceded by an exclamation mark (!). Any characters following a !, which is not part of a character constant enclosed in quotes, are treated as a comment.
(iii) A line whose last non-blank character is an ampersand (&), which is not part of a comment, is regarded as part of a statement that is continued on the next line. The continuation line(s) of the statement may optionally have & as the first non-blank character; blanks, if any, preceding such leading &, are not treated as part of the continued statement.

The basic rules for the redundant and not-recommended FIXED form source code are:-

(i) Statement labels, if any, must be numeric and in columns 1–5. Apart from possible continuation characters – see (iii) below – other parts of non-comment statements must be in columns 7–72. Characters after column 72 are ignored.
(ii) Any characters following a !, which is not in column 6, and is not part of a character constant enclosed in quotes, are treated as a comment. The presence of an asterisk (*) or C in column 1 indicates that the whole line is a comment.
(iii) A statement that requires more than a single line may be continued on to following lines by placing any non-blank character other than zero (0) in column 6 of the continuation lines.

Suppose that it is required to write code that can be compiled either as FREE or FIXED form. State rules governing (i) statement labels and positioning of statements, (ii) comments, (iii) continuation statements, that this code would have to obey.

8.2 F90 Exercises Set 2

NOTE: Do not use arrays, subprograms or statement functions in any question of this set unless explicitly stated in the question. Assume Fortran default types (i.e. INTEGER, REAL), determined by leading characters of variable names, unless otherwise specified.

1. Given the values A = 7.23, B = 14.35, I = -12, K = 5, determine, with brief reasoning, whether the following logical expressions are 'true' or 'false'
   (i) `A + B >= 6.5`
   (ii) `(ABS(-A+4*I)+ B) .LE. K*I`
   (iii) `ABS(I) > 30.OR.K/2.GT.2`
   (iv) `.NOT.(I >= K).AND.MOD(500,K) /= 0`

2. (i) In (a), (b) and (c) below, give Fortran statements that perform the steps indicated.
   (a) If the cosine of `ANGLE` is greater than 0.8, go to statement number 150.
   (b) If the value of `DEN` is less than 1E-2, write the message 'LIGHT AS A FEATHER'.
   (c) If `DIST` is less than 50.0 and `TIME` is greater than 10.0, increment `TIME` by 0.05. Otherwise increment `TIME` by 2.0.

   (ii) (a) How many values of `TEMP` will be read by the following program segment?

```
      NUM = 2
      DELTA=3.5
      ICOUNT=0
2     READ*, TEMP
         ......
        ICOUNT=ICOUNT+1
        NUM = NUM + DELTA
         ......
      IF(NUM.LT.30) GOTO 2
```

   (b)[A] As for part (a) but with the second-last statement shown replaced by

```
        NUM = NUM + MOD(25-ICOUNT,NUM)
```

3.† A small test rocket is being designed to mimic a large retrorocket that is intended to permit 'soft' landings. The designers have derived the following equations to predict the performance of the test rocket.

$$\text{Acceleration in m/sec}^2 = 1.42 - 0.005\,t^2 + \frac{2.02\,t^{2.751}}{9995}$$

$$\text{Velocity in m/sec} = 1.42\,t - \frac{0.005\,t^3}{3} + \frac{2.02\,t^{3.751}}{3.751\times 9995}$$

$$\text{Distance in m} = h_0 + \frac{1.42\,t^2}{2} - \frac{0.005\,t^4}{12} + \frac{2.02\,t^{4.751}}{4.751\times 37491}$$

The distance equation gives the height above ground level at time *t* secs after launch, and the first term h_0 =30 is the height in metres above ground level of the launch platform. In order to check the predicted performance, the rocket will be 'flown' on a computer, using the derived equations. You are required to write and run a program covering a maximum flight of 2 mins. The results are to be printed strictly in the following format:-

```
   TIME      ACCELERATION      VELOCITY      DISTANCE
   (SEC)      (M/SEC**2)        (M/SEC)         (M)
    0.00          ...              ...          ...
  xxx.xx         xx.xx           xxxx.xx       xxxx.xx
    ...          ...              ...          ...
```

Increments of time are 5 secs, from launch through the ascending and descending portions of the trajectory until the rocket descends to within 25 metres of ground level. Below that level the time increments are 0.2 secs. If the rocket impacts the ground, the program is to stop immediately impact is detected. Adhere to the logical structure outlined in the incomplete file $mc3fp/rocket.inc. Thus, use the command

```
cpi   $mc3fp/rocket.inc   rocket.f90
```

to copy rocket.inc from the directory $mc3fp to your current directory under the name rocket.f90 .

(i) Edit rocket.f90 and fill in the missing details to make a complete program. Use GOTO to repeat the computations for successive times. Compile and run the program to generate the required table.
(ii) Modify your program for (i) replacing the GOTO-loop by a DO-loop, retaining but commenting out the GOTO. Test run the new program.

[Notes: The coefficients in the model are given to at most 4 leading digits and some much less. So not all digits shown in the table will be significant. If selected for handing in, then hand in only the program for part (ii), with part (i) lines commented out, together with processing commands and output for part (ii).]

4.† **[**If handing in, then handin only the program and 3 outputs for part (iii), or for part (iv), not both.]

A parachutist drops from a stationary balloon at an altitude of 500 metres, and falls in a spread-eagled position for a time t_o seconds, the associated terminal speed being 54 ms^{-1}. The chute is then opened to brake the fall and to allow a terminal speed of only 5 ms^{-1}. Assuming that the aerodynamic drag is proportional to the square of the speed and taking the *z*-axis vertically downwards from the balloon at $z = 0$, the equations of motion can be integrated (e.g. see P.D. Thomas, *Mathematics Applied to Mechanics* if interested) to give

$$\left.\begin{aligned} z &= \frac{U_1^{\,2}}{g}\ln\{\cosh(gt/U_1)\} \\ \dot{z} &= U_1\tanh(gt/U_1) \end{aligned}\right\} \text{ for } 0 \le t \le t_0;$$

$$\left.\begin{aligned} z &= z_0 + \frac{U_2^{\,2}}{g}\ln\left[\cosh\{g(t-t_0)/U_2\} + (u_0/U_2)\sinh\{g(t-t_0)/U_2\}\right] \\ \dot{z} &= U_2\left[\frac{U_2\sinh\{g(t-t_0)/U_2\} + u_0\cosh\{g(t-t_0)/U_2\}}{U_2\cosh\{g(t-t_0)/U_2\} + u_0\sinh\{g(t-t_0)/U_2\}}\right] \end{aligned}\right\} \text{ for } t \ge t_0;$$

where $g = 9.8\ \mathrm{m\,s^{-2}}$, $U_1 = 54\ \mathrm{m\,s^{-1}}$, $U_2 = 5\ \mathrm{m\,s^{-1}}$,

$$z_0 = \frac{U_1^{\,2}}{g}\ln\{\cosh(g\,t_0/U_1)\}, \qquad u_0 = U_1\tanh(g\,t_0/U_1).$$

(i) Write a program to tabulate $t, z, \dot{z}$ from $t = 0$ to $t = 15$ in 1s intervals and then (in the same run) from $t = 20$ upwards in 10 s intervals until landing has occurred, when an appropriate message should be printed. Use the formulae for z and $\dot{z}$ as shown above. Your program should be based on the incomplete program in the file $mc3fp/chute.inc which you can copy via the command

```
cpi  $mc3fp/chute.inc  chute.f90
```

The program should produce a headed table strictly in the following format:-

```
   TIME        DISTANCE        VELOCITY
  (SEC)          (M)           (M/SEC)
   0.0           ...              ...
  xxx.x         xxxx.xx         xxx.xxx
   ...             ...             ...
```

Run your program for $t_0 = 0, 5, 10$. You should find that your program crashes due to floating point *overflow*. This is due to the fact that as t increases for $t > t_0$ both the cosh and sinh terms in z and $\dot{z}$ exceed the OFL (overflow level), i.e. the largest number that can be stored in single precision real arithmetic (about 10^{38} in the IEEE standard). As a check of your program, compare with the correct values $z = 117.38$, $\dot{z} = 5.156$ at $t = 6$ for the case $t_0 = 5$.

(ii) The overflow experienced in (i) can be overcome in various ways. One way is to use double precision calculations in which the OFL is about 10^{308} in the IEEE standard. To declare that all real variables starting with letters in the ranges A-H, O-Z are double precision, just insert the type declaration

```
IMPLICIT DOUBLE PRECISION (A-H,O-Z)
```

after the `PROGRAM` statement in chute.f90 , and before any explicit type declarations. Recompile and run your program to see that this removes the overflow problem.
[Note:- The use of `DOUBLE PRECISION` does not necessarily lead to higher accuracy. In the current question g=9.8 is good to at most 2 leading digits, so at most about 2 leading digits shown in z and $\dot{z}$ in the tables are significant. However, showing somewhat more digits in the tables is useful for comparing with known answers, and providing a smoother variation of the distance and speed for plotting.]

(iii) Whilst double or higher precision may overcome overflow problems, it is generally better, if possible, to reformulate the calculations so that overflow is avoided in single precision. To this end, copy your file from (ii) under the new name chute2.f90 and edit the latter. First comment out the `IMPLICIT` declaration of part (ii) so that only single precision is used. Then reformulate the expressions for z and $\dot{z}$ for $t > t_0$, since that is where the overflow occurred. Suppose $x=g(t-t_0)/U_2$. Then for $\dot{z}$ divide numerator and denominator by $\cosh x$ (which is never zero) and rewrite to avoid overflow. For z, use an identity **like**

$$\log(\cosh x) = \log[\frac{e^x}{2}\ (1+e^{-2x})] = x+\log(\frac{1+e^{-2x}}{2})$$

For larger t and hence larger x, this may result in *underflow*, i.e. e^{-2x} may be smaller than the UFL (underflow level), the smallest nonzero magnitude that can be stored, and which is about 10^{-38} in single precision. However, by default many F90 systems set numbers with magnitudes < UFL to zero and associated program crashes do not occur.
Run chute2 for the same 3 values of t_0 as in (i) and check from your tables that to the nearest

10s, landings occur at t = 100s, 80s, 40s respectively.

(iv)[A] Your program uses 2 `IF` constructs testing one or both of the variables `t` and `t0`. In the current question no problem arises since `t` and `t0` take only integer values. But in general, to avoid errors, one should allow for roundoff when testing expressions, e.g. `t` , that have been derived by floating point calculations. For the current question, roundoff errors can be avoided in various ways, e.g. (a) by using `INTEGER t,t0` and allowing mixed-mode arithmetic, in calculating z etc; or (b) by using `REAL t,t0`, and then using the intrinsic function `NINT` (defined on p.75) in the `IF` tests. However, a more general and robust way is to (c) utilize a variable `EPS`, in a similar way to that described in Section 7 of fortut4. The variable `EPS` should have a value greater than any expected roundoff but less than the increments in `t`. Method (c) allows the same program to be used when `t` and `t0` take quite general non-integer values. Modify your program from (iii) by employing method (c). Compile and run your program for the same 3 values of t_0 as in (i).

5.†

(i) Write a complete program that will read and process a student registration file, which you can obtain via

```
cpi   $mc3fp/student.dat   .
```
(Note the second dot .)

This file contains 4-digit integer student identification numbers in columns 1–4 and the whole hours completed in columns 8-10, for each student that attends the university. The file does not have any end-of-data flag, so you should use a `READ` statement with an `END` directive. Use a `GOTO` to read successive lines of the file. Use a `CHARACTER` variable `CLASS` of length 9 characters, declared by

```
CHARACTER  CLASS*9
```

to take values 'FRESHMAN', 'SOPHOMORE' etc. Use a block `IF` construct to classify each student according to the following table:-

Classification	Hours completed
FRESHMAN	0 < hours ≤ 30
SOPHOMORE	30 < hours ≤ 60
JUNIOR	60 < hours ≤ 90
SENIOR	90 < hours
UNKNOWN	otherwise

Your program should print the following report (where leading zeros if any need not be printed in the ID column):-

```
                  < 2 blank lines >
                 REGISTRATION REPORT
                 -------------------
 STUDENT ID        HOURS COMPLETED        CLASSIFICATION
   XXXX                 XXX                   XXXXXXXXX
   ....                 ...                     .........
   ....                 ...                     .........
```

(ii) Modify your program so that it uses a `DO`-loop rather than `GOTO`. Comment out the replaced lines rather than delete them. Test run the new program.

(iii)[A] Redo (ii), but modify and rerun your program so that a final summary report is also printed following the registration report, with format:-

```
                    < 2 blank lines >
                  REGISTRATION SUMMARY
                 ----------------------
                 FRESHMEN            XXXX
                 SOPHOMORES          XXXX
                 JUNIORS             XXXX
                 SENIORS             XXXX
                 UNKNOWNS            XXXX
                                    -----
                 TOTAL STUDENTS     XXXXX
```

(iv)[A] Redo (iii) but use a `CASE` construct rather than an `IF`.

[If selected for handing in, then submit only one program and one lot of output, as described in the assignment specifications.]

6.

(i) In (a), (b) and (c) below, indicate the number of times that statements would be executed if placed inside the associated `DO` loops:-

`(a)  DO KTR=3,26,4` `(b)  DO LL=5,-203,-5` `(c)  DO I=10,5,4`

(ii) Find the value of `COUNTR` after the following nested loops are executed. Assume `COUNTR` is type `INTEGER`.

```
COUNTR = 0
DO MM=25,5,-2
  COUNTR = COUNTR + 12
  DO LL=2,50,4
    COUNTR = COUNTR - 1
  ENDDO
ENDDO
```

(iii) What is the value of L1 after the following statements are executed?

```
      L1 = 0
      I1 = 15
      I2 = 10
      DO 100 I=1,I1
        IF(I.LT.I2) L1 = L1 + 2*I
100   ENDDO
```

(iv)[A] Redo (iii) with `L1=100, I1=500, I2=400.`

7. For the following REAL variable F dependent on REAL variables X, Y, construct a double DO-loop to print a table of F, X, Y; showing 3 numbers per line, for the specified X, Y ranges.

$$F = \frac{X^2 - Y^2}{2XY} \; ; \quad X = 1.0,\ 2.0, \cdots, 20 \; ; \quad Y = 0.5,\ 0.75,\ 1.0, \cdots, 2.5$$

Do this by using only `INTEGER` loop control variables. [Note: The F77 standard allows `REAL` loop indices, but the F90/95 standards do not. Use of `REAL` loop control variables is dangerous, since it may produce an incorrect number of loops due to round-off error.]

8.† The function $\sin^2 x$ can be represented by the series

$$\sin^2 x = x^2 - \frac{2^3 x^4}{4!} + \frac{2^5 x^6}{6!} - \cdots = \sum_{n=1}^{\infty} u_n = \sum_{n=1}^{\infty} \frac{(-)^{n+1} 2^{2n-1} x^{2n}}{(2n)!}$$

Show that for $n > 1$, the n^{th} term u_n can be obtained from u_{n-1} via the *recurrence relation* $u_n = -2x^2 u_{n-1}/[n(2n-1)]$. Design and run a complete program (with no `GOTO` statements) that (a) prompts for and reads x from standard input, (b) uses a DO-loop and the recurrence relation to evaluate the partial sums S_N (N = 1,2,3,...,7) of the series, and (c) prints S_N for each N and, for comparison, an accurate approximation to $\sin^2 x$ obtained by using the Fortran intrinsic function `SIN(x)` Your output should have the form:-

< blank line >

```
COMPARISON OF VALUES OF SINE SQUARED
```

< blank line >

NUMBER OF TERMS	SERIES SUMMATION	INTRINSIC FUNCTION	ABSOLUTE DIFFERENCE
1	---	---	---
2	---	---	---
.	.	.	.
.	.	.	.
7	---	---	---

Use the intrinsic function ABS for the 4th column. Run your program 3 times using x=2 and (i) F8.5, (ii) E12.5, (iii) G12.5 for the floating point output columns. [You should find poor accuracy with the series for N=1, but increasing accuracy for larger N .]

[If selected for handing in, then hand in the program for (iii) with modifications for (i), (ii), included but commented out. Hand in all 3 outputs and the processing commands.]

9.†[A] Reconsider the earlier Stirling approximation question Set1 Q11. Write a program that processes cases N = 1,7,14,23,36 in a single run, and includes for comparison a recursive evaluation of N! for each such N < 36. Make sure that each evaluation of N! re-uses N! already found for lower N, i.e. pay attention to computational efficiency as part of the exercise. The program should construct headings

```
                Recursive        STIRLING
N                  N!             APPROX
```

then use a `DO` loop containing an `IF` and a `CASE` construct. The `IF` is only to avoid a floating-point overflow crash, which occurs in N! for N ≥ 34 (on the present system), and which you should have observed for N=36 earlier in Set 1 Q11. Use the `CASE` construct to print out results in 3 columns for each of the above N < 36, and to print an appropriate message in the N! column when N = 36. Use G12.5 format for the floating-point columns. [Note: Whilst not part of the Fortran standards, most Fortran implementations include a library of system routines, including a routine that would trap overflow errors, and allow alternative action other than a program crash. This question does not require you to use that sort of error trapping, but merely to avoid calculating N! for those larger N that would cause a crash.]

10.

(i) In (a) and (b) below, indicate the value of each element in each of the arrays `LST` and `K` after executing the given sets of statements. If no value is assigned to a particular element, indicate that using a question mark.

(a)
```
DIMENSION  LST(6)
DO K=1,6
  J = 7 - K
  LST(J) = J/K
ENDDO
```

(b)
```
DIMENSION  K(5,4)
DO M=1,4
  DO N=1,4
    K(M,N) = MOD(4*M-N,3)
  ENDDO
ENDDO
```

(ii) Assume that `K` is a one-dimensional array of 50 `INTEGER` values, filled with data.

(a) Give Fortran statements to find and print the maximum element of `K` and its position, in the format:-

```
Maximum element is IIIII, at position II.
```

Do this 2 ways:- (α) without using any Fortran intrinsic functions, and (β) using the intrinsic functions `MAXVAL` and `MAXLOC` (see FORSUMM, p.68).

(b) Give Fortran statements to count the number of positive values, zero values, and negative values in K. The output form should be

```
XXX  POSITIVE VALUES
XXX  ZERO VALUES
XXX  NEGATIVE VALUES
```

(iii) Give Fortran statements to print the last 10 elements of an array M of given size $N \leq 1000$. For instance, if M contains 25 elements, the output form is

```
M(16) = XXX
M(17) = XXX
M(18) = XXX
..... = ...
..... = ...
M(25) = XXX
```

Do this problem 3 ways, i.e. using (a) an ordinary `DO` loop, (b) an IMPLIED `DO`-loop, (c) an array section.

(iv) Give a single array section assignment that will replace the first N elements of a 1D array `K` of size $M \geq N$, by the reciprocals of the last `N` elements in reverse order. For example `K(1)` is replaced by `1/K(M)`.

In parts (v) and (vi) that follow, consult the list of Common Intrinsic Procedures in FORSUMM, p.74, and the descriptions of array constructors in FORSUMM, p.70, and in fortut6. The `PROGRAM`

statement is optional and may be omitted in the 2 small test programs required here.

(v)†[A] Fortran does not have an intrinsic Factorial function. Write a Fortran program that utilises the `PRODUCT` function and an array constructor `(/…/)`, or `[…]`, to print *n*! . Your program should contain only 4 statements, namely a `DO N=10,40` statement, a `PRINT*` statement, an `ENDDO` and an `END` statement. The `PRINT` statement should print *n* and 2 `PRODUCT` results on the one output line, one using integer arithmetic, and one using real arithmetic. Use your program to determine the highest factorials that can be computed in integer and in real arithmetic before overflow occurs in each case. [Ans: 12!, 34!. Integer overflow occurs at lower magnitudes than floating overflow. Integer overflow is usually not trapped by Fortran, and so does not cause program crashes. The overflow bits are usually just left truncated in the internal representation of the integer, causing incorrect answers to be given. Thus integer overflow is indicated by disagreement of your integer and real answers. Integer arithmetic is clearly very dangerous and should not be used for general purposes. Since the `f90o` command may trap integer overflow, for this question use the more standard UNIX command
`f90 -o fname fname.f90`
instead. This produces an executable file fname from your program file fname.f90.]

(vi)†[A] Write a Fortran program utilising the `SUM` function, an array constructor and an implied-`DO` loop, to print the value of

$$\sum_{n=2}^{100}(1+2n)\log_{10} n$$

Your program should contain only 2 statements, namely a PRINT statement and an END statement. [The Fortran $\log_{10}$ function is `LOG10`. Ans: 18187.05]

11.†

(i) The reservations for an aeroplane flight are stored in a file flight.dat, which you can copy via

cpi $mc3fp/flight.dat . (Note the second "**.**")

The plane contains 38 rows with six seats in each row. The seats on each row are numbered one to six as follows:

1 Window seat, left side
2 Centre seat, left side
3 Aisle seat, left side
4 Aisle seat, right side
5 Centre seat, right side
6 Window seat, right side

The data file contains 38 lines of information corresponding to the 38 rows, each line containing six values corresponding to the six seats. A value 0 or 1 is assigned for each seat, representing empty or occupied respectively. Write and run a program that (a) reads the data into a two-dimensional `INTEGER` array `SEAT(38,6)`, and (b) finds and prints the seat numbers of all pairs of adjacent empty seats, one pair per output line. Seats separated by the aisle are not regarded as adjacent. If all three seats on one side of an aisle are empty, then 2 pairs of adjacent seats numbers should be printed. The program should use 1 `OPEN` and 1 double implied-`DO` loop to read the data, 1 nested pair of ordinary `DO` loops to scan and test each row of seats, and 1 `IF` construct without `ELSEIF` or `ELSE`, to detect and print adjacent empty seats. The `DO`-loop nested pair should set a flag which can then be tested after the inner loop, by 1 simple `IF` statement without `ENDIF`, to print when no empty pairs are found in a row. Output should be in the format:-

```
AVAILABLE SEAT PAIRS
--------------------
   ROW      SEATS
    XX       X,Y available
    XX       Y,Z available
    ..       ............
    YY       No adjacent seats available
    ..       ............
    ZZ       X,Y available
```

(ii)[A] Consider re-doing (i) but supposing that it is not known beforehand that there are 38 rows. Declare `SEATS` sufficiently large to hold any reasonable number of seats, and make your program determine the number of rows, before entering the row loop. For your answer to this part, do not show an entire modified progam but merely indicate any new or modified statements. Do not show the output if already shown as part of (i). For this part of the exercise you may not read the data file more than once, but you are not restricted to an implied-`DO` loop to read the data.

12.[A] Once upon a time, namely in 1999, a hyper-conservationist decided that in order to help save the forests, they would start re-using old calendars. For their purpose, 2 calendars were the same if they were both leap, or both non-leap, years and both started with January 1 on the same weekday.

(i) There are only 14 distinct calendars. Why? So the conservationist soliloquized as to how many years back they must go to get a calendar that could be re-used for 2000. (Alternatively, if you don't like Greenies, think of a computer nerd trying to beat the then imminent Y2K bug by setting their computer's clock-calendar back.)

(ii) Let day numbers 1, ..., 7 represent Sunday, ..., Saturday. If year L starts on day I (I=1,⋯,7), and year L-4 starts on day $I+x$ where $x > 0$, what is the value of $I+x$? (Express your answer using the mathematical *modulo* function.)

(iii)† Write and run a program to determine the most recent calendar before 2000 that was the same as the 2000 calendar, given that the start-day of year 2000 was a Saturday. The program should:-

(a) include a `DO` loop with index `LEAPYR` regressing from 2000; in steps of 4.

(b) compute an `INTEGER` variable `DAYNO`, being the first day of `LEAPYR`;

(c) printout `LEAPYR` and `DAYNO` in the format

```
2000    7
1996 ...
...
```

(d) immediately after the line for the calendar matching 2000 is printed, print an answer to the soliloquy in (i), then stop.

(iv)† For interest, and to cater for non-leap years, the intrepid conservationist decides to find the starting days of all years 2000 ... 1901, and to obtain the printout in a nicer format. Thus, modify your program in (iii) as indicated by the following.

(a) In the printout, days are to be indicated by name rather than number. This can be done by using a character array `DAYNAM` with 7 elements of length 3, declared as such via the type declaration

```
CHARACTER DAYNAM(7)*3
```

DAYNAM's elements can be assigned by 7 statements like DAYNAM(1) = 'Sun', etc. Alternatively, DAYNAM can be initialized in the declaration statement

```
CHARACTER :: DAYNAM(7)*3=['Sun','Mon','Tue',&
                          'Wed','Thu','Fri','Sat']
```

(b) For each `LEAPYR`, determine an `INTEGER` array `PDAYNOS(3)` containing the preceding 3 January-1 day numbers, i.e. for the 3 non-leap years preceding `LEAPYR`. The value `DAYNAM(PDAYNOS(i))` then contains the day name of the i^{th} preceding day.

(c) Output format is to be 25 lines:-

```
2000  leapyear  Sat; 1999  Fri; 1998 ...; 1997 ...
1996  leapyear  ...;.........;...
...
1904  ...
```

This is to be accomplished using a print statement in the LEAPYR loop of the form

`PRINT 100, ...,...,`(implied-DO loop for 3 non-leap years)

where the implied-`DO` makes use of `PDAYNOS`.

(v) On the basis of your output in (iv), empirically infer the period of recurrence of equivalent calendars for leap years. Discuss the periodicity or otherwise of recurrence of non-leap year calendars.

[Historical Note: Julius Caesar, with advice from the Egyptian astronomer Sosigenes, in 45 BC, defined the Julian calendar as having 365 days in normal years and 366 in leap years, i.e. those with year numbers divisible by 4. The solar year, i.e. the time for the Earth to pass from perihelion to perihelion (point of closest approach to the Sun of the Earth's elliptic orbit) is approximately 365.2422 days. The inaccuracy in the mean Julian calendar year of 365.25 days gradually accumulated so that in 1582 Pope Gregory XIII was led to pontificate that October 5-14 did not exist for 1582, thus allowing the calendar and seasons to be brought back into line. To avoid future disparity he introduced the Gregorian calendar, by adding a second correction to the Julian calendar: viz. that century years not divisible by 400 would not be leap years. Thus 2000 was a leap year, but 1900 was not. The Gregorian calendar spread rapidly throughout Catholic Europe, and gradually elsewhere. It was adopted in England in 1752, when a correction was made by skipping the days September 3-13. This caused daylight-saving-like protests, including street riots by mobs fearing that their lives were being shortened by decree.]

13.†[A] This question demonstrates 'internal files' and 'dynamic formatting'. Write and run a program to generate the Pascal triangle for the expansions of $(1 + x)^n$, n = 0,...,10. Use I4 formatting and produce a symmetric array in columns 4 to 44, being the extension of

```
          1
        1   1
      1   2   1
    1   3   3   1
..................
```

Your program should be based on the following ideas:-

(i) Declare a 23×1 array A and initialize it to zero via statements like

```
INTEGER, DIMENSION(-11:11) :: A=0
INTEGER :: A(-11,11)=0
```

These illustrate the F90 extended form of declaration statements. The **::** separator allows type (e.g. INTEGER) and other attributes (e.g. SHAPE) and initial values if any to be prescribed in a single statement. At most 11 coefficients of A will be printed per line, i.e. A_{-10} , A_{-8} , ..., A_0 , ..., A_8 , A_{10} when n = 10. Both A_{-11} and A_{11} are fixed at zero.

(ii) Your program should loop for n = 0,10, for each n:-
   (a) dynamically forming the format to be used;
   (b) recursively calculating the elements of A to be printed;
   (c) printing out the relevant elements of A using that format.

For n = 0, only one coefficient is printed, namely $A_0 = 1$. The central columns are 21–24, so the first row might be (but isn't) printed via

```
PRINT '(T21,I4)', A(0)      or      WRITE(*,'(T21,I4)') A(0)
```

Here the format has been included in the PRINT and WRITE statements as a character constant. Alternatively, the format can be stored in a character variable as in

```
CHARACTER C*20
C = '(T21,I4)'
PRINT C, A(0)      or      WRITE(*,C) A(0)
```

In fact, inside the n-loop, don't use "A(0)" or "T21", but functions of n, that produce the preceding when n = 0 and also work for other n. So you must dynamically determine the format for each n, and store it in a character variable with a statement like

```
C = '(T'//C1//',11I4)'
```

In place of the "11" you could put any integer larger than the maximum number of elements to be printed. In the above, "//" is the Fortran operator for concatenating character strings, and C1 is a character variable that holds the first output column number for line n+1 (n=0,...,10). Thus for line 2, C1 is '19', and C is '(T19,11I4)'; and similarly '(T17,11I4)' for line 3. To store numeric data in the character variable C1 you must convert the numbers into character data. This is best done using 'internal files', i.e. character variables, into/from which you can write/read data. The internal file name is simply used in place of a logical unit number. Thus for line 2,

```
WRITE(C1,'(I2)') 19
```

before defining C as above. Of course, your loop will have some simple integer function of n in place of the "19". Having stored the format in C, you can then

```
PRINT C, (A(ICOL),ICOL=...)   or   PRINT C, A(...:...:...)
```

8.3 F90 Exercises Set 3

For the computing exercises (†) in this set, use style and debugging tips as described in Appendix A p.63, which can also be viewed on the screen via the command cv $fortips. Unless otherwise stated, assume default variable types, based on first characters of variable names.

1. In (a)—(d). below, give the value returned by function `TEST`. Assume that `A=-5.4, B=-9.2, C=.3E2,` and `D=-61.3E-1` have been assigned in the program unit that references `TEST`.

```
FUNCTION  TEST(X, Y, Z)
!RETURNS 0.5 OR -0.5 DEPENDING ON THE VALUES OF X,Y,Z
   IF(X.GT.Y.OR.Y.GT.Z)THEN
      TEST = 0.5
   ELSE
```

```
            TEST = -0.5
         ENDIF
      END
```

(a) `TEST(B,B-1,B),` (b) `TEST(B,C,A),`
(c) `TEST(ABS(B),C,A),` (d) `TEST(COS(3-C),SIN(C),D**3)`

2. (i) Write a statement function, or a function subprogram, `ABSURD(A,B,X)` to compute

$$x^2 + \sqrt{1 + ax + bx^2}$$

(ii) Write a single Fortran statement using `ABSURD` to assign a value to

$$\text{SINFUL} = \frac{1}{\sin^2 y + \sqrt{1 + 2\sin y + 3\sin^2 y}}$$

3. (i) Write an external function subprogram `MAX_ELT(K)` whose input is an integer array `K(50)`, and which returns the maximum element of `K`. [Do not use the F90 intrinsic function `MAXVAL` which does this.]
(ii) Redo (i) using an internal function subprogram `MAX_ELT()` with no dummy arguments.
(iii) Give a Fortran statement that defines a statement-function (a) `RAD` to convert a given number *D* of degrees to radians; (b) `VOL` to compute the volume of a rectangular pyramid of given height *h* and base side lengths *a*,*b* .

4. In (i) and (ii) below, give the values returned in all the arguments `N, K, L` after each of the following subroutine calls to the subroutine `ANSWER`. Assume that `N(1) = 5, N(2) = 4, N(3) = 1` have been assigned in the program unit that calls `ANSWER`.

```
SUBROUTINE  ANSWER(N, K, L)
!RETURNS A VALUE FOR L BASED ON N AND K
   INTEGER  N(3)
   IF(K.LT.0)THEN
      L = K
   ELSE
      L = 0
      DO I=1, 3
         L = L + 2N(I)
      ENDDO
   ENDIF
END
```

(i) `CALL ANSWER(N, 18, NUM)`
(ii)[A] `CALL ANSWER(N, N(1), N(2))` (Hint: see fortut7 §4.)

5.† [If selected for marking, then handin only one of (i), (ii) or (iii). Also handin (iv) if required.]

A firm's employee records use a 1D array `SSN` to hold 50 social security numbers. The number of actual employees of the firm is never more than 50, and the array is always ordered from smallest to

largest. Each social security number is a 6 digit integer, and unused elements at the end of SSN are set to 999999. The array is initially stored in data file $mc3fp/ssnumbers.old .

Write a main program that reads ssnumbers.old, stores the header line in a character variable HEADING and social security numbers in SSN; prompts for and reads SSNO from the keyboard; calls an external subroutine according to the specifications in (i), (ii) or (iii) below, then writes a new social security file ssnumbers.new in the same format as ssnumbers.old, including the original unchanged heading. Use the same variable names SSN, SSNO in the subroutine as in the main program. No input to the program is allowed, other than SSNO and ssnumbers.old.

Test your program out with (α) SSNO equal to the fifth last employee number in ssnumbers.old, and (β) SSNO = 345678. Your answer file should comprise your program, ssnumbers.old, processing and output for (α) and (β), including ssnumbers.new when different from ssnumbers.old .

(i) When a new employee starts employment, the corresponding social security number SSNO is to be added to the sorted array SSN. To accomplish this, your main program should

```
CALL  ADD(SSNO,SSN)
```

where ADD is an external subroutine that uses 2 DO-loops only. The first DO should loop upwards (i.e. in order of increasing index) through SSN, and contain 1 IF-ELSEIF-ENDIF construct. The IF should test whether SSNO already exists in SSN, and if so print the message

```
**  INPUT ERROR: EMPLOYEE ..... ALREADY PRESENT ON FILE **
```

and immediately STOP. The ELSEIF should efficiently test to find the insertion place for SSNO, add SSNO to SSN after pushing the higher magnitude numbers in SSN up one position, discarding the old SSN(50) (which you may assume was only 999999, since no more than 50 employees is ever allowed). The pushing should be done in a single array-section assignment statement. After the insertion, the loop should be EXITed. The second DO should use negative stride to loop in order of decreasing index through SSN. This loop should use 1 simple IF without ENDIF to EXIT once the new number of employees is effectively determined. Then ADD should print a screen message like

```
**  EMPLOYEE xxxxxx ADDED: xx EMPLOYEES NOW ON FILE  **
```

(ii) When an employee terminates employment, the corresponding social security number SSNO is to be removed from the sorted array SSN. To accomplish this, your main program should

```
CALL DROP(SSNO,SSN)
```

where DROP is an external subroutine that uses 2 DO-loops. The first DO should loop upwards (i.e. in order of increasing index) through SSN, and contain 1 IF-ELSEIF-ENDIF construct. The IF should test for the presence of SSNO, and if found remove SSNO by dragging the higher elements of SSN down one position, and fill in the last position. The dragging should be done in a single array assignment statement. The ELSEIF should efficiently detect if SSNO is absent, in which case it should print a screen message like

```
**  INPUT ERROR: EMPLOYEE ..... NOT PRESENT ON FILE **
```

and immediately STOP.

If SSN0 was present, DROP should EXIT the first loop. As in part (i), the second DO-loop

should use negative stride and a simple IF without `ENDIF` to `EXIT` once the new number of employees is effectively determined. Then `DROP` should print a screen message like

```
**  EMPLOYEE xxxxxx REMOVED: XX EMPLOYEES NOW ON FILE  **
```

(iii)[A] Modify (i) or (ii) to use a single external subroutine `ADD_or_DROP(SSNO,SSN)` that will (a) insert `SSNO` in the appropriate place in `SSN` if not already present, or (b) remove `SSNO` if it does already exist. Shuffling of `SSN` to open a hole for a new `SSNO`, or close a gap after `SSNO` is removed should be done using array-section assignments. The same structure used in (i) and (ii) should be employed. Only 2 `DO`-loops are permitted, the first containing an `IF-ELSEIF-ENDIF` construct, and the second containing a simple `IF` without `ENDIF`. The first `DO` should do the adding or removing, and assign a value to a `CHARACTER` variable `ACTION`. The second `DO` should be the same as the second loop in parts (i) and (ii), and after `EXIT`ing, `ADD_or_DROP` should use `ACTION` to print an appropriate screen message indicating the action taken and the adjusted number of employees. Test your program as in (α) and (β). Hand in as before but with a newly created file for each case (α), (β).

(iv) For any one of parts (i),(ii),(iii) above:-

(a) Consider rewriting your program using an internal rather than external subroutine, with no dummy arguments. List (one per line) the programming changes required (about 3 to 5 changes depending on your original program).

(b) Write and run the new program and check that it works.

6. F90 has an intrinsic subroutine `RANDOM_NUMBER(RN)` for generating random numbers uniformly distributed over (0,1). Here `RN` is declared as either a `REAL` scalar or array, and `RANDOM_NUMBER` fills it with pseudo-random numbers. The dimension of `RN` then, is the number of random numbers generated.

(i) Give Fortran statements using `RANDOM_NUMBER` to compute `N` random numbers from a uniform distribution over (a,b), rather than over (0,1). Assume that `N` has been initialized in a statement such as

```
INTEGER,PARAMETER :: N = ...
```

This statement gives a value to `N` at compile time, and the `PARAMETER` attribute ensures that `N` is fixed, i.e. cannot be changed later in the program. This allows `N` to be used in a declaration statement such as

```
REAL RN(N)
```

(ii)† Suppose that a coordinate pair (x,y) is chosen at random in the interval I $\{-1 < x,y < 1\}$. Show theoretically that the probability $P(x^2 + y^2 < 1)$ is $\pi/4$. Write and run a Fortran program that performs 20000 trials choosing (x,y) randomly in I. Your program should keep count on the number of trials in which $x^2 + y^2 < 1$. Use `MOD` to print out after every 1000 trials, showing the number of trials (1000, 2000, ….) and an associated estimate for π. The printout should be in two columns under appropriate headings. [This is a simple example of *the Monte Carlo Method*, a computationally expensive, low-accuracy but easy method for numerical simulation.]

7. [If selected for marking, then handin only one of (i), (ii), (iii).]

(i)† After a post-examination party, an inebriated student (hopefully not mc3) decides to depart from the lamp-post on which he has been leaning and try to stagger home. Being an ideal infinitely thin perfectly vertical lanky drunk he takes steps of exactly 1m, but due to orientation problems the angle between his direction and the kerbline varies completely randomly at each step. After taking 16 steps he is rather dismayed at his lack of progress. Assume there are no physical obstacles to impede him. Use the mean distance from a random number simulation of 1000 trials to estimate his probable distance to the nearest 0.1m from the post after 16 steps. Each trial should use a single call to `RANDOM_NUMBER` to generate an array of 16 random angles. [Hint: if `THETA` is an array, `SUM(COS(THETA))` is the sum of the elements in the array `COS(THETA)`.] Use the Fortran inverse cosine function `ACOS` to estimate π. Use the `MOD` function to help print out current estimates of the probable distance after every 100 trials, in table format:-

```
MONTE-CARLO ESTIMATES OF DISTANCES
     No. of          Mean distance
     trials          after 16 steps
      100                X.XXX
      ...                 ...
```

[The theoretical answer is approx 3.6m. If selected for handing in, then include your program, processing commands and output table showing distance to 3 decimal places.]

(ii)†[A] Modify your program in (ii) for the following changes. Assume (a) the inebriate retains sufficient visual and tactile sensations to ensure that all his steps have a non-negative forward component (in the x-direction say), (b) the length of each stride is random between 0.25m and 1.0m. Each trial may make only one call to `RANDOM_NUMBER`. Use the `MOD` function and a logical `.OR.` relation to print estimates of the probable distance from the lamp-post every 50 trials for the first 500 trials and every 100 trials thereafter up to 1000 trials.

(iii)†[A] Redo (ii) changed for a farewell party as follows. The inebriate is off-loaded at the airport and is placed face-forward at the central point of one end ('the starting line') of a 1.5m wide, 100m long footway moving at 0.5m/s. His choice of direction is random but with the following restrictions:- (a) given the anisotropic nature of his visual and tactile sensations, he ensures that all his steps must have a non-negative forward component (i.e. in the direction of motion of the footway), (b) if he collides with the side-walls then the point of intersection (on the moving footway) becomes his next take-off point and his next directional choice is correspondingly limited. Assume (α) his length of stride is random between 0.25m and 1.0m, except it is restricted by (b) above, (β) he hesitates randomly before steps (including the first) for 1s to 2s, (γ) the time taken per step is 2 s/m relative to the moving footway. Use a subprogram (internal or external) `STEP` which, taking into account the side boundaries, (a) uses the inebriate's current position (`XSTART, YSTART`) to determine the range of angles relative to the forward direction of his allowed next-step, (b) estimates his next position (`X,Y`), (c) calculates the time taken `T` for the step, including initial hesitation. Your main program (i.e. outside `STEP`) should estimate his probable number of steps taken, and probable distance to the nearest 0.5m from the starting line, approximately 2 minutes after being put aboard. Use 10000 trials, and use the `MOD` function to print estimates every 500 trials for the first 5000 trials and every 1000 trials thereafter. For simplicity regard each trial as finishing after the step that takes him through the 2 min barrier, i.e. don't bother proportioning the final step to make the time taken exactly 2 minutes. Produce a 3-column table with appropriate headings. Is the inebriate ahead or behind a sober person who bypasses the footway and moves at 0.7m/s parallel to the footway, and who starts at the same time and end as the inebriate?

8. The fraction f of neutrons in a certain fission reaction that have energy greater than a thresh-hold energy E_0 , is given by

$$f = 1 - 0.484 \int_0^{E_0} e^{-E} \sinh(\sqrt{2E})\, dE \quad .$$

(i) Consider using the Monte-Carlo method to estimate the value of f when E_0 = 1. What area would you choose for comparison, and why? [Hint: use a simple DO-PRINT* loop to show that the maximum value of the integrand in (0,1) is about 0.74 .]

(ii)† Use up to 20000 points, and print in 2-columns under headings POINTS and NEUTRON FRACTION. Print one output line every 1000 trials, showing the f-estimate to 4 decimal places. From your table, state a final estimate to 2 decimal places.

9.† (i) Two Applied Maths students arrange to meet at Wentworth one afternoon after completing their computer-based assignments. They are sure they will each arrive at Wentworth some time between 5 and 6 pm, but their precise arrival times are otherwise random and independent. They agree to wait only 10 minutes for each other after arriving at Wentworth and then to leave. Use 20000 random-number simulations to estimate the probability that they will actually meet. Each simulation should use only one call to `RANDOM_NUMBER`, and employ 2 `IF` constructs. The first should be a simple `IF` without `ENDIF` to test whether the virtual students meet. The second should be an `IF-ENDIF` block without `ELSE` or `ELSEIF`, and use the intrinsic `MOD` function to help print out the probability estimate every 1000 trials, to 4 decimal places. Print in two columns under the headings TRIALS and PROBABILITY. Base your program on the example program given on p.36 and use `INTEGER` variables `COUNT` and `TRIALS` as shown there. [The exact theoretical probability in the present case is 11/36 = 0.3055...]

(ii) Still using the `MOD` function, modify and re-run your program so that it prints the probability every 1000 trials for the first 10000 trials and every 2000 trials thereafter.

(iii) [A] Modify your program from part (ii) so that it applies to N students, each of whom is only prepared to wait 10 minutes until all the others arrive. For simplicity of programming, embed the `TRIALS` loop from Part (ii) inside a loop to cover the cases `N`=2,3,4,5. Make use of the intrinsic functions `MINVAL` and `MAXVAL` (see p.75), and use `RANDOM_NUMBER` to return `N` random numbers per call. Accumulate the resulting probabilities in a `REAL` array `PROBS(15,2:5)` and after all trials have been completed, output a 5-column table like that shown below. For the printing, use a `DO`-loop for `I=1,15` containing a `PRINT` statement. This loop should contain no `IF` statement, but instead use integer arithmetic. (Hint: first write the I-loop using an IF-ELSE-ENDIF construct. Then, noting that I/10 is zero in integer arithmetic unless $I \geq 10$, replace the `IF`-construct using an integer expression (perhaps but not necessarily using `MOD` or `MAX`) to generate 1,2,…,9,10,12,…,18,20 as I=1,2,…,14,15 .

```
No. of            Probability estimates
trials       N=2        N=3        N=4        N=5
-------------------------------------------------
 1000      0.3200     0.0790     0.0230     0.0000
  ...        ...        ...        ...        ...
20000      0.3088     0.0737     0.0160     0.0033
```

[You should find that the probability estimates decrease with increasing `N`, but your answers may differ somewhat from the above since they are only Monte-Carlo estimates.]

(iv) **A** In (iii), `RANDOM_NUMBER` is called 80,000 times, generating 280,000 random numbers as `N` loops from 2 to 5 . These numbers can be reduced to 20,000 calls and 100,000 random numbers if the `N` loop is embedded in the `TRIALS` loop. Modify your program from Part (iii) to do this. Use an array `ICOUNT(2:5)` to associate the success counter with `N`. Inside the `TRIALS` loop you may use at most 2 `IF` constructs. The first should be a simple `IF` without `ENDIF` to increment `ICOUNT(N)`; the second should be an `IF-ENDIF` block without `ELSE`, to test the `TRIALS` value. This `IF-ENDIF` should determine the row of `PROB` in which to place the probability, by using an integer expression.

[Note: whilst integer expressions are used instead of `IF` constructs in (iii) and (iv), the `IF` constructs are arguably better programming practice, being more transparent and easy to check.]

10.† The Maclaurin series (a.k.a. the Taylor expansion about $x = 0$)

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

converges for all x. Write a statement function `COS1(X)` which estimates cos$x$ by using this series up to terms of degree 6. Write a `FUNCTION` subprogram `COS2(X,N)` that estimates cos$x$ by using terms up to degree `N` in the series. The subprogram should use a recurrence relation to generate the successive terms, and choose `N` such thatsuccessive partial sums agree to 4 decimal places. This can be ensured by stopping when the add-on term $x^N / N!$ is less than 0.5E-4 in magnitude. Write and run a program that will produce a 5 column table with appropriate headings, showing the values of `X`, `COS1(X)`, `COS2(X,N)`, `N`, and the intrinsic function `COS(X)`, for `X` = 0,2,4,6,...,30. The value of `N` is determined in the subprogram `COS2` and returned to the main program which does the printing. Use format G11.4 for your output of cos$x$ estimates. Discuss the accuracy of your answers. [Note that whilst `COS2` converges, it does so to the wrong answers for larger `X`. It is typical that such *alternating series* (i.e. series of terms with alternating signs) suffer from large round-off errors for large `X`-values.]

11.† The n^{th} order *Bessel function*

$$J_n(x) = \sum_{m=0}^{\infty} \frac{(-)^m x^{n+2m}}{2^{n+2m}(n+m)!\,m!} \qquad (1)$$

arises naturally in many applications. (For example the natural frequencies of vibration of a circular drum skin are simply related to the roots of $J_n(x)$ divided by the skin radius.) The above series converges for all x. The aim of this question is to compare various different approximations for the zeroth order Bessel function $J_0(x)$. A skeleton program for (i) and (ii) can be obtained via the command

```
cpi  $mc3fp/bessel.inc  bessel.f90
```

Complete this program according to the specifications below. [If selected for handing in, then submit only part (i), or part (ii), that follow, but not both.]

(i) Write a STATEMENT FUNCTION called `SERIES8(X)` that estimates $J_0(x)$ by using the series (1) up to degree 8 terms. Write an external `FUNCTION SUBPROGRAM` called

`SERIES(X)` that estimates $J_0(x)$ by using the partial sum of M+1 terms in (1), i.e. up to degree 2M. The subprogram should use a recurrence relation to generate successive terms, and choose M such that the partial sums up to the M^{th} and $(M+1)^{th}$ terms agree to 4 leading digits. This requires stopping the summation loop when the magnitude of the last term added on, i.e. the $(M+1)^{th}$, is less than 0.5E-4 times the magnitude of the last partial sum. Write and run a program – call the main program `BESSEL` – that includes `SERIES8` and `SERIES`, and produces a 4-column table, with meaningful headings, showing the values of `X`, `SERIES8(X)`, `SERIES(X)`, and the J_0 estimate returned by the subroutine `BSSLJ` in the NSWC (US Naval Surface Warfare Centre) scientific subroutine library. The table should display results for `X` = 0.,2.,4.,6.,...,30., using format `G11.4` for the J_0 estimates. Notes: `BSSLJ` does not use (1), but an alternative approximation. The formula in `BSSLJ` has a maximum error less than 10^{-14} for the range of `X` being considered, but the accuracy actually obtained depends on the number of digits employed in calculating `BSSLJ`. Access the NSWC library via

```
f90o  bessel.f90  $nswclib
```

and `CALL BSSLJ(Z,0,W)` in `BESSEL`. Subroutine `BSSLJ(Z,N,W)` takes complex `Z`, integer `N`, and returns a complex value `W` for $J_N(Z)$. So your main program must declare `Z` and `W` to be `COMPLEX`, and assign `Z = CMPLX(X,0.)` (or equivalently just `Z = X` using mixed mode) inside the `X`-loop before the `CALL` to `BSSLJ`. Since $J_0(X)$ is real for real `X`, your table should just display `REAL(W)` as the estimate for J_0. Note that `X` is real, but the `X`-loop index must be an integer.

(ii)[A] Proceed as in Part (i) but with the following modifications.

(a) Change the argument list of `SERIES` to `X, M`. Reference to `SERIES(X,M)` then returns the value of `SERIES` and also of `M (2M` being the highest power used to construct `SERIES`.

(b) Copy your `FUNCTION SERIES(X,M)` subprogram to the end of the entire program file, change the subprogram copy's name to `DSERIES(X,M)`, and modify it internally to reflect this change.

(c) In your main program declare `DSERIES` to be `DOUBLE PRECISION` and introduce another double precision variable `DX` that will range through the values 0D0 to 30D0 (the double precision extension of the range 0 to 30) as `X` ranges from 0. to 30.0 (i.e. 0E0 to 30E0). In `DSERIES`, declare all `REAL` variables (including `X` and `DSERIES`) to be double precision - use `IMPLICIT` or explicit type declarations - see fortut8. Most constants in `DSERIES` must also be double precision, otherwise they might reduce the value of `DSERIES` to single precision accuracy. `INTEGER` constants are automatically converted to double precision in mixed mode expressions, but `REAL` constants like 1., if any, must be changed to 1D0 etc.

(d) Your final program should produce a 7-column table, with meaningful headings, showing the values of `X, SERIES8(X), SERIES(X,M1)`, the number of terms used in `SERIES, DSERIES(DX,M2)`, the number of terms in `DSERIES`, and finally the J_0 estimate returned by `BSSLJ`.

[You should observe as follows:- (α) The `SERIES8` approximation is inaccurate for $x > 2$ due to *truncation error* caused by stopping at degree 8. (β) The `SERIES` approximation converges, but is inaccurate for $x > 12$, due to *roundoff error*. Such errors will be discussed in Part 2 of the course. (γ) The `DSERIES` approximation agrees with NSWC, and is accurate across the whole range considered. However, note that the use of higher precision as in `DSERIES` merely defers the effect of roundoff error to values of x higher than those considered here.]

12.† [A] The task here is to piecewise-linear scale a set of at most 1000 marks in the range [0,100]. The scaling is done using 2 linear pieces, with range $[0 \le x \le \mu]$ mapping to $[0 \le y \le 60]$, and $[\mu \le x\ 100]$ mapping to $[60 \le y \le 100]$. Here x is a raw mark, μ is the average raw mark, and y is the scaled mark derived from x . (y is a piecewise linear function of x, comprising 2 pieces.) Proceed as follows:-

(i) In the usual fashion obtain copies of files marks.inc, ave.inc, scale.inc, marks.dat from directory $mc3fp. File marks.inc contains an incomplete main program that reads the marks from marks.dat, calculates their average using an external function ave.f90, and scales them using external subroutine `SCALE`, an incomplete version of which is contained in scale.inc . You should name your copies marks.f90, ave.f90 and scale.f90 .

(ii) Complete and modify marks.f90, ave.f90 and scale.f90 wherever "..." occurs. Note that subroutine `SCALE` must be 'portable', i.e. must work for any prescribed `Y0`, where `Y0` is the value to which μ is scaled.) To receive full marks, routines ave.f90 and scale.f90 should contain no `DO` loops or `IF` constructs. Avoid these by using array functions and constructs. If done this way, ave.f90 requires only a single executable statement, and therefore does not really warrant a subprogram. However, use a function subprogram as part of the exercise.

(iii) Compile the .f90 routines separately and link the associated .o files to produce an executable file called marks. Run the executable to produce output in the format

```
              RAW MARK     SCALED MARK
              ------------------------
                xx.x           yy.y
                xx.x           yy.y
                ....           ....
                ....           ....
-----------------------------------
AVERAGES        xx.x           yy.y
```

Your answer file should contain marks.f90, ave.f90, scale.f90 and the results. [Note: do not edit marks.dat, use it exactly as given.]

9. Appendix A: FORTIPS

Tips on STYLE and DEBUGGING of Fortran programs

Checklist:-

(1) TYPE CAREFULLY. Be sure not to confuse the letters I and l, and number 1; or letter O with number 0.

(2) CHECK VARIABLE TYPES, ARRAY DIMENSIONS in each program unit. Remember that variables starting with letters I,...,N are assumed INTEGER, and other variables REAL unless you declare otherwise.

(3) USE INFORMATIVE NAMES. Use names that indicate something about the object represented. For instance, a variable representing velocity might be named VEL rather than A, or X1, or something obscure. For checkability reasons it might be useful to use labels as given in the printed description of the problem that you are investigating. Use the optional PROGRAM statement to give a name to the program. Choose meaningful/descriptive names for all program units.

(4) COMMENT STATEMENTS. Use just sufficient comment statements and blank lines to improve program readability and organization. There should generally be comments at the beginning of the program units to describe aim, to describe the main variables, and to list subprograms used. The amount of commentary can be reduced by sensible choice of names as in (3).

(5) NO UNNECESSARY ARRAYS. Only use arrays where absolutely necessary. Otherwise you may needlessly complicate the program and waste storage space.

(6) INITIALIZE VARIABLES. Make sure all variables on the right side of the assignment statement have been previously initialized (i.e. given a value). Not all compilers set non-initialized variables to be zero by default. So play safe.

(7) ECHO INPUT. During program development, immediately after reading input data, print it back out for a check.

(8) AVOID LENGTHY/COMPLICATED STATEMENTS. Break long expressions into smaller expressions and recombine them in another statement. For example calculate a fraction by first calculating the numerator, then the denominator, and then finally dividing in a separate statement.

(9) INDENT STATEMENTS IN DO-LOOPS, IF and CASE CONSTRUCTS etc. This helps with readability as it facilitates locating the start and finish of such structures. See the example program after this checklist.

(10) USE INFORMATIVE INPUT/OUTPUT HEADINGS/LABELS. E.g. use (' X = ',F3.1) instead of just (F3.1), or use headings. Print physical units with corresponding numerical values. Use prompts like 'Enter values for X, Y'·

(11) CARRIAGE CONTROL IN FORMAT STATEMENTS. If intending to use Fortran carriage control, then when using a slash character (/) to skip one or more lines, remember to follow it by ' ' or the appropriate carriage control character. Example: FORMAT('1',3E16.4,3/,' ',2F10.3) means go to a new page, print according to 3E16.4, go 3 lines down (i.e. leave 2 blank lines), then print according to 2F10.3.

(12) AVOID MEANINGLESS DIGITS. Do not print more digits than are warranted. For example, if iterations are to stop when 4 leading digits of successive iterates are in agreement, then don't print more than about 5 leading digits, i.e. just enough to visually demonstrate convergence. In final answers generally do not print more digits than are possibly significant. Take into account the number of significant digits in the input data.

(13) LOCATE FORMATS WHERE THEY CAN BE EASILY SEEN. For FORMAT statements used by only a single READ or WRITE, place the FORMAT immediately following the READ/WRITE statement. When used by more than one READ/WRITE and the latter are spread out throughout the (sub)program then place the FORMATs in a block near the front end of the program.

(14) SUBPROGRAM ARGUMENT LISTS. Be sure that the variables in the main program are the same type (REAL, INTEGER, etc.) and have the same dimensions as the corresponding variables in the subprogram. Try to use arrays of the same shape in both the main program and subprograms. Check that the order of arguments in the CALL statement or function reference matches that in the SUBROUTINE or FUNCTION statement. For clarity and checkability in your own subprograms (as opposed to supplied subprograms from external libraries), use the same variables in the subprogram argument list as in the main program (or invoking subprogram). If using a provided subprogram with obscure variable names, then in your main program choose meaningful names as recommended in (3), rather than the general names used by the subprogram. When there are many arguments, place input arguments before output arguments.

(15) TEMPORARY PRINT/WRITE STATEMENTS. Make liberal use of these when developing your program. These can be very useful for isolating trouble spots, and are easily removed from the final program.

Example Program:-

The aim of the following program is mainly to illustrates a satisfactory style of programming. Many variants are acceptable; just make sure yours is at least as satisfactory as this. Concentrate on (i) readability, (ii) checkability, and (iii) efficiency; but not (iii) at the expense of (i) and (ii). To test this program out, copy the file $mc3fp/fortips.f90, then compile and link via 'f90o fortips.f90', and run via 'fortips'.

```
!=====================================================================
      PROGRAM QUADRATIC
!  Aim: to solve the equation A*X**2 + B*X + C = 0,
!       where A, B, C are real coefficients.
```

```
!-------------------------------------------------
!  Main variables:-
!  COEFF - Array containing A, B, C
!  AGAIN - Flag input by user to determine stop or repeat.
!-------------------------------------------------
!  Subroutine used:-
!  SOLVE - Solves quadratic and prints roots.
!-------------------------------------------------
      REAL COEFF(3)
      INTEGER AGAIN

10    PRINT*
20    PRINT*,'Input the coefficients A B C separated by blanks'
      READ*, COEFF
!Note: 3 numbers are expected since COEFF has 3 elements

!  Find roots
      CALL SOLVE (COEFF)

!  Try again?
      PRINT*
40    PRINT*,'Enter 1 to try another quadratic, or 0 to stop.'
      READ*, AGAIN
      IF (AGAIN.EQ.0) THEN
        STOP
      ELSE IF (AGAIN.NE.1) THEN
        PRINT*,"I don't understand you. Try again."
        GO TO 40
      END IF
      GO TO 20

      END
!=======================================================================
      SUBROUTINE SOLVE (COEFF)
!  Solves quadratic using standard formula and prints roots.
!  Checks the discriminant and prints messages relevant to real,
!  complex and double roots, and degenerate cases.
!-------------------------------------------------
!  Main variables:-
```

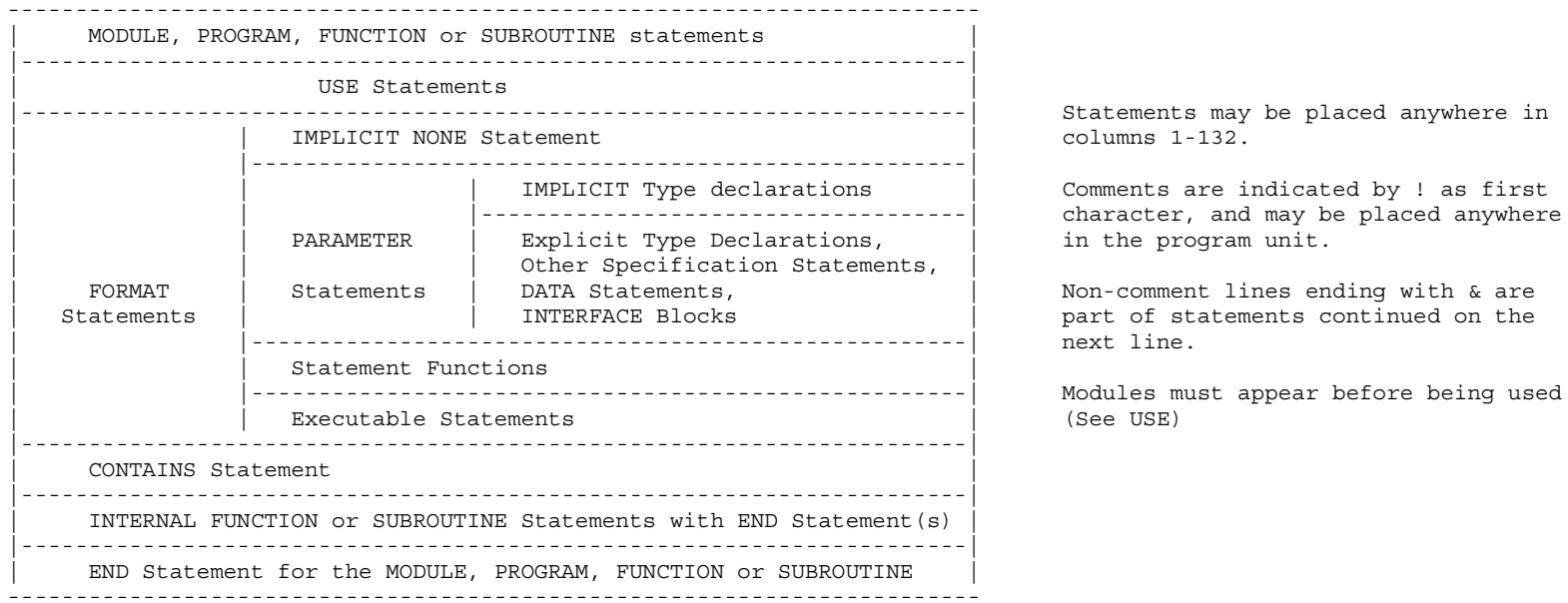
```
!  COEFF  Array of real coefficients of quadratic
!         A*X**2+B*X+C  in order A, B, C.
!  DISC   Discriminant  B**2-4*A*C
!  X1,X2  Real roots
!  Z1,Z2  Complex roots
!-------------------------------------------------
      REAL COEFF(3),X1,X2
      COMPLEX Z1,Z2
!Note: Arrays must be dimensioned in each subprogram in which they occur.
      PRINT 100, COEFF
100   FORMAT (/' Solving equation',ES14.6,SP,'*X**2',ES14.6,&
               '*X',ES14.6,' = 0'/)
!Note: SP format-directive forces signs to be printed even when +;
!      ES is scientific exponent, i.e. same as E but with 1<|mantissa|<10

! Check if quadratic is degenerate
      IF (COEFF(1).EQ.0) THEN
        PRINT*,'No X**2 term. The equation is at most linear.'
        IF (COEFF(2).EQ.0) THEN
          PRINT*,'No X term either.'
          IF (COEFF(3).EQ.0) THEN
            PRINT*,'The equation is trivial. Any X is a solution.'
          ELSE
            PRINT*, "Pull the other leg. It's a contradiction!"
          ENDIF
        ELSE
          X1=-COEFF(3)/COEFF(2)
          PRINT 110, X1
110       FORMAT(' There is only one root:'/G14.6)
        ENDIF
        RETURN
      END IF

!  Otherwise the equation is a proper quadratic. So solve it.
!  Test the discriminant
      DISC=COEFF(2)**2 - 4*COEFF(1)*COEFF(3)
      DENOM=2*COEFF(1)
      IF (DISC==0) THEN
        X1=-COEFF(2)/DENOM
```

```
        PRINT 115, X1
115     FORMAT(' There is one double root:'/G14.6)
      ELSE
        SRDISC=SQRT(ABS(DISC))
        IF (DISC.LT.0) THEN
          PRINT*,'There are two complex roots:'
          Z1=CMPLX(-COEFF(2),-SRDISC)/DENOM
          Z2=CMPLX(-COEFF(2),SRDISC)/DENOM
          PRINT 120, Z1,Z2
120       FORMAT(E14.6,SP,E14.6,'*I',SS)
!Note: Each complex number output requires two real format-specifiers
!Note: The SS format-directive suppresses printing of + signs.
        ELSE
          PRINT*, 'There are two real roots:'
          X1=(-COEFF(2)-SRDISC)/DENOM
          X2=(-COEFF(2)+SRDISC)/DENOM
          PRINT 130, X1,X2
130       FORMAT(G14.6)
!Note: G14.6 displays 6 leading digits in an F format if possible
!      and in an E format otherwise.
        ENDIF
      ENDIF
      RETURN
      END
!=====================================================================
```

10. Appendix B: FORSUMM

Fortran Summary Information Sheet (Subset of 1995 Fortran standard, unless otherwise stated; 1/2/04, RWJ, Carslaw 624.)
This document is not introductory, but rather a brief reminder for those who already substantially know F90/95.
STUDENTS MAY OPTIONALLY IGNORE ALL THOSE ITEMS MARKED BY # IN COLUMN 1.

ORDER OF STATEMENTS IN EACH MODULE, PROGRAM OR SUBPROGRAM



Names/labels & Statements

Variables and program unit names are 1-31 characters chosen from A-Z (case insensitive) and underscore _ .

```
 Statements occupy columns 1-132, and continue to the next line if the last non-blank character is &.
 If the first non-blank character on the next line is &, then the continued statement restarts after that &, otherwise it restarts
 at the beginning of the line.
 Statement labels range 0-99999. Logical unit numbers for files range 1-99.

         Expression operators
         --------------------
 arithmetic:     **  exponentiation;  * /  multiplication, division;  + -  addition, subtraction

 character:     //   concatenation

 comparison:   .GT. or >   greater than;   .GE. or >=  greater than or equal to
               .LT. or <  less than;       .LE. or <=  less than or equal to
               .EQ. or ==  equal to;       .NE. or /=  not equal to

 logical:      .NOT.     .NOT.A is true if and only if A is false
               .AND.     A.AND.B is true if and only if both A nd B are true
               .OR.      A.OR.B is true if either A or B or both are true
#              .EQV.     A.EQV.B is true if and only if A and B are both true or both false
#              .NEQV.    A.NEQV.B is true if and only if one of A and B is true and the other false
 ================================================================================================================================
 In what follows, certain lower case letters will be used to represent objects, as below:-
 a              an actual argument in CALL statements or function references, i.e. a variable, or FUNCTION or SUBROUTINE name.
 arr            an array name
 block          a sequence of Fortran statements
 clist          a list of constants separated by commas
 constr         any of the construct names DO, IF, INTERFACE, SELECT, WHERE
 d              array dimensions or shape
 e              an (arithmetic or logical) expression
 f              a format specification
 i              an integer expression
 ic             an integer or character expression, or range in form exp1:exp2
 iv             an integer variable
 nam            a name for PROGRAMs, FUNCTIONs, MODULEs, COMMON blocks, IF statements etc (1st char letter; max length 31 chars)
 p              a dummy argument, to be replaced by an actual argument a (see above).
 progu          any of the program unit headers PROGRAM, FUNCTION, SUBROUTINE, MODULE
 s              a statement label (number)
 st             any executable statement except DO, END, IF, ELSE, WHERE or similar
 typ            any one of LOGICAL, INTEGER, REAL, DOUBLE PRECISION, COMPLEX or CHARACTER
 u              a logical unit number (1-99)
 v              a variable name (1st char letter; max length 31 chars)
 vlist          a comma-separated list of variable names, array names or array declarators
```

Common FORTRAN Statements/Constructs (Avoid entries labelled "non-standard")

```
 arr(i1:i2:i3)            Rank-1 array section of elements arr(i1+i*i3), i=0,1,2,... maximum subscript not exceeding i2.
 arr(i1:i2:i3,i4:i5:i6)   Rank-2 array section as above; higher dimensions similarly. For READ or PRINT the order is 'array-element
                          order', i.e. 1st subscript varies fastest. For assignments the order is unspecified and may be parallel.
 (arr(iv),iv=i1,i2,i3)    'Implied DO-loop' for rank-1 array
 ((arr(iv1,iv2),iv1=i1,i2,i3),iv2=i4,i5,i6)   Implied DO-loop for rank-2 array, showing 'array element order' (i.e. columnwise)
 ((arr(iv1,iv2),iv2=i4,i5,i6),iv1=i1,i2,i3)   Implied DO-loop for rank-2 array, showing row-wise order. Higher ranks similarly.
 (/ vlist /)              Rank-1 'array constructor' with comma-separated elements in vlist (which may include IMPLIED DO-loops).
 [ vlist ]                Same as (/ vlist /). (Non-standard prior to Fortran2003)
 (vlist,iv=i1,i2,i3)      'Implied DO-loop where variables in vlist depend on iv. May be nested similarly to rank-2 array above.
#ALLOCATE(arr1(i1),arr2(i3,i4),...)   Dynamically allocates shapes to allocatable arrays that have been declared via a statement
                          such as REAL,ALLOCATABLE :: arr1(:),arr2(:,:) or REAL,DIMENSION(:,:) :: arr (See DEALLOCATE)
#BACKSPACE(u)             Rewind file on logical unit u one record (line).
 CALL nam(a1,a2...)       Calls (i.e. processes) the SUBROUTINE subprogram with name nam, passing the actual arguments a1,...  to
                          replace the dummy arguments in the SUBROUTINE statement. The arguments a1,... may be variables, or
                          subprogram names (see EXTERNAL and INTRINSIC). (E.g. CALL SUB1(A,B,SUB2,FUN1) would process subroutine
                          SUB1, passing variables A and B, and routine names SUB2 and FUN1.) (See also RETURN.)
 CASE                     See SELECT
 CHARACTER(len) vlist          'Explicit type declaration', that variables in vlist are type CHARACTER with length len, or v1
 CHARACTER v1*len1,v2*len2...  having length len1, v2 having length len2,...  If the length specifier is omitted 1 is assumed.
 CHARACTER(len) v1*len1,...    As above, using default length len when any of len1 etc are omitted.
#CHARACTER*len             Same as CHARACTER(len) but obsolete post F95.
#CLOSE(u)                  Close the file on unit u. (See also OPEN.)
#CLOSE(u,STATUS='DELETE'))    Close the file on unit u and delete it. ('DELETE' is case insensitive.)
 COMMON vlist              'Unlabelled COMMON'. Specifies that corresponding variables in vlist occupy the same 'common block'
                          of storage, for each subprogram containing the COMMON statement.
 COMMON /nam/vlist                      'Labelled COMMON'. Same as 'unlabelled COMMON' but
#COMMON /nam1/vlist1/nam2/vlist2...     associates the names nam,nam1,... with each COMMON BLOCK.
 COMPLEX vlist             'Explicit type declaration', that variables in vlist are type single precision COMPLEX.
#COMPLEX(KIND(1E0)) vlist Ditto
#COMPLEX(KIND(1D0)) 'Explicit type declaration', that variables in vlist are type double precision COMPLEX.
#COMPLEX(4) vlist   Common equivalent to COMPLEX vlist (4 suggesting two 4-byte REALs). (Non-standard, avoid)
#COMPLEX*8       Equivalent to COMPLEX (8 being total number of bytes). (Non-standard and outdated)
#COMPLEX(8) vlist   Common equivalent to COMPLEX(KIND(1D0)) (8 suggesting two 8-byte REALs). (Non-standard, avoid)
#COMPLEX*16         Equivalent to COMPLEX(KIND(1D0)) (16 being total number of bytes). (Non-standard and out-dated)
#COMPLEX(KIND=i)    Equivalent to COMPLEX(i), where typically i=4,8 (i-values are not specified by the F90-standard.)
 CONTAINS                 Indicates that the current program unit contains one or more internal procedures following CONTAINS.
 CONTINUE                 Causes no processing, but permits transfer of control when labelled.
 CYCLE                    Go to the next terminating statement (usually ENDDO) of a DO loop and continue the loop after that.
 CYCLE nam                Go to the terminating statement (usually ENDDO) of the DO loop named nam, and continue that loop.
 DATA vlist/clist/           Initializes the elements of vlist to the values in clist at compile time. (E.g. DATA A,B,C/1,2,3,4/
 DATA vlist1/clist1/vlist2/clist2/.../   sets A=1, B(1)=2, B(2)=3, C=4 if B has dimension 2. Equivalent to DATA A/1/B/2,3/C/4/.)
```

```
                                    DATA initializations occur once only, at compile time-time, i.e. prior to execution.
#DEALLOCATE(arr1,arr2)   Deallocate allocatable arrays arr1,... (See ALLOCATE)
 DIMENSION arr1(d1),arr2(d2)...          Specifies storage space for arrays. (E.g. DIMENSION A(2,3),B(0:10,-10:10) )
 DO s iv=i1,i2,i3        Executes a DO loop with start value iv=i1, and increment i3. When iv is increased to a value beyond i2,
                         the loop is bypassed, and processing continues after the statement labelled s. (The F77 Standard
                         permitted non-integer loop variables. Note also that |iv|>|i2| following completion of the loop. E.g.
                         DO 1 iv=-1,10,3   loops 4 times. The final iv in the loop is 8, and after looping ceases, iv=11 .)
 DO s iv=i1,i2           As above with i3=1
 DO iv=i1,i2,i3          As above but loop terminates with ENDDO statement rather than statement s.
#DO s WHILE (e)          As above but processing of loop terminates when logical expression e is false. (Obsolete)
#DO WHILE (e)            As above but loop terminates with ENDDO statement rather than statement s. (Obsolete)
 DO                      Loops to the following ENDDO until some some statement internal to the loop causes a stop or exit.
 nam:  DO ...            As above but naming the loop nam.
 DOUBLE PRECISION vlist 'Explicit type declaration', that vbles are DOUBLE PRECISION, precision about 15 digits, range 1E+-308
 DOUBLE COMPLEX vlist   'Explicit type declaration', same as COMPLEX(KIND(1D0)). (Non-standard)
 ELSE, ELSEIF            See IF
 END                     Ends a program unit or construct.
 END progu               As above, where progu is one of PROGRAM, FUNCTION, SUBROUTINE, MODULE   |
 END constr              As above, where constr is one of DO, IF, INTERFACE, SELECT, WHERE       |
 END progu nam           As above but even more specific, e.g. END FUNCTION FUN                  | The blank after END is optional
 END constr nam          As above but even more specific, e.g. END DO OUTER                      |
 EXIT                    Go to the statement after the next terminating statement (usually ENDDO) of a DO loop.
 EXIT nam                Go to the statement after the terminating statement (usually ENDDO) of the DO loop named nam.
 EXTERNAL nam1,nam2...   Used in invoking (sub)programs to declare FUNCTION or SUBROUTINE names nam1.. that are arguments in
                         SUBROUTINE or FUNCTION statements. Nam1 etc. are user-supplied subprograms (but not statement
                         functions). As opposed to INTRINSIC.
 FORMAT(specification)   Describes format in which one or more records are to be transmitted; a statement label must be present.
                         See FORMAT specifications and carriage control characters towards the end of this summary.
 prefix FUNCTION nam(p1,p2,...)       Header line of a function subprogram, indicating the program name nam and dummy
                          arguments p1,p2,... Each of these is a variable or subprogram name. (See CALL, RETURN and EXTERNAL.)
                          The prefix if present may be a type (e.g. REAL etc.), or RECURSIVE
 GOTO s                  Transfers control to statement number s.
 IF(e)st                 'Logical IF'. Executes the statement st if the logical expression e is true.
                         Otherwise st is bypassed and processing continues with the next statement.
 IF(e1)THEN              'Block IF'. Defines blocks of statements and conditionally executes them. If  the logical
   block1                expression e1 is true block1 is executed and control  transfers to the first executable
 ELSEIF(e2)THEN          statement after ENDIF. If e1 is false but e2 true, then block2 is executed and control
   block2                transfers past ENDIF. If all logical expressions e1,e2... are false then block3 is executed.
 ELSE                    If there is no ELSE block, control transfers below ENDIF. The ELSEIF(...)THEN and ELSE
   block3                statements, with their associated blocks, are optional. To test any or all elements of an array
 ENDIF                   see the intrinsic functions ANY, ALL and COUNT; or use a WHERE construct.
INTRINSIC nam1,nam2...    Used in a CALLing (sub)program to define INTRINSIC FUNCTION names nam1,... (e.g. SIN, LOG...)
                          to be used as arguments in calls to subprograms. See also EXTERNAL.
 IMPLICIT NONE           Specifies that variable names do not determine REAL or INTEGER type of variable.
```

```
                                Precedes other type declarations.
 IMPLICIT typ(l1,l2-l3,..)      Specifies that variables starting with the letter l1, or a letter in the range l2 to l3,
                                are type typ. (E.g. IMPLICIT COMPLEX (C) declares all variables starting with C to be complex,
                                IMPLICIT DOUBLE PRECISION(A-H,O-Z), declares all variables except those starting with the
                                letters I,...,N to be double precision.) IMPLICIT statements must precede explicit type
                                declarations (REAL,..), but are overridden by such explicit declarations.
 'Implied DO-loop'      See near top of this listing.
 INTEGER vlist          'Explicit type declaration', that variables in vlist are type 4-byte INTEGER, range about 1E9.
                        (Fortran assumes variables beginning with letters I,J,K,L,M,N are INTEGER unless otherwise specified.)
#INTEGER(KIND=i) vlist  where i is typically 2,4,8 (approx ranges +-1E4,+-1E9,+-1E18). As above, but i-bytes INTEGER type.
#INTEGER(i)             Same as INTEGER(KIND=i). (i-values are not specified by the F90-standard)
#INTEGER*i              Same as INTEGER(i) (Non-standard, out-dated)
 LOGICAL vlist          'Explicit type declaration', that variables in vlist are type LOGICAL (values .TRUE. or .FALSE.)
 MODULE nam             Header statement for a MODULE program unit.
 OPEN(u,FILE=nam,STATUS=stat)   Opens file nam (a character expression, e.g. 'MYDATA.DAT') on logical unit u (integer 1-99)
                        stat may be 'OLD', 'NEW', 'REPLACE' and other (case-independent) values. (See manual, and see CLOSE)
 PARAMETER(v1=init1,v2=...)      Declare v1,v2,... to be unchangeable 'named constants' given values init1,... at compile time
 PRINT f, vlist         Same as WRITE(*,f) vlist
 PROGRAM nam            Specifies a name for the main program. Optional.
 READ f, vlist          Same as READ(*,f) vlist
 READ(u,f) vlist        Reads from unit u and assigns values to the elements in vlist. The records are converted according
 READ(u,f,ERR=s1,END=s2) vlist  to the format specification f, which may be a FORMAT statement label, a character constant
                                (e.g. '(3E8.2)'), a character variable or expression, or * representing 'list-directed format'
                                (i.e. data separated by blanks or commas). If f is omitted, the data on unit u are assumed
                                'unformatted', i.e. the output of an unformatted WRITE statement. If u=* then data are read from
                                the system default input file (stdin), usually the keyboard. The ERR and END are optional,
                                but transfer control to statement s1 if an error occurs, and to s2 if end-of-file is detected.
 READ(u,f,...,ADVANCE='NO')  READ but do not advance to next line. 'NO' is case independent. f=* is not permitted. (See WRITE)
 REAL vlist             'Explicit type declaration', that variables in vlist are type REAL, precision 6-7 digits, range 1E+-37.
#REAL(KIND(1E0)) vlist  (Fortran assumes variables beginning with letters A-H,O-Z are REAL unless otherwise specified.)
#REAL(4) vlist          Common equivalent to REAL vlist. (Non-standard - avoid)
#REAL(KIND(1D0)) vlist  Same as DOUBLE PRECISION vlist
#REAL(8) vlist          Common equivalent to DOUBLE PRECISION vlist. (Non-standard - avoid)
#REAL(16) vlist         Quadruple precision, approx precision 33 digits, range 1E+-4930. (Non-standard)
#REAL(KIND=i) vlist     where i is typically 4,8,16. i.e. i-bytes REAL type. (i-values are not specified by the F90-standard)
#REAL(i)                Same as REAL(KIND=i)
#REAL*i                 Same as REAL(i) (Non-standard, out-dated)
 RETURN                 Returns control to the invoking (sub)program from the current subprogram.
 REWIND(u)              Rewinds logical unit u to beginning of file.
 f(p1,p2...)=e          Creates a user-defined STATEMENT FUNCTION f having variables p1,p2... as dummy arguments. When
                        f(a1,a2...) is referred to, expression e is evaluated using the actual arguments a1,... in place of p1,...
 SAVE vlist             Save values of variables in vlist between successive calls to a subprogram.
 SAVE                   As above but save all local variables in the current subprogram.
 SELECT CASE(ic)        'SELECT CASE' construct. Defines blocks of statements and conditionally executes them depending on
```

```
 CASE(ic1:ic2, ...)      the values of the character or integer (not both) expression ic. If ic is in one of the comma
   block1                separated ranges ic1:ic2,...  block1 is executed and control transfers to the first executable
 CASE(ic3:ic4, ... )     statement after ENDSELECT. Similar for the other CASE blocks.
   block2
 CASE DEFAULT            If no CASE is matched then the default block, if present, is executed; otherwise
   block                 control transfers below END SELECT.
 END SELECT
 nam:  SELECT ...        As above but naming the SELECT construct nam.
 STOP 'message'          Terminates program execution and prints the optional message.
 prefix SUBROUTINE nam(p1,p2,...)       Header line of a subroutine subprogram, indicating the program name nam and dummy
                         arguments p1,p2,... Each of these is a variable or subprogram name. (See CALL, RETURN and EXTERNAL.)
                         The prefix if present may be a type (e.g. REAL etc.), or RECURSIVE
 USE nam                 Specifies that MODULE nam be used
#USE nam, ONLY :: vlist  As above, but only using variables in vlist from MODULE nam.
#USE nam :: v1=>v11, v2=>v2, ..  As above but referencing variables v1 etc from MODULE nam under names v11 etc.
 WHERE(e) st             Where the logical array expression e is true, the corresponding elements of the array section on the LHS
                         of st are assigned to the values on the RHS.
 WHERE(e1)               As above; defines blocks of statements and conditionally executes them. Where  the logical array
   block1                expression e1 has elements that are true, array statement block1 is executed and control  transfers
 ELSEWHERE(e2)           to the first executable statement after END WHERE. Where e1 has false elements, but e2 true, then
   block2                block2 is executed and control transfers past ENDWHERE. Where both e1 and e2 have false elements,
 ELSEWHERE               then block3 is executed. If there is no ELSEWHERE block, control transfers below ENDWHERE. The
   block3                ELSEWHERE(...) with their associated blocks, are optional.
 ENDWHERE                (The conditional form ELSEWHERE(e2), as opposed to plain ELSEWHERE is F95.)
 nam:  WHERE ...         As above but naming the WHERE construct nam.
 WRITE(u,f) vlist                Writes to unit u the values of the elements in vlist. The records are converted according to
 WRITE(u,f,ERR=s) vlist          format specification f, which may be a FORMAT statement label, a character constant
                                 (e.g. '(3E8.2,5X,I4)'), a character variable or expression, or * representing 'list-directed
                                 format'. If f is omitted, the data sent to unit u are 'unformatted' (i.e. binary).
                                 If u=* then the data is written to the system default output file (stdout), i.e. usually the
                                 screen. The optional ERR specifier tranfers control to statement s if an error occurs.
 WRITE(u,f,..,ADVANCE='NO')      As above, but do not advance to the next line after writing. f=* is not permitted in this case.
                                 'NO' is case independent. Useful in prompts like WRITE(u,'(A)',ADVANCE='no') 'Enter x:'

 ================================================================================================================================

          INITIALIZATION and TYPE DECLARATIONS
          ------------------------------------

 Initialization, i.e. assignment of values at compile time,  may be done 3 ways: (i) DATA statements, (ii) PARAMETER
 statements, (iii) type declarations. Both (i) and (ii) are listed in the previous section. (ii) is only used for unchangeable
 named constants like PI. The general form of (iii) is:-

 type,attribute1,attribute2, ... :: v1=init_val1, v2=init_val2, ...
```

The ,attributei and =init_vali are optional. The :: is optional but must be present if an attribute or initial value is present. Notation v1(d) indicates that the dimensions d over-ride the default dimensions if any in attributes. Examples:-

```
INTEGER K, X(4,2)
```
declares K to be an integer, and X a 4×2 integer array;

```
REAL, DIMENSION(3,2) :: A=2**3, B=RESHAPE([1,2,3,4,5,6],[3,2]), C(2,2)
```
declares A to be a real 3x2 array with all elements 8.; B 3x2 with col1=[1.,2.,3.], col2=[4.,5.,6.]; C 2x2 but not initialized.

```
CHARACTER(5), DIMENSION(2) :: C1, C2(3)*4    or equiv    CHARACTER(5) C1(2), C2(3)*4
```
declares C1 to be a CHARACTER array with 2 elements each of length 5 characters; and C2 to be a character array with 3 elements of length 4 characters.

type is one of INTEGER, REAL, DOUBLE PRECISION, COMPLEX, CHARACTER, LOGICAL, and others not in this summary. attribute is one of ALLOCATABLE, DIMENSION, PARAMETER, SAVE, EXTERNAL, INTRINSIC, and others not in this summary. Since initialization is done at compile time, the init_vali can only be constants, or contain a few simple functions (such as exponentiation with integer powers, RESHAPE, TRIM and others not in this summary). Any variable initialized in a type declaration, automatically has the SAVE attribute, but this is not so for DATA or PARAMETER initializations. Any attribute may be attributed in a command of its own -- e.g. see SAVE, PARAMETER, DIMENSION in the previous section.

==========

COMMON FORTRAN INRINSIC PROCEDURES (a small selection from over 100 intrinsic subprograms)

Procedures are FUNCTIONs unless otherwise specified. SUBROUTINEs are CALLed.

Type of arguments

arr	- array (rarr is real array etc)
c	- COMPLEX
ch	- CHARACTER
d	- DOUBLE PRECISION
i	- INTEGER
l	- LOGICAL
r	- REAL
rd	- REAL or DOUBLE PRECISION. Similarly for irdc etc.

Function	Result (Results are the same type(s) as the function argument(s), or as specified.)
ABS(irdc)	Absolute value of irdc (Value for COMPLEX argument a+ib is REAL equal to SQRT(a**2+b**2))
ACOS(rd)	Value y in range [0,PI] such that COS(y)=rd
ADJUSTR(ch)	Adjust right, moving any trailing blanks to the left end. (See also TRIM.)
ADJUSTL(ch)	Adjust left, moving any leading blanks to the right end. (See also TRIM.)
AIMAG(c)	Imaginary Part of c. (Type REAL)

```
 ALL(larr)                Value true if all elements of the logical array are true. (See also COUNT.)
 ANY(larr)                Value true if any element of the logical array is true. (See also COUNT.)
 ASIN(rd)                 Value y in range [-PI/2,PI/2] such that SIN(y)=rd
 ATAN(rd)                 Value y in range (-PI/2,PI/2) such that TAN(y)=rd
#ATAN2(rd1,rd2)           Argument of the complex number (rd2,rd1), in range (-PI,PI].
 CEILING                  Next higher INTEGER: e.g. CEILING(1.2)=2, CEILING(-1.2)=-1 (See FLOOR, INT, NINT)
 CMPLX(ird1,ird2)         Conversion to COMPLEX. (Result has REAL real and imaginary parts.)
 CONJG(c)                 Complex Conjugate of c.
 COS(rdc)                 Cosine of rdc
 COSH(rd)                 Hyperbolic Cosine of rd
 COUNT(larr)              Integer number of elements of logical array that are true. (See also ALL and ANY.)
#DATE_AND_TIME(chdate,chtime,chzone)    Returns date, time, zone.
 DBLE(irdc)               Conversion to DOUBLE PRECISION (for complex argument converts real part)
 DOT_PRODUCT(irdc1arr,irdc2arr)   Dot product of the 1D conforming arrays irdc1, irdc2
 EXP(rdc)                 Exponential of rdc, i.e. the mathematical e raised to the power rdc
#EPSILON(rd)              'Machine-epsilon': smallest eps such that (r) 1.+eps > 1. (d) 1d0+eps > 1d0 (See also HUGE, TINY)
 FLOOR                    Next lower INTEGER: e.g. FLOOR(1.2)=1, FLOOR(-1.2)=-2 (See CEILING, INT, NINT)
#HUGE(ird)                Overflow level. Largest number of type ird that may be stored. (See also TINY, EPSILON)
#INDEX(ch1,ch2)           Integer start position of the first occurrence of substring ch2 in ch1
 INT(rdc)                 Conversion to INTEGER (for complex argument converts real part): INT(1.7)=1, INT(-1.2)=-1.
 LEN(ch)                  Length of string ch (type INTEGER)
 LOG(rdc)                 Natural Logarithm (i.e. to base e) of rdc (of form a+ib where b is in [0,2*PI))
 LOG10(rd)                Common Logarithm (i.e. to base 10) of rd
 MATMUL(irdc1,irdc2)      Matrix multiplication of irdc1 by irdc2 (on right)
 MAX(ird1,ird2,...,irdn) Maximum of values ird1,...,irdn
#MAXLOC(irdcarr)             Location of maximum element in array irdc
 MAXVAL(irdcarr)             Maximum element in array irdc
 MIN(ird1,ird2,...,irdn) Minimum of values ird1,...,irdn
#MINLOC(irdcarr)             Location of minimum element in array irdc
 MINVAL(irdcarr)             Minimum element of array irdc
 MOD(ird1,ird2)           Remainder ird1-ird2*INT(ird1/ird2)
 NINT(rd)                 Nearest INTEGER: e.g. NINT(1.2)=1, NINT(1.7)=2, NINT(-1.7)=-2. (See FLOOR, INT, CEILING)
 PRODUCT(irdcarr)         Product of the elements of array irdc
 RANDOM_NUMBER(rarr)      SUBROUTINE. Returns real scalar or array of random numbers uniformly distributed over [0,1].
 RANDOM_SEED(PUT=iarr)    SUBROUTINE. Uses integer array to seed RANDOM_NUMBER.
 REAL(irdc)               Conversion to REAL (for complex argument gives real part)
 RESHAPE(arr1,arr2)       Reshape the 1D array arr1 to have dimensions specified by the elements of the 1D array arr2.
 SIGN(ird1,ird2)          Transfer of Sign, i.e. ABS(ird1)*SIGN(ird2)
 SIN(rdc)                 Sine of rdc
 SINH(rd)                 Hyperbolic Sine of rd
 SQRT(rdc)                Square Root of rdc (of form a+ib where a> or = 0, and b> or =0 if a=0)
 SUM(irdcarr)             Sum of the elements of array irdc
#SYSTEM_CLOCK(i1,i2,i3)   SUBROUTINE. Returns value, rate/sec, max-value of clock counts, in i1,i2,i3, all of which are optional.
 TAN(rd)                  Tangent of rd
```

```
 TANH(rd)                Hyperbolic Tangent of rd
#TINY(ird)               Underflow level. Smallest number of type ird that may be stored. (See also HUGE, EPSILON)
 TRANSPOSE(irdcarr)         Transpose of the array irdc
 TRIM(ch)                Remove trailing blanks from ch. (See ADJUSTL, ADJUSTR)
```

```
                FORMAT Separators
                -----------------
   ,     Comma is standard separator.
   /     Slash is end-of-record. Causes jump to a new record (line). Similarly // or 2/ jumps 2 lines, etc.

                FORMAT Descriptors
                ------------------
 'any'    Transfer the characters between quotes to output unit (e.g. screen or file) or from input unit (e.g. keyboard or file).
 Aw       Transfer character data in a field width of w columns.
#Bw,Bw.d  Same as Iw,Iw.d but for integer representations in the binary number system.
#:        Stop here if no further items in the output variable list. Used to avoid headings for non-present variables.
#Dw.d     Sames as Ew.d, except D printed instead of E in outputs exponent. Out-dated, used for double precision numbers.
 Ew.d     Transfer REAL or DOUBLE PRECISION variables with a mantissa rounded to d digits after the decimal point, and an E
          exponen indicator, in a field width of w columns.
#ENw.d    (Engineering) Same as Ew.d except that exponent is made a multiple of 3 by adjusting the fraction.
#ESw.d    (Scientific) Same as Ew.d except that exponent is adjusted to give a mantissa between 1 and 10 in magnitude.
 Fw.d     Transfer REAL or DOUBLE PRECISION variables rounded to d digits after decimal point, in a field width of w columns.
          If w=0, minimum width is used to display the number without any blanks.
 Gw.d     May be used for any data type. For real or complex it produces Ew.d or F(w-4).d depending on size of number.
#Hw,Hw.d  Same as Iw,Iw.d but for integer representations in the hexadecimal number system.
 Iw       Transfer integer values in field width w . If w=0, minimum width is used to display the number without blanks.
#Iw.d     Transfer integer values in field width w . Prints at least d digits by padding left with zeros; useful in dates.
 Lw       Transfer logical data in field width w. (Input may be T, .T, F or .F followed by arbitrary characters. Ouput is
          just T or F right justified.)
#Ow,Ow.d  Same as Iw,Iw.d but for integer representations in the octal number system.
#SP       (Sign Print) Print + signs of numbers for rest of format or until countered by SS.
#SS       (Sign Suppress) Omit + signs of numbers for rest of format or until countered by SP.
 Tn       Absolute tabulation. I.e. read or write starting at column n.
#TRn      Relative tabulation. Start n places to the right of the current column.
#TLn      Relative tabulation. Start n places to the left of the current column.
 nX       Print n blanks or skips n characters. Same as TRn.

                Repeat Count
                ------------
 Specified by integers preceding FORMAT descriptors or group of descriptors in brackets.
 E.g. 2F10.2 for 2 REAL or 1 COMPLEX variable, or 3(2/,7X,4F10.3).
```

Format Reversion (i.e. IOLIST exceeds FORMAT Length)

Convention: FORMAT restarts (with a new record or new line) at the open-bracket matching the second last closed-bracket, including repeat count if present. If no such bracket, then FORMAT restarts at the beginning. Also, a new record (line) is started for READ or WRITE each time the right-most bracket is reached.