Fortran90/95 Course Notes

J.S.B. Gajjar
j.gajjar@manchester.ac.uk

Course Objectives

- To become familiar with basic concepts of F90/F95.

- Use the knowledge gained to write (correct) F90/F95 codes.

- Be able to compile and run these codes and produce meaningful output.

Resources

A good place to look for resources on Fortran 90, Fortran 95 is

http://www.fortran.com/fortran

There are many good books available.

Students at Manchester University can download the NAG F95 compiler (and NAG libraries) for non-commercial use from

http://www.nag.co.uk

and obtain the licence from
www.itservices.manchester.ac.uk/applications/licensing/codes/nag/
for both Windows and Linux.

Students may still be able to purchase the Salford F95 compiler for Windows from the

Manchester Computing Shop for a nominal amount, about £15.

For those into Linux, an excellent (free) Fortran 90 compiler is available from Intel -search the Intel web site

http://www.intel.com

Programming

What is it? - A set of instructions to enable a computer to perform a given task.

E.g. adding two numbers

$$a + b$$

Programming can take various forms:

- Low-Level- assembly language coding.
- High-Level - Instructions coded in a special language using a set of well defined rules and grammar.

  Examples of high-level languages include, F77, F90, C, C++. Also symbolic languages such as MAPLE, MATHEMATICA, MATLAB, EXCEL, ...

Our aim in this course is to learn F90 as a tool for solving certain problems- for example, solution of ODE and PDE's, eigenvalue problems, linear systems,...

<u>Various Stages</u>

- Problem formulation.
- Analysis and breakdown into smaller tasks.
- Coding.
- Testing.

Coding

Coding involves translating a task/algorithm into a language that the computer can understand.

There are 3 main steps to go through for Fortran 90 (or C, C++) codes.

- Write a program. Involves special syntax and grammar.
- Compile the program. This translates the high level statements into machine code.
- Running the code.

Steps needed for coding in Fortran 90

- Using your favourite editor, create a file containing the statements to be executed.
- The file should have a suffix .f90 for example, program.f90
- To compile code do
  f95 program.f90
- Last step creates an executable file called a.out (by default).
- To run the code, do
  ./a.out
- and await answers.

To create an executable called something different, say b.out , do

```
f95 -o b.out program.f90
```

Program.f90

What does this file contain?

The basic structure of this as follows:

Program specification statements

...

Declaration statements (where type of variables are declared)

...

Executable statements (heart of coding of algorithm/task)

...

end

Example

Here is an example Fortran90 program .

```
PROGRAM  Test
IMPLICIT NONE
REAL :: a,b,c,d,e
INTEGER :: i,j,k

a=10.8
b=9.2
i=2
j=3
k=2

c= a*b ;d=(i*a)/j + k
e = (i/j)*a &
  + k
i= a*j

PRINT *,'a,b,c' , a,b,c
PRINT *, 'd,e' , d,e
PRINT *, 'i,j,   &
 &  k', i,j,k

END PROGRAM test
```

Grammar and Syntax

- Lines upto 132 characters.
- Lower and upper case letters (not case sensitive).
- Variable names upto 31 characters including underscore.
- Semicolon to separate multiple statements on one line.
- Ampersand (&) used for continuation lines.
- Many keywords. Will adopt convention in this course that KEYWORDS will be in CAPITALS.

My first program

```
PROGRAM test
PRINT *, 'my name is jitesh'
END PROGRAM test
```

Here the KEYWORDS are PROGRAM, PRINT, END.

The PRINT *, will output the text in quotes to the default output device which is the screen.

The name of this program unit is test. Can call it anything you want.

Instead of PROGRAM, could have other keywords such as FUNCTION, SUBROUTINE.

Then the name assigned to this program unit is important.

Variables

To be able to do any useful computations we need to introduce the idea of variables.

Variables can be of the following types:

INTEGER, REAL, COMPLEX,

LOGICAL, CHARACTER

We can also define our own data types, see later.

Consider integers and reals first.

INTEGER variables are stored *exactly* in the computer.

REAL variables stored in binary form with a part for exponent and a part for mantissa.

REAL variables involve loss of precision- can only store a certain no of significant digits in mantissa.

To use variables one has to declare them.

Declaration statements take the form:

```
TYPE ::   name1, name2, ....
```

where TYPE can be
INTEGER, REAL, LOGICAL, CHARACTER, COMPLEX

IMPLICIT TYPES

In FORTRAN 90 if you do not declare variables, there is an implicit convention which is used to determine the type of variable.

- Variables names beginning with A-H, O-Z are REAL variables.

- Variables names beginning with I-N are INTEGER variables.

TO AVOID implicit typing add the line

IMPLICIT NONE

as the second line in your program (and all subprograms). This will help avoid many common bugs which you will generate in writing codes.

Examples

```
REAL :: a,b,c ! declares a,b,c to be real

INTEGER :: i,j,k ! declares i,j,k integer
```

Arithmetic Operations

$+$, $-$, $*$, / , $**$

Precedence: $**$, $*/$, $+-$

in order from left to right.

Use brackets () to signify which is to be done first.

Assignments

These take the form

a = expr

where a is a variable and expr some arithmetic expression.

After the statement is executed the values calcuated via expr are stored in the variable a.

Example

a= 10. ! assign value 10. to variable a

a= b+ c ! takes b and c and stores their sum in a

a = SIN(x) ! computes sin(x) and stores in a

c= 2.*a ! computes c=2a

d = a**6 ! computes a^6

e = (d/6)**2 ! compute $(d/6)^2$

a= 2.*a !

Mixed mode expressions (containing integers, reals) are possible, but care needs to be exercised.

Example

```
PROGRAM  Test
IMPLICIT NONE
REAL :: a,b,c,d,e
INTEGER :: i,j,k

a=10.8
b=9.2
i=2
j=3
k=2

c= a*b
d=(i*a)/j + k
e = (i/j)*a + k
i= a*j

PRINT *,'a,b,c' , a,b,c
PRINT *, 'd,e' , d,e
PRINT *, 'i,j,k', i,j,k

END PROGRAM test
```

What will the output be?

Mathematically the values for d and e should be the same.

But different values are obtained.

Why because in expression for e

(i/j) is computed first. This involves integers and thus the answer is chopped to an integer which is zero here.

If necessary use FLOAT(i) to convert INTEGER variable i to a REAL variable.

CHARACTER variables

Variables of type character declarations and assignments take the form for example

```
CHARACTER (LEN=2) :: string  ! string of  length 2

CHARACTER (LEN=5) :: st1, st2  ! strings of length 5
CHARACTER :: st3           ! of length 1

string = "abc"
st1 = "defg"
st2 = string // st1
st3= st1
```

Here the // symbol stands for concatenate the two strings

The LEN=5 means a variable of length 5 characters.

Substrings

```
"abcde"(1:3)         !     picks up "abc"



CHARACTER (LEN=6) :: alpha,beta
alpha = "myname"
beta = alpha(2:3)   !   is "yn"
beta = alpha(:3)    !    is "myn"
beta = alpha(3:)    !  is "name"

alpha(4:) = "this"   ! alpha now is "mynthi"
```

In the same way that

```
PRINT *, ....
```

works, we can use

```
READ *, a,b,c, ...
```

to input values a,b,c from the keyboard.

```
REAL :: a,b
CHARACTER (LEN=20) :: string

READ *, a,b,string
```

will expect two real values and a character string to be input.

Functions, Subroutines, Modules

Key to successful programming is to break it up into smaller independent tasks.

Functions and subroutines can be developed and tested independently.

Fortran90 has a number of mathematical and other INTRINSIC functions.

Examples

```
SIN(x) , LOG(x), FLOAT(x), MATMUL(a,b),...
```

These can be used anywhere without having needing to be declared

E.g. if we want to code up the expression

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

we would use

```
x1 = (-b + SQRT(b*b - 4.*a*c))/(2.*a)
x2 = (-b - SQRT(b*b - 4.*a*c))/(2.*a)
```

We can also write our own functions. These take the form

```
FUNCTION name(arg1,...)

specification statements

executable statements

END FUNCTION name
```

Thus for example suppose we wanted to write a function for calculating

$$\text{sinh}(x) = \frac{e^x - e^{-x}}{2}$$

We can use

```
FUNCTION hsinh(x)
IMPLICIT NONE
REAL, INTENT(IN) :: x
REAL :: hsinh

hsinh = (exp(x) - exp(-x))*0.5

END FUNCTION hsinh
```

Note the INTENT(IN) is an attribute. It tells the compiler that the variable x is not to be changed in the function.

The function would be used in the following way:

```
REAL, EXTERNAL :: hsinh
REAL :: y,x

y= hsinh(x)
```

SUBROUTINES are very similar except in the way program control is exercised.

```
SUBROUTINE name (arg1,....)

specification statements

executable statements

END SUBROUTINE name
```

This would be called in the calling routine via

```
CALL name(arg1,...)
```

Example. The following is a subroutine to swap two numbers.

```
SUBROUTINE swap(a,b)
IMPLICIT NONE
REAL, INTENT(INOUT) :: a,b
REAL :: c

c=a
a=b
b=c

END SUBROUTINE swap

PROGRAM main
IMPLICT NONE
REAL :: a=1,b=0., c=2
CALL swap(a,b)
PRINT *, a,b,c
END PROGRAM main
```

Notice that the variable c is local to the subroutine

ALso the INTENT(INOUT) attribute signifies that those variables can change values.

Similarly we have the INTENT(OUT) attribute.

The argument need not be defined on entry but should be assigned on exit.

Logicals Flow Control

F90 permits variables of LOGICAL type.

```
LOGICAL :: a,b

a= .TRUE.
b= .FALSE.
```

Relational operators give rise to logical outcomes.

```
a < b        a.LT.b  !  true if a < b
a <=b        a.LE.b  !  true if a less
                         than or equal to b

a > b        a.GT.b  !  true if  a > b

a >= b       a.GE.b  !  true if a is greater
                          than or equal to b

a == b       a.EQ.b  !  true if a equals b

a /= b       a.NE.b  !  true if a is not
                        equal to b
```

Thus for example to check whether the discriminant "$b^2 - 4ac$" is positive or not.

```
LOGICAL :: log

log = (b*b - 4.*a*c > 0. )
```

Usually one combines logical variables in decision making and flow control.

One such combation makes use of the IF ... THEN statement.

```
IF (....) THEN

action
ENDIF
```

The above is the simplest. If (...) is TRUE the statements are executed. Otherewise the statement following the ENDIF is executed.

A more complex version is

```
IF (criterion 1) THEN

action 1

ELSEIF (criterion2) THEN

action 2
...

ELSE

other action

ENDIF
```

Example

```
PROGRAM quad
IMPLICIT NONE
REAL :: a,b,c,d,x1,x2
PRINT *, 'enter coefficents of quadratic'
READ *, a,b,c

d = b*b - 4.*a*c
IF( d >= 0) THEN
 x1 = (-b + SQRT(d))/(2.*a)
 x2 = (-b - SQRT(d))/(2.*a)
 PRINT *, 'real roots',x1,x2
 ELSE
 PRINT *, 'complex roots'
ENDIF

END PROGRAM quad
```

Repetition and Looping

Most of the simple examples that we have encountered have not really required a computer to do. Tasks which require repetition are where the use of a computer becomes necessary.

Fortran90 allows repetition through the DO ... ENDDO statement. These take the form

```
DO count = initial,final,inc

block of statements

END DO
```

Here *count* is an integer, and the loop is executed starting at *initial*, with increments *inc* and finishing when *count* the *final* value is reached.

Variations of the above are

```
DO count = initial,final !inc is taken to be one
DO          !  execute loop once.
```

Examples

```
DO i=1,10         ! 10    i=1,2,3,4,5,6,7,8,9,10
DO j=20,50,5      ! 7    j=20,25,30,35,40,45,50
DO x=-20,20,6     ! 7     x=-20,-14,-8,-2,4,10,16
DO k=6,4,3        ! 0 times
```

NOTE: It is NOT permitted to alter the value of the DO variable inside the DO loop.

It is possible to give names to DO loops.

```
myloop:   DO i=1,10

      .

      .
  ENDDO myloop
```

The EXIT statement allows transfer out of a loop.

```
DO  count = initial, final, inc
   .
   .
 IF(condition)EXIT



   .
   .
ENDDO
```

If *condition* is satisfied then execution continues from the statements after the ENDDO statement.

Example

```
DO i=1,maxits
call sub(residual,a)
If(residual.lt.1.e-6)THEN
PRINT *, 'converged',i,residual
EXIT
ENDIF
ENDDO
```

There are other situations when we need more radical control.

The STOP statement immediately stops the program.

The RETURN statement in a subroutine exits the subroutine

Another widely used statement in older versions of FORTRAN is the GOTO label and CONTINUE.

This transfers control to the statement identified by *label*.

```
DO i=1,maxits
call sub(residual,a)
If(residual.lt.1.e-6)THEN
PRINT *, 'converged',i,residual
GOTO 100
ENDIF
ENDDO
PRINT *,'not converged after ',maxits, 'iterations'
STOP
100    CONTINUE
```

Statements if labelled have at most a 5 digit number in front.

The label must be unique to the subprogram or program.

```
1000   READ *, a,b,c
1010   STOP
```

Arrays

The array processing features of Fortran90 are extremely powerful. Arrays are used extensively in scientific computation.

Arrays are very much like matrices. For example a one-dimensional array would have elements

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

There are stored contiguously in memory.

Array Declarations

Arrays can be declared as seen in the following examples

```
REAL, DIMENSION(5) :: a,b,c
! declares a,b,c to be arrays of 5 elements

REAL :: a(50),b(0:20),c(-2:5)
! a has 50 elements
! b has  21 elements starting with b(0)
!  c has 8 elements starting at c(-2)
```

The above are all examples of 1D arrays.

Multidimensional arrays are similarly defined.

```
INTEGER, DIMENSION(3,4) ::  a
! defines an integer array of size 3x4
CHARACTER(LEN=4), DIMENSION(-1:4,2:6,8) :: string
! defines a 3 dimensional character array
```

More I/O and files

So far we have met the READ and PRINT statements in primitive form. We will now look at more advanced formatting, the WRITE statement and using files.

So far we have met

```
READ *, a,b,c
PRINT *, a,b,c
```

This is called list-directed I/O. There is no control over how the data is input or output.

A more useful form of the above is to use

```
READ '(edit description)', input_list
```

or

```
READ "(edit description)",input_list
```

The *edit description* describes how the data is to be handled.

The following edit descriptors are available for READ.

```
Iw    Read the next w characters as an integer

Fw.d  Read the next w characters as a real, d digits
Ew.d      after the decimal point.

Aw    Read the next w characters.

A     Read sufficient characters to fill the input list
      as characters.

Lw    Read the next w characters as the representation
      of a logical value.

nX    Ignore the next n characters

Tc    Next character to be read is at position c

TLn   Next character to be read is n characters
TRn         before (TL)       or after (TR)
               the current position.
```

Examples

```
INTEGER :: n,m,p
READ '(I4,I3,I2)', n,m,p

! if input was 1234567890
! n=1234  m=567 p=89

READ '(2X,I4)',n

! would make n=3456


! suppose data is 987654321
REAL :: r1,r2,r3,r4
READ '(F3.1,F2.2,F3.0,TL6,F4.2)',r1,r2,r3,r4
! r1= 98.7  r2 = 0.65  r3=432.0 r4=76.54

! suppose data is .32.56.9
! then r1=0.32    r2=0.5  r3=6.9 r4=2.56
```

The Ew.d descriptor is intrepreted in a similar way.

The Aw is similar however, if character varaible has a defined length *len* then

If w is less than *len*, blank characters are added to make length *len*

If w is greater than *len* the rightmost *len* characters of the input list are used!

A without the field width w is treated as though the length is the same as the corresponding variable.

The Lw expects data in the form

```
.Tcccc...c        or Tccccc..c
.Fccccc..c        or Fccccc..c
where c is any character.
```

We can also use labels with the FORMAT statement to read the edit description list

```
READ 100, r1,r2
100 FORMAT(1X,E12.5,E12.5)
```

Output formatting of variables is very similar.

```
PRINT '(edit description)', output list

Iw    Output the next w characters as an integer

Fw.d  Output a real number w positions , d digits
      after the decimal point.

Ew.d  Output a real number in the next w positions in
      exponent form, with  d digits decimal places in
      mantissa and four characters for exponent.

Aw    Output a character string in  the next w positions.

A     Output a character string at next position

Lw    Output  w-1 blanks with T or F in next position
      logical value.

nX    Ignore the next n character positions.

Tc     Output next item starting at position c.

TLn   Next item to be output  n characters before (TL)
TRn      or after (TR) the current position.

"cdedfff..'   Output string in quotes,
'cdefff...'   Output string in quotes.

/      start on a new line
```

We can also use a repeat count before the descriptor.

```
PRINT 100, a,b,c
100 FORMAT(1x,3(E12.5,1x))

! is equivalent to
100 FORMAT(1x,E12.5,1x,E12.5,1x,E12.5,1x)
```

So far the read and writing have used the default units, ie the keyboard or screen for input/output. We can also input from files and output to files.

To do this we need to use the UNIT specifier. The UNIT=* is equivalent to what we have done so far.

Usually (this is site-dependent) UNIT=5 refers to an input unit and UNIT=6 to an output unit.

The following examples should suffice

```
READ(UNIT=5,FMT='(3(I5,1x)'))i1,i2,i3
READ(FMT='(3(I5,1x)'))i1,i2,i3
READ(5,FMT='(3(I5,1x)'))i1,i2,i2
READ(5,100)i1,i2,i3
READ(5,'(3(I5,1x)'))i1,i2,i3
100 FORMAT(3(I5,1x))
```

are all equivalent

For output we do exactly the same but now use WRITE instead of PRINT.

```
WRITE(6,100)r1,r3,r3
100 FORMAT(1x,e12.5,1x,F12.5,1x,E12.5)
WRITE(6,120)(A(i),i=1,12)
120 FORMAT(4(E12.5,1x))
```

Note the last WRITE outputs 12 elements of an array in E12.5 format with 4 numbers per row.

COMPLEX VARIABLES

Another type of variable is the COMPLEX data type

```
COMPLEX :: a,b,d,e
REAL :: c

a = (1.0,1.0)  ! defines a= 1+i

b= (0.,2.)     ! defines b=2i

d= a*b +c      ! computes the complex number ab+ c

b  = a/d       ! complex division

e = EXP(a)     ! complex exponential
```

Array Processing

```
REAL , DIMENSION :: a(-3:4,7), b(8,2:8),e(10)
INTEGER ::c
REAL, DIMENSION(8,1:8) ::d
```

- Rank = Number of dimensions.
  - `a` and `b` have rank 2
  - `e` has rank 1

Array Processing

```
REAL , DIMENSION :: a(-3:4,7), b(8,2:8),e(10)
INTEGER ::c
REAL, DIMENSION(8,1:8) ::d
```

- Extent = Number of elements in a dimension.
  - `a` has extents 8 and 7
  - `b` has extents 8 and 7
  - `d` has extents 8 and 9
  - `e` has extents 10

Array Processing

```
REAL , DIMENSION :: a(-3:4,7), b(8,2:8),e(10)
INTEGER ::c
REAL, DIMENSION(8,1:8) ::d
```

- Shape = vector of extents
- Size = product of extents
  - `a` and `b` have shape /(8,7)/ and size 56
  - `d` has shape /(8,9)/ and size 72
  - `e` has shape /(10)/ and size 10

Array Processing

```
REAL , DIMENSION :: a(-3:4,7), b(8,2:8),e(10)
INTEGER ::c
REAL, DIMENSION(8,1:8) ::d
```

- Conformance = same shape
  - `a` is conformable with `b` and `c` but not `d` and `e`.
- For whole array operations arrays must be conformable.

Example, whole array operations

```
REAL, DIMENSION (20) :: a,b,c
REAL :: e(5,5), d(5,5), f(0:4,5)
...

 a=0.0

a=a/3.1 + b*SQRT(c)
 ! a(i)=a(i)/3.1 + b(i)*sqrt(c(i)) i=1:20

e=2*d
 ! e(i,j)= 2.*d(i,j)  i=1:5, j=1:5



f= e*d
 ! f(i-1,j)=e(i,j)*d(i,j)  i=1:5,j=1,5
```

Here scalars 'broadcast'. Also `SQRT` operates element by element on the array `c`

Dynamics Arrays

FORTRAN 90 allows dynamic arrays, ie size does not have to be specified until runtime.

How does this work? Use ALLOCATE and DEALLOCATE.

```
REAL, DIMENSION (:,:), ALLOCATABLE :: a,b
INTEGER :: n,m,error

...
READ(5,*) n,m
ALLOCATE(a(n,m),b(2*n,2*n), STAT= error)
IF(error >0)STOP 'Failed to allocate memory'

IF(ALLOCATED(a))DEALLOCATE(a)
```

Automatic arrays

FORTRAN 90 allows one to write subprograms with the bounds of the array passed via dummy arguments

```
SUBROUTINE(a,b,n)
IMPLICIT NONE
INTEGER :: n
REAL, DIMENSION(n,n) :: a
REAL :: b(0:n)
```

INTERFACE blocks

One can write functions which return array values. This requires the use of INTERFACE blocks.

INTERFACE blocks are just copies of the declaration parts of the function put into the calling program unit.

In the function one can make use of assumed shape arrays.

INTERFACE blocks are also needed when using assumed shape arrays in subroutines.

INTERFACE block should be put with the declaration statements and before the first executable statement.

```
  INTERFACE
   FUNCTION l2norm(a)
  IMPLICIT NONE
  REAL, DIMENSION(:,:), INTENT(IN) ::a
  REAL :: work,l2norm
  INTEGER ::n,j,m
  END FUNCTION  l2norm
  END INTERFACE

REAL a(20,20),result
...
result=l2norm(a)
....
END PROGRAM main
 FUNCTION l2norm(a)
  IMPLICIT NONE
  REAL, DIMENSION(:,:), INTENT(IN) ::a
  REAL :: work,l2norm
  INTEGER ::n,j,m,i
! first find size of array
...
  END FUNCTION l2norm
```

Example of using INTERFACE BLOCKS, ALLOCATABLE statements.

```
  PROGRAM example17
  IMPLICIT NONE



  REAL , ALLOCATABLE, DIMENSION(:,:):: a
  REAL:: result
  INTEGER :: error
  INTEGER:: n,m

 INTERFACE
   FUNCTION l2norm(a)
     IMPLICIT NONE
     REAL, DIMENSION(:,:), INTENT(IN) ::a
     REAL :: work,l2norm
     INTEGER ::n,j,m
    END FUNCTION  l2norm
   END INTERFACE

  PRINT *, 'ENTER SIZE OF matrix  A'
  READ *, n,m
  ALLOCATE(a(n,m),STAT=error)
  IF( error /= 0) THEN
   PRINT *, ' FAILURE IN ALLOCATING SPACE FOR V'
   STOP
  ENDIF


!  initialize v with random values
   CALL RANDOM_NUMBER(a)

!  now calculate l2norm
   result=l2norm(a)
   PRINT *, 'L2 NORM =',result

  END PROGRAM example17

  FUNCTION l2norm(a)
  IMPLICIT NONE
  REAL, DIMENSION(:,:), INTENT(IN) ::a
  REAL :: work,l2norm
  INTEGER ::n,j,m,i
! first find size of array
```

```
  n=SIZE(a,1)
  m=SIZE(a,2)

! now we can find the l2 norm
   work=0.
   DO j=1,m
    DO i=1,n
    work=work+ a(i,j)*a(i,j)
   ENDDO
  ENDDO
   l2norm=SQRT(work)
  END FUNCTION l2norm
```

Example code showing simple use of modules.

```
!--------------------------------------------------------------------
!
!  Program name:
!
!    choose_prec
!
!  Purpose:
!
!    This program uses the precision module and shows how to select
!    single or double precision by changing a single value.
!
!--------------------------------------------------------------------

PROGRAM choose_prec

  ! Select desired precision from module.
 USE precision, ONLY : wp => dp        ! dp for double, sp for single

  IMPLICIT NONE
  REAL (KIND = wp) :: a

  WRITE(*, *) HUGE(a)

END PROGRAM choose_prec
!--------------------------------------------------------------------
!
!  Module name:
!
!    precision
!
!  Purpose:
!
!    This module defines kind values for single and double
!    precision real variables.
!
!--------------------------------------------------------------------
```

```
MODULE precision

  ! Set up values for single or double precision

INTEGER, PARAMETER :: sp = KIND(1.0E0)   ! Returns kind value for single
INTEGER, PARAMETER :: dp = KIND(1.0D0)   ! Returns kind value for double

END MODULE precision
```