Basic Fortran Programming

National Computational Infrastructure

Outline

Fortran Programming Language

Fortran (FORmula TRANslation) is a high level language aimed at numerical calculations. Evolved over 30-40 years. All compilers should conform to the Fortran standard so that code is completely portable. Recommended textbooks:

- Fortran 90 Explained, Metcalf and Reid, Oxford Science Publications
- Fortran 90/95 Explained, Metcalf and Reid, Oxford University Press
- Fortran 95/2003 Explained, Metcalf, Reid and Cohen, Oxford University Press

Internet resources are

- http://www.fortran.com contains a FAQ and lots of other information.
- news://comp.lang.fortran to speak directly to those who develop Fortran

Free compilers

There are a number of free compilers available

- Fortran 77 - F77 part of the GNU GCC project
- Fortran 95 - g95 based on the GNU GCC project
- Fortran 95 - gfortran part of the GNU GCC 4.0 project
- Fortran 95 - ifort Intel Fortran compilers for linux

Fortran Program Structure

Here is an example Fortran code.

```
program circle
real  :: r, area
!This program reads a real number r and prints
!the area of a circle with radius r
read (*,*) r
area = 3.14159*r*r
write (*,*)' Area = ',area
stop
end program circle
```

Note that all variables are declared at the beginning of the program and before they are used.

Fortran77 Formalities

- Fortran77 requires that all lines except comment or continuation lines start in the 7th column (convention left over from Fortran coding sheets) and are restricted to 72 columns as follows:
  - Col 1 : Blank or c, C or * for comments
  - Col 1-5 : Statement label (optional)
  - Col 6 : Continuation of previous line (optional)
  - Col 7-72 : Statements
- Fortran is not case specific.
- Comments can appear anywhere in a program - use them.
- Any symbol can be used in the 6th column for continuation but the general practice is to use +, & or numbers.
- Implicit typing depending on first letter of variable name, i-n are integers. This easily leads to mistakes so don't rely on it.

Fixed or Free Format

The standard Fortran77 layout (i.c. columns 7-72) is called fixed format but with compiler options can be extended to a line width of 132.

Some compilers accept the non-standard tab source form where tab automatically skips to the 7th column. Then line continuations can be a number in the 8th column.

Free source form is generally used for Fortran90. Continuation lines are marked by an ampersand & at the end of the line that is to be continued.

Fortran77 or Fortran90

Fortran90 compilers are backward compatible so will interpret Fortran77 code. In fact, Fortan77 is a subset of Fortran90.

If you are modifying existing Fortran77 code you will have to continue to use the fixed format layout in any program that is already in fixed format.

Compilers decide whether code is fixed or free format depending on the extension of the file name e.g. prog.f or prog.f90

From now on we will present Fortran90 but mention any Fortran77 constructs that are now deprecated and replaced by Fortran90 constructs.

Outline

Declaration of Variables

Start every program with the statement

```
implicit none
```

This tells the compiler that all variables are to be declared and the compiler will report an error if any variable is not declared. This is good programming practice. Do not rely on the compiler to decide if variables are integers or real. This is different from e.g. Matlab

Declaration of Variables (Contd)

Variables can be of the following types:

- LOGICAL
- CHARACTER
- INTEGER
- REAL
- COMPLEX
- user constructed called derived types

Declaration of Variables (Contd)

For example the following all declare x as a single precision real variable, that is, its representation in memory is 4 bytes long

- real :: x
- real(4) :: x
- real*4 :: x

A double precision variable z which takes 8 bytes of memory can be represented in any of the following ways:

- real(8) :: z
- real*8 :: z
- double precision :: z

Declaration of Variables (Contd)

- `COMPLEX` (`COMPLEX*8` or `COMPLEX(4)`) - single precision complex number
- `COMPLEX(8)` or `COMPLEX*16` - double precision complex number
- `CHARACTER(LEN=n)`, `CHARACTER(n)`, or `CHARACTER*n` where n is an integer value represents the number of bytes in the character variable or can be *.
- `LOGICAL` can be `.TRUE.` or `.FALSE.`

Declaring arrays

Here are some examples of array declarations, there are several ways to do this.

- a is a real one-dimensional array of length 4, indexed 1 to 4.

```
real :: a(4)
```

- a is a real one-dimensional array of length 20

```
integer, parameter :: n=20
real   :: a(n)
```

- b is an integer one-dimensional array of length 20, indexed 0 to 19

```
integer :: b(0:19)
```

- c is double precision 2-dimensional array

```
double precision, dimension(10,10) :: c
```

In these the array a has elements `a(1)` to `a(20)`, b has elements `b(0)` to `b(19)` and c has elements `c(1,1)` to `c(10,10)`

Example 1

Compile and execute the following code `variables.f90` by doing

```
>f90 variables.f90 -o variables
>./variables
```

```
program variables

implicit none

integer    :: i
integer(2) :: j
integer(4) :: k
integer(8) :: l

real    :: a
real(4) :: b
real(8) :: c

write (*,*) ' Huge:   ',huge(i),huge(j),huge(k),huge(l)
write (*,*) ' Digits: ',digits(i),digits(j),digits(k),digits(l)

write (*,*) ''

write (*,*) 'Huge:   ',huge(a),huge(b),huge(c)
write (*,'(a,i,i,i)') 'Digits: ',digits(a),digits(b),digits(c)
write (*,'(a8,e,e,e)') 'Epsilon: ',epsilon(a),epsilon(b),epsilon(c)
write (*,'(a9,en,en,en)') 'Epsilon: ',epsilon(a),epsilon(b),epsilon(c)
write (*,'(a10,3es)') 'Epsilon: ',epsilon(a),epsilon(b),epsilon(c)
write (*,'(a11,3e20.10e4)') 'Epsilon: ',epsilon(a),epsilon(b),epsilon(c)

end program variables
```

Example 1 continued

The output from variables.f90 is

```
 Huge:     2147483647  32767  2147483647   9223372036854775807
 Digits:           31          15          31          63

 Huge:     3.4028235E+38  3.4028235E+38  1.797693134862316E+308
Digits:           24          24          53
Epsilon:  0.1192093E-06  0.1192093E-06   0.2220446049250313E-15
Epsilon: 119.2092896E-09119.2092896E-09 222.0446049250313081E-18
 Epsilon:   1.1920929E-07  1.1920929E-07   2.2204460492503131E-16
  Epsilon:   0.1192092896E-0006  0.1192092896E-0006  0.2220446049E-0015
```

Note how the output is presented and how this relates to the `write` statements.

- * is a special character meaning standard or default
- `a` represents ASCII characters
- `i` represents integers
- `e` represents exponential notation
- `es` represents scientific notation (similar to `e`)
- `en` represents engineering notation
- and more

`huge` and `digits` represent intrinsic functions in the Fortran language and are defined in the standard.
`huge` returns the largest number representable by a number of the same type and kind as the argument.

`digits` returns the number of significant digits for numbers of the same type and kind as the argument.

Kind Parameter

In Fortran90 the `KIND` parameter was introduced to allow more flexibility for the user in declaring variable precision. The following code shows how KIND can be used.

```
program kindtype

implicit none

integer, parameter   :: i4=SELECTED_INT_KIND(4)
integer, parameter   :: i8=SELECTED_INT_KIND(8)
integer, parameter   :: r4=SELECTED_REAL_KIND(6,37)
integer, parameter   :: r8=SELECTED_REAL_KIND(15,307)
integer(KIND=i4) :: ia
integer(KIND=i8) :: ib
real(KIND=r4) :: ra
real(KIND=r8) :: rb
print *,' Integer 4 ',huge(ia),kind(ia)
print *,' Integer 8 ',huge(ib),kind(ib)
print *,' Real 4 ',huge(ra),kind(ra)
print *,' Real 8 ',huge(rb),kind(rb)
stop
end program kindtype
```

Kind Example

Compile and run this example `kindtype.f90` and compare the output with the results from `variables.f90`.

Note the use of the `PARAMETER` statement used to define a constant that may be used several times throughout the program.

Try adding another variable or two with different values for the selection of the kind and see what results.

Outline

Numeric Expressions

```
+       Addition
-       Subtraction
*       Multiplication
/       Division
**      Exponentiation
```

There is an order of precedence with ** the highest followed by * and /. Next are the unary operators + and - and last the binary operators + and -. For example,

A**B*C is evaluated as (A**B)*C

A/B*C is evaluated as (A/B)*C

Use parantheses to force a particular order of evaluation.

Data Type of Numeric Expressions

If every operand in a numeric expression is of the same type, the result is also of that type.

If operands of different data types are combined in an expression, the data type of the result is the same as the highest ranking operand. For example,

```
double precision :: x, y
y = x*2
```

The integer constant 2 will be promoted to double precision 2.0D0 before doing the multiplication.

Best not to rely on this but be sure that all expressions contain variables or constants of the same type.

Integer Arithmetic

Care must be taken when using all integer variables. The code segment

```
real    :: a
a = 3/2
```

will produce the result a=1.0 not a=1.5

Compile and run the code exercise_int.f90 and see how the value of a can be calculated.

Outline

Do Loops

Loops are used for repeated of execution of similar instructions. For example,

```
program looping
implicit none
integer  :: i, n
real*8   :: sum
n = 10
sum = 0.0D0
do i=1,n
  sum = sum + dble(i)
  write (*,*) ' Value of sum at step ',i,' is ',sum
enddo
stop
end program looping
```

Do Loops (Contd)

Things to note:

- Loop is bounded by `do` and `enddo`
- This can be replaced by a statement number such as

```
      do 5 i=1,n
         sum = sum + dble(i)
5     continue
```

- Loop index is an integer.
- There can be loops within loops using a different index.
- Note 0.0D0 meaning zero in double precision.
- Note the matching of types in the summation step.

Exercise 1

Write Fortran code to calculate and print out the value of the variable a where a=5+1/i for i=1,..,n and n=10.

Then modify the code so that a is a one-dimensional array and the iteration of the do loop for i calculates a(i).

Outline

Conditionals

Logical expressions can only have the values .TRUE. or .FALSE.
There are several relational operators, Fortran77 uses the form on the left and Fortran90 can use either form.

```
.LT.  meaning  <   less than
.LE.           <=  less than or equal
.GT.           >   greater than
.GE.           >=  greater than or equal
.EQ.           ==   equal
.NE.           /=  not equal
```

There are also logical operators, .AND., .OR., .NOT.

Here is an example

```
logical  :: x,y,z
x = .TRUE.
y = x .AND. 2>1
z = x .OR. 2<1
```

In this example all three logical variables will be defined as .TRUE.

Conditional (IF) Statements

Can be written on one line e.g.

```
if (resid < 5.0D-10) stop
```

or, in the general form e.g.

```
if (resid < 5.0D-10) then
   write (*,*)' Residual is less than 5.0D-10'
   stop
else
    write (*,*)' Continue execution'
endif
```

There can even be more levels

```
if (resid < 5.0D-10) then
   write (*,*)' Residual is less than 5.0D-10'
   stop
elseif (num_iters > 100) then
   write (*,*)' Number of iterations exceeded'
   go to 25
else
   write (*,*)' Continue execution'
endif
```

Conditional (IF) Statements

When the IF statement is used within a DO loop and you want to exit the loop but continue the code use EXIT as in the following

```
loop: do i=1,10
        if (resid < 5.0D-10) exit loop
      enddo loop
```

Note the use of optional naming of the DO loop.

DO WHILE statements

The DO WHILE statement executes the range of a DO construct while a specified condition remains true. For example,

```
i=0
do while (resid >= 5.0D-10)
   resid = abs(x(i))
   write (*,*) ' Continue execution'
   i = i+1
end do
```

abs stands for absolute value.

Exercise 2

Extend the code you wrote for Exercise 1 so that it exits the loop when a(i) is within .01 of the asymptotic result 5.

Do this first using an IF statement then using the DO WHILE construct. For the DO WHILE case use the very first example code you wrote where a is not an array. You will need to initialise both the variable a and a variable for the iteration.

Print out the value of the iteration count at which the code completes. Do these two methods give the same answer?

Outline

Fortran Intrinsics

In the previous exercise we introduced ABS for the absolute value of a double precision variable. These are Fortran intrinsic functions. Other examples are

```
sin(a)    sin of the angle a
cos(b)    cosine of the angle b
sqrt(x)   square root of x
exp(z)    exponential of z
log(x)    natural logarithm of x
max(a,b)  maximum of a and b
mod(a,b)  remainder when a is divided by b
```

See the Fortran Reference Manual for a complete list.

Functions and Subroutines

There are two types of subprograms in Fortran, functions and subroutines. The difference is that a function subprogram is invoked in an expression and returns a single value. A subroutine is invoked by a CALL statement and does not return a particular value.

For example,

```
real*8 :: x,y
x = func(y)
```

shows a function being used, whilst

```
real*8 :: x,y
call subr(x,y)
```

demonstrates a subroutine call where x returns the result and y contains the input value.

Subprograms are used to simplify coding by keeping the main program as uncluttered as possible and calculations which are used several times coded as subprograms.

Example Integration code

To show how to use subprograms we will use this simple integration code.

```
program integration1

implicit none

integer  :: i, j
real(8)  :: x, y, integ

integ = 0.0D0

do j=1,10
  y = dble(j)*1.0D0
  do i=1,10
     x = dble(i) * 2.5D-1
     integ = integ + sin(x+y)
   enddo
enddo

write (*,'(a,e20.10e3)')' Integration value = ',integ

end program integration1
```

Compile and run this code, `integration1.f90`.

Functions

We can use a function call to calculate sin(x+y) as follows:

```
program integration2
implicit none

integer  :: i, j
real(8)  :: x, y, func, integ
real(8)  :: val

integ = 0.0D0

do j=1,10
  y = dble(j)*1.0D0
  do i=1,10
     x = dble(i) * 2.5D-1
     val = func(x,y)
     integ = integ + val
   enddo
enddo

write (*,'(a,e20.10e3)')' Integration value = ',integ

end program integration2
```

Functions (Cont'd)

```
function func(x,y) result(result)
  implicit none
  real(8) :: x,y,result
  result = sin(x+y)
end function func
```

Note the use of the optional Fortran90 keyword RESULT.

The function subprogram can be in a separate file from the main program and compiled separately but then linked in to create the executable.

Subroutines

The calculation of sin(x+y) can be written as a subroutine as follows:

```
program integration3
implicit none

integer  :: i, j
real(8)  :: x, y, integ
real(8)  :: val

integ = 0.0D0

do j=1,10
  y = dble(j)*1.0D0
  do i=1,10
     x = dble(i) * 2.5D-1
     call subr(x,y,val)
     integ = integ + val
   enddo
enddo

write (*,'(a,e20.10e3)')' Integration value = ',integ

end program integration3

subroutine subr(a,b,c)
  implicit none
  real(8) :: a,b,c
  c = sin(a+b)
  return
end subroutine subr
```

Subprogram Exercise

Start with the following code fragment

```
program quadratic
implicit none
real(8) :: a,b,c
complex(8) :: r1,r2
a = 2.0D0
b = 9.0D0
c = 4.0D0
```

then continue the program to calculate the roots of the quadratic ax**2+b*x+c=0 using a subroutine call.

If the roots are complex print a warning to that effect but not the solutions. Print the solutions if they are real.

Try with different values of a, b and c.

There are two possible solutions in `quadratic.f90` and `quadratic2.f90`.

Using Arrays in Subroutines

In most scientific code subroutines will require using arrays for both input and output. In Fortran77 this will generally require passing the size of the arrays as a dummy argument as well as the arrays. We will look at other ways to do this in the Fortran90 course.

Using Arrays in Subroutines (Cont'd)

The integration code can be written using arrays in the subroutines.

```
program integrationarray
implicit none

integer  :: i, j
integer, parameter :: n=10
real(8), dimension(n)  :: x, y
real(8), dimension(n,n) :: val
real(8)  :: integ

integ = 0.0D0

do j=1,10
  y(j) = dble(j)*1.0D0
  x(j) = dble(j) * 2.5D-1
enddo
call subr(x,y,val,n)
do j=1,n
  do i=1,n
     integ = integ + val(i,j)
   enddo
enddo

write (*,'(a,e20.10e3)')' Integration value = ',integ

end program integrationarray
```

Using Arrays in Subroutines (Cont'd)

```
subroutine subr(a,b,c,n)
  implicit none
  integer :: n, i, j
  real(8) :: a(n),b(n),c(n,n)
  do j=1,n
     do i=1,n
     c(i,j) = sin(a(i)+b(j))
     enddo
  enddo
  return
end subroutine subr
```

Outline

Fortran I/O

There are many ways of writing out data.

- `print *,result`
- `write (*,*) result`
- `write (6,*) result`
- `write (6,'(f10.4)') result`
- `write (*,'(10F10.4)') (result(i),i=1,10)`
- ```
      write (*,200) result
  200 format(f10.4)
  ```
  ```

Fortran I/O Cont'd

To open files and read or write data from or to them.

- ```
  open (unit=2,file='ascii_data',form='formatted',status='old')
  read (2,'(10f20.4)') (input(i),i=1,10)
  close(2)
  ```
- ```
  open (unit=3,file='binary_data',form='unformatted',status='un
  if (ierr = 0) write(3,*) data
  close(3)
  ```
- Unit numbers 5 and 6 are conventionally reserved for STDIN and STDOUT

Putting it all Together

Copy the code `integration4.f90` and the file `input.dat`. Compile `integration4.f90` and run it. The output all appears in the file `integration4.dat`.

Try editing `input.dat` and rerunning the code.

Have a good look at the Fortran code and make sure you understand how it works.

Code

```
program integration4
implicit none
real(8)  :: factor = 1.0D0
integer, parameter :: filelen=20
real(8) :: x, y, integ, init
real(8) :: userfunc
real(8), dimension(2) :: step
real(8), dimension(4) :: boundary

call loadinput('input.dat', filelen, init, boundary, step)

integ = init

y = boundary(3)

do while (y <= boundary(4))
   x = boundary(1)
   do while (x <=boundary(2))
      integ = integ + userfunc(x,y,factor)
      x = x + step(1)
   enddo
   y = y + step(2)
enddo

call writeoutput('integration4.dat', filelen, integ, boundary, x, y)

end program integration4
```

Code (Cont'd)

```
function userfunc(x,y,factor) result(func)
  implicit none
  real(8) :: x, y, func, factor
  func = factor * sin(x + y)
end function userfunc

subroutine loadinput(inputfile, filelen, init, boundary, step)
  implicit none
  integer :: filelen
  real(8) :: init
  real(8), dimension(2) :: step
  real(8), dimension(4) :: boundary
  character(filelen) :: inputfile

  open(unit=1, file=inputfile, action='read')

  read(1, *) init
  read(1, *) boundary
  read(1, *) step
  close(1)
  return
end subroutine loadinput
```

Code (Cont'd)

```
subroutine writeoutput(inputfile, filelen, integ, boundary, x, y)
  integer :: filelen
  real(8) :: integ, x, y
  real(8), dimension(4) :: boundary
  character(filelen) :: inputfile

  open(unit=1, file=inputfile, action='write')

  write(1, *) 'Integration value = ',integ
  write(1, *) 'x-range:   ',boundary(1),'<= x <= ',boundary(2)
  write(1, *) 'y-range:   ',boundary(3),'<= y <= ',boundary(4)
  write(1, *) 'Final x and y:   ',x,y
  close(1)
  return
end subroutine writeoutput
```

Outline

Using Libraries

There is a vast literature of highly optimised Fortran code for doing many things, much of it is open source.

For example, the LAPACK library provides a large suite of dense linear algebra routines. The source for LAPACK is available from NETLIB at http://netlib2.cs.utk.edu/ so you can install it anywhere.

To get more information on, enter

```
man lapack
```

To show how it is used we will call the routine ZGEEV to find the eigendecomposition of a 2-dimensional complex matrix.

Read

```
man zgeev
```

to see the full documentation for this routine.

Using libraries (Cont'd)

```
program eigen
implicit none
integer, parameter :: n=10
integer, parameter :: lwork=3*n
complex(8), dimension(n,n) :: array, leftvectors, rightvectors
complex(8), dimension(n) :: eigenvalues
complex(8), dimension(lwork) :: work1
real(8), dimension(2*n) :: work2
real(8), dimension(2) :: rand
integer :: i, j, info
do j=1,n
   do i=1,n
      call random_number(rand)
      array(i,j) = cmplx(rand(1),rand(2))
   enddo
enddo
call zgeev('V','V',n,array,n,eigenvalues,leftvectors,n,rightvectors,  &
      n, work1,lwork,work2,info)
if (info==0) then
   write (*,*)' Eigenvalues '
   do i=1,n
      write (*,*) eigenvalues(i)
   enddo
else
   write (*,*)' ZGEEV failed with info = ',info
endif
stop
end program eigen
```

Using Libraries (Cont'd)

The code eigen.f90 shows this routine in action. Have a good look at it, then compile and run it as follows to link to the LAPACK library.

```
f90 -o eiegen eigen.f90 -lcxml
./eigen
```

Notice that there are two work arrays declared which must be passed to the subroutine. Also an error parameter is returned to flag the success or otherwise of the routine. Always check these error parameters.

Outline

Conclusion

- Finish the problems
- Look at other Fortran codes
- Come to the course next week