Appendix A

A Fortran 90/95 primer

A.1 Introduction to F90/F95

Fortran (FORmula TRANslation) was introduced in 1957 and remains the language of choice for most scientific programming. The latest standard, Fortran 90 and 95, includes extensions that are familiar to users of C. Some of the most important features of Fortran 90 include recursive subroutines, dynamic storage allocation and pointers, user defined data structures, modules, and the ability to manipulate entire arrays.
Fortran 90 is compatible with Fortran 77 and includes syntax that is no longer considered desirable. Fortran 90 does however not allow for pointer algebra. This feature is included in Fortran 95 together with other features such as the FORALL statement.

A.1.1 Introduction

In order to get started, consider the following simple Fortran 90 program which sets up Newton's second law.

```
     PROGRAM newton
     IMPLICIT NONE
     DOUBLE PRECISION :: mass, acceleration, force

! write to screen and ask for the mass in kg
     WRITE(*,*) 'Give the mass in units of kg'
! read in the value
     READ(*,*) mass
! write to screen and ask for the acceleration in ms-2
     WRITE(*,*) 'Give the acceleration'
! read in the value
     READ(*,*) acceleration
     force = mass*acceleration
! write back to screen
     WRITE(*,*) mass, acceleration, force

     END PROGRAM newton
```

The first statement must be a program statement; the last statement must have a corresponding end program statement.
Integer numerical variables and floating point numerical variables are distinguished. The names of all variables must be between 1 and 31 alphanumeric characters of which the first must be a letter and the last must not be an underscore.

The types of all variables must be declared. Real numbers are written as 2.0 rather than 2 and declared as DOUBLE PRECISION. In general we discorauge the use of single precision in scientific computing, the achieved precision is in general not good enough. Comments begin with a ! and can be included anywhere in the program. Statements are written on lines which may contain up to 132 characters. The asterisks (*,*) following WRITE represent the default format for output, i.e., the output is e.g., written out on the screen. Similarly, the READ(*,*) statement means that the program is expecting a line input. Note also the IMPLICIT NONE statement which we strongly recommend the use of. In many Fortran 77 one can see statements like IMPLICIT REAL*8(a-h,o-z), meaning that all variables beginning with any of the above letters are by deafult floating numbers. However, such a usage makes it hard to spot eventual errors due to misspelling of variable names. With IMPLICIT NONE you have to declare all variables and therefore detect possible errors already while compiling.

A.1.2 DO construct

Fortran 90/95 use a do construct to have the computer execute the same statements more than once. An example of a do construct follows from example 4 in chapter 2. There we summed $1/n$ up to a given number, say 1000

```
     PROGRAM series
     IMPLICIT NONE
     DOUBLE PRECISION :: sum
     INTEGER :: n

!    Initialize the sum
     sum = 0.
     DO n = 1, 1000
        sum = sum + 1.0/FLOAT(n)
        WRITE(*,*) n,sum
     ENDDO

     END PROGRAM series
```

Note that n is an integer variable. In this case the do statement specifies the first and last values of n; n increases by unity (default). Note here that we wish to avoid a division with by an integer through the use of FLOAT(n). Moreover, Fortran does not allow floating numbers as loop variables.

A.1.3 Logical constructs

In the next program example, the do loop is exited by satisfying a test.

```
     PROGRAM series_test
     IMPLICIT NONE
     DOUBLE PRECISION :: sum, newterm, relative_change
     INTEGER :: n
!  initialize sum, newterm and relative_change
     sum = 0.; newterm=0.; relative_change=0.

     DO n = 1, 1000
        newterm=1.0/FLOAT(n)
        sum = sum + newterm
        relative_change = newterm/sum
        IF ( relative_change < 0.00001 ) EXIT
     ENDDO
     WRITE(*,*) n, sum, relative_change

     END PROGRAM series_test
```

The features included in the above program include:
A do construct can be exited by using the EXIT statement. The IF construct allows the execution of a sequence of statements (a block) to depend on a condition. The if construct is a compound statement and begins with IF ... THEN and ends with ENDIF. Examples of more general IF constructs using ELSE and ELSEIF statements are given in the program library or in the main text. Another feature to observe is the CYCLE command, which allows the loop variable n to start at a new value. As a rule of thumb, if possible, you should avoid IF statements or calls to other functions if you operate on arrays inside loops. This may reduce the effect of optimizations gained through various compiler options.

A.1.4 Subprograms

Subprograms are called from the main program or other subprograms. Subprograms (subroutines and functions) can be included in modules. The form of a module, subroutine, and a function is similar to that of a main program. A module is accessed in the main program by the use statement. Subroutines are invoked in the main program by using the call statement. A subprogram always has access to other entities in the module. The subprograms in a module are preceded by a contains statement.
Variables and subprograms may be declared public in a module and be available to the main program (and other modules). An example follows here.

```
!
!     This module contains the parametrization of the EOS as
!     a polynomial in density. The number of terms kept in the
!     polynomial expansion is given by number_terms.

      MODULE eos
         DOUBLE PRECISION, ALLOCATABLE, DIMENSION(:), PUBLIC :: polynom_terms
         INTEGER, PUBLIC :: number_terms

         CONTAINS
!
!     rho: energy per particle in units of MeV/fm^3
!     rho=\sum_{i=1}^{number of polynoms} a_i*density^{(i-1)/3}
!
            DOUBLE PRECISION FUNCTION rho(x)
            IMPLICIT NONE
            DOUBLE PRECISION,  INTENT(IN) :: x
            INTEGER :: i

            rho=polynom_terms1(1)
            DO i=2,number_terms1
               rho=rho+polynom_terms1(i)*(x**(FLOAT(i-1)/3.d0))
            ENDDO

            END FUNCTION rho
!
!     pressure in units of MeV/fm^3
!     pressure = density*d rho/d density - rho
!
            DOUBLE PRECISION FUNCTION press(x)
            IMPLICIT NONE
            DOUBLE PRECISION, INTENT(IN)  :: x
            INTEGER :: i

            press=-rho(x)
```

```
            DO i=2,number_terms1
               press=press+(FLOAT(i-1)/3.0*polynom_terms1(i)*(x**(FLOAT(i-1)/3.)))
            ENDDO

            END FUNCTION press

      END MODULE eos
```

INTENT(IN) means that the dummy argument cannot be changed within the subprogram. INTENT(OUT) means that the dummy argument cannot be used within the subprogram until it is given a value with the intent of passing a value back to the calling program. The statement INTENT(INOUT) means that the dummy argument has an initial value which is changed and passed back to the calling program. We recommend that you use these options when calling subprograms
The module(s) can be included in a separate file.

A.1.5 Arrays

An array is declared in the declaration section of a program, module, or procedure using the dimension attribute. Examples include

```
DOUBLE PRECISION, DIMENSION (10) :: x,y
REAL, DIMENSION (1:10) :: x,y
INTEGER, DIMENSION (-10:10) :: prob
INTEGER, DIMENSION (10,10) :: spin
```

The default value of the lower bound of an array is 1. For this reason the first two statements are equivalent to the first. The lower bound of an array can be negative. The last statement is an example of two-dimensional array.
Rather than assigning each array element explicitly, we can use an array constructor to give an array a set of values. An array constructor is a one-dimensional a list of values, separated by commas, and delimited by "(/" and "/)". An example is

```
a(1:3) = (/ 2.0, -3.0, -4.0 /)
```

is equivalent to the separate assignments

```
a(1) = 2.0
a(2) = -3.0
a(3) = -4.0
```

A.1.6 Allocate statement and mathematical operations on arrays

One of the better features of Fortran 90 is dynamic storage allocation. That is, the size of an array can be changed during the execution of the program. To see how the dynamic allocation works in Fortran 90, consider the following simple example where we set up a 4×4 unity matrix.

```
       ......
       IMPLICIT NONE
!      The definition of the matrix, using dynamic allocation
       DOUBLE PRECISION, ALLOCATABLE, DIMENSION(:,:) :: unity
!      The size of the matrix
       INTEGER :: n
!      Here we set the dim n=4
       n=4
!  Allocate now place in memory for the matrix
       ALLOCATE ( unity(n,n) )
!  all elements are set equal zero
```

```
       unity=0.
!      setup identity matrix
       DO i=1,n
          unity(i,i)=1.
       ENDDO
       DEALLOCATE ( unity)
       .......
```

We always recommend to use to deallocation statement, since this frees space in memory. If the matrix is transferred to a function from a calling program, one can transfer the dimensionality n of that matrix with the call. Another possibility is to determine the dimensionality with the SIZE command, i.e.,

```
   n=SIZE(unity,DIM=1)
```

will give the size of the rows, while using DIM=2 gives that of the columns.
Other useful Fortran 90 intrisic functions are given by the following examples. Suppose we need to find the maximum absolute value of of of the column elements in a two-dimensional matrix. In Fortran 77 we would have to code something like

```
!  loop over rows i
      DO i=1,n
         max_value=0.
!  the loop over columns
         DO j=1,n
            IF (ABS(a(i,j)) > max_value) max_value=ABS(a(i,j))
         ENDDO
!  then we store this number in a one-dimensional vector
         max_value_row(i)=max_value
      ENDDO
```

In Fortran 90 this statement is replaced by one line

```
      max_value_row=MAXVAL(ABS(a), DIM=2)
```

where DIM=2 tells that we are searching among columns. It is understood that all rows i are evaluated simultaneously.
As another example, suppose we need to evaluate

$$w_i = \sum_{j=1}^{n} |x_i + x_j|,$$

which in Fortran 77 would involve a sum over i and j can be written in Fortran 90 using the intrisic function SUM

```
      DO i=1,n
         w(i)=SUM(ABS(x(i)+x)
      ENDDO
```

Similarly, the product

$$w_i = \prod_{j=1, j\neq i}^{n} (x_i - x_j),$$

can be coded by aid of the PRODUCT function

```
      DO i=1,n
         w(i)=PRODUCT(x(i)-x, MASK=(x /= x(i)) )
      ENDDO
```

where the MASK argument prevents that the diagonal terms are included. Another way of writing the above MASK statement is through teh WHERE statement. Consider a division of all matrix elements of a matrix B with a matrix C, i.e., we have

$$\mathbf{A} = \mathbf{B}/\mathbf{C}$$

meaning that we are performing a division for i and j of b_{ij}/c_{ij}. If we wish to avoid division by zero we could write the above equation as

```
      WHERE ( c /= 0) a=b/c
```

and that's all which is needed. The statement inside WHERE checks all matrix elements of the matrix **C** . Other useful functions are the dot products of two vectors, i.e.,

```
      a=DOT_PRODUCT ( b,c)
```

which means $a = b(1)c(1) + b(2)c(2) + \ldots + b(n)c(n)$. This could also have been written as

```
      a=SUM ( b*c)
```

Here's an example of a module for matrix operations which employs the module mesh_variables through the USE statement.

```
      MODULE matrix_manipulations

      USE mesh_variables
      IMPLICIT NONE

      CONTAINS

!  perform the outer product of two vectors a and b

          FUNCTION outer_product(a,b)

          DOUBLE PRECISION, INTENT(IN) , DIMENSION(:) :: a, b
          DOUBLE PRECISION, DIMENSION(SIZE(a), SIZE(b)) :: outer_product

          outer_product = SPREAD( a,DIM=2, NCOPIES=SIZE(b) ) * &
                          SPREAD( b,DIM=1, NCOPIES=SIZE(a) )

          END FUNCTION outer_product

!  multiply a matrix with a vector

          FUNCTION matrix_vector_mult(a,b)

          DOUBLE PRECISION, INTENT(IN) , DIMENSION(:) :: b
          DOUBLE PRECISION, DIMENSION(:,:), INTENT(IN)  :: a
          DOUBLE PRECISION, DIMENSION(SIZE(b) ) :: matrix_vector_mult
          INTEGER :: i
          DO i=1, SIZE(b)
             matrix_vector_mult(i)=SUM(a(i,:)*b(:))
          ENDDO

          END FUNCTION matrix_vector_mult

      END MODULE matrix_manipulations
```

For more functions, we recommend that you consult a F90 manual or see the link to F90 lectures on the web page.

A.1.7 Complex variables

Fortran 90 is uniquely suited to handle complex variables through variable definitions like COMPLEX and functions like AIMAG.

```
     PROGRAM complex_example
     IMPLICIT NONE
     DOUBLE PRECISION, parameter :: pi = 3.141592654
     COMPLEX :: b,bstar,f,arg, a
     DOUBLE PRECISION :: c
     INTEGER :: d

! A complex constant is written as two real numbers, separated by
! a comma and enclosed in parentheses.

     a = (2.D0,-3.D0)
! If one of part has a kind, the other part must have same kind
     b = (0.5D0,0.8D0)
     WRITE(*,*) 'a =, a
     WRITE(*,*) 'a*a =', a*a
     WRITE(*,*) 'b =', b
     WRITE(*,*) 'a*b =', a*b
! real part of b
     c = REAL(b)
     WRITE(*,*) 'real part of b =', c
! imaginary part of b
     c = aimag(b)
     WRITE(*,*) 'imaginary part of b =', c
     arg = CMPLX(0.0D0,pi)
     b = EXP(arg)
! complex conjugate of b
     bstar = CONJG(b)
! absolute value of b
     f = ABS(b)
     WRITE(*,*) 'properties of b =', b,bstar,b*bstar,f

     END PROGRAM complex_example
```

As another example, you could define your own complex operations through the following example.

```
      MODULE complex_operations
      IMPLICIT NONE
      TYPE complex_variable
          DOUBLE PRECISION :: real_part, complex_part
      END TYPE complex_variable

      CONTAINS

         FUNCTION addition(a,b)

         TYPE (complex_variable), INTENT(IN)  :: a, b
         TYPE (complex_variable) :: addition

         addition = &
             complex_variable(a%real_part+b%real_part,a%complex_part+b%complex_part )
```

```
         END FUNCTION addition

         FUNCTION subtraction(a,b)

         TYPE (complex_variable), INTENT(IN)  :: a, b
         TYPE (complex_variable) :: subtraction

         subtraction = &
             complex_variable(a%real_part-b%real_part,a%complex_part-b%complex_part

         END FUNCTION subtraction

         FUNCTION multiplication(a,b)

         TYPE (complex_variable), INTENT(IN)  :: a, b
         TYPE (complex_variable) :: multiplication

         multiplication%real_part = &
             a%real_part*b%real_part-a%complex_part*b%complex_part

         multiplication%complex_part = &
             a%real_part*b%complex_part+a%complex_part*b%real_part

         END FUNCTION multiplication

      END MODULE complex_operations
```