Basic concept of Fortran Programming
M.Sc. 2nd Semester
MPHYCC-5: Modelling and Simulation
Unit I (Part 1)

Compiled by

Dr. Ashok Kumar Jha
Assistant Professor
Department of Physics, Patna University
Mob:7903067108, Email: ashok.jha1984@gmail.com

Originally Designed by
Erik Boman,
Stanford, December 1995.

1. What is Fortran?

Fortran is a general purpose programming language, mainly intended for mathematical computations in e.g. engineering. Fortran is an acronym for FORmula TRANslation, and was originally capitalized as FORTRAN. However, following the current trend to only capitalize the first letter in acronyms, we will call it Fortran. Fortran was the first ever high-level programming languages. The work on Fortran started in the 1950's at IBM and there have been many versions since. By convention, a Fortran version is denoted by the last two digits of the year the standard was proposed. Thus we have

* Fortran 66
* Fortran 77
* Fortran 90 (95)

The most common Fortran version today is still Fortran 77, although Fortran 90 is growing in popularity. Fortran 95 is a revised version of Fortran 90. There are also several versions of Fortran aimed at parallel computers.

2. Fortran 77 Basics

A Fortran program is just a sequence of lines of text. The text has to follow a certain syntax to be a valid Fortran program. We start by looking at a simple example:

```
      program circle
      real r, area

c This program reads a real number r and prints
c the area of a circle with radius r.

      write (*,*) 'Give radius r:'
      read (*,*) r
      area = 3.14159*r*r
      write (*,*) 'Area = ', area

      stop
      end
```

The lines that begin with with a "c" are comments and has no purpose other than to make the program more readable for humans. Originally, all Fortran programs had to be written in all upper-case letters. Most people now write lower-case since this is more legible, and so will we.

Program organization

A Fortran program generally consists of a main program (or driver) and possibly several subprograms (or procedures or subroutines). For now we will assume all the statements are in the main program; subprograms will be treated later.

The structure of a main program is:

```
Program Name
Declarations
Statement
Stop
End
```

The stop statement is optional and may seem superfluous since the program will stop when it reaches the end anyways, but it is recommended to always terminate a program with the stop statement to emphasize that the execution flow stops there.

Column position rules

Fortran 77 is not a free-format language, but has a very strict set of rules for how the source code should be formatted. The most important rules are the column position rules:

Col. 1 : Blank, or a "c" or "*" for comments
Col. 2-5 : Statement label (optional)
Col. 6 : Continuation of previous line (optional)
Col. 7-72 : Statements
Col. 73-80: Sequence number (optional, rarely used today)

Most lines in a Fortran 77 program starts with 6 blanks and ends before column 72, i.e. only the statement field is used. Note that Fortran 90 allows free format.

Comments

A line that begins with the letter "c" or an asterisk in the first column is a comment. Comments may appear anywhere in the program. The exclamation mark may appear anywhere on a line (except in positions 2-6). Continuation

Occasionally, a statement does not fit into one single line. One can then break the statement into two or more lines, and use the continuation mark in position 6. Example:

```
c23456789 (This demonstrates column position!)

c The next statement goes over two physical lines
      area = 3.14159265358979
     +       * r * r
```

Any character can be used instead of the plus sign as a continuation character. It is considered good programming style to use either the plus sign, an ampersand, or numbers (2 for the second line, 3 for the third, and so on). Blank spaces

Blank spaces are ignored in Fortran 77. So if you remove all blanks in a Fortran 77 program, the program is still syntactilly correct but almost unreadable for humans.

3. Variables, types, and declarations

Variable names

Variable names in Fortran consist of 1-6 characters chosen from the letters a-z and the digits 0-9. The first character must be a letter. (Note: Fortran 90 allows variable names of arbitrary length). Fortran 77 does not distinguish between upper and lower case, in fact, it assumes all

input is upper case. However, nearly all Fortran 77 compilers will accept lower case. If you should ever encounter a Fortran 77 compiler that insists on upper case it is usually easy to convert the source code to all upper case. Types and declarations

Every variable should be defined in a declaration. This establishes the type of the variable. The most common declarations are:

```
integer   list of variables
real      list of variables
double precision list of variables
complex   list of variables
logical   list of variables
```

should be declared exactly once. If a variable is undeclared, Fortran 77 uses a set of implicit rules to establish the type. This means all variables starting with the letters i-n are integers and all others are real. Many old Fortran 77 programs uses these implicit rules, but you should not! The probability of errors in your program grows dramatically if you do not consistently declare your variables.

Integers and floating point variables

Fortran 77 has only one type for integer variables. Integers are usually stored as 32 bits (4 bytes) variables. Therefore, all integer variables should take on values in the range [-m,m] where m is approximately 2*10^9.

Fortran 77 has two different types for floating point variables, called real and double precision. While real is often adequat, some numerical calculations need very high precision and double precision should be used. Usually a real is a 4 byte variable and the double precision is 8 bytes, but this is machine dependent. Some non-standard Fortran versions use the syntax real*8 to denote 8 byte floating point variables.

The parameter statement

Some constants appear many times in a program. It is then often desirable to define them only once, in the beginning of the program. This is what the parameter statement is for. It also makes programs more readable. For example, the circle area program should rather have been written like this:

```
program circle
real r, area, pi
parameter (pi = 3.14159)

c This program reads a real number r and prints
c the area of a circle with radius r.

write (*,*) 'Give radius r:'
read (*,*) r
area = pi*r*r
write (*,*) 'Area = ', area

stop
end
```

The syntax of the parameter statement is

parameter (name = constant, ... , name = constant)

The rules for the parameter statement are:

* The "variable" defined in the parameter statement is not a variable but rather a constant whose value can never change
* A "variable" can appear in at most one parameter statement
* The parameter statement(s) must come before the first executable statement

Some good reasons to use the parameter statement are:

* it helps reduce the number of typos
* it is easy to change a constant that appears many times in a program

4. Expressions and assignment

Constants

The simplest form of an expression is a constant. There are 6 types of constants, corresponding to the 6 data types. Here are some integer constants:

1
0
-100
32767
+15

Then we have real constants:

1.0
-0.25
2.0E6
3.333E-1

The E-notation means that you should multiply the constant by 10 raised to the power following the "E". Hence, 2.0E6 is two million, while 3.333E-1 is approximately one third.

For constants that are larger than the largest real allowed, or that requires high precision, double precision should be used. The notation is the same as for real constants except the "E" is replaced by a "D". Examples:

2.0D-1
1D99

Here 2.0D-1 is a double precision one-fifth, while 1D99 is a one followed by 99 zeros.

The next type is complex constants. This is designated by a pair of constants (integer or real), separated by a comma and enclosed in parantheses. Examples are:

(2, -3)
(1., 9.9E-1)

The first number denotes the real part and the second the imaginary part.

The fifth type is logical constants. These can only have one of two values:

.TRUE.
.FALSE.

Note that the dots enclosing the letters are required.

The last type is character constants. These are most often used as an array of characters, called a string. These consist of an arbitrary sequence of characters enclosed in apostrophes (single quotes):

'ABC'
'Anything goes!'
'It is a nice day'

Strings and character constants are case sensitive. A problem arises if you want to have an apostrophe in the string itself. In this case, you should double the apostrophe:

'It''s a nice day'

Expressions
The simplest expressions are of the form

operand operator operand

and an example is

x + y

The result of an expression is itself an operand, hence we can nest expressions together like

x + 2 * y

This raises the question of precedence: Does the last expression mean x + (2*y) or (x+2)*y? The precedence of arithmetic operators in Fortran 77 are (from highest to lowest):

**** {exponentiation}**
***,/ {multiplication, division}**
+,- {addition, subtraction}

All these operators are calculated left-to-right, except the exponentiation operator **, which has right-to-left precedence. If you want to change the default evaluation order, you can use parentheses.

The above operators are all binary operators. there is also the unary operator - for negation, which takes precedence over the others. Hence an expression like -x+y means what you would expect.

Extreme caution must be taken when using the division operator, which has a quite different meaning for integers and reals. If the operands are both integers, an integer division is performed, otherwise a real arithmetic division is performed. For example, 3/2 equals 1, while 3./2. equals 1.5.

Assignment
The assignment has the form

variable_name = expression

The interpretation is as follows: Evaluate the right hand side and assign the resulting value to the variable on the left. The expression on the right may contain other variables, but these never change value! For example,

area = pi * r2**

does not change the value of pi or r, only area.
Type conversion
When different data types occur in the same expression, type conversion has to take place, either explicitly or implicitly. Fortran will do some type conversion implicitly. For example,

real x
x = x + 1

will convert the integer one to the real number one, and has the desired effect of incrementing x by one. However, in more complicated expressions, it is good programming practice to force the necessary type conversions explicitly. For numbers, the following functions are available:

int
real
dble
ichar
char

The first three have the obvious meaning. ichar takes a character and converts it to an integer, while char does exactly the opposite.

Example: How to multiply two real variables x and y using double precision and store the result in the double precision variable w:

w = dble(x)*dble(y)

Note that this is different from

w = dble(x*y)

5. Logical expressions
Logical expressions can only have the value .TRUE. or .FALSE.. A logical expression can be formed by comparing arithmetic expressions using the following relational operators:

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

So you cannot use symbols like < or = for comparison in Fortran 77, but you have to use the correct two-letter abbreviation enclosed by dots! (Such symbols are allowed in Fortran 90, though.)

Logical expressions can be combined by the logical operators .AND. .OR. .NOT. which have the obvious meaning.
Logical variables and assignment
Truth values can be stored in logical variables. The assignment is analagous to the arithmetic assignment. Example:

```
logical a, b
a = .TRUE.
b = a .AND. 3 .LT. 5/2
```

The order of precedence is important, as the last example shows. The rule is that arithmetic expressions are evaluated first, then relational operators, and finally logical operators. Hence b will be assigned .FALSE. in the example above.

Logical variables are seldom used in Fortran. But logical expressions are frequently used in conditional statements like the if statement.

6. The if statements

An important part of any programming language are the conditional statements. The most common such statement in Fortran is the if statement, which actually has several forms. The simplest one is the logical if statement:

```
if (logical expression) executable statement
```

This has to be written on one line. This example finds the absolute value of x:

```
if (x .LT. 0) x = -x
```

If more than one statement should be executed inside the if, then the following syntax should be used:

```
if (logical expression) then
   statements
endif
```

The most general form of the if statement has the following form:

```
if (logical expression) then
   statements
elseif (logical expression) then
   statements
 :
 :
else
   statements
endif
```

The execution flow is from top to bottom. The conditional expressions are evaluated in sequence until one is found to be true. Then the associated code is executed and the control jumps to the next statement after the endif. Nested if statements

if statements can be nested in several levels. To ensure readability, it is important to

use proper indentation. Here is an example:

```
if (x .GT. 0) then
  if (x .GE. y) then
    write(*,*) 'x is positive and x >= y'
  else
    write(*,*) 'x is positive but x < y'
  endif
elseif (x .LT. 0) then
  write(*,*) 'x is negative'
else
  write(*,*) 'x is zero'
endif
```

You should avoid nesting many levels of if statements since things get hard to follow.

7. Loops

For repeated execution of similar things, loops are used. If you are familiar with other programming languages you have probably heard about for-loops, while-loops, and until-loops. Fortran 77 has only one loop construct, called the do-loop. The do-loop corresponds to what is known as a for-loop in other languages. Other loop constructs have to be simulated using the if and goto statements. do-loops

The do-loop is used for simple counting. Here is a simple example that prints the cumulative sums of the integers from 1 through n (assume n has been assigned a value elsewhere):

```
integer i, n, sum

sum = 0
do 10 i = 1, n
  sum = sum + i
  write(*,*) 'i =', i
  write(*,*) 'sum =', sum
10 continue
```

The number 10 is a statement label. Typically, there will be many loops and other statements in a single program that require a statement label. The programmer is responsible for assigning a unique number to each label in each program (or subprogram). Recall that column positions 2-5 are reserved for statement labels. The numerical value of statement labels have no significance, so any integer numbers can be used. Typically, most programmers increment labels by 10 at a time.

The variable defined in the do-statement is incremented by 1 by default. However, you can define any other integer to be the step. This program segment prints the even numbers between 1 and 10 in decreasing order:

```
      integer i

      do 20 i = 10, 1, -2
        write(*,*) 'i =', i
   20 continue
```

The general form of the do loop is as follows:

```
      do label var = expr1, expr2, expr3
        statements
label continue
```

var is the loop variable (often called the loop index) which must be integer. expr1 specifies the initial value of var, expr2 is the terminating bound, and expr3 is the increment (step).

Note: The do-loop variable must never be changed by other statements within the loop! This will cause great confusion.

Many Fortran 77 compilers allow do-loops to be closed by the enddo statement. The advantage of this is that the statement label can then be omitted since it is assumed that an enddo closes the nearest previous do statement. The enddo construct is widely used, but it is not a part of ANSI Fortran 77.

while-loops
The most intuitive way to write a while-loop is

```
      while (logical expr) do
        statements
      enddo
```

or alternatively,

```
      do while (logical expr)
        statements
      enddo
```

The statements in the body will be repeated as long as the condition in the while statement is true. Even though this syntax is accepted by many compilers, it is not ANSI Fortran 77. The correct way is to use if and goto:

```
label if (logical expr) then
        statements
        goto label
      endif
```

Here is an example that calculates and prints all the powers of two that are less than or equal to 100:

```
    integer n

    n = 1
 10 if (n .le. 100) then
      n = 2*n
      write (*,*) n
      goto 10
    endif
```

until-loops
If the termination criterium is at the end instead of the beginning, it is often called an until-loop. The pseudocode looks like this:

```
    do
      statements
    until (logical expr)
```

Again, this should be implemented in Fortran 77 by using if and goto:

```
label continue
      statements
    if (logical expr) goto label
```

Note that the logical expression in the latter version should be the negation of the expression given in the pseudocode!

Loops in Fortran 90
Fortran 90 has adopted the do-enddo construct as its loop construct. So our "counting down in twos" example will look like this:

```
    do i = 10, 1, -2
      write(*,*) 'i =', i
    end do
```

For while and until loops you also use the do-enddo construct, but you have to add a conditional exit statement. The general case is:

```
    do
      statements
      if (logical expr) exit
      statements
    end do
```

If you have the exit condition at the beginning it is a while loop, and if it is at the end you have an until loop.

8. Arrays

Many scientific computations use vectors and matrices. The data type Fortran uses for representing such objects is the array. A one-dimensional array corresponds to a vector, while a two-dimensional array corresponds to a matrix. To fully understand how this works in Fortran 77, you will have to know not only the syntax for usage, but also how these objects are stored in memory in Fortran 77.

One-dimensional arrays

The simplest array is the one-dimensional array, which is just a linear sequence of elements stored consecutively in memory. For example, the declaration

```
real a(20)
```

declares a as a real array of length 20. That is, a consists of 20 real numbers stored contiguously in memory. By convention, Fortran arrays are indexed from 1 and up. Thus the first number in the array is denoted by a(1) and the last by a(20). However, you may define an arbitrary index range for your arrays using the following syntax:

```
real b(0:19), weird(-162:237)
```

Here, b is exactly similar to a from the previous example, except the index runs from 0 through 19. weird is an array of length 237-(-162)+1 = 400.

The type of an array element can be any of the basic data types. Examples:

```
integer i(10)
logical aa(0:1)
double precision x(100)
```

Each element of an array can be thought of as a separate variable. You reference the i'th element of array a by a(i). Here is a code segment that stores the 10 first square numbers in the array sq:

```
      integer i, sq(10)

      do 100 i = 1, 10
         sq(i) = i**2
  100 continue
```

A common bug in Fortran is that the program tries to access array elements that are out of bounds or undefined. This is the responsibility of the programmer, and the Fortran compiler will not detect any such bugs!

Two-dimensional arrays

Matrices are very important in linear algebra. Matrices are usually represented by two-dimensional arrays. For example, the declaration

```
real A(3,5)
```

defines a two-dimensional array of 3*5=15 real numbers. It is useful to think of the first index as the row index, and the second as the column index. Hence we get the graphical picture:

```
(1,1) (1,2) (1,3) (1,4) (1,5)
(2,1) (2,2) (2,3) (2,4) (2,5)
(3,1) (3,2) (3,3) (3,4) (3,5)
```

Two-dimensional arrays may also have indices in an arbitrary defined range. The general syntax for declarations is:

```
name (low_index1 : hi_index1, low_index2 : hi_index2)
```

The total size of the array is then

```
size = (hi_index1-low_index1+1)*(hi_index2-low_index2+1)
```

It is quite common in Fortran to declare arrays that are larger than the matrix we want to store. (This is because Fortran does not have dynamic storage allocation.) This is perfectly legal. Example:

```
      real A(3,5)
      integer i,j
c
c     We will only use the upper 3 by 3 part of this array.
c
      do  20  j  =  1,  3
        do 10 i = 1, 3
          a(i,j)                  =
   real(i)/real(j) 10 continue
   20 continue
```

The elements in the submatrix A(1:3,4:5) are undefined. Do not assume these elements are initialized to zero by the compiler (some compilers will do this, but not all).

Storage format for 2-dimensional arrays

Fortran stores higher dimensional arrays as a contiguos linear sequence of elements. It is important to know that 2-dimensional arrays are stored by column. So in the above example, array element (1,2) will follow element (3,1). Then follows the rest of the

second column, thereafter the third column, and so on.

Consider again the example where we only use the upper 3 by 3 submatrix of the 3 by 5 array A(3,5). The 9 interesting elements will then be stored in the first nine memory locations, while the last six are not used. This works out neatly because the leading dimension is the same for both the array and the matrix we store in the array. However, frequently the leading dimension of the array will be larger than the first dimension of the matrix. Then the matrix will not be stored contiguously in memory, even if the array is contiguous. For example, suppose the declaration was A(5,3) instead. Then there would be two "unused" memory cells between the end of one column and the beginning of the next column (again we are assuming the matrix is 3 by 3).

This may seem complicated, but actually it is quite simple when you get used to it. If you are in doubt, it can be useful to look at how the address of an array element is computed. Each array will have some memory address assigned to the beginning of the array, that is element (1,1). The address of element (i,j) is then given by

addr[A(i,j)] = addr[A(1,1)] + (j-1)*lda + (i-1)

where lda is the leading (i.e. column) dimension of A. Note that lda is in general different from the actual matrix dimension. Many Fortran errors are caused by this, so it is very important you understand the distinction!

Multi-dimensional arrays
Fortran 77 allows arrays of up to seven dimensions. The syntax and storage format are analogous to the two-dimensional case, so we will not spend time on this.

The dimension statement
There is an alternate way to declare arrays in Fortran 77. The statements

```
real A, x
dimension x(50)
dimension A(10,20)
```

are equivalent to

```
real A(10,20), x(50)
```

This dimension statement is considered old-fashioned style today.

Assignment

1. Write a fortran program for addition of two integers.
2. Write a fortran program for finding area of a circle with given radius.
3. Describe the construction of two dimensional array in fortran programming.
4. Describe the utility of do loop in fortran program.

Thank you