

Notes on FORTRAN Programming

L. R. Schaeffer
September 2002

1. Introduction

The basics of FORTRAN 77 are provided in these notes. These notes do not cover FORTRAN 90, which is similar to the C language in concepts and features. FORTRAN 77 is basic and should run on nearly all machines, and should compile under FORTRAN 90. However, FORTRAN 90 programs will not compile under FORTRAN 77 compilers.

The FORTRAN language was developed many years ago for use in scientific computing. The language is fairly simple, but tedious for doing complicated functions. A FORTRAN program consists of a main program and often functions, which are subroutines of the main program. The idea is to keep the main program as short, concise and easy to follow as possible, and to use subroutines to do the main work.

FORTRAN 77 has a specific formatting protocol which has to be followed. This was originally based on computer cards which had 80 columns per card. FORTRAN program statements are required to start in column 7 and end in column 72. The last 8 columns were used to number the cards sequentially (in case they were ever dropped). Column 6 is known as the continuation column and should have an `x` in that column to continue the statement from the previous line. Columns 2 to 5 were used to provide statement labels for branching purposes, and column 1 was used to indicate a comment card. Examples are given below.

```
1|2345|6|789012345678901234567890123456789012345678901234567890
|   |   |
c| Thi|s| is a comment card - can have anything on it
|   | |a=0.0
|   | |b=5.0
|   | |limit=80.0
| 10 | |a = a + b
|   | |write(10,4001)a,
|   | |x| b
|   | |if(a.gt.limit)go to 20
|   | |go to 10
| 20 | |stop
|   | |end
```

2. Variable Types

Although FORTRAN does not require all variables to be declared before they are used, declaring all variables is often useful to the programmer. Only three basic types are discussed here. They are integer, real, and character data types. Variables take up memory space in the computer. Integer variables typically take up 4 bytes of memory, but there are variables of type `Integer*2` which only use 2 bytes. Similarly, real variables come in `Real*4` (single precision) or

Real*8 (double precision), and on some machines Real*16 or larger. Character variables can be of any length up to 256 characters and each character takes up 1 byte of memory. Below are some examples of declaring variables.

```
Real*8 a, diag, x(10)
Real*4 count
Integer limit, max, min, no(20,40)
Character*12 name, list(100)
```

Integers or non-decimal numbers are in the range of -2147483647 to +2147483647. `no(20,40)` is an array of 800 integer elements arranged in 20 rows and 40 columns. Real variables are floating point numbers with decimal places, such as 3.141519. Real*8 variables can store larger numbers than Real*4 variables, but take up twice as much memory. `x(10)` is an array of 10 floating point numbers while `a` and `diag` are scalar variables. `name` is a character variable consisting of 12 characters, and `list` is an array of 100 character variables each having 12 characters.

FORTTRAN has a convention that any variable name starting with the letters i, j, k, l, m, or n are automatically taken to be Integer variables, unless they are declared to be something else, and all other variable names were automatically taken to be Real*4 variables. Thus, FORTRAN programmers have been lax in declaring variable types. Always declare your variable types. These declarations are made at the beginning of every program or subroutine.

Variable names can be up to 6 characters in length. Thus, choosing names for variables can be a problem for a programmer because 6 characters is very limiting. FORTRAN does not distinguish between capital or small letters, therefore, the following statement would produce an error during compilation because the user is trying to declare the same variable four times.

```
integer mu, MU, Mu, mU
```

There are reserved words or names which should not be used as identifiers. Some of these words are as follows:

mod	float	include	stop	end
abs	do	continue	if	return
dabs	double	fortran	common	subroutine
else	go to	function	parameter	write
read	print	format	call	then

All variables in FORTRAN are only applicable to the main program or to the subroutines. Thus, if the name MILK was declared in the main program, then subroutines may also have a variable called MILK, but they would be two completely different variables with different values. The PARAMETER statement or the COMMON statement are ways to ensure that a variable name and its value are the same between the main program and the subroutine, but this is usually not needed.

3. PARAMETER Statement

The PARAMETER statement is a way of defining fixed values to specific variables. The value remains constant throughout the running of the program. This allows the user to change the dimensions of the program variables by making one change. Consider the following example:

```
PARAMETER (nam=2000, nfx=30)
Real*8 adia(nam), asoln(nam), brhs(nfx), bhat(nfx)
Integer MS(nam), MD(nam)
```

`nam` has been set to 2000 and might represent the number of animals in an analysis. `nfx` might represent the number of levels of a fixed effect in the model. The variable type declarations follow the PARAMETER statement, and note that `nam` and `nfx` have been used to allocate the size of the arrays. Thus, if you had 200,000 animals instead of 2,000, then just change the value of `nam` in the PARAMETER statement. `nam` and `nfx` have the same numeric value throughout the entire program. The user needs to build in checks into the program so that the number of animals does not exceed `nam`, otherwise the program could fail and/or produce undesirable results.

If the user wants to use those same values of `nam` and `nfx` in a subroutine, then the PARAMETER statement must also appear in that subroutine.

4. COMMON Statement

COMMON blocks are needed if the size and number of arrays is very large. This allows for more efficient utilization of memory during compilation. COMMON blocks are also needed for transferring large arrays between the main program and subroutines. A COMMON block should be set up separately for each variable type in order to avoid boundary alignment problems in memory.

Boundary alignments - 8 byte boundaries occur every 8 bytes, integer (integer*2) boundaries occur every 4 (2) bytes. Suppose the following statements were in a program:

```
COMMON MS, MD, AN, ADIAG
REAL*8 ADIAG
INTEGER MS, MD, AN
```

A boundary alignment problem would occur because the COMMON statement is forcing ADIAG to be stored immediately after AN in memory on a 4 byte boundary rather than on an 8 byte boundary. Boundary alignment problems can be avoided by having a separate COMMON block for each variable type. The above example would be re-written as

```
COMMON /vint/MS, MD, AN
COMMON /vreal8/ADIAG
REAL*8 ADIAG
INTEGER MS, MD, AN
```

The COMMON blocks are given names (by the user), and in this case they are `vint` and `vrea18`. Each block begins on an 8 byte boundary (which is also a 4 and 2 byte boundary). The COMMON statement is repeated in subroutines in which the variables in the COMMON block are utilized and changed in a subroutine.

5. Operations on Variables

Comments can and should be added to programs to improve readability and to explain what the program is doing. Any line of the program that has a `C` in column 1 is treated as a comment line. Readability is also improved by adding a blank line. FORTRAN statements may also be indented to indicate nesting within previous statements.

Arithmetic operations on variables are multiply (`a*b`), divide (`a/b`), addition (`a+b`), subtraction (`a-b`), and exponentiation (`a**b`). A concern to FORTRAN programmers is efficiency of operations. In order of longest time of operation they are exponentiation, division, multiply, and add or subtract. Therefore, to save time in squaring a number you should use `c2=c*c` in place of `c2=c**2` because exponentiation takes more time than a multiplication to perform. Similarly, `d=x/4.0` should be replaced by `d=x*0.25` because a multiply operation is faster than a division operation. The differences are in microseconds, but if the same operation were repeated a billion times in a program, then the time difference can become very noticeable. In animal breeding, simulation studies or programs that use Gibbs sampling can involve billions of operations, so that efficiency must be considered all the time.

5.1 Assignments

Assignment statements are statements that perform an arithmetic operation and assign the result to a variable. The following are examples of assignment statements.

```
x = 5.0
n = x + 2.0
y = n * (x - 3.93)
g = p / 3.0
iy = 10
```

5.2 Conditional Expressions

Suppose the minimum of `a` or `b` are to be found. The following alternatives exist to accomplish that task.

```
if( a.lt.b )then
    min = a
else
    min = b
endif
```

or

```
min = b
if( a.lt.b ) min=a
```

The second alternative is concise and takes only two lines of code. FORTRAN programmers may prefer the second alternative. The first alternative is required if other computations are needed, for example,

```
if( a.lt.b )then
  min = a
  c = b / a
else
  min = b
  c = a / b
endif
```

Conditional IF statements require operational time somewhere between that of an addition and that of a multiplication. The following are examples of IF statements.

```
if( a.lt.b )c=a
if( b.gt.a )c=a
if( a.eq.b )c=d
if( a.lt.b.and.c.gt.d)x=0.5
if( a.lt.b.or.a.gt.c)y=2.0
```

Note that when `if(a.eq.b)` is used, then the computer must see if `a` is less than `b`, and if not, then it must test if `a` is greater than `b`, and if not, then `a` must equal `b`. Therefore, tests using `.lt.` or `.gt.` are more time efficient.

5.3 Loops

Loops are very important in programming. In genetic evaluation, for example, processing needs to proceed from 1 to the total number of animals, many times during the iterative solving of mixed model equations. There are two kinds of loops in FORTRAN. The most frequently used loop is the DO loop.

```
tss=0.0
v=0.0
DO 25 i=1,nobs
  obs = y(i)
  v = v + obs
```

```

        tss = tss + obs*obs
25  continue

```

When the line 25 `continue` is reached then the value of `i` is increased by 1 and if `i` is less than `nobs+1` then flow of the program goes back to the top of the loop (first statement after `DO 25 i=1,nobs`) otherwise the loop is exited to the next statement after 25 `continue`.

Another kind of loop makes use of the `GO TO` statement and some `IF` statements. An example is reading a data file where the number of records is unknown (or could be variable).

```

        tss=0.0
        v=0.0
        n=0
10  read(inf,1001,end=20)obs
1001 format(2x,f12.4)
        v = v + obs
        tss = tss + obs*obs
        n=n+1
        go to 10
20  print *, n

```

Note that `inf` is a number referring to a particular data file. A `read` operation reads one record from a file, and in this case just one number is retrieved and stored in the variable `obs`. Then `v` is used to sum all of the observations together, `tss` is used to sum all of the squares of the observations, and `n` counts the number of observations read. In the `read` statement is `end=20` which means when the read function reaches the end of the data file and no more records are available, then the program will branch to statement 20, which prints the number of records onto the terminal screen. The `go to 10` keeps sending the program back to statement labeled 10 which it would do endlessly without the `end=20` in the read statement.

6. Input and Output

Input and output (I/O) in FORTRAN is relatively easy compared to the C language. There is the `READ` command and the `WRITE` or `PRINT` commands. However, there are different types of data files that could exist.

6.1 Free Formatted

With a free formatted file there needs to be at least one space inserted between every variable. A space is used to determine where one variable ends and the next begins. This only works for numeric data. An example of such a file is shown below.

```
33 297 3409 62 14
101 155 2874 23 45
8 88 4390 128 53
```

To read this data file the following read statement is used.

```
read(13,*,end=9992)a, b, c, d, e
```

Hence every read command is expecting to find 5 variable values in the data file on each line of the file. Note that * is used to indicate the free formatted style in the read statement.

6.2 Fixed Formatted

A fixed format means that each variable is assigned to specific columns of the data file. The above example in free format could be also stored as

```
33297 3409 62 14
101155 2874 23 45
8 88 4390128 53
```

Note that there does not have to be a space separating variable values. The first variable is contained within the first four columns, the second variable in columns 5, 6, and 7; the third variable in columns 8 to 12; the fourth variable in columns 13 to 15, and the last variable in columns 16 to 18. The read statement would be

```
read(13,2001,end=9992)a, b, c, d, e
2001 format(f4.0, f3.0, f5.0, f3.0, f3.0)
```

Users have to be careful. Suppose the 8 in the third line of the example was in column 2 instead of column 4. The FORTRAN program could read the number as 800.0 instead of 8.0. That is, blank spaces are converted to zeros. One way to avoid that problem is to put the decimal point with the number in the datafile itself. In FORTRAN, the decimal point in the datafile takes precedence over the format specified.

SAS output files often indicate a missing value by a period, '.'. FORTRAN has trouble reading variables that have only a '.' as the value. Missing values have to be indicated by a 0 or some other number that is not possible for that variable.

Character data has to be read in fixed format.

6.3 Unformatted

Unformatted files are the most efficient in terms of saving space. Suppose `iv=2345786` which takes up 7 spaces plus a blank in front of it (in free format) in a data file. Because `iv` is an Integer variable it can be written to a data file as unformatted and will only take up 4 spaces

corresponding to the 4 bytes that an integer variable takes in memory. The binary representation in memory is directly written to the file. Similarly, Real*8 variables would only take up 8 spaces or bytes. Generally, unformatted files take up much less space, and they are faster to read than formatted files. The main disadvantage is that the user can not browse an unformatted file because it has been written in binary on the file. Thus, the user can not visually check the contents of a binary file.

Binary files should be used if the data file is to be read by the program many times, or if it is a very large file, or has very large record length. Otherwise, formatted (ASCII) files are sufficient if the data are read only once, or if the files are very small (numbers of records or record size). Sometimes a system may have special read and write routines for unformatted files, which will speed up reading and writing also.

6.4 Opening-Closing

In UNIX FORTRAN, all files must first be opened before they can be read. The open statement is

```
open(13,file='mydata.d',form='formatted',status='old')
```

where 13 is the unit number assigned to the datafile called `mydata.d`. The unit number can be any number from 9 on up. The file name must be given. The form indicates formatted or unformatted. Status equal to old indicates a file that has already been created by a previous program, and that this program is simply going to read the contents of the file. Other status options are 'new' and 'unknown'. The 'new' option indicates that the file will be created by this program. The file name must, therefore, also be new in that it does not already exist in the directory from which the program will be run. If a file with that name already exists then the program will not get past that open statement and will terminate. The 'unknown' option is similar to 'new' except that the program will go ahead and write over on top of the file that already exists - creating new contents. Thus, the user has to be careful not to writeover previous results, if they are important.

The file name could include the pathway to find it, such as,

```
open(13,file='/u/lrs/Dairy/TYPE/H028type.d',  
x form='formatted',status='unknown')
```

Files can be closed in the program using `CLOSE 13`, to close unit 13. Subsequent reads to file unit 13 would be unsuccessful and would cause the program to terminate. Alternatively, when the program reaches the `STOP` statement, then all files are closed automatically. To be complete and ensured of a successful finish, the user should close files within the program.

If a file is to be read several times in a program, the `REWIND 13` command can be used to close and re-open the file to position the reader at the first record again.

7. Math Functions

Math functions available in FORTRAN are the following:

cosh	atanh	
acos	dabs	
asin	dexp	dlog10
atan	mod	sinh
atan2	dlog	dsqrt

8. Programs

Consider the problem of reading a datafile with a number of data items per record and an unknown number of records. Missing data are indicated in the data by -999. The problem is to read the datafile and find the minimum and maximum values of each data item, the number of non-missing values per data item, and the mean and variance of those that are non-missing. A program might be as follows:

```

c
c Program to find min and max value of all data items, counts,
c and means and variances - written Sept 6, 2002 - LRS
c
c Initialization Section
c
      Parameter (nitem=5, miss=-997)
      real*8 nobs(nitem),mean(nitem),var(nitem),
x nmax(nitem),nmin(nitem),y(nitem),ob, z, zm,zv
      integer i
      do 5 i=1,nitem
          nobs(i)=0
mean(i)=0
          var(i)=0
nmax(i)=miss-2
          nmin(i)=99999999.
      5 continue
c
c Open data files Section
c
      open(9,file='pizza.d',form='formatted',status='old')
      open(17,file='results.d',form='formatted',
x status='unknown')
c
c Read Data and Perform Counts, ETC.
c
10 read(9,*,end=900) y
c
      do 15 i=1,nitem
          if(y(i).lt.miss)go to 15
              ob=y(i)
              nobs(i) = nobs(i) + 1.0

```

```

        mean(i) = mean(i) + ob
        var(i) = var(i) + ob*ob
        if(ob.gt.nmax(i))nmax(i)=ob
        if(ob.lt.nmin(i))nmin(i)=ob
15 continue
    GO TO 10
c
c All data records have been read, do final calculations
c Write out results to file 17
c
900 CLOSE 9
    do 20 i=1,nitem
        z=nobs(i)
        zm = mean(i)/z
        zv = (var(i) - zm*mean(i))/(z-1.0)
        write(17,4001)i,z,nmin(i),zm,nmax(i),zv
4001 format(1x,i3,f10.0,4d20.10)
    20 continue
    CLOSE 17
    Stop
    End

```

Prepare a sample data file and run the program on your data. Do you get the correct results? What happens if a data item has only 1 non-missing value? Or all values are missing? What are the other limitations of this program?