Lecture 3 – More Fortran 95/03

- **So far, we have only briefly mentioned input and output**

- **Now lets look at input and output in a more general way**

- **Formats: Format statements are used to read or write information with headings or for placement of information in desired columns/rows**

Format Statements

- **Formatted read/write statements are assigned numbers.**
- **Real variables can be read/written in a "fixed" format that is specified as F#1.#2 where**
  - #1 is the total number of fields in the number and
  - #2 is the number of digits to the right of the decimal

- **Real variables can also be read/written in "scientific" format that is specified as E#1.#2**
  - #1 is the total number of fields in number (3 for -0., #2 digits, and 4 for powers of 10 such as E+10) Note that $\#1 \geq \#2 + 7$
  - #2 is the number of digits to the right of the decimal
  - Note that this will display a number between 0.1 and 1.0 multiplied by a power of 10.
  - EXAMPLE:

```
TEST = -4096.0
WRITE(*,10) TEST
10 FORMAT(E11.4)
```

would be written as -0.4096E+04

Format Statements

- **Real variables can also be read/written in "true scientific" format that is specified as ES#1.#2**
  - #1 is the total number of fields in the number and
  - #2 is the number of digits to the right of the decimal
  - Note that this will display a number between 1.0 and 10.0 multiplied by a power of 10.
  - EXAMPLE:

```
TEST = -4096.0
WRITE(*,10) TEST
10 FORMAT(ES12.3)
```

    would be written as -4.096E+03

- **Integer variables are read/written in an "integer" format that is specified as I#1 where**
  - #1 is the maximum number of digits expected (right justified)

Format Statements

- **Character variables are read/written in an "alphanumeric" format that is specified as A#1 where**
  - #1 is the maximum number of characters expected (right justified)

EXAMPLE: CHARACTER (LEN=17) :: STRING='This is a string."

```
        WRITE(*,10) STRING
                                  0         1         2
                                  12345678901234567890
10 FORMAT(A)   ➔    This is a string. (exactly 17 characters)
    WRITE(*,20) STRING
20 FORMAT(A20) ➔              This is a string. (right justified)
    WRITE(*,30) STRING
30 FORMAT(A6) ➔     This i (truncated)
```

Format Statements

- **Logical data are read/written with a format that is specified as L#1 where**
  - #1 is the maximum number of fields (right justified)
  - Logical data are read/written as either T (for a .TRUE. variable) or F (for a .FALSE. variable)

EXAMPLE: LOGICAL :: OUTPUT=.TRUE., DEBUG=.FALSE

WRITE(*,10) OUTPUT,DEBUG

```
                                  0         1         2
                                  12345678901234567890
10  FORMAT(2L5) ➔                    T    F
```

Format Statements

- **Multiple variables can be read/written in a string**
- **Example:**

```
REAL :: TEST
INTEGER :: ICHECK
CHARACTER(10) :: CHAR
WRITE(*,100) TEST, ICHECK, CHAR
100 FORMAT(F10.5,I14,A20)
```

- **Note that commas “,” are used to separate the various items included in the format statement**
- **Spaces can be added in format statements using the notation #x where # is the number of spaces**
- **In/output can be performed at a specified column by using the notation T# where # is the column to go to**

Format Statements

- **Multiple variables of a given type can be read/written out by simply putting a number in front of the format descriptor**
- **Example:**

```
WRITE(*,100) TEST1, TEST2, I,J,K
100 FORMAT(2F10.5,3I8)
```

- **Text can be added in format statements by placing the text inside single quotes**
- **Example:**

```
WRITE(*,100) TEST,ICHECK,CHAR
100  FORMAT('TEST =  ',F10.5,2x,  &
   & 'ICHECK = 'I14,3X,'CHAR = ',A20)
```

Format Statements

- **Input and output may be performed from/to different devices**
  - Printer, laser printer, terminal (screen)

- **The computer builds a complete image of each line in memory before sending it to an output device. The computer memory containing the image is called an *output buffer***

- **The first character in the buffer is known as the *control character* which specifies the vertical spacing for the line**

Control Characters

- **The first character in the format statement then specifies line spacing for that statement**

- **Possibilities for the control character are:**

1	**- Skip to new page**
blank	**- Single spacing**
0	**- Double spacing**
+	**- No spacing (print over previous line)**

- **EXAMPLE:**

```
WRITE(*,100) TEST,ICHECK,CHAR
100 FORMAT('0','TEST=',F10.5,I20,A25)
```

would double space prior to writing

Changing To Next Input/Output Line

- **We can change to the next line of input/output by using the "/" character in the format statement. Note that commas are not required to separate /'s**
- **EXAMPLE:**

```
WRITE(*,100) TEST,ICHECK,CHAR
1OO FORMAT('1',F10.5,2X,I20,/,A25)
```

- **EXAMPLE:**

```
WRITE(*,100) TEST,ICHECK,CHAR
100 FORMAT('0',10X,'OUTPUT LISTING',///,&
   & 'INPUT TEST    = ',F10.5,/ &
   & 'INPUT CHECK = ',I20/,     &
   & 'INPUT NAME   = ',A30,//)
```

- **Each new read/write statement automatically skips to a new line as a default**

Special Rules for Read Statements

- **For Real variables using the format descriptor F#1.#2, the number may be input as**
  - Real number with a decimal point
  - Real number in exponential notation
  - Number without a decimal point
- **For the later (number without a decimal point), then a decimal point is assumed to be in the position specified by the #2 term.**
- **For Real and Integer variables, the number may be placed anywhere within the field of #1**
  - The exception to this rule is the real number without a decimal point. This can be dangerous giving improper input.

Special Rules for Read Statements

- **For Logical variables, the input value of T or F must be the first nonblank character in the input field #1**

- **For Character variables,**
  - One can use a general format of just A. In this case, the number of characters equal to the declared character length will be read
  - One can be more descript with A#1. In this case,
    - If #1 is larger than the declared character length, the data from the right-most portion of the field is loaded into the character variable
    - If #1 is smaller than the declared character length, the characters in the field will be stored in the left-most characters of the variable and the remainder of the variable will be padded with blanks

Free Format

- **We have shown examples of the free format in which variables are read in arbitrary fields separated by spaces or commas. The WRITE statements for this are simply**

```
WRITE(*,*) TEST,ICHECK,CHAR
```

which is similar to the read

```
READ(*,*) TEST,ICHECK,CHAR
```

Files and File Processing

- **Today in most cases, information is read and written from/to electronic files**
- **There are different types of electronic files including**
  - Sequential access
  - Direct access
- **And we may want to interact with multiple files**
- **So we need a means of describing to the computer what file and what type of file we wish to use**

Files and File Processing

- **Reads and writes may be performed to different *devices* (file, printer, etc.) that are given numbers. In the past, we have used READ(*,10) and WRITE(*,100) types of statements.**
- **When a "*" is used, the READ or WRITE is performed from the standard device (usually the monitor, if interactive or standard output file if not interactive)**
- **When a # is used, such as READ(10,20) or WRITE(20,100), then the # (10 for the READ or 20 for the WRITE) is the *unit* number given to the device**

Files and File Processing

- **Devices corresponding to electronic files can have different attributes**
  - Each line of a file is called a *record*
  - Files that are read/written one record after another are called *sequential access*
  - Files where records can be read in a specified, non-sequential manner are called *direct access*
  - Files must be *opened* before any read/write and must be *closed* when read/writes are completed

OPEN(UNIT=int_expr,FILE=char_expr,STATUS=char_expr, ACTION=char_expr,
IOSTAT=int_var, FORM=form_expr)

where int_expr is a non-negative integer representing the unit number of the file

char_expr is a character string of the FILE name

STATUS ('OLD', 'NEW', 'REPLACE', 'SCRATCH' OR 'UNKNOWN')

Note: SCRATCH files are automatically deleted upon closing or at program termination

ACTION ('READ', 'WRITE', OR 'READWRITE' (default))

int_var is an integer value with 0 – successful opening, >0 – unsuccessful opening

FORM ('FORMATTED', 'UNFORMATTED', or 'BINARY')

Files and File Processing

- **Files should be closed prior to termination or when they will no longer be used in order to safely save the buffer to secondary storage**

  CLOSE(close_list)
  where close_list usually contains the unit number of the file (e.g. CLOSE(10))

- **Positioning within a file can be performed by:**
  - Moving forward by performing a READ or WRITE
  - Moving backwards one record by performing
    BACKSPACE (UNIT=int_var)
  - Moving back to the beginning of the file by performing
    REWIND (UNIT=int_var)

Arrays

- **Arrays are a group of variables, all with the same type (integer, real, character, logical)**
  - An individual value within the array is called an *element*
  - Its location within the array is identified with a *subscript*
- **Arrays are very useful in engineering computations since they can represent**
  - Experimental data points
  - A structured- or unstructured-grid of coordinate points or variables
- **We need to *declare* the attributes of an array in a program prior to using it. This can be done several ways:**

```
REAL, DIMENSION(25) :: VOLTAGE
CHARACTER (LEN=35), DIMENSION(50) :: GATE_STATUS
```

Note that the dimensions of arrays can be variables. We will discuss this later.

Initializing Arrays

- **The elements of an array should be initialized prior to using them. This can be done in different ways. Examples:**

```
REAL, DIMENSION(10) :: ARRAY1
ARRAY1 = 0.

REAL, DIMENSION(10) :: ARRAY1
ARRAY1 = (/0., 0., 0., 0., 0., 0., 0., 0.,0., 0./)

REAL, DIMENSION(10) :: ARRAY1 = 0.

INTEGER, DIMENSION(5) :: ARRAY2 = (/1, 2, 3, 4, 5/)

INTEGER, DIMENSION(5) :: ARRAY2 = (/ (I, I=1,5) /)

INTEGER, PARAMETER :: ISIZE = 200
REAL :: DIMENSION(ISIZE) :: ARRAY1 = 0.
```

Whole Array Operations

- **If two arrays have the same shape, then ordinary arithmetic or intrinsic operations can be performed on the entire array using scalar-like statements**

```
REAL, DIMENSION(4) :: A = ( /-1., -2., -3., -4./ )
REAL, DIMENSION(4) :: B = (/5., 6., 7., 8./ )
REAL, DIMENSION(4) :: C, D
!WHOLE ARRAY OPERATION
C = A+B
!WHOLE ARRAY INTRINSIC FUNCTION
C = ABS(A)*B
!ELEMENT BY ELEMENT OPERATION
DO I = 1,4
   D = A(I) + B(I)
END DO
```

Arrays

- **Array elements may be referenced by using indicies:**

  array(subscript_1: subscript_2: stride)

  EXAMPLE:

```
REAL, DIMENSION(4) :: A = ( /-1., -2., -3., -4./ )
REAL, DIMENSION(2) :: C
!ARRAY SUB-SET OPERATION
C = A(2:4:2)
```

Higher Rank Arrays

- **We can declare higher rank arrays in the following manner:**

```
REAL, DIMENSION(3,6) :: TEST1

REAL, DIMENSION(-30:20) :: TEST2

INTEGER, DIMENSION(-5:30,0:10,20:30) :: TEST3
```

- **Higher-rank arrays are stored in *column major order* (e.g. array(2,3) is stored in a pipe:**

array(1,1)
array(2,1)
array(1,2)
array(2,2)
array(1,3)
array(2,3)

Two-dimensional arrays can be thought as: ARRAY(COLUMN,ROW)

This is the opposite of how arrays are stored using C.

Higher Rank Arrays

- **Note that this is not the same as one might think about a grid!**

column

(row,column) MATRX/ARRAY

row →

A(1,1) A(1,2) A(1,3)….A(1,j)

A(2,1) A(2,2) A(2,3)….A(2,j)

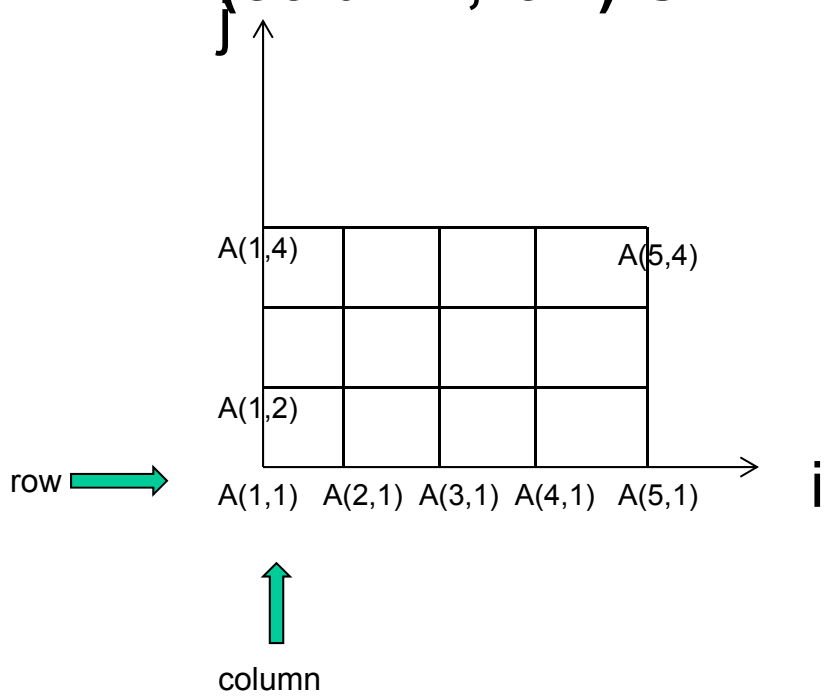
. . . .

. . . .

A(i,1) A(i,2) A(i,3) …. A(i,j)

column-major refers to indexing along column of matrx/array first

(column,row) GRID



column-major refers to indexing the columns of grid first

Whole Array Operations on Higher-Rank Arrays

- **Whole-array operations may be performed on higher-rank arrays as rank-1 arrays**
- **Operations on subsets of arrays may also be performed using subscripts**

```
REAL, DIMENSION(4,4) A
REAL, DIMENSION(4,4) B
REAL, DIMENSION(2) C
REAL, DIMENSION(4) D
! Matrix multiply
D = A(1:)*B(:1)+A(2:)*B(:2)+A(3:)*B(:3)+A(4:)*B(:4)
! Sub-set matrix multiply
C = A(1:3:2,2:4:2)*B(2:4:2,1:3:2)
```

Inquiry Intrinsic Whole Array Functions

- **Here are some intrinsic functions that will determine the characteristics of an array**
  - LBOUND(array,dim) returns all the lower bounds of array (see section 8.3.2 of Chapman)
  - SHAPE(array) returns the shape of array
  - SIZE(array,dim) returns the extent along dimension (see section 8.3.2 of Chapman)
  - UBOUND(array,dim) returns all the upper bounds of array (see section 8.3.2 of Chapman)

Transformational Whole Array Functions

- **There are additional intrinsic functions that can be used to manipulate multiple arrays**

DOT_PRODUCT(vectorA, vectorB) calculates the dot product of two equal-size vectors

MATMUL(matrixA, matrixB) performs matrix multiplication of two conformable arrays

RESHAPE(source,shape) constructs an array of the specified shape from the elements of source. Shape is a rank-1 array containing the extents of each dimension in the array to be built. Reshape maps from the old shape to the new shape in column order.

```
REAL, DIMENSION(3,2) ARRAY1
! ARRAY1 ELEMENTS ARE:
!                ARRAY1(1,1)    ARRAY1(1,2)   ARRAY1(1,3)
!                ARRAY1(2,1)    ARRAY1(2,2)    ARRAY1(2,3)
REAL, DIMENSION(2) SHAPE (/2,3/)
REAL, DIMENSION(2,3) ARRAY2
ARRAY2 = RESHAPE(ARRAY1,SHAPE)
! ARRAY2 ELEMENTS ARE:
!                ARRAY1(1,1)    ARRAY1(1,2)
!                ARRAY1(1,3)    ARRAY1(2,1)
!                ARRAY1(2,2)    ARRAY1(2,3)
```

Structured Programming Style - Subroutines

- **Codes can be programmed as sub-tasks in which each sub-task is in a self-contained *subroutine***
  - The subroutine may be invoked using a *CALL* statement
  - Arguments may be used as part of the CALL statement to pass information

EXAMPLE: In the calling routine:

```
CALL HYPOTENUSE(SIDE1, SIDE2, HYPOT)
```

HERE, SIDE1 and SIDE2 are information passed to the HYPOTENUSE subroutine and HYPOT is information passed back

  - Scalars, arrays, characters, functions, and even subroutines can be passed through the argument list. This is a little slower than using common blocks(old way) or modules (which will be described later).

Intent Statements with Subroutines

- **One can use the *INTENT* statement to declare if the arguments of a subroutine call are *input* or *output.* This is not required but it allows the compiler to catch errors associated with mismatches in the argument lists.**

EXAMPLE: In the subroutine:

```
SUBROUTINE HYPOTENUSE(SIDE1, SIDE2, HYPOT)
REAL, INTENT(IN) :: SIDE1, SIDE2
REAL, INTENT(OUT) :: HYPOT
```

Passing Arrays and Characters

- **The dimensions of arrays and characters may also be passed through the argument list**

EXAMPLE: In the calling routine:

```
REAL, DIMENSION(4) :: xcoord, ycoord
CHARACTER (LEN=10) type = ‘quad’
n = 4
CALL CELL_AREA(xcoord,ycoord,n,type,area)
```

In the subroutine:

```
SUBROUTINE CELL_AREA(xcoord,ycoord,n,type,area)
REAL, INTENT(IN), DIMENSION(N) :: xcoord, ycoord
REAL, DIMENSION(N) :: temp
CHARACTER (LEN=*) type
```

The SAVE Attribute

- **Variables used inside of a subroutine are generally considered *local*. That is, when execution leaves the subroutine, the local variables are not stored.**
  - Reference to those variables in the calling routine will give erroneous results unless the variables are passed back through the argument list (or part of a common block or module)
  - In subsequent returns to the subroutine, reference to the variable may also give erroneous results unless a *SAVE* statement is used

    EXAMPLE: In the subroutine:

    ```
    SUBROUTINE HYPOTENUSE(SIDE1,SIDE2,HYPOT)
    REAL, SAVE :: SUM_SIDE
    ```

    would save the variable SUM_SIDE so that its value would stay the same in between calls to HYPOTENUSE

Functions

- **Programmers can also define *Functions* to perform simple computations**
  - The function can be invoked by simply using its name and arguments rather than using a call statement

EXAMPLE:

```
FUNCTION HYPOT(SIDE1,SIDE2)
REAL, INTENT(IN) :: SIDE1,SIDE2
REAL, INTENT(OUT) :: HYPOT
HYPOT = SQRT(SIDE1**2+SIDE2**2)
RETURN
END FUNCTION
```

So in the calling routine, we could simply write something like:

```
SUM = 0.
DO I = 1,NTOT
   SUM = SUM + HYPOT(SIDE1,SIDE2)
END DO
```

Passing Functions or Subroutines Through Argument Lists

- **Functions and Subroutines may be passed through argument lists using the *External* statement.**

EXAMPLE:

```
Program TEST
REAL, EXTERNAL :: EVAL1, EVAL2
REAL :: X, Y, OUT
CALL EVALUATE(EVAL1,X,Y,OUT1)
CALL EVALUATE(EVAL2,X,Y,OUT2)
END PROGRAM

SUBROUTINE EVALUATE(EVAL,X,Y,OUT)
REAL, EXTERNAL :: EVAL
REAL, INTENT(IN) :: X,Y
REAL, INTENT(OUT) :: OUT
OUT = X*Y +EVAL(X**2+Y**2)
RETURN
END SUBROUTINE EVALUATE
```

Where EVAL1 and EVAL2 are user-supplied functions (with a single argument)

Passing Information with MODULES

- **If a large amount of data needs to be exchanged between routines, or multiple routines need the same information, then it is more efficient to use a *Module***
- **A module is a separately compiled program unit that contains definitions, data values, and even computations (routines) that we wish to share**

EXAMPLE:

```
MODULE TEST
IMPLICIT NONE
SAVE
INTEGER, PARAMETER :: NUMVAL=5
REAL, DIMENSION(NUMVAL) :: VALUES
END MODULE

In another routine:
SUBROUTINE CALC_AREA
USE TEST
…..
END SUBROUTINE CALC_AREA
```

Modules have replaced "common blocks" that were previously used in Fortran77

Routines as Part of Modules

- **Routines and/or Functions can be included inside of Modules using the *Contains* statement**

EXAMPLE:

```
MODULE MY_SUBS
IMPLICIT NONE
…declarations
CONTAINS
   SUBROUTINE SUB1(A,B,C,X,ERROR)
   IMPLICIT NONE
   REAL, DIMENSION(3), INTENT(IN) :: A
   REAL, INTENT(IN) :: B,C
   REAL, INTENT(OUT) :: X
   LOGICAL, INTENT(OUT) :: ERROR
   ….computations
   END SUBROUTINE SUB1
END MODULE MY_SUBS
```

The advantage of using subroutines within modules is that the compiler can track input and output variables between the calling routine and the subroutine. This is the strategy used in Object-Oriented Programming (OOP).

Homework 2

- **Read Chapters 5-10 of Fortran 95/2003 (Chapman)**
- **Begin reading Chap. 1-2 of Introduction to Parallel Computing by Grama et al. if you have the book.**
- **Due Thursday, Oct. 8:**
  - Exercises below and plot the results
  - Provide listing of code
  - Provide output of slope and intercept
  - Provide plot of results

Problem (Chapman)

6–27 Linear Least-Squares Fit Develop a subroutine that will calculate slope m and intercept b of the least-squares line that best fits an input data set. The input data points (x,y) will be passed to the subroutine in two input arrays, X and Y. The equations describing the slope and intercept of the least-squares line are

$$y = m\,x + b \tag{4–5}$$

$$m = \frac{(\Sigma xy) - (\Sigma x)\bar{y}}{(\Sigma x^2) - (\Sigma x)\bar{x}} \tag{4–6}$$

and

$$b = \bar{y} - m\,\bar{x} \tag{4–7}$$

where

Σx is the sum of the x values.
Σx^2 is the sum of the squares of the x values.
Σxy is the sum of the products of the corresponding x and y values.
$\bar{x}$ is the mean (average) of the x values.
$\bar{y}$ is the mean (average) of the y values.

Test your routine using a test driver program and the following 20-point input data set:

Sample data to test least-squares fit routine					
No.	x	y	No.	x	y
1	−4.91	−8.18	11	−0.94	0.21
2	−3.84	−7.49	12	0.59	1.73
3	−2.41	−7.11	13	0.69	3.96
4	−2.62	−6.15	14	3.04	4.26
5	−3.78	−5.62	15	1.01	5.75
6	−0.52	−3.30	16	3.60	6.67
7	−1.83	−2.05	17	4.53	7.70
8	−2.01	−2.83	18	5.13	7.31
9	0.28	−1.16	19	4.43	9.05
10	1.08	0.52	20	4.12	10.95

6–28 Correlation Coefficient of Least-Squares Fit Develop a subroutine that will calculate both the slope m and intercept b of the least-squares line that best fits an input data set and also calculate the correlation coefficient of the fit. The input data points (x,y) will be passed to the subroutine in two input arrays X and Y. The equations describing the slope and intercept of the least-squares line are given in the previous problem, and the equation for the correlation coefficient is

$$r = \frac{n(\Sigma xy) - (\Sigma x)(\Sigma y)}{\sqrt{[(n\Sigma x^2) - (\Sigma x)^2]\,[(n\Sigma y^2) - (\Sigma y)^2]}} \qquad (4\text{–}10)$$

where

Σx is the sum of the x values.
Σy is the sum of the y values.
Σx^2 is the sum of the squares of the x values.
Σy^2 is the sum of the squares of the y values.
Σxy is the sum of the products of the corresponding x and y values.
n is the number of points included in the fit.

Test your routine using a test driver program and the 20-point input data set given in the preceding problem.