FORTRAN Handbook

Second Edition

Michael Thelen

Contents

Preface

This document is the supplemental laboratory text for North Carolina State University's introductory course in FORTRAN computer programming, CSC112L. It is not intended for this handbook to replace the course textbook or lecture components. However, this manual was compiled to be kept by students as a reference throughout the semester for laboratory assignments, as well as future work in upcoming engineering classes that utilize FORTRAN programming.

Please feel free to report any errors to Michael Thelen, who can be contacted at

mythelen@ncsu.edu

Last corrected and updated: 2/16/07

Chapter 1

Introduction to FORTRAN

- Variables
  1. Must have a unique name
  2. Must be ≤ 31 characters
  3. Must begin with an alphabetic letter, not a number
  4. Can contain only
     - Letters
     - Digits
     - Underscores "_"
  5. Variable types
     (a) Integer
     (b) Real
     (c) Complex
        - Rarely used
     (d) Logical
        - Can only retain the value of "TRUE" or "FALSE"
        - Also rare, but use more frequently than complex numbers
     (e) Character
        - Can store character strings instead of numerical values
        - NOTE: mathematical computations *cannot* be performed on character variables–**COMMON MISTAKE**
        - Note, character variables, by default, will store characters of length *one*. However, in the real world, it is often the case that we have character strings that take up more than just a space. Thus, we need to specify to FORTRAN to allot more spaces. For example, if we wanted to store the string "TRUE" or the string "FALSE" inside a variable, call it "answer,"

notice that the longer of the two strings is "FALSE," which has 5 spaces. Then in our type declaration, we would need to say something along the lines of
`CHARACTER(5)::answer`
which will allow us to allocation memory for 5 character spaces for our variable, "answer."

6. Keeping track of and using variables
   (a) IMPLICIT NONE
      – Requires the programmer to declare each variable *explicitly*, bringing forth error messages if variables are used but not found in the type declarations (variable list at the beginning of each program).
   (b) Parameters
      – e.g. $\pi \approx 3.14159265$, $e \approx 2.71828183$
      – Variables which are unchanging
      – Values associated with parameter variables are declared once at the beginning of a program. Their initialized value will be retained throughout the entire program.
      – Example:

```
PROGRAM circle
IMPLICIT NONE
REAL, PARAMETER :: pi = 3.14159265
REAL:: radius, area, circumference

WRITE(*,*) ‘‘What is the radius of your circle?’’
READ(*,*) radius

area = pi*radius**2
circumference = 2*pi*r

WRITE(*,*) ‘‘The area of your circle is: ’’, area
WRITE(*,*) ‘‘The circumference of your circle is: ’’, circumference

END PROGRAM circle
```

- Variable Operations
  – Assignment statement
    * "="
    * "var" = "expression"
    * Examples:

```
x = 5 + 2
y = x*3
```

Assuming x and y are not modified elsewhere in the program, y will end up with a value of 21.

- Order of operations
  - PEMDAS, allowing us to:

Operation	Commonly Known Math Symbol	FORTRAN Command
Add	$+$	`+`
Subtract	$-$	`-`
Multiply	$\times$	`*`
Divide	$\div$	`/`
Exponentiate	$\widehat{\ }$	`**`

  - No two operators can be side-by-side!
    - For example, if we wanted to multiply 3 and -4, we could not say `3*-4`, but rather we would want to say `3*(-4)`

- Other fundamental commands
  - WRITE statements
  - READ statements
- Template for every program:
  1. Declaration
  2. Execution
  3. Termination
- "My First Program" (lab example–no need to submit for a grade)
  - CSC112 uses the "NEDIT" program editor. To open this, type `nedit` or `nedit&` in the shell window.
    NOTE: Adding the ampersand option will allow you to type in commands after your NEDIT window opens.

```
PROGRAM myfirstprog
  IMPLICIT NONE
  INTEGER::test

  !Execution
  WRITE(*,*)``Enter an integer''
  READ(*,*)test
  WRITE(*,*)test
```

```
    !Termination
    STOP  !an optional command to tell the computer to stop executing
END PROGRAM
```

  - Notice the comment statements "!" and also the indentation (see programming practices link on lab website).
  - To save as a FORTRAN file, use the "*.f95" suffix (e.g. myfirstprog.f95).

- To compile a FORTRAN program:
  - We use the NAGWARE compiler
  - We must add the compiler in order to use it. To do so, type:

    `add nagf95`

    in a shell window.
  - To compile a FORTRAN file, type:

    `f95 -o` "`name of executable file`" "`name of FORTRAN file`"

    This will produce an new executable, which we can run.
  - For example, to compile "My First Program":
    1. Make sure your FORTRAN file under the NEDIT program editor is saved as a FORTRAN file with the appropriate *.f95 suffix.
    2. Open a terminal shell window.
    3. At the eos prompt, type the appropriate commands. Your terminal window should look like this:

```
eos% add nagf95
eos% f95 -o myfirstprog myfirstprog.f95
eos% ./myfirstprog
```

- Debugging
  - No one is a perfect programmer.
  - Perhaps the most important focus you will have in this course is not how to code brilliantly, but how to debug.
  - Error types:

    Syntax An error that occurs during the composition of the program (e.g. you typed something in wrong).

    Run-time An error that occurs during the execution of a program, in contrast to compile-time errors, which occur while a program is being compiled. For example, running out of memory, illegal math operations (e.g. divide by zero) or reading in from a file that does not exist.

Logical Code executes without errors but produces incorrect or inaccurate output based on incorrect or inaccurate code that was composed.

Chapter 2

Selective Execution

- Logical Operators

Operator	**Command**
Equal to	`==`
Not equal to	`/=`
Greater than	`>`
Greater than or equal to	`>=`
Less than	`<`
Less than or equal to	`<=`

- Combinational Logical Operators: for any logical statements, a_1 and a_2, we have

Combinational Operator	**Command**	**TRUE, if**
Conjunction	a_1 `.AND.` a_2	both a_1 and a_2 are TRUE
Disjunction	a_1 `.OR.` a_2	either a_1 or a_2 or both are TRUE
Equivalence	a_1 `.EQV.` a_2	a_1 is the same as a_2 (either both TRUE or both FALSE)
Nonequivalence	a_1 `.NEQV.` a_2	one of a_1 and a_2 is TRUE and the other is FALSE
Negation	`.NOT.` a_1	a_1 is FALSE

- **Logic 101** The following is a truth table commonly seen in logic courses:

		Conjunction	**Disjunction**	**Equivalence**	**Nonequivalence**
p	q	$p \wedge q$	$p \vee q$	$p = q$	$p \neq q$
.TRUE.	**.TRUE.**	.TRUE.	.TRUE.	.TRUE.	.FALSE.
.TRUE.	**.FALSE.**	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.FALSE.	**.TRUE.**	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.FALSE.	**.FALSE.**	.FALSE.	.FALSE.	.TRUE.	.FALSE.

- Simple IF
  - `IF (logical_expression) executable_statement`
  - Example:
    `IF (x > 100) WRITE(*,*) ‘‘The number is greater than 100’’`
- Block IF
  - ```
    IF (logical_expression) THEN
      Executable_Statement_1
      Executable_Statement_2
      Executable_Statement_3
      .
      .
      .
    END IF
    ```
    Note: The above executable statements can be Simple IF or Block IF/THEN statements within this current Block IF. Such structure is known as a “nested IF.”
  - Example:

    ```
    IF (x > 100) THEN
      WRITE(*,*) ‘‘The number is greater than 100’’
      x = x - 1
      WRITE(*,*) ‘‘The number minus 1 is: ’’, x, ‘‘.’’
    END IF
    ```

    Note: Notice the above processes, using a running difference of “$x = x - 1$” and also the outputting of three items in the last WRITE statement, namely the period. The juxtaposition of these three items in this manner will allow the appearance of a full sentence in conventional English to be outputted to the terminal window.
- CASE Construct
  - Template:

    ```
    example: SELECT CASE(case_expr)
    CASE (case_selector_1) name1  ! name is optional
      Executable_Statement_1
      Executable_Statement_2
      .
      .
      .
    CASE (case_selector_2) name2  ! name is optional
    ```
    ```

```
  Executable_Statement_1
  Executable_Statement_2
  .
  .
  .
CASE DEFAULT name3  ! name is optional
  Executable_Statement_1
  Executable_Statement_2
  .
  .
  .
END SELECT example
```

– Example:

```
example: SELECT CASE(grade)
CASE(100) one_hundred  ! executes if grade = 100
  WRITE(*,*)‘‘You got a 100’’
CASE(90:99) A  ! executes if grade >= 90 but less than 100
  WRITE(*,*)‘‘You have an A’’
CASE(80:89) B  ! executes if grade >= 80 and <=89
  WRITE(*,*)‘‘You have a B’’
CASE(:79) work_harder  ! executes if grade <= 79
  WRITE(*,*)‘‘You need to work harder’’
CASE DEFAULT  ! executes if no other case executes
  WRITE(*,*)‘‘You can’t make over 100!!’’
END SELECT example
```

– Example of IF/ELSE construct that is equivalent to the previous CASE construct:

```
example: IF (grade == 100) THEN ! executes if grade = 100
  WRITE(*,*)‘‘You got a 100’’
ELSE IF (grade >= 90 .AND. grade <= 99) THEN ! if 90 <= grade < 100
  WRITE(*,*)‘‘You have an A’’
ELSE IF (grade >= 80 .AND. grade <= 89) THEN ! if 80 <= grade < 89
  WRITE(*,*)‘‘You have a B’’
ELSE IF (grade <= 79) THEN ! if grade <= 79
  WRITE(*,*)‘‘You need to work harder’’
ELSE  ! if no other system executes
  WRITE(*,*)‘‘You can’t make over 100!!’’
END IF example
```

Notice how CASE constructs are very nice for certain situations (looking for ranges of numbers) and IF/ELSE blocks are better suited for other situations, such as looking at several variables (not just one, e.g. “grade”). Usually one can

accomplish the same things using either construct. Pick your favorite or use the one that requires less typing. Note: Exclamation marks ("!") *within* WRITE statements will not comment out character strings.

- Programming Practices: ALWAYS USE INDENTATION!

Chapter 3

Repetitive Execution

- Two basic types of repetition:
  1. Repetition controlled by a logical expression
  2. Repetition controlled by a counter
- DO construct:

```
DO
  Executable_Statement_1
  Executable_Statement_2
  .
  .
  .
END DO
```

  The loop body contains executable statements that are executed over and over again. But when will it stop executing? If it doesn't, this is known as an **infinite loop**. NOTE: Press Ctrl+C at the terminal window when in infinite loops.

- "While" loops: keeps executing *while* some logical statement is true

```
DO
  ...  ! statement sequence 1
  IF (logical_expression) EXIT
  ...  ! statement sequence 2
END DO
```

  This code will execute the first statement sequence. When it gets to the simple IF, if the logical expression is true, it will EXIT out of the loop and the second statement sequence will never get executed. Otherwise (if the logical expression is false), both statement sequences get executed.

- Counter Controlled DO Loops
  - Counter Controllers:
    - Must be an integer
    - Usually represented by an i.
  - Template:

```
DO i = initial_value, limit, step_size
  statement_1
  statement_2
  ...
  statement_n
END DO
```

    This will execute statements 1 through n from the **initial value** (or **index**) to the **limit**, by the **step size** (or **increment**).

- Example:

```
DO i = 1, 10, 2
  statement_1
  statement_2
  ...
  statement_n
END DO
```

- Another Example:

```
DO i = 1, 10, -1
  ...
END DO
```

  Notice, that $(index \times increment) > (limit \times increment)$ because

$$index \times increment = 1 * -1 = -1$$

  and

$$limit \times increment = 10 * -1 = -10$$

  This condition will bring up errors when you go to compile your code.

- Final Example:

```
DO i = 1, 10
  ...
END DO
```

The default increment = 1.

- Never modify an index within the DO loop code. This can cause some bad results and even an infinite loop.
- Cycle Statement:
  - Example:

```
DO i = 100, 1000, 2
  ...
  IF (i >= 600 .AND. i < 700) CYCLE
  ...
END DO
```

  - If you did not want to use any numbers within the range of 600 and 699 you can implement the CYCLE statement.
  - Cycle statements can be used within more complex IF/THEN constructs.
- EXIT statement
  - Example:

```
DO
  ...
  IF (i > 233) EXIT
  ...
END DO
```

  - If you wanted to EXIT the loop if a certain condition (controlling a *logical expression*) was true.
  - In this case, if the variable i was greater than 233.
- Nested Loops:
  - Example:

```
DO i = 1, m
  ...  ! some code here, if you want
  DO j = 1, n
    ...  ! more code here, if you want
  END DO
  ... ! more code here, if you want
END DO
```

  - Just like nested IF/THEN constructs.

- Named loops:

```
outer: DO i = 1, 100, 1
  ...
  inner: DO
    ...
    IF (test <= 0) EXIT
    ...
  END DO inner
  ...
END DO outer
```

- Programming Practices: ALWAYS USE INDENTATION!

Chapter 4

Formatting

- For this course, we won't modify input, only output (doesn't make sense to modify input only to modify it again when outputting). So we'll only really need WRITE or PRINT statements to perform our formatting, since we can format numbers and character strings which are found either in variables or explicitly written (written in a WRITE or PRINT statement, inside quotation marks). We won't need to use READ statements for formatting with this lab.

- Print Statement:

  - `PRINT format_specifier, output_list`

  - This command is a little more primitive than the WRITE statement, but the two can essentially be used interchangeably.

- How do we use the WRITE statement?

  - `WRITE(*,*) output_list`

  - Ever wonder what the two asterisks are for?

    * The first asterisk will be addressed next lab.

    * The second asterisk is to put in a format specifier.

- Numerical format specifiers:

Type:	Statement:	Description:	First x:	Second x:
Integer	`Ix` or `Ix.x`	Integer	# of columns the user wants the integer to have (1 character per column)	forces the number of digits shown
Real	`Fx.x`	Real	# of columns (decimal point takes up one column!)	# of columns past decimal point
Real	`Ex.x`	Exponential Notation (normalized from 0.1 to 10)	# of columns (including exponential notation!)	# of columns past decimal point
Scientific	`ESx.x`	Standard Scientific Notation (normalized from 1 to 10)	(same as exponential)	(same as exponential)

- Non-Numerical format specifiers:

Type:	Statement:	Description:	x
Logical	`Lx`	Logical	column justification (which column to place output)
Character	`Ax`	Character	column justification (which column to *begin* output)

- Examples:

1. "3687"

```
      123456 <-- COLUMN NUMBER
  I4: 3687   <-- Exactly enough space to fit 4 digits
  I3: ***    <-- Not enough space to fit 4 digits in 3 columns
  I6:   3687 <-- Right justified in 6 columns
I6.6: 003687 <-- Inserts zeros in front of integer
                 (useful for time display--6 hours, 5 minutes = 6:05)
```

2. "322.6587"

```
      12345678 <-- COLUMN NUMBER
F8.4: 322.6587 <-- Exactly enough space to fit the number
F8.2:   322.66 <-- Rounds to 2 decimal places and right justifies
F6.4: ******   <-- 6 asterisks to indicate not enough space to fit
                   with 4 decimals places

       1234567890123 <-- COLUMN NUMBER
E13.7: 0.3226587E+02 <-- Exactly enough space to fit the number
```

```
E10.4: 0.3227E+03    <-- Four significant figures (rounded)
E13.4:    0.3227E+03 <-- Right justified
E10.7: **********    <-- Not enough space to fit

       123456789012 <-- COLUMN NUMBER
E12.6: 3.226587E+02 <-- Exactly enough space to fit the number
 E9.3: 3.227E+02    <-- Four significant figures (rounded)
E12.3:    3.227E+02 <-- Right justified
 E9.6: *********    <-- Not enough space to fit
```

3. Logical Variables (Recall: they can either hold a value of .TRUE. or .FALSE. and output as T or F)

```
    12345678 <-- COLUMN NUMBER
L5:     T    <-- If example is true (center justified)
L1: F        <-- If example is false (left justified)
L8:        F <-- If example if false (right justified)
```

4. Character String Example: "FORTRAN is Cool"

```
     12345678901234567890 <-- COLUMN NUMBER
  A: FORTRAN is Cool      <-- Printed as is
 A6: FORTRA               <-- Truncated variable (not enough columns,
                              but NO asterisks here like with
                              numerical formatting truncation)
A20:      FORTRAN is Cool <-- Right justified (more columns than needed)
```

- Another Example: Consider the command:
  `WRITE(*,?)  ‘‘The total number of groceries is ’’, groceries, &`
  `& ‘‘ and the cost is $’’, cost, ‘‘.’’`
  - First, notice the spacing and the use of the ampersand.
  - Second, what command can we put in the question mark ("?") above?
    1. Place format specifiers in a list inside the WRITE statement:
       `WRITE(*,‘(A, I2, A, F5.2, A)’)...`
       Notice the order of the format descriptors. The first item in the WRITE statement is a character and the first format descriptor in the format statement is an 'A'.
    2. Use a FORMAT statement:
       `some_label FORMAT(descriptor_list)`
       So in code, it would look something like:

```
100 FORMAT(A, I2, A F5.2, A)
WRITE(*, 100)...
```

This is similar to the first one, except that the format number is inserted into the format location. Then, a FORMAT statement is written with the format number in front of it. This FORMAT statement can be written anywhere in the code and can be used in multiple WRITE or PRINT statements.

3. Define a character formatter that can be used as necessary:

```
CHARACTER:: formatter
formatter = '(A, I2, A, F5.2, A)'
WRITE(*,formatter)...
```

A character variable can store a formatter as well.

- Additional Formatting Techniques:
  - **Spacing:** The x descriptor. In a format statement, you would type the number of spaces you wanted directly in front of an 'x' as follows:
    `110 FORMAT(I4, 3x, F5.1)`
    The '$3x$' puts 3 spaces between the integer and the real number when printed.
  - **Tabbing:** use the T descriptor as follows:
    `111 FORMAT(I4, T10, F5.1)`
    The '$T10$' will put the real number starting in the 10^{th} column of the screen. Be careful with this one because if the 10^{th} column is already used, the computer will rewrite over the 10^{th} column with what immediate follows the tab descriptor.
  - **Repeating Descriptors:** It is also possible to repeat a single or a group of format descriptors without typing the descriptor over and over. Any single descriptor can be duplicated by typing the number of times you want to duplicate the descriptor directly in front of the descriptor. For example:
    `112 FORMAT(4I4, T10, 3F5.1)`
    This would print out 4 integers then tab to column 10 and then print 3 real numbers (remember that there must be that many numbers to be printed–i.e. 4 separate integers followed by 3 separate real numbers). A group of descriptors can be repeated as follows:
    `113 FORMAT(I4, 5(3x, F5.1))`
    The descriptors inside of the inner parenthesis will be repeated 5 times because there is a 5 in front of that group of descriptors.
  - **Changing Output Lines:** Another useful tool is the ability to skip to the next line in the middle of a format statement. The slash ("/") descriptor is used to do that. This is one descriptor that can be used without a comma separator between other descriptors. For example:
    ```
    114 FORMAT(I4, /3x, F5.1) !  the slash skips to the next line
    115 FORMAT(I4, 3x, F5.1///, 3I5) !  3 slashes skips 3 lines down
    ```
  - **ADVANCE = "NO":** You may have noticed that before the cursor waits on the next line for the response to be read. `ADVANCE = ''NO''` allows you to read

from the same line that the input was asked for on. For example, perhaps you wanted to ask the user to input his/her age. Here are two ways that it could be done:

```
WRITE(*,*) ‘‘Enter your age:’’
```

which will produce the following output in the terminal window:

```
Enter your age:
_
```

with the cursor waiting where the underscore ("_") is, underneath the output from the WRITE statement. The second way, in your code we'll have:

```
WRITE(*,‘(A)’, ADVANCE = ‘‘NO’’) ‘‘Enter your age:  ’’
```

which will produce the output:

```
Enter your age:  _
```

and the cursor waits on the same line, right beside the WRITE statement's character string.

Note: Notice the space after the colon in the code and the space that is translated onto the output. With the `ADVANCE = ‘‘NO’’` option, spacing becomes more important!

Chapter 5

File Processing

- This lab, we'll figure out what the first asterisk is for inside WRITE(*,*) and READ(*,*) statements.
- File processing involves reading in and writing out to files. We'll be using and creating data files with, but not limited to, the suffixes *.dat or *.txt.
- When opening and reading files, we have two basic types of access:

  Sequential Access lines are read in one at a time, or sequentially

  Direct Access you tell FORTRAN which record you want to retrieve

  We'll be using a little of both.
- General File Processing Command: `OPEN(open_list)`
- What can we put in the 'open list'?

  1. `UNIT = statement`
     Indicates the input/output unit number to associate with the file (non-negative integer value, usually greater than 10, but less than 100–to not inhibit other FORTRAN processes that use numbers 10 and below and also to not get in the way of our FORMAT statements, which use integers 100 and above).
  2. `FILE = statement`
     Indicates filename of file to be opened (character value).
  3. `STATUS = statement`
     Specifies the status of the file to be opened (character value of one of the following: `'OLD'`,`'NEW'`, `'REPLACE'`, `'SCRATCH'` or `'UNKNOWN'`).
  4. `ACTION = statement`
     Specifies whether a file is to be opened for reading, writing, or both (character value of one of the following: `'READ'`, `'WRITE'` or `'READWRITE'`).

5. `POSITION = statement`
   Specifies the position of the file to be used (character value of one of the following:

`'REWIND'`	position the file at its initial point
`'APPEND'`	position the file at the end of the file
`'ASIS'`	leave the position of the file unchanged (default)

6. `IOSTAT = statement`
   Specifies the name of an integer variable in which the status of the open operation can be returned.

$= 0$	You're in good shape.
> 0	Not good; system error will be returned, possibly due to a missing file or corrupt data.
< 0	Not quite so bad, but the best scenario is if IOSTAT equals 0.

   Some examples:

   - General example:
     Let "`ierror`" be a declared integer variable. In the code, we can have:
     ```
     OPEN(UNIT = 17, FILE = ''data.dat'', STATUS = ''OLD'',&
     & ACTION = ''READ'', IOSTAT = ierror)
     ```
   - Example of writing out to a file:
     ```
     OPEN(UNIT = 13, FILE = ''output.dat'', STATUS = ''NEW'', &
     & ACTION = ''WRITE'', IOSTAT = ierror)
     ```
     * If the file "output.dat" already exists, IOSTAT will not equal zero, causing ierror to not equal zero. Using this, we can produce an error message:
       `IF (ierror /= 0) WRITE(*,*) ''Error:  File Creation Failure''`
       or
       `IF (ierror /=0) STOP 'Error creating file'`
       Notice here how the STOP command has a built in ability to output a character string if we put the desired error message to be outputted within quotations (like a miniature WRITE statement).
     * A `STATUS = 'REPLACE'` statement can be used, which will cause the computer to overwrite the current file.
   - `STATUS = ''SCRATCH''`:
     * No file name can be specified.
     * Creates a temporary (scratch) file that can hold information during the program's execution.
     * Will be deleted when the program has terminated.

7. Closing Files
   - CLOSE statement: `CLOSE(unit_var)`, where the unit variable is the integer variable used to attribute or identify the input/output statement.
   - You must close the file(s) you've opened.

8. How do we put it all together? Some examples:

   (a) Assume the file "michael.dat" has three columns of data and we want to read it in as x, y, and z from the first line of the input file:

   ```
   OPEN(UNIT = 17, FILE = ‘‘michael.dat’’, STATUS = ‘‘OLD’’, &
   & IOSTAT = ierror)
   READ(17,*) x, y, z  ! 17 is the unit number of the file
   ```

   (b) Now, let's write out the data we used from 'michael.dat" out to another file (call it "michout.dat") and incorporate some formatting.

   ```
   OPEN(UNIT = 18, FILE = ‘‘michout.dat’’, STATUS = ‘‘REPLACE’’, &
   & IOSTAT = ierror)
   100 FORMAT(‘ X = ’, F10.2, ‘ Y = ’, F10.2, ‘ Z = ’, F10.2)
   WRITE(18,100) x, y, z
   ```

   Comment: Formatting can automate some things to make your life much easier.

   (c) Sometimes we don't know how long a file is and when to stop reading it. Thus, we can incorporate the IOSTAT tool to help us.

   ```
   ...  ! preliminary code

   INTEGER:: ierror, eof
   REAL:: a

   ...  ! more boring code here

   ! let’s open the file
   OPEN(UNIT = 14, FILE = ‘‘test.dat’’, STATUS = ‘‘OLD’’, &
   & IOSTAT = ierror)
   IF(ierror /= 0) STOP  ! if the data is corrupt or we get to
                         ! the end of the file, then stop compiling

   ...  ! more code

   ! let’s output data into another file, ‘‘testout.dat,’’ as long
   ! as the initial file, ‘‘test.dat,’’ is good and reading out
   ! (i.e. ierror /= 0)
   OPEN(UNIT = 15, FILE = ‘‘testout.dat’’, STATUS = ‘‘REPLACE’’, &
   & IOSTAT = ierror)
   IF(ierror/=0) STOP

   DO
     READ(14, *, IOSTAT = eof) a
     IF(eof /= 0) EXIT
   ```

```
  WRITE(15, *) a
END DO
```

Comments: When the end of a file is reached, the variable "eof" will be less than zero. If there is corrupt data, "eof" will be greater than zero.

9. REWIND statement: `REWIND(unit_var)`
   - Used to rewind the file back to the beginning.
   - Nice if you don't want to go through the trouble of closing and reopening a file, but want to reuse previous values.
10. BACKSPACE statement:
   - Used to rewind back one record to reuse the previous values. (Note: Useful with DO loops! You can keep backspacing using a loop until you get the record you want. You can control this with some sort of logical statement or counter.)

Chapter 6

Arrays

- An array is basically a matrix, which we can think of in terms of dimension or (# of rows) $\times$ (# of columns).

- Matrices, and arrays, can be one-dimensional or multi-dimensional. Matrix operations may be emphasized in this semester's project, and have been slightly emphasized in previous semesters, so let's have a quick review:

  - **One-dimensional:**
    Column Matrix:

$$a = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$$

    which is an $n \times 1$ vector matrix. Row Matrix:

$$a^T = \begin{pmatrix} a_1 & a_2 & \dots & a_n \end{pmatrix}$$

    Denote a vector b, where

$$b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

    Then, since a^T is a $1 \times n$ matrix and b is an $n \times 1$ matrix, then their inner dimensions are the same. This allows us to calculate their product, $a^T b$, which will yield a matrix with dimensions equal to the outer dimension of the product

$a^T b$, which will be 1×1, or a scalar. So we'll get:

$$\begin{aligned} a^T b &= \begin{pmatrix} a_1 & a_2 & \dots & a_n \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \\ &= a_1 b_1 + a_2 b_2 + ... + a_n b_n \\ &= \sum_{i=1}^{n} a_i b_i \end{aligned}$$

– **Two-dimensional:**

$$\begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \dots & c_{mn} \end{pmatrix}$$

- Arrays don't have to be numbers. We can also have character or logical arrays types.
- One-Dimensional Arrays in FORTRAN:
  - Can be either the # of rows or the # of columns, however you choose to think about it.
    Note: FORTRAN uses the first declared dimension number as the number of rows. So if you're thinking in terms of columns, know that it might not print out the way you want. If you're not printing or outputting, then it doesn't matter.
  - Can be declared as follows, for an array that will hold 500 real numbers using **static memory allocation** (as opposed to **dynamic memory allocation**, which we'll discuss later):
    `REAL, DIMENSION(500):: array`
    or
    `REAL:: array(500)`
  - Using arrays: We can write out to the 10th position of an array, call it "mike," like so, by using a WRITE statement:
    `WRITE(*,*)mike(10)`
    We can also assign value inside a matrix like so:
    `mike(12) = num1`
    This will take the value in the variable "num1" and assign it to the 12th position of the array "mike."
    Note: Positions 10 and 12 must exist for this to be error-free.

– The position of an array begins with subscript 1 in FORTRAN, not 0. However, the 0th subscript can be used, if desired. For example, if for some reason we had the following vector:

$$\begin{pmatrix} a_{-15} & a_{-14} & \dots & a_{-1} & a_0 & a_1 & \dots & a_{14} & a_{15} \end{pmatrix}$$

and lets say it was full of angle measurements, we can force FORTRAN to use negative subscripts when we declare arrays, like so:
`INTEGER:: angles(-15:15)`
and we can write out to the -15th position like so:
`WRITE(*,*) angles(-15)`

– Another example:
Let's declare two arrays, "array1" and "array2" and two arrays that will store some sums, call them "sum_array1" and"sum_array2." We can do things like this:

```
INTEGER:: array1(4) = (/3, 4, 5, 6/)  ! array constructor
INTEGER:: array2(4) = (/1, 2, 7, 8/)  ! array constructor
INTEGER:: i, sum_array1(4), sum_array2(4)

DO i =1, 4
  sum_array1(i) = array1(i) + array2(i)
END DO

sum_array2 = array1 + array2
```

In this situation, sum_array1 and sum_array2 would equal each other, because they are essentially performing the same process. Also, notice how we can initialize arrays using the array constructor, and we can use a DO loop to reference and store values into particular position in arrays. Using sum_array2, we can add two arrays, which will add element-by-element of the two initial arrays and store them into the corresponding spot of the resulting sum_array2.

– Using subset arrays (emphasized on previous end-of-year projects):

```
REAL:: array(12), sumarray(6)
INTEGER:: i

DO i = 1, 12
  array(i) = i*0.5
END DO

sumarray = array(1:6) + array(7:12)
```

This will add up the 1st half of the array with the 2nd half, like this:
The whole array:

$$\left(\begin{array}{cccc} a_1 & a_2 & \ldots & a_{12} \end{array} \right) = \left(\begin{array}{cccccc} a_1 & a_2 & \ldots & a_5 & a_6 & | \quad a_7 \quad a_8 \quad \ldots \quad a_{12} \end{array} \right)$$

with the two halves split up. Then the sum array will be:

$$\left[\begin{array}{cccc} a_1 & a_2 & \ldots a_6 \end{array} \right] + \left[\begin{array}{cccc} a_7 & a_8 & \ldots & a_{12} \end{array} \right] = \left[\begin{array}{ccc} (a_1 + a_7) & (a_2 + a_8) & \ldots (a_6 + a_{12}) \end{array} \right]$$

- Two-Dimensional Arrays in FORTRAN:
  - Declaration involves using the same **static memory allocation** idea: `array_name(rows, columns)`
    Note: Rows and columns are interchangeable depending on how you think about it, just as before, but things might not print out correctly if the array is declared incorrectly.
  - Example: `INTEGER:: number(2, 5)` for an integer array "number" with 2 rows and 5 columns, yielding to us 10 spaces we can put in integer numbers
  - We can store values into two-dimensional arrays as follows:
    1. Element by element: For example,
    ```
    number(1, 1) = 1*1
    number(1, 2) = 1*2
    ...
    number(2, 1) = 2*1
    ...
    number(2, 5) = 2*5
    ```
    2. With nested DO loops:
    ```
    DO i = 1, 5
      DO j = 1, 2
        number(i, j) = i*j
      END DO
    END DO
    ```
    3. Using an array constructor, just as with one-dimensional arrays. (Note: Array constructors cannot be used to fill arrays with more than 2 dimensions.):
    `number = RESHAPE((/1, 2, 2, 4, 5, 6, 4, 8, 5, 10/), (/2, 5/))`
    Note: The RESHAPE command is used because the array (/1, 2, 2, 4, 3, 6, 4, 8, 5, 10/) is a one-dimensional array and must be reshaped into a 2 dimensional array. The array is filled by columns so that the first 2 numbers will fill the first column of the 2×5 array. In essence, we are doing this:

$$\left[\begin{array}{cccccccccc} 1 & 2 & 2 & 4 & 5 & 6 & 4 & 8 & 5 & 10 \end{array} \right] \longrightarrow \left[\begin{array}{ccccc} 1 & 2 & 2 & 4 & 5 \\ 6 & 4 & 8 & 5 & 10 \end{array} \right]$$

- To see how arrays can be handled and manipulated computationally, see *Intrinsic Functions* under section 7.2 in the next chapter.
- Allocatable Arrays:
  - Uses **dynamic memory allocation** (not static, like before): can declare dimensions later in the program and not in the type declarations
  - Declaration as follows:
    ```
    REAL, ALLOCATABLE, DIMENSION(:,:):: array
    ```
    or
    ```
    REAL, ALLOCATABLE:: array(:,:)
    ```
  - Allocation comes within the program but is highly recommended to use a status variable (declared as an integer) to keep track of the status of the matrix, like so:
    ```
    INTEGER:: status, rows, columns
    ... ! boring code where you figure out # of rows/columns
    ALLOCATE(array(rows,columns), STAT = status)
    IF(status /= 0) STOP
    ```
    STAT will return 0 to the integer variable "status" if the allocation is successful (if there's enough memory on the computer).
  - Thus, deallocation can occur so we don't waste memory and the memory can be reused, like so:
    ```
    DEALLOCATE(array, STAT = status)
    IF(status /= 0) STOP
    ```
  - It is highly recommended to use STAT variables to keep track of the status of arrays during allocation and allocation.

Chapter 7

Procedures

- Procedures are subprograms, which hold subalgorithms.
- Why use procedures?
  1. Can be tested independently of the main program.
  2. Can be reused within a program without copying the code.
  3. Can be isolated from the main program to avoid confusion of variables within the program.

7.1 Subroutines

- Template: The template for subroutines is the same for a main program in FORTRAN (recall program template from chapter 1).

```
SUBROUTINE subroutine_name (argument_list)  ! <-- this is the heading
  IMPLICIT NONE
  ...  ! declaration section
  ...  ! execution section
  ...
  RETURN  ! RETURN command optional; only necessary if you need to
          ! ‘‘return’’ back the main program before the end of your
          ! subroutine
END SUBROUTINE subroutine_name  ! termination section
```

  Note: The “argument list” is a list of dummy variables. These variables are carried from the main program to the subroutine, where they can be referenced and used.

- Subroutines are independent. Thus, usually with some minor modifications, they can be compiled separate from the main program for testing purposes.

- We can use a subroutine within a program by using a CALL statement:
  `CALL subroutine_name(argument_list)`
  The argument list here is, again, the list of dummy variables. These variables *can* be called different names in the main program than they are in the subroutine. However, each distinct variable, regardless of name, *must* be in the same respective spots in each argument list!

- A subroutine that calls itself is known as a **recursive subroutine** (see chapter 9).

- Variable Types:

  1. Dummy: placeholders, which will be passed from the calling program when the subroutine is called; stratified into incoming, outgoing, or (both) incoming/outgoing, as indicated with the INTENT specifier (telling the subprogram how to transfer information)
     (a) Incoming: when variables come from the calling program, but will not be changed within the subroutine, like so:
     `REAL, INTENT(IN):: ... ! var list`
     (b) Outgoing: when arguments are only sent out of the subroutine and cannot be received from the calling program, like so:
     `REAL, INTENT(OUT):: ... !var list`
     (c) Incoming/Outgoing: when arguments come from the calling program and will be modified within the subroutine and returned to the calling program, like so:
     `REAL, INTENT(INOUT):: ... !var list`
  2. Local: used only within the subroutine; cannot be referenced outside of the subroutine

7.2 Functions

- Results in a single number, logical variable, character string, or array.

- A function that calls itself is called a **recursive function** (see chapter 9).

- All functions are either intrinsic or user-defined. We will concentrate mostly on writing our own functions, which will then be "user-defined." As for **intrinsic functions**, these are functions that area already built into the FORTRAN language (e.g. SIN(argument), COS(argument), MOD(argument), and ABS(argument)). There are three basic types of intrinsic functions, many of which are used for matrix operations:

  1. **Elemental Intrinsic Functions:** These are functions that are specified for scalar arguments, but also may be applied to array arguments. If the argument of an elemental function is a scalar, then the result is a scalar. If the argument is

an array, then the result is an array of the same shape as the input array. For example, in the following code, we have two equivalent, but comparative pieces of code: the single SIN function statement and the DO loop utilizing the SIN function but calculating each angle individually.

```
REAL, DIMENSION(4):: x = (/0., 3.141592, 1., 2./)
REAL, DIMENSION(4):: y
INTEGER::i

y = SIN(x)  ! compute the sine of all angles in the entire array,
              all at one time

DO i = 1, 4
  y(i) = SIN(x(i))  ! compute element by element
END DO
```

Most FORTRAN intrinsic functions that accept scalar arguments are elemental and so can be used with arrays, including, but not limited to: ABS, SIN, COS, TAN, EXP, LOG, LOG10, MOD, and SQRT.

2. **Inquiry Intrinsic Functions:** These are functions whose value depends on the properties of an object being investigated. For example, the function `UBOUND(array)` is an inquiry function that returns the element of greatest magnitude of "array."
3. **Transformational Intrinsic Functions:** These are functions that have one or more array arguments or an array-valued result. These operate on arrays as whole arrays, rather than element-by-element, and may produce arrays of other sizes and shapes than that of the original input array. An example of this type of intrinsic function is the `DOT_PRODUCT`, which has two array (vector) inputs of the same size and produces a scalar output.

- User-Defined: user writes for their own purposes within their software. A template:

```
REAL FUNCTION function_name (argument_list)
  IMPLICIT NONE
  ...  ! declaration section
  ...  ! execution section
  ...
  function_name = some_expression
  RETURN
END FUNCTION function_name
```

- Functions must be named.

7.3 Subroutines vs. Functions

- Functions are designed to return a single value to the program unit that has referenced the function. Subroutines often return more than one value, or they may return no values (e.g. banner subroutines), but they do perform some sort of task for the user.

- An interesting example:

```
PROGRAM test
  REAL, EXTERNAL:: func_1, func_2
  REAL:: x, y, output
  ...
  CALL evaluate(func_1, x, y, output)
  CALL evaluate(func_2, x, y, output)
  ...
END PROGRAM

!--------------------------------------
SUBROUTINE evaluate (func, a, b, result)
  REAL, EXTERNAL:: func
  REAL, INTENT(IN):: a, b
  REAL, INTENT(OUT):: result
  result = b*func(a)
END SUBROUTINE evaluate
```

  Notice a few very key things with the above program set. "func_1" and "func_2" are two separate user-defined functions (assumed to be present in the above program set, but are obviously not declared there). We can point to these two exterior functions from the subroutine by using an EXTERNAL statement. This idea is heavily emphasized in an upcoming outlab (hint, hint).

- The INTENT command is used at times with functions, also, but is more common with subroutines.

Chapter 8

Modules

- By definition, **modules** are used to share data (which is saved using a SAVE statement) throughout other FORTRAN programs, subroutines, and functions by putting declarations, subprograms, and definitions of new data types into a "package" or "library" that can be used in any other program unit
- Modules are *separately* compiled program units.
- USE statement:
  - Looks like this: `USE module_name`
  - Implies that the data values declared (or saved using a SAVE statement) in the module may be used in the main program unit.
  - Must appear before any other statements in a program, *but* must follow PROGRAM, SUBROUTINE, and/or FUNCTION heading statements.
  - Rough template (Note: The first part is the module file and the second part is the program file. There are TWO, SEPARATE files.):

```
! save this file as module_name.f95
MODULE module_name
  IMPLICIT NONE
  SAVE  ! this command is optional
  ...   ! saved data declaration section; optional
  ...   ! module body (further declarations, and execution)
END MODULE module_name
```

    and then we would have a separate file with:

```
! save this file as program_name.f95 (THIS IS A SEPARATE FILE!!!)
PROGRAM example
  USE module_name  ! only include this command if you need the module
  IMPLICIT NONE
```

```
  ...  ! declarations, execution
END PROGRAM

SUBROUTINE example1(arg_list)
  USE module_name  ! only include this command if you need the module
  IMPLICIT NONE
  ...  ! declarations, execution
END SUBROUTINE example1

FUNCTION example2(arg_list)
  USE module_name
  IMPLICIT NONE  ! only include this command if you need the module
  ...  ! declarations, execution
END FUNCTION example2
```

- SAVE statement: an optional, but useful, command that allows data values declared in the module to be preserved between references in different procedures
- External Module Procedures (or subprograms):
  - Must come after all data declarations in the module.
  - Preceded by a CONTAINS statement.
  - CONTAINS block:
    - General template (for n subprograms):

```
CONTAINS
  subprogram_1
    ...
  subprogram_2
    ...
  .
  .
  .
  subprogram_n
    ...
```

  - Example:

```
MODULE module_name
  IMPLICIT NONE
  SAVE
  ...  ! shared data declaration section

  CONTAINS
    SUBROUTINE example (arg_list)
      IMPLICIT NONE
```

```
    ...
  END SUBROUTINE example
END MODULE module_name
```

- Interfacing: Interfaces can check to see if each reference to each program or subprogram is correct (i.e. correct number and types of variables)
  - Like a large, global IMPLICIT NONE statement.
  - Explicit Interface: automated when employing a module
  - Implicit Interface: can be used for external programs; compiler may not always be able to check whether the references are correct
  - Interfaces can be done two different ways:
    1. Place procedures within a module. When the module is compiled, the module will have information about every argument in the procedures. This will then be an *explicit* interface.
    2. Interface block. Here's a template:

       ```
       INTERFACE
         ...  ! Interface Body (type declarations WITH IMPLICIT NONE!!!
       END INTERFACE
       ```

       An example:

       ```
       INTERFACE

         FUNCTION mich(x)
           IMPLICIT NONE
           REAL:: mich
           REAL, INTENT(IN):: x
         END FUNCTION mich

         SUBROUTINE mike(y)
           IMPLICIT NONE
           ...  ! subroutine type declarations...
         END SUBROUTINE mike

       END INTERFACE
       ```

- Compiling: Module compilation can be a tedious process. It is important to be meticulous and make sure that your commands are correct.

  1. At a shell window, type in the following command:

     ```
     f95 -c module_name.f95
     ```

(If you're compiling your programs using remote access, this first command can be interchanged with "`f95 module_name.f95`" In the past, I have had numerous problems compiling modules remotely, but I believe the latter command is the most stable if you need to compile a module using remote access.)

2. Once your module file is compiled, we must compile our program file *along* with our module file. So, use this command:

```
f95 -o prog_name prog_name.f95 module_name.f95
```

3. These commands will make an executable file called "prog_name" (hopefully everyone is familiar with executable files by now).

4. Then you can say

```
./prog_name
```

and your program will start.

Chapter 9

Recursion

- A recursive function in mathematics is defined to consist of the following:

  1. an anchor (or base) case: the value of the function, specified for one or more values of the argument(s)
  2. an inductive (or recursive) step: the function's value for the current value of the argument(s); defined in terms of previously defined function values and/or argument values

  In other words, we define some anchor case and then inductive steps. By having both components, we can calculate recursive values at essentially any iteration.

- Powers–classic example of recursion: Let's say we want to know what 3^5 equals. Recursively, we have:
  $3^0 = 1$ (anchor; we define this–in fact, we define any number to the power 0 to equal 1)
  $3^1 = 3$ (inductive step)
  $3^2 = 3 \times 3 = 9$
  $3^3 = 3 \times 3 \times 3 = 27$
  $3^4 = 3 \times 3 \times 3 \times 3 = 81$
  $3^5 = 3 \times 3 \times 3 \times 3 \times 3 = 243$, our answer.

- Recursive Procedures (in software): procedures that "call" themselves.

  – Often very difficult to visualize.
  – Very similar to a DO loop. In fact, many recursive procedures can be coded in a DO loop (it is recommended in some situations to code first in an ordinary loop to get a feel for the repetitive processes, before coding recursively). Also, as with DO loops, some sort of EXIT condition will usually be necessary, so the procedure will stop calling itself. Infinite loops are also common here in recursion, as they were with repetitive execution (back in chapter 3).

- Factorials–another classic example.

In English	In Programming Language
Define $0! = 1$ (anchor)	IF (N == 0) N! = 1
So, $n! = n * (n-1)!$, for $n > 0$ (induction)	IF (N > 0) N! = N*(N-1)!

Using these definitions, the computer undergoes several processes in order to calculate a factorial. Let's say we want to know the value of 5! If we run a program that computes factorials using a recursive function (call it "Fact(x)"), and we wish to program according to the above definitions, the question becomes: what process does the computer undergo after the user presses 5 and the black box begins its magic? What does the computer want to know or what does it ask itself to be able calculate 5!? Essentially, the computer works through the following:

Iteration (n)	Inductive Step	Answer
5	$= n * \text{Fact}(n-1) = 5 * \text{Fact}(4)$	120
4	$= n * \text{Fact}(n-1) = 4 * \text{Fact}(3)$	24
3	$= n * \text{Fact}(n-1) = 3 * \text{Fact}(2)$	6
2	$= n * \text{Fact}(n-1) = 2 * \text{Fact}(1)$	2
1	$= n * \text{Fact}(n-1) = 1 * \text{Fact}(0)$	1
0	(none, $0! \stackrel{\text{def}}{=} 1$)	1

At first, our computer doesn't know what 5! is, because it doesn't have this value memorized. However, it works its way down, each time needing a factorial value of a smaller number. For example, the next iteration, the computer finds itself needing to know 4! in order to calculate 5!, 3! in order to calculate 4!, etc. The computer will make its way down until it finds something that it knows the value of, the anchor, where we defined $0! = 1$. Then, it can calculate 1!, since it has the value for 0! now. Then, in turn, comes 2!, then 3!, 4!, and finally 5!, which equals 120. This is a recursive process. How could we define this "factorial" process using a recursive subroutine? Take a look:

```
RECURSIVE SUBROUTINE fact(n, result)
  IMPLICIT NONE

  INTEGER, INTENT(IN):: n
  INTEGER, INTENT(OUT):: result
  INTEGER:: temp

  IF (n > 0) THEN
    CALL fact(n - 1, temp)  ! recursion by using a CALL statement
    result = n * temp
  ELSE
    result = 1  ! exit condition, or anchor
  END IF
END SUBROUTINE fact
```

- Recursive Functions:
  - Must have a recursive step and exit step, but, the stored value returned to the calling program confuses the recursive function because the function doesn't know whether or not to use the value stored in function name or call the function again
    - **Solution** $\Longrightarrow$ RESULT statement: declares what variable the function answer will be stored in other than the function name so the function name can be reused
  - Let's check out the factorial example again:

```
RECURSIVE INTEGER FUNCTION fact(n) RESULT(answer)
  IMPLICIT NONE

  INTEGER, INTENT(IN):: n

  IF (n > 0) THEN
    answer = n*fact(n-1)
  ELSE
    answer = 1
  END IF
END FUNCTION fact
```

    Note: The variable "answer" does not have to be declared because it is already declared as an integer, since the function will have been declared as an integer function in the main program.