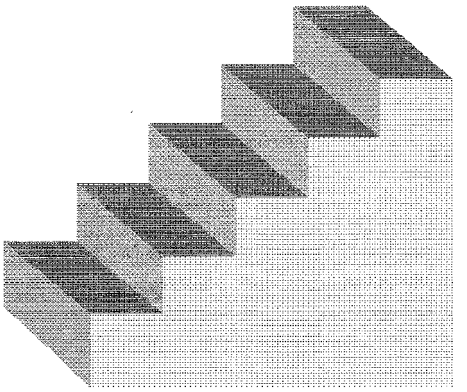

PROGRAMMING IN FORTRAN

Fourth Edition

Ömer Akgiray

PERMISSION TO COPY AND DISTRIBUTE:

e-mail:

omer.akgiray@marmara.edu.tr

CHAPTER 7: *FORTRAN 90*

7.1 General Remarks

As noted in Chapter 1, Fortran 90 contains all of FORTRAN 77. Any standard FORTRAN 77 program or subprogram is therefore a valid Fortran 90 program or subprogram, and should behave in an identical manner. Thus the large number of existing FORTRAN 77 programs can continue to be utilized for as long as necessary without the need for modification. Ellis et al. note that "it is precisely this care for the protection of existing investment that explains why FORTRAN, which is the oldest of all current programming languages, is still by far the most widely used language for scientific programming"[1]. As a matter of fact, a great advantage of using an established language like FORTRAN is the wealth of existing software upon which programmers can draw[2]. Such software is normally in the form of libraries of subroutines and functions[3]. The backward compatibility of Fortran 90 means that these FORTRAN 77 libraries can be used directly by Fortran 90 programs.

7.2 Fortran 90: Introducing a New Style

Although many programmers probably continue to write Fortran 90 programs in a style not too far removed from that permitted by FORTRAN 77, the new standard introduces several new features that lead to a new style which is worth learning and adopting. In this section we shall learn many of these new features of Fortran 90.

The Fortran 90 Character Set

Programs in the Fortran 90 language are written using the 58 characters (26 alphabetic characters, 10 digits, and 22 symbols) taken from the **Fortran 90 Character Set**:

Upper case alphabetic characters:	A to Z
Lower case alphabetic characters:	a to z
Digits:	0 to 9
Symbols:	□ + - * / . , ' = $ ( ) : ! " % & ; < > ? _

[1] Ellis et al. (1994).
[2] Smith (1995).
[3] See, for example, *Numerical Recipes* by Press et al. (1986).

where □ represents the space, or blank, character. Note that the last 9 symbols listed above were not in the FORTRAN 77 Character Set. Note also that lower case letters are treated as identical to upper case letters except, of course, when they appear within character strings. Thus, for example, `READ` and `read` are considered identical whereas the character constants `'string'` and `'STRING'` are not.

Names in Fortran 90

The rules that must be followed in the formation of Fortran 90 names are as follows:

- A name may contain up 31 characters.
- A name must begin with a letter, either upper case or lower case.
- A name may contain the letters (`A` to `Z` and `a` to `z`), the digits (`0` to `9`), and the underscore character (_).

Free Form of Source Files

Fortran 90 allows the use of a **free form** in typing programs. In the free form, statements can be written anywhere on the line. The rules are as follows:

1. A statement can be typed starting anywhere on a line.
2. A line containing an exclamation mark `!` as the first non-blank character is a comment line (see the next subsection for more details).
3. A line may contain more than one statement. A semicolon `;` must be used to separate successive statements on a line.
4. A trailing ampersand `&` is used to indicate that a statement is continued on the next line. A statement may have a maximum of 39 continuation lines.
5. A statement label consists of up to 5 consecutive digits (a number from 1 to 99999) which precedes the statement. The label is separated from the statement by at least one blank.
6. There may be any number of blanks between successive words in a Fortran 90 statement, as long as there is at least one. They will be treated by the compiler as though there was only one blank.

Note that the old **fixed form** of writing programs, which owed its origin to punched cards, is still acceptable in Fortran 90. You should, however, be consistent

in your style: use either the fixed form or the free form in a program. When writing new programs the use of the free form is recommended.

Comments Lines and Trailing Comments

In FORTRAN 77, a comment can be written on a line by typing "C" or "*" in column 1 of that line. An exclamation mark "!" may also be used to initiate comments in Fortran 90. It is not necessary, however, to place ! in the first column when using the free form. If the first non-blank character of a line is a !, that line is a **comment line**. Furthermore, a comment, preceded by !, may follow any Fortran statement or statements on a line. This is termed a **trailing comment**. For example:

```
READ *, X, Y      ! Read two real numbers
```

Continuation Lines

If the last non-blank character in a line is an ampersand, &, then this means that the statement is continued on the next line. For example,

```
CALL least_squares(x, y, n, &
                   &m, b)
```

is identical, as far as the compiler is concerned, with

```
CALL least_squares(x, y, n, m, b)
```

Notice that the first non-blank character in the second line is also an ampersand, and the statement is continued from the character after that ampersand. Omitting the ampersand in the continuation line would also be acceptable in this particular case:

```
CALL least_squares(x, y, n, &
                   m, b)
```

Here the first non-blank character (m) on the continuation line is not an ampersand. Therefore, the effect is as if the whole of that line follows the previous one (excluding the ampersand):

```
CALL least_squares(x, y, n,                          m, b)
```

Since the extra spaces come between items in the list of arguments, they do not have any effect. Remember, however, that blanks within character strings are significant.

If the ampersand occurs in a character context (in the middle of a character string enclosed in quotes or apostrophes), then the first non-blank character on the next line must be an ampersand. For example,

```
PRINT *, 'Please type the values of x, y, z &
         &in that order: '
```

has the same effect as

```
PRINT *, 'Please type the values of x, y, z in that order: '
```

Character Constants

In FORTRAN 77 only apostrophes could be used to delimit character constants. In Fortran 90, a character string constant may also be enclosed between double quotation marks:

```
PRINT *, 'This is a character constant'," and so is this."
```

As long as the same character is used at the beginning and at the end, it does not matter which is used. When an apostrophe or a quote must be included in a character string, however, the choice is important.

```
PRINT *, "This string's got an apostrophe in it", &
         ' and this string contains a "quotation"!'
```

Note that a single apostrophe is used to indicate contraction or possession when the character string is enclosed between quotation marks. Thus, "`I don't remember`" and "`Murat's book`" are valid character constants. When using a FORTRAN 77 compiler one would have to type `'I don''t remember'` and `'Murat''s book'`, respectively. (Note that `''` is two adjacent apostrophes, not a quotation mark ".) The need for double apostrophe or double quote rarely arises in Fortran 90. Thus

```
PRINT *, 'This string''s got an apostrophe in it', &
         " and this string contains a ""quotation""!"
```

is also valid, but the previous version that avoids this double apostrophe and double quote is preferable.

Type Declaration Statements

In Fortran 90 there is a new method for declaring variables. Some examples are as follows:

```
REAL :: x, y, z
INTEGER :: first_integer, second_integer, third_integer
LOGICAL :: found, flag
COMPLEX :: a, b, c
DOUBLE PRECISION :: var1, var2
```

Character variables can be declared in any one of the following three forms:

```
CHARACTER(LEN = length) :: name1, name2, ...
CHARACTER(length) :: name1, name2, ...
CHARACTER*length :: name1, name2, ...
```

Each of the variables declared in one of these ways will hold exactly *length* characters. The full form (the first version above) results in greater clarity and its use is recommended. The following is also valid (but is not recommended):

```
CHARACTER(LEN = length) :: name1, name2*length2, &
                           name3, name4*length4, ...
```

In this case *name1* is of length *length*, as are any other variables (such as *name3*) in the list without a specific length specification. *name2* has a length of *length2* and *name4* has a length of *length4*.

Arrays are declared using the **DIMENSION attribute**. For example,

```
REAL, DIMENSION(100)  :: x, y, z
```

has the same effect as

```
REAL x(100), x(100), x(100)
```

and

```
REAL :: x(100), x(100), x(100)
```

Note that the older DIMENSION statement is still valid (and should not be confused with the DIMENSION attribute). We could therefore use the alternative form

```
REAL x, y, z
DIMENSION x(100), x(100), x(100)
```

As noted in Chapter 3, however, it is more efficient to declare the type and the size of an array using a single statement.

The following statement is also valid in Fortran 90:

```
REAL, DIMENSION(20) :: x, y, a(10), b, c(100)
```

In this approach the value specified in the DIMENSION attribute applies to all variables which do not have their own array size specification. For clarity, however, it is advisable to use a separate declaration for each array size:

```
REAL, DIMENSION(20) :: x, y, b
REAL, DIMENSION(10) :: a
REAL, DIMENSION(100) :: c
```

The most general form of an array declaration is as follows:

```
type, DIMENSION(low1:high1, low2:high2, ..., lown:highn) :: &
                array_1, array_2, ...
```

where *type* is REAL, INTEGER, etc. As noted in Chapter 3, both zero and negative subscripts are allowed. Remember that Fortran allows up to seven subscripts ($n \leq 7$).

The END Statement

Execution of the END statement brings the execution of the main program unit to an end, and control is returned to the computer's operating system. In Fortran 90, the END statement may take any one of the following three forms:

```
END
END PROGRAM
END PROGRAM name
```

Similarly, the END statement at the end of a function can take any of the following forms:

```
END
END FUNCTION
END FUNCTION name
```

In the case of a subroutine, one may use any one of the following three alternative forms:

```
END
END SUBROUTINE
END SUBROUTINE name
```

where *name* is the name of the program unit in question. When the END statement is executed in a subprogram, it causes control to return to the calling program unit.

Example 7.1:

Consider the following simple Fortran 90 program. Type and run it using a Fortran 90 compiler. Note the use of two PRINT statements on a single line.

```
PROGRAM example_1
    IMPLICIT NONE  ! A standard Fortran 90 statement
    ! Program illustrates some new features of Fortran 90
    CHARACTER (len=40)::message_1, message_2
    message_1 = 'Welcome to Fortran 90!'
    message_2 = 'Fortran 90 is a powerful language.'
    PRINT*, message_1;  PRINT*, message_2;
END PROGRAM example_1
```

Initial Values and Parameters

Note that the type declaration form with a double colon is an alternative to the older (but still valid) form without a double colon. The simple declarations,

```
INTEGER num_1, num_2
INTEGER :: num_1, num_2
```

for example, are entirely equivalent. The newer form, however, is required when making use of certain new features of Fortran 90. Two of these new features which are not available when using the older form are inclusion of initial values in variable declaration and the use of the **PARAMETER attribute** to define a named constant (i.e. a parameter). The following are some examples of initialization in type declaration statements:

```
INTEGER :: max_iter = 50
CHARACTER(LEN=20) :: name = 'Unknown'
REAL :: x, y=2.5, a, b, c=1.e-06
```

Remember that the DATA statement is used in FORTRAN 77 to assign initial values. Similarly, the PARAMETER statement is used in a FORTRAN 77 program to define a named constant. Fortran 90 allows the programmer to define a named constant via the PARAMETER attribute in a type declaration statement. For example, instead of writing

```
INTEGER max_iter
PARAMETER(max_iter = 100)
```

one can write

```
INTEGER, PARAMETER :: max_iter = 100
```

The following is another valid example:

```
REAL,  PARAMETER :: pi = 3.14159, pi_half = pi/2.0
```

Note that this statement is valid because it is evaluated from left to right. If the two parameters were typed in the reverse order then there would be an error.

Relational Operators

The six relational operators have two different forms in Fortran 90 (cf. Footnote 1 in Chapter 2). The newer forms are <, >, ==, /=, <=, and >= corresponding to the older and still valid forms .LT., .GT., .EQ., .NE., .LE., and .GE..

7.3 Derived Data Types

Fortran 90 allows programmers to define their own data types. These new data types are derived from the six **intrinsic data types** (REAL, INTEGER, COMPLEX, LOGICAL, CHARACTER, DOUBLE PRECISION) and/or previously defined new data types. A **derived data type** is defined using the following format:

```
TYPE new_type
     component_definition
             .
             .
             .
END TYPE new_type
```

There may be as many component definitions as needed. Consider the following example data type:

```
TYPE person
    CHARACTER(LEN = 15) :: first_name, last_name
    CHARACTER(LEN = 15) :: father_name
    INTEGER age
    CHARACTER :: sex
END TYPE person
```

Once such a new type is defined, variables of that type can be declared. For example,

```
TYPE(person) ::  nalan, murat
```

A constant value of a derived type is written using what is called a **structure constructor**.

```
nalan = person('Nalan', 'Yakın', 'Mehmet', 22, 'F')
murat = person('Murat', 'Kara', 'Hasan', 25, 'M')
```

A component of a variable of a derived data type is referred by following the variable name by a percentage sign and the name of the component. When `murat` grows one year older, for example, one would set

```
murat%age = murat%age + 1
```

In case `murat` and `nalan` are married, one can write

```
nalan%last_name = murat%last_name
```

A previously defined derived data type can be used in defining a new type. Consider, for example, the following type definition:

```
TYPE employee
  TYPE(person) :: employee
  CHARACTER(LEN=25) :: department
  REAL :: salary
END TYPE employee
```

As this example illustrates, a component name can be the same as data type name (`employee`), but it will be clearer if the names are kept distinct. The following segment illustrates the assignment of values to a variable of type `employee`:

```
TYPE(employee) :: murat
...
murat%employee%sex = 'M'
```

Example 7.2:

Consider the following program that illustrates the use of a derived data type. Type and run the program. Note that this program could be written without using a derived data type and other new features of Fortran 90 (e.g. long variable names, free form of typing, etc.) As an exercise, rewrite the program in FORTRAN 77. Which version would you prefer (in terms of readability, elegance, ease of writing, etc.)?

It may be noted that the program assumes that February contains 28 days, i.e. the possibility of a leap year is not taken into account. This deficiency will be removed when we consider this problem again in Example 7.9.

```
PROGRAM tomorrows_date
IMPLICIT NONE
TYPE date
    INTEGER month
    INTEGER day
    INTEGER year
END TYPE date
INTEGER, DIMENSION(12) :: days_per_month = &
                          (/31,28,31,30,31,30,31,31,30,31,30,31/)
TYPE(date):: today, tomorrow
PRINT*, "Type today's date (dd mm yyyy): "
READ*, today%day, today%month, today%year
```

```
IF(today%day /= days_per_month(today%month) )THEN
    tomorrow%day = today%day + 1
    tomorrow%month = today%month
    tomorrow%year = today%year
ELSE IF(today%month == 12)THEN     ! End of year
    tomorrow%day = 1
    tomorrow%month = 1
    tomorrow%year = today%year + 1
ELSE                               ! End of month
    tomorrow%day = 1
    tomorrow%month = today%month + 1
    tomorrow%year = today%year
ENDIF
PRINT '(1X,A,I3,".",I2,".",I4)', "Tomorrow's date is", &
        tomorrow%day, tomorrow%month, tomorrow%year
END PROGRAM tomorrows_date
```

7.4 The INTENT Attribute

The **INTENT attribute** is one of the attributes that may follow the double colon in a type declaration statement. This attribute can be used only for a dummy argument in a subprogram and takes one of three forms:

- `INTENT(IN)` informs the compiler that the dummy argument is an input argument, and the subprogram is not allowed to change its value.
- `INTENT(OUT)` informs the compiler that the dummy argument is an output argument, i.e. it is used to return information to the calling program unit. The value of the argument will be undefined on entry to the subprogram. It must therefore be given a value by some means before being used in a context that requires a value (e.g. in an expression).
- `INTENT(INOUT)` informs the compiler that the dummy argument may be used for transfer of information in both directions.

If an attempt is made to modify a dummy argument with the `INTENT(IN)` attribute, for example, the compiler will detect this mistake at compile-time, as opposed to having a hard-to-detect error that manifests itself at execution-time. While the use of the `INTENT` attribute is demonstrated in the following examples, a more complete understanding of how the `INTENT` attribute can be utilized to minimize certain programming errors requires that we learn about explicit procedure interfaces (cf. Section 7.9).

Example 7.3:

Review the program of Example 5.9. That program utilizes two user-defined functions, namely `STRLEN` and `BEGSTR`, to print a character variable without the trailing or leading

blanks stored in the variable. The part of that program that generates a full name without the redundant blanks is rewritten here as a character function:

```
CHARACTER(LEN=*) FUNCTION full_name(title, first_name, mid_name, surname)
  ! Joins title, first name, middle name, and the last name
  ! to form a full name with a single space between each word.
  IMPLICIT NONE
  CHARACTER(LEN=*), INTENT(IN)::title, first_name, mid_name, surname
  full_name = TRIM(ADJUSTL(title))//' '//TRIM(ADJUSTL(first_name))//' '&
            //TRIM(ADJUSTL(mid_name))//' '//ADJUSTL(surname)
END FUNCTION full_name
```

The `INTENT` attribute is used here to indicate that the dummy arguments `title`, `first_name`, `mid_name`, and `surname` are to be used for input only.

Fortran 90 contains two new library functions, namely `TRIM` and `ADJUSTL`, that obviate the need to employ user-defined functions such as `STRLEN` and `BEGSTR` (Chapter 5). The intrinsic function `TRIM` returns the value of the input argument with any trailing blanks removed. `ADJUSTL` returns the value of the input argument with the leading blanks removed and the same number of blanks added at the end.

A main routine to test the above function is given below. Note that the lengths of the character variables `title`, `first_name`, `mid_name`, `surname`, and the length of the character value returned by `FUNCTION full_name` are all declared in the main program; the function is designed to handle varying character lengths. While this kind of flexibility may not be necessary in this simple case, it will be indispensable when you develop code for more meaningful and useful character processing applications.

```
PROGRAM hello_2
  ! Main program to demonstrate the use of FUNCTION full_name
  IMPLICIT NONE
  CHARACTER(LEN=15) :: title, first_name, mid_name, surname
  CHARACTER(LEN=63), EXTERNAL :: full_name
  PRINT*, 'Type your full name in the form requested.'
  PRINT*, 'Title (Mr.,Mrs.,Ms.,Dr.,etc.): '
  READ*, title
  PRINT*, 'First name: '
  READ*, first_name
  PRINT*, 'Middle name: '
  READ*, mid_name
  PRINT*, 'Last name: '
  READ*, surname
  PRINT*, 'Hello ', full_name(title, first_name, mid_name, surname)
  PRINT*, 'May I call you ', TRIM(ADJUSTL(first_name)),'?'
END PROGRAM hello_2
```

Note that the **`EXTERNAL` attribute** can be used as illustrated here instead of a separate `EXTERNAL` statement.

7.5 The `CASE` Construct

Fortran 90 introduced the **`CASE` construct**, which is an alternative to the block-`IF` structure in certain situations. The `CASE` structure has the following form:

```
SELECT CASE (case expression)
CASE (case selector)
    block of statements
CASE (case selector)
    block of statements
   .
   .
CASE DEFAULT
    block of statements
END SELECT
```

where *case expression* is either an integer expression, a character expression or a logical expression. A real expression cannot be used here. The CASE DEFAULT statement is optional and it may be omitted. When SELECT CASE statement is encountered, *case expression* is first evaluated. The block of statements which follow the appropriate CASE statement (if any) is then executed. The *case selector* may take one of the following four forms:

value
low_value:
:high_value
low_value:high_value

The *case selector* may also be a list of any combination of these. If none of the *case selector* values or value ranges matches the value of *case expression*, then the block of statements following the CASE DEFAULT statement (if present) is executed. If there is no CASE DEFAULT statement, then the CASE structure is exited without any code being executed. The use of the CASE structure is best explained with an example.

Example 7.4:

Review the program of Example 3.14. In that program, we have employed the computed GO TO statement "to select a case." The CASE structure is a much better form of selection than the computed GO TO statement, and it should be used when writing new programs. Compare the following version of the program with that given in Example 3.14:

```
PROGRAM seasons
!   Program determines the season of the year
!   for a given month number between 1 and 12.
IMPLICIT NONE
INTEGER month

! Read month of the year
PRINT*, 'Enter month number (between 1 and 12): '
READ*, month
! Determine the season
SELECT CASE (month)
CASE(1,2,12)
    PRINT*,'Season is winter.'
CASE(3:5)
    PRINT*,'Season is spring.'
CASE(6:8)
    PRINT*,'Season is summer.'
CASE(9:11)
    PRINT*,'Season is autumn.'
CASE DEFAULT
    PRINT*, month, ' is not a valid month.'
END SELECT
END PROGRAM seasons
```

Note that because the case expression is an integer variable (month), the case selectors must be expressed as integer constants. If the value of month is 1, 2, or 12 (representing January, February, and December, respectively), then the string 'Season is winter.' is printed. If the value of month is less that 1 or larger than 12, then the statement

```
PRINT*, month, ' is not a valid month.'
```

will be executed. Note that the second CASE statement CASE(3:5) could alternatively be written as CASE(3,4,5). Similarly, CASE(6:8) could be replaced by CASE(6,7,8), etc. Note also that the CASE statements could be placed in any order without affecting the result of the program.

The following points regarding the CASE construct should be remembered: The decision criteria in the CASE construct must not overlap. As a result of this, the order in which the CASE statements are placed does not matter. On the other hand, the order in which the decision criteria of a block-IF structure are evaluated may be important when the decision criteria overlap. Consider, for example

```
IF(temperature > 30)THEN
    PRINT*, 'Hot.'
ELSE IF(temperature > 15) THEN
    PRINT*, 'Warm.'
ELSE
    PRINT*, 'Cold.'
ENDIF
```

If the value of temperature is 35, for example, both of the conditions temperature > 30 and temperature > 15 will be .TRUE. (i.e. the two criteria overlap). If the decision criteria are written in the following different order, then, an incorrect result (the message 'Warm.') will be obtained:

```
IF(temperature > 15)THEN
    PRINT*, 'Warm.'
ELSE IF(temperature > 30) THEN
    PRINT*, 'Hot.'
ELSE
    PRINT*, 'Cold.'
ENDIF
```

In general, the CASE structure is more appropriate than the block-IF construct when the various alternatives are mutually exclusive, and the order in which they are evaluated is unimportant. Since the order of the CASE statements does not matter, the CASE DEFAULT statement does not have to be placed as the last CASE statement. For clarity, however, it may be good practice to place it either as the first or the last CASE statement.

Exercises:

1. Recall the programs of Example 5.4. Consider also the following function subprogram. Write a Fortran 90 main program to test this function. Note how the CASE construct is employed. Note also the use of the Fortran 90 functions IACHAR and ACHAR. Does the function handle the Turkish characters (e.g. ş, Ç, etc.) correctly? Would this function work as intended on a processor employing the EBCDIC code?

```
CHARACTER FUNCTION change_case(char)
  ! Changes the case of the argument if it is alphabetic
  ! Returns char unchanged if it is not alphabetic
  IMPLICIT NONE
  CHARACTER, INTENT(IN)::char
  INTEGER, PARAMETER:: upper_to_lower = IACHAR('a')-IACHAR('A')
  SELECT CASE(char)
  CASE('A':'Z')
    change_case = ACHAR(IACHAR(char)+upper_to_lower)
  CASE('a':'z')
    change_case = ACHAR(IACHAR(char)-upper_to_lower)
  CASE DEFAULT     ! Not alphabetic
    change_case = char
  END SELECT
END FUNCTION change_case
```

7.6 The DO...END DO Construct

The DO...END DO structure and the EXIT statement have already been discussed in Chapter 3. There, it has been shown that all types of iteration (i.e. Simple iteration, Do-While iteration, Do-Until iteration, and Break iteration) can be implemented employing the DO...END DO structure in conjunction with the EXIT statement. If you have access to a Fortran 90/95 compiler, therefore, it is recommended that you use this control structure instead of the older form of the DO loop (which uses statement labels) and the DO WHILE loop (which is less flexible).

Example 7.5:

Consider *N* measurements x_i, *i=1, ...,N*. The *arithmetic mean* $\bar{x}$ of the measurements is

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

The *standard deviation s* of the set of measurements is defined as:

$$s = \sqrt{\frac{N\sum_{i=1}^{N} x_i^2 - \left(\sum_{i=1}^{N} x_i\right)^2}{N(N-1)}}$$

Remember that standard deviation is a measure of the extent of scatter in the data. Here is a program that implements these formulas:

```
PROGRAM stat_analysis
IMPLICIT NONE
INTEGER :: count = 0, n
REAL :: std_dev, sum_x=0., sum_x2=0., x, x_avg
```

```
! Get input, accumulate sums
DO
    PRINT*, 'Enter value: '
    READ*, x
    IF( x < 0.) EXIT
    count = count + 1
    sum_x = sum_x + x
    sum_x2 = sum_x2 + x*x
END DO
! Check if input data are sufficient
IF(count <= 1)THEN
    PRINT*, 'Less than two values were entered.'
    STOP 'Execution terminated.'
ENDIF
! Calculate arithmetic mean and standard deviation
n = count
x_avg = sum_x/n
std_dev = SQRT((n*sum_x2-sum_x**2)/n/(n-1))
! Print results
PRINT*, 'The mean is: ', x_avg
PRINT*, 'Standard deviation: ', std_dev
END PROGRAM
```

Example 7.6:

Consider again the problem of reading and analyzing a set of exam scores (cf. Examples 3.4 and 3.8). The following program employs a count-controlled `DO` loop to read and sum the grades. Note the use of the Fortran 90 function `HUGE`.

```
PROGRAM exam_results
IMPLICIT NONE
INTEGER :: sum = 0, score, i, n
INTEGER :: minimum = HUGE(1), maximum = -HUGE(1)
REAL :: average
PRINT*, 'Enter number of scores: '
READ*, n
IF(n < 1) STOP 'Number < 1!'
PRINT*, 'Now type', n, ' numbers one by one:'
DO i = 1, n
    READ*, score
    IF(score < 0. .OR. score > 100.) STOP 'Illegal input!'
    sum = sum + score
    maximum = MAX(score, maximum)
    minimum = MIN(score, minimum)
END DO
average =  NINT(REAL(sum)/n)
PRINT*, 'Highest score: ', maximum
PRINT*, 'Lowest score : ', minimum
PRINT '(1X,A,F6.2)', 'Average      :', average
END PROGRAM
```

7.7 Modules

A complete program usually consists of a number of program units, of which exactly one must be a main program. Execution of a program always starts at the beginning

of the main program unit. Function subprograms, subroutine subprograms, and block data program units are the other types of program unit in FORTRAN 77. A new type of program unit, called a **module,** has been introduced in Fortran 90.

A module starts with an initial statement of the form

`MODULE` *module_name*

and ends with an `END` statement which takes any one of the following forms:

```
END MODULE module_name
END MODULE
END
```

The purpose of a module is quite different from that of a procedure. A module is written to make some or all of the entities declared within it accessible to more than one program unit. As will be apparent as we proceed, modules can be very useful in a number of different situations and they give considerable power and flexibility to the Fortran 90 programming language.

One use of modules has to do with **global accessibility** of constants, variables, and derived type definitions. Thus, a module allows a set of constants, variables, and/or derived type definitions to be made available to any program unit which accesses them by means of a `USE` **statement**. This statement has the form

`USE` *module_name*

where *module_name* is the name of the module in which the declarations of the mentioned constants and variables as well as any derived type definitions are placed. The `USE` statement is typed immediately after the initial statement (`PROGRAM`, `SUBROUTINE`, `FUNCTION`, or `MODULE`), before the `IMPLICIT NONE` statement.

Example 7.7:

Consider the module named `global_items` given below. This module contains the definitions of two named constants (`pi` and `pi_half`) and type declarations for three real variables (`x`, `y`, `z`). These constants and variables will be accessible from within any program unit that `USE`s `global_items`.

```
MODULE global_items
  IMPLICIT NONE
  SAVE
  REAL,PARAMETER :: pi=3.14159, pi_half=pi/2
  REAL :: x, y, z
END MODULE global_items
```

Note the SAVE statement that follows the IMPLICIT NONE statement. You should place a SAVE statement in any module that declares variables to make sure that the values of these variables do not become undefined upon return from a procedure that USEs that module.

The main program module_demo given below calls sub1, which in turn calls sub2. Note that the main program does not USE the module, as it does not need access to any of the constants and variables declared in the module. As an exercise, rewrite the program below using a common block instead of a module. Can the subroutines share the parameters pi and pi_half without using a module?

```
PROGRAM module_demo
  CALL sub1
END PROGRAM module_demo

SUBROUTINE sub1
  USE global_items
  IMPLICIT NONE
  x = pi          !x and pi accessed from module
  y = pi_half     !y and pi_half accessed from module
  PRINT*, x       !prints 3.141590
  PRINT*, y       !prints 1.570795
  CALL sub2       !x now has the value 6.283180
  PRINT*, x       !prints 6.283180
END SUBROUTINE

SUBROUTINE sub2
  USE global_items
  IMPLICIT NONE
  x = pi + 2*pi_half  !all accessed from module
END SUBROUTINE
```

The use of modules can significantly simplify the interface of procedures by eliminating long argument lists: When dealing with large programs, it often happens that several procedures need to have access to the same constants and variables. When the number of such constants and variables is small, one could pass them as arguments from one procedure to the next. When the number of passed entities is large, however, the resulting long argument lists become awkward. In FORTRAN 77, one would normally employ common blocks to simplify the interfaces of the procedures in question. In Fortran 90, modules provide a more flexible and less error-prone alternative way to accomplish this simplification. Remember, for example, that parameters cannot be placed in common blocks. Furthermore, as will be elaborated on in the next section, the *definition* of a derived type cannot be passed as an argument or by means of a common block. Such a type definition can be shared and accessed by different program units only by means of a module.

7.8 Modules and Derived Data Types

All entities (e.g. named constants, variables, arrays, derived type definitions) within a program unit are local to that unit (i.e. they cannot be accessed by other program units) unless they are either passed as arguments or shared by means of common blocks or modules. As noted before, a parameter cannot be placed in a common block. Moreover, the definition of a derived data type cannot be passed either as an argument or by means of a common block. The only method to have different program units share the definition of a derived data type is to place that definition in a module. Any program unit that `USE`s that module can then declare and use variables of that derived data type.

It should be added that repeating the definition of a derived type (using the same name and identical components) within all the program units in question will result in a different data type in each program unit, each one being local to the program unit in which it is defined. Local variables of each such type can then be declared and used within each program unit; but it will not be possible to pass constants and variables of these types as arguments between these program units. How a module can be employed to share the definition of a derived data type is illustrated in the following examples.

Example 7.8:

A straight line is represented by an equation of the form $ax+by+c=0$. Given two distinct points (x_1,y_1) and (x_2,y_2) through which the line passes, the coefficients can be calculated as follows: $a=y_2-y_1$, $b=x_1-x_2$, and $c=y_1x_2-y_2x_1$. Given below is a main program that determines the equation of a line passing through two points specified by the user.

```
MODULE geometric_types
  IMPLICIT NONE
  TYPE point
      REAL :: x,y
  END TYPE point
  TYPE line
      REAL :: a,b,c
  END TYPE line
END MODULE geometric_types
```

```
PROGRAM equation_line
  USE geometric_types
  IMPLICIT NONE
  ! Variable declarations
  TYPE(point) :: p1,p2
  TYPE(line)  :: p1_to_p2
  ! Read input data
  PRINT*, 'Enter x,y coordinates of first point: '
  READ*, p1
  PRINT*, 'Enter x,y coordinates of second point: '
  READ*, p2
  ! Calculate the equation of the line
  p1_to_p2%a = p2%y - p1%y
  p1_to_p2%b = p1%x - p2%x
  p1_to_p2%c = p1%y * p2%x - p2%y*p1%x
  ! Report results
  PRINT*, 'The coefficients of line ax + by + c = 0'
  PRINT*, 'passing through these two points are:'
  PRINT*, 'a =', p1_to_p2%a
  PRINT*, 'b =', p1_to_p2%b
  PRINT*, 'c =', p1_to_p2%c
END PROGRAM equation_line
```

Note that, in this particular example, the type definitions for `point` and `line` could be placed directly in the main program, i.e. the problem could be solved without using a module. The purpose here was to illustrate how a derived data type definition is accessed from a module.

Example 7.9:

We next look at a program that calculates "tomorrow's date" for a given "today's date." This program is an improved and generalized version of the program of Example 7.2. The derived data type `date` is defined in `MODULE date_structure`. The main program and the three functions given below `USE` this module to access the definition of `date`.

```
MODULE date_structure
  IMPLICIT NONE
  TYPE date
    INTEGER month
    INTEGER day
    INTEGER year
  END TYPE date
END MODULE date_structure
```

The main program reads the current date, invokes the function `date_update` to determine tomorrow's date, and then prints tomorrow's date. Note that the value returned by `date_update` is of the derived data type `date`.

```
PROGRAM tomorrows_date
  USE date_structure
  IMPLICIT NONE
  TYPE(date):: today, tomorrow
  TYPE(date), EXTERNAL :: date_update
  PRINT*, "Type today's date (dd mm yyyy): "
  READ*, today%day, today%month, today%year
  tomorrow = date_update(today)
  PRINT '(1X,A,I3,".",I2,".",I4)', "Tomorrow's date is", &
          tomorrow%day, tomorrow%month, tomorrow%year
END PROGRAM tomorrows_date
```

The external function `date_update` considers three possibilities: (i) today is not the last day of a month, (ii) today is the last day of December (12^{th} month), (iii) today is the last day of a month but not the last day of the year. Note the use of the integer function `number_of_days` that returns the number of days in the given month. See the definition of the function `number_of_days` given below. Note that the possibility of a leap year has to be considered. To that end the function `LEAPYR` of Example 2.4 is rewritten here in the Fortran 90 style.

```
FUNCTION date_update(today)
  USE date_structure
  IMPLICIT NONE
  TYPE(date) :: date_update
  TYPE(date) :: today, tomorrow
  INTEGER, EXTERNAL :: number_of_days
  IF(today%day /= number_of_days(today) )THEN
    tomorrow%day = today%day + 1
    tomorrow%month = today%month
    tomorrow%year = today%year
  ELSE IF(today%month == 12)THEN     ! End of year
    tomorrow%day = 1
    tomorrow%month = 1
    tomorrow%year = today%year + 1
  ELSE                               ! End of month
    tomorrow%day = 1
    tomorrow%month = today%month + 1
    tomorrow%year = today%year
  ENDIF
  date_update = tomorrow
END FUNCTION date_update


INTEGER FUNCTION number_of_days(d)
  USE date_structure
  IMPLICIT NONE
  TYPE(date) :: d
  LOGICAL, EXTERNAL :: is_leap_year
  INTEGER, DIMENSION(12) :: &
         days_per_month = (/31,28,31,30,31,30,31,31,30,31,30,31/)
  IF(is_leap_year(d) .AND. d.month == 2)THEN
    number_of_days = 29
  ELSE
    number_of_days = days_per_month(d.month)
  ENDIF
END FUNCTION number_of_days


LOGICAL FUNCTION is_leap_year(d)
  USE date_structure
  IMPLICIT NONE
  TYPE(date) :: d
  IF( (MOD(d.year,4) == 0 .AND. MOD(d.year,100) /= 0) .OR. &
                       MOD(d.year,400) == 0)THEN
    is_leap_year = .TRUE.
  ELSE
    is_leap_year = .FALSE.
  ENDIF
END FUNCTION is_leap_year
```

7.9 Explicit Procedure Interfaces

The **interface** of a procedure consists of the following: The name of the procedure, whether it is a function or subroutine, the number of arguments, the name and characteristics of each of its dummy arguments, and, in the case of a function, the characteristics of the result variable. Thus, the number, types and intents of the arguments are part of the interface of a procedure. The interface of a procedure determines the forms of reference through which it may be invoked.

In FORTRAN 77 (and earlier versions of FORTRAN), a reference to a function or call to a subroutine is made without the calling program unit knowing anything about the procedure. In this situation, the called procedure is said to have an **implicit interface** in the calling program unit. This means that information necessary for checking that the actual arguments and the corresponding dummy arguments match (in type, in intent, etc.) is not available during the compilation of the calling program unit. With implicit interfaces the compiler in effect assumes that the programmer has specified a valid procedure call and has correctly matched actual argument and dummy argument data types, etc. Consequently, many programming errors in procedure calls may not be detected during compilation; such errors may manifest themselves in unpredictable ways during execution.

A procedure interface is said to be **explicit** if the interface information is known at the point of call and does not have to be assumed. In this case, the compiler can check and guarantee the validity of a procedure call, thus providing much greater security at compile time. For certain features (such as the `INTENT` attribute of dummy arguments) provided in Fortran 90 for security and other purposes to operate properly, a procedure should have an explicit interface in any program unit that calls or references it. Furthermore, some of the new features of Fortran 90 (e.g. assumed-shape arrays, generic procedures, etc.) can only work if procedure interfaces are made explicit.

There are several ways to make the interface of a procedure explicit in a calling program unit. These can be summarized as follows:

1. The entire procedure can be placed in a module. The `CONTAINS` **statement** is utilized to do this. The interface of the procedure will then be explicit in any program unit that `USE`s that module.
2. The interfaces of all the procedures defined within a single module are explicit to each other.

3. Sometimes it is not possible or convenient to place a procedure in a module. In such a case, an **interface block** can be placed in the calling program unit to make the procedure's interface explicit in that program unit.
4. An interface block of a procedure can also be placed in a module; the definition of the procedure being placed elsewhere. The interface of the procedure will be explicit in any program unit that `USE`s that module.

As regards to the first approach, it should be added that the definitions of several procedures can be placed within a single module. The `CONTAINS` statement is placed before the definition of the first procedure within the module. Note that a procedure in a module is a program unit nested within another program unit; the `CONTAINS` statement is required to do this.

The last two approaches listed above employ the **`INTERFACE` statement**. An interface block for a procedure is specified by duplicating the heading information of that procedure, and takes the following general form:

```
INTERFACE
    Interface_body_1
    Interface_body_2
    ...
END INTERFACE
```

Each *interface_body* consists of the initial `FUNCTION` or `SUBROUTINE` statement of the corresponding procedure, followed by type declarations for the dummy arguments, and the final `END` statement. The best way to generate an interface block is to copy and paste the relevant lines of code directly from the definition of the procedure itself into the interface block; this will ensure that they are identical.

Example 7.8:

Consider the following program. Note that the comment in the subroutine indicates that the calculation to be carried out is `arg3 = arg1*arg2`, whereas this statement has been typed as `arg1 = arg2*arg3` (presumably by mistake). Let us first assume that the comment is correct and the statement is wrong.

```
PROGRAM intent_demo
  IMPLICIT NONE
  INTEGER, PARAMETER :: a=2
  INTEGER :: b=3, c=4, d
  CALL sub(a,b,c)
  CALL sub(b,c,d)
  PRINT*, 'a=', a, ' b=', b
END PROGRAM intent_demo
```

```
SUBROUTINE sub(arg1, arg2, arg3)
  !Subroutine calculates arg3
  !using arg3 = arg1*arg2
  IMPLICIT NONE
  INTEGER arg1, arg2, arg3
  arg1 = arg2*arg3
END SUBROUTINE sub
```

Using Microsoft Fortran PowerStation Version 4.0, this program compiles and runs without any error messages. The output is

```
a = 2    b = 0
```

Note, however, that there are two errors in the program: In the first subroutine call, the parameter (named constant) `a` has been used as the first actual argument. This is a programming error that results in an attempt to modify the named constant `a`. Unfortunately, some Fortran systems may not detect such an error. On some processors, a reference to the literal constant `2` or the named constant `a` later in the program may result in the value `12` being used (although this did not happen with the Microsoft compiler).

The second programming error is the use of the variable `d` as an actual argument in the second subroutine call. A value has not been assigned to `d` (i.e. `d` is undefined) at this point and therefore it should not be used in a value demanding context. Many Fortran systems will detect such an error, but some (e.g. the MS PowerStation system) will not.

Next, consider the use of the `INTENT` attribute as follows:

```
SUBROUTINE sub(arg1, arg2, arg3)
  !Subroutine calculates arg3
  !using arg3 = arg1*arg2
  IMPLICIT NONE
  INTEGER, INTENT(OUT) :: arg3
  INTEGER, INTENT(IN) :: arg1, arg2
  arg1 = arg2*arg3
END SUBROUTINE sub
```

When this version of the program is compiled, an error message is displayed at compile-time indicating that there is an attempt to set the value of the `INTENT(IN)` dummy argument `arg1`. Thus, we see that the `INTENT` attribute helps detect such a local error. But, assume now that the comment is wrong and the calculation `arg1 = arg2*arg3` is indeed the intended one. The `INTENT` attribute of the arguments will then be specified as follows:

```
SUBROUTINE sub(arg1, arg2, arg3)
  IMPLICIT NONE
  INTEGER, INTENT(OUT) :: arg1
  INTEGER, INTENT(IN) :: arg2, arg3
  arg1 = arg2*arg3
END SUBROUTINE sub
```

Again, the two errors in the calling program go undetected. This is because the subroutine has an implicit interface in the calling program, i.e. the calling program (in this case, the main program) does not know anything about the subroutine other than its name. Specifically, the types of the arguments and the intent of each argument are not known within the calling program; as a result, the compiler cannot check if the actual arguments match the dummy arguments in type and in intent. The interface of the subroutine can be made explicit by first placing the subroutine in a module,

```
MODULE module_sub
  CONTAINS
    SUBROUTINE sub(arg1, arg2, arg3)
      IMPLICIT NONE
      INTEGER, INTENT(OUT) :: arg1
      INTEGER, INTENT(IN) :: arg2, arg3
      arg1 = arg2*arg3
    END SUBROUTINE sub
END MODULE module_sub
```

and adding the statement `USE module_sub` just before the `IMPLICIT NONE` statement in the main program. An alternative approach is to place an interface block after the `IMPLICIT NONE` statement within the main program:

```
PROGRAM intent_demo
  IMPLICIT NONE
  INTERFACE
    SUBROUTINE sub(arg1, arg2, arg3)
      INTEGER, INTENT(OUT) :: arg1
      INTEGER, INTENT(IN) :: arg2, arg3
    END SUBROUTINE sub
  END INTERFACE
  INTEGER, PARAMETER :: a=2
  INTEGER :: b=3, c=4, d
  CALL sub(a,b,c)
  CALL sub(b,c,d)
  PRINT*, 'a=', a, ' b=', b
END PROGRAM intent_demo
```

Still another alternative is to place the interface block in a module and to `USE` that module within the calling program unit:

```
MODULE sub_interface
  INTERFACE
    SUBROUTINE sub(arg1, arg2, arg3)
      INTEGER, INTENT(OUT) ::arg1
      INTEGER, INTENT(IN) :: arg2, arg3
    END SUBROUTINE sub
  END INTERFACE
END MODULE sub_interface

PROGRAM intent_demo
  USE sub_interface
  IMPLICIT NONE
  INTEGER, PARAMETER :: a=2
  INTEGER :: b=3, c=4, d
  CALL sub(a,b,c)
  CALL sub(b,c,d)
  PRINT*, 'a=', a, ' b=', b
END PROGRAM intent_demo
```

Note that with the two approaches that utilize an interface block, a separate definition of the procedure exists somewhere (either in the same source file as the main program, or in a separate file). When the entire definition of the procedure is placed within a module using the `CONTAINS` statement, however, a separate interface block is not required. All that is required is then to `USE` that module within the calling program unit.

7.10 Writing Generic Subprograms

Many of the intrinsic functions of Fortran can have arguments of more than one type. For such functions, the type of the result will usually (though not always) be of the same type as the arguments. Thus, if `X` and `I` are two variables of type real and integer, respectively, `ABS(X)` will produce a real value, whereas `ABS(I)` will produce an integer.

Functions with this property are called **generic functions** (cf. Chapter 1). This is because the name of such a function really refers to a group of functions, the appropriate one being selected by the compiler depending on the types of the actual arguments. Remember that you may refer directly to the actual function instead of using its generic name, e.g. you can write `IABS(I)` instead of `ABS(I)`.

Example 7.11:

Consider Example 2.3: Note that `FUNCTION ISIGN2` given in that example works for integer arguments only. The following program shows how a single generic name (`sgn`) can be used to refer to a group of external functions. Note that the generic name `sgn` can be used with either integer or real arguments. As an exercise, add code so that double precision arguments can also be handled. Make sure to test the resulting program.

```
MODULE sign_functions
  IMPLICIT NONE
  INTERFACE sgn
    MODULE PROCEDURE int_sgn
    MODULE PROCEDURE real_sgn
  END INTERFACE
  CONTAINS
    INTEGER FUNCTION int_sgn(a, b)
      IMPLICIT NONE
      INTEGER, INTENT(IN) :: a, b
      IF(b < 0)  int_sgn = -ABS(a)
      IF(b == 0) int_sgn = 0
      IF(b > 0)  int_sgn = ABS(a)
    END FUNCTION int_sgn
    REAL FUNCTION real_sgn(a, b)
      IMPLICIT NONE
      REAL, INTENT(IN) :: a, b
      IF(b < 0)  real_sgn = -ABS(a)
      IF(b == 0) real_sgn = 0
      IF(b > 0)  real_sgn = ABS(a)
    END FUNCTION real_sgn
END MODULE sign_functions
```

```
PROGRAM test_generic_sgn
  !Driver routine to demonstrate the use of
  !user defined generic function sgn
  USE sign_functions
  IMPLICIT NONE
  REAL :: x, y
  INTEGER :: m, n
  PRINT*, 'Enter two real numbers x, y :'
  READ*, x, y
  PRINT*, 'sgn(x,y) is',  sgn(x,y)
  PRINT*, 'Enter two integers m, n :'
  READ*, m, n
  PRINT*, 'sgn(m,n) is',  sgn(m,n)
END PROGRAM test_generic_sgn
```

7.11 Guide to Further Study

As noted in the Preface, some of the important new features of Fortran 90 had to be left out in this edition of the book. Most notable among these are (i) new library functions, (ii) new array processing capabilities and dynamic memory allocation, (iii) pointers and dynamic data structures, (iv) parameterized data types, kind type parameters, and the functions `SELECTED_INT_KIND` and `SELECTED_REAL_KIND`, (v) recursive procedures, and (vi) the `CYCLE` statement. These and other new features of the language will have to be included in the next edition of the book.

For further self-study, I recommend the books by Ellis et al. (1994) and Chapman (1998). You can also find a lot of information, including some free tutorials and books, on the World Wide Web. Start your search with the following:

www.uni-comp.com/fortran

www.uni-comp.com/fortran/FAQ/cont.html/

REFERENCES

[1] Adams, J.C. et al., *Fortran 95 Handbook: Complete ISO/ANSI Reference*, The MIT Press, Cambridge, Massachusetts, 1997.

[2] Borse, G.J., *FORTRAN 77 and Numerical Methods for Engineers*, Second Edition, PWS-KENT Publishing Company, Boston, Massachusetts, 1991.

[3] Chapman, S.J., *Fortran 90/95 for Scientists and Engineers*, WCB/McGraw-Hill, Boston, Massachusetts, 1998.

[4] Covington, M. and Downing, D., *Dictionary of Computer Terms*, Barron's Educational Series, New York, 1989.

[5] Ellis, T.M.R. and Philips, I.R., *Programming in F*, Addison-Wesley Publishing Company, Harlow, England, 1998.

[6] Ellis, T.M.R., Philips, I.R., and Lahey, T.M., *Fortran 90 Programming*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.

[7] Etter, D.M., *Structured FORTRAN 77 for Engineers and Scientists*, Fourth Edition, The Benjamin/Cummings Publishing Company, Redwood City, California, 1993.

[8] Friedman, F.L. and Koffman, E.B., *Problem Solving and Structured Programming in FORTRAN*, Second Edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.

[9] Friedman, F.L. and Koffman, E.B., *FORTRAN with Engineering Applications*, Fifth Edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1993.

[10] Kochan, S.G. *Programming in C*, Revised Edition, Hayden Books, Indiana, 1990.

[11] Kruger, A., *Efficient FORTRAN Programming*, John Wiley & Sons, New York, 1990.

[12] Lipschutz, S. and Poe, A., *Programming with FORTRAN*, Schaum's Outline Series, McGraw-Hill Book Company, New York, 1978.

[13] Microsoft Corporation, *Microsoft FORTRAN Reference (Development System Version 5.1)*, U.S.A., 1993.

[14] Organick, E.I. and Meissner, L.P., *FORTRAN IV*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1974.

[15]Press, H.W., Flannery, B.P., Teukolsky, S.A., and Vetterling, W.T., *Numerical Recipes: The Art of Scientific Computing* (Original FORTRAN 77 and Pascal version), Cambridge University Press, Cambridge, 1986.

[16]Press, H.W., Flannery, B.P., Teukolsky, S.A., and Vetterling, W.T., *Numerical Recipes in FORTRAN 77: The Art of Scientific Computing (Volume 1 of Fortran Numerical Recipes),* Second Ed., Cambridge University Press, Cambridge, 1999.

[17]Press, H.W., Flannery, B.P., Teukolsky, S.A., and Vetterling, W.T., *Numerical Recipes in Fortran 90: The Art of Parallel Scientific Computing (Volume 2 of Fortran Numerical Recipes),* Cambridge University Press, Cambridge, 1996.

[18]Ralston, A. and Reilly, E.D.Jr. (Editors), *Encyclopedia of Computer Science and Engineering*, Second Edition, Van Nostrand Reinhold Company, New York, 1983.

[19]Smith, I.M. *Programming in Fortran 90*, John Wiley & Sons, West Sussex, 1995.

[20]Spiegel, M.R., *Mathematical Handbook of Formulas and Tables*, Schaum's Outline Series, McGraw-Hill Book Company, New York, 1968.