UNIVERSITY OF OSLO
Department of Geosciences

GEF4510: Intro to Fortran 95 programming

Gunnar Wollan

Autumn 2010

1 Introduction

The purpose of this compendium is to give a good insight in the Fortran 95 programming language by going through a number of examples showing how computational problems can be solved using Fortran 95.

1.1 Why use Fortran?

In the last 15 to 20 years Fortran has been looked upon as an old-fashioned unstructured programming language by researchers and students in the field of Informatics. Fortran has lacked most of the features found in modern programming languages like C++, Java etc. Especially the lack of object orientation has been the main drawback of Fortran. Most of this is no longer true. Fortran 95 has all the modern features including use of objects, operator overloading and most of what one would expect of a modern programming language. The only thing missing is genuine OOP functionality like in C++ and Java.

But why not forget Fortran and concentrate on using available OOP languages? The answer is very simple, **speed**. In the field of natural sciences, computer simulations of natural phenomena are becoming increasingly more important. Laboratory experiments are getting too complex and too costly to be performed. The only alternative is to use a computer simulation to try to solve the problem under study. Thus the need to have your code execute faster becomes more and more important when the simulations grows larger and larger. In number-crunching Fortran still has an edge in speed over C and C++. Tests has shown that an optimized Fortran program in some cases runs up to 30 percent faster than the equivalent C or C++ program. For programs with a runtime of weeks even a small increase in speed will reduce the overall time it takes to solve a problem.

1.2 Historical background

Seen in a historical perspective Fortran is an old programming language. 1954 John Backus and his team at IBM begin developing the scientific programming language Fortran. It was first introduced in 1957 for a limited set of computer architectures. In a short time the language spread to other architectures and has since been the most widely used programming language for solving numerical problems.

The name Fortran is derived from **For**mula **Tran**slation and it is still the language of choice for fast numerical computations. A couple of years later in 1959 a new version, Fortran II was introduced. This version was more advanced and among the new features was the ability to use complex numbers and splitting a program into various subroutines. In the following years Fortran was further developed to become a programming language that was fairly easy to understand and well adapted to solve numerical problems.

In 1962 a new version called Fortran IV emerged. This version had among it's features the ability to read and write direct access files and also had a new data-type called LOGICAL. This was a Boolean data-type with two states true or false. At the end of the seventies Fortran 77 was introduced. This version contained better loop and test structures. In 1992 Fortran 90 was formally introduced as an ANSI/ISO standard. This version of Fortran has made the language into a modern programming language. Fortran 95 is a small extension of Fortran 90. These latest versions of Fortran has many of the features we expect from a modern programming languages. Further development is the Fortran 2003 version which incorporates object-oriented programming support with type extension and inheritance, polymorphism, dynamic type allocation and type-bound procedures.

2 The Fortran syntax

As in other programming languages Fortran 95 has it's own syntax. We shall now take a look at the Fortran 95 syntax and also that of the Fortran 77 syntax.

2.1 The structure of Fortran

To start programming in Fortran a knowledge of the syntax of the language is necessary. Therefore let us start immediately to see how the Fortran syntax look like.

Fortran has, as other programming languages, a division of the code into variable declarations and instructions for manipulating the contents of the variables.

An important difference between Fortran 77 and Fortran 95 is the way the code is written. In Fortran 77 the code is written in fixed format where each line of code is divided into 80 columns and each column has its own meaning. This division of the code lines into columns has an historically background. In the 1960s and 1970s the standard media for data input was the punched cards. They were divided into 80 columns and it was therefore naturally to set the length of each line of code to 80 characters. In table 1 an overview of the subdivision of the line of code is given.

Column number	Meaning
1	A character here means the line is a comment
2 - 5	Jump address and format number
6	A character here is a continuation from previous line
7 - 72	Program code
73 - 80	Comment

Table 1: F77 fixed format

Fortran 77 is a subset of Fortran 95 and all programs written in Fortran 77 can be compiled using a Fortran 95 compiler. In addition to the fixed code format from Fortran 77, Fortran 95 also supports free format coding. This means that the division into columns are no longer necessary and the program code can be written in a more structured way which makes it more readable and easier to maintain. Today the free format is the default settings for the F95 compiler.

2.2 Datatypes in Fortran

Traditionally Fortran has had four basic datatypes. These were INTEGER and REAL numbers, LOGICAL which is a boolean type and CHARACTER which represent the alphabet and other special non numeric types. Later the REAL data type was split into the REAL and COMPLEX data type. In addition to this a derived datatype can be used in Fortran 95. A derived datatype can contain one or more of the basic datatypes and other derived datatypes.

2.2.1 INTEGER

An INTEGER datatype is identified with the reserved word INTEGER. It has a valid range which varies with the way it is declared and the architecture of the computer it is compiled on. When nothing else is given an INTEGER has a length of 32 bits on a typical workstation and can have a value from $[-2^{31}]$ to $[2^{30}]$. In the last years it has become more usual to have computers with a 64 bits architecture. These computers can manipulate an INTEGER with a minimum value from $[-2^{63}]$ to a maximum value of $[2^{62}]$.

2.2.2 REAL

In the same manner a REAL number can be specified with various ranges and accuracies. A REAL number is identified with the reserved word REAL and can be declared with single or double precision. In table 2 the number of bits and minimum and maximum values are given.

Precision	Sign	Exponent	Significand	Max. value	Min. value
Single	1	8	23	2^{128}	2^{-126}
Double	1	11	52	2^{1024}	2^{-1022}

Table 2: REAL numbers

A double precision real number are declared using the reserved words DOUBLE PRECISION.

An extension of REAL numbers are COMPLEX numbers with their real and imaginary parts. A COMPLEX number are identified with the reserved word COMPLEX. The real part can be extracted by the function REAL() and the imaginary part with the function AIMAG() or just IMAG() depending on the compiler implementation. There is no need for writing explicit calculation functions for COMPLEX numbers like one has to do in C / C++ which lacks the COMPLEX data type.

2.2.3 LOGICAL

The LOGICAL datatype is identified by the reserved word LOGICAL and has only two values true or false. These values are identified with .TRUE. or .FALSE. and it is important to notice that the *point* at the beginning and end is a necessary part of the syntax. To omit one or more points will give a compilation error.

2.2.4 CHARACTER

The CHARACTER datatype is identified by the reserved word CHARACTER and contains letters and characters in order to represent data in a readable form. Legal characters are*a* to *z*, *A* to *Z* and some special characters +, -, *, / and =.

2.2.5 Derived datatypes

These are datatypes which are defined for special purposes. A derived datatype are put together of components from one or more of the four basic datatypes and also of other derived datatypes. A derived datatype is always identified by the reserved word *TYPE name* as prefix and *END TYPE name* as postfix.

2.3 Declaration of variables

In Fortran there are two ways to declare a variable. The first is called implicit declaration and is inherited from the earliest versions of Fortran. Implicit declaration means that a variable is declared when needed by giving it a value anywhere in the source code. The datatype is determined by the first letter in the variable name. An INTEGER is recognized by starting with the letters *I to N* and a REAL variable by the rest of the alphabet. It is important to notice that no special characters are allowed in a variable name only the letters *A - Z*, the numbers *0 - 9* and the underscore character _. A variable cannot start with a number. In addition a LOGICAL variable is, in most compilers, identified by the letter *L* and a CHARACTER variable by the letter *C*.

The other way of declaring a variable is by explicit declaration. This is in accordance with other programming languages where all variables has to be declared within a block of code before any instructions occurs.

As a general rule an implicit declaration is not a good way to program. It gives a code that is not very readable and also it is easily introduce errors in a program due to typing errors. Therefore always use explicit declaration of variables. Some variables must always be declared. These are arrays in one or more dimensions and character strings.

2.3.1 Declaration of INTEGERS

First an example of how to declare an INTEGER in Fortran 95. Note that the *KIND* parameter is architecture dependent.

```
INTEGER                 :: i  ! Declaration of an INTEGER
                              ! length (32 bit)
INTEGER(KIND=2)         :: j  ! Declaration of an INTEGER (16 bit)
INTEGER(KIND=4)         :: k  ! Declaration of an INTEGER (32 bit)
INTEGER(KIND=8)         :: m  ! Declaration of an INTEGER (64 bit)
INTEGER,DIMENSION(100) :: n  ! Declaration of an INTEGER array
                              ! (100 elements)
```

As seen in the preceding examples there are certain differences in the Fortran 77 and the Fortran 95 way of declaring variables. It is less to write when the variables are declared in the Fortran 77 style but this is offset by greater readability in the Fortran 95 style. One thing to note is that in Fortran 95 a comment can start anywhere on the code line and is always preceded by an exclamation point.

2.3.2 Declaration of REAL numbers

The REAL datatype is now in most compilers confirming to the IEEE standard for floating point numbers. Declarations of single and double precision is declared like in the next example.

```
REAL                    :: x  ! Declaration of REAL
                              ! default length (32 bit)
REAL(KIND=8)            :: y  ! Declaration of REAL
                              ! double precision (64 bit)
REAL, DIMENSION(200)    :: z  ! Declaration of REAL  array
                              ! (200 elements)
```

Fortran has, unlike C/C++, an intrinsic datatype of complex numbers. Declaration of complex variables in Fortran are shown here.

```
COMPLEX                 :: a   ! Complex number
COMPLEX, DIMENSION(100) :: b   ! Array of complex numbers
                               ! (100 elements)
```

2.3.3 Declaration of LOGICAL variables

Unlike INTEGERS and REAL numbers a LOGICAL variable has only two values, .TRUE. or .FALSE. and therefore only uses a minimum of space. The number of bits a LOGICAL variable are using depends on the architecture and the compiler. It is possible to declare a single LOGICAL variable or an array of them. The following examples shows a Fortran 77 and a Fortran 95 declaration. In other programming languages the LOGICAL variable is often called a BOOLEAN variable after Boole the mathematician.

```
LOGICAL                  :: l1  ! Single LOGICAL variable
LOGICAL, DIMENSION(100)  :: l2  ! Array of LOGICAL variables
                                ! (100 elements)
```

2.3.4 Declaration of characters

Characters can either be declared as a single CHARACTER variable, a string of characters or an array of single characters or character strings.

```
CHARACTER                          :: c1  ! Single character
CHARACTER (LEN=80)                 :: c2  ! String of characters
CHARACTER, DIMENSION(10)           :: c3  ! Array of single
                                          ! characters
CHARACTER (LEN=80), DIMENSION(10)  :: c4  ! Array of character
                                          ! strings (10 elements)
```

2.3.5 Declaration of derived datatypes

```
TYPE derived
   ! Internal variables
   INTEGER           :: counter
   REAL              :: num
   LOGICAL           :: used
   CHARACTER(LEN=10) :: string
END TYPE derived
! A declaration of a variable of
! the new derived datatype
TYPE (derived)       :: my_type
```

One question arises: why use derived datatypes? One answer to that is that sometimes it is desireable to group variables together and to refer to these variables under a common name. It is usually a good practice to select a name of the abstract datatype to indicate the contents and area of use.

2.4 Instructions

There are two main types of instructions. One is for program control and the other is for giving a variable a value.

2.4.1 Instructions for program control

Instructions for program control can be split into three groups, one for loops, one for tests (even though a loop usually have an implicit test) and the last for assigning values to variables and perform mathematical operations on the variables. In Fortran all loops starts with the reserved word *DO*. A short example on a simple loop is given in the following piece of code.

```
DO i = 1, 100
   !// Here instructions are performed 100 times
   !// before the loop is finished
END DO
```

The next example shows a non terminating loop where a test inside the loop is used to exit the loop when the result of the test is true.

```
DO
   a = a * SQRT(b) + c
   IF (a > z) THEN
      !// Jump out of the loop
      EXIT
   END IF
END DO
```

This small piece of code gives the variable a a value from the calculation of the square root of the variable b and multiplied with the last value of a and the addition of variable c. When the value of a is greater then the value of variable z the program transfer control to the next instruction after the loop. We assumes here that all the variables has been initialized somewhere in the program before the loop. The various Fortran instructions will be described in the following chapters through the examples on how problems can be solved by simple Fortran programs.

3 A very simple Fortran 95 program

The best way of learning a new programming language is to start using it. What we shall do now is to take a small problem and program a solution to that problem.

3.1 Programming our first problem

We want to make a small program solving the conversion of a temperature measured in Farenheit to Centigrade.
The formula is $C = (F - 32) \cdot \frac{5}{9}$.
All Fortran programs starts with the `PROGRAM programname` sentence and ends with the `END PROGRAM programname`. In contrast to other programming languages is Fortran case insensitive. It means that a variablename written in uppercase letters is the same varaible as one written inlowercase letters. So let us write some code namely the first and last line in a program.

```
PROGRAM farenheit_to_centigrades

END PROGRAM farenheit_to_centigrades
```

Next is to declare the variables and perform the conversion.

```
PROGRAM farenheit_to_centigrades
  IMPLICIT NONE
  REAL              :: F ! Farenheit
  REAL              :: C ! Centigrade
  F = 75
  C = (F - 32)*5/9
  PRINT *, C
END PROGRAM farenheit_to_centigrades
```

So what have we been doing here??
After the first line we wrote `IMPLICIT NONE`. This means that we have to declare all the variables we use in the program. A heritage from the older versions of Fortran is the use of implicit declaration of variables. As a default all variables starting with the letter $i - n$ is automaticly an integer and the rest of the alphabet is a real variable. The use of implicit declarations is a source of errors and should not be used. Next we declare the variable to hold the temperature in Farenheit (`F`) and also the same for the temperature in Centigrade (`C`). Note the construct `! Farenheit` which tells the compiler that the rest of the line is a comment. Before we can start the computation we have to give the `F` variable a value, in this case 75 degrees Farenheit. Now that we have the temperature we simply calculate the corresponding temperature in degrees Centigrade. To se the result of the calculation we simply writes the result to the computer screen using the `PRINT *` and the name of the variable containing the value we want so display. Note that the `PRINT *` and the variable name `C` is separated with a comma.

All we have to do now is to compile the program and run it to see if it works. The process of compiling a program will be dealt with later, but the simplest form for compiling can be done like this:

```
gfortran -o farenheit_to_centigrades farenheit_to_centigrades.f90
```

The name of the compiler is here `gfortran` where the `-o` tells the compiler to create the executable program with the name occurring after the `-o` and last the name of the file containing the source code. Note that all Fortran 95 source files has the ending `.f90`. To run the program we simply type:

```
./farenheit_to_centigrades
```

and then the result will be printed to the screen which in this case is 23.88889. Note the use of `./` before the program name. This tells the command interpreter to use the program in the current directory.

3.2 What have we learned here??

We have learned

- Never to use implicit delarations
- How to declare real variables
- How to assign a value to a variable
- How to perform a simple calculation
- How to print a value to the screen

3.3 A small exercise

Rewrite the tamperature conversion program to make the conversion the other way from Centigrade to Farenheit.

4 A more userfriendly Fortran 95 program

In the previous section we made a program converting a temperture from Farenheit to Centigrades. It was not a very userfriendly program since we had to change the value of the Farenheit variable and recompile the program for each new temperature we would convert. We shall now try to make the program asking the user for a temperature before we perform the calculation.

4.1 Extending our program

Let us take the source code from our first program and simply add some new lines of code to ask the user for a temperature and then read the temperature from the keyboard.

```
PROGRAM farenheit_to_centigrades2
  IMPLICIT NONE
  ! The temperature variables
  REAL               :: F ! Farenheit
  REAL               :: C ! Centigrade
  ! A text string
  CHARACTER(LEN=32)  :: prompt
  ! The text that will be displayed
  prompt = 'Enter a temperture in Farenheit:'
  ! Display the prompt
  PRINT *, prompt
  ! Get the input from the keyboard
  ! and put it into the F variable
  READ(*,*) F
  ! Perform the calculation
  C = (F - 32)*5/9
  ! Display the result
  PRINT *, C
END PROGRAM farenheit_to_centigrades2
```

The text string `prompt` is used to contain a message to the user telling the user to enter a temperature in degrees Farenheit. To get the temperature entered by the user we simply use the `READ(*,*)` function to capture the input from the keyboard. The syntax `(*,*)` tells the program to retrieve the temperature form the keyboard. The first `*` identifies the input source as the keyboard and the secont `*` tells the system to use default formatting of the input. The rest of the program is just like the first version where we hardcoded the temperature.
Again w compile the program like this:

```
gfortran -o farenheit_to_centigrades2 farenheit_to_centigrades2.f90
```

To run the program we type:

```
./farenheit_to_centigrades2
```

and then the result will be printed to the screen with the Centigrade temperature corresponding to the Farenheit temperature.

4.2 What have we learned here??

We have learned

- To declare a character string

- To give the character string a value
- Displaying the contents of the character string
- Reading a number form the keyboard

4.3 A small exercise

Extend the program with code to ask if the temperature is in Centigrades or Farenhait and perform the conversion either way. In addition print a line of test telling if it is a conversion from Farenheit to Centigrade or from Centigrade to Farenheit.
Hint: The `PRINT *,` statement can take several variables as arguments to be displayed on the screen. The arguments are separated with a comma like this `PRINT *, variable1, variable2, variable3`.

5 The use of arrays

In this compendium we use the name `array` for both a vector and a matrix. A one dimensional array is another name for a vector and an array with two or more dimensions is the same as a matrix with two or more dimensions. A one dimensional array can be seen as a set of containers where each container is an element in the array like in this figure

Here we have an array with seven elements each containing a value.

5.1 A one dimensional array example

Let us take the example array above and use the values as temperatures in degrees Farenheit. We shall now program the array, give each element the value from the example and convert the values into degrees Centigrade stored in a separate array.

```
PROGRAM array1D
  IMPLICIT NONE
  ! The temperture arrays
  REAL, DIMENSION(7)  :: farenheit
  REAL, DIMENSION(7)  :: centigrade
  ! Some help variables
  ! An index variable
  INTEGER             :: i
  ! Insert the temperature values into the
  ! farenheit array
  farenheit(1) = 68.2
  farenheit(2) = 69.6
  farenheit(3) = 73.3
  farenheit(4) = 75.0
  farenheit(5) = 77.6
  farenheit(6) = 79.2
  farenheit(7) = 81.2
  ! Performing the conversion using a loop
  DO i = 1, 7
    centigrade(i) = (farenheit(i) - 32)*5/9
  END DO
  ! Print the value pairs for each element in the
  ! two arrays
  DO i = 1, 7
    PRINT *, 'Farenheit: ', farenheit(i), &
             ' Centigrade: ', centigrade(i)
  END DO
END PROGRAM array1D
```

The declaration of the temperature arrays is atraight forward using the `DIMENSION(7)` parameter in addition to the datatype which is `REAL` in this case. To give the Farenheit temperature to each array element we have to use the element number to put the value in the right place. To convert the values we use a loop with an index variable taking the number from one to seven.

The loop is declared using the `DO - END DO` construct. Note that for each time we go through the loop the index variable is incremented by one. When the index variable get a value larger than the given end value (seven in this case) the loop is teminated and the program continues from the line after the `END DO` statement.
Then again we compile the program

```
gfortran -o array1D array1D.f90
```

and run it like this

```
./array1D
```

5.2 What have we learned here??

We have learned

- To declare a vector of real numbers
- To initiaize the vector
- Perform calculations on each element of the array
- Printing the result of our calculations to the screen

5.3 A small exercise

Try to find out how we can print out every second element from the array.

6 The use of 2D arrays

Using a one dimensional array is quite simple. Using a two dimensional array is somewhat more complex, but we will take a look at how we do this here.

6.1 A two dimensional array example

Again we shall use a temperature array like in the previous example, but now we have tempearture measurements from two different places. That means we will have to use a two dimensional array to store the temperatures from the two places. The array will look like this:

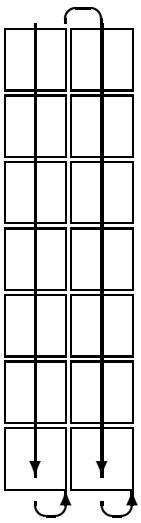
68.2	65.4
69.6	63.7
73.3	66.1
75.0	68.0
77.6	70.1
79.2	71.4
81.2	73.2

So let us go on programming it. First we will have to declare the arrays for both Farenheit and Centigrade. Note that in this declaration we define the number of rows before the number of columns. In this case the number of rows is 7 and the number of columns is 2.

```
PROGRAM array2D
  IMPLICIT NONE
  ! The temperture arrays
  REAL, DIMENSION(7,2)  :: farenheit
  REAL, DIMENSION(7,2)  :: centigrade
  ! Some help variables
  ! Two index variables
  INTEGER               :: i
  INTEGER               :: j
  ! Insert the temperature values into the
  ! farenheit array
  farenheit(1,1) = 68.2
  farenheit(2,1) = 69.6
  farenheit(3,1) = 73.3
  farenheit(4,1) = 75.0
  farenheit(5,1) = 77.6
  farenheit(6,1) = 79.2
  farenheit(7,1) = 81.2
  farenheit(1,2) = 65.4
  farenheit(2,2) = 63.7
  farenheit(3,2) = 66.1
```

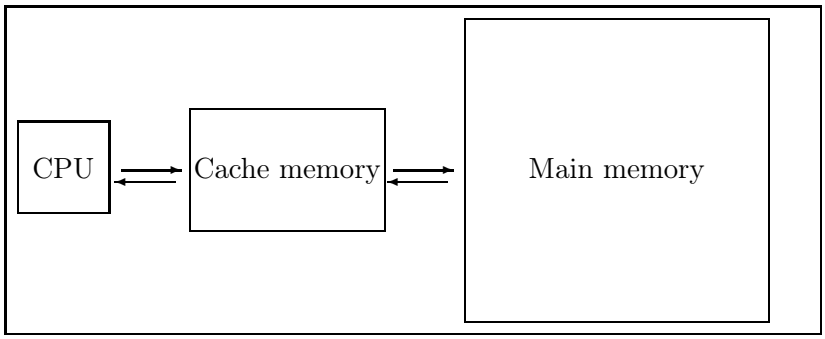
```
 farenheit(4,2) = 68.0
 farenheit(5,2) = 70.1
 farenheit(6,2) = 71.4
 farenheit(7,2) = 73.2

 ! Performing the conversion using a loop
 DO j = 1, 2
   DO i = 1, 7
     centigrade(i,j) = (farenheit(i,j) - 32)*5/9
   END DO
 END DO
 ! Print the value pairs for each element in the
 ! two arrays
 DO j = 1,2
   DO i = 1, 7
     PRINT *, 'Farenheit: ', farenheit(i,j), &
              ' Centigrade: ', centigrade(i,j)
   END DO
 END DO
END PROGRAM array2D
```

Some explanation is needed here. First of all the structure of this array consitst of two columns with seven rows in each column. Fortran allways access the element columnwise. That is each row element of a column is accessed before we proceed to the next column. It is important to notice the use of the index variables. We have the first index in the innermost loop thus traversing the array as indicated in the **??**. Note also in the two lines containing the `PRINT` statement we use an ampersand `&` at the end of the first line in the statment. This is to tell the compiler that this line continues on the next line. In Fortran we do not have any `begin - end` structure because Fortran is a line oriented language due to the use of punched cards as an input media.



Why is it so important to traverse the array in this manner? The answer has to do with how the computer is constructed. In the figure below we can see that the `CPU` (Central Processing Unit) get the data and instructions via a `Cache memory` which is almost as fast as the `CPU`. The `Cache`

memory get the data and instructions from the Main memory which is rather slow compared to the Cache memory and the CPU. To utilize the CPU as effective as possible the Cache memory has to be filled with as much relevant data as possible. If we traverse an array in the wrong direction we have to go to the slow Main memory to fetch the data we need for each iteration in the innermost loop. For very large arrays this is costly in CPU-cycles and will slow down the execution speed of the program.



So now we see how inportant it is to traverse an array in the right manner to keep the CPU operating at full speed.
Then again we compile the program

```
gfortran -o array2D array2D.f90
```

and run it like this

```
./array2D
```

6.2 What have we learned here??

We have learned

- To declare a two dimensional array of real numbers
- To initiaize the array
- Perform calculations on each element of the array in a nested loop
- Printing the result of our calculations to the screen

6.3 A small exercise

Try to find out how we can print out only the elements from the first column in the array and program the solution.

7 Getting data from a file

In the previous sections we have hardcoded the temperature in the program source. This is of course not what we normally do. The input data is usually stored in a file which can be an ASCII-file which is a readable text file or in a binary file. We will use the ASCII-file type in our prblem solving here.

7.1 How to read data from a file

The dataset containing the temperatures is stored in an ASCII-file with two real numbers separated by a space like this:

```
68.2 65.4
69.6 63.7
73.3 66.1
75.0 68.0
77.6 70.1
79.2 71.4
81.2 73.2
```

What we have to do here is to read the contents of each line into the corresponding array elements. To be able to read the contents of a file into an array we have to open the file just like we would open the drawer in a file cbinet to get access to the content of the drawer. We use the `OPEN` function with a set of arguments to do this. After the opening of the file we should always test to see if the opening was successful. If not we just terminate the program. To read the contents of the file w use the `READ` function with a set of arguments. Since we are reading the file one line at a time we have to use a loop to put the values into the correct position in the array. We again should test to see if the read operation was successful and close the file and terminate thre program if an error occurred. Finally we close the file using the `CLOSE` function with one argument. The program code below illustrates how we can do this.

```
PROGRAM readtemp
  IMPLICIT NONE
  ! The temperture arrays
  REAL, DIMENSION(7,2)  :: farenheit
  REAL, DIMENSION(7,2)  :: centigrade
  ! The filename
  CHARACTER(LEN=16)     :: filename
  ! A unit number to reference the file
  INTEGER, PARAMETER    :: lun = 10
  ! Some help variables
  ! Two index variables
  INTEGER               :: i
  INTEGER               :: j
  ! A result variable
  INTEGER               :: res
  ! Set the filename into the variable
  filename = 'temperature.txt'
  ! Open the file for reading
  OPEN(UNIT=lun,FILE=filename,FORM='FORMATTED', &
       IOSTAT=res)
```

```
  ! Test if the file was opened correctly
  IF(res /= 0) THEN
    ! Print an error message
    PRINT *, 'Error in opening file, status: ', res
    ! Stop the porgram
    STOP
  END IF
  ! Read the data into the farenheit array
  DO i = 1, 7
    READ(UNIT=lun,FMT='(F4.1,X,F4.1)',IOSTAT=res) &
        farenheit(i,1), farenheit(i,2)
    IF(res /= 0) THEN
      PRINT *, 'Errror in reading file, status: ', res
      CLOSE(UNIT=lun)
      STOP
    END IF
  END
  ! Perform the conversion
  DO j = 1, 2
    DO i = 1, 7
      centigrade(i,j) = (farenheit(i,j) - 32)*5/9
    END DO
  END DO
  ! Print the value pairs for each element in the
  ! two arrays
  DO j = 1,2
    DO i = 1, 7
      PRINT *, 'Farenheit: ', farenheit(i,j), &
               ' Centigrade: ', centigrade(i,j)
    END DO
  END DO
END PROGRAM readtemp
```

Some explanations might be useful. We already know how to declare variables, but in this program we also declare a constant value referred to by a name using this syntax `INTEGER, PARAMETER :: lun = 10`. Here the constant value is the number 10. Why, you ask, do we do it this way and not just write the number 10 instead of using the `lun` variable. The answer is to prevent errors in case we need to change the value. Using a constant we only have to change the value in one place and not everywhere in the source code as we had to do if we used the number in several places in the code. To open a file for reading we use the `OPEN` function with four keyword-argument pairs. The first argument is the unit number we use as a reference when we access the file later in the program preceeded by the keyword `UNIT=`. The next is the filename again preceded by a keyword-argument pair which is here `FILE=filename`. The third argument is the type of file wer are opening. Here it is an ASCII file (a human readable text file) which the keyword-argument pair is `FORM='FORMATTED'`. The last keyword-argument pair is `IOSTAT=res` where the argument is the name of an integer variable which will contain the result of the opening of the file. Any number except zero marks a failure to open the file. The if test `IF(res /= 0)` returns true it means that the `res` variable contains an error number telling what went wrong in opening the file. The problem could for example be that the filename is misspelled or we are in a wrong directory or some other error situation. Then we proceed to

read the contents of the file into the farenheit array. We use the function `READ` wiht a number of keyword-argument pairs. The first is the same as for the `OPEN` function. The next describes how the format is for the values in the input file. her the format of the file is `FMT='(F4.1,X,F4.1)'` which means that we have two real numbers on each line in the file. The numbers have a total of 4 positions each including the decimal point and with one decimal which is denoted by `F4.1`. Each number is separated by a character which is just one space character and is refeered to by the letter `X` which tells the system to ignore this character in the reading process. Note that we put the first number in the first column of the farenheit array and the second number in the second column. The index variable points to the corresponding row in the array while we here are hardcoding the second index. Note also the use of an ampersand `&` at the end of the line containing the `PRINT` statement and the one at the beginning of the next line. This is the syntax allowing a text constant to span over more than one line.

7.2 What have we learned here??

We have learned

- We have declared a constant using the `PARAMETER` keyword
- The process of opening a file for reading, specifying the filetype and test if the opening process was a success
- In order to read the contenst of the file we have passed the format specifications to the `READ` function and storing the values into the farenheit varray. Then testing for errors in the reading process
- To have a text constant span more than one line using two ampersands

7.3 A small exercise

Change the program to just print out the first column of the centigrade array

8 Programming formulas

Now that we have learned to read a set of data into an array we will start to program some functions which will operate on the dataset.

8.1 What is a function in Fortran?

A function in Fortran 95 is just like a mathematical function. It receives one or more input arguments and returns a result. One such function we shall program is the `mean` function which will return the mean value of the argument which will have to be a 1D array. The mathematical formula for the mean value is

$$\bar{x} = \frac{1}{n} \cdot \sum_{i=1}^{n} x_i \tag{1}$$

The skeleton of our mean value function for a dataset can be like this:

```
FUNCTION mean(arg) RESULT(res)
  IMPLICIT NONE
  ! Input array
  REAL, DIMENSION(7)    :: arg
  ! The result of the calculations
  REAL                  :: res
  ! Index variable
  INTEGER               :: i
  ! Temporary variable
  REAL                  :: tmp
  ! Initialize the temporary variable
  tmp = 0.
  DO i = 1, 7
    tmp = tmp + arg(i)
  END DO
  res = tmp/7.
  RETURN
END FUNCTION mean
```

All functions declared in Fortran begins with the keyword `FUNCTION` with the name of the function and teh input argument enclosed in paranthesis. The type of value returned by the function is defined by the datatype in the `RESULT(res)` statement. Note that we have hardcoded the length of the input argument. This is of course not desireable, but as an example on how we program a function it will be ok for now. Note the initalizing of the `tmp` variable where we use the construct `tmp = 0.` where the decimal point after the number is neccessary to tell the compiler that we have a real number and not an integer. The same notion goes for the division at the end of the function. The `RETURN` stement returns the program to the calling process. In this case it will be our main program. Let us change our program to incorporate the calculation of the mean function.

```
PROGRAM readtemp
  IMPLICIT NONE
  ! The temperture arrays
  REAL, DIMENSION(7,2)  :: farenheit
  REAL, DIMENSION(7,2)  :: centigrade
  ! The filename
```

```
CHARACTER(LEN=16)     :: filename
! A unit number to reference the file
INTEGER, PARAMETER    :: lun = 10
! External declaration of the mean function
REAL, EXTERNAL        :: mean
! The variable to hold the mean value
REAL                  ::mvalue
! Some help variables
! Two index variables
INTEGER               :: i
INTEGER               :: j
! A result variable
INTEGER               :: res
! Set the filename into the variable
filename = 'temperature.txt'
! Open the file for reading
OPEN(UNIT=lun,FILE=filename,FORM='FORMATTED', &
     IOSTAT=res)
! Test if the file was opened correctly
IF(res /= 0) THEN
  ! Print an error message
  PRINT *, 'Error in opening file, status: ', res
  ! Stop the porgram
  STOP
END IF
! Read the data into the farenheit array
DO i = 1, 7
  READ(UNIT=lun,FMT='(F4.1,X,F4.1)',IOSTAT=res) &
       farenheit(i,1), farenheit(i,2)
  IF(res /= 0) THEN
    PRINT *, 'Errror in reading file, status: ', res
    CLOSE(UNIT=lun)
    STOP
  END IF
END
! Perform the conversion
DO j = 1, 2
  DO i = 1, 7
    centigrade(i,j) = (farenheit(i,j) - 32)*5/9
  END DO
END DO
! Print the value pairs for each element in the
! two arrays
DO j = 1,2
  DO i = 1, 7
    PRINT *, 'Farenheit: ', farenheit(i,j), &
             ' Centigrade: ', centigrade(i,j)
  END DO
END DO
```

```
  ! Calculate the mean value of the first column
  ! of the centigrade array
  mvalue = mean(centigrade(:,1))
  ! Print the mena value
  PRINT *, 'The mean value of the first column in centigrade',&
           mvalue
END PROGRAM readtemp
```

All functions that is not a Fortran intrinsic function has to be declared as an external function. This is done using the `EXTERNAL` parameter in addition to the type of data returned by the function. In addition to declare a variable to contain the result of the calling of our mean function we just add one line where we set the `mvalue` equal to the function `mean` and just use the `centigrade(:,1)` to send the values from the first column to the `mean` function.

8.2 What have we learned here??

We have learned

- To create a function with one input argument and a result
- That all non Fortran intrinsic functions has to be declared external in the procedure calling the function
- To call a function in our main program

8.3 A small exercise

Extend the program to print out the mean from both series and calculate the difference of the two mean values and print the reults to the screen

9 Programming formulas using functions

Having a working function for the `mean` we shall now start to write other functions which will use functions we already have programmed

9.1 Programming the standard deviation function

Th function we shall write is the standard deviation function. The mathematical formula for the standard deviation is

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}. \tag{2}$$

Note that the standard deviation function also uses the mean value of the dataset in the calculation. So we will then call our `mean` function from within the standard deviation function which we will call `std`. The skeleton of our standard deviation function can be like this:

```
FUNCTION std(arg) RESULT(res)
  IMPLICIT NONE
  ! Input array
  REAL, DIMENSION(7)    :: arg
  ! The result of the calculations
  REAL                  :: res
  ! The mean function
  REAL, EXTERNAL        :: mean
  ! Index variable
  INTEGER               :: i
  ! Temporary variables
  REAL                  :: tmp
  REAL                  :: tmpmean
  ! Calculate the mean value
  tmpmean = mean(arg)
  ! Initialize the tmp variabløe
  tmp = 0.
  ! Calculate the standard deviation
  DO i = 1, 7
    tmp = tmp + (arg(i) - tmpmean)**2
  END DO
  ! Divide the result by the number of elements
  tmp = tmp / 7.
  ! Take the square root of the tmp
  res = SQRT(tmp)
  ! Return to calling process
  RETURN
END FUNCTION std
```

We introduce a couple of new things in the `std` function. First we declare the `mean` function as an external function of type `REAL`. A new variable is used to hold the mean value of the dataset (`tmpmean`). Calling the `mean` returns the mean value of the inpur agrument. Then we loop through all the elements in the input argument and sum up the square of the element value minus the mean value. Note the use of the `**` to denote the square of the value. We divide the sum by the number of elements and finally we take the square root of the value and place it in

the `res` variable using the Fortran intrinsic function `SQRT`. Note that we do not have to declare the `SQRT` function as an external function since it is one of a set of intrinsic functions in Fortran.

Let us change our program to incorporate the calculation of the std function.

```
PROGRAM readtemp
  IMPLICIT NONE
  ! The temperture arrays
  REAL, DIMENSION(7,2)  :: farenheit
  REAL, DIMENSION(7,2)  :: centigrade
  ! The filename
  CHARACTER(LEN=16)     :: filename
  ! A unit number to reference the file
  INTEGER, PARAMETER    :: lun = 10
  ! External declaration of the mean function
  REAL, EXTERNAL        :: std
  ! The variable to hold the mean value
  REAL                  ::mvalue
  REAL                  ::stdvalue
  ! Some help variables
  ! Two index variables
  INTEGER               :: i
  INTEGER               :: j
  ! A result variable
  INTEGER               :: res
  ! Set the filename into the variable
  filename = 'temperature.txt'
  ! Open the file for reading
  OPEN(UNIT=lun,FILE=filename,FORM='FORMATTED', &
       IOSTAT=res)
  ! Test if the file was opened correctly
  IF(res /= 0) THEN
    ! Print an error message
    PRINT *, 'Error in opening file, status: ', res
    ! Stop the porgram
    STOP
  END IF
  ! Read the data into the farenheit array
  DO i = 1, 7
    READ(UNIT=lun,FMT='(F4.1,X,F4.1)',IOSTAT=res) &
         farenheit(i,1), farenheit(i,2)
    IF(res /= 0) THEN
      PRINT *, 'Errror in reading file, status: ', res
      CLOSE(UNIT=lun)
      STOP
    END IF
  END
  ! Perform the conversion
  DO j = 1, 2
    DO i = 1, 7
      centigrade(i,j) = (farenheit(i,j) - 32)*5/9
```

```
    END DO
  END DO
  ! Print the value pairs for each element in the
  ! two arrays
  DO j = 1,2
    DO i = 1, 7
      PRINT *, 'Farenheit: ', farenheit(i,j), &
               ' Centigrade: ', centigrade(i,j)
    END DO
  END DO
  ! Calculate the mean value of the first column
  ! of the centigrade array
  mvalue = mean(centigrade(:,1))
  ! Print the mean value
  PRINT *, 'The mean value of the first column in centigrade',&
           mvalue
  ! Calculate the standard deviation value
  stdvalue = std(centigrade(:,1))
  ! Print the std value
  PRINT *, 'The standard deviation value of the first column &
          & in centigrade', stdvalue

END PROGRAM readtemp
```

9.2 What have we learned here??

We have learned

- To make a function use another function which we have written
- To use the square syntax ** to calculate the square of a number
- To take the square root of a value using the SQRT Fortran intrinsic function

9.3 A small exercise

Extend the program to print out the standard deviation from both series and calculate the difference of the two values and print the reults to the screen

10 Programming subroutines

Now that we have learned to program functions we will take a look at `subroutines` which is in many ways like a `function`.

10.1 What is a subroutine in Fortran?

A `subroutine` in Fortran 95 is just like a `function`. It receives one or more input arguments, but returns no result. Instead any returning value is passed through one or more arguments. We shall now take a look at a subroutine that calculates the `mean` value of the argument which will have to be a 1D array just like the `mean` function.

```
SUBROUTINE mean(arg,res)
  IMPLICIT NONE
  ! Input array
  REAL, DIMENSION(7),INTENT(IN) :: arg
  ! The result of the calculations
  REAL,INTENT(OUT)              :: res
  ! Index variable
  INTEGER                       :: i
  ! Temporary variable
  REAL                          :: tmp
  ! Initialize the temporary variable
  tmp = 0.
  DO i = 1, 7
    tmp = tmp + arg(i)
  END DO
  res = tmp/7.
  RETURN
END SUBROUTINE mean
```

All subroutines declared in Fortran begins with the keyword `SUBROUTINE` with the name of the subroutine and the input and output arguments enclosed in paranthesis. The result is returned int th `res` argument. Note the use of the keywords `INTENT(IN)` and `INTENT(OUT)` which will flag an error if we try to write something in the variable with the parameter `INTENT(IN)` and likewise try to read something from the variable with the parameter `INTENT(OUT)`. The internal functionality is the same as for the `mean` function. We have only two changes in the main program to be able to use the subroutine. First we remove the line declaring the external `mean` function. SEcond we change the line calling the function with the construct `CALL mean(centigrade(:,1),res)`.

```
PROGRAM readtemp
  IMPLICIT NONE
  ! The temperture arrays
  REAL, DIMENSION(7,2)  :: farenheit
  REAL, DIMENSION(7,2)  :: centigrade
  ! The filename
  CHARACTER(LEN=16)     :: filename
  ! A unit number to reference the file
  INTEGER, PARAMETER    :: lun = 10
  ! The variable to hold the mean value
  REAL                  ::mvalue
```

```
  ! Some help variables
  ! Two index variables
  INTEGER               :: i
  INTEGER               :: j
  ! A result variable
  REAL               :: res
  ! Set the filename into the variable
  filename = 'temperature.txt'
  ! Open the file for reading
  OPEN(UNIT=lun,FILE=filename,FORM='FORMATTED', &
       IOSTAT=res)
  ! Test if the file was opened correctly
  IF(res /= 0) THEN
    ! Print an error message
    PRINT *, 'Error in opening file, status: ', res
    ! Stop the porgram
    STOP
  END IF
  ! Read the data into the farenheit array
  DO i = 1, 7
    READ(UNIT=lun,FMT='(F4.1,X,F4.1)',IOSTAT=res) &
         farenheit(i,1), farenheit(i,2)
    IF(res /= 0) THEN
      PRINT *, 'Errror in reading file, status: ', res
      CLOSE(UNIT=lun)
      STOP
    END IF
  END
  ! Perform the conversion
  DO j = 1, 2
    DO i = 1, 7
      centigrade(i,j) = (farenheit(i,j) - 32)*5/9
    END DO
  END DO
  ! Print the value pairs for each element in the
  ! two arrays
  DO j = 1,2
    DO i = 1, 7
      PRINT *, 'Farenheit: ', farenheit(i,j), &
               ' Centigrade: ', centigrade(i,j)
    END DO
  END DO
  ! Calculate the mean value of the first column
  ! of the centigrade array
  CALL mean(centigrade(:,1),res)
  ! Print the mean value
  PRINT *, 'The mean value of the first column in centigrade',&
           mvalue
END PROGRAM readtemp
```

10.2 What have we learned here??

We have learned

- To create a subroutine with one input argument and an argument to hold the result of the calculations
- That we can add optional parameters to prevent overwriting values in an argument and to prevent unintentionally reading a value from an argument meant for writing
- To call a subroutine in our main program

11 Programming formulas using subroutines

11.1 Programming a subroutine calling another subroutine

Th subroutine we shall write has the same functionality as the standard deviation function. Note that the standard deviation subroutine will use the mean value subroutine from the previous section.

```
SUBROUTINE std(arg,res)
  IMPLICIT NONE
  ! Input array
  REAL, DIMENSION(7)    :: arg
  ! The result of the calculations
  REAL                  :: res
  ! Index variable
  INTEGER               :: i
  ! Temporary variables
  REAL                  :: tmp
  REAL                  :: tmpmean
  ! Calculate the mean value
  CALL mean(arg,tmpmean)
  ! Initialize the tmp variabløe
  tmp = 0.
  ! Calculate the standard deviation
  DO i = 1, 7
    tmp = tmp + (arg(i) - tmpmean)**2
  END DO
  ! Divide the result by the number of elements
  tmp = tmp / 7.
  ! Take the square root of the tmp
  res = SQRT(tmp)
  ! Return to calling process
  RETURN
END SUBROUTINE std
```

Like in the previous section we just change a couple of lines both in the main program and in the `std` subroutine

```
PROGRAM readtemp
  IMPLICIT NONE

....

  ! Calculate the standard deviation value
  CALL std(centigrade(:,1),stdvalue)
  ! Print the std value
  PRINT *, 'The standard deviation value of the first column &
           & in centigrade', stdvalue

END PROGRAM readtemp
```

11.2 What have we learned here??

We have learned

- To make a subroutine use another subroutine which we have written

A Operators

Operators in Fortran are for example `IF(a > b) THEN` which is a test using numerical values in the variables. For other types of variables like characters and character strings we use the construct `IF(C1 .GT. C2) THEN`.

A.1 Overview of the operators

Table3 gives an overview of the operators

Numerical	Other	Explanation
**		Exponentiation
*		Multiplication
/		Division
+		Addition
-		Subtraction
==	.EQ.	Equal
/=	.NE.	Not equal
<	.LT.	Less
>	.GT.	Greater
<=	.LE.	Less or equal
>=	.GE.	Greater or equal
	.NOT.	Negation, complement
	.AND.	Logical and
	.OR.	Logical or
	.EQV.	Logical equivalence
	.NEQV.	Logical not equivalence, exclusive or

Table 3: Logical operators

B Intrinsic functions

Intrinsic functions in Fortran are functions that can be used without referring to them via include files like in other languages where functions has to be declared before beeing used in the program

B.1 Intrinsic Functions

Table4, table5 and table6 gives an overview of the intrinsic functions in Fortran 95

Function	Argument	Result	Explanation
ABS	Integer real complex	Integer real complex	The absolute value
ACHAR	Integer	Character	Integer to ASCII character
ACOS	Real	Real	Arcuscosine
ADJUSTL	Character string	Character string	Left adjustment
ADJUSTR	Character	Character	Right adjustment
AIMAG	Complex	Real	Imaginary part
AINT	Real	Real	Truncate to a whole number
ALL	Logical mask, dim	Logical	True if all elements == mask
ALLOCATED	Array	Logical	True if allocated in memory
ANINT	Real	Real	Round to nearest integer
ANY	Logical mask, dim	Logical	True if all elemnts == mask
ASIN	Real	Real	Arcsine
ASSOCIATED	Pointer	Logical	True if pointing to target
ATAN	Real	Real	Arctangent
ATAN2	X=Real,Y=Real	Real	Arctangent
BIT_SIZE	Integer	Integer	Number of bits in argument
BTEST	I=Integer,Pos=Integer	Logical	Test a bit of an integer
CEILING	Real	Real	Leat integer <= argument
CHAR	Integer	Character	Integer to ASCCI character
CMPLX	X=Real,Y=Real	Complex	Convert to complex number
CONJG	Complex	Complex	Conjugate the imaginary part
COS	Real Complex	Real complex	Cosine
COSH	Real	Real	Hyperbolic cosine
COUNT	Logical mask, dim	Integer	Count of true entries in mask
CPU_TIME	Real	Real	Returns the processor time
CSHIFT	Array, shift, dim	Array	Circular shift of elements
DATE_AND_TIME	Char D,T,Z,V	Character	Realtime clock
DBLE	Integer real complex	Double precision	Convert to double precision
DIGITS	Integer real	Integer	Number of bits in argument
DIM	Integer real	Integer real	Difference operator
DOT_PRODUCT	X=Real,Y=Real	Real	Dot product
DPROD	X=Real,Y=real	Double precision	Double precision dot prod.
EOSHIFT	Array,shift,boundary,dim	Array	Array element shift
EPSILON	Real	Real	Smallest positive number
EXP	Real complex	Real complex	Exponential
EXPONENT	Real	Integer	Model exponenet of argument
FLOOR	Real	Real	Integer <= argument
FRACTION	Real	Real	Fractional pert of argument
HUGE	Integer real	Integer real	Largest number
IACHAR	Character	Integer	Integer value of argument
IAND	Integer,Integer	Integer	Bitwise logical and
IBCLR	Integer,pos	Integer	Setting bit in pos = 0
IBITS	Integer,pos,len	Integer	Extract len bits from pos
IBSET	Integer,pos	Integer	Set pos bit to one
ICHAR	Character	Integer	ASCII number of argument
IEOR	Integer,integer	Integer	Bitwise logical XOR
INDEX	String,substring	Integer	Position of substring

Table 4: Intrinsic functions

Function	Argument	Result	Explanation
INT	Integer real complex	Integer	Convert to integer
IOR	Integer,integer	Integer	Bitwise logical OR
ISHFT	Integer,shift	Integer	Shift bits by shift
ISHFTC	Integer,shift	Integer	Shift circular bits in argument
KIND	Any intrinsic type	Integer	Value of the kind
LBOUND	Array,dim	Integer	Smallest subscript of dim
LEN	Character	Integer	Number of chars in argument
LEN_TRIM	Character	Integer	Length without trailing space
LGE	A,B	Logical	String A <= string B
LGT	A,B	Logical	String A > string B
LLE	A,B	Logical	String A <= string B
LLT	A,B	Logical	String A < string B
LOG	Real complex	Real complex	Natural logarithm
LOG10	Real	Real	Logarithm base 10
LOGICAL	Logical	Logical	Convert between logical
MATMUL	Matrix,matrix	Vector matrix	Matrix multiplication
MAX	a1,a2,a3,...	Integer real	Maximum value of args
MAXEXPONENT	Real	Integer	Maximum exponent
MAXLOC	Array	Integer vector	Indices in array of max value
MAXVAL	Array,dim,mask	Array element(s)	Maximum value
MERGE	Tsource,Fsource, mask	Tsource or Fsource	Chosen by mask
MIN	a1,a2,a3,...	Integer real	Minimum value
MINEXPONENT	Real	Integer	Minimum exponent
MINLOC	Array	Integer vector	Indices in array of min value
MINVAL	Array,dim,mask	Array element(s)	Minimum value
MOD	a=integer real,p	Integer real	a modulo p
MODULO	a=integer real,p	Integer real	a modulo p
MVBITS	From pos to pos	Integer	Move bits
NEAREST	Real,direction	Real	Nearest value in direction
NINT	Real,kind	Real	Round to nearest integer value
NOT	Integer	Integer	Bitwise logical complement
PACK	Array,mask	VEctor	Vector of array elements
PRECISION	Real complex	Integer	Decimal precision of arg
PRESENT	Argument	Logical	True if optional arg is set
PRODUCT	Array,dim,mask	Integer real complex	Product along dim
RADIX	Integer real	Integer	Radix of integer or real
RANDOM_NUMBER	Harvest = real	Real $0 <= x <= 1$	Subroutine returning a random number in harvest
RANDOM_SEED	Size, put or get	Nothing	Subroutine to set a random number seed
RANGE	Integer real complex	Integer real	Decimal exponent
REAL	Integer real complex	Real	Convert to real type
REPEAT	String,ncopies	String	Concatenate n copies of string

Table 5: Intrinsic functions

Function	Argument	Result	Explanation
RESHAPE	Array,shape,pad,order	Array	Reshape source array to array
RRSPACING	Real	Real	Reciprocal of relative spacing of model
SCALE	Real,integer	Real	Returns $X \cdot b^I$
SCAN	String,set,back	Integer	Position of first of set in string
SELECTED_INT_KIND	Integer	Integer	Kind number to represent digits
SELECTED_REAL_KIND	Integer	Integer	Kind number to represent digits
SET_EXPONENT	Real,integer	Resl	Set an integer as exponent of a real $X \cdot bI - e$
SHAPE	Array	Integer vector	Vector of dimension sizes
SIGN	Integer real,integer real	Integer real	Absolute value of $A \cdot B$
SIN	Real complex	Real complex	Sine of angle in radians
SINH	Real	Real	Hyperbolic sine
SIZE	Array,dim	Integer	Number of array elements in dim
SPACING	Real	Real	Spacing of model number near argument
SPREAD	Source,dim,copies	Array	Adding a dimension to source
SQRT	Real complex	Real complex	Square root
SUM	Array,dim,mask	Integer real complex	Sum of elements
SYSTEM_CLOCK	Count,count,count	Trhough the arguments	Subroutine returning integer data from a real time clock
TAN	Real	Real	Tangent of angle in radians
TANH	Real	Real	Hyperbolic tangent
TINY	Real	Real	Smallest positive model representation
TRANSFER	Source,mold,size	Mold type	Same bits, but new type
TRANSPOSE	Matrix	Matrix	The transpose of matrix
TRIM	String	String	REmove trailing blanks
UBOUND	Array,dim	Integer	Largest subscript of dim in array
UNPACK	Vector,mask,field	Vector type, mask shape	Unpack an array of rank one into an array of mask shape
VERIFY	String,set,back	Integer	Position in string not in set

Table 6: Intrinsic functions