`Fortran 90` Lessons for Computational Chemistry

Curro Pérez-Bernal <francisco.perez@dfaie.uhu.es>

0.0

Abstract

The present document is a basic introduction to the `Fortran` programming language based in several textbooks and references (see 'Referencias' on page 75). It contains the basic scheme of `Fortran` programming taught in the *Computational Chemistry* module (fourth year, second semester) of the University of Huelva Chemistry Degree.

Contents

Chapter 1

Introduction

1.1 Objectives

The main aims of this session consist of:

1 giving a short introduction on programming and programming languages.

2 emphasize the importance of a clear understanding of the problem under study and the use of flow diagrams for achieving structured and clear source code.

3 a brief presentation of the main features of the `Fortran` programming language.

4 installation of the `GNU Fortran` compiler, `gfortran`.

5 Studying two simple codes.

6 Presenting possible sources of information for the interested student.

1.2 Main items.

By default we will use the `emacs` text editor. The first examples are the simple programs '`excode_1_1.f90`' on the next page y '`excode_1_2.f90`' on the following page.

Using the examples the student should be aware of the main sections included in a program::

1 Head of the code with the statement `PROGRAM` *program_name*.

2 Variable definition.

3 Main program body, including `I/O` operations.

4 End of the program: `END PROGRAM` *program_name*.

Things to take into account:

- Importance of remarks and comments. Include many comments in your code, trying to be as clear as possible. `Fortran` remarks are introduced with the character `!`. A correct indentation also improves the code readability. The `emacs` text editor greatly helps in this task.
- The importance of the `IMPLICIT NONE` statement. Declare and initialize properly all variables as in example '`excode_1_2.f90`' on the next page.
- Distinguish the `I/O` operations.

1.3 Example Codes.

1.3.1 excode_1_1.f90

```
PROGRAM ex_1_1
  !
  ! This program reads and displays a string.
  !
  IMPLICIT NONE
  CHARACTER(LEN=50) :: Name
  !
  PRINT *,' Write your name. Do not forget quoting it:'
  PRINT *,' (max 50 characters)'
  READ(*,*), Name
  PRINT *, Name
  !
END PROGRAM ex_1_1
```

1.3.2 excode_1_2.f90

```
PROGRAM ex_1_2
  !
  ! This program reads three numbers and compute their sum and mean value
  !
  IMPLICIT NONE
  REAL :: N1, N2, N3, Average = 0.0, Total = 0.0
  INTEGER :: N = 3
  PRINT *,' Input three numbers (return, coma, or space separated).'
  PRINT *,' '
  READ *,N1,N2,N3
  Total =  N1 + N2 + N3
  Average = Total/N
  PRINT *,'Sum: ',Total
  PRINT *,'Mean value: ',Average
END PROGRAM ex_1_2
```

Chapter 2

Basic Operations

2.1 Objectives

The main aims of this session are:

1 introducing basic `Fortran` syntax rules and the characters allowed in source files.

2 Basic arithmetic operations and operator precedence rules.

3 The `PARAMETER` declaration.

4 Explain the different kinds of numerical variables and its use.

2.2 Main items.

Basic syntax rules:

- Maximum number of characters per line of code: 132.
- Maximum length of a variable name string: 31.
- '&' denotes that the statement continues in the next line. It is added at the end of the broken line[1]
- '!' is the character that marks the rest of the line as a comment.
- ';' is the character that separates several statements in the same line.

Variable names can include the low hyphen ('_') and mix alphanumeric characters and digits, though variable names first character cannot be a number.

`Fortran` character set:

```
A-Z  Letters             0-9  Digits
_    Underscore               Blank
=    Equal               +    Plus
-    Minus               *    Asterisk
/    Slash or oblique    '    Apostrophe
(    Left parenthesis    )    Right parenthesis
,    Comma               .    Period or decimal point
:    Colon               ;    Semicolon
!    Exclamation mark    ""    Quotation mark
%    Percent             &   Ampersand
<    Less than         >   Greater than
```

Precedence of arithmetic operators:

- Operators: {`+`,`-`,`*`,`/`,`**`}.

[1]Except if a string is broken in two lines. In this particular case it is added at the end of the broken line and the beginning of the next line.

- Precedence: (1) `**` (right to left); (2) `*`,`/` (compiler dependent); (3) `+`,`-` (compiler dependent).
- Beware of floating point operations rounding, in particular when mixing different numeric variable types. Minimizing rounding errors is at times a complex and subtle task.

  The compiler transform different type variables to a common type when performing a calculation. The priority ordering, from lower to higher is: `INTEGER`, `REAL`, `DOUBLE PRECISION`, and `COMPLEX`. Therefore, an operation involving an integer and a double precision float is performed transforming the integer value to double precision and the result is given in double precision too. The final result is the transformed to the type of the variable to which is assigned.
- Integer types:

  1 32 bits :: `(2**31)-1 = 2,147,483,647` (~ 10**9)

  2 64 bits :: `(2**63)-1 = 9,223,372,036,854,774,807` (~ 10**19)
- Floats types and precision:

  1 Real 32 bits :: precision = 6-9 ~ `0.3E38 - 1.7E38`

  2 Real 64 bits :: precision = 15-18 ~ `0.5E308 - 0.8E308`
- Making use of the `PARAMETER` modifier in a variable definition we can define constant values in a program. See 'excode_2_4.f90' on page 6.
- (*) Different kinds of floats and integers in `Fortran` and the intrinsic functions[2] `KIND`, `EPSILON`, `PRECISION`, and `HUGE` and how to define a variable in each of the existing types.

  `INTEGER` VARIABLES: if we would like to define an integer variable `i0` that could take values between -999999 y 999999 we should define a variable, called e.g. `ki`, making use of the intrinsic function `SELECTED_INT_KIND()` and make use of it in the variable definition.

  ```
  INTEGER, PARAMETER :: ki =  SELECTED_INT_KIND(6)
  INTEGER(KIND=ki) :: i0
  ```

  The intrinsic function[3] `SELECTED_INT_KIND(X)` output is an integer that indicates the type (*kind*) of an integer variable capable of storing any integer in the range `(-10E+X, 10E+X)` where `X` is also an integer. If we want that any integer constant in our program to be treated with a particular type of integer this can be done as follows:

  ```
  -1234_ki
  2354_ki
  2_ki
  ```

  The error output of the `SELECTED_INT_KIND(X)` function is `-1`.

  Real numbers are more involved. We make use a *floating point representation*, and all the following are valid real numbers in `Fortran`:

  ```
  -10.66E-20
  0.66E10
  1.
  -0.4
  1.32D-44
  2E-12
  3.141592653
  ```

  In this case the statement to control the type of float is `SELECTED_REAL_KIND(p=X,r=Y)`, with two input parameters. The output is an integer associated with a float that complies with the following rules:

  – it has a precision at least equal to `X` and a range of decimal exponents given at least by `Y`. The argument labels are optional.

  – Among various possible results, the one with the minimum decimal precision will be chosen.

  – At least one of the two input parameter should be specified. Both `X` and `Y` are integers. If there is no variable type that fulfills the requested condiotions the output of the function will be `-1` if the precision does not reach the requested level, `-2` if the problem is in the exponent, and `-3` if both requirements cannot be satisfied.

  As an example, if we want to define a real variable called `a0` with 15 digit precision and exponents in the range -306 to 307:

[2] See 'excode_2_4.f90' on page 6.

[3] More info on intrinsic functions and function definition in `Fortran` can be found in 'Objectives' on page 37.

```
INTEGER, PARAMETER :: kr =  SELECTED_REAL_KIND(15,307)
REAL(KIND=kr) :: a0
```

Scalar floats can be addressed defining its particular kind as follows

```
-10.66E-20_kr
0.66E10_kr
142857._kr
-0.4_kr
2E-12_kr
3.141592653_kr
```

Program 'excode_2_5.f90' on page 7 contains several examples of the use of the KIND statement and the default value of KIND for several variable types.

Program 'excode_2_6.f90' on page 7 contains examples of the different types of viariales, how to define them, and how to test them using the intrinsics KIND, DIGITS, EPSILON, TINY, HUGE, EXPONENT, MAXEXPONENT, MINEXPONENT, PRECISION , RADIX y RANGE.

In this program variables are defined using the functions SELECTED_INT_KIND and SELECTED_REAL_KIND This is correct though it is more appropriate to define the variables according to the process in the notes.

The used functions are

1 KIND(x): integer output, type of the variable x.

2 DIGITS(x): integer output, number of significant digits of x.

3 EPSILON(x): if the input x is a float the output is another float, of the same type (*kind*) than x. It is the smallest number of this type such that 1.0 + EPSILON(X) > 1.

4 TINY(x): for float x input the output is of the same kind than x, and it is the minimum positive value that can be defined for such variables.

5 HUGE(x): for float x input the output is of the same kind than x, and it is the maximum positive value that can be defined for such variables.

6 EXPONENT(x): x variable exponent. If x = 0 then EXPONENT(x)=0 too.

7 MAXEXPONENT(x): maximum exponent possible for x type variables.

8 MINEXPONENT(x): minimum exponent possible for x type variables.

9 PRECISION(x): if x is real or complex the output is an integer equal to the number of digits of precision of the variable x.

10 RADIX(x): integer result equal to the radix basis of x.

11 RANGE(x): integer result equal to the range of exponent for the variable x.

- (*) Present how float arithmetic involves precision loss and how an appropriate use of the diferent data types can help to minimize this problem.

2.3 Example Codes.

2.3.1 excode_2_1.f90

```
  PROGRAM ex_2_1
IMPLICIT NONE
!
! Program computing the energy of a vibrational normal mode
!
!   Ge(v) = we (v+1/2) - wexe (v+1/2)^2
!
!
! Definicion de variables
REAL            :: energ_0, energ, delta_e ! deltae = energ-energ0
REAL            :: we = 250.0, wexe = 0.25 ! Units: cm-1
INTEGER         :: v = 0
CHARACTER*60    :: for_mol
! I/O
PRINT *,'Formula de la molecula : '
READ *, for_mol
PRINT *,'Num. de quanta de excitacion : '
READ *, v
! Calculations
energ = we*(v+0.5) - wexe*(v+0.5)**2
energ_0 = we*(0.5) - wexe*(0.5)**2
delta_e = energ - energ_0
```

```
  ! I/O
  PRINT *
  PRINT *,'Especie molecular: ', for_mol
  PRINT *,'num. de quanta: ', v
  PRINT *,'energ = ',energ,'cm-1'
  PRINT *,'energ_0 = ',energ_0,'cm-1'
  PRINT *,'energ - energ_0 = ',delta_e,'cm-1'
END PROGRAM ex_2_1
```

2.3.2 excode_2_2.f90

```
PROGRAM ex_2_2
  IMPLICIT NONE
  REAL :: A,B,C
  INTEGER :: I
  A = 1.5
  B = 2.0
  C = A / B
  I = A / B
  PRINT *
  PRINT *, 'Case (1), Float variable'
  PRINT *,A,'/',B,' = ',C
  PRINT *, 'Case (2), Integer variable'
  PRINT *,A,'/',B,' = ',I
END PROGRAM ex_2_2
```

2.3.3 excode_2_3.f90

```
PROGRAM ex_2_3
  IMPLICIT NONE
  INTEGER :: I,J,K
  REAL :: Answer
  I = 5
  J = 2
  K = 4
  Answer = I / J * K
  PRINT *,'I = ',I
  PRINT *,'J = ',J
  PRINT *,'K = ',K
  PRINT *,'I / J * K = ',Answer
END PROGRAM ex_2_3
```

2.3.4 excode_2_4.f90

```
PROGRAM ex_2_4
  ! Program to compute the time that takes to light to travel
  ! a given distance in AU.
  ! 1 AU = 1,50E11 m
  !
  !Definicion de variables
  IMPLICIT NONE
  ! a_u : astronomic unit in km
  REAL , PARAMETER :: a_u=1.50*10.0**8
  ! y_l : year light --> distance travelled by light during a year
  REAL , PARAMETER :: y_l=9.46*10.0**12
  ! m_l : minute light --> distance travelled by light during a minute
  REAL :: m_l
  ! dist : distance travelled in AUs (INPUT)
  REAL :: dist
  ! t_min : time in minutes needed to travel the distance dist
  REAL :: t_min
  !
  ! min : integer part of t_min
  ! seg : seconds from the decimal digits of t_min
  INTEGER :: min, seg
  !
  m_l = y_l/(365.25 * 24.0 * 60.0) ! m_l Calculation
  !
  PRINT *
  PRINT *,'Distance in AUs'
  READ *, dist
  PRINT *
  !
  t_min = (dist*a_u)/m_l
  min = t_min; seg = (t_min - min) * 60
  !
  PRINT *,' It takes light ' , min,' minutes and ', seg,' seconds'
  Print *,' to travel a distance of ',dist,' AU.'
END PROGRAM ex_2_4
```

2.3.5 excode_2_5.f90

```
PROGRAM ex_2_5
  INTEGER :: i
  REAL :: r
  CHARACTER(LEN=1) :: c
  LOGICAL :: l
  COMPLEX :: cp
  PRINT *,' Integer ',KIND(i)
  PRINT *,' Real    ',KIND(r)
  PRINT *,' Char    ',KIND(c)
  PRINT *,' Logical ',KIND(l)
  PRINT *,' Complex ',KIND(cp)
END PROGRAM ex_2_5
```

2.3.6 excode_2_6.f90

```
PROGRAM ex_2_6
  ! From Program ch0806 of Chivers & Sleightholme
  !
  ! Examples of the use of the kind
  ! function and the numeric inquiry functions
  !
  ! Integer arithmetic
  !
  ! 32 bits is a common word size,
  ! and this leads quite cleanly
  ! to the following
  ! 8 bit integers
  ! -128 to 127 10**2
  ! 16 bit integers
  ! -32768 to 32767 10**4
  ! 32 bit integers
  ! -2147483648 to 2147483647 10**9
  !
  ! 64 bit integers are increasingly available.
  ! This leads to
  ! -9223372036854775808 to
  !  9223372036854775807 10**19
  !
  ! You may need to comment out some of the following
  ! depending on the hardware platform and compiler
  ! that you use.
  INTEGER                         :: I
  INTEGER ( SELECTED_INT_KIND( 2)) :: I1
  INTEGER ( SELECTED_INT_KIND( 4)) :: I2
  INTEGER ( SELECTED_INT_KIND( 8)) :: I3
  INTEGER ( SELECTED_INT_KIND(16)) :: I4
  ! Real arithmetic
  !
  ! 32 and 64 bit reals are normally available.
  !
  ! 32 bit reals 8 bit exponent, 24 bit mantissa
  !
  ! 64 bit reals 11 bit exponent 53 bit mantissa
  !
  REAL :: R = 1.0
  REAL ( SELECTED_REAL_KIND( 6, 37)) :: R1 = 1.0
  REAL ( SELECTED_REAL_KIND(15,307)) :: R2 = 1.0
  REAL ( SELECTED_REAL_KIND(18,310)) :: R3 = 1.0
  PRINT *,' '
  PRINT *,' Integer values'
  PRINT *,'        Kind       Huge'
  PRINT *,' '
  PRINT *,KIND(I ),' ',HUGE(I    )
  PRINT *,' '
  PRINT *,KIND(I1 ),' ',HUGE(I1  )
  PRINT *,KIND(I2 ),' ',HUGE(I2  )
  PRINT *,KIND(I3 ),' ',HUGE(I3  )
  PRINT *,KIND(I4 ),' ',HUGE(I4  )
  PRINT *,' '
  PRINT *,' ----------------------------------- '
  PRINT *,' '
  PRINT *,' Real values'
  !
  PRINT *,'        Kind     ', KIND(R ), '    Digits     ', DIGITS(R )
  PRINT *,'  Huge    = ',HUGE(R ), ' Tiny =', TINY(R)
  PRINT *,'  Epsilon = ',EPSILON(R),' Precision = ', PRECISION(R)
  PRINT *,'  Exponent = ',EXPONENT(R), 'MAXExponent = ', MAXEXPONENT(R), '  MINExponent = ',MINEXPONENT(R)
  PRINT *,'  Radix    = ', RADIX(R ), ' Range =', RANGE(R)
  PRINT *,' '
  !
  !
  PRINT *,'        Kind     ', KIND(R1 ), '    Digits     ', DIGITS(R1 )
  PRINT *,'  Huge    = ',HUGE(R1 ), ' Tiny =', TINY(R1)
  PRINT *,'  Epsilon = ',EPSILON(R1),' Precision = ', PRECISION(R1)
  PRINT *,'  Exponent = ',EXPONENT(R1), 'MAXExponent = ', MAXEXPONENT(R1), '  MINExponent = ',MINEXPONENT(R1)
  PRINT *,'  Radix    = ', RADIX(R1 ), ' Range =', RANGE(R1)
  PRINT *,' '
```

```
  !
  !
  PRINT *,'         Kind     ', KIND(R2 ), '    Digits     ', DIGITS(R2 )
  PRINT *,'  Huge    = ',HUGE(R2 ), ' Tiny =', TINY(R2)
  PRINT *,'  Epsilon = ',EPSILON(R2),' Precision = ', PRECISION(R2)
  PRINT *,'  Exponent = ',EXPONENT(R2), 'MAXExponent = ', MAXEXPONENT(R2), '  MINExponent = ',MINEXPONENT(R2)
  PRINT *,'  Radix    = ', RADIX(R2 ), ' Range =', RANGE(R2)
  PRINT *,' '
  !
  !
  PRINT *,'         Kind     ', KIND(R3 ), '    Digits     ', DIGITS(R3 )
  PRINT *,'  Huge    = ',HUGE(R3 ), ' Tiny =', TINY(R3)
  PRINT *,'  Epsilon = ',EPSILON(R3),' Precision = ', PRECISION(R3)
  PRINT *,'  Exponent = ',EXPONENT(R3), 'MAXExponent = ', MAXEXPONENT(R3), '  MINExponent = ',MINEXPONENT(R3)
  PRINT *,'  Radix    = ', RADIX(R3 ), ' Range =', RANGE(R3)
  PRINT *,' '
  !
END PROGRAM ex_2_6
```

Chapter 3

Introduction to `Fortran` Arrays

3.1 Objectivos

The main aims of this session are the following

1 present one dimension arrays as `Fortran` data structures.

2 present the different ways of defining an array.

3 present the `DO` loop syntax and the implicit `DO` and their use with matrices.

4 explore dynamic arrays in `Fortran 90`

5 present multidimensional arrays as `Fortran` data structures.

3.2 Main items.

Basic Definitions:

1 *rank*: number of indices necessary to indicate unambiguously an array element.

2 *bounds*: max and min values of the indices labelling array elements in each dimension.

3 *extent*: number of elements in an array dimension.

4 *size*: total number of a matrix.

5 *conformal*: two arrays are conformal if both have the same rank and extent.

The following points should be emphasized:

- one dimensional array (vector) definition making use of the `DO` control structure (see '`excode_3_1.f90`' on the next page and exercise 2_1)
- use of the `PARAMETER` declaration for the definition of array bounds in static array declaration.
- initialize before use. Beware of surprises. The initialization to a common constant value is extremely simple: `vec = valor`. A possible alternative is the use of *array constructors*. In the following example, in order to define an integer array with six elements called `vec_int` three possible and equivalent options are given

```
do i = 0, 5
   vec_int(i) = 2*i
enddo

vec_int = (/(2*i, i = 0, 5)/)

vec_int = (/0,2,4,6,8,10/)
```

Last two options involve *array constructors* and can be carried out when the array is declared[1]

- use of the ALLOCATABLE declaration and the use of the ALLOCATE function, as it is shown in example 'excode_3_2.f90' on this page. The ALLOCATE option STAT = *var* allows to chek if the array has been properly defined. See example in program 'Programa ejemplo_9_3.f90' on page 48.
- implicit DO and multidimensional arrays. See example 'excode_3_3.f90' on this page.
- most general form of the DO control structure and possibility of introducing zero or negative array indeces. See example 'excode_3_4.f90' on the next page.
- combination of bash redirectioning with Fortran programs. Necessary for exercise 2, it is explained in 'More on Arrays' on page 13.

3.3 Example Codes.

3.3.1 excode_3_1.f90

```
PROGRAM ex_3_1
!
! VARIABLES DEFINITION
  IMPLICIT NONE
  REAL :: Total=0.0, Average=0.0
  INTEGER, PARAMETER :: Week=7
  REAL , DIMENSION(1:semana) :: Lab_Hours
  INTEGER :: Day
!
  PRINT *,' Labor Time (hours per day during a week):'
  DO Day= 1, Week
     READ *, Lab_Hours(Day)
  ENDDO
!
  DO Day = 1, Week
     Total = Total + Lab_Hours(Day)
  ENDDO
  Average = Total / Week
!
  PRINT *,' Average Weekly Workload: '
  PRINT *, Average, ' hours'
END PROGRAM ex_3_1
```

3.3.2 excode_3_2.f90

```
PROGRAM ex_3_2
  !
  ! VARIABLE DEFINITION
  IMPLICIT NONE
  REAL :: Total=0.0, Average=0.0
  REAL , DIMENSION(:), ALLOCATABLE :: Lab_Hours
  INTEGER :: Day, Number_Days
  !
  PRINT *,' Number of workdays:'
  READ *, Number_Days
  !
  ALLOCATE(Lab_Hours(1:Number_Days))
  !
  PRINT *,' Daily hours of work in ', Number_Days, ' days.'
  DO Day = 1, Number_Days
     READ *, Lab_Hours(Day)
  ENDDO
  !
  DO Day=1, Number_Days
     Total = Total + Lab_Hours(Day)
  ENDDO
  Average = Total / Number_Days
!
  PRINT *,' Average daily workhours in ',Number_Days, ' days : '
  PRINT *, Average, ' hours'
!
END PROGRAM ex_3_2
```

3.3.3 excode_3_3.f90

```
PROGRAM ATTEND_CONTROL
  IMPLICIT NONE
```

[1]Beware of this feature in functions and subroutines.

```
  INTEGER , PARAMETER :: N_students = 3
  INTEGER , PARAMETER :: N_courses = 3
  INTEGER , PARAMETER :: N_lab = 3
  INTEGER :: student, course, lab
  CHARACTER*2 , DIMENSION(1:N_lab,1:N_courses,1:N_lab) :: attend = 'NO'
  DO student = 1, N_students
     DO course = 1,N_courses
        READ *,(attend(lab,course,student),lab = 1, N_lab)
     ENDDO
  ENDDO
  PRINT *,' Lab attendance : '
  DO student=1, N_students
     PRINT *,' Student = ', student
     DO course = 1,N_courses
        PRINT *,'   Course = ', course, ' : ', (attend(lab,course,student),lab=1,N_lab)
     ENDDO
  ENDDO
END PROGRAM ATTEND_CONTROL
```

3.3.4 excode_3_4.f90

```
PROGRAM ex_3_4
  IMPLICIT NONE
  REAL , DIMENSION(-180:180) :: Time=0
  INTEGER :: Degree, Strip
  REAL :: Value
!
  DO Degree=-165,165,15
     Value=Degree/15
     DO Strip=-7,7
        Time(Degree+Strip)=Value
     ENDDO
  ENDDO
!
  DO Strip=0,7
     Time(-180 + Strip) = -180/15
     Time( 180 - Strip) = 180/15
  ENDDO
!
  DO Degree=-180,180
     PRINT *,Degree,' ',Time(Degree), 12 + Time(Degree)
  END DO
END PROGRAM ex_3_4
```

Chapter 4

More on Arrays

4.1 Objectives

The main aims of this lesson are the following:

1 presenting storage ordering of multidimensional arrays.

2 presenting how to manipulate whole matrices or arrays sections in `Fortran`.

3 matrix definition using the `WHERE` statement.

4.2 Main items.

- Storage ordering

  Multidimensional arrays are stored in memory by `Fortran` in such a way that the first subindex varies faster than the second, that varies faster than the third and so on and so forth. This is known as *column major order*.

  For example, if we define a `4 x 2` matrix as

  `REAL , DIMENSION(1:4,1:2) :: A,`

  the `A` array has eight elements stored into memory as follows

```
A(1,1), A(2,1), A(3,1), A(4,1), A(1,2), A(2,2), A(3,2), A(4,2)
```

  The `A` matrix initialization can be carried out in several ways. Assuming that each element should be initialized with a number equal to the index of the corresponding row, we could use two loops[1]

```
DO I_col = 1, 2
     DO I_row = 1, 4
        A(I_row, I_col) = I_row
     ENDDO
ENDDO
```

  An *array constructor* can also be of help, though the seemingly simple solution

```
A = (/ 1, 2, 3, 4, 1, 2, 3, 4 /)
```

  does not work. The *array constructors* produce vectors and not matrices. The vector defined above is of dimension 8, but not a matrix 4x2. The vector and the array `A` have identical sizes, but are not conformal. The statement `RESHAPE` gives a possible solution. The sytax of this statement is

```
output_array = RESHAPE(array_1, array_2)
```

  Where *array_1* is a matrix that would be reshaped and *array_2* is a vector with the dimensions of the new matrix *output_array*. The total number of elements of *array_1* and *output_array* needs to be identical. In the previous example a correct *array constructor* is

[1]The *column major order* storage makes optimal to run over columns in the inner loop, specially when running with large matrices.

```
A = RESHAPE( (/ 1, 2, 3, 4, 1, 2, 3, 4 /), (/ 4, 2 /) )
```

Another example can be found in code 'excode_4_3.f90' on page 16. The RESHAPE command can be used in the array declaration

```
INTEGER, DIMENSION(1:4,1:2) :: A = &
       RESHAPE( (/ 1, 2, 3, 4, 1, 2, 3, 4 /), (/ 4, 2 /) )
```

The data ordering in storage is specially important in I/O operations. The command

```
PRINT*, A
```

will give as a result

```
A(1,1), A(2,1), A(3,1), A(4,1), A(1,2), A(2,2), A(3,2), A(4,2)
```

It is necessary to take this into account also when making use of the READ statement to fill with values the elements of a multidimensional array: `READ(unit,*) A`. The *implicit* DO statement allow to change the standard reading sequence

```
READ(unit,*) ( ( A(row,col), col = 1, 2 ), row = 1, 4 )
```

- FORTRAN allows to define multidimensional arrays, being seven is the max number of indices. The code 'excode_4_2.f90' on page 16 an array is fully characterized making use of several *inquiry* type functions (see 'excode_8_2.f90' on page 39).
- The usage of whole matrices is a great advantage. Definig floating point vectors V1, V2, V3 y V4 as

```
REAL , DIMENSION(1:21) :: V1, V2
REAL , DIMENSION(-10:10) :: V3
REAL , DIMENSION(0:10) :: V4
```

The following tasks are simply performed using this Fortran 90 feature.

1 Assigning a particular value to the full array:

```
V1 = 0.5
```

2 Equating matrices:

```
V1 = V2
```

Making each V1 element equal to the corresponding element of V2. This is only valid when both matrices are *conformal*. It is also valid

```
V3 = V2
```

but it is *not* valid

```
V1 = V4
```

3 All arithmetic operation for scalars can be also applied to conformal matrices, though they may not be the expected mathematical operations.

```
V1 = V2 + V3
V1 = V2*V3
```

In the first case V1 is the sum of two vectors, but in the second case each V1 element is the product of the corresponding V2 and V3 elements, which is not the scalar product. In the two-dimensional matrices case, if we define

```
REAL , DIMENSION(1:4,1:4) :: A, B, C
```

The following are valid statements in Fortran 90

```
A = A**0.5
C = A + B
C = A * B
```

The last case is not the matrix product but a matrix having each element as the result of the product of the corresponding A annd B elements.

4 A matrix can also be read without a DO loop, as in example 'excode_4_1.f90' on page 16, where also the intrinsic function SUM is presented.

- The definition of array slices is possible using the index syntax `liminf:limsup:step`

```
V1(1:10) = 0.5
B(1,1:4) = 100.0
```

In the first case the first ten elements of the `V1` array take the value `0.5`, while in the second elements in the first row of `B` take the value `100.0`. See example '`excode_4_1.f90`' on the next page.

The most general syntax to define a slice is `lowlimit:upplimit:step`, the first slice element has index *lowlimit*, the last one is less than or equal to *upplimit* and *step* is the index variable increment. The default value of *step* is `step=1`. Examples:

```
V1(:)        ! the whole vector
V1(3:10)     ! elements V1(3), V1(4), ... ,   V1(10)
V1(3:10:1)   !   ""       ""      ""           ""
V1(3:10:2)   !   ""      V1(3), V1(5), ... , V1(9)
V1(m:n)      ! elements V1(m), V1(m+1), ... , V1(n)
V1(9:4:-2)   !   ""      V1(9), V1(7), V1(5)
V1(m:n:-k)   ! elements V1(m), V1(m-k), ... , V1(n)
V1(::2)      !   ""      V1(1), V1(3), ... , V1(21)
V1(m:m)      ! 1 x 1 array
V1(m)        ! Scalar
```

- The assignment of values to an array can be done making use of a *logic mask*, with the `WHERE` statement. The use of the mask allows to select those array elements that should undergo the initialization. If, e.g., we need to compute the square root of the elements of a floating point array called `data_mat` and store them in the array `sq_data_mat`, we can skip the use of loops and conditionals as in the following code

```
DO j_col = 1, dim_2
   DO i_row = 1, dim_1
         IF ( data_mat(i_row, j_col) >= 0.0 ) THEN
              sq_data_mat(i_row, j_col) = SQRT( data_mat(i_row, j_col) )
         ELSE
              sq_data_mat(i_row, j_col) = -99999.0
         ENDIF
   ENDDO
ENDDO
```

The `WHERE` statement greatly simplifies this task. The statement syntax is

```
[name:] WHERE (mask_expr_1)
   ....
   Array assignment block 1
   ....
ELSEWHERE (mask_expr_2) [name]
   ....
   Array assignment block 2
   ....
ELSEWHERE
   ....
   Array assignment block 3
   ....
ENDWHERE [name]
```

where *mask_expr_1* and *mask_expr_2* are boolean arrays conformal with the array being assigned. The previous example is therefore simplified to

```
WHERE ( data_mat >= 0.0 )
            sq_data_mat = SQRT( data_mat )
ELSEWHERE
            sq_data_mat = -99999.0
ENDWHERE
```

- These aspects are treated in the different given examples. Example '`excode_4_3.f90`' on the following page shows how to initialize vectoras and matrices, in the last case making use of the `RESHAPE` statement. The example also introduces the `Fortran` intrinsics `DOT_PRODUCT` (scalar product) and `MATMUL` (matrices product).

  Example '`excode_4_4.f90`' on page 17 exemplifies the use `WHERE` in combination with a logical mask.

  Example '`excode_4_5.f90`' on page 17 stress the fact the the elimination of `DO` loops can sometimes bring surprising results about.

  Example '`excode_4_6.f90`' on page 18 shows how to use the `RESHAPE` statement in the definition of a matrix and how to use slicing in the defined array.

4.3 Example Codes.

4.3.1 excode_4_1.f90

```
PROGRAM ex_4_1
  !
  ! VARIABLE DEFINITION
  IMPLICIT NONE
  REAL :: Total=0.0, Average=0.0
  REAL , DIMENSION(:), ALLOCATABLE :: t_worked
  ! Correction Factor
  REAL :: correction =1.05
  INTEGER :: day, num_days
  !
  PRINT *,' Number of workdays: '
  READ *, num_days
  ! Dynamic storage definition
  ALLOCATE(t_worked(1:num_days))
  !
  PRINT *,' Worked hours per day in ', num_days, ' days.'
  ! I/O
  READ *, t_worked
  !
  t_worked(num_days-1:num_days) = correction*t_worked(num_days-1:num_days)
  !
  DO day=1,num_days
     Total = Total + t_worked(day)
  ENDDO
  Average = Total / num_days
  !
  PRINT *,' Average daily hours of work in ',num_days, ' days : '
  PRINT *, Average
  !
END PROGRAM ex_4_1
```

4.3.2 excode_4_2.f90

```
PROGRAM ex_4_2
  !
  ! Program to characterize an array making use of inquiry functions
  !
  IMPLICIT NONE
  !
  REAL, DIMENSION(:,:), ALLOCATABLE :: X_grid
  INTEGER :: Ierr
  !
  !
  ALLOCATE(X_grid(-20:20,0:50), STAT = Ierr)
  IF (Ierr /= 0) THEN
     STOP 'X_grid allocation failed'
  ENDIF
  !
  WRITE(*, 100) SHAPE(X_grid)
100 FORMAT(1X, "Shape :        ", 7I7)
  !
  WRITE(*, 110) SIZE(X_grid)
110 FORMAT(1X, "Size :         ", I7)
  !
  WRITE(*, 120) LBOUND(X_grid)
120 FORMAT(1X, "Lower bounds : ", 7I6)
  !
  WRITE(*, 130) UBOUND(X_grid)
130 FORMAT(1X, "Upper bounds : ", 7I6)
  !
  DEALLOCATE(X_grid, STAT = Ierr)
  IF (Ierr /= 0) THEN
     STOP 'X_grid deallocation failed'
  ENDIF
  !
END PROGRAM EX_4_2
```

4.3.3 excode_4_3.f90

```
PROGRAM ex_4_3
  !
  ! VARIABLES DEFINITION
  IMPLICIT NONE
  REAL, DIMENSION(1:5) :: VA = (/1.0,1.0,1.0,1.0,1.0/), PMAT
  INTEGER I
  INTEGER, DIMENSION(1:5) :: VB = (/(2*I,I=1,5)/)
  REAL :: PE
  REAL , DIMENSION(1:5,1:5) :: MC
  REAL , DIMENSION(25) :: VC = &
       (/ 0.0,0.0,0.0,0.0,1.0,0.5,2.0,3.2,0.0,0.0, &
          0.0,0.0,0.0,0.0,11.0,0.5,2.3,3.2,0.0,0.0, &
```

```
          1.0,3.0,-2.0,-2.0,-0.6 /)
  ! Scalar Product
  PE = DOT_PRODUCT(VA,VB)
  !
  PRINT *, 'Scalar Product (VA,VB) = ', PE
  !
  ! Product of matrices VAxMC
  ! RESHAPE VC to make it a  5 x 5 matrix
  MC = RESHAPE(VC,(/5,5/))
  PMAT = MATMUL(VA,MC)
  !
  PRINT *, ' VA x MC = ', PMAT(1:5)
  !
END PROGRAM ex_4_3
```

4.3.4 excode_4_4.f90

```
PROGRAM ex_4_4
  IMPLICIT NONE
  REAL , DIMENSION(-180:180) :: Time=0
  INTEGER :: Degree, Strip
  REAL :: Value
  CHARACTER (LEN=1), DIMENSION(-180:180) :: LEW=' '
!
  DO Degree=-165,165,15
     Value=Degree/15
     DO Strip=-7,7
        Time(Degree+Strip)=Value
     ENDDO
  ENDDO
!
  DO Strip=0,7
     Time(-180 + Strip) = -180/15
     Time( 180 - Strip) = 180/15
  ENDDO
!
  DO Degree=-180,180
     PRINT *,Degree,' ',Time(Degree), 12 + Time(Degree)
  END DO
!
  WHERE (Time > 0)
     LEW='E'
  ELSEWHERE (Time < 0)
     LEW='W'
  ENDWHERE
!
  PRINT*, LEW
!
END PROGRAM ex_4_4
```

4.3.5 excode_4_5.f90

```
PROGRAM ex_4_5
  !
  ! VARIABLE DEFINITION
  IMPLICIT NONE
  REAL, DIMENSION(1:7) :: VA = (/1.2,2.3,3.4,4.5,5.6,6.7,7.8/)
  REAL, DIMENSION(1:7) :: VA1 = 0.0, VA2 = 0.0
  INTEGER I
  !
  VA1 = VA
  VA2 = VA
  !
  DO I = 2, 7
     VA1(I) = VA1(I) + VA1(I-1)
  ENDDO
  !
  VA2(2:7) = VA2(2:7) + VA2(1:6)
  !
  ! Previous two operations with VA1 and VA2 seem that
  ! should provide the same result. Which is not the case.
  PRINT*, VA1
  PRINT*, VA2
  !
  ! To obtain the same effect without an explicit DO loop we can do
  ! the following
  VA2 = VA
  VA2(2:7) = (/ (SUM(VA2(1:I)), I = 2,7) /)
  !
  PRINT*, VA1
  PRINT*, VA2
END PROGRAM ex_4_5
```

4.3.6 excode_4_6.f90

```
PROGRAM ex_4_6
  !
  ! DEFINITION OF VARIABLES
  IMPLICIT NONE
  INTEGER, DIMENSION(1:3,1:3) :: A = RESHAPE( (/ 1,2,3,4,5,6,7,8,9 /), (/ 3,3 /) )
  !
  !
  !       1  4  7
  !   A = 2  5  8
  !       3  6  9
  !
  PRINT*, "Matrix Element", A(2,3)
  PRINT*, "Submatrix", A(1:2,2:3)
  PRINT*, "Submatrix", A(::2,::2)
  PRINT*, "Matrix Column", A(:,3)
  PRINT*, "Matrix Row", A(2,:)
  PRINT*, "Full Matrix", A
  PRINT*, "Transposed Matrix", TRANSPOSE(A)
END PROGRAM ex_4_6
```

Chapter 5

Control Structures

5.1 Objectives

The main aims of this session consist of:

1 presenting the different conditional control structures in `Fortran` (*branching*).

2 presenting the different way of building loops in `Fortran` code.

These structures allows the programmer to control the program flow, allowing the conditional execution of statements according to the user input values or the values acquired by variables during the program execution.

It is extremely important to take into account before starting to write code in any programming language that a previous step should be accomplished. In encompasses having a clear idea of the problem, the inputs and outputs, the program structure, breaking complex tasks into simpler subtasks, and the optimal algorithm. A flow diagram can be of great help at this stage.

The division of the problem into simpler and simpler tasks is called *top-down design*. Each subtasks should be coded and checked in an independent manner.

5.2 Main items.

We provide a scheme of the main control structures, strting with conditionals and later of loops.

- Conditionals.

  Depend on the evaluation of boolean expressions for which the following operators are defined:

  - == To be equal to.
  - /= Not to be equal to.
  - > Greater than.
  - < Lesser than.
  - >= Greater or equal than.
  - <= Lesser or equal than.

  There exist also logical operators to combine several logical expressions:

  - `.AND.`
  - `.OR.`
  - `.NOT.`
  - `.EQV.` (Boolean '==' operator)
  - `.NEQV.` (Boolean '/=' operator)

The == and /= shouldn't be used to compare real type variables, due to their nonexact nature. If e.g. `A` and `B` are real variables, the following code is discouraged

```
...
IF (A==B) same = .TRUE.
...
```

The alternative would be to define a tolerance and compare the variables as follows

```
REAL :: TOL = 0.00001
...
IF (ABS(A-B) < TOL) same = .TRUE.
...
```

The possible conditional statements are

1 `IF THEN ENDIF`
The syntax of this conditional statement is

```
.
. code
.
IF (Boolean Expression) THEN
         .
         . code_1
         .
ENDIF
.
. code
.
```

Only if the *Boolean Expression* is true the `code_1` block instructions are executed.
If there is only one statement in the `code_1` block the command can be simplified to a one liner removing the `THEN` and `ENDIF` keywords as follows

```
.
. code
.
IF (Boolean Expression) statement
.
. code
.
```

2 `IF THEN ELSE ENDIF`
The syntax of this conditional statement is

```
.
. code
.
IF (Boolean Expression) THEN
         .
         . code_1
         .
ELSE
         .
         . code_2
         .
ENDIF
.
. code
.
```

If the *Boolean Expression* is true the `code_1` block instructions are executed, if it is false then `code_2` block is run.

3 `IF THEN ELSE IF ENDIF`
The syntax of this conditional statement is

```
.
. code
.
IF (Boolean Expression_1) THEN
         .
         . code_1
         .
ELSE IF (Boolean Expression_2) THEN
         .
         . code_2
         .
ENDIF
.
. code
.
```

In case that the *Boolean Expression_1* is true the `code_1` block instructions are executed, if it is false but *Boolean Expression_2* is true then `code_2` block is run.

4 `IF THEN ELSE IF ELSE ENDIF`

The syntax of this conditional statement is

```
.
. code
.
IF (Boolean Expression_1) THEN
         .
         . code_1
         .
ELSE IF (Boolean Expression_2) THEN
         .
         . code_2
         .
ELSE
         .
         . code_3
         .
ENDIF
.
. code
.
```

In case that the *Boolean Expression_1* is true the `code_1` block instructions are executed, if it is false but *Boolean Expression_2* is true then `code_2` block is run. If both are false then the `code_3` block is run.

5 `SELECT CASE`

The `CASE` statement allows to choose among different options in a clear and efficient way, though it has some limitations.

The syntax of this conditional statement is

```
SELECT CASE (selector)
   CASE (label-1)
      block-1
   CASE (label-2)
      block-2
   CASE (label-3)
      block-3
     ............
   CASE (label-n)
      block-n
   CASE DEFAULT
      block-default
END SELECT
```

The `selector` is either a variable or an expression of the *integer*, *logical*, or *character* type. It cannot be a real or complex number.

The `label-1` ... `label-n` labels have the following syntax

```
value
value_1 : value_2
value_1 :
 : value_1
```

The first one is positive if the selector is equal to `value` and the second if the selector takes a value in the range `value_1` to `value_2`. The third(fourth) is true if the selector has a value larger(less) than `value_1`. The `valor`, `value_1`, and `value_2` should be constants or variables defined with the `PARAMETER` declaration.

The `selector` expression is evaluated first. The result is compared with the values in each one of the labels, running the block of instructions of the first successful comparison. If none of the labels is true the `block-default` is run if it exists.

A simple example:

```
SELECT CASE (I)
     CASE (1)
       PRINT*, "I = 1"
     CASE (2:9)
       PRINT*, "I in [2,9]"
     CASE (10:)
       PRINT*, "I in [10,INF]"
     CASE DEFAULT
       PRINT*, "I is negative"
END SELECT CASE
```

The `SELECT CASE` statement is more elegant than a series of `IF`'s as only one expression controls the access to the different alternatives.

Conditional control structures can be nested in several levels. For the sake of clarity in this case the different levels should be labeled as follows

```
firstif: IF (a == 0) THEN
            PRINT*,  "a is zero"
            secondif: IF (c /= 0) THEN
              PRINT*,  "a is zero and c is not zero"
            ELSE secondif
              PRINT*,  "a and c are zero"
            ENDIF secondif
          ELSEIF (a > 0) THEN firstif
            PRINT*,  "a is positive"
          ELSE firstif
            PRINT*,  "a is negative"
          ENDIF firstif
```

The role of the lables `firstif` and `secondif` is to clarify the source code for the reader. Once a label is included in the `IF` statement, then it has to be present also in the `ENDIF`, while it is optional in the `ELSE` and `ELSEIF`. The number of nested conditionals is unlimited.

The example code '`excode_5_1.f90`' on this page contains the `IF THEN ELSE IF ELSE ENDIF` structure and, apparently, the same task is coped with in example '`excode_5_2.f90`' on the next page with the `CASE` structure.

- Loops

  1 Basic loop: The `DO` statement
  We have been already introduced to the basic `DO` loop:

  ```
  DO Var = initial_value, final_value, increment
         Block of Code
  END DO
  ```

  The variable `Var` changes from `initial_value` to `final_value` adding `increment` each iteration.

  2 The `DO WHILE` loop
  This loop has this structure:

  ```
  DO WHILE (conditional)
        Block of code
  ENDDO
  ```

  In this case the block of code is run until the `conditional` in the head of the block is false. E.g. see example 'Programa ejemplo_5_4.f90' on page 24.

  3 The `REPEAT UNTIL` loop
  This type of loop has the following structure:

  ```
  DO
     Block of code
     #
     IF (conditional) EXIT
  END DO
  ```

  The loop is executed until the conditional is evaluated `True`. This case differs from the previous two in that the code block is run at least once.
  In this case we make use of the `EXIT` statement. When this statement is run into a loop the program leaves inmediately the loop and keeps running from the order following the corresponding `ENDDO`. Another interesting statement when working with loops is `CYCLE`. The execution of the `CYCLE` statement makes the program to return to the beginning of the loop, without running the statements in the loop block between the `CYCLE` statement and the end of the loop.
  As in the conditionals case, nested loops can be labeled. This greatly clarifies the source code and, in particular, allows to indicate to which loop level refers the statements `EXIT` and `CYCLE`. By default, they address the inner loop.
  There is a last statement, worth to mention, the `GOTO` command, though its use is highly discouraged in the moder programming standards.

5.3 Example codes.

5.3.1 `excode_5_1.f90`

```
PROGRAM ex_5_1
  !
  IMPLICIT NONE
  !
  REAL :: Grade
  CHARACTER(LEN = 2), DIMENSION(1:5) :: List_Grades=(/'D ','C ','B ','A ','A+'/)
  INTEGER :: IN
```

```
    ! READ NOTE
    PRINT *, "Student mark??"
    READ *, Grade
    !
    IF (Grade>=0.0.AND.Grade<5.0) THEN
       IN=1
    ELSE IF (Grade>=5.0.AND.Grade<7.0) THEN
       IN=2
    ELSE IF (Grade>=7.0.AND.Grade<9.0) THEN
       IN=3
    ELSE IF (Grade>=9.0.AND.Grade<10.0) THEN
       IN=4
    ELSE IF (Grade==10.0) THEN
       IN=5
    ELSE
       IN=0
    ENDIF
  !
    IF (IN==0) THEN
       PRINT *, "The input : ", Grade," has a wrong value. Only [0,10]"
    ELSE
       PRINT *,  "The student grade is ", LISTNT(IN)
    ENDIF
  !
  END PROGRAM EX_5_1
```

5.3.2 excode_5_2.f90

```
  PROGRAM ex_5_2
    !
    IMPLICIT NONE
    !
    REAL :: Grade
    INTEGER :: Index, Integer_Grade
    CHARACTER(LEN=2), DIMENSION(1:5) ::  List_Grades=(/'D ','C ','B ','A ','A+'/)
    ! READ Grade
    PRINT *, "Nota del estudiante?"
    READ *, Grade
    !
    Integer_Grade = NINT(Grade)
    !
    SELECT CASE (Integer_Grade)
    CASE (0:4)
       Index = 1
    CASE (5,6)
       Index = 2
    CASE (7,8)
       Index = 3
    CASE (9)
       Index = 4
    CASE (10)
       Index = 5
    CASE DEFAULT
       Index = 0
    END SELECT
    !
    IF (Index==0) THEN
       PRINT *, "The input grade : ", Grade," is out of bounds. Only  [0,10]."
    ELSE
       PRINT*, "The student grade is ", List_Grades(Index)
    ENDIF
    !
  100 FORMAT(1X,'LA Grade DEL ALUMNO ES ',F4.1,' (',A3,')')
    !
  END PROGRAM EX_5_2
```

5.3.3 excode_5_3.f90

```
  PROGRAM ex_5_3
    !
    IMPLICIT NONE
    !
    REAL :: PIover2 = ASIN(1.0)
    REAL :: ANGLE1 = 0.0, ANGLE2 = 0.0
    INTEGER :: I
    !
    DO I = 0, 16, 2
       ANGLE1 = I*PIO2/4.0
       !
       WRITE(*,*)
       WRITE(*,*) 'Cos(',I/2,'PI/4) = ',COS(ANGLE1),'; Cos(',I/2,'PI/4) = ',COS(ANGLE2)
       WRITE(*,*) 'Sin(',I/2,'PI/4) = ',SIN(ANGLE1),'; Sin(',I/2,'PI/4) = ',SIN(ANGLE2)
       WRITE(*,*)
       !
       ANGLE2 = ANGLE2 + PIO2/2.0
       !
    ENDDO
```

```
END PROGRAM ex_5_3
```

5.3.4 Programa ejemplo_5_4.f90

```
PROGRAM excode_5_4
  !
  IMPLICIT NONE
  !
  REAL :: X_val = 0.0
  REAL :: X_app = 0.0, X_sum = 0.0
  INTEGER :: I_flag = 1, I_count = 0
  !
  ! Taylor Series:  SIN(X) = X - X^3/3! + X^5/5! - X^7/7! + ...
  WRITE(*,*) "Introduce the angle X (RAD) :"
  READ(*,*) X_val
  !
  I_count = 1
  X_app = X_val
  X_sum = X_val
  !
  PRINT*, '         Order     Approx.    SIN(X)       Approx. - SIN(X)'
  !
  DO WHILE (I_flag == 1)
     !
     PRINT*, I_count, X_app, SIN(X_val), X_app - SIN(X_val)
     !
     X_sum = X_sum*(-1)*X_val*X_val/((I_count*2+1)*(I_count*2))
     X_app = X_app + X_sum
     !
     I_count = I_count + 1
     !
     WRITE(*,*) "STOP? (0 yes, 1 no)"
     READ(*,*) I_flag
     IF (I_flag /= 1 .AND. I_flag /= 0) I_flag = 1
     !
  ENDDO
  !
END PROGRAM excode_5_4
```

Chapter 6

INPUT/OUTPUT (I)

6.1 Objectivos

The main aims of this lesson are the following:

1 present how to make use of the standard `bash` redirection for reading and writing data in `Fortran`.

2 present the `FORMAT` statement, as well as its differents descriptors and its use with the commands `PRINT` and `WRITE`.

3 get a basic knowledge about file handling in `Fortran` with the commands `OPEN`, `CLOSE`, and `WRITE`.

6.2 Main Items.

- `bash` shell redirection

  The standard input and output (`STDIN/STDOUT`) redirection in `bash` with < and > allows a `Fortran` program in a simple and direct way to read from and write to a file.

  As an example, the following commands run from a terminal execute a program called `a.out`. Its output is sent to a file called `output.dat` in the first case. In the second case, the program reads its input from a file called `input.dat`, instead of the standard option, the keyboard. In the third case bot options are combined.

```
a.out > output.dat
a.out < input.dat
a.out <input.dat > output.dat
```

  The assignment number 4 can be quite done quite easily making use of standard redirection.

  The error output (`STDERR`) can be redirected too as follows

```
a.out 2> output.dat
a.out 2>&1 ouput.dat
```

  In the second case `STDERR` and `STDOUT` are merged together in file `output.dat`.

- In order to gain a finer control of the format of input and output statementes the so called *format descriptors* are introduced. We have make use of the default options or free format up to now, indicated with the symbol `*` as in `READ(*,*)`, `READ*`, and `PRINT*`.

  To specify a particular format for the input and output in the above mentioned commands the syntax used is `PRINT` *nlin*, *output_list*, or `READ` *nlin*, *output_list*; where *nlin* is a label driving to a `FORMAT` statement with the necessary descriptors and *output_list* are the constant and variables that will be read or written. It is possible to include directly the descriptors in the statement.

  The format descriptors in `FORTRAN`, due to historical reasons (line printers), treated the first character as a control character. If the first character is

  1 `0` : double spacing.

  2 `1` : new page.

3 `+` : no spacing. Print over the previous line.

4 `blank` : simple spacing.

But this is not anymore true unless you are using a line printer (quite bizarre situation in the XXI century).

The format descriptors can fix the vertical position in a line of text, alter the horizontal position of characters in a line, control the display of integers (`I`), floats (`F` and `E`), strings `A` and logical variables (`L`).

The following symbols are used

1 c : column number

2 d : number of digits after decimal point (real values)

3 m : minimum number of digits displayed

4 n : number of spaces

5 r : times a descriptor is repeated

6 w : number of characters affected by a descriptor

Descriptors in I/O operations

1 Integers: `I`: General form r`I`w

This descriptor indicates that r integer values will be read or written, and they occupy w characters or columns. The number is right justified and if the number of digits is less than the number of spacings the rest of the space is filled with space characters. The example

```
PRINT 100, I, I*I
100 FORMAT(' ',I3, ' squared is ', I6)
```

outputs a space, a three-digit integer, the string '`squared is`' and finish with the square of the variable `I`, with a maximum number of six digits. More examples can be found in '`excode_6_1.f90`' on page 28, where the reader can see the effect of having a number with more digits than the allocated space in the format. In this example we also include the `X` descriptor, such that n`X` includes n space characters in the output, or skip n characters from the input.

Format descriptors can be also included directly in the `PRINT` statement, though the resulting code is generally less readable.

```
PRINT "(' ',I3, ' squared is ', I6)", I, I*I
```

As can be seen in code example '`excode_6_1.f90`' on page 28 we can have an arithmetic overflow in a variable and the solution is shown in example '`excode_6_2.f90`' on page 29.

2 Real values descriptor `F`: General form r`F`$w.d$

Where w is the total number of columns used to fit the number, d the number of figures after the decimal point, and r the number of times this descriptor is applied.

For example if the descriptor is `F7.3` the number will be displayed with three figures after the decimal point and occupies seven spaces. This implies that this format descriptor is valid for numbers between `-99.999` and `999.999`. The truncated decimal part of the number is properly rounded. It may happens that as a result of the truncation the number has more digits than expected. The output will be changed for w asterisk characters (`*`). In source code '`excode_6_3.f90`' on page 29 we face such kind of problems.

3 Real descriptor `E`: General form r`E`$w.d$

Introduces scientific notation. The number that multiplies the power of ten takes values between 0.1 to 1.0. This case differs from the previous one that some space should be devoted to the exponent. In fact, apart from the multiplier, it is needed one character for the sign of the number if it is negative, another character for the decimal point, another one for the `E` symbol (stands for Exponent), and the magnitude and sign of the exponent. Therefore the minimum size in this case is $w = d+7$. Example code '`excode_6_4.f90`' on page 29 is identical to example '`excode_6_3.f90`' on page 29 changing the `F` descriptors to `E`. This change facilitates to work with numbers whose value vary into a big range.

4 Real data descriptor `ES`: general format r`ES`$w.d$

It allows the use of the standard scientific notation, with the factor that multiplies the power of ten taking velues in the range 1.0 to 10.0. Apart from this it is similar to the previous float descriptor.

5 Logical data descriptor `L`: general format r`L`w

Logical or boolean data only take the values `TRUE` or `FALSE` and the output of this descriptor will be a right justified `T` or `F`.

6 Character descriptor A: general format rA or rAw
This format implies that there are r string fields w character wide. If w is missing the string is taken with the same length of the character variable. The example 'excode_6_5.f90' on page 29 shows how this descriptor is used.

7 X descriptor: general format nX
The X descriptor controls horizontal displacement, and it implies that n spaces should be included in the output. You can find an example of this descriptor in source code 'excode_6_5.f90' on page 29.

8 Descriptor T:
El descriptor Tc controla el desplazamiento horizontal e indica que se salte directamente a la columna c.

9 / descriptor:
The / descriptor flush the output buffered and feeds a new line. It does not need to be included between commas.

10 The repetition of a set of descriptors can be easily indicated combining them between parentheses. For example

```
100 FORMAT(1X, I6, I6, F9.3, F9.3, F9.3)
```

can be simplified to

```
100 FORMAT(1X, 3(I6, F9.3))
```

- Fortran allows file manipulation with the commands OPEN, WRITE and CLOSE. Other, more advanced, commands are REWIND and BACKSPACE.

  The OPEN command allows to initiate a file. The simplest instance of this command is

```
OPEN(UNIT=unit_number,FILE='filename')
```

  where the file name and the integer number of the associated unit are indicated. The file is therefore associated to this number for any Read/Write operation. We can write something in this file as follows

```
OPEN(UNIT=33, FILE='program_OUT.dat')
WRITE(UNIT=33,FMT=100) variable_lists
```

  which indicates that the data included in *variable_list* will be written in the file associated with unit number 33, following the format specified in line labeled 100. It is possible to abbreviate the command to WRITE(33,100) or WRITE(33,*) if free format is required. In order to send the data to STDOUT, WRITE(UNIT=6,*format*), WRITE(6,*), WRITE(*,*), or PRINT* are all valid and equivalent commands. Standard input STDIN is associated with unit number 5 or the * symbol[1].

  Once the write process takes place the unit should be closed using the statement CLOSE(UNIT=*unit_number*). In our case

```
CLOSE(UNIT=33)
```

  Example 'excode_6_6.f90' on page 30 shows how data are sent to a file and introduces the intrinsic function CPU_TIME that allows to estimate the cpu time spent in a program and its different sections.

  The OPEN command can be more specific, adding the following arguments:

```
OPEN(UNIT=unit_number,FILE=file_name,STATUS=file_status, ACTION=action_var, IOSTAT=integer_var)
```

  These options control the following aspects:

  1 STATUS=*file_status*
  The constant or variable *file_status* is of character type and can take the following values:
  - 'OLD'
  - 'NEW'
  - 'REPLACE'
  - 'SCRATCH'
  - 'UNKNOWN'

  2 ACTION=*action_var*
  The constant or variable *action_var* is of character type and can have the following forms:
  - 'READ'
  - 'WRITE'

[1] STDERR is associated with unit 0.

  - 'READWRITE'

  By default. archives are opened with both read and write permissions active.

3 `IOSTAT=integer_var`

  The variable *integer_stat* is of integer type and gives feedback about the success of the opening of the file. If the final value is `0` the file has been correctly opened. Any other value indicates a problem.

A complete example will be

```
INTEGER ierr
OPEN(UNIT=33, FILE='input_program.dat', STATUS='OLD', ACTION='READ', IOSTAT=ierr)
```

If we want to create a file to store some data:

```
INTEGER ierr
OPEN(UNIT=33, FILE='output_program.dat', STATUS='NEW', ACTION='WRITE', IOSTAT=ierr)
```

- It is possible some degree of control on the access to the elements stored sequentially using the commands

```
BACKSPACE(UNIT = unit_number)
REWIND(UNIT = unit_number)
```

The `BACKSPACE` statement set the register one line back in the associated file while `REWIND` move back to the first register of the file.

- The default is to open formatted files. Thus, the following two statements are equivalent

```
OPEN(UNIT=33,FILE='file_name')
OPEN(UNIT=33,FILE='file_name',FORM='FORMATTED')
```

Formatted files can be edited and read by the user, but they have a couple of cons. Data storage and reading in formatted files takes longer than in unformatted files and there may be some precision loss in float numbers. In order to write data without format files should be opened including the `FORM='UNFORMATTED'` option:

```
OPEN(UNIT=33,FILE='file_name',FORM='UNFORMATTED')
```

To write in a file declared unformatted the `WRITE` command takes the form

```
WRITE(UNIT=33) variable_list
```

The combination of fortran descriptors and different kinds of loop in a code can be found in the example '`excode_6_7.f90`' on page 30. This program reads a data file (a template of this fila can be found under the program, and can be saved removing the trailing `!` symbols). When the program opens the datafile with `OPEN` it uses the `STATUS = 'OLD'` and `ACTION='READ'` options. It reads the file, skipping some files making use of a `REPEAT UNTIL` loop, until it arrives to a line that provides the number of data pairs in the file[2]. Knowing the number of data pairs the appropriate matrices are allocated and the points are read and saved into vectors `data_X` and `data_Y`, and computes the maximum (minimum) value of `data_X` (`data_Y`) making use of the intrinsic functions `MAXVAL` and `MINVAL` (see 'Objectives' on page 37).

6.3 Example Codes

6.3.1 excode_6_1.f90

```
PROGRAM ex_6_1
  !
  IMPLICIT NONE
  !
  ! Variables
  INTEGER :: i, big=10
  !
  DO i=1,20
     PRINT 100, i, big
     big=big*10
  END DO
  !
  ! Format Statements
100 FORMAT(1X, '10 to the ',I3,2X,'=',2X,I12)
  !
END PROGRAM ex_6_1
```

[2]This is achieved making use of the `IERR = label` option in the `READ` command. opción indica que si se ha producido un error de lectura el programa debe saltar a la línea marcada por *label*.

6.3.2 excode_6_2.f90

```
PROGRAM ex_6_2
  !
  IMPLICIT NONE
  !
  INTEGER, PARAMETER :: Long=SELECTED_INT_KIND(16) ! 64 bits integer
  INTEGER :: i
  INTEGER (KIND=Long) :: big=10
  !
  DO i=1,18
     !
     PRINT 100, i, big
100  FORMAT(1X, '10 to the ', I3, 2X, '=', 2X, I16)
     !
     big=big*10
     !
  END DO
  !
END PROGRAM ex_6_2
```

6.3.3 excode_6_3.f90

```
PROGRAM ex_6_3
  ! Program to produce numeric overflow and underflow
  IMPLICIT NONE
  INTEGER :: I
  REAL     :: small = 1.0
  REAL     :: big   = 1.0
!
  DO i=1,45
     PRINT 100, I, small, big
100  FORMAT(' ',I3,' ',F9.4,' ',F9.4)
     !
     small = small/10.0
     big = big*10.0
     !
  END DO
END PROGRAM ex_6_3
```

6.3.4 excode_6_4.f90

```
PROGRAM ex_6_4
  ! Program to produce numeric overflow and underflow
   IMPLICIT NONE
  INTEGER :: I
  REAL     :: small = 1.0
  REAL     :: big   = 1.0
!
  DO i=1,45
     PRINT 100, I, small, big
100  FORMAT(' ',I3,' ',E10.4,' ',E10.4)
     !
     small = small/10.0
     big = big*10.0
     !
  END DO
END PROGRAM ex_6_4
```

6.3.5 excode_6_5.f90

```
PROGRAM ex_6_5
  ! Program to compute the Body Mass Index (Quetelet Index) according to the formula:
  !    BMI = (weight (kg))/(height^2 (m^2))
  !
  IMPLICIT NONE
  CHARACTER (LEN=25) :: Name
  INTEGER :: height_cm = 0, weight_kg = 0 ! height in cm and weight in kg
  REAL     :: height_m = 0.0 ! height in m units
  REAL     :: BMI ! Body Mass Index
  !
  PRINT*,  'Full Name:'; READ*, Name
  !
  PRINT*, 'Weight (kg)?:'; READ*, weight_kg
  !
  PRINT*, 'Height (cm)?:'; READ*, height_cm
  !
  height_m = height_cm/100.0
  BMI = weight_kg/(height_m**2)
  !
  PRINT 100, Name, BMI, BMI
100  FORMAT(1X,A ' BMI  is ', F10.4,' or ', E10.4)
!
END PROGRAM ex_6_5
```

6.3.6 excode_6_6.f90

```
PROGRAM ex_6_6
!
  IMPLICIT NONE
  INTEGER , PARAMETER :: N=1000000
  INTEGER , DIMENSION(1:N) :: X
  REAL    , DIMENSION(1:N) :: Y
  INTEGER :: I
  REAL :: T
  REAL    , DIMENSION(1:5) :: TP
  CHARACTER*10 :: COMMENT
!
  OPEN(UNIT=10,FILE='/tmp/ex_6_6.txt')
!
  CALL CPU_TIME(T)
!
  TP(1)=T
  COMMENT=' Initial Time : '
  PRINT 100, COMMENT, TP(1)
  !
  DO I=1,N
     X(I)=I
  END DO
  !
  CALL CPU_TIME(T)
  !
  TP(2)=T-TP(1)
  COMMENT = ' Integer vector. Time :  '
  PRINT 100,COMMENT,TP(2)
  !
  Y=REAL(X)
  !
  CALL CPU_TIME(T)
  !
  TP(3)=T-TP(1)-TP(2)
  COMMENT = ' Real vector. Time :  '
  !
  PRINT 100,COMMENT,TP(3)
  !
  DO I=1,N
     WRITE(10,200) X(I)
200  FORMAT(1X,I10)
  END DO
  !
  CALL CPU_TIME(T)
  TP(4)=T-TP(1)-TP(2)-TP(3)
  !
  COMMENT = ' Write Integer vector. Time :  '
  PRINT 100,COMMENT,TP(4)
  !
  DO I=1,N
     WRITE(10,300) Y(I)
300  FORMAT(1X,f10.0)
  END DO
  !
  CALL CPU_TIME(T)
  TP(5)=T-TP(1)-TP(2)-TP(3)-TP(4)
  !
  COMMENT = ' Write Real vector. Time :  '
  PRINT 100,COMMENT,TP(5)
  !
100 FORMAT(1X,A,2X,F7.3)
END PROGRAM ex_6_6
```

6.3.7 excode_6_7.f90

```
PROGRAM ex_6_7
  !
  IMPLICIT NONE
  !
  REAL , DIMENSION(:), ALLOCATABLE :: X_vec, Y_vec ! Data Vectors
  INTEGER :: Index, Ierr, Numpoints = 0
  REAL :: Max_x, Min_y
  CHARACTER(LEN=64) :: Filename
  !
  ! READ FILENAME
  READ(5,*) Filename
  ! OPEN FILE (READONLY)
  OPEN( UNIT=10, FILE=Filename, STATUS='OLD', ACTION='READ' )
  !
  DO
     READ(UNIT=10, FMT=100, ERR=10) Numpoints
     IF (Numpoints /= 0) EXIT
10   READ (UNIT=10, FMT=*) ! JUMP ONE LINE
     CYCLE
  ENDDO
  !
  PRINT*, 'NUMPOINTS = ', Numpoints
```

```
   !
   ! ALLOCATE X, Y VECTORS
   ALLOCATE(X_vec(1:NUMPOINTS), STAT = IERR)
   IF (Ierr /= 0) STOP 'X_vec MEM ALLOCATION FAILED'
   ALLOCATE(Y_vec(1:NUMPOINTS), STAT = IERR)
   IF (Ierr /= 0) STOP 'Y_vec MEM ALLOCATION FAILED'
   !
   DO I = 1, Numpoints
      !
      READ(UNIT=10, FMT=110) X_vec(I), Y_vec(I)
      !
   ENDDO
   !
   Max_x = MAXVAL(X_vec)
   Min_y = MINVAL(Y_vec)
   !
   PRINT*, "MAXIMUM X VALUE = ", Max_x
   PRINT*, "MINIMUM Y VALUE = ", Min_y
   ! DEALLOCATE AND CLOSE FILE
   DEALLOCATE(X_vec, STAT = IERR)
   IF (Ierr /= 0) STOP 'X_vec MEM DEALLOCATION FAILED'
   DEALLOCATE(Y_vec, STAT = IERR)
   IF (Ierr /= 0) STOP 'Y_vec MEM DEALLOCATION FAILED'
   !
   CLOSE(10)
   ! FORMAT STATEMENTS
100 FORMAT(19X,I3)
110 FORMAT(F6.3,1X,F6.3)
   !
END PROGRAM ex_6_7
!# Remark 1
!# Remark 2
!Useless line 1
!Useless line 2
!Number of points = 4
!+1.300;-2.443
!+1.265;-1.453
!+1.345;-8.437
!+1.566;+4.455
!+1.566;+4.455
!+3.566;+7.755
!+1.566;+4.457
!+2.366;+2.454
!+1.566;+4.405
!+0.566;+9.450
!+1.545;+4.465
!+9.566;+6.455
!+1.466;+8.405
!+0.566;+7.055
```

Chapter 7

Input/Output (II)

7.1 Objectives

The main aims of this session consist of:

1 present the use of FORMAT in reading operations.

2 basic techniques about the reading of files in Fortran.

3 present possible alternatives to the standard I/O: *here documents* and the NAMELIST type input.

4 present internal files.

This chapter is very much linked with the previous one, having an emphasis in reading data instead of writing them. We present interesting options for providing input data to a program. Formatted input is seldom used with the keyboard, though it is very important when reading data stored in a file.

7.2 Main items.

- The FORMAT statement acts in a completely equivalent way to the one explained in 'INPUT/OUTPUT (I)' on page 25.
- A useful option of the READ command is IOSTAT. It allows to detect if the read process has reached the end-of-file:

```
READ(UNIT=unit_number, FMT=format_label, IOSTAT=integer_var) variable_list
```

  Thus, if if we read a set of data, e.g. coordinates in space as (var_X,var_Y,var_Z) from a file and we do not know the total number of coordinates included we can proceed as follows

```
num_data = 0
readloop: DO
   !
   READ(UNIT=33, FMT=100, IOSTAT=io_status) var_X, var_Y, var_Z
   !
   ! Check reading
   IF (io_status /= 0) THEN
     ! Error in the input or EOF
     EXIT
   ENDIF
   num_data = num_dat + 1
   !   work with the coordinates
   !
      ......
   !
   ! Format statement
100 FORMAT(1X, 3F25.10)
   !
ENDDO readloop
```

  The integer variable num_data is a counter that indicates the number of points read and the integer io_status check if the reading has been correct.

- The example 'Programa ejemplo_7_1.f90' on the current page presents how to read array slices from a file were students' grades are indicated in rows (students) and columns (subjects).
- A convenient way to convey the input to a Fortran program is making use of a *here documents* from the bash shell. A *here document* is a brief script[1], such that apart from compiling (if necessary) and running the program, the input is given in a way that comments can also be included. Example 'excode_7_2.f90' on the facing page is a program that computes the roots of a second order algebraic equation `y = A*x**2 + B*x + C` and ej_here_file included in 'Script ej_here_file' on the next page, is an application of a *here document*. In order to run this program proceed as follows

  ```
  . ej_here_file
  ```

- The namelist format is quite informative, consisting in a list of values assigned to variables labeled with their names. The command NAMELIST syntax is

  ```
  NAMELIST/var_group_name/ var1 [var2 var3 ... ]
  ```

  This statement define a set of variables assigned to the *var_group_name* and should appear in the program prior to any executable statemnt. The reading of variables included in a NAMELIST is done with a READ statement where, instead of specifying a format with the FMT option, is used the option NML as follows[2]

  ```
  READ(UNIT=unit_number, NML=var_group_name, [...])
  ```

  The NAMELIST file with the variable information must start each line with the "&" character, followed by the variable group name, *var_group_name*, ending the line with the character "/". The values in the file can be in different lines but always between the two mentioned characters.

  Program 'excode_7_3.f90' on the facing page is almost identical to program 'excode_7_2.f90' on the next page but it has been modified to make use of a namelist file, called sec_order.inp, included as 'namelist input file' on the facing page.
- In the example 'excode_7_4.f90' on page 36 you can find an *internal file*, where the I/O takes place in an internal buffer instead than in a file. This is rather handy to treat data of unknowkn format, reading them first in a character variable and treating them later, or to handle data mixing variables of different types, like character and integer. This is the case in the example 'excode_7_4.f90' on page 36 where a series of different numbered files are defined and data saved in them. sucesivamente. In this example the intrinsic function TRIM is used to remove trailing spaces from the variable pref.

7.3 Example Codes

7.3.1 Programa ejemplo_7_1.f90

```
PROGRAM EJEMPLO_7_1
  IMPLICIT NONE
  !Definicion de variables
  INTEGER , PARAMETER :: NROW=5
  INTEGER , PARAMETER :: NCOL=6
  REAL , DIMENSION(1:NROW,1:NCOL)   :: RESULT_EXAMS = 0.0
  REAL , DIMENSION(1:NROW)          :: MEDIA_ESTUD = 0.0
  REAL , DIMENSION(1:NCOL)          :: MEDIA_ASIGN = 0.0
  INTEGER :: R,C
  !
  ! Abrir fichero para lectura
  OPEN(UNIT=20,FILE='notas.dat',STATUS='OLD')
  !
  DO R=1,NROW
     READ(UNIT=20,FMT=100) RESULT_EXAMS(R,1:NCOL),MEDIA_ESTUD(R) ! Lectura de notas y luego de promedio
     100 FORMAT(6(2X,F4.1),2X,F5.2) ! Se leen 6 numeros seguidos y luego un septimo
  ENDDO
  READ (20,*) ! Saltamos una linea con esta orden
  READ (20,110) MEDIA_ASIGN(1:NCOL) !
110 FORMAT(6(2X,F4.1))
!
! IMPRESION DE LAS NOTAS EN LA SALIDA ESTANDAR
  DO R=1,NROW
     PRINT 200, RESULT_EXAMS(R,1:NCOL), MEDIA_ESTUD(R)
200  FORMAT(1X,6(1X,F5.1),' = ',F6.2)
  END DO
  PRINT *,'  ====  ====  ====  ====  ====  ==== '
  PRINT 210, MEDIA_ASIGN(1:NCOL)
210 FORMAT(1X,6(1X,F5.1))
END PROGRAM EJEMPLO_7_1
```

[1]From The Free On-line Dictionary of Computing (8 July 2008) [foldoc]: *script*: A program written in a scripting language.
[2]The NAMELIST format could also be used with the WRITE command to save labeled variables.

7.3.2 excode_7_2.f90

```
PROGRAM ex_7_2
! Second degree equation solver
! y = A*x**2 + B*x + C
  IMPLICIT NONE
  ! Variables
  REAL :: A = 0.0
  REAL :: B = 0.0
  REAL :: C = 0.0
  REAL, DIMENSION(2) :: SOL
  REAL :: TEMP
  INTEGER :: I
  !
  ! Input: A, B, C
  READ*, A
  READ*, B
  READ*, C
  !
  ! Calculations
  TEMP = SQRT(B*B-4.0*A*C)
  !
  SOL(1) = (-B+TEMP)/(2.0*A)
  SOL(2) = (-B-TEMP)/(2.0*A)
  !
  !
  !
  DO I=1, 2
     PRINT 200, I, SOL(I)
200  FORMAT(1X,'SOLUTION ', I2,' = ',F18.6)
  END DO
!
END PROGRAM EX_7_2
```

7.3.3 Script ej_here_file

```
# Compile..
gfortran -o second_order  excode_7_2.f90
# And Run...
./second_order <<eof
2.0     # A
1.0     # B
-4.0    # C
eof
```

7.3.4 excode_7_3.f90

```
PROGRAM ex_7_3
  ! Solving second order algebraic equation
  ! y = A*x**2 + B*x + C
  IMPLICIT NONE
  ! Variables
  REAL :: A = 0.0
  REAL :: B = 0.0
  REAL :: C = 0.0
  REAL, DIMENSION(2) :: SOL
  REAL :: TEMP
  INTEGER :: I
  !
  !     NAMELIST DEFINITION
  NAMELIST/INP0/ A, B, C
  !     NAMELIST FILE
  OPEN(UNIT=10,FILE='sec_order.inp',STATUS='OLD')
  !     Inpot of A, B, C
  READ(10,INP0)
  !
  ! Calculations
  TEMP = SQRT(B*B-4.0*A*C)
  !
  SOL(1) = (-B+TEMP)/(2.0*A)
  SOL(2) = (-B-TEMP)/(2.0*A)
  !
  !
  ! OUTOPUT
  DO I=1, 2
     PRINT 200, I, SOL(I)
200  FORMAT(1X,'SOLUTION ', I2,' = ',F18.6)
  END DO
!
END PROGRAM EX_7_3
```

7.3.5 namelist input file

```
#
```

```
#       INPUT FILE FOR excode_7_3.f90
#
 &INP0 A=2.0, B=1.0, C=-4.0 /
```

7.3.6 excode_7_4.f90

```
PROGRAM ex_7_4
  !
  ! Internal file example
  !
  IMPLICIT NONE
  ! Variables
  REAL :: x_var
  INTEGER :: unit_n, index_X
  CHARACTER(LEN=65) :: filename
  CHARACTER(LEN=56) :: pref
  !
  PRINT*, "Introduce file name preffix: "
  READ(*,*) pref
  !
  DO unit_n = 10, 20
     !
     WRITE(filename, '(A, "_", i2,".dat")') TRIM(pref), unit_n
     OPEN(UNIT = unit_n, FILE = filename, STATUS = "UNKNOWN", ACTION = "WRITE")
     !
     DO index_X = 0, 100
        x_var = REAL(index_X)*0.01
        WRITE(unit_n, '(1X,2ES14.6)') x_var, SIN(REAL(unit_n)*x_var)
     ENDDO
     !
     CLOSE(UNIT = unit_n)
     !
  ENDDO
!
END PROGRAM ex_7_4
```

Chapter 8

Subprograms (I): FUNCTIONS

8.1 Objectives

The main aims of this lesson are the following:

1 presenting the advantages of using functions, subroutines and modules.

2 presenting the function concept in `Fortran`.

3 showing the different types of functions: intrinsic, generic, elemental, transformational, and internal.

4 making possible the definition of new functions by the user.

5 evinving the difference beyween external and internal functions.

The use of subporgrams allows a more structured and efficient programming owing to

- the possibility of developing and testing different subtasks in an independent manner.
- it makes possible to recycle subprograms in different programs, diminishing the necessary time for coding.
- the isolation in different subtasks of possible errors and the minimization of unexpected side effects, due to variable encapsulation.

8.2 Main items.

We first focus in functions and will follow with subroutines and modules.

- General characteristics of functions.

  The main characteristics of a function are:

  - May require the input of one or several arguments.
  - Arguments can take the form of an expression.
  - In general, a function produces a single output, which is a function of the arguments, and this output is of scalar type though in some cases it also can be of an array type.
  - The arguments can be of different types.

  There are more than one hundred predefined functions in `Fortran`, highly tested, and of easy usage. E.g. we need trigonometric functions we can make use of the following:

  - `Y = SIN(X)`
  - `Y = COS(X)`
  - `Y = TAN(X)`

where X and Y are real variables[1]

This predefined functions are called *intrinsic functions*. In this link URL (http://gcc.gnu.org/onlinedocs/gfortran/Intrinsic-Procedures.html#Intrinsic-Procedures) you can find a complet list of the intrinsic functions at your disposal with the gfortran compiler.

In general intrinsic functions are also *generic*, which means that they can admit different argument types, with the exception of the functions LGE, LGT, LLE, and LLT.

- The *elemental* functions may have as an argument both scalars or vectors. The example source codes 'excode_8_1.f90' on the next page and 'excode_8_2.f90' on the facing page show the elemental and generic character of some intrinsic functions. When an elemental function is applied to an array the fucntions is applied to each array element.

- Other type of functions are of *inquiry* type, giving information about the characterictics of an array, e.g. the SIZE and ALLOCATED functions. Examples of the latter are found in 'excode_6_7.f90' on page 30 and 'Programa ejemplo_9_3.f90' on page 48.

  The *transformational* functions transform between different data types, e.g. REAL and TRANSPOSE, or functions that work with time data variables as SYSTEM_CLOCK and DATE_AND_TIME.

- Conversion between data types:
  - REAL(i): integer i is converted to a float. The argument i can be an integer, a double precision real or a complex number.
  - INT(x): transforms the real variable x to an integer, truncating the decimal part. No rounding is performed. The x variable can be a real, double precision real, or a complex variable.
  - The functions that follow allow to transform from real to integer values with an adequate control:
    - CEILING(x): real value x to the minimum integer value larger than or equal than x.
    - FLOOR(x): real value x to the maximum integer value less than or equal than x.
    - NINT(x): round the real value x to the nearest integer.
  - DBLE(a): transforms a to double precision. The argument can be integer, real, or complex.
  - CMPLX(x) or CMPLX(x,y): transform to complex values, where the second argument is the imaginary part.

- Además de las funciones intrínsecas, pueden definirse funciones. La definición de una función implica por una parte la propia definición y la posterior llamada a la función desde un programa. La definición de una función sigue el siguiente esquema:

```
FUNCTION fun_name(argument_list)
  IMPLICIT NONE
  Declaration section (including arguments and fun_name)
  ....
  Local variables declaration
  ....
  fun_name = expr
  RETURN ! Optional
END FUNCTION fun_name
```

  En el ejemplo 'Programa ejemplo_8_3.f90' on the next page se muestra como se define e invoca una función que calcula el máximo común divisor de dos números enteros. Es importante tener en cuenta lo siguiente:

  - En este ejemplo podemos distinguir dos bloques. Un primer bloque con el programa principal y un segundo bloque donde se define la función. De hecho la función puede definirse en un fichero diferente al programa principal, y dar ambos ficheros al compilador para que prepare el programa ejecutable.
  - Es importante tener en cuenta que las variables definidas en la función tienen carácter local respecto a las variables que se definen en el programa.
  - La función en este caso tiene como nombre MCD y su tipo es INTEGER. Por tanto, el programa espera que el valor que dé la función como resultado sea un entero.
  - El atributo INTENT(IN) en la definición de las variables A y B de la función:

```
INTEGER , INTENT(IN) :: A,B
```

    indica que dichas variables son variables de entrada y sus valores no pueden ser modificados por la función.

  Todos los argumentos de una función deben tener este atributo para evitar que inadvertidamente sus valores se modifiquen al evaluar la función.

[1] You should take into account that in Fortran angles are expressed in radian units.

- Es posible definir funciones que sean *internas*, esto es, que se restrinjan a un determinado segmento de código, y no puedan ser llamadas desde otro punto del programa. Para ello se utiliza la orden `CONTAINS` como en los ejemplos 'Programa ejemplo_8_4.f90' on the following page y '`excode_8_5.f90`' on the next page. El primero de estos dos programas define una función con la que calcular la energía de un nivel vibracional teniendo en cuenta la frecuencia *we* y la anarmonicidad *wexe*. El segundo, dado un número entero, calcula los factores primos de dicho número. En este ejemplo '`excode_8_5.f90`' on the following page podemos ver también el uso de un bucle del tipo *REPEAT UNTIL*.

8.3 Example Codes

8.3.1 `excode_8_1.f90`

```
PROGRAM ex_8_1
  IMPLICIT NONE
  ! Variable Definition
  INTEGER, PARAMETER :: Long=SELECTED_REAL_KIND(18,310)
  !
  REAL (KIND=Long), PARAMETER :: DPI = ACOS(-1.0_Long) ! Pi number double precision
  REAL (KIND=Long) :: DANGLE, DANGLERAD
  !
  REAL, PARAMETER :: PI = ACOS(-1.0) ! Pi number single precision
  REAL :: ANGLERAD
  !
  PRINT*, 'ANGLE INPUT (Degrees)'
  READ*, DANGLE
  PRINT*
  ! Transform to  RAD
  DANGLERAD = DPI*DANGLE/180.0_Long
  ANGLERAD  = PI*DANGLE/180.0
  !
  PRINT 20, DANGLE, DANGLERAD
  PRINT 21, DANGLE, ANGLERAD
  PRINT*
  PRINT*
  !
  PRINT 22, DANGLERAD, SIN(DANGLERAD), COS(DANGLERAD), SIN(DANGLERAD)**2+COS(DANGLERAD)**2,&
       1.0_Long-(SIN(DANGLERAD)**2+COS(DANGLERAD)**2)
  PRINT*
  PRINT 22, ANGLERAD, SIN(ANGLERAD), COS(ANGLERAD), SIN(ANGLERAD)**2+COS(ANGLERAD)**2,1.0 - (SIN(ANGLERAD)**2+COS(ANGLERAD)**2)
  !
20 FORMAT (1X, 'An angle of ',F14.8,' degrees = ', F14.8, ' rad. (dp)')
21 FORMAT (1X, 'An angle of ',F14.8,' degrees = ', F14.8, ' rad. (sp)')
22 FORMAT (1X, 'ANGLE ',F14.8,', SIN = ', F13.9, ', COS =',F13.9,/'SIN**2+COS**2 = ', F18.14, ', 1 - SIN**2+COS**2 = ', F18.14)
END PROGRAM EX_8_1
```

8.3.2 `excode_8_2.f90`

```
PROGRAM ex_8_2
  IMPLICIT NONE
  ! VARIABLE DEFINITION
  INTEGER , PARAMETER :: NEL=5
  REAL, PARAMETER :: PI = ACOS(-1.0) ! Pi number
  REAL, DIMENSION(1:NEL)   :: XR = (/ 0.0, PI/2.0, PI, 3.0*PI/2.0, 2.0*PI/)
  INTEGER , DIMENSION(1:NEL):: XI = (/ 0, 1, 2, 3, 4/)
  !
  PRINT*, 'Sin  ', XR, ' = ', SIN(XR)
  PRINT*, 'LOG10  ', XR, ' = ', LOG10(XR)
  PRINT*, 'REAL ', XI, ' = ', REAL(XI)
END PROGRAM ex_8_2
```

8.3.3 Programa ejemplo_8_3.f90

```
PROGRAM ex_8_3
  IMPLICIT NONE
  INTEGER :: I,J,Result
  INTEGER :: MCD
  EXTERNAL MCD
  PRINT *,' INTRODUCE TWO INTEGERS:'
  READ *,I,J
  RESULT = MCD(I,J)
  PRINT *,' THE GREATEST COMMON DIVISOR OF ',I,' AND ',J,' IS ',RESULT
END PROGRAM ex_8_3
!
INTEGER FUNCTION MCD(A,B)
IMPLICIT NONE
INTEGER , INTENT(IN) :: A,B
INTEGER :: Temp
  IF (A < B) THEN
     Temp=A
```

```
    ELSE
       Temp=B
    ENDIF
    DO WHILE ((MOD(A,Temp) /= 0) .OR. (MOD(B,Temp) /=0))
       Temp=Temp-1
    END DO
    MCD=Temp
  END FUNCTION MCD
```

8.3.4 Programa ejemplo_8_4.f90

```
PROGRAM ex_8_4
  IMPLICIT NONE
  ! Internal function example:
  ! E(v) = we (v+1/2) - wexe (v+1/2)**2.
  INTEGER :: V, VMAX
  REAL :: we, wexe, Energy
  PRINT *,' Vmax?:'
  READ *, VMAX
  PRINT *,' we and wexe?'
  READ *,we, wexe
  DO V = 0, VMAX
     Energy = FEN(V)
     PRINT 100, V, Energy
  ENDDO
100 FORMAT(1X,'E(',I3,') = ',F14.6)
CONTAINS
!
  REAL FUNCTION FEN(V)
    IMPLICIT NONE
    INTEGER , INTENT(IN) :: V
    FEN = we*(V+0.5)-wexe*(V+0.5)**2
  END FUNCTION FEN
!
END PROGRAM EX_8_4
```

8.3.5 `excode_8_5.f90`

```
PROGRAM ex_8_5
  !
  ! Simple program to compute the prime divisors of a given integer number.
  !
  IMPLICIT NONE
  INTEGER :: NUMVAL
  INTEGER :: NUM
  !
  READ*, NUMVAL ! input
  !
  DO
     NUM = QUOT(NUMVAL)
     IF (NUM == NUMVAL) THEN
        PRINT*, NUM
        EXIT
     ELSE
        PRINT*, NUMVAL/NUM, NUM
        NUMVAL = NUM
     ENDIF
  ENDDO
  !
CONTAINS
  !
  INTEGER FUNCTION QUOT(NUM1)
    !
    INTEGER, INTENT(IN) :: NUM1
    INTEGER :: I
    !
    QUOT = NUM1
    !
    DO I = 2, NUM1-1
       IF (MOD(NUM1,I) == 0) THEN
          QUOT = NUM1/I
          EXIT
       ENDIF
    ENDDO
    !
  END FUNCTION QUOT
  !
END PROGRAM ex_8_5
```

8.3.6 `excode_8_6.f90`

```
PROGRAM ex_8_6
  !
  ! Program to evaluate a 1D potential function on grid points
```

```
    !
    IMPLICIT NONE
    !
    REAL, DIMENSION(:), ALLOCATABLE :: X_grid, Pot_grid
    !
    REAL :: X_min, X_max, Delta_X
    REAL :: V_0 = 10.0, a_val = 1.0
    INTEGER :: Index, X_dim
    INTEGER :: Ierr
    !
    !
    INTERFACE Potf
       ELEMENTAL FUNCTION Potf(Depth, Inv_length, X)
         !
         IMPLICIT NONE
         !
         REAL, INTENT(IN) :: Depth, Inv_length, X
         REAL :: Potf
         !
       END FUNCTION Potf
    END INTERFACE Potf
    !
    !
    READ(*,*), X_min, X_max, X_dim ! input minimum and maximum values of X and number of points
    !
    ALLOCATE(X_grid(1:X_dim), STAT = Ierr)
    IF (Ierr /= 0) THEN
       STOP 'X_grid allocation failed'
    ENDIF
    !
    ALLOCATE(Pot_grid(1:X_dim), STAT = Ierr)
    IF (Ierr /= 0) THEN
       STOP 'Pot_grid allocation failed'
    ENDIF
    !
    !
    Delta_X = (X_max - X_min)/REAL(X_dim - 1)
    !
    X_grid = (/ (Index, Index = 0 , X_dim - 1 ) /)
    X_grid = X_min + Delta_X*X_grid
    !
    Pot_grid = Potf(V_0, a_val, X_grid)
    !
    DO Index = 1, X_dim
       PRINT*, X_grid, Pot_grid
    ENDDO
    !
    DEALLOCATE(X_grid, STAT = Ierr)
    IF (Ierr /= 0) THEN
       STOP 'X_grid deallocation failed'
    ENDIF
    !
    DEALLOCATE(Pot_grid, STAT = Ierr)
    IF (Ierr /= 0) THEN
       STOP 'Pot_grid deallocation failed'
    ENDIF
    !
    !
  END PROGRAM ex_8_6
  !
  ELEMENTAL FUNCTION Potf(Depth, Inv_length, X)
    !
    IMPLICIT NONE
    !
    REAL, INTENT(IN) :: Depth, Inv_length, X
    !
    REAL :: Potf
    !
    Potf = -Depth/(COSH(Inv_length*X)**2)
    !
  END FUNCTION Potf
```

8.3.7 excode_8_7.f90

```
  PROGRAM ex_8_7
    !
    ! Program to characterize an array making use of inquiry functions
    !
    IMPLICIT NONE
    !
    REAL, DIMENSION(:,:), ALLOCATABLE :: X_grid
    INTEGER :: Ierr
    !
    !
    ALLOCATE(X_grid(-20:20,0:50), STAT = Ierr)
    IF (Ierr /= 0) THEN
       STOP 'X_grid allocation failed'
    ENDIF
    !
    WRITE(*, 100) SHAPE(X_grid)
```

```
100 FORMAT(1X, "Shape :        ", 7I7)
  !
  WRITE(*, 110) SIZE(X_grid)
110 FORMAT(1X, "Size :         ", I7)
  !
  WRITE(*, 120) LBOUND(X_grid)
120 FORMAT(1X, "Lower bounds : ", 7I6)
  !
  WRITE(*, 130) UBOUND(X_grid)
130 FORMAT(1X, "Upper bounds : ", 7I6)
  !
  DEALLOCATE(X_grid, STAT = Ierr)
  IF (Ierr /= 0) THEN
     STOP 'X_grid deallocation failed'
  ENDIF
  !
END PROGRAM ex_8_7
```

Chapter 9

Subprogramas (II): subrutinas

9.1 Objetivos

Los objetivos de esta clase son los siguientes:

1 Considerar la diferencia entre funciones y subrutinas y por qué son precisas estas últimas.

2 Introducir los conceptos e ideas más útiles en la definición de subrutinas.

3 Argumentos de una subrutina.

4 Los comandos `CALL` e `INTERFACE`.

5 Alcance (*scope*) de las variables.

6 Variables locales y el atributo `SAVE`

7 Diferentes formas de transmitir matrices como argumentos a una subrutina.

8 Definición de matrices automáticas.

9.2 Puntos destacables.

1 El uso de subrutinas favorece una programación estructurada, mediante la definición de subtareas y su realización en las correspondientes subrutinas y evitando con su uso la duplicación innecesaria de código. Además hacen posible el uso de una extensa colección de librerías o bibliotecas de subrutinas programadas y extensamente probadas para una enorme cantidad de posibles aplicaciones.

2 Para explicar este punto vamos a usar un ejemplo práctico, como es el de la solución de una ecuación de segundo grado. Una posible forma de dividir este programa en subtareas es la siguiente:

   1 Programa principal.

   2 Input de los coeficientes de la ecuación por el usuario.

   3 Solución de la ecuación.

   4 Impresión de las soluciones.

   El programa 'Programa ejemplo_9_1.f90' on page 46 se ajusta a este esquema usando dos subrutinas, llamadas `Interact` y `Solve`.

3 La definición de una subrutina tiene la siguiente estructura:

```
SUBROUTINE nombre_subrutina(lista de argumentos [opcional])
         IMPLICIT NONE
         Arguments (dummy variables) definition (INTENT)
         ...
         Local variables definition
         ...
         Execution Section
         ...
         [RETURN]
 END SUBROUTINE nombre_subrutina
```

Los argumentos se denominan *dummy arguments* porque su definición no implica la asignación de memoria alguna. Esta asignación se llevará a cabo de acuerdo con los valores que tomen los argumentos cuando se llame a la subrutina.

Cuando el compilador genera el ejecutable cada subrutina se compila de forma separada lo que permite el uso de *variables locales* con el mismo nombre en diferentes subrutinas, ya que cada subrutina tiene su particular alcance (*scope*).

En el programa 'Programa ejemplo_9_1.f90' on page 46 se ve como este esquema se repite para las dos subrutinas empleadas.

4 Para invocar una subrutina se emplea el comando `CALL` de acuerdo con el esquema

```
CALL nombre_subroutina(argumentos [opcional])
```

Tras la ejecución de la subrutina invocada con la orden `CALL`, el flujo del programa retorna a la unidad de programa en la que se ha invocado a la subrutina y continúa en la orden siguiente al comando en el que se ha llamado la subrutina con `CALL`. Desde la subrutina se devuelve la ejecución con el comando `RETURN`. Si la subrutina llega a su fin también se devuelve el control al programa que la ha invocado, por lo que generalmente no se incluye el comando `RETURN` justo antes de `END SUBROUTINE`. Si es posible, las subrutinas deberían tener un solo punto de salida.

5 La subrutina y el programa principal se comunican a través de los argumentos (también llamados parámetros) de la subrutina. En la definición de la subrutina dichos argumentos son *dummies*, encerrados entre paréntesis y separados con comas tras el nombre de la subrutina. Dichos argumentos tienen un tipo asociado, pero *NO* se reserva ningún espacio para ellos en memoria. Por ejemplo, los argumentos `E`, `F` y `G` de la subrutina `Solve` en el ejemplo 'Programa ejemplo_9_1.f90' on page 46 son del tipo `REAL`, pero no se reserva para ellos ningún espacio en memoria. Cuando la subrutina es invocada con el comando `CALL Solve(P,Q,R,Root1,Root2,IFail)` entonces los argumentos `E`, `F` y `G` pasan a ser reemplazados por unos punteros a las variables `P`, `Q` y `R`. Por tanto es muy importante que el tipo de los argumentos y el de las variables por las que se ven reemplazados coincidan, ya que cuando esto no sucede se producen frecuentes errores.

6 Alguno de los argumentos proporcionan una información de entrada (input) a la subrutina, mientras que otros proporcionan la salida de la subrutina (output). Por último, también es posible que los argumentos sean simultáneamente de entrada y salida.

Aquellos parámetros que solo sean de entrada es conveniente definirlos con el atributo `INTENT(IN)`. Este atributo ya lo vimos en 'Subprograms (I): FUNCTIONS' on page 37 aplicándolo a funciones. Cuando un argumento posee este atributo el valor de entrada del parámetro se mantiene constante y no puede variar en la ejecución de la subrutina.

Si los parámetros solo son de salida es conveniente definirlos con el atributo `INTENT(OUT)`, para que se ignore el valor de entrada del parámetro y debe dársele uno durante la ejecución de la subrutina.

Si el parámetro tiene el atributo `INTENT(INOUT)`, entonces se considera el valor inicial del parámetro así como su posible modificación en la subrutina.

Hay ejemplos de los tres casos arriba citados en la subrutina `Solve` del ejemplo 'Programa ejemplo_9_1.f90' on page 46. Es muy conveniente etiquetar con el atributo `INTENT` todos los argumentos.

7 De acuerdo con lo anterior es de vital importancia que no exista contradicción entre la declaración de variables en el programa que invoca a la subrutina y en la propia subrutina. Para facilitar este acuerdo entre ambas declaraciones existen los llamados *interface blocks*. En el programa 'Programa ejemplo_9_2.f90' on page 47 podemos ver el programa 'Programa ejemplo_9_1.f90' on page 46 al que se han añadido en el programa principal los *interface blocks* correspondientes a las subrutinas `Interact` y `Solve`.

8 Al igual que en el caso de las funciones, las variables declaradas en una subrutina que no sean parámetros o argumentos de la misma se consideran locales. Por ejemplo, en la subrutina `Interact` del 'Programa ejemplo_9_1.f90' on page 46 la variable `IO_Status` es una variable local de la subrutina.

Generalmente las variables locales se crean al invocarse la subrutina y el valor que adquieren se pierde una vez que la subrutina se ha ejecutado. Sin embargo, usando el atributo `SAVE` es posible salvar el valor que adquiera la variable de una llamada a la subrutina hasta la siguiente llamada. Por ejemplo

```
INTEGER, SAVE:: It = 0
```

El valor que tome en este caso la variable `It` entre llamadas al subprograma en el que se haya declarado se conserva.

Como en el caso de las funciones, es posible hacer que el programa principal "conozca" las variables de las subrutinas que invoque mediante la orden `CONTAINS` y haciendo que de hecho las subrutinas formen parte del programa principal. Esta solución resulta difícil de escalar cuando crece la longitud del problema y no es recomendable.

9 Cuando el argumento de una subrutina no es una variable escalar (del tipo que fuera) sino una matriz (*array*) es necesario dar una información extra acerca de la matriz. El subprograma al que se pasa la matriz ha de conocer el tamaño de la matriz para no acceder a posiciones de memoria erróneas. Para conseguir esto hay tres posibles formas de especificar las dimensiones de una matriz que se halle en la lista de argumentos de una subrutina:

1 *explicit-shape approach*:

En este caso se incluyen como argumentos en la llamada a la subrutina las dimensiones de las matrices implicadas, declarando posteriormente las matrices haciendo uso de dichas dimensiones. Por ejemplo, si en una subrutina llamada `test_pass` se incluye un vector de entrada llamado `space_vec_in` y uno de salida`space_vec_out` con la misma dimensión, si hacemos uso del *explicit-shape approach* la subrutina comenzaría como

```
SUBROUTINE test_pass(space_vec_in, space_vec_out, dim_vec)
         IMPLICIT NONE
         INTEGER, INTENT(IN) :: dim_vec
         REAL, INTENT(IN), DIMENSION(1:dim_vec) :: space_vec_in
         REAL, INTENT(OUT), DIMENSION(1:dim_vec) :: space_vec_out
         ......
 END SUBROUTINE test_pass
```

2 *assumed-shape approach*:

En este caso es necesario incluir el correspondiente bloque `INTERFACE` en el subprograma que invoca la subrutina. Como veremos en el ‘Subprogramas (III): módulos’ on page 53 esto se puede evitar incluyendo la subrutina en un módulo.

En el ‘Programa ejemplo_9_3.f90’ on page 48 puede verse un programa en el que se calcula la media, la mediana[1], la varianza y la desviación estándar de un conjunto de números generados aleatoriamente. En el programa hemos marcado algunos de los puntos de interés que queremos explicar con detalle.

- *(1-3)* Hemos definido la matriz con dimensión variable, de forma que se dimensione mediante una orden `ALLOCATE`. En la orden que dimensiona a la matriz se indica que es un vector (`DIMENSION(:)`) y del mismo modo se hace en el *interface block*. El uso del *interface block* es recomendable, y en casos como este, con matrices definidas de este modo, resulta obligatorio. La orden *(3)*, `ALLOCATE(X(1:N), STAT = IERR)` hace que `X` pase a ser un vector N-dimensional. Usamos también el campo opcional `STAT` que nos permita saber si se ha podido dimensionar el arreglo solicitado. Solo si la salida (`IERR`) es cero la matriz se ha creado sin problemas. El uso de esta opción debe generalizarse.

```
REAL , ALLOCATABLE , DIMENSION(:) :: X  !! (1)

...

INTERFACE
   SUBROUTINE STATS(X,N,MEAN,STD_DEV,MEDIAN)
     IMPLICIT NONE

     ...

     REAL      , INTENT(IN) , DIMENSION(:)   ::    X  !!  (1)

     ...

   END SUBROUTINE STATS
END INTERFACE
```

  Es importante tener en cuenta que se puede definir como `ALLOCATABLE` el argumento con el que se llama a una subrutina, así como a variables internas o locales de la subrutina, pero una variable *dummy* no puede tener este atributo.
  A diferencia de en `FORTRAN77`, la forma recomendada de transmitir arreglos de datos entre un programa y una subrutina es como en el ejemplo, usando *assumed shape arguments* en los que no se da ninguna información acerca del tamaño del arreglo. Sí deben coincidir ambas variables en tipo, rango y clase (`KIND`).

- *(4)* y *(6)*: En estas órdenes se aprovecha la capacidad de `Fortran90` para trabajar con arreglos de variables, ya sean estos vectores o matrices. Por ejemplo, el comando `X=X*1000` multiplica todas las componentes del vector `X` por un escalar y el comando `SUMXI=SUM(X)` aprovecha la función `SUM` para sumar las componentes del vector. En estilo `Fortran 77` estas operaciones conllevarían un bucle `DO`, por ejemplo

```
SUMXI = 0.0
DO I = 1, N
   SUMXI = SUMXI + X(I)
ENDDO
```

- *(5)* En esta parte del programa se libera la memoria reservada para el vector `X` usando el comando `DEALLOCATE`. Este paso no es obligatorio en este programa, pero sí cuando la matriz del tipo `ALLOCATE` se ha definido en una función o subrutina y no tiene el atributo `SAVE`.

[1]Se define la *mediana* de un conjunto de números como aquel valor de la lista tal que la mitad de los valores sean inferiores a él y la otra mitad sean superiores. Coincide con el valor medio en distribuciones simétricas. Para su cálculo es preciso ordenar previamente la lista de números.

- *(7)* Aquí se aprovecha el comando `CONTAINS` para hacer que la subrutina de ordenamiento `SELECTION`, que como puede verse no posee argumentos, *conozca* las misma variables que la subrutina `STATS`, en la que está contenida. Por ello, en la subrutina `SELECTION` solo es preciso definir las variables locales. Esta subrutina se encarga de ordenar la lista de números según un algoritmo que consiste en buscar el número más pequeño de la lista y hacerlo el primer miembro. Se busca a continuación el más pequeño de los restantes que pasa a ser segundo, y así prosigue hasta tener ordenada la lista de números.

La definición de bloques `INTERFACE` se facilita con el uso de módulos, que describimos en la siguiente unidad.

3 *assumed-size approach*

En este caso no se da información a la subrutina acerca de las dimensiones de la matriz, es fácil caer en errores de difícil diagnóstico y se desaconseja su uso.

10 Arreglos multidimensionales. El 'Programa ejemplo_9_5.f90' on page 50 es un ejemplo de como pasar como argumentos arreglos multidimensionales como *assumed shape arrays*. En él, tras que el usuario defina dos matrices, `A` y `B`, el programa calcula la matriz `C` solución del producto `AB` y tras ello calcula la matriz traspuesta de `A`. Se hace uso de las funciones de `Fortran 90` `MATMUL` y `TRANSPOSE`.

11 En las subrutinas pueden dimensionarse *automatic arrays*, que pueden depender de los argumentos de la subrutina. Estos arreglos son locales a la subrutina, no pueden tener el argumento `SAVE` y se crean cada vez que se invoca la subrutina, siendo destruidos al salir de ella. Esto hace que si no hay memoria suficiente para dimensionar el arreglo el programa no funcione. Para evitar esto deben definirse arreglos no automáticos, del tipo `ALLOCATABLE`.

12 Al pasar como argumento una variable de tipo `CHARACTER` dicho argumento se declar con una longitud `LEN = *` y cuando se llame a la subrutina la longitud de la variable pasa a ser la longitud de la variable en la llamada.

El 'Programa ejemplo_9_4.f90' on page 49 muestra un programa en el que, al darle el nombre de un fichero y el número de datos almacenados en dicho fichero; el programa abre el fichero y lee dos columnas de valores que almacena en los vectores `X` e `Y`. En estos casos, dado que el tamaño de la variable `CHARACTER` es variable, es preciso usar un *interface block*.

El 'Programa ejemplo_9_6.f90' on page 50 es un ejemplo donde se construyen dos vectores de números aleatorios de dimensión definida por el usuario usando el método *Box-Mueller*. Para ello se definen dos matrices de tipo `ALLOCATABLE`, `X` e `Y`, y en la subrutina interna `BOX_MULLER` se definen dos vectores de tipo automático: `RANDOM_u` y `RANDOM_v`.

Para calcular el valor medio, la desviación estándar y la mediana de los vectores `X` e `Y` se hace uso de la subrutina `STATS` del 'Programa ejemplo_9_3.f90' on page 48. Se incluye el necesario `INTERFACE` en el programa principal y la subrutina se debe compilar en un fichero por separado. 'Programa ejemplo_9_6.f90' on page 50

13 Sí es importante tener en cuenta que en el caso que se transfiera un *array* usando *assumed shape arguments* como en los ejemplos, el primer índice de la variable en la subrutina *se supone que comienza con el valor `1`, a menos que explícitamente se indique lo contrario*. En el ejemplo 'Programa ejemplo_9_7.f90' on page 51 se muestra un caso simple donde es necesario indicar el índice inicial del vector cuando este no es cero. En este programa se calcula el factorial de los enteros entre `IMIN` e `IMAX` y se almacenan en un vector real. Se puede compilar y correr el programa haciendo `IMIN = 1` e `IMIN = 0` con y sin la definición del índice inicial en la subrutina, para ver la diferencia en las salidas.

9.3 Programas usados como ejemplo.

9.3.1 Programa ejemplo_9_1.f90

```
PROGRAM ejemplo_9_1
!
IMPLICIT NONE
! Ejemplo simple de un programa con dos subrutinas.
! subrutina (1):: Interact :: Obtiente los coeficientes de la ec. de seg. grado.
! subrutina (2):: Solve :: Resuelve la ec. de seg. grado.
!
! Definicion de variables
REAL :: P, Q, R, Root1, Root2
INTEGER :: IFail=0
LOGICAL :: OK=.TRUE.
!
   CALL Interact(P,Q,R,OK) ! Subrutina (1)
!
   IF (OK) THEN
!
      CALL Solve(P,Q,R,Root1,Root2,IFail) ! Subrutina (2)
!
      IF (IFail == 1) THEN
         PRINT *,' Complex roots'
         PRINT *,' calculation aborted'
```

```
      ELSE
         PRINT *,' Roots are ',Root1,' ',Root2
      ENDIF
!
   ELSE
!
      PRINT*,' Error in data input program ends'
!
   ENDIF
!
END PROGRAM ejemplo_9_1
!
!
SUBROUTINE Interact(A,B,C,OK)
   IMPLICIT NONE
   REAL , INTENT(OUT) :: A
   REAL , INTENT(OUT) :: B
   REAL , INTENT(OUT) :: C
   LOGICAL , INTENT(OUT) :: OK
   INTEGER :: IO_Status=0
   PRINT*,' Type in the coefficients A, B AND C'
   READ(UNIT=*,FMT=*,IOSTAT=IO_Status)A,B,C
   IF (IO_Status == 0) THEN
      OK=.TRUE.
   ELSE
      OK=.FALSE.
   ENDIF
END SUBROUTINE Interact
!
!
SUBROUTINE Solve(E,F,G,Root1,Root2,IFail)
   IMPLICIT NONE
   REAL , INTENT(IN) :: E
   REAL , INTENT(IN) :: F
   REAL , INTENT(IN) :: G
   REAL , INTENT(OUT) :: Root1
   REAL , INTENT(OUT) :: Root2
   INTEGER , INTENT(INOUT) :: IFail
! Local variables
   REAL :: Term
   REAL :: A2
   Term = F*F - 4.*E*G
   A2 = E*2.0
! if term < 0, roots are complex
   IF(Term < 0.0)THEN
      IFail=1
   ELSE
      Term = SQRT(Term)
      Root1 = (-F+Term)/A2
      Root2 = (-F-Term)/A2
   ENDIF
END SUBROUTINE Solve
```

9.3.2 Programa ejemplo_9_2.f90

```
PROGRAM ejemplo_9_2
  !
  IMPLICIT NONE
  ! Ejemplo simple de un programa con dos subrutinas.
  ! subrutina (1):: Interact :: Obtiente los coeficientes de la ec. de seg. grado.
  ! subrutina (2):: Solve :: Resuelve la ec. de seg. grado.
  !
  ! Interface blocks
  INTERFACE
     SUBROUTINE Interact(A,B,C,OK)
       IMPLICIT NONE
       REAL , INTENT(OUT) ::  A
       REAL , INTENT(OUT) ::  B
       REAL , INTENT(OUT) ::  C
       LOGICAL , INTENT(OUT) :: OK
     END SUBROUTINE Interact
     SUBROUTINE Solve(E,F,G,Root1,Root2,IFail)
       IMPLICIT NONE
       REAL , INTENT(IN) :: E
       REAL , INTENT(IN) :: F
       REAL , INTENT(IN) :: G
       REAL , INTENT(OUT) :: Root1
       REAL , INTENT(OUT) :: Root2
       INTEGER , INTENT(INOUT) :: IFail
     END SUBROUTINE Solve
  END INTERFACE
  ! Fin interface blocks
  !
  ! Definicion de variables
  REAL :: P, Q, R, Root1, Root2
  INTEGER :: IFail=0
  LOGICAL :: OK=.TRUE.
  !
  CALL Interact(P,Q,R,OK) ! Subrutina (1)
  !
```

```
    IF (OK) THEN
       !
       CALL Solve(P,Q,R,Root1,Root2,IFail) ! Subrutina (2)
       !
       IF (IFail == 1) THEN
          PRINT *,' Complex roots'
          PRINT *,' calculation aborted'
       ELSE
          PRINT *,' Roots are ',Root1,' ',Root2
       ENDIF
       !
    ELSE
       !
       PRINT*,' Error in data input program ends'
       !
    ENDIF
    !
  END PROGRAM ejemplo_9_2
  !
  !
  SUBROUTINE Interact(A,B,C,OK)
    IMPLICIT NONE
    REAL , INTENT(OUT) :: A
    REAL , INTENT(OUT) :: B
    REAL , INTENT(OUT) :: C
    LOGICAL , INTENT(OUT) :: OK
    INTEGER :: IO_Status=0
    PRINT*,' Type in the coefficients A, B AND C'
    READ(UNIT=*,FMT=*,IOSTAT=IO_Status)A,B,C
    IF (IO_Status == 0) THEN
       OK=.TRUE.
    ELSE
       OK=.FALSE.
    ENDIF
  END SUBROUTINE Interact
  !
  !
  SUBROUTINE Solve(E,F,G,Root1,Root2,IFail)
    IMPLICIT NONE
    REAL , INTENT(IN) :: E
    REAL , INTENT(IN) :: F
    REAL , INTENT(IN) :: G
    REAL , INTENT(OUT) :: Root1
    REAL , INTENT(OUT) :: Root2
    INTEGER , INTENT(INOUT) :: IFail
    ! Local variables
    REAL :: Term
    REAL :: A2
    Term = F*F - 4.*E*G
    A2 = E*2.0
    ! if term < 0, roots are complex
    IF(Term < 0.0)THEN
       IFail=1
    ELSE
       Term = SQRT(Term)
       Root1 = (-F+Term)/A2
       Root2 = (-F-Term)/A2
    ENDIF
  END SUBROUTINE Solve
```

9.3.3 Programa ejemplo_9_3.f90

```
  PROGRAM ejemplo_9_3
    !
    IMPLICIT NONE
    !
    ! Definicion de variables
    INTEGER :: N
    REAL , ALLOCATABLE , DIMENSION(:) :: X  !! (1)
    REAL :: M,SD,MEDIAN
    INTEGER :: IERR
    !
    ! interface block   !! (2)
    INTERFACE
       SUBROUTINE STATS(VECTOR,N,MEAN,STD_DEV,MEDIAN)
         IMPLICIT NONE
         INTEGER , INTENT(IN)                     ::  N
         REAL      , INTENT(IN) , DIMENSION(:)    :: VECTOR  !!  (1)
         REAL      , INTENT(OUT)                  :: MEAN
         REAL      , INTENT(OUT)                  :: STD_DEV
         REAL      , INTENT(OUT)                  :: MEDIAN
       END SUBROUTINE STATS
    END INTERFACE
    PRINT *,' Cuántos valores vas a generar aleatoriamente ?'
    READ(*,*),N
    ALLOCATE(X(1:N), STAT = IERR)     !!  (3)
    IF (IERR /= 0) THEN
       PRINT*, "X allocation request denied."
       STOP
    ENDIF
```

```
  CALL RANDOM_NUMBER(X)
  X=X*1000              !!  (4)
  CALL STATS(X,N,M,SD,MEDIAN)
!
  PRINT *,' MEAN = ',M
  PRINT *,' STANDARD DEVIATION = ',SD
  PRINT *,' MEDIAN IS = ',MEDIAN
!
  IF (ALLOCATED(X)) DEALLOCATE(X, STAT = IERR)   !! (5)
  IF (IERR /= 0) THEN
     PRINT*, "X NON DEALLOCATED!"
     STOP
  ENDIF
END PROGRAM ejemplo_9_3
!
SUBROUTINE STATS(VECTOR,N,MEAN,STD_DEV,MEDIAN)
  IMPLICIT NONE
  ! Defincion de variables
  INTEGER , INTENT(IN)                    ::  N
  REAL      , INTENT(IN) , DIMENSION(:)    ::  VECTOR    !! (1)
  REAL      , INTENT(OUT)                  ::  MEAN
  REAL      , INTENT(OUT)                  ::  STD_DEV
  REAL      , INTENT(OUT)                  ::  MEDIAN
  REAL      , DIMENSION(1:N)              ::  Y
  REAL :: VARIANCE = 0.0
  REAL      :: SUMXI = 0.0, SUMXI2 = 0.0
  !
  SUMXI=SUM(VECTOR)       !! (6)
  SUMXI2=SUM(VECTOR*VECTOR)    !! (6)
  MEAN=SUMXI/N
  VARIANCE=(SUMXI2-SUMXI*SUMXI/N)/(N-1)
  STD_DEV = SQRT(VARIANCE)
  Y=VECTOR
  ! Ordena valores por proceso de seleccion
  CALL SELECTION
  IF (MOD(N,2) == 0) THEN
     MEDIAN=(Y(N/2)+Y((N/2)+1))/2
  ELSE
     MEDIAN=Y((N/2)+1)
  ENDIF
CONTAINS     !! (7)
  SUBROUTINE SELECTION
    IMPLICIT NONE
    INTEGER :: I,J,K
    REAL :: MINIMUM
    DO I=1,N-1
       K=I
       MINIMUM=Y(I)
       DO J=I+1,N
          IF (Y(J) < MINIMUM) THEN
             K=J
             MINIMUM=Y(K)
          END IF
       END DO
       Y(K)=Y(I)
       Y(I)=MINIMUM
    END DO
  END SUBROUTINE SELECTION
END SUBROUTINE STATS
```

9.3.4 Programa ejemplo_9_4.f90

```
PROGRAM ejemplo_9_4
  IMPLICIT NONE
  REAL,DIMENSION(1:100)::A,B
  INTEGER :: Nos,I
  CHARACTER(LEN=32)::Filename
  INTERFACE
     SUBROUTINE Readin(Name,X,Y,N)
       IMPLICIT NONE
       INTEGER , INTENT(IN) :: N
       REAL,DIMENSION(1:N),INTENT(OUT)::X,Y
       CHARACTER (LEN=*),INTENT(IN)::Name
     END SUBROUTINE Readin
  END INTERFACE
  PRINT *,' Type in the name of the data file'
  READ '(A)' , Filename
  PRINT *,' Input the number of items in the file'
  READ(*,*) , Nos
  CALL Readin(Filename,A,B,Nos)
  PRINT * , ' Data read in was'
  DO I=1,Nos
     PRINT *,' ',A(I),' ',B(I)
  ENDDO
END PROGRAM ejemplo_9_4
SUBROUTINE Readin(Name,X,Y,N)
  IMPLICIT NONE
  INTEGER , INTENT(IN) :: N
  REAL,DIMENSION(1:N),INTENT(OUT)::X,Y
  CHARACTER (LEN=*),INTENT(IN)::Name
```

```
    INTEGER::I
    OPEN(UNIT=10,STATUS='OLD',FILE=Name)
    DO I=1,N
       READ(10,*)X(I),Y(I)
    END DO
    CLOSE(UNIT=10)
  END SUBROUTINE Readin
```

9.3.5 Programa ejemplo_9_5.f90

```
  PROGRAM ejemplo_9_5
    IMPLICIT NONE
    REAL , ALLOCATABLE , DIMENSION &
         (:,:)::One,Two,Three,One_T
    INTEGER :: I,N
    INTERFACE
       SUBROUTINE Matrix_bits(A,B,C,A_T)
         IMPLICIT NONE
         REAL, DIMENSION (:,:), INTENT(IN) :: A,B
         REAL, DIMENSION (:,:), INTENT(OUT) :: C,A_T
       END SUBROUTINE Matrix_bits
    END INTERFACE
    PRINT *,'Dimensión de las matrices'
    READ*,N
    ALLOCATE(One(1:N,1:N))
    ALLOCATE(Two(1:N,1:N))
    ALLOCATE(Three(1:N,1:N))
    ALLOCATE(One_T(1:N,1:N))
    DO I=1,N
       PRINT*, 'Fila ', I,' de la primer matriz?'
       READ*,One(I,1:N)
    END DO
    DO I=1,N
       PRINT*, 'Fila ', I,' de la segunda matriz?'
       READ*,Two(I,1:N)
    END DO
    CALL Matrix_bits(One,Two,Three,One_T)
    PRINT*,' Resultado: Matriz Producto:'
    DO I=1,N
       PRINT *,Three(I,1:N)
    END DO
    PRINT *,' Matriz traspuesta A^T:'! Calcula la matriz transpuesta.
    DO I=1,N
       PRINT *,One_T(I,1:N)
    END DO
  END PROGRAM ejemplo_9_5
  !
  SUBROUTINE Matrix_bits(A,B,C,A_T)
    IMPLICIT NONE
    REAL, DIMENSION (:,:), INTENT(IN) :: A,B
    REAL, DIMENSION (:,:), INTENT(OUT) :: C,A_T
    C=MATMUL(A,B)
    A_T=TRANSPOSE(A)
  END SUBROUTINE Matrix_bits
```

9.3.6 Programa ejemplo_9_6.f90

```
  PROGRAM ejemplo_9_6
    !
    IMPLICIT NONE
    !
    INTEGER :: I, IERR
    REAL, DIMENSION(:), ALLOCATABLE :: X, Y
    REAL :: M, SD, MEDIAN
    ! interface block
    INTERFACE
       SUBROUTINE STATS(VECTOR,N,MEAN,STD_DEV,MEDIAN)
         IMPLICIT NONE
         INTEGER , INTENT(IN)                     ::  N
         REAL      , INTENT(IN) , DIMENSION(:)    :: VECTOR
         REAL      , INTENT(OUT)                  :: MEAN
         REAL      , INTENT(OUT)                  :: STD_DEV
         REAL      , INTENT(OUT)                  :: MEDIAN
       END SUBROUTINE STATS
    END INTERFACE
    !
    READ*, I
    !
    ALLOCATE(X(1:I), STAT = IERR)
    IF (IERR /= 0) THEN
       PRINT*, "X allocation request denied."
       STOP
    ENDIF
    !
    ALLOCATE(Y(1:I), STAT = IERR)
    IF (IERR /= 0) THEN
       PRINT*, "Y allocation request denied."
```

```
      STOP
   ENDIF
   !
   CALL BOX_MULLER(I)
   !
   PRINT*, X
   CALL STATS(X,I,M,SD,MEDIAN)
   !
   PRINT *,' MEAN = ',M
   PRINT *,' STANDARD DEVIATION = ',SD
   PRINT *,' MEDIAN IS = ',MEDIAN
   !
   IF (ALLOCATED(X)) DEALLOCATE(X, STAT = IERR)
   IF (IERR /= 0) THEN
      PRINT*, "X NON DEALLOCATED!"
      STOP
   ENDIF
   PRINT*, Y
   CALL STATS(Y,I,M,SD,MEDIAN)
   !
   PRINT *,' MEAN = ',M
   PRINT *,' STANDARD DEVIATION = ',SD
   PRINT *,' MEDIAN IS = ',MEDIAN
   !
   IF (ALLOCATED(Y)) DEALLOCATE(Y, STAT = IERR)
   IF (IERR /= 0) THEN
      PRINT*, "Y NON DEALLOCATED!"
      STOP
   ENDIF
   !
CONTAINS
   !
   SUBROUTINE BOX_MULLER(dim)
     !
     ! Uses the Box-Muller method to create two normally distributed vectors
     !
     INTEGER, INTENT(IN) :: dim
     !
     REAL, PARAMETER :: PI = ACOS(-1.0)
     REAL, DIMENSION(dim) :: RANDOM_u, RANDOM_v ! Automatic arrays
     !
     CALL RANDOM_NUMBER(RANDOM_u)
     CALL RANDOM_NUMBER(RANDOM_v)
     !
     X = SQRT(-2.0*LOG(RANDOM_u))
     Y = X*SIN(2*PI*RANDOM_v)
     X = X*COS(2*PI*RANDOM_v)
     !
   END SUBROUTINE BOX_MULLER
   !
END PROGRAM ejemplo_9_6
```

9.3.7 Programa ejemplo_9_7.f90

```
PROGRAM EJEMPLO_9_7
   !
   IMPLICIT NONE
   !
   INTERFACE
      SUBROUTINE SUBEXAMPLE(IMIN, IMAX, FACT_MAT)
        INTEGER, INTENT(IN) :: IMIN, IMAX
        REAL, DIMENSION(IMIN:), INTENT(OUT) :: FACT_MAT
      END SUBROUTINE SUBEXAMPLE
   END INTERFACE
   !
   REAL, DIMENSION(:), ALLOCATABLE :: FACT_MAT
   INTEGER :: IMIN, IMAX, I
   !
   IMIN = 0
   IMAX = 5
   !
   ALLOCATE(FACT_MAT(IMIN:IMAX))
   !
   PRINT*, "MAIN", SIZE(FACT_MAT)
   !
   CALL SUBEXAMPLE(IMIN, IMAX, FACT_MAT)
   !
   DO I = IMIN, IMAX
      PRINT*, I, FACT_MAT(I)
   ENDDO
   !
END PROGRAM EJEMPLO_9_7
!!!!!!!!!!
SUBROUTINE SUBEXAMPLE(IMIN, IMAX, FACT_MAT)
   !
   IMPLICIT NONE
   INTEGER, intent(in) :: IMIN, IMAX
   REAL, DIMENSION(IMIN:), intent(out) :: FACT_MAT
! The subroutine with the next line only would work for IMIN = 1
!  REAL, DIMENSION(:), intent(out) :: FACT_MAT
```

```
    !
    INTEGER :: j,k
    !
    PRINT*, "SUB", SIZE(FACT_MAT)
    !
    DO j = imin, imax
       fact_mat(j) = 1.0
       do k = 2, j
          fact_mat(j) = k*fact_mat(j)
       enddo
    ENDDO
    !
    !
  END SUBROUTINE SUBEXAMPLE
```

Chapter 10

Subprogramas (III): módulos

10.1 Objetivos

Los objetivos de esta clase son los siguientes:

1. Presentar los módulos y las ventajas que aportan.
2. Uso de módulos para la definición de variables. Reemplazo de bloques `COMMON`.
3. Uso de módulos para la definición de funciones y subrutinas.
4. Definición de variables públicas y privadas en módulos. Visibilidad en el módulo.

10.2 Puntos destacables.

1. La definición de módulos permite escribir código de forma más clara y flexible. En un módulo podemos encontrar

   1. Declaración global de variables.
      Reemplazan a las órdenes `COMMON` e `INCLUDE` de `FORTRAN 77`.
   2. Declaración de bloques `INTERFACE`.
   3. Declaración de funciones y subrutinas. La declaración de funciones y subrutinas en un módulo es conveniente para evitar la inclusión de los correspondientes `INTERFACE`, ya que estos están ya implícitos en el módulo.
   4. Control del acceso a los objetos, lo que permite que ciertos objetos tengan carácter público y otros privado.
   5. Los módulos permiten empaquetar tipos derivados, funciones, subrutinas para proveer de capacidades de programación orientada a objetos. Pueden también usarse para definir extensiones semánticas al lenguaje `FORTRAN`.

   La sintaxis para la declaración de un módulo es la siguiente:

   ```
   MODULE module name
      IMPLICIT NONE
      SAVE
        declaraciones y especificaciones
      [ CONTAINS
        definición de subrutinas y funciones ]
   END MODULE  module name
   ```

   La carga del módulo se hace mediante la orden `USE MODULE module name` que debe preceder al resto de órdenes de la unidad de programa en el que se incluya. Desde un módulo puede llamarse a otro módulo. A continuación desarrollamos brevemente estas ideas.

2. Una de las funciones de los módulos es permitir el intercambio de variables entre diferentes programas y subrutinas sin recurrir a los argumentos. La otra función principal es, haciendo uso de `CONTAINS`, definir funciones, subrutines y bloques `INTERFACE`.

   La inclusión de estas unidades en un módulo hace que todos los detalles acerca de las subrutinas y funciones implicadas sean conocidas para el compilador lo que permite una más rápida detección de errores. Cuando una subrutina o una función se compila en un módulo y se hace accesible mediante `USE MODULE` se dice que tiene una interfaz explícita (*explicit interface*), mientras que en caso contrario se dice que tiene una interfaz implícita (*implicit interface*).

3 La definición de módulos favorece la llamada *encapsulación*, que consiste en definir secciones de código que resultan fácilmente aplicables en diferentes situaciones. En esto consiste la base de la llamada programación orientada a objetos. En el 'Programa ejemplo_10_1.f90' on page 56 presentamos como se define un módulo (usando la orden `MODULE` en vez de `PROGRAM` para la definición de un *stack* de enteros. Es importante tener en cuenta como se definen en el módulo las variables `STACK_POS` y `STORE` con el atributo `SAVE`, para que su valor se conserve entre llamadas. Esto es especialmente importante cuando el módulo se llama desde una subrutina o función en vez de desde el programa principal.

4 Este módulo puede ser accedido por otra unidad de programa que lo cargue usando la orden `USE`. Debe compilarse previamente a la unidad de programa que lo cargue.

```
PROGRAM Uso_Stack
!
USE Stack     ! CARGA EL MODULO
!
IMPLICIT NONE
....
....
CALL POP(23); CAL PUSH(20)
....
....
END PROGRAM Uso_Stack
```

5 Como vemos en el 'Programa ejemplo_10_1.f90' on page 56 las variables dentro de un módulo pueden definirse como variables privadas, con el atributo `PRIVATE`. Esto permite que no se pueda acceder a estas variables desde el código que usa el módulo. El programa que carga el módulo solo puede acceder a las subrutinas `POP` y `PUSH`. La visibilidad por defecto al definir una variable o procedimiento en un módulo es `PUBLIC`. Es posible añadir el atributo a la definición de las variables

```
  INTEGER, PRIVATE, PARAMETER :: STACK_SIZE = 500
  INTEGER, PRIVATE, SAVE :: STORE(STACK_SIZE) = 0, STACK_POS = 0
```

6 En ocasiones es posible que variables o procedimientos definidos en un módulo entren en conflicto con variables del programa que usa el módulo. Para evitar esto existe la posibilidad de renombrar las variables que carga el módulo, aunque esto solo debe hacerse cuando sea estrictamente necesario.

Si, por ejemplo, llamamos al módulo `Stack` desde un programa que ya tiene una variable llamada `PUSH` podemos renombrar el objeto `PUSH` del módulo a `STACK_PUSH` al invocar el módulo

```
USE Stack, STACK_PUSH => PUSH
```

Se pueden renombrar varios objetos, separándolos por comas.

7 Es posible hacer que solo algunos elementos del módulo sean accesibles desde el programa que lo invoca con la cláusula `ONLY`, donde también es posible renombrar los objetos si es necesario. Por ejemplo, con la llamada

```
USE Stack, ONLY: POP, STACK_PUSH => PUSH
```

Solamente se accede a `POP` y `PUSH`, y este último se renombra a `STACK_PUSH`.

8 Para definir variables comunes a diferentes partes de un programa se debe evitar el uso de variables en `COMMON` y, en vez de ello, se siguen los pasos siguientes.

1 Declarar las variables necesarias en un `MODULE`.

2 Otorgar a estas variables el atributo `SAVE`.

3 Cargar este módulo (`USE` *module_name*) desde aquellas unidades que necesiten acceso a estos datos globales.

Por ejemplo, si existen una serie de constantes físicas que utilizaremos en varios programas podemos definirlas en un módulo:

```
MODULE PHYS_CONST
  !
  IMPLICIT NONE
  !
  SAVE
  !
  REAL, PARAMETER :: Light_Speed = 2.99792458E08  ! m/s
  REAL, PARAMETER :: Newton_Ctnt = 6.67428E-11    ! m3 kg-1 s-2
  REAL, PARAMETER :: Planck_Ctnt = 4.13566733E-15 ! eV s
  !
  REAL :: Otra_variable
  !
END MODULE PHYS_CONST
```

En este módulo se definen tres constantes físicas (con el atributo PARAMETER, ya que son constantes) y una cuarta variable a la que se desea acceder que no permanece constante. En cualquier programa, función o subrutina que quieran usarse estas variables basta con cargar el módulo

```
PROGRAM CALCULUS
  !
  USE PHYS_CONST
  !
  IMPLICIT NONE
  !
  REAL DISTANCE, TIME
  !
  ...
  DISTANCE = Light_Speed*TIME
  ...
  !
END PROGRAM CALCULUS
```

9 El 'Programa ejemplo_10_2.f90' on the next page es un programa simple donde se utiliza el módulo para el manejo de un stack presentado para realizar operaciones (adición y substracción) con enteros en notación polaca inversa (RPN, reverse Polish notation).

Esta notación permite no usar paréntesis en las operaciones algebraicas y resulta más rápida que la notación usual. Si, por ejemplo, en el stack existen los números (23, 10, 33) y tenemos en cuenta que un stack se rige por el principio *last in, first out*, tendremos que si introducimos un número más (p.e. 5) y realizamos las operaciones de suma (plus) y substracción (minus) tendremos lo siguiente

```
-       -          -               -
-       23         -               -
23      10         23              -
10      33         10              23
33   ->  5    ->   38  (=33+5)  -> -28  (=10-38)

5       plus       minus
```

Para llevar a cabo esta tarea se carga el módulo Stack en *(1)*. Una vez cargado el módulo podemos acceder a las subrutinas POP y PUSH que nos permiten manejar el stack. En *(2)* comienza el bucle principal, con la etiqueta inloop, que termina cuando el usuario da como input Q, q o quit.

Para controlar este bucle se utiliza una estructura SELECT CASE que comienza en *(3)*. Esta estructura analiza cuatro casos posibles:

- *(4)*: salir del programa
- *(5)*: suma
- *(6)*: resta
- *(7)*: introduce número en el stack (DEFAULT)

En el último caso se transforma la variable de carácter leída en una variable entera para almacenarla en el stack.

Para compilar y correr este programa podemos hacerlo compilando previamente el módulo, si lo hemos salvado en el fichero ejemplo_10_1_Stack.f90

```
$ gfortran -c ejemplo_10_1_Stack.f90
$ gfortran -o ejemplo_10_2 ejemplo_10_2.f90 ejemplo_10_1_Stack.o
```

10 El uso de módulos también permite, de forma flexible, segura y fácil de modificar, controlar la precisión de los números reales (o enteros) en los cálculos que se lleven a cabo. Una posible forma de definir de forma portable la doble precisión es mediante un sencillo módulo, llamado dble_prec. Como vimos en el programa 'excode_2_6.f90' on page 7 los números reales de doble precisión tienen un KIND = 8. Para hacer el código independiente de la plataforma donde compilemos podemos hacer

```
MODULE dble_prec
  IMPLICIT NONE
  INTEGER, PARAMETER :: dbl = KIND(1.0D0)
END MODULE dble_prec
```

Por tanto podemos definir esa precisión cargando este módulo, p.e.

```
PROGRAM TEST_MINUIT
  !
  USE dble_prec
  !
  IMPLICIT NONE
  !
  ! Variable Definition
  REAL(KIND=dbl), PARAMETER :: PI = 4.0_dbl*ATAN(1.0_dbl)
  REAL(KIND=dbl) :: ENERF
   ....
   ....
```

Esto favorece la portabilidad y reduce el riesgo de errores ya que para cambiar la precisión con la que se trabaja solamente es necesario editar el módulo. En el 'Programa ejemplo_10_3.f90' on the next page introducimos esta mejora en el programa 'Programa ejemplo_9_6.f90' on page 50. Se almacena el módulo simple anteriormente descrito en un fichero llamado, p.e., `dble_prec.f90` y se compila previamente:

```
$ gfortran -c dble_prec.f90
$ gfortran -o ejemplo_10_3 ejemplo_10_3.f90 dble_prec.o
```

Un módulo más completo, donde se definen diferentes tipos de enteros y de reales es el dado en el programa 'Programa ejemplo_10_4.f90' on page 59.

En un ejercicio se plantean al alumnos diferentes maneras de mejorar el programa simple 'Programa ejemplo_10_2.f90' on this page.

10.3 Programas usados como ejemplo.

10.3.1 Programa ejemplo_10_1.f90

```
MODULE Stack
  !
  ! MODULE THAT DEFINES A BASIC STACK
  !
  IMPLICIT NONE
  !
  SAVE
  !
  INTEGER, PARAMETER :: STACK_SIZE = 500
  INTEGER :: STORE(STACK_SIZE) = 0, STACK_POS = 0
  !
  PRIVATE :: STORE, STACK_POS, STACK_SIZE
  PUBLIC :: POP, PUSH
  !
  CONTAINS
    !
    SUBROUTINE PUSH(I)
      !
      INTEGER, INTENT(IN) :: I
      !
      IF (STACK_POS < STACK_SIZE) THEN
         !
         STACK_POS = STACK_POS + 1; STORE(STACK_POS) = I
         !
      ELSE
         !
         STOP "FULL STACK ERROR"
         !
      ENDIF
      !
    END SUBROUTINE PUSH
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    SUBROUTINE POP(I)
      !
      INTEGER, INTENT(OUT) :: I
      !
      IF (STACK_POS > 0) THEN
         !
         I = STORE(STACK_POS); STACK_POS = STACK_POS - 1
         !
      ELSE
         !
         STOP "EMPTY STACK ERROR"
         !
      ENDIF
      !
    END SUBROUTINE POP
    !
END MODULE Stack
```

10.3.2 Programa ejemplo_10_2.f90

```
PROGRAM RPN_CALC
  !
  ! SIMPLE INTEGER RPN CALCULATOR (ONLY SUM AND SUBSTRACT)
  !
  USE Stack                  !!        (1)
  !
  IMPLICIT NONE
  !
  INTEGER :: KEYB_DATA
  CHARACTER(LEN=10) :: INPDAT
```

```
  !
  INTEGER :: I, J, K, DATL, NUM, RES
  !
  !
  inloop: DO      !! MAIN LOOP        (2)
     !
     READ 100, INPDAT
     !
     SELECT CASE (INPDAT)   !!        (3)
        !
     CASE ('Q','q')  !! EXIT          (4)
        PRINT*, "End of program"
        EXIT inloop
     CASE ('plus','Plus','PLUS','+')   !! SUM              (5)
        CALL POP(J)
        CALL POP(K)
        RES = K + J
        PRINT 120, K, J, RES
        CALL PUSH(RES)
     CASE ('minus','Minus','MINUS','-')   !! SUBSTRACT        (6)
        CALL POP(J)
        CALL POP(K)
        RES = K - J
        PRINT 130, K, J, RES
        CALL PUSH(RES)
     CASE DEFAULT !! NUMBER TO STACK  (7)
        !
        DATL = LEN_TRIM(INPDAT)
        !
        RES = 0
        DO I = DATL, 1, -1
           NUM = IACHAR(INPDAT(I:I)) - 48
           RES = RES + NUM*10**(DATL-I)
        ENDDO
        !
        PRINT 110, RES
        CALL PUSH(RES)
     END SELECT
     !
  ENDDO inloop
  !
100 FORMAT(A10)
110 FORMAT(1X, I10)
120 FORMAT(1X, I10,' + ', I10,' = ', I20)
130 FORMAT(1X, I10,' - ', I10,' = ', I20)
END PROGRAM RPN_CALC
```

10.3.3 Programa ejemplo_10_3.f90

```
PROGRAM ejemplo_10_3
  !
  USE dble_prec
  !
  IMPLICIT NONE
  !
  INTEGER :: I, IERR
  REAL(KIND=dbl), DIMENSION(:), ALLOCATABLE :: X, Y
  REAL(KIND=dbl) :: M, SD, MEDIAN
  ! interface block
  INTERFACE
     SUBROUTINE STATS(VECTOR,N,MEAN,STD_DEV,MEDIAN)
       !
       USE dble_prec
       !
       IMPLICIT NONE
       INTEGER , INTENT(IN)                    ::  N
       REAL(KIND=dbl)      , INTENT(IN) , DIMENSION(:)   :: VECTOR
       REAL(KIND=dbl)      , INTENT(OUT)                 :: MEAN
       REAL(KIND=dbl)      , INTENT(OUT)                 :: STD_DEV
       REAL(KIND=dbl)      , INTENT(OUT)                 :: MEDIAN
     END SUBROUTINE STATS
  END INTERFACE
  !
  READ*, I
  !
  ALLOCATE(X(1:I), STAT = IERR)
  IF (IERR /= 0) THEN
     PRINT*, "X allocation request denied."
     STOP
  ENDIF
  !
  ALLOCATE(Y(1:I), STAT = IERR)
  IF (IERR /= 0) THEN
     PRINT*, "Y allocation request denied."
     STOP
  ENDIF
  !
  CALL BOX_MULLER(I)
  !
  PRINT*, X
```

```
  CALL STATS(X,I,M,SD,MEDIAN)
  !
  PRINT *,' MEAN = ',M
  PRINT *,' STANDARD DEVIATION = ',SD
  PRINT *,' MEDIAN IS = ',MEDIAN
  !
  IF (ALLOCATED(X)) DEALLOCATE(X, STAT = IERR)
  IF (IERR /= 0) THEN
     PRINT*, "X NON DEALLOCATED!"
     STOP
  ENDIF
  PRINT*, Y
  CALL STATS(Y,I,M,SD,MEDIAN)
  !
  PRINT *,' MEAN = ',M
  PRINT *,' STANDARD DEVIATION = ',SD
  PRINT *,' MEDIAN IS = ',MEDIAN
  !
  IF (ALLOCATED(Y)) DEALLOCATE(Y, STAT = IERR)
  IF (IERR /= 0) THEN
     PRINT*, "Y NON DEALLOCATED!"
     STOP
  ENDIF
  !
CONTAINS
  !
  SUBROUTINE BOX_MULLER(dim)
    !
    ! Uses the Box-Muller method to create two normally distributed vectors
    !
    INTEGER, INTENT(IN) :: dim
    !
    REAL(KIND=dbl), PARAMETER :: PI = ACOS(-1.0_dbl)
    REAL(KIND=dbl), DIMENSION(dim) :: RANDOM_u, RANDOM_v ! Automatic arrays
    !
    CALL RANDOM_NUMBER(RANDOM_u)
    CALL RANDOM_NUMBER(RANDOM_v)
    !
    X = SQRT(-2.0_dbl*LOG(RANDOM_u))
    Y = X*SIN(2.0_dbl*PI*RANDOM_v)
    X = X*COS(2.0_dbl*PI*RANDOM_v)
    !
  END SUBROUTINE BOX_MULLER
  !
END PROGRAM ejemplo_10_3
SUBROUTINE STATS(VECTOR,N,MEAN,STD_DEV,MEDIAN)
  USE dble_prec
  IMPLICIT NONE
  ! Defincion de variables
  INTEGER , INTENT(IN)                    ::  N
  REAL(KIND=dbl)      , INTENT(IN) , DIMENSION(:)    ::  VECTOR    !! (1)
  REAL(KIND=dbl)      , INTENT(OUT)                  ::  MEAN
  REAL(KIND=dbl)      , INTENT(OUT)                  ::  STD_DEV
  REAL(KIND=dbl)      , INTENT(OUT)                  ::  MEDIAN
  REAL(KIND=dbl)      , DIMENSION(1:N)              ::  Y
  REAL(KIND=dbl)      :: VARIANCE = 0.0_dbl
  REAL(KIND=dbl)      :: SUMXI = 0.0_dbl, SUMXI2 = 0.0_dbl
  !
  SUMXI=SUM(VECTOR)       !! (6)
  SUMXI2=SUM(VECTOR*VECTOR)    !! (6)
  MEAN=SUMXI/N
  VARIANCE=(SUMXI2-SUMXI*SUMXI/N)/(N-1)
  STD_DEV = SQRT(VARIANCE)
  Y=VECTOR
  ! Ordena valores por proceso de seleccion
  CALL SELECTION
  IF (MOD(N,2) == 0) THEN
     MEDIAN=(Y(N/2)+Y((N/2)+1))/2
  ELSE
     MEDIAN=Y((N/2)+1)
  ENDIF
CONTAINS     !! (7)
  SUBROUTINE SELECTION
    IMPLICIT NONE
    INTEGER :: I,J,K
    REAL :: MINIMUM
    DO I=1,N-1
       K=I
       MINIMUM=Y(I)
       DO J=I+1,N
          IF (Y(J) < MINIMUM) THEN
             K=J
             MINIMUM=Y(K)
          END IF
       END DO
       Y(K)=Y(I)
       Y(I)=MINIMUM
    END DO
  END SUBROUTINE SELECTION
END SUBROUTINE STATS
```

10.3.4 Programa ejemplo_10_4.f90

```
MODULE NUMERIC_KINDS
  ! 4, 2, AND 1 BYTE INTEGERS
  INTEGER, PARAMETER :: &
       i4b = SELECTED_INT_KIND(9), &
       i2b = SELECTED_INT_KIND(4), &
       i1b = SELECTED_INT_KIND(2)
  ! SINGLE, DOUBLE, AND QUADRUPLE PRECISION
  INTEGER, PARAMETER :: &
       sp = KIND(1.0), &
       dp = SELECTED_REAL_KIND(2*PRECISION(1.0_sp)), &
       qp = SELECTED_REAL_KIND(2*PRECISION(1.0_dp))
END MODULE NUMERIC_KINDS
```

Chapter 11

Subprogramas (IV)

11.1 Objetivos

Los objetivos de esta clase son los siguientes:

1 Explicar como se deben gestionar los errores en la invocación de funciones y subrutinas.

2 Explicar como se pasa el nombre de una función o subrutina como argumento declarando las funciones o subrutinas implicadas con el atributo `EXTERNAL`.

3 Explicar como se pasa el nombre de una función o subrutina como argumento declarando las funciones o subrutinas en un módulo.

11.2 Puntos destacables.

1 Se debe evitar que un programa termine sin que una subprograma (función o subrutina) devuelva el control al programa que lo ha invocado. Por ello se debe no usar la orden `STOP` en el interior de subprogramas. La mejor forma de gestionar errores en una subrutina, sobre todo aquellos debidos a una incorrecta definición de los argumentos de entrada de la subrutina, es mediante el uso de varibles *flag* (bandera) que marquen que ha tenido lugar un error. En el siguiente ejemplo se calcula la raíz cuadrada de la diferencia entre dos números, y la variable `sta_flag` es cero si la subrutina se ejecuta sin problemas o uno si se trata de calcular la raíz cuadrada de un número negativo.

```
SUBROUTINE calc(a_1, a_2, result, sta_flag)
   IMPLICIT NONE
   REAL, INTENT(IN) :: a_1, a_2
   REAL, INTENT(OUT) :: result
   INTEGER, INTENT(OUT) :: sta_flag
   !
   REAL :: temp
   !
   temp = a_1 - a_2
   IF (temp >= 0) THEN
      result = SQRT(temp)
      sta_flag = 0
   ELSE
      result = 0.0
      sta_flag = 1
   ENDIF
END SUBROUTINE calc
```

Una vez ejecutada la subrutina se debe comprobar el valor de la variable `sta_flag` para informar si ha existido algún problema.

2 Al invocar una subrutina los argumentos pasan como una serie de punteros a ciertas posiciones de memoria. Eso permite que como argumento figure una función o subrutina.

3 En el caso de funciones, cuando se incluye el nombre de una función en la lista de argumentos se transforma en un puntero a dicha función. Para ello las funciones han de ser declaradas con el atributo `EXTERNAL`. Si, por ejemplo, desde un programa llamamos a una subrutina llamada `evaluate_func` para evaluar las funciones `fun_1` y `fun_2` podemos hacer algo como

```
PROGRAM test
  IMPLICIT NONE
  REAL :: fun_1, fun_2
  EXTERNAL fun_1, fun_2
  REAL :: x, y, output

  ......

  CALL evaluate_func(fun_1, x, y, output)
  CALL evaluate_func(fun_2, x, y, output)

  ......

END PROGRAM test

SUBROUTINE evaluate_func(fun, a, b, out)
   REAL, EXTERNAL :: fun
   REAL, INTENT(IN) :: a, b
   REAL, INTENT(OUT) :: out
   !
   out = fun(a,b)
END SUBROUTINE evaluate_func
```

En el 'Programa ejemplo_11_1.f90' on the current page se muestra un ejemplo en el que se evalua, dependiendo de la elección del usuario, el producto o el cociente entre dos números. Dependiendo de la elección se utiliza la subrutina `Eval_Func`, que acepta como uno de sus argumentos el nombre de la función que se va a evaluar, `prod_func` o `quot_func`. Debe indicarse el tipo de variable asociado a la función, pero no se puede especificar el atributo `INTENT`.

4 También pueden usarse nombres de subrutinas como argumentos. Para pasar el nombre de una subrutina como argumento dicha subrutina debe ser declarada con el atributo EXTERNAL. En el siguiente ejemplo una subrutina llamada `launch_sub` acepta como argumentos de entrada las variables `x_1` y `x_2` y el nombre de una subrutina a la que invoca con las variables anteriores como argumentos y tiene como argumento de salida la variable `result`.

```
SUBROUTINE launch_sub(x_1, x_2, sub_name, result)
  IMPLICIT NONE
  REAL, INTENT(IN) :: x_1, x_2
  EXTERNAL sub_name
  REAL, INTENT(OUT) :: result


  ......

  CALL sub_name(x_1, x_2, result)

  ......

END SUBROUTINE launch_sub
```

Como puede verse en este ejemplo, el argumento que indica la subrutina (`sub_name`) no lleva asociado el atributo `INTENT`. En el 'Programa ejemplo_11_2.f90' on the facing page se muestra un ejemplo similar al anterior, en el que se evalua dependiendo de la elección del usuario el producto o el cociente entre dos números. Dependiendo de la elección se utiliza la subrutina `Eval_Sub`, que acepta como uno de sus argumentos el nombre de la subrutina que se va a evaluar, `prod_sub` o `quot_sub`.

5 En el 'Programa ejemplo_11_3.f90' on page 64 se muestra un ejemplo algo más complejo en el que se evalua, dependiendo de la elección del usuario, una función entre tres posibles para un intervalo de la variable independiente. En este caso las funciones se declaran como `EXTERNAL` y se utiliza una subrutina interna para la definición del vector de la variable independiente, de acuerdo con la dimensión que proporciona el usuario, y la subrutina `Eval_Func` que acepta como uno de sus argumentos el nombre de la función que se evalue mostrando los resultados en pantalla.

6 Es posible también comunicar a un subprograma el nombre de una función o una subrutina mediante el uso de módulos. En el 'Programa ejemplo_11_4.f90' on page 66 se muestra un programa similar al 'Programa ejemplo_11_3.f90' on page 64 utilizando módulos. El módulo `Functions_11_4` debe compilarse en un fichero separado al del programa principal. Si, por ejemplo el módulo se llama `ejemplo_11_4_mod.f90` y el programa principal `ejemplo_11_4.f90` el procedimiento sería el siguiente

```
$ gfortran -c ejemplo_11_4_mod.f90
$ gfortran ejemplo_11_4.f90 ejemplo_11_4_mod.o
```

Como ocurría en el caso anterior, el o los argumentos que indican funciones o subrutinas no llevan el atributo `INTENT`.

11.3 Programas usados como ejemplo.

11.3.1 Programa ejemplo_11_1.f90

```
PROGRAM func_option
```

```
  !
  ! Select between funs to compute the product of the quotient of two quantities
  !
  IMPLICIT NONE
  !
  !
  REAL :: X_1, X_2
  INTEGER :: I_fun
  INTEGER :: I_exit
  !
  REAL, EXTERNAL :: prod_fun, quot_fun
  !
  I_exit = 1
  !
  DO WHILE (I_exit /= 0)
     !
     PRINT*, "X_1, X_2?"
     READ(UNIT = *, FMT = *) X_1, X_2
     !
     PRINT*, "function 1 = X_1 * X_2, 2 = X_1/X_2 ? (0 = exit)"
     READ(UNIT = *, FMT = *) I_fun
     !
     SELECT CASE (I_fun)
        !
     CASE (0)
        I_exit = 1
     CASE (1)
        CALL Eval_func(prod_fun, X_1, X_2)
     CASE (2)
        CALL Eval_func(quot_fun, X_1, X_2)
     CASE DEFAULT
        PRINT*, "Valid options : 0, 1, 2"
        !
     END SELECT
     !
     PRINT*, "Continue? (0 = exit)"
     READ(UNIT=*, FMT = *) I_exit
     !
     !
  ENDDO
  !
END PROGRAM func_option
!
SUBROUTINE Eval_Func(fun, X_1, X_2)
  !
  IMPLICIT NONE
  !
  REAL, INTENT(IN) :: X_1, X_2
  REAL, EXTERNAL :: fun
  !
  PRINT 10, fun(X_1, X_2)
  !
  10 FORMAT(1X, ES16.8)
  !
END SUBROUTINE Eval_Func
!
!
FUNCTION prod_fun(x1, x2)
  !
  IMPLICIT  NONE
  !
  REAL, INTENT(IN) :: x1, x2
  !
  REAL prod_fun
  !
  prod_fun = x1*x2
  !
END FUNCTION prod_fun
!
FUNCTION quot_fun(x1, x2)
  !
  IMPLICIT  NONE
  !
  REAL, INTENT(IN) :: x1, x2
  !
  REAL quot_fun
  !
  quot_fun = x1/x2
  !
END FUNCTION quot_fun
```

11.3.2 Programa ejemplo_11_2.f90

```
PROGRAM sub_option
  !
  ! Select between subs to compute the product or the quotient of two quantities
  !
  IMPLICIT NONE
  !
  !
```

```
    REAL :: X_1, X_2
    INTEGER :: I_sub
    INTEGER :: I_exit
    !
    EXTERNAL :: prod_sub, quot_sub
    !
    I_exit = 1
    !
    DO WHILE (I_exit /= 0)
       !
       PRINT*, "X_1, X_2?"
       READ(UNIT = *, FMT = *) X_1, X_2
       !
       PRINT*, "function 1 = X_1 * X_2, 2 = X_1/X_2 ? (0 = exit)"
       READ(UNIT = *, FMT = *) I_sub
       !
       SELECT CASE (I_sub)
          !
       CASE (0)
          I_exit = 0
       CASE (1)
          CALL Eval_Sub(prod_sub, X_1, X_2)
       CASE (2)
          CALL Eval_Sub(quot_sub, X_1, X_2)
       CASE DEFAULT
          PRINT*, "Valid options : 0, 1, 2"
          !
       END SELECT
       !
       PRINT*, "Continue? (0 = exit)"
       READ(UNIT=*, FMT = *) I_exit
       !
    ENDDO
    !
  END PROGRAM sub_option
  !
  SUBROUTINE Eval_Sub(sub, X_1, X_2)
    !
    IMPLICIT NONE
    !
    EXTERNAL :: sub
    REAL, INTENT(IN) :: X_1, X_2
    !
    REAL :: res_sub
    !
    CALL sub(X_1, X_2, res_sub)
    PRINT 10, res_sub
    !
  10 FORMAT(1X, ES16.8)
    !
  END SUBROUTINE Eval_Sub
  !
  !
  SUBROUTINE prod_sub(x1, x2, y)
    !
    IMPLICIT  NONE
    !
    REAL, INTENT(IN) :: x1, x2
    REAL, INTENT(OUT) :: y
    !
    y = x1*x2
    !
  END SUBROUTINE prod_sub
  !
  !
  SUBROUTINE quot_sub(x1, x2, y)
    !
    IMPLICIT  NONE
    !
    REAL, INTENT(IN) :: x1, x2
    REAL, INTENT(OUT) :: y
    !
    y = x1/x2
    !
  END SUBROUTINE quot_sub
```

11.3.3 Programa ejemplo_11_3.f90

```
  PROGRAM call_func
    !
    ! Select which curve is computed and saved in a given interval e.g. (-2 Pi, 2 Pi)
    !
    ! 1 ---> 10 x^2 cos(2x) exp(-x)
    ! 2 ---> 10 (-x^2 + x^4)exp(-x^2)
    ! 3 ---> 10 (-x^2 + cos(x)*x^4)exp(-x^2)
    !
    IMPLICIT NONE
    !
    !
    REAL, DIMENSION(:), ALLOCATABLE :: X_grid
```

```
  !
  REAL, PARAMETER :: pi = ACOS(-1.0)
  !
  REAL :: X_min, X_max, Delta_X
  INTEGER :: X_dim, I_fun
  INTEGER :: I_exit, Ierr
  !
  REAL, EXTERNAL :: fun1, fun2, fun3
  !
  X_min = -2*pi
  X_max = 2*pi
  !
  I_exit = 0
  !
  DO WHILE (I_exit /= 1)
     !
     PRINT*, "number of points? (0 = exit)"
     READ(UNIT=*, FMT = *) X_dim
     !
     IF (X_dim == 0) THEN
        !
        I_exit = 1
        !
     ELSE
        ALLOCATE(X_grid(1:X_dim), STAT = Ierr)
        IF (Ierr /= 0) THEN
           STOP 'X_grid allocation failed'
        ENDIF
        !
        CALL make_Grid(X_min, X_max, X_dim)
        !
        PRINT*, "function 1, 2, or 3? (0 = exit)"
        READ(UNIT = *, FMT = *) I_fun
        !
        SELECT CASE (I_fun)
           !
        CASE (0)
           I_exit = 1
        CASE (1)
           CALL Eval_func(fun1, X_dim, X_grid)
        CASE (2)
           CALL Eval_func(fun2, X_dim, X_grid)
        CASE (3)
           CALL Eval_func(fun3, X_dim, X_grid)
        CASE DEFAULT
           PRINT*, "Valid options : 0, 1, 2, 3"
           !
        END SELECT
        !
        DEALLOCATE(X_grid, STAT = Ierr)
        IF (Ierr /= 0) THEN
           STOP 'X_grid deallocation failed'
        ENDIF
        !
     ENDIF
     !
  ENDDO
  !
CONTAINS
  !
  SUBROUTINE make_Grid(X_min, X_max, X_dim)
    !
    REAL, INTENT(IN) :: X_min, X_max
    INTEGER, INTENT(IN) :: X_dim
    !
    INTEGER :: Index
    REAL :: Delta_X
    !
    !
    Delta_X = (X_max - X_min)/REAL(X_dim - 1)
    !
    X_grid = (/ (Index, Index = 0 , X_dim - 1 ) /)
    X_grid = X_min + Delta_X*X_grid
    !
  END SUBROUTINE make_Grid
  !
END PROGRAM call_func
!
SUBROUTINE Eval_Func(fun, dim, X_grid)
  !
  IMPLICIT NONE
  !
  INTEGER, INTENT(IN) :: dim
  REAL, DIMENSION(dim), INTENT(IN) :: X_grid
  REAL, EXTERNAL :: fun
  !
  INTEGER :: Index
  !
  DO Index = 1, dim
     PRINT 10, X_grid(Index), fun(X_grid(Index))
  ENDDO
  !
  10 FORMAT(1X, ES16.8,2X, ES16.8)
```

```
  !
END SUBROUTINE Eval_Func
!
!
FUNCTION fun1(x)
  !
  IMPLICIT  NONE
  !
  REAL, INTENT(IN) :: x
  !
  REAL fun1
  !
  fun1 = 10.0*x**2*cos(2.0*x)*exp(-x)
  !
END FUNCTION fun1
!
FUNCTION fun2(x)
  !
  IMPLICIT  NONE
  !
  REAL, INTENT(IN) :: x
  !
  REAL fun2
  !
  fun2 = 10.0*(-x**2 + x**4)*exp(-x**2)
  !
END FUNCTION fun2
!
FUNCTION fun3(x)
  !
  IMPLICIT  NONE
  !
  REAL, INTENT(IN) :: x
  !
  REAL fun3
  !
  fun3 = 10.0*(-x**2 + cos(x)*x**4)*exp(-x**2)
  !
END FUNCTION fun3
```

11.3.4 Programa ejemplo_11_4.f90

```
PROGRAM call_func
  !
  ! Select which curve is computed and saved in a given interval e.g. (-2 Pi, 2 Pi)
  !
  ! 1 ---> 10 x^2 cos(2x) exp(-x)
  ! 2 ---> 10 (-x^2 + x^4)exp(-x^2)
  ! 3 ---> 10 (-x^2 + cos(x)*x^4)exp(-x^2)
  !
  USE Functions_11_4
  !
  IMPLICIT NONE
  !
  !
  REAL, DIMENSION(:), ALLOCATABLE :: X_grid
  !
  REAL, PARAMETER :: pi = ACOS(-1.0)
  !
  REAL :: X_min, X_max, Delta_X
  INTEGER :: X_dim, I_fun
  INTEGER :: I_exit, Ierr
  !
  X_min = -2*pi
  X_max = 2*pi
  !
  I_exit = 0
  !
  DO WHILE (I_exit /= 1)
     !
     PRINT*, "number of points? (0 = exit)"
     READ(UNIT=*, FMT = *) X_dim
     !
     IF (X_dim == 0) THEN
        !
        I_exit = 1
        !
     ELSE
        ALLOCATE(X_grid(1:X_dim), STAT = Ierr)
        IF (Ierr /= 0) THEN
           STOP 'X_grid allocation failed'
        ENDIF
        !
        CALL make_Grid(X_min, X_max, X_dim)
        !
        PRINT*, "function 1, 2, or 3? (0 = exit)"
        READ(UNIT = *, FMT = *) I_fun
        !
        SELECT CASE (I_fun)
           !
```

```
         CASE (0)
            I_exit = 1
         CASE (1)
            CALL Eval_func(fun1, X_dim, X_grid)
         CASE (2)
            CALL Eval_func(fun2, X_dim, X_grid)
         CASE (3)
            CALL Eval_func(fun3, X_dim, X_grid)
         CASE DEFAULT
            PRINT*, "Valid options : 0, 1, 2, 3"
            !
         END SELECT
         !
         DEALLOCATE(X_grid, STAT = Ierr)
         IF (Ierr /= 0) THEN
            STOP 'X_grid deallocation failed'
         ENDIF
         !
      ENDIF
      !
   ENDDO
   !
CONTAINS
   !
   SUBROUTINE make_Grid(X_min, X_max, X_dim)
     !
     REAL, INTENT(IN) :: X_min, X_max
     INTEGER, INTENT(IN) :: X_dim
     !
     INTEGER :: Index
     REAL :: Delta_X
     !
     !
     Delta_X = (X_max - X_min)/REAL(X_dim - 1)
     !
     X_grid = (/ (Index, Index = 0 , X_dim - 1 ) /)
     X_grid = X_min + Delta_X*X_grid
     !
   END SUBROUTINE make_Grid
   !
END PROGRAM call_func
!
SUBROUTINE Eval_Func(fun, dim, X_grid)
   !
   USE Functions_11_4
   !
   IMPLICIT NONE
   !
   REAL :: fun
   INTEGER, INTENT(IN) :: dim
   REAL, DIMENSION(dim), INTENT(IN) :: X_grid
   !
   INTEGER :: Index
   !
   DO Index = 1, dim
      PRINT 10, X_grid(Index), fun(X_grid(Index))
   ENDDO
   !
   10 FORMAT(1X, ES16.8,2X, ES16.8)
   !
END SUBROUTINE Eval_Func
!
MODULE Functions_11_4
   IMPLICIT NONE
   !
CONTAINS
   !
   !
   FUNCTION fun1(x)
     !
     IMPLICIT  NONE
     !
     REAL, INTENT(IN) :: x
     !
     REAL fun1
     !
     fun1 = 10.0*x**2*cos(2.0*x)*exp(-x)
     !
   END FUNCTION fun1
   !
   FUNCTION fun2(x)
     !
     IMPLICIT  NONE
     !
     REAL, INTENT(IN) :: x
     !
     REAL fun2
     !
     fun2 = 10.0*(-x**2 + x**4)*exp(-x**2)
     !
   END FUNCTION fun2
   !
   FUNCTION fun3(x)
```

```
      !
      IMPLICIT  NONE
      !
      REAL, INTENT(IN) :: x
      !
      REAL fun3
      !
      fun3 = 10.0*(-x**2 + cos(x)*x**4)*exp(-x**2)
      !
    END FUNCTION fun3
  END MODULE Functions_11_4
```

Chapter 12

Instalación y uso de las bibliotecas BLAS y LAPACK

12.1 Objetivos

Los objetivos de esta clase son los siguientes:

1 familiarizar el alumno con la compilación de programas y la instalación de librerias o biliotecas usando el compilador `gfortran`.

2 Instalar las bibliotecas de interés científico `BLAS` y `LAPACK`.

3 Aprender a hacer uso de dichas bibliotecas.

Existe una gran cantidad de código `Fortran` accesible de forma abierta, ya sea como código fuente o en forma de biblioteca. En la presente clase el alumno se familiariza con la obtención, compilación, instalación y uso de dos bibliotecas de subrutinas de interés algebraico, `BLAS` y `LAPACK`.

12.2 Puntos destacables.

Indicaremos de forma escalonada los diferentes pasos que hay que seguir para la instalación de estas bibliotecas.

El código fuente de las bibliotecas `BLAS` y `LAPACK` puede descargarse de diferentes lugares, o instalarse a partir de paquetes de la distribución `Debian` o `Ubuntu` que se esté utilizando. En vez de ello las instalaremos compilándolas en nuestro ordenador.

- Descarga de código fuente de la biblioteca `BLAS`.

  Se puede descargar de la web de `NETLIB`[1], usando este enlace BLAS tgz (Netlib) (`http://www.netlib.org/blas/blas.tgz`).

- Una vez descargado el código fuente, se descomprime, se compila y se crea finalmente la libreria.

```
tar xzf blas.tgz
cd BLAS
gfortran -O2 -c *.f
ar cr libblas.a *.o
```

  Con lo que se debe haber creado la librería estática `libblas.a`.

- A continuación se sitúa dicha libreria en un lugar apropiado, por ejemplo con

```
sudo cp libblas.a /usr/local/lib
```

[1] El repositorio de `Netlib` contiene software gratuito, documentación y bases de datos de interés para la comunidad científica en general y para aquellos interesados en la computación científica en particular. El repositorio es mantenido por los Laboratorios AT&T-Bell, la Universidad de Tennessee y el laboratorio nacional de Oak Ridge, con la ayuda de un gran número de colaboradores en todo el mundo. La web de `Netlib` es `http://www.netlib.org`.

y se comprueba que tiene los permisos adecuados.

- Descarga del código fuente de la biblioteca LAPACK.

  Se puede descargar también de la web de NETLIB usando este enlace LAPACK tgz (Netlib) (http://www.netlib.org/lapack/lapack.tgz). Tras su descarga se desempaquetan los ficheros.

```
tar xzf lapack.tgz
cd lapack-3.2.1
```

- Esta biblioteca si tiene una serie de ficheros makefile para su compilación. Hemos de preparar un fichero make.inc adecuado, como el que hemos incluido en 'Ejemplo de fichero make.inc para LAPACK' on the facing page y que está disponible en Moodle en un fichero llamado make.inc.lapack.ubuntu (http://moodle.uhu.es/contenidos/file.php/245/src_fortran_clase/make.inc.lapack.ubuntu).

  Usando este fichero compilamos la librería haciendo

```
make
```

- Por último, instalamos la librería copiando los ficheros creados al lugar que creamos más adecuado para su ubicación.

```
sudo cp lapack_LINUX.a /usr/local/lib
sudo cp tmglib_LINUX.a /usr/local/lib
```

- Para terminar descargamos el código fuente de la biblioteca LAPACK95.

  Se puede descargar también de la web de NETLIB usando el enlace LAPACK95 tgz (Netlib) (http://www.netlib.org/lapack95/lapack95.tgz). Tras su descarga se desempaquetan los ficheros.

```
tar xzf lapack95.tgz
cd LAPACK95
```

- Esta biblioteca también tiene una serie de ficheros makefile para su compilación. Hemos de preparar de nuevo un fichero make.inc adecuado, como el que hemos incluido en 'Ejemplo de fichero make.inc para LAPACK95' on the next page y que está disponible en Moodle en un fichero llamado make.inc.lapack95.ubuntu (http://moodle.uhu.es/contenidos/file.php/245/src_fortran_clase/make.inc.lapack95.ubuntu).

  Usando este fichero compilamos la librería haciendo

```
cd SRC
make single_double_complex_dcomplex
```

  La opción escogida es la más general, pues general la librería para precisión simple, doble, compleja simple y compleja doble.

- Por último, instalamos la librería copiando los ficheros creados al lugar que creamos más adecuado para su ubicación.

```
sudo cp lapack95.a /usr/local/lib
sudo cp -r lapack95_modules /usr/local/lib
```

- En los directorios de ejemplos (LAPACK95/EXAMPLES1 y LAPACK95/EXAMPLES2) encontramos un gran número de ejemplos que podemos correr y comprobar las salidas obtenidas con las que se encuentran en Lapack95 User's guide (http://www.netlib.org/lapack95/lug95/node1.html). Las instrucciones para compilar y correr los ejemplos proporcionados pueden verse en el fichero README del directorio donde se encuentra el código fuente de los ejemplos.

- En el ejemplo 'Ejemplo de programa que invoca LAPACK95' on page 72 se encuentra el código de un programa donde se recurre a la subrutina la_spsv para hallar la solución de un sistema lineal de ecuaciones, $Ax = B$, donde la matriz del sistema, A, es simétrica y se almacena de forma compacta y x, B son vectores. Es importante que comprenda como funciona este programa, así como que se sepa extraer de la documentación de LAPACK95 el significado de los argumentos de entrada y salida de la subrutina.

- Para correr este programa es necesario descargar el código ejemplo_la_spsv.f90 y los ficheros de datos spsv.ma y spsv.mb de la web del curso. Para compilar el programa se ejecuta la orden

```
gfortran -o ejemplo_la_spsv -I/usr/local/lib/lapack95_modules ejemplo_la_spsv.f90  /usr/local/lib/lapack95.a /usr/local/lib/t
```

  En esta orden de compilación se incluyen todas las librerías y módulos necesarios para que pueda crearse el ejecutable, haciendo uso de las librerías BLAS, LAPACK y LAPACK95 que hemos instalado.

- Para proyectos más complejos y evitar tener que escribir comandos de compilación tan complejos como el anterior es posible usar un fichero makefile como el que se proporciona en el ejemplo 'Ejemplo de makefile para compilar programas que invocan LAPACK95' on page 73. Para usar este fichero en la compilación del ejemplo 'Ejemplo de programa que invoca LAPACK95' on page 72 es preciso copiar el fichero proporcionado o descargar el fichero makefile_lapack95 y ejecutar la orden

```
make -f makefile_lapack95 ejemplo_la_spsv
```

12.3 Programas usados como ejemplo.

12.3.1 Ejemplo de fichero make.inc para LAPACK

```
####################################################################
#  LAPACK make include file.                                       #
#  LAPACK, Version 3.2.1                                           #
#  MAY 2009                                                        #
#  Modified by Currix                                              #
####################################################################
#
SHELL = /bin/sh
#
#  The machine (platform) identifier to append to the library names
#
PLAT = _LINUX
#
#  Modify the FORTRAN and OPTS definitions to refer to the
#  compiler and desired compiler options for your machine.  NOOPT
#  refers to the compiler options desired when NO OPTIMIZATION is
#  selected.  Define LOADER and LOADOPTS to refer to the loader and
#  desired load options for your machine.
#
FORTRAN  = gfortran
OPTS     = -O2
DRVOPTS  = $(OPTS)
NOOPT    = -O0
LOADER   = gfortran
LOADOPTS =
#
# Timer for the SECOND and DSECND routines
#
# Default : SECOND and DSECND will use a call to the EXTERNAL FUNCTION ETIME
#TIMER    = EXT_ETIME
# For RS6K : SECOND and DSECND will use a call to the EXTERNAL FUNCTION ETIME_
# TIMER    = EXT_ETIME_
# For gfortran compiler: SECOND and DSECND will use a call to the INTERNAL FUNCTION ETIME
TIMER    = INT_ETIME
# If your Fortran compiler does not provide etime (like Nag Fortran Compiler, etc...)
# SECOND and DSECND will use a call to the INTERNAL FUNCTION CPU_TIME
# TIMER    = INT_CPU_TIME
# If neither of this works...you can use the NONE value... In that case, SECOND and DSECND will always return 0
# TIMER     = NONE
#
#  The archiver and the flag(s) to use when building archive (library)
#  If you system has no ranlib, set RANLIB = echo.
#
ARCH     = ar
ARCHFLAGS= cr
RANLIB   = ranlib
#
#  Location of the extended-precision BLAS (XBLAS) Fortran library
#  used for building and testing extended-precision routines.  The
#  relevant routines will be compiled and XBLAS will be linked only if
#  USEXBLAS is defined.
#
# USEXBLAS    = Yes
XBLASLIB     =
# XBLASLIB    = -lxblas
#
#  The location of the libraries to which you will link.  (The
#  machine-specific, optimized BLAS library should be used whenever
#  possible.)
#
#BLASLIB      = ../../blas$(PLAT).a
BLASLIB      = /usr/local/lib/libblas.a
LAPACKLIB    = lapack$(PLAT).a
TMGLIB       = tmglib$(PLAT).a
EIGSRCLIB    = eigsrc$(PLAT).a
LINSRCLIB    = linsrc$(PLAT).a
```

12.3.2 Ejemplo de fichero make.inc para LAPACK95

```
#
#  -- LAPACK95 interface driver routine (version 2.0) --
#     UNI-C, Denmark; Univ. of Tennessee, USA; NAG Ltd., UK
#     August 5, 2000
#
FC  = gfortran
FC1      = gfortran
# -dcfuns  Enable recognition of non-standard double
#          precision  complex intrinsic functions
# -dusty   Allows the compilation and execution of "legacy"
#          software by downgrading the category  of  common
#          errors  found  in  such software from "Error" to
# -ieee=full enables all IEEE arithmetic facilities
#          including non-stop arithmetic.
```

```
OPTS0    = -u -V -dcfuns -dusty -ieee=full
MODLIB   = -I./../lapack95_modules
OPTS1    = -c $(OPTS0)
OPTS3    = $(OPTS1) $(MODLIB)
OPTL     = -o
OPTLIB   =

LAPACK_PATH = /usr/local/lib/

LAPACK95 = ../lapack95.a
LAPACK77 = $(LAPACK_PATH)/lapack_LINUX.a
TMG77    = $(LAPACK_PATH)/tmglib_LINUX.a
BLAS     = $(LAPACK_PATH)/libblas.a

LIBS     = $(LAPACK95) $(TMG77) $(LAPACK77) $(BLAS)
SUF      = f90

XX = 'rm' -f $@; \
        'rm' -f $@.res; \
 $(FC) $(OPTS0) -o $@ $(MODLIB) $@.$(SUF) $(OPTLIB) $(LIBS); \
        $@ < $@.dat > $@.res; \
        'rm' -f $@

YY = $(FC) $(OPTS0) -o $@ $(MODLIB) $@.$(SUF) $(OPTLIB) $(LIBS)

.SUFFIXES: .f90 .f .o

.$(SUF).o:
 $(FC) $(OPTS3) $<

.f.o:
 $(FC1) $(OPTS3) $<
```

12.3.3 Ejemplo de programa que invoca LAPACK95

```
PROGRAM LA_SSPSV_EXAMPLE

  !  -- LAPACK95 EXAMPLE DRIVER ROUTINE (VERSION 1.0) --
  !     UNI-C, DENMARK
  !     DECEMBER, 1999
  !
  !  .. "Use Statements"
  USE LA_PRECISION, ONLY: WP => SP
  USE F95_LAPACK, ONLY: LA_SPSV
  !  .. "Implicit Statement" ..
  IMPLICIT NONE
  !  .. "Local Scalars" ..
  INTEGER :: I, N, NN, NRHS
  !  .. "Local Arrays" ..
  INTEGER, ALLOCATABLE :: IPIV(:)
  REAL(WP), ALLOCATABLE :: B(:,:), AP(:)
  !  .. "Executable Statements" ..
  WRITE (*,*) 'SSPSV Example Program Results.'
  N = 5; NRHS = 1
  WRITE(*,'(5H N = , I4, 9H; NRHS = , I4)') N, NRHS
  NN = N*(N+1)/2
  ALLOCATE ( AP(NN), B(N,NRHS), IPIV(N) )
  !
  OPEN(UNIT=21,FILE='spsv.ma',STATUS='UNKNOWN')
  DO I=1,NN
     READ(21,'(F3.0)') AP(I)
  ENDDO
  CLOSE(21)
  !
  WRITE(*,*)'Matrix AP :'
  DO I=1,NN; WRITE(*,"(15(I3,1X,1X),I3,1X))") INT(AP(I));
  ENDDO
  !
  OPEN(UNIT=21,FILE='spsv.mb',STATUS='UNKNOWN')
  DO I=1,N
     READ(21,'(F3.0)') B(I,1)
  ENDDO
  CLOSE(21)
  !
  WRITE(*,*)'Matrix B :'
  DO I=1,N; WRITE(*,"(10(I3,1X,1X),I3,1X)')") INT(B(I,1));
  ENDDO
  !
  WRITE(*,*)" CALL LA_SPSV( AP, B, 'L', IPIV )"
  !
  CALL LA_SPSV( AP, B, 'L', IPIV )
  !
  WRITE(*,*)'AP on exit: '
  DO I=1,NN; WRITE(*,"(15(E13.5))") AP(I);
  ENDDO
  !
  WRITE(*,*)'Matrix B on exit :'
  DO I=1,N; WRITE(*,"(F9.5)") B(I,1);
  ENDDO
  WRITE(*,*)'IPIV = ', IPIV
```

```
  !
END PROGRAM LA_SSPSV_EXAMPLE
```

12.3.4 Ejemplo de `makefile` para compilar programas que invocan `LAPACK95`

```
#
#  -- LAPACK95 makefile (version 1.0) --
#
FC  = gfortran
#
MODLIB   = -I/usr/local/lib/lapack95_modules
OPTS1    = -c
OPTS3    = $(OPTS1) $(MODLIB)
OPTL     = -o
OPTLIB   =

LAPACK_PATH = /usr/local/lib
LAPACK95_PATH = /usr/local/lib

LAPACK95 = $(LAPACK95_PATH)/lapack95.a
LAPACK77 = $(LAPACK_PATH)/lapack_LINUX.a
TMG77    = $(LAPACK_PATH)/tmglib_LINUX.a
BLAS     = $(LAPACK_PATH)/libblas.a

LIBS     = $(LAPACK95) $(TMG77) $(LAPACK77) $(BLAS)
SUF      = f90

YY = $(FC) -o $@ $(MODLIB) $@.$(SUF) $(OPTLIB) $(LIBS)

.SUFFIXES: .f90 .f .o

.$(SUF).o:
 $(FC) $(OPTS3) $<

ejemplo_la_spsv:
 $(YY)

clean:
 'rm' -f *.o *.mod core
```

Chapter 13

Referencias

1 Stephen J. Chapman; *Fortran 95/2003 for Scientists and Engineers*, 3a Ed. Mc Graw Hill 2008.

2 Michael Metcalf, John Reid, and Malcolm Cohen; *Modern Fortran Explained*, Oxford University Press 2011.

3 Jeanne C. Adams *et al.*; *Fortran 95 Handbook*, MIT Press 1997.

4 Ian D. Chivers and Jane Sleightholme; *Introduction to Programming with Fortran*, Springer 2006.

5 An Interactive Fortran 90 Programming Course (http://www.liv.ac.uk/HPC/HTMLFrontPageF90.html)

6 gfortran: The GNU Fortran compiler (http://gcc.gnu.org/onlinedocs/gfortran)

7 Gfortran - GCC Wiki (http://gcc.gnu.org/wiki/GFortran)

8 USER NOTES ON FORTRAN PROGRAMMING (UNFP) (http://sunsite.informatik.rwth-aachen.de/fortran/)

9 Fortran 90 Tutoria by Michael Metcalf (http://wwwasdoc.web.cern.ch/wwwasdoc/WWW/f90/f90.html)