1 ELEMENTS OF FORTRAN

1.1 Why Fortran?

"High level" programming languages: Fortran, BASIC, Pascal, C, C++. Each is best suited for particular types of applications. Symbolic names are used to represent mathematical quantities. Mathematical formulae are written in a comprehensible form, such as

$\mathtt{X} = (\mathtt{A} + \mathtt{B})/(2 * \mathtt{C})$

Fortran — FORmula TRANslation — the first "high level" programming language. Proposed in 1953 by John Backus; the first Fortran program was run in 1957. FORTRAN 66 was the first ever standard for a programming language. A program written in standard Fortran is guaranteed to run at any installation claiming to support the standard. Further standards followed: Fortran 77 (in 1978), Fortran 90 (in 1991), Fortran 95...

The superiority of Fortran manifests itself mainly in the area of numerical, scientific, and technical applications. There is still no significant competitor in spite of the appearance of newer languages (like Pascal and C). Being clearly oriented towards scientific application, Fortran disposes of a vast built-in collection of intrinsic procedures. Huge commercial libraries of programs and procedures are available. Important mathematical and statistical libraries, like IMSL and NAG, have been designed and developed over decades to .

Main new features of Fortran 90:

- array operations using a concise but powerful notation
- user-defined data types
- pointers
- dynamic storage allocation
- free source form (as opposed to FORTRAN 66)
- modules — program units, useful for global data definitions and procedure libraries.

1.2 Control statements

The program execution evolves strictly in the appearance order of the statements in the source code only in very simple cases. Often the implemented algorithms imply repetitive or alternative calculations, in direct dependence on certain conditions, which themselves are realized during the program execution.

The modern programming languages are based on control structures, which are *block constructs*, which begin with an initial keyword statement, end with a matching terminal statement, may have intermediate statements, and which may be entered only at the initial statement. Executable constructs may be *nested*, that is a block construct may contain another executable construct. Execution of a block always begins with its first executable statement.

The programming based on block constructs is called *structured programming*. Among the undisputed advantages of structured programming are worth mentioning: program clarity and versatility, reduction of error sources and simplified debugging.

1.2.1 `GO TO` statement

It is the most disputed statement in programming languages, since it usually violates the principles of structured programming. There are, however, certain situations, especially when dealing with error conditions, when `GO TO` statements are necessary even in high level programming languages. In any case, this statement should be avoided whenever possible.

The general form of the `GO TO` statement is:

`GO TO` *label*

where *label* is the label of an *executable statement.*

1.2.2 `IF` statement

The `IF` statements provide a mechanism for branching depending on a condition. In the `IF` statement the value of a scalar logical expression is tested and a single statement (typically an assignment or a branch) is executed if its value is true.

The general form of the `IF` statement is:

`IF (`*scalar-logical-expr*`)` *action-stmt*

where *scalar-logical-expr* is any scalar logical expression and *action-stmt* is any executable statement other than the initial or terminal statement of a block.

Typical examples are:

```
IF (a*b > 0.e0)  GO TO 999
IF ((ABS(f) <= eps).AND.(ABS(dx)<=eps*ABS(x)))  EXIT
IF (x > xmax)  xmax = x
```

1.2.3 `IF` construct

The `IF` construct allows either the execution of a block depending on a condition, or the execution of alternative blocks depending on alternative conditions.

In its simplest form the `IF` construct is:

```
[name:] IF (scalar-logical-expr) THEN
            block
        END IF [name]
```

where *scalar-logical-expr* is any scalar logical expression and *block* is any executable block, except an `END` statement or an incomplete construct. The *block* is executed only if *scalar-logical-expr* evaluates to the value true.

The `IF` construct may be optionally named.

In the next example the values of two variables are interchanged such that the first one is larger:

```
swap: IF (x < y) THEN
         temp = x; x = y; y = temp
      END IF swap
```

It is good practice to indent the block inside the `IF` construct, since this makes the logic easier to understand.

In the case of the second form of the `IF` construct, an alternative block is executed in the case where the condition is false:

```
[name:] IF (scalar-logical-expr) THEN
            block1
        ELSE [name]
            block2
        END IF [name]
```

In the next example the larger of two values is determined:

```
IF (x >= y) then
   max = x
ELSE
   max = y
END IF
```

The most general form of the `IF` construct uses the `ELSE IF` statement to make a series of successive logical tests, each associated with an executable block of statements.

```
[name:] IF (scalar-logical-expr) THEN
            block
        [ELSE IF (scalar-logical-expr) THEN [name]
            block]...
        [ELSE [name]
            block]
        END IF [name]
```

The logical testes are performed one after the other until one is satisfied and the associated block is executed, or until all tests are exhausted. Control then passes to the `END IF` statement and the execution exits the `IF` construct. The statements within an `IF` construct may be labelled, however control should not be passed into the `IF` construct from outside it

The next example is a typical sequence for performing a step within a two-dimensional random walk:

```
call RANDOM_NUMBER(p)
IF (p <= 0.25) THEN
   x = x + h
ELSE IF (p <= 0.50) THEN
   x = x - h
ELSE IF (p <= 0.75) THEN
   y = y + h
ELSE
   y = y - h
END IF
```

1.2.4 `CASE` construct

The `CASE` construct provides an alternative to the `IF` construct for selecting one of several options. The main difference is that for the `CASE` construct only one expression is evaluated for deciding which case to select.

The general form of the `CASE` construct is:

```
[name:] SELECT CASE (scalar-expr)
        [CASE selector[name]
            block]...
        END SELECT [name]
```

The expression *scalar-expr* must be of type character, logical, or integer, and the specified values in each *selector* must be of the same type.

```
SELECT CASE (monnam)
   CASE ('H2O')
      CALL CellH2O(xcell,ycell,zcell)
   CASE ('NH3')
      CALL CellNH3(xcell,ycell,zcell)
   CASE DEFAULT
      STOP 'No such molecule !'
END SELECT
```

For character or integer *scalar-expr*, a range may be specified:

`CASE` (*low*:*high*)

Either *low* or *high* may be absent, indicating that the case is selected whenever *scalar-expr* is less or equal to *high*, or greater or equal *low*, respectively.

As example, we consider again the step selection in a two-dimensional random walk:

```
call RANDOM_NUMBER(p)
ip = int(100*p)
SELECT CASE (ip)
   CASE ( 1:25)
```

```
      x = x + h
   CASE (26:50)
      x = x - h
   CASE (51:75)
      y = y + h
   CASE DEFAULT
      y = y - h
END SELECT
```

Selectors are not permitted to overlap, therefore at most one may be satisfied.

1.2.5 `DO` construct

Repetitive algorithm sequences are coded in Fortran most commonly by means of the `DO` construct. Its general form is:

```
[name:] DO [variable = expr1,expr2[,expr3]]
            block
        END DO [name]
```

where *variable* is a named scalar integer variable, *expr1*, *expr2*, and *expr3* are any valid scalar integer expressions. *expr1* is the initial value and *expr2* is the final value of the loop counter *variable*. *expr3* is the counter increment; it must be nonzero if present and it is assumed 1 if absent.

The number of iterations performed by the `DO` construct is given by

$$\max((expr2 - expr1 + expr3)/expr3, 0).$$

The simplest form of a `DO` construct implements an endless loop:

```
[name:] DO
            block
        END DO [name]
```

The `EXIT` statement provides a means to exit from a `DO` loop, usually when certain conditions are satisfied. Its general form is:

```
EXIT [name]
```

where *name*, if present, specifies from which construct execution should exit in the case of nested constructs. Execution of an `EXIT` statement causes control to be transferred to the next executable statement after the `END DO` statement to which it refers. If *name* is not specified, the execution of the innermost loop is terminated.

Example:

```
DO k = 1,kmax
   . . .
   t = 0.5e0*(t0 + h*sum)
   IF (ABS(t - t0) <= eps*ABS(t)) EXIT
   . . .
   t0 = t
END DO
```

Certain iterations of a `DO` loop may be skipped by using the `CYCLE` statement:

`CYCLE` [*name*]

This statement transfers control to the `END DO` statement of the corresponding construct, the loop counter being incremented for the next iteration.

`DO` constructs may be nested, provided each loop completely contains the next inner one, if any.

The next program sequence illustrates matrix multiplication:

```
DO i = 1,n
   DO j = 1,m
      c(i,j) = 0.d0
      DO k = 1,l
         c(i,j) = c(i,j) + a(i,k)*b(k,j)
      END DO
   END DO
END DO
```

1.3 Arrays

1.3.1 Arrays of fixed sizes

An array is collection of elements of the same type. Each array element itself is a scalar and is specified by the array name and by one or more subscripts. The number of elements of an array is called its *size*. The number of elements along a dimension of an array is called the *extent* in that dimension. Fortran 90 allows up to seven dimensions to be specified. The number of dimensions is called the *rank* of the array. The sequence of extents is known as the *shape* of the array.

An array named `A` of 10 real elements may be declared by the statement:

```
REAL, DIMENSION(10) :: A
```

and the elements may be referenced as `A(1)`, `A(2)`, ... `A(10)`. The arrays in Fortran are by default of *offset* 1. It is however possible to declare indexes in particular intervals, by using the statement

`DIMENSION(`*low*`:`*high*`)`

like in the next example:

```
REAL, DIMENSION(-4:5) :: A
```

The vector `A` will now have the elements `A(-4)`, `A(-3)`, ... `A(5)`. The specification of the upper bound of the extent is mandatory, whereas the lower bound is optional and it defaults to 1 if absent.

A two-dimensional array might be declared as:

```
REAL, DIMENSION(-10:10,20) :: B
```

The extents in the two dimensions are 21 and 20, respectively, and the size of the array is 420.

Individual elements of an array are referenced by their subscript values. As a general rule, each subscript is a *scalar integer expression.* Each subscript must take values within the range defined in the array declaration.

In Fortran 90 *subarrays* (or *sections*) may be referenced by specifying ranges for the subscripts. For example, `a(-i:i,-j:j,k,l)` is a rank-two array with extents `2*i+1` and `2*j+1`.

1.3.2 Allocatable arrays

Quite often arrays are required to be of a size and shape that result during program execution. Fortran 90 provides dynamic allocation of storage based on a *heap* storage mechanism. For arrays that are not dummy arguments or function results, dynamic allocation may be accomplished by first specifying the `ALLOCATABLE` attribute in their declaration, like in the example:

```
REAL, DIMENSION(:,:), ALLOCATABLE :: a, b
```

Such arrays are called *allocatable* and their rank is given by the declaration. The actual shape is specified at the point in the program execution when the arrays are effectively allocated by an `ALLOCATE` statement:

```
ALLOCATE(a(-m:m,n),b(m,0:n))
```

Each *array-bound* has the form:

[*lower-bound*:]*upper-bound*

where *lower-bound* and *upper-bound* are scalar integer expressions. The default value for the lower bound is 1.

When an allocatable array is no longer needed, the corresponding heap storage may be recovered by using the `DEALLOCATE` statement, as in the example:

```
  DEALLOCATE(a,b)
```

In the next example, the module `global` declares the global array `a`, which can be allocated, deallocated and used by any program unit which includes the module:

```
MODULE global
   INTEGER n
   REAL, DIMENSION(:,:), ALLOCATABLE :: a
END MODULE

PROGRAM main
   USE global
   :
   READ(*,*) n
   ALLOCATE(a(-n:n,-n:n))
   :
   DEALLOCATE (a)
   :
END PROGRAM
```

1.3.3 Array constructors

Array constructors play the role of rank-one array constants and they may be constructed as lists of elements enclosed between the tokens `(/` and `/)`. They provide a convenient way of initializing arrays. Consider the equivalent examples

```
   REAL, DIMENSION(5) :: a = (/ 0.2, 0.4, 0.6, 0.8, 1.0/)
```

and

```
   REAL, DIMENSION(5) :: a = (/ (0.2*i,i=1,5) /)
```

The second example illustrates a *constructor-implied-do* and the general form is:

`(/ (`*array-constructor-value-list*`,` *variable* `=` *expr1*`,`*expr2*[`,`*expr3*]`) /)`

where *variable* is a named scalar variable, and *expr1*, *expr2*, and *expr3* are scalar integer expressions.

The syntax permits nesting of a *constructor-implied-do* inside another, as in:

```
(/ ((i+j,i=1,3),j=0,6,3) /)
```

which is equivalent to

```
(/ 1, 2, 3, 4, 5, 6, 7, 8, 9 /)
```

1.4 Program units and procedures

It is possible to write a complete Fortran program as a single unit, but this is considered obsolete. Presently it is considered good practice to break the program down into *program units*. Program units are conceived to execute well-defined program tasks. They can be written, compiled and tested *independently.*

Types of program units:

- *main program*
- *subprograms* (or *procedures*) — *functions* and *subroutines*
- *modules.*

An executable program must include at least one *main program.* The main program controls the overall execution of the application. It normally contains besides usual statements *calls* to *subprograms.*

A *function* returns a single object and usually does not alter its arguments. A *subroutine* usually performs a more complex task and it returns several objects through its arguments.

A *module* is a collection of definitions and subprograms. It may be expected to provide facilities associated with some particular task, acting as library or data base.

1.4.1 Main program

The main program has the form:

```
[PROGRAM program-name]
    [specification-stmts]
    [executable-stmts]
[CONTAINS
    internal-subprograms]
END [PROGRAM [program-name]]
```

The `PROGRAM` statement is optional, though recommendable.

The *program-name* may be any valid Fortran name. The only compulsory statement in a Fortran program is `END`. It may be labeled and branched to from one of the executable statements. It signals to the compiler the end of the program unit and, when executed, it causes the program to stop.

Another way to stop program execution is by means of the `STOP` statement. The `STOP` statement may be placed in any executable program unit and may issue a message.

The *specification statements* are usually type and dimension specifications for the objects used by the program. The *executable statements* specify the actions that are to be performed by the program.

The `CONTAINS` statement indicates the presence of one or more internal subprograms. The internal subprograms are executed only by explicit calls from executable statements of the main program. If the execution of the last statement ahead of the `CONTAINS` statement does not result in a branch, control passes over the internal subprograms to the `END` statement and the program stops.

1.4.2 External subprograms

External subprograms are called from the main program or another external subprogram. They usually perform a well-defined task within the complete program. Apart from the leading statement, their form is similar to that of the main program:

```
SUBROUTINE subroutine-name [(arguments)]
    [specification-stmts]
    [executable-stmts]
[CONTAINS
    internal-subprograms]
```

`END` [`SUBROUTINE` [*subroutine-name*]]

or

`FUNCTION` *function-name* [`(`*arguments*`)`]
  [*specification-stmts*]
  [*executable-stmts*]
[`CONTAINS`
  internal-subprograms]
`END` [`FUNCTION` [*function-name*]]

The names of program units and external procedures are *global.*

The effect of the `END` statement in a subprogram is to return control to the caller. The `RETURN` statement provides an alternative possibility of returning control from the subprogram. It may be labelled, may be a part of an `IF` statement, and is executable. The `RETURN` statement must not appear among the executable statements of the main program

Example of a program calling a function to compute the factorial:

```
!==========================================================================
PROGRAM Factorial
!--------------------------------------------------------------------------
   INTEGER n

   PRINT '(" n = "\)'; READ(*,*) n
   PRINT '(i6,"! = ",1pe12.6)', n,Fact(n)
END

!==========================================================================
FUNCTION Fact(n)
!--------------------------------------------------------------------------
   INTEGER i, n
   REAL f

   f = 1.d0
   DO i = 2,n
      f = f * i
   END DO
   Fact = f
END
```

1.4.3 Modules

Modules are conceive to be collections of definitions and subprograms. Modules are used to package:

- global data (replaces `COMMON` and `BLOCK DATA`);
- type definitions;

- subprograms;
- interface blocks;
- namelist groups.

The modules have the general form:

`MODULE` *module-name*
[*specification-stmts*]
[`CONTAINS`
module-subprograms]
`END` [`MODULE` [*module-name*]]

Any program unit gains access to the definitions in a module by including at the head of its specification statements the statement:

`USE` *module-name*

A module subprogram has the same form as an external subprogram, except that `FUNCTION` or `SUBROUTINE` is mandatory on the `END` statement. It has access to the other entities defined in the module, including the ability to call the other subprograms of the module.

A module can have access to other modules by appropriate `USE` statements. However, it must not access itself directly or indirectly.

An example of a module containing subprogram definitions is:

```
MODULE vector_operations
CONTAINS
   !=========================================================================
   FUNCTION prddot(a,b)
   !-------------------------------------------------------------------------
      IMPLICIT REAL(8) (a-h,o-z)
      REAL(8) a(3),b(3)

      prddot = a(1)*b(1) + a(2)*b(2) + a(3)*b(3)
   END FUNCTION prddot

   !=========================================================================
   SUBROUTINE prdvec(a,b,c)
   !-------------------------------------------------------------------------
      IMPLICIT REAL(8) (a-h,o-z)
      REAL(8) a(3),b(3),c(3)

      c(1) = a(2)*b(3) - a(3)*b(2)
      c(2) = a(3)*b(1) - a(1)*b(3)
      c(3) = a(1)*b(2) - a(2)*b(1)
   END SUBROUTINE prdvec
END MODULE vector_operations
```

1.4.4 Internal subprograms

Internal subprograms may be defined inside main programs, external and module subprograms. The structure of an internal subprogram is similar to that of a module subprogram, however it may not contain further internal subprograms (maximum one level of nesting).

Example:

```
SUBROUTINE outer
   REAL x, y
   :
CONTAINS
   SUBROUTINE inner
      REAL y
      x = y + 1.
      :
   END SUBROUTINE inner
END SUBROUTINE outer
```

The `outer` subroutine is said to be the *host* of `inner`. The `inner` subroutine has access to entities in `outer` by *host association* (e.g. to x). y is a local variable to inner and cannot be accessed from `outer`. The presence of `FUNCTION` or `SUBROUTINE` on the `END` statement is mandatory.

An internal subprogram automatically has access to all the host's objects, including the ability to call its other internal subprograms.

1.4.5 Recursive procedures

A subprogram may invoke itself (recursively) directly or indirectly only if the leading statement is prefixed `RECURSIVE`.

Example of a program calling a recursive function to compute the factorial:

```
!=============================================================================
PROGRAM Factorial
!-----------------------------------------------------------------------------
   INTEGER n

   PRINT '(" n = "\)'; READ(*,*) n
   PRINT '(i6,"! = ",1pe12.6)', n,Fact(n)
END

!=============================================================================
RECURSIVE FUNCTION Fact(n) RESULT(f)
!-----------------------------------------------------------------------------
   INTEGER i, n
   REAL f

   IF (n > 1) THEN
```

```
      f = n * Fact(n-1)
   ELSE
      f = 1.e0
   END IF
END
```

In the case of a recursive function, the `RESULT` clause is necessary, since the function name itself cannot be used as a local variable.

1.4.6 Arrays as procedure arguments

1.4.7 Application: two-dimensional integration

The application deals with the integration of a function of two variables. The implementation makes use of modules, indirect recursion, and procedural arguments.

A two-dimensional integral can be represented as two embedded one-dimensional integrals:

$$\int_{x_{\min}}^{x_{\max}} \int_{y_{\min}(x)}^{y_{\max}(x)} f(x,y)dxdy = \int_{x_{\min}}^{x_{\max}} F_x(x)dx, \quad F_x(x) = \int_{y_{\min}(x)}^{y_{\max}(x)} f(x,y)dy,$$

The integrand $F_x(x)$ of the outer integral is the integral with respect to y along a line of fixed x.

Let us consider the actual example:

$$I = \int_0^1 dx \int_0^{\sqrt{1-x^2}} \sqrt{1-x^2-y^2}\, dy = \frac{\pi}{6} \approx 0.523599.$$

The integration domain is the first quadrant of the circle of radius 1. The result is obviously the volume of the first octant of the sphere of radius 1.

The implementation of the user function is contained in the module:

```
!==============================================================================
MODULE user                ! defines the integrand and the integration domain
!------------------------------------------------------------------------------
   REAL :: xmin = 0e0
   REAL :: xmax = 1e0
CONTAINS
   !===========================================================================
   FUNCTION f(x,y)
   !---------------------------------------------------------------------------
      REAL x, y

      f = SQRT(ABS(1e0 - x*x - y*y))
   END FUNCTION f
   !==========================================================================
   FUNCTION ymin(x)
   !--------------------------------------------------------------------------
      REAL x

      ymin = 0e0
   END FUNCTION ymin

   !==========================================================================
   FUNCTION ymax(x)
   !--------------------------------------------------------------------------
      REAL x

      ymax = SQRT(ABS(1e0 - x*x))
   END FUNCTION ymax
END MODULE user
```

As one-dimensional integrator we use the function `TrapezControl`, defined in the module:

```
!=============================================================================
MODULE integrator
CONTAINS
   !==========================================================================
   RECURSIVE FUNCTION TrapezControl(Func,a,b) RESULT(t)
   !--------------------------------------------------------------------------
   ! Integrates function Func over interval [a,b] by using the trapezoidal
   ! rule with automatic step control
   !--------------------------------------------------------------------------
      PARAMETER (eps = 1e-6)          ! relative precision of the integration
      PARAMETER (kmax = 30)               ! maximum number of step bisections
      REAL h, sum, t, t0
      INTEGER i, n
      INTEGER k
      INTERFACE; FUNCTION Func(x); REAL x; END FUNCTION Func; END INTERFACE

      h = b-a; n = 1
      t0 = 0.5e0*h*(Func(a) + Func(b))                ! initial approximation

      DO k = 1,kmax                                    ! loop of step halving
         sum = 0e0
         DO i = 1,n
            sum = sum + Func(a+(i-0.5e0)*h)
         END DO
         t = 0.5e0*(t0 + h*sum)                           ! new approximation
         IF (ABS(t - t0) <= eps*ABS(t)) EXIT               ! convergence test
         h = 0.5e0*h; n = n*2;                              ! halve step size
         t0 = t
      END DO
      if (k >= kmax) &
         PRINT *, 'TrapezControl: maximum iteration number exceeded !'
   END FUNCTION TrapezControl
END MODULE integrator
```

The main program and the interface functions usable with a one-dimensional integrator should look like:

```
!=============================================================================
MODULE global                                       ! defines a global variable
!-----------------------------------------------------------------------------
   REAL xglobal
END MODULE global

!=============================================================================
PROGRAM Integral2D                                              ! main program
!-----------------------------------------------------------------------------
   USE user
   USE integrator
   INTERFACE; FUNCTION Fx(x); REAL x; END FUNCTION Fx; END INTERFACE

   PRINT '(" Integral = ",1pe12.6)', TrapezControl(Fx,xmin,xmax)
END

!=============================================================================
FUNCTION Fx(x)                                     ! integrand of outer integral
!-----------------------------------------------------------------------------
   USE global
   USE user
   USE integrator
   REAL x
   INTERFACE; FUNCTION Fy(y); REAL y; END FUNCTION Fy; END INTERFACE

   xglobal = x
   Fx = TrapezControl(Fy,ymin(x),ymax(x))
END

!=============================================================================
FUNCTION Fy(y)                                    ! interface to user function f
!-----------------------------------------------------------------------------
   USE global
   USE user
   REAL y

   Fy = f(xglobal,y)
END
```

To compute the outer integral, the integrator is called for the function `Fx`, which returns the inner integral. The boundaries `xmin` and `xmax` are provided by module `user`. Function `Fx` computes the inner integral not by calling directly the two-dimensional user function `f`, but calling instead the one-dimensional interface function `Fy`. Function `Fy` receives the argument x through the global variable `xglobal`, defined in module `global` and set by function `Fx`.

Bibliografie

[1] M. Metcalf, J. Reid, *Fortran 90 Explained* (University Press, Oxford, 1994).

[2] M. Metcalf, J. Reid, *Fortran 90/95 Explained* (University Press, Oxford, 1996).

[3] B.D. Hahn, *Fortran 90 for Scientists and Engineers* (University Press, Cambridge, 1993).