Fortran 2003
Part 1

École normale supérieure
L3 geosciences
2020/2021

Lionel GUEZ
guez@lmd.ens.fr
Laboratoire de météorologie dynamique
Office E212

Modification date: Dec 10, 2020
Contents (1/2)

- Introduction
- ABC
- Types, data objects, declarations
- Assignment, expressions
- Input and output
- Selection control structure
- Iteration control structure
- Arrays
Contents (2/2)

- Standard intrinsic functions
Introduction
Introduction

• Fortran: *Formula translator*

• One of the oldest among still used programming languages. Created by IBM in 1954.

• Intended use: scientific computation.
The future of 1954

Scientists from the RAND Corporation have created this model to illustrate how a “home computer” could look like in the year 2004. However, the needed technology will not be economically feasible for the average home. Also, the scientists readily admit that the computer will require not-yet-invented technology to actually work, but 50 years from now scientific progress is expected to solve these problems. With teletype interface and the Fortran language, the computer will be easy to use.
History (1/3)

- 1954: Fortran, first definition of the language, by IBM
- 1957: Fortran II
- 1958: Fortran III, not published
History (2/3)

- 1961: Fortran IV
- Standardization in 1966 by a public American organization: ANSI (American National Standards Institute). The standard version becomes known as « Fortran 66 ».
- 1978: Fortran 77
  Standardization of Fortran becomes international: ISO (*International Standards Organization*).
History (3/3)

- 1991 : Fortran 90 (major revision)
- 1997 : Fortran 95 (minor revision)
- 2004 : Fortran 2003 (major revision)
- 2010 : Fortran 2008 (minor revision)
- 2018 : Fortran 2018 (minor revision)
The compilers

- Some free compilers: GNU Fortran, Intel Fortran (for non-commercial use).
- Some commercial compilers: IBM XL Fortran, NAG, PGI, Cray...
Implementation of Fortran standards by compilers

- Takes years
- First compiler for Fortran I (1954) in 1957
- Complete implementation of the Fortran 2008 standard by two compilers only: Cray and Intel.
Implementation of Fortran 2003 standard by compilers (1/2)

- IBM XL Fortran compiler since version 13.1, released in 2010
- Cray compiler since 2010
- Portland Group (PGI) compiler since version 13.7 in 2013
- Intel Fortran compiler since version 16.0, released in 2015
Implementation of Fortran 2003 standard by compilers (2/2)

• NAG Fortran compiler since version 6.2, released in 2018

• GNU Fortran compiler since version 9.1, released in 2019
Consequences of history (1/3)

- The standardization process implies **stability**, slow evolution of the language.
  (Note that not all programming languages are standardized. For example: Python, Perl...)
Consequences of history (2/3)

- There has been a large build-up of programs and libraries of procedures in Fortran since its birth.
- This work must not be wasted. This old code must remain functional. → Archaic features in the language must remain valid. Successive Fortran standards add a lot to the language and remove very little.
Consequences of history (3/3)

- The Fortran language becomes larger and larger. There are several valid (standard-conforming) ways to code the same algorithm.

- The choice between those different programming styles should be made based on conciseness, safety (little room left for bugs), clarity, efficiency (speed of execution). Good news: based on these criteria, new programming styles (from most recent Fortran standards) are the best.
This course

• Fortran 2003

• A selection, not a complete description of the language.
  • Selection of the most useful features
  • Selection of a programming style
Bibliography (1/2)

(From more specialized to more general, from tutorial to reference book.)


- Hanson et Hopkins *Numerical Computing with Modern Fortran* Siam, 2013
Bibliography (2/2)

- Adams et al.  
  Publisher: Springer  
  Date: 2009

- Freely available reference document:  
  draft Fortran standard 2003
• Identifiers (names of variables, main program unit, procedures, etc.): 1 to 63 characters, ABC...Z or abc...z, 0 1 ... 9, underscore _, beginning with a letter. For example: pressure, p_surface, t0 but not: température 2nd_level
Nota bene: unlike other languages, Fortran does not distinguish lower and upper case. $x$ and $X$, for example, refer to the same variable.

Some special characters play a role in Fortran:

`blank = + - * / ( ) [ ] , . ' : ! “ % & `
Other special characters have no meaning in Fortran:
~ \ { } ? ` ^ | $ # @
So you can only find them in character strings or comments.
You should be careful not to use keywords or predefined identifiers of the language for your own identifiers. Cf. list of Fortran keywords and list of intrinsic procedures (§§ 13.7) (126 intrinsic procedures).

132 characters at most per line. You can write a statement on several lines: type an ampersand & at the end of a line to be continued.
ABC (5/5)

- **Comments** start after the exclamation mark `!`
  Alone on one line or at the end of a line, after a statement.
Notation

In this document, curly brackets \{\} delimit an optional part. They are not part of the language, do \textit{not} write the curly brackets in your program!
A basic whole program

- **program** name
  
  {declarations}

  executable constructs

  end **program** name

- You can stop in the middle of a program with the following statement:
  
  **stop 1**
Types, data objects, declarations
Types

- types
  - intrinsic types ...
  - numeric types
    - logical
    - character
  - integer
  - real ...

- ...
God is real... unless explicitly declared an integer.

- Default implicit typing depending on the first letter of the identifier. Advice: deactivate it in all your programs with the following statement:
  ```
  implicit none
  ```
  This statement must be before all the declarations.

- ```
  program name
  implicit none
  {declarations}
  executable constructs
  end program name
  ```
Declaration of variables

• Write a line with a type followed by a list of variables.

• Examples:
  integer i, j
  real x
  real u, v ! speed
Numerical operations

• The relational operators on integer and real numbers are:
  <, <=, == for test of equality, /= means “different than”, >=, >
  The result is a logical value. Example:
  \[ x = i == 3 \]

• The numeric operators are: + - * / 
  ** for the exponentiation
Integer type (1/3)

• Integer literal constants: without a decimal point

• Example of declaration of a variable: `integer i`

• Largest integer value: `huge(θ)` (depends on the compiler, but usually $2^{31}$)
Integer type (2/3)

- Nota bene: / with integer operands gives an integer result (the floor of the ratio).
  Examples:
  1 / 2 equals 0
  5 / 2 equals 2
Integer type (3/3)

• Some intrinsic functions that you can apply to integers:
  • `abs(a)` absolute value
  • `mod(a, p)`: a modulo p
    That is: `a - (a / p) * p`
Real type (1/2)

• Real literal constants:
  Without exponent: \(-8.4\)
  With exponent: \(1e4, 64.2e-5\)

• Example of declaration of a variable:
  `real x`
Real type (2/2)

- Smallest non-null positive real value: tiny(0.) (usually # 10^{-38})
- Largest real value: huge(0.) (usually # 10^{38})
- Smallest $\varepsilon$ such that $1. + \varepsilon > 1.$: epsilon(0.) (usually # 10^{-7})

(These three properties depend on the compiler)
Character type (1/2)

• Write character literal constants between single or double quotes. Examples:
  "*" 'A' '9' "x = ?"
  "Isn't valid"
  'Created file "plouf.txt"'.

Character type (2/2)

- Length of a string: number of characters. Examples of declaration of variables:
  `character(len=5) a`  
  `character b`  
  (default length is 1)

- Concatenation operator: `//`
Logical type

• Logical literal constants:
  `.false. `.true.`

• Example of declaration of variable:
  `logical found`

• Logical operators:
  `.not. `.or. `.and. `.eqv. `.neqv.`
Nota bene

Logical and integer types are different. You cannot write an integer value in lieu of a logical one (unlike in other languages, such as C or Python).
Declaration of a named constant (1/2)

- Adding the `parameter` attribute to the declaration of an object means the object is a named constant and not a variable.
- You have to provide the value of the named constant in its declaration statement.
Declaration of a named constant (2/2)

- Examples:
  ```fortran
  real, parameter:: pi = 3.14159265
  integer, parameter:: m = 5, n = 10
  ```

- Nota bene: the coma before `parameter` and the double colon “::” are compulsory.
Assignment, expressions
Assignment

• receiving variable = expression
  Example: $n = n + 1$

• The variable and the expression must be both of numerical type, or both logical, or both of character type.
Precedence rules

There are precedence rules between operators (numeric first, then relational, then logical, / and * before + and -, etc.). Use parentheses if necessary.

Examples:

\[ a / b / c \] is the same as: \[ a / (b * c) \]

\[ x > y \ .or. \ i == k \ .and. \ j == l \] is the same as:

\[ x > y \ .or. \ (i == k \ .and. \ j == l) \]

(.and. has precedence over .or.)
Conversion between numeric types (1/2)

- Implicit conversion when assigning a value. Examples:
  
  ```
  integer i
  real r
  i = -1.9 ! receives -1 (truncation, not rounding)
  r = 2 ! receives 2.
  ```
Conversion between numeric types (2/2)

- For explicit conversion, use the functions `floor` and `real`. Examples: `floor(a) real(i)`

- Automatic conversion to real type during evaluation of an expression. Example: `1. / 2`
Reminder on integer division

- The integral part of the ratio of two integer numbers is written in Fortran directly with the `/` operator, without need for conversion, without calling `floor` nor `real`.

- Example:
  ```fortran
  integer i, j
  Mathematical notation: \[ \left\lfloor \frac{i+j}{2} \right\rfloor \]
  
  In Fortran, if `i + j` is positive, write simply: `(i + j) / 2`
Substring and assignment to character string (1/2)

• Substring:
  \[ v(d:f) \quad v(d:) \quad v(:f) \]
  The position of the first character is 1 (and not 0, unlike in Python).

• Examples:
  character(len=8) mot
  mot = "aversion"
  mot(2:5) equals "vers"
  mot(6:6) equals "i"
  mot(2:1) equals ""
Substring and assignment to character string (2/2)

- Assignment: the expression if truncated or padded on the right side with blanks, if necessary.

- Note: in Fortran, you can modify part of a character variable. Example:
  ```fortran
  mot = "aversion"
mot(3:) = "ion"
mot equals "avion   "
  ```
  (unlike in Python, there is no such thing as an immutable type in Fortran)
Input and output
Printing to screen

- `print *`, list of expressions
- `print *` without a list to skip a line
Reading from the keyboard

- `read *`, list of variables
- Values read are interpreted automatically according to the type of the corresponding variable (so you do not write code to convert from character type, as in Python).
- If the variable is of type character, there is usually no need to delimit the corresponding input values with quotes.
- You can type input values on several lines.
Selection control structure
The **if** construct (1/2)

```java
if (logical expression 1) then
    block 1
{else if (logical expression 2) then
    block 2
...
else if (logical expression n) then
    block n}
{else
    block n + 1}
end if
```
The **if** construct (2/2)

- One block at most is executed.
- Note the compulsory parentheses around each logical expression.
Iteration control structure
The « do » loop (1/2)

• `do` variable = `expr1`, `expr2{, `expr3}`
  block
end do

• The variable must be of integer type and it is forbidden to modify it inside the loop.
• The 3 expressions must be of integer type.
  • `expr1` is the initial value.
  • `expr2` is the limiting value.
  • `expr3` is the increment (must be ≠ 0 if present).
The « do » loop (2/2)

- The 3 expressions are evaluated only once, when the do construct is encountered.
- The default value of the increment is 1.
- The number of iterations can be 0 (this does not produce a run-time error). For example if expr2 < expr1 and increment is 1.
The « do while » construct

- `do while (condition)`
  block
  end do

- Note that the parentheses delimiting the condition are compulsory.
Arrays
Definitions (1/3)

- Rank 0: Scalar
- Rank 1: Vector
- Rank 2: Arrays
- Rank 3: 
- ...
Definitions (2/3)

array a(m, n, p)
total size: \( m \times n \times p \)

shape of “a”: \[ \begin{bmatrix} m & n & p \end{bmatrix} \]
length of shape = rank of “a”
Definitions (3/3)

- Note: an array may have a null size, it is no problem (if a null-size array appears in an expression, no operation is done).
Declaration of an array with constant shape (1/2)

- Constant explicit shape: the (lower bounds and) extents are known at compile-time.

- Example:
  ```
  real a(10), b(10), c(10)
  ```

- The default lower bound is 1. But you can choose another lower bound. Example:
  ```
  real x(-2:3, 5)
  ```
Declaration of an array with constant shape (2/2)

- You can use named constants for the extents. Example:
  
  ```
  integer, parameter:: m = 10, n = 5
  real a(m,n)
  ```
Array constructor

- [list of scalar expressions]
- All the expressions must have the same type.
- Examples:
  - real v(3)
    v = [3.2, 4., 6.51]
  - character(len=3), parameter:: &
    currency(3) = ["EUR", "FRA", "USD"]
Selection of an array element

• Through a list of subscripts. 
  Number of subscripts: rank of the array. 
  Each subscript must be an integer expression.

• Example:
  \[ \text{t3}(i * j, 1, k / i + 2) \]
Choose an arithmetic progression for one or more dimensions.

\[ a : b : r \]

- \( a \) : first subscript, defaults to lower bound
- \( b \) : limiting subscript, defaults to upper bound
- \( r \) : stride, defaults to 1, may be \(< 0\)

Examples:

\[ t1(m : n), t1(m + n : n : -1) \]

\[ \text{mat}(i, :) \), mat(:i, j:) \]
Array expression (1/2)

- All the intrinsic operators (arithmetic, logical, relational, character) apply to array operands.
- Array operands must be *conformable*: they must have the same shape.
- A scalar and an array are also considered as conformable.
- The result is an array with the same shape as the operands.
Array expression (2/2)

• Examples:
  
  • Scalar and array:
    mat / 2
  
  • Several arrays:
    mat(:, 0) * t3(:, 1, 1)
    real a(5), b(5)
    a == b: logical array
Array assignment

- You can assign an array expression to a conformable array.
- Examples:
  
a = a(10:1:-1)
mat = 0
a(::2) = 0.
Reading or printing an array

In a read or print statement, you can use not only array elements but also whole arrays (or array sections). Examples:

```fortran
integer a(10, 2, 3), b(8)
```

Not only:
```fortran
read *, a(3, 2, 1)
print *, a(4, 1, 2)
```

but also possible:
```fortran
read *, b
print *, a(5:, :, 2)
```
Array element order (1/2)

• Reading or printing an array is done in “array element order”.

• Order: the subscripts along the first dimension vary most rapidly.

Exemple:
real a(2, 2)
a(1, 1), a(2, 1), a(1, 2), a(2, 2)
Array element order (2/2)

- Nota bene: array element order normally corresponds to storage order. So, if you need to loop over the elements of an array, do it in array element order.

- Example, which programming is more efficient?

```plaintext
do j = 1, n             do i = 1, m
  do i = 1, m           do j = 1, n
    a(i,j) = i+j         a(i,j) = i+j
  end do               end do
end do                 end do
```
Example of printing an array

integer a(10, 2, 3)
print *, a(5::, ::, ::)
prints:
a(5, 1, 1), a(6, 1, 1), ..., a(10, 1, 1),
a(5, 2, 1), ...
on as many lines as necessary (newlines may be
inserted, depending on the compiler).
Reading an array

read *, array
When the program runs, the user can enter elements in the terminal, separated by commas, space or newlines.
Allocatable arrays (1/2)

• When you do not know the extents at compile-time.

• Declare only the rank of the array, using a colon character : for each dimension

• Add the attribute allocatable real, allocatable:: a(:,), b(:, , :)


Allocatable arrays (2/2)

- Set the shape of the array:
  - Implicitly by assigning an array expression:
    \[ a = [3, 5, 1] \]
  - Or explicitly with an allocate statement:
    \[ \text{allocate}(a(n:m), b(p, q)) \]
    This is useful if, afterwards, you are not going to define the whole array in a single statement.
The thing to remember about arrays

- Fortran allows you to use not only array elements but also whole arrays or array sections in expressions, assignments, input and output statements.

- You improve concision and clarity by using this feature instead of programming loops on array elements.
Standard intrinsic functions
Standard intrinsic functions

• Many of them
• Useful: for concision, clarity, speed
• Difficulty: to guess there is an intrinsic function for what you want to compute
• Advice: skim the list of intrinsic functions. Cf. §13.5, physical pages 310 to 314, in the Fortran standard. Detailed descriptions are in §13.7, physical page 316.
Elemental intrinsic functions

- An elemental function can take either a scalar or an array argument. The function correspondingly returns a scalar or an array with the same shape.
- If the argument is an array, then the operation is applied to each element of the array.
- Example: \( \sin(x), \sin([x, 2x, 3x]) \)
Intrinsic functions for type conversion

- `int`, `floor`, `ceiling`, `nint`, `real`, etc.
- All of them are elemental.
Elemental functions for numeric computation (1/2)

- **abs**
  (of an integer or real number)

- **mod, modulo**

- **max, min**

  Example:
  
  \[
  \begin{align*}
  a & \text{ equals } [1, 2, 0, 0] \\
  b & \text{ equals } [2, 1, 2, 3] \\
  c & \text{ equals } [3, 1, 0, 4] \\
  \text{max}(a, b, c) & \text{ equals } [3, 2, 2, 4]
  \end{align*}
  \]

- **sign**
Elemental functions for numeric computation (2/2)

- \( \text{acos, asin, atan, cos, sin, tan} \)
- \( \text{atan2} \)
  \[ \text{atan2}(x, y) : \text{returns the argument in } [-\pi, \pi] \text{ of } x+iy \]
- \( \text{sqrt, log, log10, exp, cosh, sinh, tanh} \)

Etc.
The elemental intrinsic function \texttt{merge}

- To merge two arrays. \texttt{merge(tsource, fsourc e, mask)} returns \texttt{tsource} where \texttt{mask} is true and \texttt{fsource} elsewhere

- \texttt{tsource} and \texttt{fsource} must have the same type and be conformable.

- \texttt{mask} : logical, conformable with \texttt{tsource} and \texttt{fsource}
Example use of `merge`

\[ T = \text{merge}(\text{land}_T, \text{sst}, \text{land}_\text{mask}) \]
Array **reduction** intrinsic functions (1/2)

- Basic use case: a reduction function applies some operation to a whole array and returns a single scalar (« reduction » : from array to scalar)

- On a numerical array: 
  *sum, product, minval, maxval*

- On a logical array: 
  *any (logical or), all (logical and), count* 
  (number of true values)
Array reduction intrinsic functions
(2/2)

\[
m = \begin{bmatrix}
1 & 3 & 5 \\
2 & 4 & 6 \\
\end{bmatrix}
m2 = \begin{bmatrix}
0 & 3 & 5 \\
7 & 4 & 8 \\
\end{bmatrix}
\]

• Examples:
  \( \text{any}(m == m2) \) returns `.true.`
  \( \text{all}(m == m2) \) returns `.false.`
  \( \text{count}(m == m2) \) returns 3
  \( \text{sum}(m) \) returns 21
Dot product

dot_product(vector_a, vector_b)
On vectors of the same size. Equivalent to:
sum(vector_a * vector_b)
Better to use dot_product, clearer and possibly faster.
Matrix product and transpose

- \texttt{matmul(matrix\_a, matrix\_b)}
  - 2 arrays of rank 2 or 1 array of rank 2 and 1 vector
  - The first subscript for a rank 2 array is here assumed to be the row subscript.
  - Constraint on extents so that the matrix product is well defined.

- \texttt{transpose(matrix)}
  On a rank-2 array of any type.
Location of an extremum

- `minloc(array[, mask])`
- `maxloc(array[, mask])`
  
  Result: vector of subscripts of the first extremum found

- Examples:
  
  \[ V: [7, 6, 9, 6] \]
  
  \[ minloc(V) \text{ returns } [2] \]
  
  \[ A:\begin{bmatrix} 8 & -3 & 0 & -5 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & 6 & 4 \end{bmatrix} \]
  
  \[ minloc(A, \text{ mask}=A>-4) \text{ returns } [1, 2] \]