

# INTRODUÇÃO À PROGRAMAÇÃO EM FORTRAN 90

Resumo do livro

FORTRAN 90 PROGRAMMING  
T. M. R. Ellis, Ivor R. Philips, Thomas M. Lahey  
Addison-Wesley: 1994

F. J. ROMEIRAS

Novembro de 2003

Apontamentos para a disciplina de  
Análise e Simulação Numérica

## FIRST STEPS IN FORTRAN 90 PROGRAMMING

- A program is composed of the **main program unit** and program units of other types, in particular **subroutines**.
- The structure of a **main program unit** is:

```
PROGRAM name
  Specification statements
  ...
  Executable statements
  ...
END PROGRAM name
```

- **Specification statements** provide information about the program to the compiler.
  - The IMPLICIT NONE statement.
  - The **variable declaration**.
- The **Executable statements**.
  - **list-directed input statements** – ex: via the keyboard
  - **list-directed output statements** – ex: via the screen
  - CALL statements are used to execute **subroutines**.
- A Fortran 90 **name** must obey the following rules:
  - it must consist of a maximum of 31 characters;
  - it may only contain: A–Z, a–z, 0–9 and `_`; upper and lower case letters are considered to be identical in this context;
  - it must begin with a letter.
- **Keywords** are Fortran names which have a special meaning in the Fortran language; other names are called **identifiers**.

- A **comment line** is a line whose first non-blank character is an exclamation mark, !.
- A line may contain a maximum of **132** characters.
- A line may contain more than one statement, separated by a ;.
- **Continuation line** – a long line can be broken by & . Next line should begin with &.
- A statement may have a maximum of **39** continuation lines.
- Errors in programs are of different types. A **syntactic error** is an error in the syntax, or grammar, of the statement. A **semantic error** is an error in the logic of the program; that is, it does not do what it was intended to do. **Compilation errors** are errors detected during the compilation process. **Execution errors** are errors that occur during the execution of the compiled program.

Problem (1)

Write a program which will ask the user for the  $x$  and  $y$  coordinates of three points and which will calculate the equation of the circle passing through those three points, namely

$$(x - a)^2 + (y - b)^2 = r^2$$

and then display the coordinates  $(a, b)$  of the centre of the circle and its radius  $r$ .

Analysis (1)

- |   |
|---|
| <ol style="list-style-type: none"><li><b>1</b> Read three sets of coordinates <math>(x_1, y_1)</math>, <math>(x_2, y_2)</math> and <math>(x_3, y_3)</math></li><li><b>2</b> Calculate the equation of the circle using the procedure <i>calculate_circle</i></li><li><b>3</b> Display the values <math>a, b</math> and <math>r</math></li></ol> |
|---|

Solution (1)

```
PROGRAM circle
  IMPLICIT NONE
  !
  ! This program calculates the equation of a circle passing
  ! through three points
  !
  ! Variable declarations
  !
  REAL :: x1, y1, x2, y2, x3, y3, a, b, r
  !
  ! Step 1
  !
  PRINT *, "Please type the coordinates of three points"
  PRINT *, "in the order x1, y1, x2, y2, x3, y3"
  READ *, x1, y1, x2, y2, x3, y3      ! Read the three points
  !
  ! Step 2
  !
  CALL calculate_circle(x1, y1, x2, y2, x3, y3, a, b, r)
  !
  ! Step 3
  !
  PRINT *, "The centre of the circle through these points is &
          &(", a, ",", b, ")"
  PRINT *, "Its radius is ", r
  !
END PROGRAM circle
```

Result of running the Solution (1)

```
Please type the coordinates of three points
in the order x1, y1, x2, y2, x3, y3
4.71 4.71
6.39 0.63
0.63 0.63
The centre of the circle through these points is ( 3.510, 1.830)
Its radius is  3.120
```

## ESSENCIAL DATA HANDLING

- Data types: INTEGER, REAL, CHARACTER, user defined, etc.
- The **type declaration statement**

```
REAL :: real_1, real_2, real_3
INTEGER :: integer_1, integer_2
CHARACTER(10) :: name_1, name_2
```

- An IMPLICIT NONE forces the compiler to require that all variables appear in a type declaration statement.
- An **assignment statement**

```
a = b + c*d/e - f**g/h + i*j + k
a = b + (c*d)/e - (f**g)/h + (i*j) + k
```

- If an integer value is assigned to a real variable it is converted to its real equivalent before assignment; if a real value is assigned to an integer variable it is truncated before conversion to integer, and any fractional part is lost.
- Arithmetic operators in Fortran:

<i>Operator</i>	<i>Meaning</i>	<i>Priority</i>
+	Addition	Low
-	Subtraction	Low
*	Multiplication	Medium
/	Division	Medium
**	Exponentiation	High

- If one of the operands of an arithmetic operator is real, then the evaluation of that operation is carried out using real arithmetic, with any integer operand being converted to real.

- The **integer division**

```

REAL :: C, F, F_1, F_2, F_3, F_4
...
F = 9.0 * C/5.0 + 32.0
F_1 = 9.0/5.0 * C + 32.0    ! F_1 = F
F_2 = 1.8 * C + 32.0       ! F_2 = F
F_3 = 9 * C/5 + 32         ! F_3 = F
F_4 = 9/5 * C + 32         !!! F_4 /= F

```

- **Constants**

- **literal constants** (ex: exponential form mEe)

```
temp = 3*aux/2.5 + 0.1E-5/1E-6
```

- constants with name – initial value

```

REAL :: a = 0.0,  b,  c = 1.0E-6
INTEGER :: n_max = 100
CHARACTER(LEN=10) :: name="Undefined"

```

- constants with name – PARAMETER

```

REAL, PARAMETER :: pi = 3.1415926
INTEGER, PARAMETER :: max_iter = 100

```

- **List-directed input/output statements**

```

READ *, var_1, var_2, ...
PRINT *, "O produto custa ",var_3," Euros."

```

- READ – only variables
- PRINT – variables, constants and expressions
- A value that is to be input to an integer variable must **not** contain a decimal point, and the occurrence of one will cause an error.

- **Value separator**

- a comma, a space, a slash (/) or the end of the line;
- If there are two consecutive commas, then the effect is to read a **null value**, which results in the value of the corresponding variable in the input list being left unchanged.
- If the terminating character is a slash (/) then no more data items are read, and processing of the input statement is ended.

- **Character variables – Fortran Character Set**

---

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z  
 (a b c d e f g h i j k l m n o p q r s t u v w x y z)  
 0 1 2 3 4 5 6 7 8 9  
 \_ # = + - \* / ( ) , . ' : ! " % & ; < > ? \$

(where # represents the space, or blank, character)

---

- **declaration**

CHARACTER(LEN = *length*) :: *name\_1*, *name\_2*, ...

CHARACTER(*length*) :: *name\_1*, *name\_2*, ...

CHARACTER\**length* :: *name\_1*, *name\_2*, ...

- if LEN is not specified then LEN=1 is assumed
- input/output – between " " or ' '
- **concatenation** – double slash (//)

- A **derived type** is a user-defined data type, each of whose components is either an intrinsic type or a previously defined derived type.

```

TYPE person
  CHARACTER :: first_name*12, middle_initial*1, last_name*12
  INTEGER :: age
  CHARACTER :: sex ! M or F
  CHARACTER(LEN=11) :: social_security
END TYPE person

```

- declaration

```
TYPE(person) :: jack, jill
```

- assignment

```
jack = person("Jack", "R", "Brown", 47, "M", "123-45-6789")  
jill = person("Jill", "M", "Smith", 39, "F", "987-65-4321")
```

- referring to a component of a derived type variable

```
jill%last_name = jack%last_name
```

Problem (2)

Define a data type which can be used to represent complex numbers, and then use it in a program which reads two complex numbers and calculates and prints their sum and their product.

Analysis (2)

Given two complex numbers

$$z_1 = x_1 + iy_1, \quad z_2 = x_2 + iy_2,$$

the rules for addition and multiplication are the following:

$$z_1 + z_2 = x_1 + x_2 + i(y_1 + y_2)$$

$$z_1 \times z_2 = x_1 \times x_2 - y_1 \times y_2 + i(x_1 \times y_2 + x_2 \times y_1)$$

- |   |
|---|
| <ol style="list-style-type: none"><li><b>1</b> Define a data type for complex numbers</li><li><b>2</b> Read two complex numbers</li><li><b>3</b> Calculate their sum and their product</li><li><b>4</b> Print results</li></ol> |
|---|

Solution (2)

```
PROGRAM complex_arithmetic
  IMPLICIT NONE
  !
  ! A program to illustrate the use of a derived type to perform
  ! complex arithmetic
  !
  ! Type definition
  TYPE complex_number
    REAL :: real_part, imag_part
  END TYPE complex_number
  !
  ! Variable definitions
  TYPE(complex_number) :: z1, z2, sum, prod
  !
  ! Read data
  PRINT *, "Please supply two complex numbers"
  PRINT *, "Each complex number should be typed as two numbers,"
  PRINT *, "representing the real and imaginary parts of the number"
  READ *, z1, z2
  !
  ! Calculate sum and product
  sum%real_part = z1%real_part + z2%real_part
  sum%imag_part = z1%imag_part + z2%imag_part
  !
  prod%real_part = z1%real_part * z2%real_part - &
    & z1%imag_part * z2%imag_part
  prod%imag_part = z1%real_part * z2%imag_part + &
    & z1%imag_part * z2%real_part
  !
  ! Print results
  PRINT *, "The sum of the two numbers is (", &
    & sum%real_part, ", ", sum%imag_part, ")"
  PRINT *, "The product of the two numbers is (", &
    & prod%real_part, ", ", prod%imag_part, ")"
  !
END PROGRAM complex_arithmetic
```

## BASIC BUILDING BLOCKS

### FUNCTION in Fortran 90

- intrinsic function (ex: SQRT(x), EXP(x), etc.)
- external functions (user-defined)

```
type FUNCTION name( dum_1, dum_2, ... )  
  IMPLICIT NONE  
  ...  
  Specification statements, etc.  
  ...  
  Executable statements  
  ...  
END FUNCTION name
```

or

```
FUNCTION name( dum_1, dum_2, ... )  
  IMPLICIT NONE  
  type :: name  
  ...  
  Specification statements, etc.  
  ...  
  Executable statements  
  ...  
END FUNCTION name
```

- arguments should be defined as INTENT(IN)
- local variables have no meaning outside the function
- declaration in the main program unit

```
REAL, EXTERNAL :: function_name
```

## SUBROUTINE in Fortran 90

- general form

```
SUBROUTINE name(dum_1, dum_2, ...)
  IMPLICIT NONE
  ...
  Specification statements, etc.
  ...
  Executable statements
  ...
END SUBROUTINE name
```

- arguments should be defined as INTENT(IN/OUT/INOUT)
- local variables have no meaning outside the subroutine
- a subroutine may have no arguments (ex: CALL *name*)
- a subroutine may return nothing
- Do NOT use **recursion** in subroutines (!)
- a subroutine is accessed with the CALL statement

## MODULE in Fortran 90

- permits **global accessibility** of variables, constants and derived type definitions
- general form

```
MODULE name
  IMPLICIT NONE
  SAVE
  ...
  Other specification statements, etc.
  ...
  Executable statements
  ...
  CONTAINS
  ...
END MODULE name
```

- One module can USE another module
- A module may not USE itself, either directly or indirectly
- accessed through a USE statement (`USE module_name`)
- the USE statement comes after the initial statement (PROGRAM, SUBROUTINE or FUNCTION) but before any other statements.
- a MODULE may contain FUNCTIONS or SUBROUTINES, placed after a CONTAINS statement

Problem (3)

Write a program which will demonstrate the use of the function *cube\_root* to calculate the cube root of a positive real number.

Analysis (3)

- 1 Read positive real number *pos\_num*.
- 2 Obtain *root\_3* by reference to the function *cube\_root*.
- 3 Print *pos\_num* and *root\_3*.

Problem (4)

Write a program which will demonstrate the use of the subroutine *roots* to calculate the square root, the cube root and the fourth root of a positive real number.

Analysis (4)

- 1 Read positive real number *pos\_num*.
- 2 Obtain *root\_2*, *root\_3* and *root\_4* by calling the subroutine *roots*.
- 3 Print *pos\_num*, *root\_2*, *root\_3* and *root\_4*.

### Solution 3

```
PROGRAM function_demo
  IMPLICIT NONE
  !
  REAL, EXTERNAL :: cube_root
  REAL :: pos_num, root_3
  !
  PRINT *, "Please type a positive real number: "
  READ *, pos_num
  !
  root_3=cube_root(pos_num)
  !
  PRINT *, "The cube root of ", pos_num, " is ", root_3
  !
END PROGRAM function_demo
```

```
REAL FUNCTION cube_root(x)
  IMPLICIT NONE
  !
  ! Function to calculate the cube root of a positive real number
  !
  ! Dummy argument declaration
  REAL, INTENT(IN) :: x
  !
  ! Local variable declaration
  REAL :: log_x
  !
  ! Calculate cube root by using logs
  log_x = LOG(x)
  cube_root = EXP(log_x/3.0)
  !
END FUNCTION cube_root
```

## Solution 4

```
PROGRAM subroutine_demo
  IMPLICIT NONE
  !
  REAL :: pos_num, root_2, root_3, root_4
  !
  PRINT *, "Please type a positive real number: "
  READ *, pos_num
  !
  CALL roots(pos_num, root_2, root_3, root_4)
  !
  PRINT *, "The square root of ", pos_num, " is ", root_2
  PRINT *, "The cube root of ", pos_num, " is ", root_3
  PRINT *, "The fourth root of ", pos_num, " is ", root_4
  !
END PROGRAM subroutine_demo
!-----
SUBROUTINE roots(x, square_root, cube_root, fourth_root)
  IMPLICIT NONE
  !
  ! Subroutine to calculate various roots of a positive real number,
  ! supplied as the first argument, and return them in the
  ! second to fourth arguments
  !
  ! Dummy argument declarations
  REAL, INTENT(IN) :: x
  REAL, INTENT(OUT) :: square_root, cube_root, fourth_root
  !
  REAL :: log_x      !!! Local variable declarations
  !
  ! Calculate square root using intrinsic SQRT
  square_root = SQRT(x)
  !
  ! Calculate other roots by using intrinsic LOG and EXP
  log_x = LOG(x)
  cube_root = EXP(log_x/3.0)
  fourth_root = EXP(log_x/4.0)
  !
END SUBROUTINE roots
```

Problem (5)

Write two functions for use in a complex arithmetic package using the `complex_number` derived type which was created in Example 3.4. The functions should each take two complex arguments and return as their result the result of adding and multiplying the two numbers.

Analysis (5)

This was already done in Problem 2.

- 1** Place the derived type `complex_number` in a MODULE `complex_data` for USE association by the program and the functions
- 2** Read two complex numbers
- 3** Calculate their sum using FUNCTION `c_add`
- 4** Calculate their product using FUNCTION `c_mult`
- 5** Print the results

Solution (5)

```
MODULE complex_data
  IMPLICIT NONE
  SAVE
  !
  TYPE complex_number
    REAL :: real_part, imag_part
  END TYPE complex_number
  !
END MODULE complex_data

PROGRAM complex_example
  USE complex_data
  IMPLICIT NONE
  !
  TYPE(complex_number), EXTERNAL :: c_add, c_mult
  TYPE(complex_number) :: z1, z2
  !
  PRINT *, "Please supply two complex numbers as two pairs &
    &of real numbers"
  PRINT *, "Each pair represents the real and imaginary parts &
    &of a complex number"
  READ *, z1, z2
  !
  ! Calculate and print sum and product
  PRINT *, "The sum of the two numbers is ", c_add(z1, z2)
  PRINT *, "The product of the two numbers is ", c_mult(z1, z2)
  !
END PROGRAM complex_example
```

```
FUNCTION c_add(z1, z2)
  USE complex_data
  IMPLICIT NONE
  !
  TYPE(complex_number) :: c_add
  TYPE(complex_number), INTENT(IN) :: z1, z2
  !
  c_add%real_part = z1%real_part + z2%real_part
  c_add%imag_part = z1%imag_part + z2%imag_part
  !
END FUNCTION c_add
```

```
FUNCTION c_mult(z1, z2)
  USE complex_data
  IMPLICIT NONE
  !
  TYPE(complex_number) :: c_mult
  TYPE(complex_number), INTENT(IN) :: z1, z2
  !
  c_mult%real_part = z1%real_part * z2%real_part - &
    & z1%imag_part * z2%imag_part
  c_mult%imag_part = z1%real_part * z2%imag_part + &
    & z1%imag_part * z2%real_part
  !
END FUNCTION c_mult
```

## CONTROLLING THE FLOW OF YOUR PROGRAM

- A **logical expression** can take one of the two logical values—*true* or *false*

- relational expression – `a.GT.b`, `a <= b` `a.EQ.b`, `a /= b`
- arithmetic operations have a higher priority

```
b**2 >= 4*a*c           ! these two
b**2 - 4*a*c >= 0      ! are equivalent
```

- **Logical variables** (`.TRUE.` or `.FALSE.`) – declaration

```
LOGICAL :: var_1, var_2
```

- **FUNCTIONs** which deliver a logical value

```
FUNCTION logical_fun(var_1, var_2)
  LOGICAL :: logical_fun
  ...
```

- **Logical operators**, `.OR.` ; `.AND.` ; `.EQV.` ; `.NEQV.` – combine two logical expressions

L1	L2	L1.OR.L2	L1.AND.L2	L1.EQV.L2	L1.NEQV.L2
true	true	true	true	true	false
true	false	true	false	false	true
false	true	true	false	false	true
false	false	false	false	true	false

- The logical operator `.NOT.`
  - Any arithmetic operators or relational operators (*in that order*) have a higher priority than any logical operators
- ```
(.NOT.(a < b).OR.(c < d)).EQV.((x < y).OR.(z < d**2))
```

- **The block IF construct in Fortran 90**

```
IF (logical_expression) THEN
    block of Fortran statements
ELSE IF (logical_expression) THEN
    block of Fortran statements
...
ELSE
    block of Fortran statements
END IF
```

- **Logical IF statement** – IF (*logical\_expr*) *Fortran statement*

- **The SELECT CASE construct in Fortran 90**

```
SELECT CASE (case_expression)
CASE (case_selector)
    block of Fortran statements
CASE (case_selector)
    block of Fortran statements
...
CASE DEFAULT
    block of Fortran statements
END SELECT
```

- *case\_expression* is either an integer expression, a character expression or a logical expression; real expressions are *prohibited* for this purpose.
- The *case\_selector* can take one of four forms:
  - case\_value*
  - low\_value* :
  - : high\_value*
  - low\_value* : *high\_value*
- the decision criteria **must not overlap**.

Problem (6)

Write a program to read the coefficients of a quadratic equation and print the roots.

Analysis (6)

The program will use the formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

where

$$ax^2 + bx + c = 0 \quad \text{and} \quad a \neq 0.$$

There are three possible cases:

- (1)  $b^2 - 4ac \geq \varepsilon$  Equation has two real roots
- (2)  $|b^2 - 4ac| < \varepsilon$  Equation has one real root
- (3)  $b^2 - 4ac \leq -\varepsilon$  Equation has no real roots

$\varepsilon$  is a very small positive number (ex:  $\varepsilon = 10^{-6}$ )

Solution (6a) – using a block IF construct

```
PROGRAM quadratic_by_block_IF
  IMPLICIT NONE
  !
  ! A program to solve a quadratic equation using a block IF
  ! statement to distinguish between the three cases
  !
  ! Constant declarations
  REAL, PARAMETER :: epsilon = 1.0E-6
  !
  ! Variable declarations
  REAL :: a, b, c, d, sqrt_d, x1, x2
  !
  ! Read coefficients
  PRINT *, "Please type the three coefficients a, b and c"
  READ *, a, b, c
  !
  ! Calculate  $b^2-4ac$ 
  d = b**2 - 4.0 * a * c
  !
  ! Calculate and print roots, if any
  IF (d >= epsilon) THEN
    ! Two roots
    sqrt_d = SQRT(d)
    x1 = (-b + sqrt_d)/(a+a)
    x2 = (-b - sqrt_d)/(a+a)
    PRINT *, "The equation has two roots: ", x1, " and ", x2
  ELSE IF (d > -epsilon) THEN
    ! One root
    x1 = -b/(a+a)
    PRINT *, "The equation has one root: ", x1
  ELSE
    ! No roots
    PRINT *, "The equation has no real roots"
  END IF
  !
END PROGRAM quadratic_by_block_IF
```

Solution (6b) – using a CASE construct

```
PROGRAM quadratic_by_case
  IMPLICIT NONE
  !
  ! Constant declarations
  REAL, PARAMETER :: epsilon = 1.0E-6
  !
  ! Variable declarations
  REAL :: a, b, c, d, sqrt_d, x1, x2
  INTEGER :: selector
  !
  ! Read coefficients
  PRINT *, "Please type the three coefficients a, b and c"
  READ *, a, b, c
  !
  ! Calculate  $b^2-4ac$  and resulting case selector
  d = b**2 - 4.0 * a * c
  selector = d/epsilon
  !
  ! Calculate and print roots, if any
  SELECT CASE (selector)
  CASE (1:)
    ! Two roots
    sqrt_d = SQRT(d)
    x1 = (-b + sqrt_d)/(a+a)
    x2 = (-b - sqrt_d)/(a+a)
    PRINT *, "The equation has two roots: ", x1, " and ", x2
  CASE (0)
    ! One root
    x1 = -b/(a+a)
    PRINT *, "The equation has one root: ", x1
  CASE (:-1)
    ! No roots
    PRINT *, "The equation has no real roots"
  END SELECT
  !
END PROGRAM quadratic_by_case
```

## REPEATING PARTS OF YOUR PROGRAM

### The DO loop in Fortran 90

```
DO count = initial, final, inc
  ...
  block of statements
  ...
  IF (logical_expression) EXIT
  ...
  IF (logical_expression) CYCLE
  ...
  block of statements
  ...
END DO
```

- *count*, *initial*, *final*, *inc* should be **integers** (\*)
- *inc* is optional. DO without *count* is also possible
- several DO loops may be **nested**
- EXIT – aborts the whole DO loop
- CYCLE – aborts the current iteration
- STOP – terminates the whole program
- RETURN – terminates the current procedure
- GOTO – transfers execution to the statement in the same procedure having a specified label.
- A **statement label** – 1 to 5 digit integer followed by a space

Problem (7)

Write a program to generate the Fibonacci sequence of numbers,

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \dots$$

until the absolute value of the difference between the ratio of two consecutive numbers  $\frac{x_{n+1}}{x_n}$  and the Golden Ratio  $\frac{\sqrt{5} + 1}{2}$  is less than some arbitrary small value  $\varepsilon$ .

Analysis (7)

The Fibonacci sequence is generated by the formula

$$x_{n+1} = x_n + x_{n-1}, \quad n = 1, 2, \dots$$

The exit criterion will be:

$$\left| \frac{x_{n+1}}{x_n} - \frac{\sqrt{5} + 1}{2} \right| < \varepsilon$$

*Structure plan:*

- 1** Initialize  $x1$  and  $x2$  to 1
- 2** Read  $\epsilon$  and  $max\_iterations$
- 3** Repeat  $max\_iterations$  times:
  - 3.1** Calculate next term  $x = x1 + x2$  in the sequence
  - 3.2** Calculate  $ratio$  and  $error$
  - 3.3** If  $error \leq \epsilon$  then exit
  - 3.4** Otherwise, set  $x1 = x2$  and  $x2 = x$
- 4** Print results with indication of ERROR if the loop was executed  $max\_iterations$  times without the exit condition being verified

Solution (7)

```
PROGRAM Fibonacci_sequence
  IMPLICIT NONE
  !
  ! This program illustrates that the ratio of two consecutive
  ! numbers of the Fibonacci sequence converges to the
  ! Golden-Ratio
  !
  ! Constant and variable declarations
  INTEGER :: max_iterations, count, x, x1=1, x2=1
  REAL :: epsilon, ratio, error, golden_ratio
  !
  golden_ratio = (1.0+SQRT(5.0))/2.0
  !
  ! Read epsilon and max_iterations
  PRINT*, "epsilon=?, max_iterations=?"
  READ*,  epsilon, max_iterations
  !
  ! Loop to obtain sequence and error
  DO count=1, max_iterations
    x=x1+x2
    ratio=REAL(x)/REAL(x2)
    error=ABS(ratio-golden_ratio)
    IF(error < epsilon) EXIT
    x1=x2
    x2=x
  END DO
  !
  ! Output results
  IF(count == max_iterations+1) THEN
    PRINT*, count, x1, x2, x, error, "  ERROR"
  ELSE
    PRINT*, count, x1, x2, x, error
  END IF
  !
END PROGRAM Fibonacci_Sequence
```

## AN INTRODUCTION TO ARRAYS - ONE DIMENSIONAL

- **declaration** of an array

```
REAL, DIMENSION(50) :: a, b, c
REAL, DIMENSION(11:60) :: d, e, f
REAL :: a(50), b(50), c(50)
REAL :: d(11:60), e(11:60), f(11:60)
```

- intrinsic function SIZE → ex: SIZE(f) ! answer 50
- the **shape** of an array → in one dimension is equal to the **size**
- An array-valued **constant** is specified by an **array constructor**

```
array_name = (/ list of values /)
array_name = (/ (value_list, implied_do_control) /)
```

```
INTEGER, DIMENSION(10) :: array_1=(/1,2,3,4,5,6,7,8,9,10/)
INTEGER, DIMENSION(10) :: array_1=(/ (i, i=1,10) /)
INTEGER, DIMENSION(100) :: array_3= &
    &( / ( (0, i=1,9), 10*j, j=1,10 ) /)
```

- referring to an element of an array → a(5), b(INT(a(3)+c(4)))
- referring to a group of element → a(5:17), d(11:17)
- **input/output** of an array → whole array, element, implied DO

```
PRINT *, b(16), a
READ *, (a(2*i) , i=1,10,2)
```

- two arrays are **conformable** if they have the same shape

- a scalar or a constant is conformable with any array
- intrinsic operations are defined between two conformable objects
- intrinsic operations on arrays take place element-by-element
- an **elemental intrinsic procedure** may use an array as an argument

```
REAL :: a(1:20), b(1:20), c(1:20), d(1:20)
...
b = c / d
c = SIN(b)
```

- an **assumed-shape array** is a dummy argument array whose bounds are not specified in the declaration of the array

```
REAL, DIMENSION(10:30) :: a, b
...
CALL array_example(a,b)
...
SUBROUTINE array_example(dum_arr_1, dum_arr_2)
  IMPLICIT NONE
  REAL, DIMENSION(:) :: dum_arr_1, dum_arr_2
  ...
END SUBROUTINE array_example
```

- an **array valued function**

```
FUNCTION fun_1(arg_1,arg_2)
  IMPLICIT NONE
  REAL, DIMENSION(:), INTENT(IN) :: arg_1, arg_2
  REAL, DIMENSION(SIZE(arg_1)) :: fun_1
  ...
```

Problem (8)

Write a program to calculate: (1) the dot product of two three-dimensional vectors; (2) the vector product of two three-dimensional vectors; (3) the scalar triple product of three three-dimensional vectors.

Analysis (8)

Consider the two three-dimensional vectors

$$\mathbf{a} = (a_1, a_2, a_3), \quad \mathbf{b} = (b_1, b_2, b_3).$$

The dot product of the two vectors  $\mathbf{a}$  and  $\mathbf{b}$  is the scalar  $\mathbf{a} \cdot \mathbf{b}$  defined by:

$$\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2 + a_3 b_3$$

The vector product of the two vectors  $\mathbf{a}$  and  $\mathbf{b}$  is the vector  $\mathbf{c} = \mathbf{a} \times \mathbf{b}$  defined by:

$$\mathbf{c} = (a_2 b_3 - a_3 b_2, a_3 b_1 - a_1 b_3, a_1 b_2 - a_2 b_1)$$

The scalar triple product of the three vectors  $\mathbf{a}$ ,  $\mathbf{b}$  and  $\mathbf{c}$  is the scalar  $[\mathbf{abc}]$  defined by:

$$[\mathbf{abc}] = \mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})$$

*Structure plan:*

- 1** Read *selector* to choose the product required
- 2** Read two vectors in cases (1) and (2) and three vectors in case (3)
- 3** Calculate required product
- 4** Print result

Solution (8)

```
MODULE products
  IMPLICIT NONE
  SAVE
  !
  CONTAINS
  !
  !
  FUNCTION dot_product(a, b)
    IMPLICIT NONE
    REAL, DIMENSION(3), INTENT(IN) :: a, b
    REAL :: dot_product
    !
    dot_product = a(1)*b(1) + a(2)*b(2) + a(3)*b(3)
  END FUNCTION dot_product
  !
  FUNCTION vector_product(a, b)
    IMPLICIT NONE
    REAL, DIMENSION(3), INTENT(IN) :: a, b
    REAL, DIMENSION(3) :: vector_product
    !
    vector_product(1) = a(2)*b(3) - a(3)*b(2)
    vector_product(2) = a(3)*b(1) - a(1)*b(3)
    vector_product(3) = a(1)*b(2) - a(2)*b(1)
  END FUNCTION vector_product
  !
END MODULE products
```

```
PROGRAM test_products
  USE products
  IMPLICIT NONE
  REAL, DIMENSION(3) :: a, b, c, vp
  REAL :: dp, stp
  INTEGER :: selector
  !
  PRINT*, "selector = ? (1(dp), 2(vp), 3(stp))"
  READ*, selector
  !
  CASE SELECT (selector)
  CASE (1)
    PRINT*, "a = ?"
    READ*, a(1), a(2), a(3)
    PRINT*, "b = ?"
    READ*, b(1), b(2), b(3)
    dp = dot_product(a, b)
    PRINT*, "a . b = ", dp
  CASE (2)
    PRINT*, "a = ?"
    READ*, a(1), a(2), a(3)
    PRINT*, "b = ?"
    READ*, b(1), b(2), b(3)
    vp = vector_product(a, b)
    PRINT*, "a x b = ", vp
  CASE (3)
    PRINT*, "a = ?"
    READ*, a(1), a(2), a(3)
    PRINT*, "b = ?"
    READ*, b(1), b(2), b(3)
    PRINT*, "c = ?"
    READ*, c(1), c(2), c(3)
    vp = vector_product(b, c)
    stp = dot_product(a, vp)
    PRINT*, "a . b x c = ", stp
  END SELECT
  !
END PROGRAM test_products
```

## MORE CONTROL OVER INPUT AND OUTPUT

- The READ and PRINT statements

```

READ format_specifier, input_lis
READ *, input_lis

PRINT format_specifier, output_lis
PRINT *, output_lis

```

- list of input data: 123456789

```

READ '(I9)', n                ! n=123456789
READ '(I3, I3)', n1, n2      ! n1=123; n2=456
READ '(4X,I5)', num          ! num=56789
READ '(I2,3X,I3)', i, j      ! i=12; j=678
READ 10, x, y, z             ! x=45; y=89; z=2345

```

```
10 FORMAT(T4,I2,T8,I2,T2,I4)
```

- consider the real number  $x = 0.0000361764$

```

PRINT '(F10.4)', x           ! will print   ####0.0000
PRINT '(F12.6)', x           !             ####0.000036
PRINT '(F14.8)', x           !             ####0.00003618
PRINT '(E10.4)', x           !             0.3618E-04
PRINT '(E12.6)', x           !             0.361764E-04
PRINT '(E14.8)', x           !             0.36176400E-04

```

- a **repeat count** → *before* the I, F, E, A or L edit descriptors

- a **character constant** are allowed in the *format\_specifier*

```
PRINT '("The two integers are ", 2I5)', m, n
```

## USING FILES TO PRESERVE DATA

- READ and WRITE statements with **control information lists**

```

READ (cilist) input_list
WRITE (cilist) output_list

```

- the **control information list** *cilist* consists of **specifiers**

---

```

UNIT = unit_number  a positive integer
UNIT = *              the default i/o device
FMT = ch_var        '(.,.)'
FMT = label
FMT = *              list directed i/o
IOSTAT = io_status  integer (+,- or 0)

```

---

- OPEN a file for READ and WRITE

```

OPEN (open_specifier_list)

```

- *open\_specifier\_list* is a list of specifiers

---

```

UNIT = unit_number
FILE = file_name
STATUS = file_status
FORM = format_mode
ACTION = allowed_actions
POSITION = file_position
IOSTAT = ios

```

---

- the simplest form

```

OPEN(UNIT=4, FILE='data.txt', STATUS='UNKNOWN')
  WRITE(4,*) var_1, var_2, ...
CLOSE(UNIT=4)

```

Example:

```
WRITE (UNIT=6, FMT=201) a, b, a+b, a*b
201 FORMAT("1", T10, "Multi-record example"/           &
          "0", "The sum of", F6.2, " and", F6.2, " is", F7.2/ &
          1X, "Their product is", F10.3)
```

will cause the following output:

```
----- (new page}
      Multi-record example

The sum of 12.25 and 23.50 is  35.75
Their product is  287.875
```

Example:

```
READ '(3F8.2//3I6)', a, b, c, p, q, r
```

will read three real numbers from the first record and three integers from the third.

Example:

```
WRITE (UNIT=6, FMT=202) a, b, a+b, a*b
202 FORMAT("1" / T10, "Multi-record example"//       &
          "0", "The sum of", F6.2, " and", F6.2, " is", F7.2// &
          1X, "Their product is", F10.3)
```

will cause the following output:

```
----- (new page}
      Multi-record example

The sum of 12.25 and 23.50 is  35.75

Their product is  287.875
```

## HOW TO READ A DATA FILE IN MATHEMATICA

- the data file should contain only real numbers separated by a blank space. May be in exponential form  $\rightarrow 0.3463247E-005$
- the `SetDirectory["dir"]` command sets the current working directory.
- `OpenRead["file"]` opens a file to read data from, and returns an `InputStream` object.
- `ReadList["file", types, n]` reads the first `n` objects of the specified types.
- `Partition[list, n]` partitions list into non-overlapping sublists of length `n`.
- `ListPlot[{{x1, y1}, {x2, y2}, ...}]` plots a list of values with specified `x` and `y` coordinates.

```
SetDirectory["C:\\users\\trabalhos\\ASN"];  
...  
str1 = OpenRead["data.txt"];  
aaa = ReadList[str1, Number];  
Close[str1];  
...  
temp=Partition[aaa,2];  
ListPlot[temp,PlotJoined->True,AspectRatio->Automatic]
```

## SETTING THE NUMERICAL PRECISION OF REAL VARIABLES

- REAL, single-precision – aprox. 7 significant digits, 32 bits
- REAL, double-precision – aprox. 14 significant digits, 64 bits
- **the default precision is the *single-precision* !**
- KIND (precision) declaration – **compiler-dependent** !

```
REAL(KIND=4) :: x, f
REAL(KIND=3) :: s(8), t(5)
```

- recommended, **compiler-independent** form

```
INTEGER, PARAMETER :: DP=SELECTED_REAL_KIND(P=14,R=50)
...
REAL(KIND=DP) :: x, f
```

- SELECTED\_REAL\_KIND(P, R) – intrinsic **integer** function
  - P – minimal number of decimal digits required
  - R – minimal decimal exponent range required (optional)
  - the result is the kind type that meets, or minimally exceeds, the requirements.
  - if more than one kind type parameter meets the requirements, the value returned is the one with the smallest P.
- literal constants – `-1.57485_DP`
- double-precision intrinsic functions – `DEXP(x)`, `DASIN(x)`, ...

## ARRAY PROCESSING AND MATRIX MANIPULATION

- **Explicit-shape arrays**

```
REAL, DIMENSION(15, 50) :: p
REAL, DIMENSION(15, 15, 2:10) :: q
```

- the `SIZE(., DIM=.)` intrinsic function, `DIM` is optional
- the `RESHAPE(., .)` intrinsic function

```
RESHAPE((/ 1, 2, 3, 4, 5, 6 /), (/ 2, 3 /))
```

produces the matrix  $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$

- input and output of arrays

```
REAL, DIMENSION(50,6) :: x
...
PRINT '(6F8.2)', x
--- output ---
x(1,1)  x(2,1)  x(3,1)  x(4,1)  x(5,1)  x(6,1)
x(7,1)  x(8,1)  x(9,1)  x(10,1) x(11,1) x(12,1)
...     ...     ...     ...     ...     ...
x(49,1) x(50,1) x(1,2)  x(2,2)  x(3,2)  x(4,2)
x(5,2)  x(6,2)  x(7,2)  x(8,2)  x(9,2)  x(10,2)
...     ...     ...     ...     ...     ...
```

```
PRINT '(6F8.2)', ((x(i,j), j=1,6), i=1,50)
--- output ---
x(1,1)  x(1,2)  x(1,3)  x(1,4)  x(1,5)  x(1,6)
x(2,1)  x(2,2)  x(2,3)  x(2,4)  x(2,5)  x(2,6)
...     ...     ...     ...     ...     ...
```

- **Assumed-shape arrays** – dummy arguments only

```

REAL FUNCTION assumed_shape(a, b)
  IMPLICIT NONE
  INTEGER, DIMENSION( : , : ), INTENT(IN) :: a
  REAL, DIMENSION( 5: , : , : ), INTENT(IN) :: b
  ...
END FUNCTION assumed_shape

```

- **Automatic arrays** – local variables in procedures

```

SUBROUTINE abc(x, y, n)
  IMPLICIT NONE
  !
  ! Dummy arguments
  INTEGER, INTENT(IN) :: n
  REAL, DIMENSION(n), INTENT(INOUT) :: x ! Explicit-shape
  REAL, DIMENSION(:), INTENT(INOUT) :: y ! Assumed-shape
  !
  ! Local variables
  REAL, DIMENSION(SIZE(y,1)) :: e ! Automatic
  REAL, DIMENSION(10) :: g ! Explicit-shape
  ...
END SUBROUTINE abc

```

- **Allocatable arrays** – dynamically defined extents

```

INTEGER :: stat_var, dim1, dim2
REAL, DIMENSION(:, :), ALLOCATABLE :: matriz
...
ALLOCATE(matriz(dim1, dim2), STAT=stat_var)
IF (stat_var /= 0) STOP "* Not Enough Memory *"
...
DEALLOCATE(matriz)

```

- **conformable** arrays may appear as operands in an expression or an assignment which will be carried element-by-element
- array valued **FUNCTIONS** are possible
- important intrinsic functions for array manipulation

| <i>Name</i> | <i>Result</i>                                                                                                 |
|-------------|---------------------------------------------------------------------------------------------------------------|
| MATMUL      | Matrix product of two matrices, or a matrix and a vector                                                      |
| DOT_PRODUCT | Scalar (dot) product of two vectors                                                                           |
| TRANSPOSE   | Transpose of a matrix                                                                                         |
| MAXVAL      | Maximum value of all the elements of an array, or of all the elements along a specified dimension of an array |
| MINVAL      | Minimum value of all the elements of an array, or of all the elements along a specified dimension of an array |
| PRODUCT     | Product of all the elements of an array, or of all the elements along a specified dimension of an array       |
| SUM         | Sum of all the elements of an array, or of all the elements along a specified dimension of an array           |

- the **WHERE statement**

```
REAL, DIMENSION(100) :: array
...
WHERE (array < 0.0) array = -array
```

- the **WHERE construct**

```
REAL, DIMENSION(100) :: arr
...
WHERE (array /= 0.0)
  array = 1.0/array
ELSEWHERE
  array = 1.0
END WHERE
```

- extracting an **array section**, subscript triplets

```
REAL, DIMENSION(3,4) :: arr
```

```
...
```

```
arr(:, :2, 1)    is a rank-one real array whose elements are
                arr(1, 1), arr(3, 1)
```

```
arr(2, :3)      is a rank-one real array whose elements are
                arr(2, 1), arr(2, 2), arr(2, 3)
```

```
arr(1:2, 3:4)   is a rank-two real array whose elements are
                arr(1, 3), arr(2, 3), arr(1, 4), arr(2, 4)
```

- A **vector subscript** is an integer array expression of rank 1

```
LOGICAL, DIMENSION(3) :: p
```

```
INTEGER, DIMENSION(3) :: u = (/ 3, 2, 2, 3, 1 /)
```

```
...
```

```
p(u) ! gives the array p(3),p(2),p(2),p(3),p(1)
```

- Subscripts, subscript triplets and vector subscripts can be used together

```
CHARACTER(LEN = 10), DIMENSION(3, 4, 9) :: string
```

```
INTEGER, DIMENSION(5) :: vec = (/ 7, 1, 3, 1, 4 /)
```

```
...
```

```
string(vec, 3, 5:9:4) ! a rank-2 5x2 array
```

```
--- output ---
```

```
string(7, 3, 5)    string(7, 3, 9)
```

```
string(1, 3, 5)    string(1, 3, 9)
```

```
string(3, 3, 5)    string(3, 3, 9)
```

```
string(1, 3, 5)    string(1, 3, 9)
```

```
string(4, 3, 5)    string(4, 3, 9)
```

## ELIMINAÇÃO DE GAUSS COM PESQUISA PARCIAL DE PIVOT

```

Function GaussPPP(matt,vecc)   !!! por Svilen S. Valtchev 01/11/04
  IMPLICIT NONE
  Real*8, Dimension(:,:), Intent(In) :: matt
  Real*8, Dimension(:), Intent(In) :: vecc
  Integer :: dim, pivotpos, i, j, k, m
  Real*8, Dimension(Size(vecc)) :: GaussPPP, vector
  Real*8, Dimension(Size(vecc),Size(vecc)) :: matriz
  Real*8 :: aux, pivot, ssv, coef
  !
  matriz=matt
  vector=vecc
  dim=Size(vecc)
  !
  Do k=1,dim   !!!! --> DO principal
  !
  ! Determina a linha do PIVOT
  !
  pivot=0.0d0
  Do m=k,dim
    IF ( DABS(matriz(m,k)) > pivot ) Then
      pivot=DABS(matriz(m,k))
      pivotpos=m
    End IF
  End Do
  !
  ! Troca linhas da matriz e do vector
  !
  IF (pivotpos /= k) Then
    Do m=k,dim
      ssv=matriz(k,m)
      matriz(k,m)=matriz(pivotpos,m)
      matriz(pivotpos,m)=ssv
    End Do
    ssv=vector(k)
    vector(k)=vector(pivotpos)
    vector(pivotpos)=ssv
  End IF

```

```
!  
! Eliminação de Gauss  
!  
Do i=k+1,dim  
  coef=matriz(i,k)/matriz(k,k)  
  matriz(i,k)=0.0d0  
  Do j=k+1,dim  
    matriz(i,j)=matriz(i,j)-coef*matriz(k,j)  
  End Do  
  vector(i)=vector(i)-coef*vector(k)  
End Do  
!  
End Do      !!!! --> Fim do D0 principal  
!  
! Resolução do sistema triangular superior  
!  
Do i=0,dim-1  
  aux=0.0d0  
  Do k=0,i-1  
    aux=aux+matriz(dim-i,dim-k)*GaussPPP(dim-k)  
  End Do  
  GaussPPP(dim-i)=(vector(dim-i)-aux)/matriz(dim-i,dim-i)  
End Do  
!  
End Function GaussPPP
```

---

Svilen S. Valtchev  
Departamento de Matemática  
Instituto Superior Técnico  
[www.math.ist.utl.pt/~ssv](http://www.math.ist.utl.pt/~ssv)  
[ssv@math.ist.utl.pt](mailto:ssv@math.ist.utl.pt)

---