Introducción a Fortran

Mario Merino Martínez
mario.merino@upm.es

Escuela Técnica Superior de Ingenieros Aeronáuticos
Universidad Politécnica de Madrid

4 de marzo de 2011

Índice

¿Qué es programar?

"Un ordenador es una herramienta enormemente tonta, pero increíblemente rápida"

- Programar es dar instrucciones de forma **secuencial** a un ordenador para que realice por nosotros una tarea.
- El ordenador sólo entiende instrucciones en forma de unos y ceros (código máquina). No queremos escribir en binario: nos inventamos **lenguajes de programación**
- Nosotros escribimos nuestro código o algoritmo en un lenguaje, y un programa (**compilador**) se encarga de traducirlo a código máquina (".exe").
- Cada lenguaje tiene su **sintaxis** y **vocabulario** propio. El compilador **no admite fallos ni ambigüedades**.

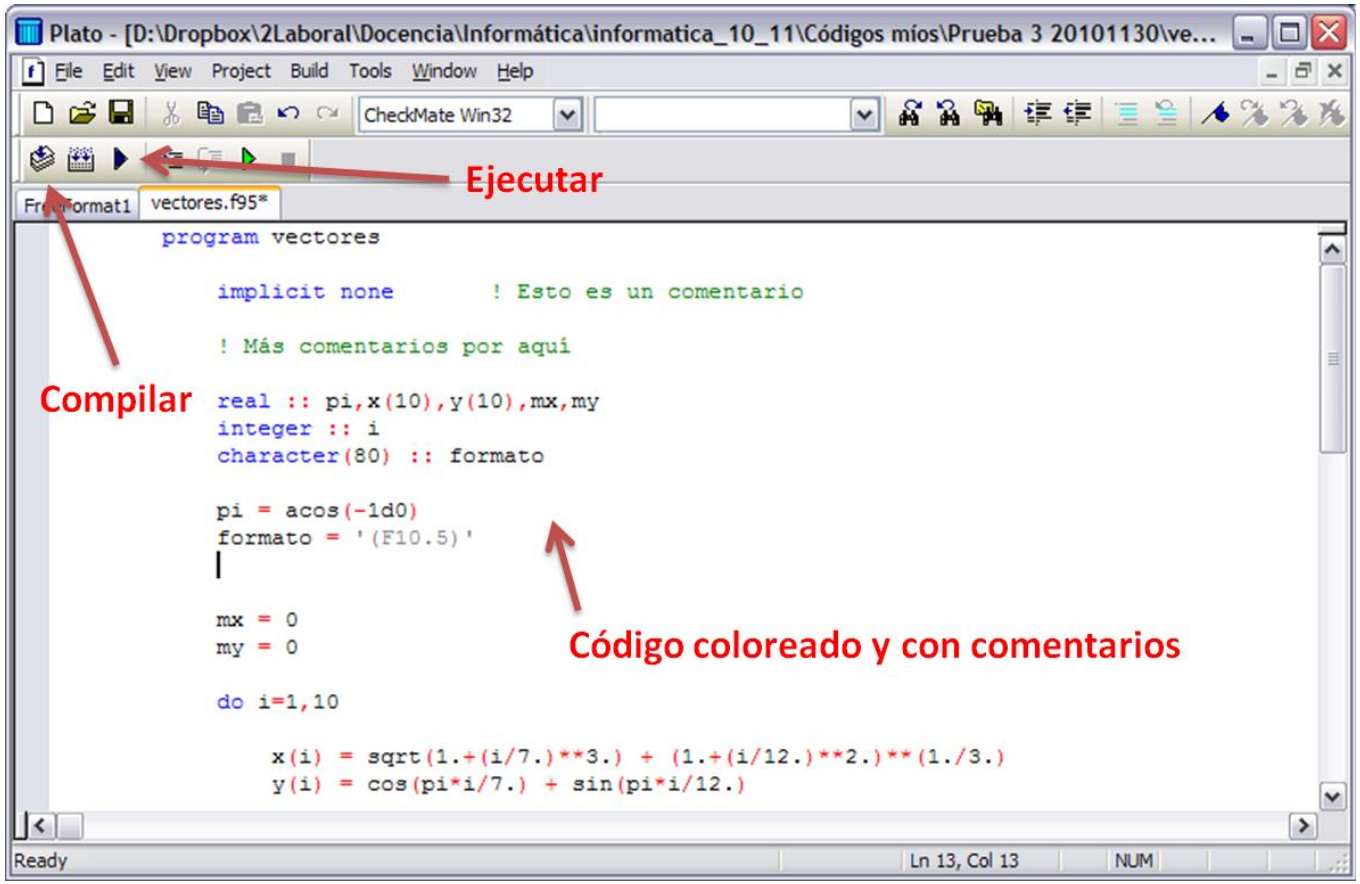
Estructura de un programa en Fortran

- **Fortran** (de "formula translation") es el lenguaje científico e ingenieril por excelencia. Se inventó por y para el cálculo numérico.
- Un programa en Fortran es una **lista secuencial y finita de instrucciones**. El programa empieza y termina siempre así:
- Fortran no distingue entre mayúsculas y minúsculas, e ignora espacios y líneas en blanco extra.

```
program nombre_del_programa
  ! Declaración de variables
  ! Instrucciones SECUENCIALES
  ! Esto es un comentario
end program nombre_del_programa
```

El entorno de desarrollo

Para escribir el código de un programa, sirve cualquier editor de texto. ¡Pero mejor utilizar una herramienta especializada!



Index

Tipos y declaración I

Una **variable** es un nombre al que le asociamos un número o dato, para poder referirnos a él en el código de forma genérica. Hay que **declarar cada variable en la parte inicial del programa**, para que el compilador sepa que existe y de qué tipo es. Existen varios tipos de variables:

- Variables enteras: `integer`
- Variables reales, precisión simple: `real*4`
- Variables reales, precisión doble: `real*8`
- Otros tipos (`character`, `logical`...)

Tipos y declaración II

Se declaran así:

```
program nombre_del_programa
  implicit none ! ponedlo SIEMPRE
  integer :: i, n
  real*4 :: s, temp
  real*8 :: h, x, y
  ! ...
end program nombre_del_programa
```

Asignación de valores y uso I

Una vez las variables han sido declaradas, se les puede dar un valor con el **operador asignación** "=":

```
! ...
integer :: n
real*4 :: x, h
real*8 :: y
! ...
n = 34
h = 4.91234
x = 5.   ! Los reales SIEMPRE con punto o exponente (1e0)
y = 1d0  ! El exponente con "d" se utiliza para doble precisión
! ...
```

Asignación de valores y uso II

Una variable **se puede sobreescribir cuantas veces se desee**, y utilizar en operaciones y funciones:

```
! ...
x = x + x + 124.5d0 - cos(x)
! ...
```

Este ejemplo calcula $x + x + 124.5 - \cos(x)$, y después lo guarda en la variable x, olvidando su valor anterior.

- Fortran **NO tiene constantes** físicas ni matemáticas "de serie". Hay que escribirlas cada vez. Un buen truco para tener el número π es usar: `pi = acos(-1d0)`

Index

Operadores algebraicos (+,-,*,/,**)

Suma, resta, producto, cociente y potencia se escriben así: `+`, `-`, `*`, `/`, `**`. La potencia tiene **prioridad** sobre producto y cociente, y estos sobre suma y resta. Se pueden usar **paréntesis** para establecer el orden de las operaciones:

`5.+2.*4.**(-3)` es distinto de `((5.+2.)*4.)**(-3)`

¡OJO! EL COCIENTE DE ENTEROS PRODUCE COMO RESULTADO UN ENTERO. SI SE QUIERE UN RESULTADO REAL, NO OLVIDAR EL PUNTO NUNCA (o usar DBLE).

```
! ...
x = 2/3 ! El resultado es 0
x = 2./3. ! El resultado es 0.66666...
x = dble(2)/dble(3) ! El resultado es 0.66666...
! ...
```

Leer y escribir por pantalla

Se puede **escribir** cualquier cosa por pantalla usando:

```
write(*,*) 'Hola. El valor de x es: ', x
```

Se puede pedir al usuario que introduzca por **teclado** un valor n con:

```
read(*,*) n
```

Funciones matemáticas "de serie"

Tabla 1: Algunas de las funciones matemáticas de Fortran 90.

SIN(x)	$\longrightarrow$	$sen(x)$	ASIN(x)	$\longrightarrow$	$arcsen(x)$
COS(x)	$\longrightarrow$	$cos(x)$	ACOS(x)	$\longrightarrow$	$arccos(x)$
TAN(x)	$\longrightarrow$	$tg(x)$	ATAN(x)	$\longrightarrow$	$arctg(x)$
SQRT(x)	$\longrightarrow$	$\sqrt{x}$	ABS(x)	$\longrightarrow$	$\|x\|$
EXP(x)	$\longrightarrow$	e^x	LOG(x)	$\longrightarrow$	$ln(x)$
1e2, 1e-3	$\longrightarrow$	10^2, 10^{-3}	LOG10(x)	$\longrightarrow$	$log_{10}(x)$
INT(x)	$\longrightarrow$	Parte entera de x	CEILING(x)	$\longrightarrow$	Menor entero mayor que x
MAX(x,y)	$\longrightarrow$	Máximo de x e y	MIN(x,y)	$\longrightarrow$	Mínimo de x e y

Algunas funciones sólo funcionan con reales `real*4` y `real*8`.

Ejemplo 1: raíz

Practicamos declaración y uso de variables, read y write, y funciones intrínsecas.

```
program ejemplo1
  implicit none
  integer :: n
  real*8 :: raiz

  write(*,*) 'Por favor, introduzca un numero n'
  read(*,*) n
  raiz = sqrt(dble(n))
  write(*,*) 'El numero introducido es: ', n
  write(*,*) 'Su raiz cuadrada es: ', raiz
end program ejemplo1
```

Index

Ejecución condicional (if)

A veces interesa que una parte del código **sólo se ejecute si se verifica una condición**. Por ejemplo, si el número n introducido en el ejemplo anterior es negativo, nos gustaría avisar al usuario. Esto puede hacerse con la construcción:

`if (`*condición*`) then ... else ... end if:`

```
! ...
write(*,*) 'Por favor, introduzca un numero n'
read(*,*) n
if (n<0) then
  write(*,*) 'Atencion: el numero es negativo'
else
  raiz = sqrt(dble(n))
end if
! ...
```

Condiciones

Las condiciones van **entre paréntesis**. Los **operadores de comparación** son

- <, >: menor y mayor que
- <=, >=: menor o igual y mayor o igual que
- ==: igual que
- /=: distinto que

Se pueden construir condiciones más complicadas con los operadores lógicos `.AND.`, `.OR.`, `.NOT.`:

```
(x<-3. .OR. x>=7.)
```

Index

Tareas repetitivas (do) I

Cuando se quiere hacer una operación muchas veces (o un número indeterminado de veces), es muy útil emplear **bucles** do.

```
! ...
do i=1,10
  write(*,*) 'Esta es la iteracion numero ', i
end do
! ...
```

- El código comprendido entre el do y el end do **se ejecuta tantas veces como se especifique** (10)
- Un bucle do posee un **contador** (i). El contador va **desde el valor inicial hasta el final**, de uno en uno.

Tareas repetitivas (do) II

- La variable contador (`i`) **se puede utilizar** en nuestras operaciones si queremos.

```
! ...
s = 0
do i=1,10
  s = s + i**2
end do
! ...
```

- En general, se puede **especificar también el incremento del contador**: `do i=20,0,-2` *(ve desde i=20 hasta i=0 de -2 en -2).*
- Se pueden "**anidar**" varios bucles uno dentro de otro (ver ejercicios).

Ejemplo 2: sumatorios

Se realizan con una variable de **acumulación**, que hay que poner a **cero** inicialmente. En cada iteración se suma sobre ella la expresión deseada. En este ejemplo, $s = \sum_{i=1}^{n} i$:

```
program ejemplo2
  implicit none
  integer :: n, i
  real*8 :: s
  write(*,*) 'Por favor, introduzca un numero n'
  read(*,*) n
  s = 0d0 ! Ponemos a cero la variable de acumulación
  do i=1,n
    s = s + i ! Acumulamos el valor a sumar
  end do
  write(*,*) 'El sumatorio es: ', s
end program ejemplo2
```

Ejemplo 3: productos

También con variable de **acumulación**, que hay que poner a **uno** inicialmente. En cada iteración se multiplica con la expresión deseada. En este ejemplo, $s = \prod_{i=1}^{n} i$:

```
program ejemplo3
  implicit none
  integer :: n, i
  real*8 :: p
  write(*,*) 'Por favor, introduzca un numero n'
  read(*,*) n
  p = 1d0 ! Ponemos a uno la variable de acumulación
  do i=1,n
    p = p * i ! Acumulamos el valor del producto
  end do
  write(*,*) 'El producto es: ', p
end program ejemplo3
```

Index

Subprogramas

A veces queremos **reutilizar** un trozo de código en múltiples ocasiones para realizar una misma tarea en distintos lugares de nuestro programa.

- Las **funciones** y **subrutinas** son trozos de código (programas independientes) que pueden ser ejecutados desde el programa principal en cualquier momento.
- Tienen la **misma estructura que un programa**.
- Son "**cajas negras**:" mi programa las llama con unos argumentos de entrada, y recibe unos argumentos de salida, sin preocuparse de lo que sucede en ellas.
- Se escriben **fuera** del programa principal.

Funciones I

- Se escriben entre las instrucciones `function` y `end function`.
- Hay que **declararlas en el programa principal** como una variable con el atributo `external`.
- Devuelven el valor asignado **al nombre de la función**.
- Hay que **declarar el nombre de la función** dentro de la función (no deja de ser una variable más).
- Una vez creadas, se usan igual que las funciones intrínsecas: e.g.: `x = 1. + f(x) * 2.6`.

Funciones II

```
program main
implicit none
real*8, external :: f
  ! ...
  write(*,*) f(3.52)
  ! ...
end program main

function f(x)
  implicit none
  real*8 :: f, x
  f = x**2 +log(x)
end function f
```

Subrutinas I

- Se escriben entre las instrucciones `subroutine` y `end subroutine`.
- **No** hay que declararlas en el programa principal si están en el mismo archivo de texto.
- Trabajan sobre los argumentos que se les entrega, que pueden ser de **entrada**, de **salida**, o de **entrada/salida**.
- **No** hay que declarar su nombre dentro de la subrutina (el nombre no es variable)
- Se llaman desde el programa principal con la instrucción `call`: e.g.: `call mi_sub(x,y,z)`

Subrutinas II

```
program main
implicit none
  ! ...
  call mi_sub(x,y)
  write(*,*) y
  ! ...
end program main

subroutine mi_sub(a,b)
  implicit none
  real*8 :: a, b
  ! ...
end subroutine mi_sub
```

Ejercicio guiado 1

Se desea elaborar un programa que pida al usuario un número entero n, y que a partir de él calcule

$$r = \sqrt[3]{n} + n,$$

el sumatorio

$$S = \sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \ldots + n^2,$$

y el factorial

$$P = n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n.$$

Ejercicio guiado 2

Encapsular el código anterior dentro de una subrutina, que sea llamada desde el programa principal para los siguientes valores n:

$$n = 1, 7, 15, -4$$

¿Qué sucede en el último caso?

Ejercicios

- Número 17 (cálculos sencillos)
- Número 31 (funciones intrínsecas). Intentad encapsular el código en funciones después.
- Número 46, apartados *d)*, *f)*, *i)* (bucles)
- Número 90 (funciones)

Index

¡Desde luego que no es el fin!

Fortran es **ampliamente extenso**. Os animo a que primero dominéis estos rudimentos, y que según os surja la necesidad, ampliéis vuestro conocimiento de Fortran. En concreto:

- Vectores y matrices (arrays).
- Escribir con formato; escribir y leer archivos.
- Sentencias "exit," "select case," "do while."
- Argumentos opcionales de las sentencias "read," "write."
- Opciones de Plato y cómo usar el debugger.
- Variables con atributo "allocatable," "pointer."
- Modules, interfaces, estructuras.

Cómo continuar

Para más información:

- "Introducción a FORTRAN 90", J.J. Sánchez y C. Vázquez.
- Ayuda de Plato "Fortran 95 language overview."
- Libros y manuales de Fortran (biblioteca).
- Búsquedas en internet.

Cualquier duda: mario.merino@upm.es (o subid al departamento).

Imprescindibles

- No olvidar **NUNCA** el punto "." en real*4 y "d" en real*8.
- Usar **SIEMPRE** "implicit none."
- Leer **COMPLETAMENTE** los errores del compilador: nos dicen cómo resolverlos.
- Hacer los bucles "do" con contadores INTEGER.
- No hacer líneas demasiado largas. Fortran sólo lee hasta la columna 132 (continuar línea con &).
- Escribir códigos profusamente comentados (!)