Introducción a FORTRAN

Miguel Alcubierre
Instituto de Ciencias Nucleares, UNAM

Abril 2005

Índice

1. Introducción

En este curso se presenta una breve introducción a FORTRAN 90, el lenguaje de programación de más amplio uso en el cómputo científico. El nombre FORTRAN proviene de "FORmula TRANslator" (traductor de fórmulas), y fue desarrollado originalmente por IBM en 1954, con el objetivo de poder escribir programas de cómputo científico en un lenguaje de alto nivel en vez de tener que recurrir a lenguaje de máquina o ensamblador. En 1958 se presentó una segunda versión y varias compañías comenzaron a desarrollar compiladores independientes a IBM para usar el lenguaje en otras máquinas.

El primer estandard de FORTRAN se introdujo en 1962 y se llamó FORTRAN IV. En 1966 se presentó el primer estandard ANSI (American National Standards Institute), que se conoció como FORTRAN 66. El segundo estandard ANSI, con muchas mejoras, se introdujo en 1977 (FORTRAN 77), y se convirtió en el estandard utilizado por la comunidad científica por muchos años. Incluso a la fecha es común encontrar muchos programas escritos en FORTRAN 77.

FORTRAN 77 tenía una serie de desventajas. Entre ellas una estructura muy rígida adaptada al uso de tarjetas perforadas ("forma fija"), que requería que ciertas columnas tuvieran usos específicos. Además, no permitía un uso dinámico de la memoria y no permitía realizar operaciones entre arreglos de números. Para mejorar esta situación, en 1990 se presentó un tercer estandard ANSI conocido como FORTRAN 90, que contenía muchas nuevas características y permitía una programación más estructurada. Una serie de cambios menores se presentaron en 1995 (FORTRAN 95), y actualmente se trabaja en un nuevo estandard ANSI (FORTRAN 2003).

El día de hoy la mayor parte de los programas en FORTRAN siguen el estandard de FORTRAN 90, pero aún existe un número importante de aplicaciones de FORTRAN 77. FORTRAN esta específicamente diseñado para el cómputo científico, y no es particularmente bueno para otro tipo de aplicaciones (control, administración, manejo de documentos, etc). Para estas aplicaciones otros lenguajes somo C, JAVA o PERL son más adecuados. En la actualidad, la mayor parte del cómputo científico de alto rendimiento a nivel internacional se lleva a cabo en FORTRAN (FORTRAN esta muy lejos de ser obsoleto), aunque los lenguajes C y C++ han ganando cierta popularidad recientemente.

Parafraseando a Mark Twain:

"Las noticias de la muerte de FORTRAN han sido enormemente exageradas".

2. Compilar y correr programas en FORTRAN

Los programas en FORTRAN se escriben en un editor de texto cualquiera (vi, emacs, etcétera). Normalmente, el nombre del archivo debe llevar como sufijo `.f`, o `.f90`. A este archivo se le llama el "código fuente".

Una vez escrito el programa, este debe compilarse, es decir, debe ser leído por un programa llamado "compilador" que lo traduce a lenguaje de máquina y produce un nuevo archivo con el programa ejecutable. Algunos lenguajes de programación no usan un compilador, sino un "intérprete" (por ejemplo BASIC y PERL). La diferencia es que un interprete traduce y ejecuta linea por linea, sin nunca crear un archivo ejecutable. Un interprete es más transparente de usar, pero mucho más lento. FORTRAN funciona siempre con un compilador.

Hay muchos compiladores diferentes, con diversas opciones al compilar (por ejemplo, algunos prefieren el sufijo `.f` y otros el sufijo `.f90`). Existe un compilador gratuito de FORTRAN 90 para Linux de producido por INTEL que puede bajarse de la red.

La manera estandard de compilar un programa en FORTRAN es abrir una terminal, ir al directorio que contiene el programa, y escribir:

```
f90 nombre1.f90
```

donde `nombre1.f90` es el nombre del programa. Es importante notar que el comando `f90` puede variar de nombre, por ejemplo el compilador de INTEL usa el comando `ifc` o `ifortran`. Pero siempre es posible hacer un alias.

Al compilar el programa se obtiene un ejecutable que tiene por default el nombre `a.out`. Esto puede cambiarse haciendo:

```
f90 nombre1.f90 -o nombre2
```

La opción `-o` le dice al compilador que el ejecutable debe llamarse `nombre2`. Para correr el programa se escribe simplemente

```
./nombre2
```

donde se usa `./` para indicar a Unix que debe buscar el ejecutable en el directorio local.

Para programas que están contenidos en varios archivos, hay una etapa más. El compilador primero genera "archivos objeto" con sufijo `.o` que contienen la traducción a lenguaje de máquina de cada archivo individual. Después se hace una liga (link) entre los diversos archivos objeto para construir el ejecutable.

3. Elementos básicos de un programa en FORTRAN

Un programa en FORTRAN tiene los siguientes elementos básicos:

- Nombre del programa. El nombre del programa es en realidad opcional, pero es muy buena idea tenerlo.
- Declaraciones de variables utilizadas en el programa.
- Cuerpo del programa. Comandos a ejecutar en el código. Los comandos se ejecutan en orden de aparición. El programa siempre debe terminar con el comando `END`.
- Subprogramas. El cuerpo del programa puede llamar a subprogramas que realicen tareas específicas. Es buena práctica de programación separar un programa en bloques y poner cada bloque en diferentes subprogramas. De hecho, para programas largos es buena idea tener cada subprograma en archivos separados.

La estructura de un comando en FORTRAN 90 tiene las siguientes propiedades:

- Los comandos se escriben en lineas de a lo más 132 caracteres (aunque algunos compiladores aceptan lineas más largas).
- Espacios en blanco al principio de una linea se ignoran. Esto ayuda a mejorar visualmente la estructura del programa. Hay que recordar que no solo la máquina va a leer el programa, sino también seres humanos (por lo menos el autor), por lo que una estructura visualmente clara en importante.
- Un signo `&` al final de una linea indica que el comando continua en la linea siguiente.
- Todo lo que siga de un signo `!` se considera un comentario y es ignorado por el compilador. Los comentarios son para las personas, no para la máquina. Hacen más fácil de entender el programa para personas que no lo escribieron, y sirven incluso para que el autor sepa lo que hace el programa si lo vuelve a ver tiempo después.
- Es posible poner varios comandos en una linea separándolos con punto y coma.
- Importante: FORTRAN no distingue entre mayúsculas y minúsculas en un programa, también ignora más de un espacio en blanco y lineas en blanco. Que se use es cuestión de estilo personal.

Ejemplo: Programa “nada”.

```
program nada

! Este programa no hace nada.

end program nada
```

La primera línea dice como se llama el programa. La segunda linea es un comentario y el compilador la ignora. La última linea dice que el programa ha terminado.

Ejemplo: Programa “hola”.

```
program hola

! Ahora vamos a saludar al mundo

print *, 'hola mundo!'  ! Aqui es donde saludamos

end program hola
```

Lo nuevo aquí es la linea: `print *, 'hola mundo!'`. La instrucción `print *` indica imprimir en la salida estandard (es decir, la pantalla). Lo que se va a imprimir debe colocarse entre comillas después de una coma. Nótese como hay un comentario al final de la linea que el compilador simplemente ignora.

4. Constantes y variables

Los programas en FORTRAN manejan datos de dos tipos: constantes y variables. Las constantes tienen un valor fijo, mientras que las variables se identifican con nombres y pueden cambiar de valor durante al ejecución del programa. Constantes y variables deben tener tipos específicos que indican al programa como almacenarlas y manejarlas.

4.1. Tipos de datos

Los tipos de datos permitidos son los siguientes:

`logical` Las variables lógicas solo pueden tener dos valores: `.true.` (verdadero) y `.false.` (falso).

`integer` Valores enteros guardados en 4 bytes. Se indican como números sin punto decimal: 1, 2, -3, 25, etc.

`real` Valores reales guardados en 4 bytes y con 8 cifras significativas. Se indican con punto decimal, y de ser necesario el exponente de la potencia de 10 después de una `E`: `1., -3.1416, 6.25E-10`, etc.

`double` Valores reales de doble precisión guardados en 8 bytes y con 16 cifras significativas, también se denotan por `real(8)`. Se indican con punto decimal y el exponente de la potencia de 10 después de una `D`: `1.D0, -3.1416D0, 6.25D-10`, etc. Son muy útiles en cálculos numéricos largos, donde los errores de redondeo pueden hacer que las últimas 4 o 5 cifras significativas de un número real sean basura.

`quadruple` Valores reales de cuádruple precisión guardados en 16 bytes y con 32 cifras significativas, también se denotan por `real(16)`. Se indican con punto decimal y el exponente de la potencia de 10 después de una `Q`: `1.Q0, -3.1416Q0, 6.25Q-10`, etc.

`complex` Dos valores reales formando un par y que en operaciones matemáticas se tratan como la parte real e imaginaria de un número complejo: `(1.,-2.), (1.0E0,-2.0E0)`. También existen versiones de doble y cuádruple precisión.

`character` Variables que corresponden a cadenas de caracteres. Al declarar una variable de este tipo se debe especificar cuantos caracteres puede tener. Estas variables deben estar contenidas en comillas: `'hola', 'abcdfe', 'Me llamo Luis', '128.3'`, etc.

4.2. Declaración de variables

Las variables utilizadas en un programa FORTRAN deben declararse como uno de los tipos mencionados en la sección anterior. Por compatibilidad con versiones viejas de FORTRAN, se asume que aquellas variables que no han sido declaradas tienen un tipo implícito de acuerdo a la siguiente regla: variables cuyos nombres empiezan con {i,j,k,l,m,n} se asumen enteras, y todas las demás se asumen reales. El uso de declaraciones implícitas

es indeseable, pues hace más difícil detectar posibles errores de tecleo. Para evitar usar declaraciones implícitas se debe poner al principio de las declaraciones la linea:

```
implicit none
```

Ejemplo: Programa "valores".

```
program valores

implicit none

logical flag
integer i
real a
character(30) texto

i = 1
a = 2.5
texto = 'Estas son las variables:'

print *, texto
print *, flag,i,a

end program valores
```

El programa anterior primero declara que no hay variables implícitas, luego declara una variable lógica, una entera, una real y una cadena de caracteres. Luego les asigna valores y las imprime a la pantalla. Nótese el uso de comillas para la variable de tipo `character`, y como se ha declarado como de 30 caracteres (suficientes para contener la frase 'Estas son las variables').

También es importante señalar el uso del operador `=`. Este operador no significa "igual", sino que significa "asignar", y sirve para asignar valores a variables.

4.3. Conversión entre tipos

En FORTRAN es muy importante declarar las variables correctamente. Muchos errores comunes están relacionados con utilizar tipos equivocados de variables. Por ejemplo,

en operaciones entre números reales debe evitarse el utilizar variables enteras ya que fácilmente se obtendrán valores equivocados. Supongamos que se han declarado las variables `i` y `j` como enteras, y la variable `x` como real, y queremos calcular:

```
x = i / j
```

Si tenemos `i=10` y `j=4` el resultado será 2 y no 2.5 como uno podría esperar. Esto se debe a que la división entre números enteros siempre se considera un entero. Para evitar este tipo de errores es importante convertir un tipo de variables a otras. Esto puede lograrse con los comandos: `int(),nint(,real(),dble()`. Los comandos `real()` y `dble()` convierten la cantidad entre paréntesis a una variable real o de doble precisión, respectivamente. Los comandos `int()` y `nint()` convierten la cantidad entre paréntesis a un número entero, la diferencia está en que `int()` simplemente trunca el número, mientras que `nint()` lo reduce al entero más cercano.

Ejemplo: Programa "convertir".

```
program convertir

print *, 10/4
print *, 10.0/4.0
print *, real(10)/real(4)
print *, real(10/4)
print *, dble(10)/dble(4)

print *, int(10.7)
print *, nint(10.7)
print *, int(-3.7)
print *, nint(-3.7)

end program convertir
```

5. Operaciones en FORTRAN

Las operaciones aritméticas en FORTRAN involucran los siguientes operadores:

`=` Asignación. Es muy importante recalcar que este operador NO significa igualdad. En FORTRAN, tiene sentido escribir lineas que algebraica-

mente son absurdas como por ejemplo:

```
a = a + 1
```

Esto significa tomar la variable `a`, sumarle 1, y asignar el resultado de nuevo a la variable `a`.

`+` Suma (`2.0+3.0` → 5.0).

`-` Resta (`2.0-3.0` → -1.0).

`*` Multiplicación (`2.0*3.0` → 6.0).

`/` División (`2.0/3.0` → 0.66666667).

`**` Exponenciación (`2.0**3.0` → 8.0).

Las operaciones se realizan en el siguiente orden:

1. Términos entre paréntesis, comenzando por los de más adentro.
2. Exponenciación.
3. Multiplicación y división de izquierda a derecha.
4. Sumas y restas de izquierda a derecha.

IMPORTANTE: Igual que cuando uno hace operaciones a mano, es mucho más difícil dividir que multiplicar, por lo que resulta mucho más lento. A FORTRAN le toma mucho más tiempo calcular `5.0/2.0` que calcular `5.0*0.5`. Siempre que sea posible se deben evitar las divisiones para tener un código más eficiente.

6. Arreglos

FORTRAN puede almacenar en memoria vectores y matrices utilizando variables llamadas "arreglos" que contienen muchos elementos. Los arreglos pueden ser de cualquiera de los tipos aceptados por FORTRAN.

6.1. Arreglos de tamaño fijo

La manera más directa de declarar un arreglo es dando su tamaño desde el principio. Por ejemplo, para declarar un vector de 3 componentes y una matriz de 4 por 5 se escribe:

```
real, dimension(3) :: v
real, dimension(4,5) :: m
```

Es posible declarar arreglos con un número arbitrario de dimensiones. Por ejemplo, un arreglo de 4 índices cuyos valores van de 1 a 2 se declara como:

```
real, dimension(2,2,2,2) :: R
```

También se puede dar explícitamente el rango permitido para los índices:

```
real, dimension(0:8), v1
real, dimension(2:5), v2
```

Esto declara al vector `v1` con índices que van de 0 a 8, y al vector `v2` con índices que van de 2 a 5.

Existe una abreviación de la declaración de arreglos. Por ejemplo, los arreglos `v,m,R` de los ejemplos anteriores se pueden también declarar como:

```
real :: v(3), m(4,5), R(2,2,2,2)
```

Para referirse a un miembro específico del arreglo, se utilizan expresiones del tipo:

```
a = v(1) + m(1,1) + R(1,2,1,2)
```

Es posible también realizar operaciones con arreglos completos a la vez (esto no era posible en FORTRAN 77). Por ejemplo, si `a,b,c` son arreglos con las mismas dimensiones, podemos escribir:

```
a = b+c
a = b*c
a = b**2
```

La primera operación suma uno a uno los elementos de los arreglos `b` y `c` y los coloca

en los elementos correspondientes de `a`. La segunda operación hace lo mismo, pero multiplicando los elementos de `b` y `c` (Ojo: esto NO es una multiplicación de matrices, sino una multiplicación elemento a elemento). La última operación eleva cada elemento de `b` al cuadrado y los asigna al elemento correspondiente de `a`.

Para este tipo de operaciones es muy importante que los arreglos involucrados tengan las mismas dimensiones, o habrá un error.

Ejemplo: Programa "arreglos".

```
program arreglos

implicit none

! Declarar vectores.

real, dimension(3) :: v1,v2

!Declarar matrices.

real, dimension (0:1,0:1) :: a,b,c

! Dar componentes de v1.

v1(1) = 1.0
v1(2) = 2.0
v1(3) = 3.0

! Dar v2 de golpe.

v2 = 4.0

! Dar componentes de b.

b(0,0) = 1.0
b(0,1) = 2.0
b(1,0) = 3.0
b(1,1) = 4.0
```

```
! Dar la matriz c de golpe.

c = 2.0

! Definir a como la division de b entre c.

a = b/c

! Imprimir v1 completo.

print *, 'Vector v1:'
print *, v1
print *

! Imprimir v2 componente a componente.

print *, 'Vector v2:'
print *, v2(1),v2(2),v2(3)
print *

! Imprimir componente c(0,0).

print *, 'Componente c(0,0):'
print *, c(0,0)
print *

! Imprimir componente b(1,1).

print *, 'Componente b(1,1):'
print *, b(1,1)
print *

! Imprimir a completo.

print *, 'Matriz a completa:'
print *, a
print *
```

```
end program arreglos
```

En el ejemplo anterior se muestran declaraciones y operaciones con arreglos. Nótese que al imprimir es posible imprimir componentes individuales, o arreglos completos. En este último caso el arreglo se escribe recorriendo primero los índices de la izquierda y luego los de la derecha (al revés de lo que uno esperaría normalmente). Otro punto a notar es el comando `print *` sin nada que le siga, esto hace que FORTRAN imprima una linea en blanco.

6.2. Asignación dinámica de memoria

En algunas ocasiones resulta útil declarar un arreglo sin asignarlo un número fijo de componentes desde el principio, y solo asignarlas después (debido por ejemplo a que se leen datos del exterior, ver sección 9). Para hacer esto, los arreglos se pueden declarar como:

```
real, allocatable, dimension (:) :: v
real, allocatable, dimension (:,:) :: m
```

Antes de realizar cualquier operación con un arreglo de este tipo se le debe asignar memoria de la siguiente forma:

```
allocate(v(3),m(4,5))
```

Al terminar los cálculos es posible eliminar esta memoria con el comando:

```
deallocate(v,m)
```

Esto es particularmente útil en códigos largos que utilizan mucha memoria, o en casos donde el tamaño de los arreglos depende ya sea de valores externos o de que se cumplan ciertas condiciones en el código.

7. Funciones intrínsecas

FORTRAN cuenta con una serie de funciones matemáticas pre-definidas llamadas "funciones intrínsecas". Una lista de algunas de ellas (no todas) es:

`sqrt(x)` Raíz cuadrada de `x`.

`abs(x)`	Valor absoluto de `x`.
`sin(x)`	Seno de `x`.
`cos(x)`	Coseno de `x`.
`tan(x)`	Tangente de `x`.
`asin(x)`	Arco-seno de `x`.
`acos(x)`	Arco-coseno de `x`.
`atan(x)`	Arco-tangente de `x`.
`exp(x)`	Exponencial de `x` (e^x).
`alog(x)`	Logaritmo natural de `x`.
`alog10(x)`	Logaritmo en base 10 de `x`.
`max(x,y)`	Máximo entre `x` y `y`.
`min(x,y)`	Mínimo entre `x` y `y`

IMPORTANTE: Las funciones trigonométricas se usan siempre en radianes, y no en grados.

Ejemplo: Programa "intrínsecas".

```
program intrinsecas

implicit none

real pi

! Imprimir seno y coseno de cero.

print *
print *, 'Seno y coseno de 0'
print *, sin(0.0),cos(0.0)
print *
```

```
! Ahora a calcular pi.  Recuerden que cos(pi) = -1.

pi = acos(-1.0)

print *, 'El valor de pi es'
print *, pi
print *

! Ya tenemos pi, ahora calcular su seno, coseno y tangente

print *, 'Seno, coseno y tangente de pi'
print *, sin(pi),cos(pi),tan(pi)
print *

! Cual es la raiz cuadrada de 2?

print *, 'Raiz de 2'
print *, sqrt(2.0)
print *

end program intrinsecas
```

Cosas por notar en este ejemplo son las siguientes: FORTRAN no conoce el valor de π. En general no es buena idea definir este valor directamente, es mejor calcularlo usando funciones trigonométricas inversas para obtener mayor precisión. Nótese también que al correr este programa, el seno y la tangente de π no dan cero sino números de orden 10^{-7}. Esto es debido al error de redondeo en la última cifra significativa de los números reales. Si utilizamos doble precisión, el error será del orden 10^{-15}. Cuando uno realiza muchas operaciones numéricas, el error de redondeo se propaga y puede contaminar varias cifras decimales. Por ello es buena idea usar doble precisión para cálculos numéricos complejos.

8. Control de flujo del programa

En general, un programa FORTRAN ejecuta las comandos en el orden en el que se escribieron, uno a la vez. Sin embargo, frecuentemente hay situaciones en las que esto es demasiado simple para lo que uno quiere hacer. A veces es necesario repetir una misma operación muchas veces con ligeras variaciones, y otras veces hay operaciones que solo

deben realizarse si se cumple alguna condición. Para este tipo de situaciones existen los llamados "comandos de control del programa" y que caen en dos tipos básicos: "loops" y condicionales.

8.1. Loops

Un "loop" es un conjunto de instrucciones que deben realizarse muchas veces y tiene la forma estandard:

```
do i=start,end,increment

   comando 1
   comando 2
   comando 3
   ...

end do
```

Todos los comandos que están contenidos entre la linea `do i=...` y la linea `end do` se repiten varias veces, dependiendo de los valores de `start,end,increment` que deben ser enteros. la primera vez, la variable `i` toma el valor `start`. Al terminar, el valor de `i` se incrementa en `increment`, si el valor final es mayor que `end` el loop ya no se ejecuta de nuevo y el programa continua en el comando que siga después de `end do`, de lo contrario se vuelven a ejecutar los comandos dentro del loop hasta que se obtenga `i>end`. Nota: El indentar los comandos dentro del loop no es necesario (FORTRAN ignora los espacios extra), pero es una buena idea pues hace más fácil identificar visualmente donde comienza y termina el loop.

Ejemplo: Programa "fibonacci".

```
program fibonacci

! Este programa calcula los numeros de la serie
! de fibonacci hasta nmax.

implicit none

integer i,nmax
```

```
integer jnew,jold,aux

! Poner un limite.

nmax = 10

! Inicializar (jnew,jold).

jnew = 1; jold = 1

! Iniciar loop.

print *

do i=1,nmax

!   Imprimir elemento i de la serie.

    print *, 'Elemento ',i,' de la serie de Fibonacci: ', jnew
    print *

!   Calcular nuevo elemento de la serie.

    aux  = jnew
    jnew = jnew + jold
    jold = aux

end do

end program fibonacci
```

8.2. IF

En ocasiones uno desea que una parte del programa solo sea ejecutada si cierta condición específica se satisface. Esto se logra utilizando los "condicionales", que en FORTRAN se controlan con el comando `IF`. La estructura de este comando es la siguiente:

```
if (expresión lógica) then
```

```
   comando 1
   comando 2
else
   comando 3
   comando 4
end if
```

Al ejecutar esto, el programa verifica si la expresión lógica entre paréntesis después del `if` es verdadera. De serlo, se ejecutan los comandos siguientes. De ser falsa, se ejecutan los comandos después del `else` (nótese que el [else] es opcional).

La expresión lógica puede utilizar cualquiera de los siguientes operadores lógicos:

`==` Igualdad: `if (i==3) then`. Una versión equivalente de este operador (que de hecho es la única válida en FORTRAN 77) es `.eq.`: `if (i.eq.3) then`.

`>` Mayor que: `if (i>3) then`. Una versión equivalente de este operador (que de hecho es la única válida en FORTRAN 77) es `.gt.`: `if (i.gt.3) then`.

`>=` Mayor o igual: `if (i>=3) then`. Una versión equivalente de este operador (que de hecho es la única válida en FORTRAN 77) es `.ge.`: `if (i.ge.3) then`.

`<` Menor que: `if (i>3) then`. Una versión equivalente de este operador (que de hecho es la única válida en FORTRAN 77) es `.lt.`: `if (i.lt.3) then`.

`<=` Menor o igual: `if (i<=3) then`. Una versión equivalente de este operador (que de hecho es la única válida en FORTRAN 77) es `.le.`: `if (i.le.3) then`.

`/=` No igual: `if (i/=3) then`. Una versión equivalente de este operador (que de hecho es la única válida en FORTRAN 77) es `.ne.`: `if (i.ne.3) then`.

.not. Negación lógica: `if (.not.flag) then`, donde `flag` ha sido declarada como una variable lógica.

.or. "O" lógico: `if ((i==3).or.(i==5)) then`.

.and. "Y" lógico: `if ((i==3).and.(j==5)) then`.

El comando `if` puede simplificarse si solo hay un comando en su interior:

```
if (i==3) comando
```

Este comando indica que si se cumple la condición el comando indicado debe ejecutarse. También es posible tener muchas alternativas en un condicional:

```
if (expresión lógica 1) then
   comando 1
else if (expresión lógica 2)
   comando 2
else if (expresión lógica 3)
   comando 3
else
   comando 4
end if
```

Ejemplo: Programa "condicional".

```
program condicional

implicit none

! Declaramos una variable logica y una entera.

logical flag
integer i

! Dejar un espacio para que se vea menos feo.

print *

! Contar de 1 a 20.

do i=1,20
```

```
!  Estamos abajo de 10?

   if (i<=9) then
      print *, 'No hemos llegado al 10, vamos en ', i

!  Estamos justo en 10?

   else if (i==10) then
      print *, 'Vamos justo en 10'

!  Estamos abajo de 16?

   else if (i<16) then
      print *, 'Ya nos pasamos del 10, vamos en ', i

!  Nada de lo anterior.

   else
      print *, 'Ya pasamos del 15, vamos en ', i
   end if

!  En caso de estar en 13 avisar.

   if (i==13) print *, 'Por cierto, ahora vamos en 13'

!  Definir la variable logica "flag".  Sera verdadera
!  si no estamos en 19.

   flag = (i /= 19)

!  Ahora avisar siempre que "flag" sea falsa.

   if (.not.flag) print *, 'Y ahora estamos en 19'

end do

! Adios
```

```
print *
print *, 'Hey, ya acabamos!'
print *

end program condicional
```

8.3. Control lógico de loops

Es posible utilizar condiciones lógicas para controlar un loop. Esto se logra utilizando el comando `do while()`:

```
do while(expresión lógica)
   comando 1
   comando 2
   ...
end do
```

En este caso no se cuenta con una variable entera como en el caso estandard, sino que el loop se ejecuta una y otra vez mientras la expresión lógica sea verdadera. El loop se detiene una vez que dicha expresión es falsa. Por ejemplo, el siguiente código actúa de la misma forma que un loop estandard de 1 a 10:

```
i = 1
do while (i <=10)
   i = i + 1
end do
```

CUIDADO: Con este tipo de loops se corre el riesgo de caer en un ciclo eterno, donde la condición lógica nunca deja de satisfacerse y la máquina sigue ejecutando el loop para siempre. Cuando se trabaja en una terminal uno nota esto fácilmente si el código continua mucho tiempo más del que esperábamos sin hacer aparentemente nada. En ese caso siempre se puede detener al código tecleando [CTRL C]. Pero cuando uno esta corriendo en una cola en una super-computadora se corre el riesgo de terminarse la cuota que debería haber durado un año de una sola vez, y peor aún, los encargados del sistema pueden decidir retirar los privilegios de trabajar en esa máquina ante semejante desperdicio de recursos de cómputo y falta de cuidado.

9. Entrada y salida de datos (input/output)

En la mayoría de los códigos científicos es necesario dar datos desde fuera y sacar datos hacia afuera. Por default, la entrada de datos es desde el teclado y la salida es a la pantalla. La entrada y salida de datos se manejan con los comandos:

```
read(,)
write(,)
```

Ambos comandos tienen dos argumentos, el primero de los cuales indica la "unidad" de entrada o salida, y el segundo el formato en el que están los datos. La versión mas simple es:

```
read(*,*)
write(*,*)
```

Aquí, el primer asterisco indica entrada o salida estandard (teclado y pantalla respectivamente), y el segundo formato libre. El comando `write(*,*)` puede substituirse por la forma equivalente `print *` seguido de una coma.

Ejemplo: Programa "fulano".

```
program fulano

! Declarar variables.

implicit none

character(20) nombre

! Preguntar como te llamas.

print *
write(*,*) 'Como te llamas?'
print *

! Leer respuesta desde el teclado.
```

```
read(*,*) nombre

! Saludar.

print *
write(*,*) 'Hola ', nombre
print *

end program fulano
```

9.1. Unidades de entrada y salida

Además de utilizar el teclado y la pantalla, los datos también pueden leerse o enviarse a un archivo. Para ello debe primero abrirse el archivo con el`open` que debe tener al menos dos argumentos (puede tener varios más). El primer argumento es la "unidad" a la que se asigna la salida o entrada de datos, que debe ser un numero entero (comúnmente se usa 10). El segundo argumento es el nombre del archivo entre comillas (con su `PATH` completo si no esta en el mismo directorio que el ejecutable). Al terminar de leer o escribir se debe cerrar el archivo con el comando `close` cuyo único argumento es el número de la unidad.

Ejemplo: Programa "archivos". Para este ejemplo es necesario crear primero un archivo de datos llamado "entrada.dat" que contenga solo una linea con: "juan,25,masculino".

```
program archivos

! Declarar variables.

implicit none

integer edad

character(20) nombre,sexo

! Abrir archivo de entrada.

open(10,file='entrada.dat')

! Leer datos.
```

```
read (10,*) nombre,edad,sexo

! Cerrar achivo.

close(10)

! Abrir archivo de salida.

open(11,file='salida.dat')

! Escribir datos.

write(11,*) 'Me llamo ',nombre
write(11,*) 'Tengo ',edad,' anios'
write(11,*) 'Mi sexo es ',sexo

! Cerrar archivo.

close(11)

end program archivos
```

9.2. Formato de entrada y salida

En ocasiones se requiere que los datos de entrada o salida estén en un formato muy específico. El control del formato se lleva a cabo en el segundo argumento del los comandos `open` y `close`. Existen dos posibilidades: Para formatos sencillos, el formato se coloca simplemente el formato entre comillas y paréntesis dentro del segundo argumento, mientras que para formatos más complicados se coloca un número entero que identifica la linea donde está el formato. Por ejemplo, las dos opciones siguientes son equivalentes:

```
read(10,'(i4)') m

read(10,100) m
100 format(i4)
```

En ambos casos se indica al código que debe leer un número entero de a lo más 4 dígitos de longitud "i4". La ventaja del segundo método es que se puede usar el mismo

formato en más de un comando:

```
read(10,100) i
read(10,100) j
read(10,100) k
100 format(i4)
```

Los diferentes opciones de formato son:

Tipo	sintaxis	ejemplo	datos	descripción
`integer`	nIw	2i4	1234, -12	n es el número de enteros por leer y w el número total de caracteres contando el signo.
`real`	nFw.d	2f8.3	1234.678, -234.678	n es el de reales por leer, w es el número total de caracteres incluyendo el punto decimal y el signo, y d es el número de cifras después del punto.
`character`	nAw	2a6	'abcdef','qwerty'	n es el de palabras por leer y w el número de caracteres en cada palabra.
espacios	nx	2x		n es el número total de espacios en blanco por leer .

Ejemplo: Programa "seno".

```
program seno

! Declarar variables.

implicit none

integer i
real x,s,pi

! Calcular pi.

pi = acos(-1.0)
```

```
! Abrir archivo de salida.

open(10,file='seno.dat')

! Loop desde 0 hasta 2*pi en 100 pasos.

do i=0,100

!  Calcular x entre 0 y 2*pi, y su seno.

   x = 2.0*pi*real(i)/100.0
   s = sin(x)

!  Escribir al archivo.

   write(10,'(2f10.4)') x,s

end do

! Cerrar archivo.

close(10)

end program seno
```

10. Subprogramas

En muchas ocasiones existen tareas que se deben realizar muchas veces durante la ejecución de un código, por lo que resulta conveniente que sean subprogramas en si mismos. La existencia de subprogramas también hace que un código sea más modular, y permite que diferentes personas programes diferentes partes de un mismo código.

Los diversos subprogramas pueden ser parte de un mismo archivo, o estar en archivos separados que se unen durante el proceso de compilación. Si están en un mismo archivo, deben aparecer después de la linea que termina el programa principal. En FORTRAN 90 hay tres tipos básicos de subprogramas: funciones, subrutinas y módulos. Consideraremos cada uno de ellos por separado.

10.1. Funciones

Así como existen funciones intrínsecas en FORTRAN (`sin, cos`, etc.), también es posible definir funciones nuevas de una o más variables. El objetivo de una función es obtener un número a partir de los argumentos. Al igual que el programa principal, una función comienza por su nombre `function nombre`. Al final de la función deben aparecer los comandos `return` seguido de `end function nombre`, esto regresa el control al programa principal. La función es una unidad autónoma, por lo que debe declarar todas las variables que utiliza, incluyendo el nombre de la función y los argumentos.

Es muy importante que los argumentos de la función sean del mismo tipo cuando se llama la función y en la función misma, de otra forma habrá errores al ejecutar el código. El nombre de la función debe declarase en el programa principal y cualquier subprograma que la llame.

Ejemplo: Programa "distancia".

```
! Programa principal.

program distancia

! Declarar variables.

implicit none

integer i,j
real x,y,r
real radio

! Abrir archivo de salida.

open(10,file='radio.dat')

! Loop en dos dimensiones.

do i=1,10
   do j=1,10

!     Coordenadas en el plano.
```

```
      x = real(i)
      y = real(j)

!     Calcular distancia al origen llamado a funcion "radio".

      r = radio(x,y)

!     Escribir a archivo.

      write(10,'(3f10.3)') x,y,r

   end do
end do

! Cerrar archivo.

close(10)

end program distancia


! Funcion "radio".

function radio(x,y)

! Declarar variables.

implicit none

real radio,x,y

! Calcular distancia al origen.

radio = sqrt(x**2 + y**2)

! Terminar funcion.
```

```
return
end function radio
```

10.2. Subrutinas

Una subrutina es similar a una función, pero más complicada, de la que no solo se espera un número, sino toda una secuencia de operaciones que pueden requerir regresar muchos números al programa principal (o ninguno).

Las subrutinas se llaman usando el comando `call nombre`. Igual que las funciones, las subrutinas comienzan por su nombre `subroutine nombre` y terminar con los comandos `return` y `end subroutine nombre`. También son unidades autónomas que deben declarar todas las variables que utilizan, incluyendo a sus argumentos.

A diferencia de las funciones, el nombre de una subrutina no tiene un tipo (el nombre de las funciones si tiene un tipo pues corresponde al valor de regreso).

Una característica importante de las subrutinas es que cuando se pasa un arreglo como uno de sus argumentos, no es necesario dar el tamaño. Se puede pasar el tamaño como un argumento, digamos N, y simplemente declarar el arreglo como `real, dimension(N) :: nombre`. Algo similar sucede con los variable de tipo `character` que se pueden declarar con dimensión indeterminada usando * cuando son parte de los argumentos de la subrutina, de manera que heredan el tamaño que tenían el en programa que llamo a la subrutina.

Ejemplo: Programa "rutinas".

```
! Programa rutinas

program rutinas

! Declarar variables.

implicit none

integer i,j,N
real radio
real, dimension(10,10) :: r,x,y

! Abrir archivo de salida.

open(10,file='radio.dat')
```

```
! Loop en dos dimensiones.

N = 10

do i=1,N
   do j=1,N

!     Coordenadas en el plano.

      x(i,j) = real(i)
      y(i,j) = real(j)

!     Calcular distancia al origen llamado a funcion "radio".

      r(i,j) = radio(x(i,j),y(i,j))

   end do
end do

! Llamar subrutina "escribir".

call escribir(N,r,x,y)

! Cerrar archivo.

close(10)

end program rutinas


! Funcion "radio".

function radio(x,y)

! Declarar variables.

implicit none
```

```
real radio,x,y

! Calcular distancia al origen.

radio = sqrt(x**2 + y**2)

! Terminar funcion.

return
end function radio


! Subrutina "escribir".

subroutine escribir(N,r,x,y)

! Declarar variables y arreglos.

implicit none

integer i,j,N
real, dimension(N,N) :: r,x,y

! Abrir archivo.

open(10,file='distancia2.dat')

! Loop de dos dimensiones para escribir a archivo.

do i=1,N
   do j=1,N
      write(10,'(3f10.3)') x(i,j),y(i,j),r(i,j)
   end do
end do

! Cerrar archivo.
```

```
close(10)

! Terminar subrutina.

return
end subroutine escribir
```

10.3. Módulos

El último tipo de subprogramas son los módulos que solo existen a partir de FORTRAN 90. Los módulos sirven para declarar variables que se usan en muchos subprogramas, o para agrupar muchos subprogramas en una sola unidad. Los módulos comienzan por su nombre `module nombre` y terminan con `end module nombre` (en los módulos no se utiliza el comando `return`). A diferencia de las funciones y subrutinas, si el módulo está en el mismo archivo que el programa principal, debe estar antes que este. Cualquier subprograma que haga uso del módulo debe hacerlo mediante el comando `use nombre` inmediatamente después del nombre del subprograma.

El uso más común de los módulos es para declarar variables que van a ser utilizadas por muchos subprogramas.

Ejemplo: Programa "usamodulos"

```
! Modulo "constantes".

module constantes

! Declarar parametros.

implicit none
real, parameter :: pi=3.141592, ee=2.71828

! Termina modulo.

end module constantes


! Programa principal
```

```
program usamodulos

! Usar modulo "constantes".

use constantes

! Declarar variables.

implicit none

real radio,area
real ee2

! Definir radio y calcular area y ee2.

radio = 10.0
area = pi*radio**2
ee2 = ee**2

! Escribir resultados a pantalla.

print *
print *, 'El area de un circulo de radio ',radio,' es ',area
print *
print *, 'El cuadrado el numero de Euler es ',ee2
print *

! Termina programa.

end program usamodulos
```