Capítulo 4

Vectores y matrices

En FORTRAN se puede utilizar un tipo especial de variable que sirve, en particular, para almacenar vectores y matrices. De esta forma, se utiliza un sólo nombre para referirse al conjunto de direcciones de memoria en las que se almacena la matriz o vector. En lenguaje informático a estos conjuntos de datos se les suele llamar **tablas** (también **arrays** o **tableros**).

4.1 Declaración de tablas

Una de las funciones del compilador es contar las unidades de memoria que un programa necesitará cuando se cargue en memoria para ser ejecutado. Por esta razón, cuando se utilizan tablas es necesario indicarle al compilador el número de elementos que contendrá (o podrá contener).

Para poder usar una tabla en un programa FORTRAN es imprescindible, pues, declararla: hay que indicar su nombre (nombre válido de variable), su tipo (entero, real s.p., ...) y sus dimensiones (número de subíndices y extensión de los mismos).

La declaración puede hacerse de forma **estática** o de forma **dinámica**.

Cuando se usa la forma estática, se declaran las dimensiones de la tabla ya durante la compilación. Esto significa que el recuento de la memoria necesaria para almacenar los elementos de la tabla tiene lugar durante la compilación, que dicha memoria se reserva cuando se carga el programa en memoria y que el tamaño de la tabla, y por tanto la cantidad de memoria que usa, se mantiene constante durante la ejecución del programa.

La declaración dinámica, por el contrario, permite indicar que se va a utilizar una tabla, pero sin especificar el numero y extensión de sus dimensiones hasta el momento en que se necesite, durante la ejecución del programa.

4.2 Declaración estática de tablas

En este tipo de declaración, la tradicional del lenguaje FORTRAN, se define el número de elementos de una tabla ya en tiempo de compilación, es decir se indican el numero de sus dimensiones y la extensión de las mismas de forma que el compilador puede hacer el recuento de la memoria necesaria. Puesto que en tiempo de compilación las variables no tienen aún

valor asignado, **no se pueden utilizar variables para declarar las dimensiones en este tipo de declaración**.

Hay varias formas posibles de escribir la declaración estática de una tabla. Recomendamos una de las siguientes:

```
TIPO, DIMENSION  :: nombre(dimensiones)
TIPO  ::  nombre(dimensiones)
```

Por ejemplo, para declarar una matriz real s.p. de nombre `rigidez` con `20` filas y `30` columnas se utilizaría una de las dos sentencias siguientes (equivalentes):

```
real*4, dimension  :: rigidez(20,30)
real*4  ::  rigidez(20,30)
```

Para declarar un vector entero, de nombre `nomvec`, con `100` elementos, se escribiría:

```
integer  :: nomvec(100)
```

Por supuesto, también se pueden utilizar tablas con más de dos dimensiones.

4.3 Declaración dinámica de tablas

En Fortran90 es posible declarar tablas de forma dinámica, lo que significa que la extensión de sus dimensiones se decide **durante la ejecución del programa**, aunque es preciso indicar, ya durante la compilación, el número de dimensiones, es decir, de subíndices, que va a tener.

La creación de este tipo de tablas se hace en dos etapas:

1. Declaración de la tabla mediante el atributo `ALLOCATABLE`, sin información sobre el tamaño de la misma.

   ```
   TIPO, ALLOCATABLE :: nombre_tabla(ind-dim)
   ```

   donde `ind-dim` es la indicación sobre el número de dimensiones de la tabla.

2. Cuando se desee crear la tabla, se usará la instrucción

   ```
   ALLOCATE(nombre_tabla(dim))
   ```

```
EJEMPLO:
      real*4, allocatable :: tabla(:,:)
      ...
      print*,'>> Dimension de la matriz ? '
      read*,n
      allocate(tabla(n,n))
```

Con la sentencia `ALLOCATE` se pueden crear varias tablas a la vez, separándolas por comas.

Una tabla creada con la sentencia `ALLOCATE` puede ser destruida, cuando ya no se use, con la sentencia:

```
DEALLOCATE(nombre_tabla)
```

Una tabla dinámica que ya haya sido creada (`ALLOCATE`) no puede ser creada de nuevo a menos que se destruya previamente. Para saber si una tabla dinámica está creada o no en un momento dado se puede emplear la función intrínseca

```
ALLOCATED(tabla)
```

que devuelve el valor lógico `.TRUE.` si `tabla` está creada y `.FALSE.` en caso contrario

4.4 Utilización de los elementos de una tabla

Las tablas se pueden utilizar globalmente, es decir referenciando el conjunto de sus elementos mediante su nombre, como en:

```
EJEMPLO:
      real*4  ::  a(10,10)
      a=0.
```

Este ejemplo declara una matriz real 10×10 y da a todos sus elementos el valor cero.

En general, una asignación de un escalar a una tabla es entendida por el compilador como una asignación **elemento a elemento**, es decir, que se le asigna el valor escalar a cada uno de los elementos de la tabla.

También se pueden utilizar elementos de una tabla de forma individual. Para ello se referencian mediante su nombre y los valores de sus subíndices encerrados entre paréntesis: `v(i)` para v_i y `a(i,j)` para a_{ij}

```
EJEMPLO:
      real*4  ::  a(10,10)
      do j=1,10
      do i=1,10
        a(i,j)=real(i+j)
      end do
      end do
```

Este ejemplo asigna a cada elemento `a(i,j)` de la matriz `a` el valor de la suma de sus subíndices.

Los subíndices de una tabla pueden ser constantes, variables o expresiones, necesariamente de **tipo entero**.

4.5 ¿Cómo se almacenan las tablas?

Las tablas se almacenan en direcciones consecutivas de memoria. Internamente, sólo se "anota" la dirección en la que se encuentra el primer elemento de la tabla. Para acceder al resto el control debe calcular su dirección "relativa", es decir, cuántos elementos hay almacenados entre el primero y el que se necesita.

En el caso de tablas de una sóla dimensión esto es fácil. Así, por ejemplo, si el vector real o entero `v` está almacenado a partir de la dirección de memoria número **iv**, el elemento `v(k)` estará almacenado en la dirección de memoria $\mathbf{iv} + \mathbf{k} - \mathbf{1}$.

	v(1)	v(2)	v(3)	...				v(12)	...		

En el caso de matrices, sus elementos también se almacenan en direcciones consecutivas de memoria **por columnas**. Por ejemplo, los elementos de una matriz `a(4,3)` estarán almacenados en la memoria en el orden siguiente:

	a(1,1)	a(2,1)	a(3,1)	a(4,1)	a(1,2)	a(2,2)	a(3,2)	a(4,2)	a(1,3)	...	a(4,3)

Así, si el primer elemento de la matriz anterior está almacenado en la dirección de memoria **ia**, el elemento `a(i,j)` (`i`-ésimo elemento de la `j`-ésima columna) está almacenado en la dirección de memoria $\mathbf{ia} + \mathbf{4}(\mathbf{j} - \mathbf{1}) + \mathbf{i} - \mathbf{1}$.

En general, para una tabla entera o real s.p. con `m` filas y `n` columnas, si el primer elemento está en la dirección **ia**, el elemento `a(i,j)` está en la dirección $\mathbf{ia} + \mathbf{m}(\mathbf{j} - \mathbf{1}) + \mathbf{i} - \mathbf{1}$. Obsérvese que no es necesario saber cuántas columnas tiene la matriz; solamente cuántos elementos hay en cada columna, es decir, cuántas filas.

Para las tablas doble precisión el procedimiento es el mismo, pero a la hora de calcular la dirección relativa de un elemento hay que tener en cuenta que cada elemento ocupa **dos palabras** de memoria.

4.6 Asignación de valores a tablas

Las versiones anteriores de FORTRAN sólo permitían la asignación de valores a los elementos de una tabla elemento a elemento, típicamente mediante la utilización de bucles `DO`.

FORTRAN90 incluye una buena cantidad de procedimientos para manipular tablas de forma global. De hecho, esta es, junto con la declaración dinámica de tablas, una de las principales aportaciones del FORTRAN90 frente al FORTRAN77.

Se pueden asignar valores a vectores escribiendo a la derecha del signo igual los valores encerrados entre los signos `(/` y `/)` , y separándolos por comas, como en:

```
EJEMPLO:
      real*4 :: v(5),w(5)
      v=(/ 1.,2.,3.,4.,5. /)
      x=1.1
      w=(/ 0.,x,x-1,0.,-1. /)
```

Obsérvese que se pueden utilizar otras variables para escribir los valores. También se puede incluir un vector que ya tenga valores asignados como parte de otro, como en:

```
EJEMPLO:
      real*4 :: v(5),w(3)
      w=(/ 1.2,3.1,-1.2 /)
      v=(/ 0., w ,0. /)
```

Otra posibilidad es utilizar el DO implícito:

```
EJEMPLO:
      integer :: nvec(5)
      nvec=(/ (i,i=1,5) /)
```

Este ejemplo asigna al vector entero nvec los valores nvec=(1,2,3,4,5).

La sintaxis general de un DO implícito es:

```
( valor, indice=min,max,paso )
```

donde, como en el DO indexado, si paso vale 1 se puede omitir.

```
EJEMPLO:
      integer :: nvec(10)
      real*4  :: v(10)
      pi=3.141593
      h=pi/9.
      nvec=(/ (i,i=2,20,2) /)
      v=(/ (cos((i-1)*h),i=1,10) /)
```

En este ejemplo se dan a la tabla entera nvec los valores

```
nvec=(2,4,6,8,10,12,14,16,18,20)
```

y a la tabla real s.p. v los valores v(i)=cos((i-1)*pi/9.), para i=1,2,...10.

Se pueden combinar valores con un DO implícito:

```
EJEMPLO:
      real*4 :: v(10)
      v=(/0.,(log(real(i)),i=1,9) /)
```

Debe tenerse en cuenta que con este mecanismo sólo se puede crear un **vector** (i.e. una tabla de una dimensión) luego, por sí solo, no es apto para asignar valores a una tabla con más de una dimensión. Más adelante se verá cómo puede hacerse.

También se pueden incluir tablas en expresiones con operadores aritméticos, como por ejemplo:

```
EJEMPLO:
      a=0.5 + 1./b
      c=a*b
```

Para ello las tablas deben ser compatibles, es decir, deben tener las mismas dimensiones. Las operaciones se realizan por elementos: con la instrucción a=0.5+1./b se asigna a cada elemento a(i,j) el valor 0.5+1./b(i,j) y con la instrucción c=a*b se asigna a cada elemento c(i,j) el valor a(i,j)*b(i,j).

Los escalares son compatibles con todas las tablas, ya que su aparición en una expresión aritmética con tablas se entiende siempre **elemento a elemento**. Así, la asignación a una

tabla de un valor escalar se entiende elemento a elemento, es decir, se asigna el valor escalar a cada elemento de la tabla. La suma de un escalar a una tabla se entiende como la suma del escalar a cada uno de los elementos de la tabla. La multiplicación de un escalar por una tabla tiene el sentido habitual, esto es, se multiplica cada elemento de la tabla por el escalar.

De la misma forma, se pueden utilizar tablas como argumentos de funciones instrínsecas. Por ejemplo,

```
EJEMPLO:
      real*4 :: a(5,5),b(5,5)
      a=exp(b)
```

asigna a cada elemento `a(i,j)` de la tabla `a` el valor `exp(b(i,j))`

Se puede aprovechar la instrucción de declaración de una tabla para inicializarla, igual que con las variables escalares:

```
EJEMPLO:
      real*4 :: a(5,5)=0.0
      real*4 :: v(11)=(/ (i/10.,i=0,10) /)
```

4.7 La función RESHAPE

Esta función sirve para reestructurar una tabla, interpretándola con otras dimensiones.

Puesto que los elementos de una tabla, como se explicó antes, están almacenados en direcciones consecutivas de memoria, ordenados de la forma natural si son vectores, y por columnas si se trata de matrices, es posible "reinterpretarlos", cambiando sus dimensiones.

Por ejemplo, supongamos la matriz entera M:

$$M = \begin{pmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{pmatrix}$$

Sus elementos están almacenados en la memoria en el orden siguiente:

M_{11}	M_{21}	M_{31}	M_{41}	M_{12}	M_{22}	M_{32}	M_{42}	M_{13}	M_{23}	M_{33}	M_{43}
1	2	3	4	5	6	7	8	9	10	11	12

Si "interpretamos" este conjunto de direcciones de memoria como una matriz 2×6 obtenemos:

$$\tilde{M} = \begin{pmatrix} 1 & 3 & 5 & 7 & 9 & 11 \\ 2 & 4 & 6 & 8 & 10 & 12 \end{pmatrix}$$

y si la "interpretamos" como una matriz 3×4 obtenemos

$$\hat{M} = \begin{pmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{pmatrix}$$

Para esto sirve la función `RESHAPE`. Su sintaxis es:

```
RESHAPE(input,forma)
```

donde `input` es la tabla que se quiere re-interpretar y `forma` es un vector entero indicando las nuevas dimensiones que se le quieren dar a la tabla `input`. Por ejemplo, con

```
reshape(M,(/2,6/))
```

se obtendría la matriz $\tilde{M}$.

Esta función se puede utilizar para asignar valores a tablas con más de una dimensión. Así, por ejemplo, para conseguir la matriz M anterior, se podría escribir:

```
integer :: m(4,3)
m=reshape( (/(i,i=1,12)/) , (/4,3/) )
```

y para asignar valores a la matriz 20×20:

$$A = \begin{pmatrix} 4 & 1 & 0 & \cdots & 0 \\ 0 & 4 & 1 & \cdots & 0 \\ 0 & 0 & 4 & \ddots & \\ \vdots & \vdots & & \ddots & 1 \\ 0 & 0 & 0 & \cdots & 4 \end{pmatrix}$$

se podría escribir:

```
real*4 :: a(20,20)
n=20
n1=19
a=reshape((/4.,(((/(0.,i=1,n1),1.,4./),j=1,n1)/),(/n,n/))
```

4.8 Secciones de una tabla

FORTRAN90 permite hacer referencia a un subconjunto de elementos de una tabla, "extrayendo" los elementos correspondientes a determinados subíndices.

Se puede indicar un rango de subíndices mediante la expresión `i1:i2`, que denota todos los subíndices entre `i1` e `i2`, ambos incluidos. Por ejemplo, `v(3:7)` hace referencia a la subtabla `(v(3),v(4),v(5),v(6),v(7))`, y `a(1:2,3:4)` hace referencia a la submatriz

$$\begin{pmatrix} \texttt{a}(1,3) & \texttt{a}(1,4) \\ \texttt{a}(2,3) & \texttt{a}(2,4) \end{pmatrix}$$

También pueden indicarse subíndices no consecutivos: `i1:i2:ip` representan los subíndices que van desde `i1` hasta `i2` con paso `ip`. Por ejemplo `2:11:3` representa los subíndices `2,5,8,11` y `v(2:11:3)` representa la subtabla `(v(2),v(5),v(8),v(11))`. Análogamente, `a(1:5:2,1:7:3)` representa la submatriz.

$$\begin{pmatrix} \texttt{a}(1,1) & \texttt{a}(1,4) & \texttt{a}(1,7) \\ \texttt{a}(3,1) & \texttt{a}(3,4) & \texttt{a}(3,7) \\ \texttt{a}(5,1) & \texttt{a}(5,4) & \texttt{a}(5,7) \end{pmatrix}$$

Otra forma de indicar un conjunto de subíndices es mediante un vector de números enteros. Por ejemplo `v((/1,5,5,3,9/))` representa el vector `(v(1),v(5),v(5),v(3),v(9))`.

4.9 Algunas funciones para tablas

La función

```
MATMUL(tabla1,tabla2)
```

calcula el producto matricial de las dos tablas. Ambas tablas deben ser del mismo tipo y sus dimensiones deben ser las adecuadas. El resultado es del mismo tipo que los argumentos.

La función

```
DOT_PRODUCT(vector1,vector2)
```

calcula el producto escalar de los dos vectores, que deben ser del mismo tamaño.

La función

```
MAXVAL(tabla)
```

devuelve el máximo valor entre sus elementos.

La función

```
TRANSPOSE(tabla)
```

donde `tabla` es necesariamente una tabla con dos dimensiones, es decir, una matriz, calcula la matriz transpuesta de `tabla`.

La subrutina

```
RANDOM_NUMBER(a)
```

devuelve números pseudo-aleatorios en el intervalo $[0, 1]$ con una distribución uniforme. El argumento de salida `a` tiene que ser real s.p. y puede ser un escalar o una tabla.

4.10 Lectura y escritura de tablas

La lectura y escritura de tablas puede hacerse:

1. Por elementos sueltos, que son tratados como escalares.

```
EJEMPLO:
      read*,v(k)
      print*,a(7,3)
```

2. Leyendo o imprimiendo la tabla completa, o una sección de la misma, sin especificación sobre el orden en que deben imprimirse sus elementos. En este caso, se leerán/imprimirán todos los elementos de la tabla o de la sección en el mismo orden en que están almacenados en la memoria, es decir, por columnas.

```
EJEMPLO:
      real*4 :: a(3,4)
      integer :: index(10,20)
      ...
      print*,a
      print*,index(1:5,1:5)
```

   La primera de las órdenes anteriores imprime todos los elementos de la tabla `a`. La segunda imprime una sub-tabla 5×5 de la tabla `index`. En ambos casos los elementos se imprimen de forma correlativa, recorriéndolos por columnas, sin separación entre ellas.

3. Leyendo o imprimiendo una tabla completa o una sección de la misma, pero controlando el orden y la disposición de los elementos. Ello se hace utilizando adecuadamente el `DO` indexado y el `DO` implícito.

```
EJEMPLO:
      real*4 :: a(4,4)
      do i=1,5
      print*,(a(i,j),j=1,4)
      end do
```

Las órdenes anteriores imprimen la tabla `a` por filas: para cada valor de `i` se imprimen, con una sola orden `print`, todos los elementos de la forma `a(i,j)`, para `j` variando desde `1` hasta `4`.

```
EJEMPLO:
      real*4 :: v(10)
      do i=1,10
      print*,v(i)
      end do
```

Las órdenes anteriores imprimen un elemento del vector `v` en cada fila.

4.11 Tablas y subprogramas

Cuando se habla de tablas en subprogramas hay que comenzar por distinguir claramente entre **variables locales** y **argumentos**.

La declaración de tablas locales de un subprograma se rige por las normas explicadas hasta ahora. Si son estáticas, hay que declararlas usando valores constantes para indicar la extensión de sus dimensiones. Si son dinámicas, es posible crearlas en tiempo de ejecución.

Sin embargo, cuando se trata de argumentos del subprograma, al no tenerse en cuenta en el recuento de memoria que el programa necesita, es posible declar tablas utilizando otros argumentos para definir la extensión de sus dimensiones.

```
EJEMPLO:
      program ejemplo
      real*4, allocatable :: a(:,:)
      print*,' >> Dimension: '
      read*,n
      allocate(a(n,n))
      call random_number(a)
      call  sumatodos(suma,a,n)
      print*,suma
      end program ejemplo
!
      subroutine sumatodos(suma,a,n)
      real*4 :: a(n,n)
      suma=0.
      do i=1,n
      do j=1,n
      suma=suma+a(i,j)
      end do
      end do
      end subroutine sumatodos
```