4 ARRAYS

4.1 Introducción

- En computación es frecuente trabajar con conjuntos ordenados de valores: listas o vectores y tablas o matrices. Para ello, existe una estructura de datos denominada *array*.
- Un array está constituido por un grupo de variables o constantes, todas del mismo tipo. Cada variable dentro del array se denomina *elemento del array.*
- Un array está definido por los siguientes parámetros:
    - Rango: es su número de dimensiones. Así, un escalar posee rango cero, un vector rango uno, una matriz rango dos, etc.
    - Extensión: es el total de componentes que posee en cada una de sus dimensiones. Por ejemplo, en una matriz de 5 filas y 3 columnas, la extensión de su dimensión primera (filas) es 5 y la de su dimensión segunda (columnas) es 3.
    - Tamaño: es el total de elementos que tiene, es decir, el producto de sus extensiones. En el ejemplo anterior, el tamaño de una matriz de 5 filas x 3 columnas es 15.
    - Perfil: es la combinación del rango y la extensión del array en cada dimensión. Por tanto, dos arrays tienen el mismo perfil si tienen el mismo rango y la misma extensión en cada una de sus dimensiones.

4.2 Declaración de arrays

- Antes de usar un array, es necesario declararlo en una sentencia de declaración de tipo. La sintaxis general es:

TIPO, DIMENSION (d1[,d2]...) :: lista de arrays

- TIPO es cualquiera de los tipos de datos Fortran válidos de la Tabla 1.2.
- El atributo DIMENSION permite especificar los valores de los índices máximo y, opcionalmente, mínimo, de cada una de las dimensiones del mismo. Así, la dimensión 1 o d1 de la sintaxis general anterior se sustituye por [límite_inferior_d1:]límite_superior_d1 y análogamente para las demás dimensiones, si las hay.
- La extensión de un array en una dimensión n dn viene dada por la ecuación: Extensión_dn=límite_superior_dn-límite_inferior_dn+1
- Los valores de los límites inferior y superior pueden ser positivos, negativos o 0.
- Si sólo se identifica el límite superior, el límite inferior es por defecto 1.

- El rango máximo de un array es 7.
- *lista de arrays* es un conjunto de arrays separados por comas cuyos nombres son identificadores válidos.
- Ejemplos de sentencias de declaración de arrays:

```
INTEGER, DIMENSION (4) :: vector1,vector2
INTEGER, DIMENSION (-3:0) :: vector3
CHARACTER (len=20), DIMENSION (50) :: nombres_alumnos
REAL, DIMENSION (2,3) :: matriz1,matriz2
REAL, DIMENSION (0:2,-2:3) :: matriz3,matriz4
```

- El compilador usa las sentencias de declaración de arrays para reservar el espacio adecuado en la memoria del computador Los elementos de un array ocupan posiciones *consecutivas* en la memoria. En el caso de arrays de rango 2, se almacenan por columnas, es decir, el primer índice toma todos los valores permitidos por su extensión para cada uno de los valores permitidos para el segundo índice.
- En programas largos y complicados, es conveniente usar el atributo PARAMETER para dar nombres a los tamaños de los arrays declarados. De esta manera, sus valores se cambiarán fácilmente.
- Ejemplos:

```
INTEGER, PARAMETER :: TM=50,TMF=2,TMC=3
CHARACTER (len=20), DIMENSION (TM) :: nombres_alumnos
REAL, DIMENSION (TMF,TMC) :: matriz1,matriz2
```

4.3 Referencia a los elementos de un array

- La sintaxis general de un elemento de un array es:

```
nombre_array ( i1 [,i2] ... )
```

- índice i1 es un entero que verifica la siguiente condición límite_inferior_d1 ≤ i1 ≤ límite_superior_d1 y análogamente para los demás índices. Es decir, los valores de los índices permitidos vienen determinados por la extensión de la dimensión correspondiente, definida en la sentencia de declaración del array en cuestión. Por lo tanto, no se puede usar como índice ningún entero fuera de ese intervalo.
- Ejemplo. Sea la declaración siguiente:

```
REAL, DIMENSION (-1:1,3) :: X
```

- Elementos válidos de X son:

```
X(-1,1)    X(-1,2)    X(-1,3)
X( 0,1)    X( 0,2)    X( 0,3)
X( 1,1)    X( 1,2)    X( 1,3)
```

- No todos los compiladores comprueban que los índices de un array están dentro de sus límites.

4.4 Inicialización de arrays

- Al igual que las variables, los arrays deben inicializarse antes de usarse. Hay tres formas de inicializar un array:
    - En la propia sentencia de declaración de tipo del array, en tiempo de compilación.
    - En sentencias de asignación.
    - En sentencias de lectura (READ).

4.4.1 Inicialización de arrays en sentencias de declaración de tipo

- Para inicializar vectores pequeños, se usa el constructor de arrays, con los delimitadores (/ y /).
- Por ejemplo, para inicializar un vector v1 de 5 elementos con los valores 1, 2, 3, 4 y 5:

```
INTEGER, DIMENSION (5) :: v1 = (/ 1, 2, 3, 4, 5/)
```

- Para inicializar arrays de rango>1 se usa una función intrínseca especial llamada RESHAPE que cambia el perfil de un array sin cambiar su tamaño. Esta función tiene dos argumentos principalmente:

```
RESHAPE (fuente, perfil)
```

- *fuente* es un vector de constantes con los valores de los elementos del array.
- *perfil* es un vector de constantes con el perfil deseado.
- El array construido tendrá sus elementos dispuestos en su orden natural, es decir, si se trata de un array de rango dos, por columnas.
- Ejemplo:

```
INTEGER, DIMENSION (2,3) :: mat1 = &
RESHAPE ( (/ 1, 2, 3, 4, 5, 6/) , (/2,3/) )
```

Matemáticamente, la matriz construida es: $mat1 = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$

- Para inicializar arrays de tamaño grande, se usa un bucle DO implícito. La sintaxis general de un bucle implícito es:

```
(arg1[, arg2] ..., índice = inicio, fin [, paso])
```

- *arg1* y los demás argumentos, si los hay, se evalúan cada iteración del bucle.

- El índice del bucle y los parámetros del mismo funcionan exactamente igual que en el caso de los bucles DO iterativos estudiados en la sección 3.2.
- Ejemplos:

```
INTEGER, DIMENSION (5) :: v1 = (/ (i, i= 1, 5) /)
INTEGER, DIMENSION (100) :: v2 = (/ (i, i= 100, 1, -1) /)
```

- Los bucles DO implícitos pueden anidarse. La sentencia siguiente inicializa los elementos del vector v3 a 0 si no son divisibles por 5 y al número de elemento si son divisibles por cinco.

```
INTEGER, DIMENSION (20) :: v3 = (/ ((0, i= 1,4),5*j, j=1,4) /)
```

- El bucle interno (0, i=1,4) se ejecuta completamente para cada valor del bucle externo j. Se obtiene:

```
 0, 0, 0, 0, 5, 0, 0, 0, 0, 10, 0, 0, 0, 0, 15, 0, 0, 0, 0, 20
```

- También es posible inicializar todos los elementos de un array a un valor único constante.
- Ejemplo:

```
INTEGER, DIMENSION (100) :: v = 0
INTEGER, DIMENSION (2,3) :: mat=0
```

4.4.2 Inicialización de arrays en sentencias de asignación

- Los procedimientos vistos en el apartado anterior para inicializar arrays, son también válidos cuando se realizan en el cuerpo de un programa. Por ejemplo, sean las declaraciones de arrays:

```
INTEGER, DIMENSION (5) :: v1,v2
INTEGER, DIMENSION (2,3) :: mat1
…
v1 = (/ 1, 2, 3, 4, 5/)
v2 = (/ (i, i= 10, 6, -1) /)
mat1 = RESHAPE ( (/ 1, 2, 3, 4, 5, 6/) , (/2,3/) )
```

- También se pueden inicializar arrays usando bucles DO iterativos. Por ejemplo, sean las declaraciones de arrays:

```
INTEGER :: i,j,k=0
INTEGER, DIMENSION (5) :: v1,v2
INTEGER, DIMENSION (2,3) :: mat1
…
DO i=1,5
  v1(i)=i
END DO
```

```
DO i=10,6,-1
  v2(11-i)=i
END DO

externo : DO j=1,3
  interno :DO i=1,2
    k=k+1
    mat1(i,j)=k
  END DO interno
END DO externo
```

- También es posible inicializar todos los elementos de un array a un valor único con una simple sentencia de asignación.
- Ejemplo:

```
INTEGER, DIMENSION (100) :: v1
INTEGER, DIMENSION (2,3) :: mat
v1=0
mat=0
```

4.4.3 Inicialización de arrays en sentencias de lectura

- Se pueden leer arrays usando bucles DO iterativos. Por ejemplo, sean las declaraciones:

```
INTEGER :: i,j
INTEGER, PARAMETER :: TM=5,TMF=2,TMC=3
INTEGER, DIMENSION (TM) :: v1,v2
INTEGER, DIMENSION (TMF,TMC) :: mat1
…
DO i=1,TM
  WRITE (*,*) 'DAME ELEMENTO',i,'DE v1 Y v2'
  READ (*,*) v1(i),v2(i)
END DO

DO i=1,TMF
  DO j=1,TMC
    WRITE (*,*) 'DAME ELEMENTO',i,j,'DE mat1'
    READ (*,*) mat1(i,j)
```

```
!lectura por filas
  END DO
END DO


DO j=1,TMC
  DO i=1,TMF
    WRITE (*,*) 'DAME ELEMENTO',i,j,'DE mat1'
    READ (*,*) mat1(i,j)
!lectura por columnas
  END DO
END DO
```

- En los ejemplos anteriores, la lectura de los arrays se realiza elemento a elemento en el orden establecido en los bucles.
- También se puede leer un array usando bucles DO implícitos. La sintaxis general de lectura con un bucle implícito es:

```
READ (*,*) (arg1[, arg2] ..., índice = inicio, fin [, paso])
```

- *arg1* y los demás argumentos, si los hay, son los argumentos que se van a leer.
- El índice del bucle y los parámetros del mismo funcionan exactamente igual que en el caso de los bucles DO iterativos estudiados en la sección 3.2.
- Ejemplos:

```
INTEGER, DIMENSION (5) :: w
INTEGER, DIMENSION (0:3,2) :: m
...
READ (*,*) ( w(i), i=5,1,-1)
READ (*,*) ( (m(i,j), i=0,3) , j=1,2) !lectura por columnas
READ (*,*) ( (m(i,j), j=1,2) , i=0,3) !lectura por filas
```

- En los ejemplos anteriores, la lectura de los arrays se realiza elemento a elemento en el orden establecido en los bucles.
- También es posible leer un array escribiendo únicamente su nombre, sin especificar su(s) índice(s). En ese caso, el compilador asume que se va a leer todo el array, es decir, todos sus elementos en el orden natural Fortran.
- Ejemplo:

```
INTEGER, DIMENSION (10) :: v
INTEGER, DIMENSION (2,3) :: mat
READ (*,*) v
```

READ (*,*) mat

- Todas las formas anteriores usadas para leer arrays son también válidas para escribir arrays, sin más que sustituir la palabra reservada READ por WRITE, en las instrucciones correspondientes.

4.5 Operaciones sobre arrays completos

- Cualquier elemento de un array se puede usar como cualquier otra variable en un programa Fortran. Por tanto, un elemento de un array puede aparecer en una sentencia de asignación tanto a la izquierda como a la derecha de la misma.
- Los arrays completos también se pueden usar en sentencias de asignación y cálculos, siempre que los arrays involucrados tengan el mismo perfil, aunque no tengan el mismo intervalo de índices en cada dimensión. Además, un escalar puede operarse con cualquier array.
- Una operación aritmética ordinaria entre dos arrays con el mismo perfil se realiza *elemento a elemento*.
- Si los arrays no tienen el mismo perfil, cualquier intento de realizar una operación entre ellos provoca un error de compilación.
- La mayoría de las funciones intrínsecas comunes (ABS, SIN, COS, EXP, LOG, etc.) admiten arrays como argumentos de entrada y de salida, en cuyo caso la operación se realiza elemento a elemento. A este tipo de funciones intrínsecas se las llama funciones intrínsecas *elementales*.
- Existen otros dos tipos de funciones intrínsecas, las llamadas de *consulta* y las *transformacionales*. Las primeras operan con arrays completos y las segundas proporcionan información sobre las propiedades de los objetos estudiados.
- Ejemplo de operaciones sobre arrays completos:

```
INTEGER, DIMENSION (5) :: v1 =(/ 9,8,7,6,5 /)

INTEGER, DIMENSION (20:24) :: v2 =(/ -9,-8,-7,-6,-5 /)

INTEGER, DIMENSION (50:54) :: v3,v4


v3=v1+v2

v4=v1*v2

WRITE (*,*) 'v3, dos veces exponencial de v3', v3, 2*EXP(REAL(v3))

WRITE (*,*) 'v4, valor absoluto de v4', v4, ABS(v4)
```

Se escriben v3, dos veces exponencial de v3: 0 0 0 0 0, 2, 2, 2, 2, 2 y en otra línea v4, valor absoluto de v4: -81, -64, -49, -36, -25, 81, 64, 49, 36, 25.

- Otra forma de codificar lo anterior, operando con los índices de los arrays en bucles DO iterativos es (usar el mismo rango de índices en todos los arrays, para facilitar la tarea, por ejemplo de 1 a 5):

```
INTEGER :: i
INTEGER, DIMENSION (5) :: v1 =(/ 9,8,7,6,5 /),&
v2 =(/ -9,-8,-7,-6,-5 /),v3,v4

DO i=1,5
  v3(i)=v1(i)+v2(i)
  v4(i)=v1(i)*v2(i)
  WRITE (*,*) v3(i), 2*EXP(REAL(v3(i)))
  WRITE (*,*) v4(i), ABS(v4(i))
END DO
```

4.6 Operaciones sobre subconjuntos de arrays

- Hasta ahora hemos visto que es posible usar elementos de arrays o array completos en sentencias de asignación y cálculos. Asimismo, es posible usar cualquier subconjunto de un array, llamado *sección de array*.
- Para especificar una sección de un array, basta sustituir el índice del array por:
  - un *triplete de índices* o
  - un *vector de índices*.

4.6.1 Tripletes de índices

- Sea la declaración:

```
TIPO, DIMENSION (d1[,d2]...):: nombre_array
```

- La sintaxis general de una sección del array anterior con tripletes de índices es:

```
nombre_array (trip1, [tripl2]...)
```

- El triplete de índices se puede usar para especificar secciones de un array en cualquiera de sus dimensiones o en todas, según interese. La sintaxis general de un triplete *tripl* es:

```
[indice_inicial_tripl] : [indice_final_tripl] [: paso_tripl]
```

- El triplete en una dimensión dada especifica un subconjunto *ordenado* de índices del array *nombre_array* para esa dimensión, empezando con *indice_inicial* y acabando con *indice_final* con incrementos dados por el valor de *paso*.
- Cualquier valor del triplete o los tres son opcionales:

  - Si *indice_inicial* no aparece, su valor por defecto es el índice del primer elemento del array.
  - Si *indice_final* no aparece, su valor por defecto es el índice del último elemento del array.
  - Si *paso* no aparece, su valor por defecto es 1.
- Ejemplo de secciones de array con tripletes de índices:

Sea la matriz= $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$, la sección del array formada por los elementos de la primera fila se puede especificar con el triplete: matriz(1,:)=$\begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix}$. La sección del array formada por los elementos de la primera columna se puede especificar con el triplete: matriz(:,1)=$\begin{pmatrix} 1 \\ 5 \\ 9 \\ 13 \end{pmatrix}$. La sección del array formada por los elementos de las filas impares y columnas pares es: matriz(1:3:2,2:4:2)=$\begin{pmatrix} 2 & 4 \\ 10 & 12 \end{pmatrix}$.

4.6.2 Vectores de índices

- Un vector de índices es un array entero *unidimensional* que especifica los elementos de un array a usar en un cálculo. Los elementos del array se pueden especificar en *cualquier orden* y pueden aparecer *más de una vez*. En este último caso, la sección del array no puede aparecer a la izquierda de una sentencia de asignación pues ello significaría que dos o más valores diferentes tendrían que ser asignados al mismo elemento del array.
- Ejemplo de sección de un vector con un vector de índices:

```
INTEGER, DIMENSION (5) :: v_indices =(/ 1,6,3,4,6 /)
REAL, DIMENSION (8) :: v =(/ -1.,2.,3.3,4.0,-5.5,0,-7.0,8.8 /)
```

Según las sentencias de declaración anteriores, v(v_indices) especifica los valores de los elementos v(1), v(6), v(3), v(4), v(6), es decir, -1. 0 3.3 4.0 0.

4.7 Construcción WHERE

- La construcción WHERE permite llevar a cabo asignaciones y cálculos sólo sobre determinados elementos de uno o más arrays. Para ello, esta construcción trabaja con la ayuda de un filtro o máscara formada por un *array lógico* en la propia sentencia WHERE.

- La sintaxis general de una construcción WHERE es similar a la de un bloque IF:

```
[nombre:] WHERE (expresión máscara 1)
  bloque 1 de sentencia(s) de asignación de arrays
[ELSEWHERE (expresión máscara 2) [nombre]
  bloque 2 de sentencia(s) de asignación de arrays]
...
[ELSEWHERE [nombre]
  bloque n de sentencia(s) de asignación de arrays]
END WHERE [nombre]
```

- Expresión máscara es un array lógico del mismo perfil que los arrays manipulados en las sentencias de asignación de arrays de los bloques.
- Puede haber cualquier número de cláusulas ELSEWHERE con máscara y, a lo sumo, una cláusula ELSEWHERE.
- Su funcionamiento es el siguiente:
    - Se aplica la operación o conjunto de operaciones del bloque 1 a todos los elementos del array para los cuales la expresión máscara 1 es cierta.
    - Se aplica la operación o conjunto de operaciones del bloque 2 a todos los elementos del array para los cuales la expresión máscara 1 es falsa y la expresión máscara 2 es cierta.
    - Se repite el razonamiento anterior para todas las cláusulas ELSEWHERE con máscara que haya.
    - Finalmente, se aplica la operación o conjunto de operaciones del bloque n a todos los elementos del array para los cuales todas las expresiones máscara anteriores han sido falsas.
- Ejemplo. Calcular la raíz cuadrada de los elementos positivos de un array. Poner un cero en el resto de los elementos.

```
REAL, DIMENSION (5,4) :: mati,matf
...
WHERE (mati>0)
  matf=SQRT(mati)
ELSEWHERE
  matf=0
END WHERE
```

La expresión máscara mati>0 produce un array lógico cuyos elementos son cierto cuando los elementos correspondientes de mati son positivos y falso cuando los elementos correspondientes de mati son negativos o cero.

Otro modo, combinando dos bucles anidados con un bloque IF:

```
recorre_filas: DO i=1,5
  recorre_columnas: DO j=1,4
    elemento_positivo: IF (mati(i,j)>0) THEN
      matf(i,j)=SQRT(mati(i,j))
    ELSE
      matf(i,j)=0
    ENDIF elemento_positivo
  END DO recorre_columnas
END DO recorre_filas
```

- La construcción WHERE es generalmente más elegante que las operaciones efectuadas elemento a elemento, especialmente con arrays multidimensionales.

4.8 Sentencia WHERE

- Es el caso particular más simple de una construcción WHERE.
- Su sintaxis general es:

```
WHERE (expresión máscara) sentencia de asignación de arrays
```

- La sentencia de asignación se aplica sólo sobre aquellos elementos del array para los cuales la expresión máscara es cierta.

4.9 Construcción FORALL

- Aparece en Fortran 95 para realizar un conjunto de operaciones efectuadas elemento a elemento sobre un subconjunto de elementos de un array.
- Su sintaxis general es:

```
[nombre:] FORALL (ind1=tripl1 [,ind2=tripl2]...[,expresión lógica])
  sentencia1
  [sentencia2]
  ...
END FORALL [nombre]
```

- Cada índice ind se especifica por un triplete de la forma vista en el apartado 4.6.1: [*ind_inicial_tripl*] : [*ind_final_tripl*] [: *paso_tripl*].
- La sentencia o sentencias que forman el cuerpo de la estructura se ejecutan sólo sobre aquellos elementos del array que cumplen las restricciones de índices y la expresión lógica dada en la cláusula FORALL.

- Ejemplo: calcular el inverso de cada elemento perteneciente a la diagonal principal de una matriz de 7x7 evitando las divisiones por cero.

```
REAL, DIMENSION (7,7) :: mat
...
FORALL (i=1:7, mat(i,i)/=0)
  mat(i,i)=1./mat(i,i)
END FORALL
```

Otro modo, combinando dos bucles anidados y un bloque IF:

```
recorre_diagonal: DO i=1,7
  IF (mat(i,i)/=0) mat(i,i)=1./mat(i,i)
END DO recorre_diagonal
```

- Todo lo que se programa con una construcción FORALL se puede programar con un conjunto de bucles anidados combinado con un bloque IF.
- La diferencia entre ambas formas es que mientras en los bucles las sentencias deben ser ejecutadas en un orden estricto, la construcción FORALL puede ejecutarlas en cualquier orden, seleccionadas por el procesador. Así, un computador con varios procesadores en paralelo puede repartirse los elementos de una sentencia optimizando la velocidad de cálculo. Cuando el cuerpo de la estructura FORALL contiene más de una sentencia, el computador procesa completamente todos los elementos involucrados en la misma antes de pasar a la siguiente. Por tanto, no pueden usarse funciones cuyo resultado dependa de los valores de un array completo (funciones transformacionales).
- Ejemplo:

```
REAL, DIMENSION (7,7) :: mat
...
FORALL (i=1:7, j=1:7,mat(i,j)/=0)
  mat(i,j)=SQRT(ABS(mat(i,j)))
  mat(i,j)=1./mat(i,j)
END FORALL
```

4.10 Sentencia FORALL

- Es el caso particular más simple de una construcción FORALL.
- Su sintaxis general es:

```
FORALL (ind1=tripl1 [,ind2=tripl2]...[,expresión lógica]) sentencia
```

- La sentencia de asignación se aplica sólo sobre aquellos elementos del array cuyos índices cumplen las restricciones y la expresión lógica dadas.

4.11 Arrays dinámicos

- Hasta ahora el tamaño de los arrays se ha especificado en las propias sentencias de declaración de tipo. Se dice que son arrays *estáticos* en cuanto que tamaño se fija en tiempo de compilación y a partir de entonces no se puede modificar.
- Sin embargo, lo ideal sería que el programador pudiera dar el tamaño de los arrays al ejecutar sus programas según sus necesidades, para aprovechar eficazmente la memoria del computador.
- A partir de Fortran 90, es posible usar arrays cuyo tamaño se fija en tiempo de ejecución, los llamados arrays *dinámicos*.
- La sintaxis general de una sentencia de declaración de arrays dinámicos es:

```
TIPO, ALLOCATABLE, DIMENSION (:[,:]...):: lista_de_arrays
```

- ALLOCATABLE es el atributo que declara que los arrays de la *lista* serán dimensionados dinámicamente. El número de sus dimensiones se declara con dos puntos :, sin especificar los límites.
- La memoria se asigna en tiempo de ejecución para la lista de arrays dinámicos, usando una sentencia ALLOCATE, que especifica los tamaños:

```
ALLOCATE ( arr1( d1 [ ,d2] ... ) [, arr2( d1 [ ,d2]] ...) [,STAT=estado] )
```

- Esta instrucción asigna memoria en tiempo de ejecución para la lista de arrays previamente declarada con el atributo ALLOCATABLE. Establece para cada array su tamaño, con los límites inferior y superior de cada dimensión.
- STAT= *estado*. Cualquier fallo en la localización de memoria causa un error en tiempo de ejecución, a menos que se use este parámetro. La variable entera *estado* devuelve un cero si se ha conseguido reservar espacio suficiente en memoria, en otro caso devuelve un número de error que depende del compilador.
- Se debe liberar la memoria reservada dinámicamente cuando ya no hace falta escribiendo la instrucción:

```
DEALLOCATE (lista_de_arrays [, STAT=estado])
```

- Esta sentencia es útil en programas grandes con necesidades grandes de memoria. En ellos, se usará la sentencia DEALLOCATE en la línea de código a partir de la cual determinados arrays ya no juegan ningún papel, liberando ese espacio de memoria que queda por lo tanto disponible para realizar nuevas reservas de espacio.

- Cualquier fallo al intentar liberar la memoria causa un error en tiempo de ejecución, a menos que el parámetro STAT= *estado* esté presente. La variable *estado* devuelve un cero si la liberación de memoria se hizo con éxito, en otro caso devuelve un número de error que depende del compilador.
- Ejemplo:

```
INTEGER ::estado_reserva, estado_libera
REAL, ALLOCATABLE, DIMENSION (:,:) :: alfa,beta
……
ALLOCATE (alfa(1:100, -1:200), STAT=estado_reserva)
IF (estado_reserva/=0) STOP ‘array alfa NO guardado en memoria’
…
DEALLOCATE (alfa, STAT= estado_libera)
IF (estado_libera/=0) STOP ‘NO liberada memoria para array alfa’
ALLOCATE (beta(1:1000, 200), STAT=estado_reserva)
IF (estado_reserva/=0) STOP ‘array beta NO guardado en memoria’
…
```

EJERCICIOS RESUELTOS

Objetivos:

Aprender a usar arrays en Fortran, comenzando por los vectores y ampliando los conceptos a matrices bidimensionales y tridimensionales.

Por un lado, se usan bucles DO iterativos cuyos índices serán los índices de los arrays, para recorrer los elementos de los mismos. Por otro lado, se manejan las estructuras vistas en este capítulo para trabajar con secciones de arrays o arrays completos.

1. Pedir cinco números por teclado y calcular su media usando un vector para almacenar los números.

```
PROGRAM cap4_1
IMPLICIT NONE
INTEGER :: i
REAL :: media=0
REAL,DIMENSION (5) :: v
DO i=1,5
  WRITE(*,*) 'DAME COMPONENTE',i
  READ(*,*) v(i)
END DO
DO i=1,5
  media=media+v(i)
END DO
media=media/5
WRITE(*,*) 'LA MEDIA ES:',media
END PROGRAM cap4_1
```

– Este programa puede resolverse declarando únicamente una variable (como se hizo en el capítulo anterior). La motivación de almacenar en memoria los cinco números estaría, en el caso de que el programa fuera más grande, y necesitara de sus valores para seguir operando con ellos más adelante.

– Los cinco números pueden guardarse en cinco variables o en un vector de 5 componentes. Lógicamente, la utilidad de un vector aumenta cuantos más números tengamos que almacenar.

2. Calcular y presentar en pantalla los valores 3 a N (100 como máximo) de la sucesión: $X_i=X_{i-1}+ 3X_{i-2} + 4 / 3$. El programa leerá por teclado los valores de N, X_1 y X_2.

```
PROGRAM cap4_2
REAL, DIMENSION(100) :: x
DO
  WRITE(*,*) 'DAME EL VALOR DE N (100 COMO MAXIMO)'
  READ(*,*) n
  IF (n<=100) EXIT
  WRITE(*,*) 'COMO MUCHO 100!'
END DO
```

```
WRITE(*,*) 'DAME EL VALOR DE X1 Y X2'
READ(*,*) x(1),x(2)
DO i=3, n
  x(i)=x(i-1)+3*x(i-2)+4.0/3
  WRITE(*,*) 'X(',i,'):',x(i)
END DO
END PROGRAM cap4_2
```

– Se dimensiona el vector X a 100 componentes, valor máximo que pide el ejercicio. De este modo, se reserva el máximo espacio en memoria que se puede ocupar, aunque generalmente sobre.

– Un bucle WHILE controla que el usuario escribe un valor para N menor o igual que el máximo, 100.

3. Buscar un número en un vector de seis componentes desordenado. Introduce el vector en la propia sentencia de declaración de tipo.

```
PROGRAM cap4_3
INTEGER :: x,switch=0,i
INTEGER, DIMENSION(6):: v=(/17,3,22,11,33,19/)
WRITE(*,*) 'DAME UN NUMERO'
READ(*,*) x
DO i=1,6
  IF (x == v(i)) THEN
    WRITE(*,*) 'ENCONTRADO EN POSICION',i
    switch=1
  END IF
END DO
IF (switch == 0) THEN
  WRITE(*,*) 'NO ENCONTRADO'
END IF
END PROGRAM cap4_3
```

– El bucle DO iterativo permite recorrer todas las componentes del vector y mostrar todas las posiciones del mismo en que encuentra el valor buscado X, si las hay.

– Cuando el valor buscado no coincide con ninguna componente del vector, la variable interruptor o switch tendrá el valor cero y sirve de “chivato” para conocer este hecho.

4. Realizar la búsqueda de un número en un vector. Reducir la búsqueda a encontrar la primera posición del array en que se encuentra coincidencia. Si no existe el nº en el array, mostrar un mensaje en consecuencia.

```
PROGRAM cap4_4
INTEGER :: num,i,chivato=0
INTEGER,PARAMETER :: NC=5
INTEGER, DIMENSION(NC) ::vec=(/1,1,6,6,7/)
WRITE(*,*) 'DAME UN NUMERO'
READ(*,*) num
DO i=1,NC
 IF (vec(i) == num) THEN
  WRITE(*,*) 'PRIMERA COINCIDENCIA EN POSICION',i,'DEL VECTOR'
  chivato=1
  EXIT
 END IF
END DO
IF (chivato == 0) THEN
 WRITE(*,*) 'NO EXISTE EL NUMERO EN EL VECTOR'
END IF
END PROGRAM cap4_4
```

– La sentencia EXIT del bucle WHILE permite reducir la búsqueda a la primera coincidencia en el vector.

5. Buscar un número en un vector de siete componentes ordenado ascendentemente. Introduce el vector en la propia sentencia de declaración de tipo.

```
PROGRAM cap4_5
INTEGER :: x,i=0
INTEGER,DIMENSION(7):: v=(/0,3,5,7,11,11,23/)
WRITE(*,*) 'DAME UN NUMERO'
READ(*,*) x
DO
 i=i+1
 IF (i==7 .OR. v(i)>=x) EXIT
END DO
IF (x == v(i)) THEN
```

```
  WRITE(*,*) 'ENCONTRADO EN POSICION',i
ELSE
  WRITE(*,*) 'NO ENCONTRADO'
END IF
END PROGRAM cap4_5
```

– Este algoritmo recorre todas las componentes del vector de izquierda a derecha hasta que, o bien llegamos a la última componente, o bien el valor buscado es inferior a algún elemento del vector (pues está ordenado).

– El programa encuentra únicamente la primera posición del vector en que se halla el valor buscado, aunque éste aparezca repetido en varias posiciones del mismo.

6. Buscar un número en un vector de siete componentes ordenado ascendentemente. Introduce el vector en la propia sentencia de declaración de tipo. Usa el algoritmo de la búsqueda binaria o dicotómica.

```
PROGRAM cap4_6
INTEGER :: x,izq=1,der=7,cen
INTEGER, DIMENSION(7)::v=(/-2,0,4,6,8,19,23/)
WRITE(*,*) 'DAME UN NUMERO'
READ(*,*) x
cen=(izq+der)/2
WRITE(*,*) 'CENTRO',cen
DO WHILE (x/=v(cen).AND.izq<der)
  IF (x>v(cen)) THEN
    izq=cen+1
  ELSE
    der=cen-1
  END IF
  cen=(izq+der)/2
  WRITE(*,*) 'CENTRO',cen
END DO
IF (x==v(cen)) THEN
  WRITE(*,*) 'ENCONTRADO EN POSICION',cen
ELSE
  WRITE(*,*) 'NO EXISTE EL VALOR BUSCADO'
END IF
```

```
END PROGRAM cap4_6
```

– La búsqueda dicotómica es un algoritmo más eficiente que el anterior. Aquí la búsqueda se realiza siempre en la componente central de un vector cuya longitud se va reduciendo a la mitad izquierda o derecha del centro (según sea el número) sucesivamente hasta que, o bien encontramos el valor buscado, o bien el intervalo de búsqueda se ha reducido a una componente.

7. Ordenar ascendentemente las seis componentes de un vector. Usar el método de la burbuja.

```
PROGRAM cap4_7
INTEGER::aux,i,j
INTEGER, PARAMETER ::NC=6
INTEGER, DIMENSION(NC) :: vec
vec=(/8,5,15,2,-19,0/)
cantidad_parejas:DO i=NC-1,1,-1
  pareja:DO j=1,i
    ordenar_pareja:IF (vec(j) > vec(j+1)) THEN
      aux=vec(j)
      vec(j)=vec(j+1)
      vec(j+1)=aux
    END IF ordenar_pareja
  END DO pareja
END DO cantidad_parejas
WRITE(*,*) 'EL VECTOR ORDENADO ES'
!$$$$$$ DO i=1,NC
!$$$$$$   WRITE(*,*) vec(i)
!$$$$$$ END DO
WRITE(*,*) (vec(i),i=1,NC)
END PROGRAM cap4_7
```

– Este algoritmo considera dos bucles DO iterativos anidados para realizar el ordenamiento; el bucle externo controla el número de parejas consideradas en el tratamiento y el bucle interno controla la pareja evaluada. A su vez, el bloque IF controla que tal pareja esté ordenada ascendentemente, intercambiando sus valores en caso contrario.

– Para ordenar el vector descendentemente (de mayor a menor) basta cambiar el operador relacional "mayor que" (>) por "menor que" (<) en el código del programa.

8. Leer una matriz de 2 filas y 3 columnas por teclado y mostrarla por pantalla.

```
PROGRAM cap4_8
IMPLICIT NONE
INTEGER::i,j
INTEGER, DIMENSION(2,3):: mat
WRITE(*,*) 'DAME LAS COMPONENTES DE LA MATRIZ'
READ(*,*) mat
filas: DO  i=1,2
 columnas: DO  j=1,3
  WRITE(*,*) 'COMPONENTE',i,j,':',mat(i,j),LOC(mat(i,j))
 END DO columnas
END DO filas
END PROGRAM cap4_8
```

– El interés del ejercicio radica en conocer cómo se almacenan las matrices en memoria. Por defecto, las matrices almacenan sus componentes en posiciones de memoria consecutivas *por columnas*.

– El nombre de la matriz es un puntero a la primera componente, es decir, es una variable que almacena la dirección de memoria de la primera componente de la matriz.

– LOC(*arg*) es una función intrínseca que recibe como argumento una variable de cualquier tipo o elemento de array y devuelve la dirección de memoria en la que está almacenado.

MAT					
MAT(1,1)	MAT(2,1)	MAT(1,2)	MAT(2,2)	MAT(1,3)	MAT(2,3)

9. Inicializar las componentes de una matriz de 2X3 en su sentencia de declaración de tipo a ar=$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$.

```
PROGRAM cap4_9
```

```
INTEGER,DIMENSION(2,3)::ar=&
RESHAPE((/1,1,2,2,3,3/),(/2,3/))
columnas: DO  j=1,3
 filas: DO  i=1,2
   WRITE(*,*)
   WRITE(*,*) 'COMPONENTE',i,j,'=',ar(i,j)
   WRITE(*,*)
   WRITE(*,*) 'DIRECCION DE MEMORIA',LOC(ar(i,j))
 END DO filas
END DO columnas
END PROGRAM cap4_9
```

– La función intrínseca RESHAPE se utiliza para inicializar el array bidimensional *ar*. Se verifica que la carga de los valores de la matriz se realiza por columnas.

10. Buscar un número en una matriz desordenada de tres filas y cinco columnas (3X5). Leer la matriz con un bucle implícito anidado.

```
PROGRAM cap4_10
INTEGER::x,sw,i,j
INTEGER, PARAMETER:: NF=3,NC=5
INTEGER, DIMENSION(NF,NC):: mat
WRITE(*,*) 'dame la matriz por columnas'
READ(*,*) ((mat(i,j),i=1,NF),j=1,NC)
sw=0
WRITE(*,*) 'DAME UN NUMERO'
READ(*,*) x
WRITE(*,*) 'LA MATRIZ ES'
!.............................
DO i=1,NF
 WRITE(*,*) (mat(i,j),j=1,NC)
END DO
!.............................
!WRITE(*,*) mat
!.............................
!filas: DO i=1,NF
!  columnas: DO j=1,NC
!    WRITE(*,*) mat(i,j)
```

```
!  END DO columnas
!END DO filas
!.........................
filas: DO i=1,NF
  columnas: DO j=1,NC
    coincidir: IF (x==mat(i,j)) THEN
      sw=1
      WRITE(*,*) 'ENCONTRADO EN FILA:',i,'COLUMNA:',j
    END IF coincidir
  END DO columnas
END DO filas
IF (sw==0) WRITE(*,*) 'NO EXISTE EL VALOR BUSCADO'
END PROGRAM cap4_10
```

– Para visualizar por pantalla el contenido de la matriz en la forma habitual hemos usado un bucle iterativo para el índice de las filas y un bucle implícito para el índice de las columnas.

– El algoritmo utilizado es el mismo que el del ejercicio CAP4_4 para un vector.

11. Calcular el vector suma de 3 vectores de 4 componentes conocidas y el módulo del vector suma, escribiendo los resultados por pantalla.

```
PROGRAM cap4_11
INTEGER:: j,k
INTEGER, PARAMETER:: NC=4
REAL :: modu
REAL,DIMENSION(NC):: v1=-1.,v2=2.,v3=-3.,s
DO j=1,NC
  s(j)=v1(j)+v2(j)+v3(j)
  WRITE(*,*) 'COMPONENTE',j,'DEL VECTOR SUMA:',s(j)
END DO
!$$$$$$ s=v1+v2+v3
!$$$$$$ WRITE(*,*) 'VECTOR SUMA:',s
modu=0
DO k=1,NC
  modu=modu+s(k)**2
END DO
modu=SQRT(modu)
```

```
WRITE(*,*) 'EL MODULO DEL VECTOR SUMA ES:',modu
END PROGRAM cap4_11
```

```
PROGRAM cap4_11
INTEGER, PARAMETER:: NC=4
REAL :: modu
REAL,DIMENSION(NC):: v1=-1.,v2=2.,v3=-3.,s
s=v1+v2+v3
WRITE(*,*) 'VECTOR SUMA:',s
modu=SQRT(SUM(s**2))
WRITE(*,*) 'EL MODULO DEL VECTOR SUMA ES:',modu
END PROGRAM cap4_11
```

– ¿Cómo cambia el programa si el número de componentes de los vectores es 5? Notar que, gracias al atributo PARAMETER usado para especificar la dimensión de los vectores en el programa, los cambios que hay que hacer en el código del mismo se reducen a sustituir 4 por 5 en esa sentencia.

– Para calcular el vector suma de un número grande de vectores, podemos seguir el mismo procedimiento que en este ejercicio. Sin embargo, es más práctico usar una matriz bidimensional en la que cada fila o cada columna de la misma sea un vector.

12. Calcular el vector suma de 3 vectores de 4 componentes conocidas y el módulo del vector suma, escribiendo los resultados por pantalla. Utilizar una matriz para almacenar los 3 vectores.

```
PROGRAM cap4_12
INTEGER:: i,j
INTEGER,PARAMETER::N=3,M=4
REAL :: modu
REAL,DIMENSION(N,M):: a
REAL,DIMENSION(M)::s
!NUMERO DE FILAS DE A ES Nº DE VECTORES A SUMAR
!NUMERO DE COLUMNAS DE A ES Nº DE ELEM. DE CADA VECTOR
filas: DO i=1,N
 columnas: DO j=1,M
  WRITE(*,*) 'INTRODUCE LA COMPONENTE',i,j,'DE LA MATRIZ A'
  READ(*,*)a(i,j)
 END DO columnas
```

```
END DO filas
DO i=1,M
 s(i)=0
END DO
columnas: DO  j=1,M
 filas: DO  i=1,N
  s(j)=s(j)+a(i,j)
 END DO filas
 WRITE(*,*) 'COMPONENTE',j,'DEL VECTOR SUMA:',s(j)
END DO columnas
sumcuad=0
DO  k=1,M
 sumcuad=sumcuad+s(k)**2
 WRITE(*,*) 'SUMA DE CUADRADOS DE COMP. DE S ES:',sumcuad
END DO
modu=SQRT(sumcuad)
WRITE(*,*) 'EL MODULO DEL VECTOR SUMA ES:',modu
END PROGRAM cap4_12
```

```
PROGRAM cap4_12
INTEGER:: j
INTEGER,PARAMETER::N=3,M=4
REAL :: modu
REAL,DIMENSION(N,M):: a
REAL,DIMENSION(M)::s
!NUMERO DE FILAS DE A ES Nº DE VECTORES A SUMAR
!NUMERO DE COLUMNAS DE A ES Nº ELEM. DE CADA VECTOR
WRITE(*,*) 'DAME LA MATRIZ POR FILAS'
READ(*,*) ((a(i,j),j=1,M),i=1,N)
columnas: DO i=1,M
 s(i)=SUM(a(1:N,i))
 WRITE(*,*) 'elemento',i,'del vector suma:',s(i)
END DO columnas
modu=SQRT(SUM(s**2))
WRITE(*,*) 'EL MODULO DEL VECTOR SUMA ES:',modu
END PROGRAM cap4_12
```

– Notar que el vector suma juega el papel de un acumulador, de ahí que sea necesario inicializarlo antes de que aparezca en la sentencia de asignación en el siguiente bucle.

– Recordar que en Fortran el orden de cierre de los bucles es inverso al orden de apertura de los mismos.

13. Calcular la suma de dos matrices de 3 filas y 2 columnas (3X2) y escribir el resultado por pantalla.

```
PROGRAM cap4_13
INTEGER:: i,j
INTEGER,PARAMETER::N=3,M=2
REAL,DIMENSION(N,M):: mat1,mat2,suma
!***LECTURA DE LAS MATRICES, POR FILAS
filas: DO  i=1,N
 columnas: DO  j=1,M
  WRITE(*,*) 'ELEMENTO',i,j,' DE MAT1 Y MAT2'
  READ(*,*) mat1(i,j),mat2(i,j)
 END DO columnas
END DO filas
!***CALCULO DE LA MATRIZ SUMA, POR FILAS
DO  i=1,N
 DO  j=1,M
  suma(i,j)=mat1(i,j)+mat2(i,j)
 END DO
END DO
!$$$$$$ suma=mat1+mat2 !operando con las matrices completas
!***VISUALIZACION DE LA MATRIZ SUMA, POR FILAS
PRINT*
WRITE(*,*) 'MATRIZ SUMA'
PRINT*
DO  i=1,N
 WRITE(*,*) (suma(i,j),j=1,M)
END DO
END PROGRAM cap4_13
```

```
PROGRAM cap4_13
IMPLICIT NONE
INTEGER, PARAMETER:: NF=3,NC=2,NM=2
```

```
REAL, DIMENSION(NF,NC,NM) :: x
REAL, DIMENSION(NF,NC):: suma=0
INTEGER:: i,j,k
!***LECTURA DE LAS MATRICES, POR FILAS
matriz: DO  k=1,NM
 filas: DO  i=1,NF
  columnas: DO  j=1,NC
   WRITE(*,*) 'ELEMENTO',i,j,' DE MATRIZ',k
   READ(*,*) x(i,j,k)
  END DO columnas
 END DO filas
END DO matriz
!***CALCULO DE LA MATRIZ SUMA
filas: DO  i=1,NF
 columnas: DO  j=1,NC
  matriz: DO  k=1,NM
   suma(i,j)=suma(i,j)+x(i,j,k)
  END DO matriz
 END DO columnas
END DO filas
!***VISUALIZACION DE ELEMENTOS DE LA MATRIZ SUMA, POR FILAS
PRINT*
PRINT*,'MATRIZ SUMA'
PRINT*
DO  i=1,NF
 WRITE(*,*) (suma(i,j),j=1,NC)
END DO
END PROGRAM cap4_13
```

– Para evitar corregir el atributo PARAMETER cada vez que cambien las dimensiones de las matrices, teniendo que compilar el nuevo programa, un truco puede ser reservar más espacio en memoria del necesario habitualmente.

– En este ejercicio, tanto la lectura de las matrices, como el cálculo de la matriz suma se ha realizado por filas. Repite el ejercicio trabajando por columnas.

– Para sumar un número grande de matrices bidimensionales podríamos usar el mismo procedimiento que en este ejercicio. Sin embargo, es más práctico usar un array tridimensional en el que la tercera dimensión se corresponda con el número de matrices bidimensionales a sumar. Por ejemplo, para sumar 15 matrices de

3X2, declarar un array de dimensiones (3,2,15). Gráficamente, equivale a imaginar 15 planos en cada uno de los cuales tenemos una matriz de 3X2. Repite la segunda versión del ejercicio trabajando con secciones de arrays.

14. Calcular el producto de dos matrices de N x M y M x L y escribir la matriz producto por pantalla. Las matrices son: $MAT1 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$ y

$$MAT2 = \begin{pmatrix} -1 & -4 \\ -2 & -5 \\ -3 & -6 \end{pmatrix}$$

```
PROGRAM cap4_14
INTEGER:: i,j,k
INTEGER,PARAMETER::N=2,M=3,L=2
REAL,DIMENSION(N,M)::mat1
REAL,DIMENSION(M,L)::mat2
REAL,DIMENSION(N,L)::prod
mat1=RESHAPE((/1,1,2,2,3,3/),(/N,M/))
mat2=RESHAPE((/-1,-2,-3,-4,-5,-6/),(/M,L/))
!***CALCULO DE LA MATRIZ PRODUCTO ***
DO  i=1,N
 DO  j=1,L
  prod(i,j)=0
  DO  k=1,M
   prod(i,j)=prod(i,j)+mat1(i,k)*mat2(k,j)
  END DO
 END DO
END DO
!$$$$$$ prod=matmul(mat1,mat2) !funcion intrinseca transformacional
!***VISUAL. DE MATRIZ PRODUCTO,POR FILAS
WRITE(*,*)
WRITE(*,*) 'MATRIZ PRODUCTO'
WRITE(*,*)
DO  i=1,N
 WRITE(*,*) (prod(i,j),j=1,L)
END DO
END PROGRAM cap4_14
```

– Recuerda que la inicialización de matrices se hace por columnas por defecto.

– Notar que la matriz PROD juega el papel de un acumulador, puesto que cada elemento se calcula como una suma de productos. Por esta razón, es imprescindible inicializar esta matriz en el bucle con índice j antes de entrar al bucle con índice k.

15. Calcular la media de NUM números. Usar un vector para almacenarlos, el cual se dimensiona dinámicamente en tiempo de ejecución.

```
PROGRAM cap4_15
INTEGER, ALLOCATABLE, DIMENSION(:) :: vector
INTEGER :: error
REAL:: media
WRITE(*,*) 'DAME NUMERO DE ELEMENTOS DEL VECTOR'
READ(*,*) num
ALLOCATE (vector(1:num),STAT=error)
IF (error /= 0) THEN
  WRITE(*,*) 'NO SUFICIENTE ESPACIO MEMORIA'
  STOP
END IF
WRITE(*,*) 'DAME EL VECTOR'
READ(*,*) vector
media=SUM(vector)/REAL(num)
WRITE(*,*) 'LA MEDIA ES',media
DEALLOCATE (vector,STAT=error)
IF (error /= 0) THEN
  WRITE(*,*) 'ERROR AL LIBERAR LA MEMORIA'
  STOP
END IF
END PROGRAM cap4_15
```

– ALLOCATABLE es el atributo que declara que el array VECTOR será dimensionado dinámicamente. Se trata de un array unidimensional puesto que sólo aparece una vez el carácter ":".

– ALLOCATE dimensiona dinámicamente el array VECTOR, previamente declarado con el atributo ALLOCATABLE. Establece los límites inferior y superior de su dimensión.

– DEALLOCATE libera la memoria para el array VECTOR, reservada previamente en la sentencia ALLOCATE.

EJERCICIOS PROPUESTOS

1) Programa que pida el grado de un polinomio (inferior a diez), sus coeficientes y un valor de x para evaluarlo.

2) Programa que pida los grados de dos polinomios (inferior a diez y no tienen porqué ser iguales), sus coeficientes y muestre por pantalla el polinomio suma.

3) Programa que pida los grados de dos polinomios (inferior a diez y no tienen porqué ser iguales), sus coeficientes y muestre el polinomio producto.

4) Programa que pida coordenadas cartesianas de los extremos de dos vectores, escriba el ángulo que forman (en grados) y su producto escalar utilizando el coseno del ángulo comprendido entre ambos.

5) Programa que pida coordenadas cartesianas de los extremos de dos vectores, escriba por pantalla el ángulo que forman (en grados) y el módulo de su producto vectorial utilizando el seno del ángulo comprendido entre ambos.

6) Programa que pida las temperaturas de los días de la semana. Posteriormente escribirá los números de los días que superen la temperatura media.

7) Programa que calcule el máximo y el mínimo de un vector de números de dimensión variable, como mucho 10.

8) Programa que lea un número natural por teclado, almacenando cada dígito del mismo en un elemento de un vector y compruebe si es *narcisista*. A saber: un número de n dígitos es narcisista si coincide con la suma de las potencias de orden n de sus dígitos. Ejemplo: 153, pues $1^3+5^3+3^3= 1+125+27=153$. Los números 370 y 371 son también narcisistas. ¿Cómo buscarías todos los números narcisistas de dos y de 4 dígitos?

9) Programa que lea por teclado una matriz 3x3 y muestre su matriz traspuesta.

10) Programa que obtenga la recta de regresión $y=mx+b$ de n parejas de valores (x,y) y el coeficiente de correlación lineal r.

$$m=\frac{n\sum xy-\sum x\sum y}{n\sum x^2-(\sum x)^2}\quad b=\frac{\sum y\sum x^2-\sum x\sum xy}{n\sum x^2-(\sum x)^2}$$

$$r=\frac{n\sum xy-\sum x\sum y}{\sqrt{(n\sum x^2-(\sum x)^2)(n\sum y^2-(\sum y)^2)}}$$

11) Programa que lea por teclado una matriz M(3x3) y muestre las matrices M+M y MxM.

12) Programa que calcule el máximo de cada columna par y el mínimo de cada columna impar de una matriz de 5 filas por 6 columnas.

13) Programa que compruebe si una matriz A es idempotente de grado 2, es decir, si $A=A^2$.

14) Programa que calcule la traza de una matriz cuadrada.

15) Programa que compruebe si la columna de una matriz está llena de ceros.

16) Programa que lea la matriz siguiente y compruebe si es un *cuadrado mágico especular*. A saber: una matriz es un cuadrado mágico si las sumas de sus filas, columnas y diagonales son el mismo número. Además, el cuadrado mágico es especular si al invertir los dígitos de cada número (elemento de la matriz) resulta otro cuadrado mágico.

$$\begin{pmatrix} 96 & 64 & 37 & 45 \\ 39 & 43 & 98 & 62 \\ 84 & 76 & 25 & 57 \\ 23 & 59 & 82 & 78 \end{pmatrix}$$