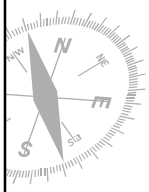
Funciones intrínsecas con arrays

Seminario de computación 2009



ALL(MASK)

Esta función es una función Lógica que opera sobre variables de tipo lógicas. Mask es un array lógico. ALL es TRUE si todos los elementos de MASK son TRUE. De lo contrario daría FALSE.

EJ:

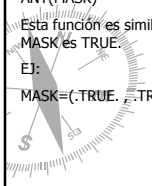
MASK=(.TRUE. , .TRUE. , .FALSE.) => el resultado de ALL(MASK) es .FALSE.

ANY(MASK)

Esta función es similar a ALL, solo que es TRUE cuando alguno de los elementos de MASK es TRUE.

EJ:

MASK=(.TRUE. , .TRUE. , .FALSE.) => el resultado de ANY(MASK) es .TRUE.



COUNT(MASK,dim)

Esta variable intrínseca toma el array lógico MASK y cuenta la cantidad de elementos .TRUE. que hay a lo largo de la dimensión dim. Si dim no está especificado cuenta por columnas. El resultado es un array que tiene la misma forma (shape) que la máscara, pero donde la dimensión dim no está presente.

EJ: MASK=(.TRUE.,.FALSE.,.TRUE.)

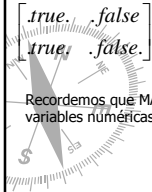
COUNT(MASK) => esto sería 2.

EJ:

$$\begin{bmatrix} .true. & .false \\ .true. & .false. \end{bmatrix}$$

En este caso COUNT(MASK,1) => (2,0)

Y COUNT(MASK,2) => (1,1)



Recordemos que MASK puede ser el resultado de una operación lógica aplicada a variables numéricas.

DOT_PRODUCT(vectorA,vectorB)

Esta función intrínseca opera con arrays de 1 dimensión tanto REAL como INTEGER que tengan la misma cantidad de elementos. El resultado es el producto interno entre ambos vectores.

EJ: A=(/1,2/) , B=(/2,3/) => DOT_PROD(A,B) sería 8.

MATMUL(matrizA,matrizB)

Esta función calcula el producto matricial entre la matrizA y la matrizB. Para que sea posible calcular dicho producto el número de columnas de matrizA debe ser igual al número de filas de matrizB. El resultado del producto matricial es una matriz que tiene en el elemento I,J el producto interno de la fila I de la matrizA por la columna J de la matrizB.

$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ matrizA $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ matrizB, => MATMUL(matrizA,matrizB) es igual a la matrizB

MAXLOC(ARRAY,dim,MASK)

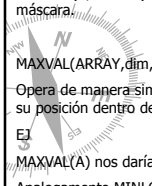
El argumento dim es opcional. Esta función devuelve la ubicación del máximo en ARRAY a lo largo de la dimensión especificada por dim. Si hay varios puntos donde se alcanza el máximo, se indica la ocurrencia del primero de ellos.

EJ:

A=(/1,3,2,5,2,1/) => MAXLOC(A) es 4 dado que es la posición donde ocurre el máximo.

Si utilizamos una máscara por ejemplo para no considerar todos aquellos valores mayores que 4 entonces:

MAXLOC(A,MASK=(A <= 4)) entonces el resultado es 2 porque 5 queda excluido por la máscara.



MAXVAL(ARRAY,dim,MASK)

Opera de manera similar a MAXLOC pero nos devuelve el valor del máximo en lugar de su posición dentro del array.

EJ

MAXVAL(A) nos daría 5 si A es el del ejemplo anterior.

Analogamente MINLOC y MINVAL operan para encontrar los mínimos.

PRODUCT(ARRAY,dim,MASK)

Calcula el producto de todos los elemntos de un ARRAY a lo largo de la dimensión dim (1 si no está presente) y sujeto a la condición expresada en la máscara MASK (que también es opcional).

EJ:

A=(/1,2,3,4/) => PRODUCT(A)=24

Analogamente la función SUM(ARRAY,dim,MASK) calcula la suma de todos los elementos del ARRAY a lo largo de la dimensión dim sujeto a la máscara MASK.

SUM(A) daría 10.

TRANSPOSE(ARRAY)

Función que solo opera sobre ARRAYS de 2 dimensiones y calcula el transpuesto del ARRAY.

$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ matrizA => TRANSPOSE(matrizA) sería $\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$

RESHAPE(SOURCE,SHAPE)

Crea un array cuyo tamaño está especificado por el vector INTEGER SHAPE y lo llena tomando elementos del array SOURCE.

EJ:

A=(/1,3,2,4/) B=(/2,2/) , => RESHAPE(A,B) nos da $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

La cantidad de elementos de A debe ser igual a la cantidad de elementos definidos por el array B.

Si A no tiene la cantidad de elementos suficientes, entonces se puede agregar un argumento más que indique el / los valores a utilizar en el caso que hubiera que rellenar más elementos.

EJ: Supongamos A el del ejemplo anterior.

B=(/2,3/) , PAD=0 => RESHAPE(A,B,PAD) nos da $\begin{bmatrix} 1 & 2 & 0 \\ 3 & 4 & 0 \end{bmatrix}$

También es posible indicarle a la función RESHAPE el orden en el cual se va "llenando" el nuevo array, el orden por defecto es el orden por columnas en el caso de los array de 2 dimensiones.

SPREAD(SOURCE,DIM,NCOPIES)

Esta función crea un nuevo array haciendo NCOPIES copias del array SOURCE a lo largo de la dimensión DIM.

EJ:

A=(/1,2,3,4,5/) => SPREAD(A,1,5) $\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix}$

Por otro lado SPRAD(A,2,5) nos da $\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 \end{bmatrix}$

Esta función puede ser útil cuando queremos restar arrays que tienen formas diferentes. Podemos generar arrays de tamaños equivalentes repitiendo varias veces el array de menor tamaño.

PACK(ARRAY,MASK)

Esta función requiere dos argumentos, un array y una máscara (variable o expresión lógica) que . El resultado de la función es un vector con todos los elementos de ARRAY que se corresponden con elementos TRUE de MASK.

EJ:

A= $\begin{bmatrix} 1 & -3 \\ 4 & -4 \end{bmatrix}$ y MASK= $\begin{bmatrix} .false. & .true. \\ .true. & .true. \end{bmatrix}$

El resultado de la funcíon en este caso es : (4 -3 -2)

El resultado de PACK siempre es un vector sin importar cuantas dimensiones tenga A.

Existe una función UNPACK que realiza la operación inversa.

UNPACK(VECTOR,MASK) Dado un vector VECTOR, la función UNPACK crea un ARRAY con la forma del array lógico MASK. Luego asigna el primer elemento de VECTOR al primer elemento donde MASK es .TRUE. y sigue la asignación en orden (es decir siguiendo el orden con el que fortran almacena los arrays en la memoria).

Si VECTOR es un vector obtenido como resultado de utilizar PACK entonces UNPACK debería colocar los elementos de VECTOR en las ubicaciones de donde vinieron origianlmente

MERGE(TSOURCE,FSOURCE,MASK)

Esta función intrínseca genera un ARRAY del mismo tipo que el array TSOURCE. Basicamente permite combinar dos arrays sujetos a la condición lógica MASK.

(TSOURCE, FSOURCE y MASK deben tener la misma forma).

El array final tiene la misma forma que los 3 argumentos de entrada.

En aquellos elementos donde MASK es TRUE, el array se rellena con los elementos de TSOURCE, donde es FALSE con los elementos de FSOURCE.

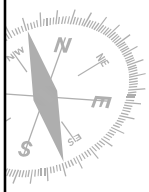
EJ:

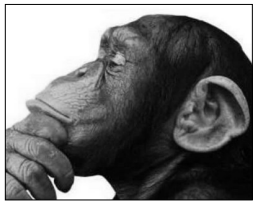
A= $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ B= $\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$ MASK= $\begin{bmatrix} .false. & .true. \\ .true. & .true. \end{bmatrix}$

Entonces MERGE(A,B,MASK) nos daría $\begin{bmatrix} 5 & 2 \\ 3 & 4 \end{bmatrix}$

Ejercicio:

Se cuentan con datos horarios de temperatura en una estación ubicada en el Alto Valle de Río Negro. Sobre dichos datos se desea saber cuantas horas estuvo la temperatura por debajo de 3ºC. Por otro lado se desea calcular un índice (inventado) que mide la severidad de la helada que consiste en sumar sobre todas las horas cuantos grados por debajo de 3º estuvo la temperatura. Si fue 3 o superior se suma 0.

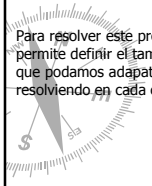




ALLOCATABLE ARRAYS

Hasta ahora el tamaño de los arrays que hemos utilizado fue declarado junto con el nombre de la variable. Esto se denomina STATIC MEMORY ALLOCATION. El espacio en memoria para almacenar el contenido del array es siempre el mismo a lo largo de todo el programa.

En algunos casos, esto puede resultar inconveniente (sobre todo si para algunos de los arrays que intervienen en nuestro programa no conocemos a priori el tamaño que van a tener). Para salvar este problema generalmente se definen arrays muy grandes que eventualmente permitan ajustarse a muchas situaciones, pero esto es ineficiente porque estamos usando mucha memoria para casos en los que no utilizamos el tamaño completo de las variables.



Para resolver este problema, se introdujo la DYNAMIC MEMORY ALLOCATION que permite definir el tamaño de un array durante la ejecución del programa. Esto hace que podamos adapatar los tamaños de los arrays al problema específico que estamos resolviendo en cada caso.

Como permitir que el tamaño de un array sea asignado durante la ejecución del programa: para eso utilizamos el atributo ALLOCATABLE en la declaración de las variables que queremos que tengan esta propiedad.

EJ:

```
REAL, ALLOCATABLE, DIMENSION(:)  :: X
```

Este atributo le dice al programa que el tamaño de X no será determinado al momento de la declaración sinó más adelante. Si bien no hay que especificar el tamaño, si las dimensiones, esto lo hacemos colocando un : , en este caso para indicar que tiene una sola dimensión.

```
REAL, ALLOCATABLE, DIMENSION(:,:)  :: X !En este otro caso X tiene 2 dimensiones.
```

Hasta este punto X es solo un nombre y no tiene asignado espacio en la memoria. Para lograr asignarle un espacio en la memoria y poder utilizar a la variable X disponemos de la sentencia ALLOCATE.

```
ALLOCATE ( X(10,20))
```

Esta sentencia le indica al programa que reserve un espacio de 10 filas por 20 columnas en la memoria para la variable X. A partir de este momento podemos utilizar la variable X como lo hacíamos antes.

```
ALLOCATE(X(N,M)) !Esto también es válido si N y M son 2 INTEGER.
```

Si queremos liberar el espacio que está ocupado por la variable X en la memoria podemos utilizar el comando DEALLOCATE

```
DEALLOCATE ( X )
```

Es importante notar que después de haber hecho esto no podremos acceder más a la variable X, su contenido se ha perdido.

EJ: Tipear y compilar el siguiente programa, ejecutarlo mirando como varía la memoria utilizada por el sistema con el monitor del sistema de windows.

```
PROGRAM pepe
REAL, ALLOCATABLE, DIMENSION(:,:) :: X
WRITE(*,*)"Antes de asignar a X espacio en memoria, presione enter"
READ(*,*)
ALLOCATE(X(50000,1000))
WRITE(*,*)"X ya está en la memoria de la máquina, presione enter"
READ(*,*)
DEALLOCATE(X)
WRITE(*,*)"X fue eliminado de la memoria de la máquina"
STOP
END PROGRAM pepe
```