PRACTICA 1d

Elementos de programación Fortran: Arreglos.

Datos compuestos indexados: arreglos. Los objetos matemáticos como *vectores*, *matrices* (y con más generalidad, *tensores*) pueden ser almacenados en Fortran en *arreglos* unidimensionales, bidimensionales (y multidimensionales), respectivamente. Un arreglo (*array* en inglés) es una estructura de datos *compuesta* formada por una colección ordenada de elementos del mismo tipo de datos (por lo que se dice que la estructura de datos es *homogénea*) y tal que los elementos se pueden acceder en cualquier orden simplemente indicando la posición que ocupa dentro de la estructura (por lo que se dice que es una estructura *indexada*). Un arreglo, como un todo, se identifica en el programa con un *nombre*, y un elemento particular del mismo es accedido a partir de cierto conjunto de *índices*, los cuales son, en Fortran, de tipo entero. La *dimensión* del arreglo es el número de índices necesarios para especificar un elemento, y su *tamaño* es el número total de elementos que contiene. Antes de ser usado, un arreglo debe ser declarado especificando el tipo de elementos que contendrá y el *rango* de valores que puede tomar cada índice (lo cual define la cantidad de elementos que lo componen). Así, un arreglo unidimensional `V` que contendrá 3 elementos reales y un arreglo bidimensional `A` que contendrá $5 \times 4 = 20$ elementos reales son declarados con la sentencia de declaración de tipo:

```
REAL :: V(3), A(5,4)
```

En el programa, un elemento particular de un arreglo es accedido a través del nombre del arreglo junto con los valores de los índices que especifican su posición. Así `V(1)` indica el primer elemento del arreglo unidimensional `V`, en tanto que `A(2,3)` corresponde al elemento en la fila 2 y la columna 3 de la matriz almacenada en el arreglo bidimensional `A`. Nótese que cada elemento de un arreglo constituye de por sí una variable, la cual puede utilizarse de la misma forma que cualquier variable del mismo tipo es utilizada.

☞ Al igual que las variables ordinarias, los valores de un arreglo debe ser inicializados antes de ser utilizados. Un arreglo puede ser inicializado en *tiempo de ejecución* a través de sentencias de asignación ya sea elemento a elemento a través de bucles `DO`, como ser, por ejemplo:

```
DO i=1,3
   V(i) = REAL(i)
ENDDO

DO i=1,5
   DO j=1,4
      A(i,j) = REAL(i+j)
   ENDDO
ENDDO
```

o bien a través del *constructor* de arreglos `[ ]` implementado en Fortran 2003:

```
V = [1.0,2.0,3.0]
A = RESHAPE([2.0,3.0,4.0,5.0,6.0, &
             3.0,4.0,5.0,6.0,7.0, &
             4.0,5.0,6.0,7.0,8.0, &
             5.0,6.0,7.0,8.0,9.0],[5,4])
```

donde para el arreglo bidimensional se utiliza la función intrínsica `RESHAPE` para construirlo a partir de un arreglo unidimensional cuyas componentes resultan de ordenar *por columna* los valores iniciales de la matriz. Estas construcciones permiten el uso de bucles `DO` implícitos lo cual es especialmente útil para inicializaciones más complicadas o arreglos de muchos elementos:

```
V = [(REAL(i), i=1,3)]
A = RESHAPE([((REAL(i+j),i=1,5),j=1,4)], &
            [5,4])
```

También es posible inicializar todos los elementos de un arreglo a un mismo y único valor con las sentencias:

```
V = 0.0
A = 0.0
```

La inicialización puede realizarse también en *tiempo de compilación* realizando la asignación en la sentencia de declaración de tipo:

```
REAL :: V(3) = [1.0,2.0,3,0]
REAL :: A(5,4) = 0.0
```

Por supuesto, la inicialización también puede realizarse leyendo los datos del arreglo con sentencias `READ`, lo cual nos lleva a la siguiente nota.

☞ La forma más flexible de leer por teclado (o, con el número de unidad apropiado, de un archivo) n elementos de un vector para almacenarlo en un arreglo unidimensional `V` consiste en utilizar un bucle `DO` *implícito*:

```
READ(*,*) (V(i), i = 1, n)
```

el cual, al involucrar sólo una sentencia de lectura, permite que los elementos del arreglo no necesariamente estén cada uno en una línea, sino que de hecho pueden estar dispuesto de cualquier manera (inclusive todos en la misma línea, o uno por línea, o varios en una línea y el resto en otra, etc.). De manera análoga, una matriz de $n \times m$ puede ser ingresada en la forma usual (esto es, fila por fila) para ser almacenada en un arreglo bidimensional `A` utilizando dos bucles `DO` implícitos:

```
READ(*,*) (A(i,j), j = 1, m), i = 1, n)
```

Por otra parte, para el vector el remplazo de la sentencia `READ` por `WRITE` permite escribir los n elementos del vector sobre una única línea. Para imprimir una matriz en la forma usual (esto es, fila por fila), en cambio, recurrimos a la combinación de un bucle `DO` usual y uno implícito:

```
DO i = 1, n
   WRITE(*,*) (A(i,j), j = 1, m)
ENDDO
```

☞ Los arreglos pueden ser utilizados *como un todo* en sentencias de asignación y operaciones aritméticas siempre y cuando todos los arreglos involucrados tengan la misma *forma*, esto es, el mismo número de dimensiones y el

mismo número de elementos en cada dimensión. Si esto es así, las correspondientes operaciones serán realizadas elemento a elemento. Por ejemplo, si `X`, `Y`, `V` son arreglos unidimensionales de 3 elementos, la sentencia:

```
V = 2.0*X + Y
```

permite asignar a `V` la combinación lineal resultante de sumar el doble de cada elemento de `X` con el correspondiente elemento de `Y`. Más aún, ciertas funciones intrínsecas de Fortran que son usadas con valores escalares también aceptarán arreglos como argumentos y devolverán un arreglo cuyo resultado será su acción elemento a elemento del arreglo de entrada. Tales funciones son conocidas como *funciones intrínsecas elementales*. Por ejemplo, para nuestro arreglo `V` inicializado como:

```
V = [-3.0,2,0,-1.0]
```

la función `ABS(V)` devolverá el arreglo `[3.0,2.0,1.0]`.

☞ Además de poder utilizar los elementos de un arreglo o, bajo las circunstancias indicadas, el arreglo como un todo en una sentencia, es posible utlizar secciones de los mismos. Por ejemplo, para nuestros arreglos `V` y `A`:

a) `V(1:2)` ó `V(:2)` es el subarreglo de `V` de elementos `V(1)` y `V(2)`,
b) `V(1:3:2)` ó `V(::2)` es el subarreglo de `V` de elementos con índice impar: `V(1)` y `V(3)`,
c) `A(1:5,2)` ó `A(:,2)` es la segunda columna de la matriz contenida en el arreglo `A`,
d) `A(3,1:4)` ó `A(3,:)` es la tercer fila de la matriz contenida en `A`.

☞ **Asignacion estática y dinámica de memoria.** En la asignación de tipo dada para los arreglos `V` y `A` sus tamaños ha sido predefinidos en tiempo de compilación y por lo tanto no varían durante la ejecución del programa. Se dice, por lo tanto, que la asignación de memoria para los arreglos es *estática*. Nótese que si el tamaño de los arreglos con los que se trabajará no es conocido *a priori*, esto obliga a definir los arreglos de tamaños suficientemente grande como para contenerlos. La manera más flexible de proceder consiste en asignar el rango de sus índices a través de parámetros enteros. Así el siguiente fragmento de código define un arreglo unidimensional `V` define un arreglo unidimensional `V` que puede contener un vector de a lo más `NMAX` elementos reales, y un arreglo bidimensional `A` que puede contener una matriz real de a lo más `NMAX` filas y `MMAX` columnas, donde fijamos, para nuestros propósitos $\texttt{NMAX} = \texttt{MMAX} = 6$.

```
INTEGER, PARAMETER :: NMAX = 6
INTEGER, PARAMETER :: MMAX = 6
REAL :: V(NMAX), A(NMAX,MMAX)
```

De este modo el vector de 3 elementos está contenido en el subarreglo `V(1:3)` de `V`, mientras que la matriz de 5×4 está almacenada en el subarreglo `A(1:5,1:4)`. Si, posteriormente, el problema requiere de arreglos de mayor tamaño que los máximos asignados, simplemente ajustamos los valores de los parámetros y recompilamos el programa.

Como una alternativa más eficiente en el uso de la memoria disponible en el sistema, Fortran 90 dispone también de un esquema conocido como *asignación dinámica de memoria*, donde el tamaño de los arreglos son asignadas durante la *ejecución* del programa. Para utilizar esta característica los arreglos deben ser declarados con el atributo `ALLOCATABLE`. En nuestro ejemplo:

```
REAL, ALLOCATABLE :: V(:), A(:,:)
```

Esta instrucción indica al compilador que los objetos `V` y `A` son un arreglo unidimensional y bidimiensional, respectivamente, cuyos tamaños serán especificados *cuando el programa esté en ejecución*. Esto es efectuado a través de la sentencia `ALLOCATE`, como ser, para nuestro ejemplo:

```
ALLOCATE(V(3), STAT = ierr)
ALLOCATE(A(5,4), STAT = ierr)
```

Si la variable entera `ierr = 0` la asignación se ha realizado sin problemas y a partir de este momento podemos trabajar con los mismos. Si, en cambio, el valor devuelto en `ierr` es no nulo, ha ocurrido un error en la asignación de memoria con lo que debemos proceder a tratar esta situación de error (usualmente abortamos el programa). Una vez que los arreglos asignados no sean más necesarios, debemos *liberar* la memoria utilizada con la sentencia `DEALLOCATE`:

```
DEALLOCATE(V)
DEALLOCATE(A)
```

☞ Un error muy común al trabajar con arreglos es hacer que el programa intente acceder a elementos que están fuera de los límites del mismo. Es *nuestra* responsabilidad asegurarnos que ésto no ocurra ya que tales errores *no* son detectados en tiempo de compilación. Un error de este tipo puede producir, durante la ejecución del programa, que el mismo se aborte con una condición de error denominada *violación de segmento* (*segmentation fault* en inglés) o bien que acceda a la posición de memoria que correspondería al elemento como si éste estuviera asignado. En este último caso, el programa no abortará pero, puesto que tal posición de memoria puede estar asignada para otro propósito, el programa conducirá a resultados erróneos.

> ✏ **Violación de segmento.**
> La violación de segmento (*segmentation fault*) es una condición de error que ocurre cuando un programa intenta acceder a una posición de memoria que no le corresponde o cuando intenta acceder en una manera no permitida (por ejemplo, cuando se trata de escribir una posición de memoria que es solo lectura). Cuando ésta situación se produce el sistema operativo aborta el programa. Esto garantiza que ningún programa incorrecto (o malicioso) puede sobreescribir la memoria de otro programa en ejecución o del sistema operativo.

El compilador `gfortan` permite detectar, *en tiempo de ejecución*, el intento de acceso fuera de límites de un arreglo si el código es compilado con la opción `-fbounds-check`. Sin embargo, el ejecutable originado correrá mucho más lento ya que la tal comprobación demanda mucho tiempo de cómputo. Por éste motivo *esta opción sólo debe utilizarse durante la fase de prueba del programa* con el propósito de detectar posibles errores.

Ejercicio 1. Sean $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ y $\mathbf{y} = (y_1, y_2, \ldots, y_n)$ $n-$uplas de elementos reales, el *producto escalar* de $\mathbf{x}$ e $\mathbf{y}$ es el número real no negativo

definido por la ecuación

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i}^{n} x_i y_i.$$

Así, por ejemplo,

$$(5, -3, 2) \cdot (2, 3, 4) = 9.$$

a) Implementar un programa Fortran para determinar el producto escalar de dos vectores de $\mathbb{R}^n$.
b) Utilizar la función intrínseca que proporciona Fortran para el producto escalar, `DOT_PRODUCT(x,y)`.
c) La *potencia mecánica* P aplicada sobre un sólido rígido viene dado por el producto escalar de la fuerza aplicada $\mathbf{F}$ por la velocidad $\mathbf{v}$ del mismo:

$$P = \mathbf{F} \cdot \mathbf{v}.$$

Si la fuerza es medida en newtons y la velocidad en metros por segundo, la potencia es medida en watts. Utilice los resultados anteriores para calcular la potencia suministrada por una fuerza $\mathbf{F} = (4, 3, -2)$ newtons a un cuerpo moviéndose a una velocidad $\mathbf{v} = (4, -2, 1)$ metros por segundo.

Ejercicio 2. Sea A una matriz de $n \times p$ y B una matriz de $p \times m$ elementos reales, la *multiplicación matricial* de A por B es la matriz C de $n \times m$ elementos c_{ij} dados por:

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}$$

Así, por ejemplo,

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} 5 & 4 \\ 15 & 14 \end{bmatrix}.$$

a) Implementar un programa Fortran para la multiplicación de dos matrices de acuerdo con la definición anterior.
b) Simplificar el algoritmo notando que el elemento c_{ij} puede considerarse como el producto interno de la fila i-ésima de A por la columna j-ésima de B y utilizar la función intrínsica `DOT_PRODUCT(x,y)`.
c) Utilizar la función intrínsica `MATMUL(A,B)` que Fortran posee para la multiplicación matricial.

Ejercicio 3. Realizar un programa que lea una matriz cuadrada desde un archivo y calcule su traza, es decir la suma de los elementos de la diagonal principal. Además el programa debe calcular la suma de los elementos de la otra diagonal. La dimensión de la matriz debe leerse también desde el archivo.

Ejercicio 4. Realizar un programa que para una dada matriz permute la primera fila con la última, la segunda con la penúltima, y así sucesivamente. Testear el programa utilizando la siguiente matriz,

$$\begin{bmatrix} 1.5 & 3.4 & -5.2 & 0.5 \\ 0.3 & -9.3 & 3.2 & 5.6 \\ 2.3 & 0.0 & 4.1 & 0.2 \\ -2.0 & 1.1 & 7.4 & 3.8 \\ -8.3 & -2.5 & 3.6 & 0.4 \end{bmatrix}$$

Ejercicio 5. A partir del vector dado más abajo, realizar un programa que indique cuales son los valores máximo y mínimo de sus componentes, indicando que posición ocupan en el vector, sin utilizar funciones intrínsicas. Además, calcular la suma de todos los elementos del vector.

$\mathbf{v}$ = (-12.32, 10.27, -5.23, -0.01, 0.00, 7.32, 7.33, 8.96, -2.23, 11.02, -4.36, 0.02, -0.09, -1.00, 7.00, -1.32, 0.75, 3.47, -0.23, 1.35, 5.23, 9.73, -2.36, -2.89, 10.39, 3.14, 2.78, -5.55, -10.00).

Ejercicio 6. Modificar el programa anterior para realizar los cálculos utilizando las funciones intrínsicas `MAXVAL(A)`, `MINVAL(A)`, `MAXLOC(A)`, `MINLOC(A)` y `SUM(A)`.

Ejercicio 7. Utilizando la estructura `WHERE`, `ELSEWHERE`, `ENDWHERE`, a partir del vector $\mathbf{v}$ anterior, construir un vector $\mathbf{w}$, donde su i-ésima componente sea -1, si la i-ésima componente de $\mathbf{v}$ es negativa, y 1 si es positiva.

Ejercicio 8. Realizar un programa que permita ordenar las componentes de un vector de menor a mayor. Para probar el programa, utilizar el vector de los ejercicios anteriores.