Departamento de Matemática Aplicada

Facultade de Matemáticas
Campus Vida
Rúa Lope Gómez de Marzoa s/n
15782 Santiago de Compostela

Análise Numérica Matricial– Curso 2018-19
Práctica 1: Repaso de Fortran - Operaciones con vectores y matrices

1 Repaso de FORTRAN 90

Nota.- *Seguiremos las notas de Fortran de* **Francisco Pena, Carmen Rodríguez y/o Juan Touriño** *(los tres subidos como material de apoyo en el Campus Virtual) y el libro* **M. Metcalf-J. Reid-M. Cohen:** *Fortran 95/2003 explained.* Oxford. 2004.

1.1 Tipos de datos y declaración de variables

1.2 Arreglos (arrays)

- **Declaración e inicialización, subarrays, rango(rank), dimensión, extensión en cada dimensión.**
- **Tipos de declaración**
  - **Declaración explícita (explicit-shape array)**
    Como primer ejemplo, para declarar un arreglo de números reales llamado `ar`, de rango 3, con extensiones 4, 8 y 5, añadimos el atributo `dimension` a la sentencia de declaración de tipo:

```
program principal

implicit none
...
real,dimension(4,8,5)::ar
...
end program principal
```

    También son de este tipo aquellos arreglos argumento cuyos límites se establecen en el momento de entrar en un procedimiento (por ejemplo, a través de otros argumentos); vemos un ejemplo:

```
subroutine eje(m,n,x,y)

implicit none
integer,intent(in)::m,n
real,intent(inout)::x(m,n),y(m,2*n) ...
```

  - **Declaración automática y asumida (assumed-shape array)**
    Un procedimiento con argumentos ficticios que son arreglos cuyo tamaño varía de una llamada a otra, puede tener necesidad de arreglos locales cuyo tamaño también varía; son los **arreglos automáticos** (automatic arrays), que se crean cuando se llama al subprograma y desaparecen cuando se sale de él; en el siguiente ejemplo se crea el espacio `aux` con la misma forma que `a`–ver la sección siguiente para la función `SIZE`–:

```
subroutine intercambio(a,b)

implicit none
real,dimension(:),intent(inout)::a,b
real,dimension(size(a))::aux !"aux" tiene la misma forma que "a"

aux=a

if(size(a)==size(b)) then !"b" debe tener la misma forma que "a"
  a=b
  b=aux
endif

end subroutine intercambio
```

En este último ejemplo, los arreglos `a` y `b` son **arreglos de forma asumida** (assumed-shape arrays), es decir, arreglos argumento de un procedimiento que sólo conoce su rango pero no su tamaño. En estos casos, el procedimiento debe tener una **interfaz explícita** (mediante un bloque `interface`) en el lugar de llamada.

– **Declaración aplazada o arreglo asignable -allocatable o deferred-shape array**
Algunas veces, el tamaño de un arreglo sólo se conoce después de leer algún dato o efectuar algún cálculo; en tal caso, podemos utilizar un **arreglo asignable de forma pospuesta o aplazada** (allocatable o deferred-shape array), que se define sólo como un nombre (y la información de su rango), pero sin memoria asociada (ya que no se conoce su tamaño). Sólo se podrá operar con este tipo de arreglos una vez que la instrucción `allocate` fija las extensiones de sus dimensiones; vemos un ejemplo:

```
program principal

implicit none
integer::n

real,allocatable::c(:,:),d(:,:) !"c" y "d" tienen rango 2
!y se van a utilizar para guardar dos matrices cuadradas

print*,"Introduce el orden de las matrices cuadradas c y d: "
read*, n

allocate(c(n,n),d(n,n))

print*,"Forma de los arreglos c y d: ",shape(c)

deallocate(c,d)

end program principal
```

Como se ve en el ejemplo, cuando el arreglo ya no es necesario, se puede, y se debe, liberar la memoria utilizada mediante la instrucción `deallocate`. Por último, debemos destacar que un arreglo declarado como `allocatable` no puede ser nunca un argumento ficticio en un procedimiento o el resultado de una función.

1.3 Funciones intrínsecas para manipulación de arreglos

Parte del potencial de FORTRAN 90, tanto en lo que se refiere a programación paralela como a una expresión concisa de muchos algoritmos, se debe a su amplio conjunto de procedimientos intrínsecos. Recordamos a continuación los que utilizaremos en esta práctica:

- `SIZE(a[,dim])`: calcula la extensión del arreglo `a` en la dimensión especificada por el valor `dim`. Si el argumento opcional `dim` se omite, esta función devuelve el número total de elementos del arreglo `a` (producto de las extensiones de sus dimensiones).

- `SHAPE(a)`: calcula un vector unidimensional, cuya longitud viene dada por el rango del arreglo `a` y cuyos coeficientes son las extensiones de las dimensiones del arreglo.
- `DOT_PRODUCT(vector_a,vector_b)`: efectúa el producto escalar de los vectores (arreglos de rango 1) del mismo tamaño `vector_a` y `vector_b`.
- `MATMUL(matrix_a,matrix_b)`: efectúa el producto de dos matrices compatibles para el producto `matrix_a` y `matrix_b`.
- `MAXVAL(array[,dim][,mask])`: calcula el valor máximo de los elementos del arreglo `array`; el argumento opcional `dim` selecciona el subíndice reducido, y el argumento opcional `mask` es un arreglo de valores lógicos, de la misma forma que `array`, que especifica el subconjunto de elementos del arreglo sobre los que actúa la función.
- `SUM(array[,dim][,mask])`: suma los elementos del arreglo `array`; los argumentos opcionales `dim` y `mask` tienen el mismo significado que el descrito anteriormente.

1.4 Entrada-salida de datos: repaso de read, print y formatos In,Rn,nA

1.5 Estructura de programas: `contains`, `interface` y `module`.

Para la buena gestión de las prácticas en programación FORTRAN es conveniente estructurar los programas de manera sencilla y asegurar una fácil revisión en la búsqueda de errores y sobre todo en la no repetición de programas ya hechos.

La forma general de un programa principal siempre tiene la forma siguiente.

```
program miprograma
!Declaración de variables locales: tipo, dimensión, allocatable...
!Entrada de datos
!Realización de cálculos: parte central del programa
!Escritura de resultados
end program miprograma
```

En este curso los programas que haremos son todos de poca complejidad, esto es, la parte central de los cálculos se realiza mediante la llamada a una subrutina o dos que también son sencillas (no muy largas). Existen tres opciones para hacerlo que vemos a continuación.

1. **Programas sencillos: uso de `contains`.**

   En esta opción las subrutinas están incluídas dentro del propio programa principal indicada por un bloque `contains` de la forma siguiente:

   ```
   program miprograma
   !Declaración de variables locales: tipo, dimensión, allocatable...
   !Entrada de datos
   !Para la realización de cálculos llamada a la(s) subrutina(s)
   call subroutine misubr(arg1,arg2,...)
   !Escritura de resultados

   contains

   subroutine misubr(x1,x2,...)
   ............
   end subroutine misubr
   end program miprograma
   ```

   En este caso todo el programa y sus subrutinas están en un mismo fichero `.f90` (que necesariamente se hace un poco más largo) pero tenemos la ventaja de poder consultar/corregir el programa principal y las subrutinas sin salir del fichero. La compilación también solo necesita el nombre de ese único fichero. Cuando las subrutinas van a ser utilizadas por varios programas (nosotros heremos algunos para utilizar durante todo el curso) es conveniente entonces escribir las subrutinas

en ficheros distintos. Si las subrutinas solo usan arreglos de declaración explícita (la dimensión es fija o se pasa en los parámetros de entrada) el programa principal no habría que retocarlo. Si las subrutinas utilizan arreglos de forma asumida –será nuetro caso– es necesario utilizar un bloque `interface` para conectar las subrutinas y el programa principal.

2. **Uso de `interface`.**

   Esta opción nos permite escribir el programa principal y las subrutinas con variables de forma asumida en **distintos ficheros** y comunicarlos mediante la información de presentación de las subrutinas que se hace en un bloque `interface`. La forma genérica sería como sigue:

```
program miprograma
implicit none
interface
  subroutine misubr(x,y,...)
  !Declaración de argumentos (COPIA DEL ENCABEZADO DE LA SUBRUTINA!)
  end subroutine misubr
end interface
!Declaración de variables locales: tipo, dimensión, allocatable...
!Entrada de datos
!Para la realización de cálculos llamada a la(s) subrutina(s)
call subroutine misubr(arg1,arg2,...)
!Escritura de resultados
end program miprograma
```

   La(s) subrutinas estará(n) en distinto(s) fichero(s) de la forma:

```
subroutine misubr(x1,x2,...)
!Declaración de argumentos
!Declaración de variables locales
...........
end subroutine misubr
```

   Dado que nosotros utilizaremos arreglos de forma asumida, esta forma de proceder es la que seguiremos. Como decíamos, tiene la ventaja de que el programa principal es más corto (más fácil de revisar para errores) y que **las subrutinas pueden ser usadas, tal cual están, por otros programas principales u otras subrutinas**. Esta es la opción más recomendable como hábito de programación –sobre todo porque los programas más avanzados suelen tener varias subrutinas, lo que complica la opción del `contains`. No obstante en las primeras prácticas usaremos la opción anterior por comodidad.

   Debemos añadir que, para hacer menos pesada la tarea de programación cuando la entrada y/o salida de datos es común a muchos programas (por ejemplo, en nuestro caso, la entrada de la matriz y el segundo miembro del sistema, la escritura de la solución) es conveniente tener también subrutinas para hacerlo.

3. **Progamas complejos: uso de `module`.**

   Finalmente, cuando se tiene experiencia en programación se puede estructurar todavía más los programas usando la estructura de módulos mediante `use module mimodulo` y definir el módulo `module mimodulo` en un fichero que comienza con `module mimodulo` y acaba en `end module mimodulo`. Esta forma de proceder es conveniente dejarla para cuando tengamos un poco más de experiencia en programación pero se puede usar desde ahora si se prefiere. Lo habitual es agrupar las variables y tipos de datos en un módulo y las subrutinas y funciones (procedimientos) en otro (aunque se pueden meter todo en el mismo módulo).

2 Ejercicios

1. (a) Escribe un programa principal que lea el número y las componentes de dos vectores del mismo tamaño, y que efectúe y escriba su producto escalar; utiliza la función intrínseca `DOT_PRODUCT (vector_a,vector_b)`. Comprueba su buen funcionamiento.

(b) Escribe un programa principal que lea los números de filas y columnas y los elementos de dos matrices compatibles para la suma, y que efectúe y escriba su suma. Comprueba su buen funcionamiento.

(c) Escribe un programa principal que lea los números de filas y columnas y los elementos de dos matrices compatibles para la multiplicación, y que efectúe y escriba su producto; utiliza la función intrínseca `MATMUL(matrix_a,matrix_b)`. Comprueba su buen funcionamiento.

2. Escribe un programa principal que lea el orden de una **matriz tridiagonal** A, sus tres diagonales, `ad`, `al` y `au`, y un vector v, y que calcule y escriba el **vector producto** $w = Av$.

3. (a) Escribe un subprograma `residuo(a,b,u,r)` que calcule el vector $r = Au - b$ siendo A una matriz de orden $m \times n$, u un vector de $\mathbb{R}^n$ y b un vector de $\mathbb{R}^m$, dados. Escribe un programa principal `residuo_ppal` que permita comprobar el buen funcionamiento del anterior subprogama, insertandolo en aquel después de una sentencia `contains`. Crea ficheros de datos adecuados para validar la tarea propuesta. En la subrutina puedes intentar alguna opción de uso eficiente de los arreglos para realizar la multiplicación más rápido. *Nota:* Cuando u es una solución del sistema lineal $Au = b$ al vector r se le llama **residuo** y de ahí el nombre del programa.

   (b) Realiza el ejercicio anterior usando la opción de `interface` incluyendo la subrutina en un fichero separado y no olvidar de incluir en la parte dedicada a especificaciones, un bloque `interface` de presentación de la subrutina `residuo(a,b,u,r)`. De nuevo, es recomendable adaptar copias de los programas anteriores y no escribirlos de nuevo.