# Notas Simulación en Física de materiales

Daniel Vazquez Lago

15 de septiembre de 2024

# Índice general

# Introducción

Usaremos $N = 500$ particulas y una densidad de 0.5 $N/V^3$. La variación máxima de energía permitida es 1/1000.

USaremos la aproximación de Lennard-Jones, hya que es suave, supone interacciones debiles ideales para los gases nobles.

$$v_{ij}(r_{ij}) = 4\epsilon \left[ (\sigma/r_{ij})^{12} - (\sigma/r_{ij})^{6} \right] \tag{1}$$

"Usar una suma doble para luego dividirlo por dos es para pegarle en la cara". La parte de los sumatorios debe estar libre de polvo y paja para que corra veloz.

$$t_p = \frac{1}{2} \sum \sum v_{ij} = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} v_{ij}$$

# 1 Introducción a Fortran

## 1.1. Estructura del programa

### 1.1.1. Formato código fuente

El formato de condigo guente puede ser libre o fijo, y no deben mezclarse ambos en un fichero de código. El código fijo se considera obsoleto en Fortran95. En cualquier caso existen ciertas normas básicas y típicas de fortran, ataño obligatorias, que todavía se mantienen, por lo que es importante mencionarlas. Estas son:

- Las sentencias de un programa se escribem en diferentes líneas.
- La posición de los caracterres dentro de las líneas es significativa.
- Columnas:
  - 1-5. Número de etiqueta (de 1 a 5 dígitos, se usan números usualmente).
  - 6. Carácter de continuación de línea.
  - Resto. Sentencia.
- Comentarios:
  - Las líneas en blanco se ignoran. Hacen más legible el programa.
  - Si el primer carácter de una línea es *, c o C la línea es de comentario.
  - Si aparece el carácter ! en una línea (salvo en la columna 6) lo que sigue es un comentario.
- Una línea puede contener varias sentencias separadas por punto y coma (;), el cual no puede estar en la columna 6. Sólo la primera de estas sentencias podría llevar etiqueta.
- Los espacios en blanco son significativos: `IMPLICIT NONE`, `DO WHILE` (obsoleto), `CASE DEFAULT`. Son opcionales en:
  - Palabras clave dobles qeu comienzan por `END` o `ELSE`.
  - `DOUBLE PRECISION`, `GO TO`, `IN OUT`, `SELECT CASE`.
- El indicador de continuación de una línea es el carácter &.

### 1.1.2. Tipos de datos

Fortran tiene los siguientes tipos de datos:

- Enteros (`INTEGER`)
- Reales (`REAL, DOUBLE PRECISION`)
- Complejos (`COMPLEX`)
- Lógicos (`LOGICAL`)
- Caracteres (`CHARACTER, CHARACTER(LEN=n), CHARACTER*n`)

**Parámetros. Variables. Declaración. Asignación.**

- Un parámetro tiene un valor que no se puede cambiar (PARAMETER).
- Una variable puede cambiar su valor cuantas veces sea necesario.
- Por defecto, todas las variables que empiecen por `i,j,k,l,m` o `n` son entreas y las demas reales. Es muy recomendable declarar las variables que se utilicen (la sentencia `IMPLICIT NONE` obliga a declarar todas las variables).

**Arrays, subíndices, substrings**

- Un array se define mediante su nombre y dimensiones (cantidad y límites).
- Por defecto el primer índice es 1. En otro caso hay que indicar el rango `i1:i2`.
- Los elementos del array se acceden por sus índices entre paréntesis.

### 1.1.3. Operadores y expresiones

**Aritméticas**

- Los operadores aritméticos son `+, -, *, /, **`.
- El orden de prioridades es el mismo que en el álgebra.
- No puede ver operadores seguidos (incorrecto `a*-b`, correcto `a*(-b)`).

**Relacion y expresiones lógicas**

- Los operadores de expresiones son:

| .EQ. | .NE. | .LT. | .LE. | .GT. | .GE. |
|---|---|---|---|---|---|
| == | / = | < | <= | > | >= |

- Se pueden relacionar expresiones aritméticas con expresiones lógicas y expresiones de caracteres.
- Es recomendable utilizar paréntesis y/ó sustituir las expresiones complicadas por combinaciones de expresiones más simples.
- Los operadores lógicos son:

| Operador | .NOT. | .AND. | .OR. | .EQV. | .NEQV. |
|---|---|---|---|---|---|
| Prioridad | 1 | 2 | 3 | 4 | 4 |

### 1.1.4. Entrada y salida estándar sin formato

Los dispositivos estándar (por defecto) de entrada y salida de datos son el teclado y la pantalla:

- Lectura de datos de teclado. Son equivalentes las siguientes sentencias:
  - `READ (*,*), listavar`
  - `READ* , listavar`
- Escritura de datos en pantalla. Son equivalentes las siguientes sentencias:
  - `WRITE (*,*), listavar`
  - `PRINT* , listavar`

Donde `listavar` es una lista de variables o elementos de arrays separados por comas.

### 1.1.5. Sentencias, `PROGRAM, END`

- Un programa puede comenzar con la sentencia `PROGRAM nombprog`.
- Un programa debe terminar con la sentencia `END [PROGRAM [nombreprog]]`.

Donde `nombreprog` es el nombre del programa, que debe empezar por una letra y admite hasta 31 letras, dígitos y guiones underscore.

## 1.2. Setencias de control

Las sentencias de control sirven para alterar la ejecución de las sentencias de un programa.

### 1.2.1. Setencia `CONTINUE`

La sentencia `CONTINUE` es ejecutable, pero no realiza acción alguna. Es útil para rupturas de secuencia y manejo de errores en lectura de datos. Su número de etiqueta puede ser referenciado en sentencia `DO`.

### 1.2.2. Setencia `STOP`

La sentencia `STOP` detiene la ejecución del programa. Tiene dos variantes `STOP ['mensaje'], STOP[n]`. Si está presente el literal `'mensaje'` ó el número `n` (que ha de tener de 1 a 5 dígitos), se visualizan en pantalla. Puede llevar etiqueta y formar parte de una sentencia `IF`, y sirve principalmente para detener la ejecución a causa de un error y con el literal 'mensaje' ó el número `n`.

### 1.2.3. Setencia `GOTO`

La sintaxis `GOTO e` transfiere el control a la secuencia ejecutable con etiqueta `e` que se encuentra en la misma unidad de programa que la setencia `GOTO`. No se puede entrar en bloques `DO, IF, CASE` desde fuera de ellos con las sentencias `GOTO`.

Es una sentencia cuyo uso genera mucha polémica. Un programa con gran cantidad de `GOTO` es difíciol de comprender, sobre todo si hay muchas transferencias a sentencias anteriores. Con una adecuada programación pueden sustituirse, con facilidad, la mayoría de las sentencias `GOTO` por otras estructuras de control. Sin embargo, hay ocasiones cuya sustitución complica enormemente la lógica del programa. Es especialmente útil para tratar condiciones de error o de terminación de un bloque. Conviene NO abusar de esta sentencia.

Si la sentencia siguiente a la sentencia `GOTO` no lleva etiqueta no se ejecutará nunca (código muerto). Es un síntoma de error de programación.

### 1.2.4. Setencia `IF`

La sintaxis es `IF (expres) sentec`. La expresión `expres` debe ser escalar lógica, si es verdadera se ejecuta `sentenc`; si es falsa no se ejecuta `sentenc` y se continua a la sentencia siguiente. La sentencia `sentenc` debe ser ejecutable, no puede ser la sentencia `END` ni la sentencia inicial o final de bloques `DO, IF, SELECT, CASE`.

Esta sentencia se suele utilizar para realizar, en función de la condición, una única asignación, una sencilla escritura de datos, una parada del programa, una ramificación del flujo del programa.

### 1.2.5. Bloque `IF-THEN-ENDIF`

La sitáxis es la siguiente:

```
[nomb:] IF (expres) THEN
  bloq....
ENDIF[nomb]
```

La expresión `expres` debe ser escalar lógica. Si es verdadera se ejectutan las sentencias del bloque `bloq` entre `THEN` y `ENDIF`. Si es falsa se continúa en la siguiente sentencia a `ENDIF`. Si lleva nombre (opcional), debe ser un nombre válido en Fortran distinto de otras nombres en la unidad de alcance en la que está el bloque `IF`.

### 1.2.6. Bloque `IF-THEN-ELSE-ENDIF`

La sitáxis es la siguiente:

```
[nomb:] IF (expres) THEN
  bloq1
ELSE [nomb]
  bloq2
EDNDIF[nomb]
```

La expresión `expres` debe ser escalar lógica. Si es verdadera se ejectutan las sentencias del bloque `bloq` entre `THEN` y `ENDIF`. Si es falsa se continúa en la siguiente sentencia a `ENDIF`. Si lleva nombre (opcional), debe ser un nombre válido en Fortran distinto de otras nombres en la unidad de alcance en la que está el bloque `IF`.

### 1.2.7. Bloque ELSE-IF

La sitáxis es la siguiente:

```
[nomb:] IF (expres) THEN
  bloq1
[ElSEIF (expres_i) THEN [nomb] b
  bloq_i]...
[ELSE [nomb]
  bloq2]
EDNDIF[nomb]
```

Cada expresión `expres, expres_i` debe ser escalar lógica. Si `expres` es verdadera se ejecutan las sentencias del bloque `bloq1` entre `THEN` y el primer `ELSEIF` y se pasa a la siguiente sentencia a `ENDIF`. Si `expres_i` es falsa se inspeccionan en orden las expresioens `expres_i` hasta que una sea verdadera, en cuyo caso se ejecutan las sentencias del bloque `bloq_i` correspondiente y el control pasa a la siguiente sentencia `ENDIF`. Si la expresión `expres` y todas las expresiones `expres_i` son falsas, se ejecutan las sentencias del bloque `bloq2` entre `ELSE` y `ENDIF`.

Las sentencias `ELSEIF` y `ELSE` pueden llevar nombre sólo si sus sentencias `IF` y `ENDIF` llevan y, en estec aso debe ser el mismo. No se puede entrar en un bloque `IF, THEN, ELSEIF` ni `ELSE` desde fuera de él con sentencias `GOTO`, aunque si se peude salir en cualquier lugar con la misma. En el siguiente código podemos ver un ejemplo del uso de este bloque para resolver una función definida a trozos.

```
REAL x, f
PRINT*,'valor de x =' ; READ*, x
IF (0<=x .AND. x<=1) THEN
  f = 3*x**2 - 1
ELSEIF (5<=x .AND. x<=10) THEN
  f = 6*x + 4
ELSEIF (20<=x .AND. x<=40) THEN
  f = -7*x + 1
ELSE
  f = 0
ENDIF
PRINT*, ' x = ', x, ' f = ', f
END
```

### 1.2.8. Selector SELECT CASE

La sintaxis es la siguiente:

```
[nomb:] SELECT CASE (expres)
CASE (selector) [nomb]
  bloq...
CASE DEFAULT [nomb]
  bloq0...
ENDSELECT [nomb]
```

La expresión `expres` debe ser escalar de tipo entera, lógica (*poco interesante*) ó carácter y los valores dados deben ser del mismo tipo (en el caso carácter las longitudes pueden ser diferentes pero no la clase, en los casos entero o lógico pueden ser diferentes). Si el valor `expres` pertenece a un selector se ejecuta su bloque de sentencias y se continúa en la siguiente sentencia a `END SELECT`. Si no pertenece a ningún selector se ejecutan las sentencias del bloque `CASE DEFAULT`, si está presente y si no lo está se continúa en la siguiente sentencia a `END SELECT`.

Si lleva nombre (opcional) debe ser un nombre válido en Fortran y distinto de otros nombres en la unidad de alcance en qeu se encuentra el bloque `SELECT`. Las sentencias `CASE` y `CASE DEFAULT` pueden llevar nombre sólo si las sentencias `SELECT CASE` y `CASE` correspondientes lo llevan y, en este caso, debe ser el mismo. Los valores de los selectores han de ser disjuntos. Se separan por comas y puede especificarse un rango de valores, también disjuntos en cada selector. No se puede entrar en un bloque `SELECT` ó `CASE` desde fuera de él con sentencias `GOTO`. Se puede salir en cualquier lugar con sentencias `GOTO`. Los bloques `SELECT CASE` pueden anidarse.

La diferencia principal entre los bloques `IF` y `CASE` es que en `CASE` sólo se evalúa una expresión cuyo valor deb estar en un conjunto predefinido de valores, mientras que en `IF` se pueden evaluar varias expresiones de naturaleza distinta. Veamos un ejemplo para entender mejor el problema:

```
1  CHARACTER car! equivale a CHARACTER(LEN=1) car
2  INTEGER indice
3  PRINT*, ' Introducir un caracter'
4  READ*, car
5  SELECT CASE (car)
6  CASE ('a', 'e', 'i', 'o', 'u')
7    PRINT*, ' Vocal minuscula : ', car
8  CASE ('A', 'E', 'I', 'O', 'U')
9    PRINT*, ' Vocal MAYUSCULA : ', car
10 CASE ('b':'d', 'f':'h', 'j':'n', 'p':'t', 'v':'z')
11   PRINT*, ' Consonante minuscula : ', car
12 CASE ('B':'D', 'F':'H', 'J':'N', 'P':'T', 'V':'Z')
13   PRINT*, ' Consonante MAYUSCULA : ', car
14 CASE ('0':'9')
15   PRINT*, ' Cifra del 0 al 9 : ', car
16 CASE DEFAULT
17   PRINT*, ' El caracter no es ni letra ni numero : ', car
18 ENDSELECT
19
20 PRINT*, ' Introducir un numero'
21 READ*, indice
22
23 SELECT CASE (indice)
24 CASE (2, 3, 5, 7, 11, 13, 17, 19)
25   PRINT*, ' Numero primo menor que 20 : ', indice
26 CASE (20:29, 40:49, 60:69, 80:89)
27   PRINT*, ' Numero menor que 100 con decena par : ', indice
28 CASE (100:999)
29   PRINT*, ' Numero de 3 cifras : ', indice
30 CASE DEFAULT
31   PRINT*, ' Resto de casos : ', indice
32 ENDSELECT
33 END
34
```

### 1.2.9. Interacciones `DO`

La sintaxis es:

```
1  [nomb:] Do [,] var = expres1, expres2, expres3
2    bloq
3  ENDDO [nomb]
```

La variable `var` y las expresiones `expres1, expres2, expres3` (esta última opcional) deben ser escalares enteras. La variable `var` toma el valor inicial de `expres1`, se ejecutan las

sentencias `bloq` del bloque `DO`; `var` se incrementa en `expres3`, se ejecutan las sentencias del bloque `DO`, y así sucesivamente hasta que `var>expres2` (si `expres3>0` ó `var<expres2` (si `expres3<0`) en cuyo caso se continúa en la siguiente sentencia a `ENDDO`.

El número de interacciones que se realizan en el bloque `Do` (si no se sale de él antes de terminar) es: `MAX{ (expres2-expres1+expres3)/expres3,0 }`. Cuando { `expres1>expres2` y `expres3>0` } ó { `expres1<expres2` y `expres3<0` } no se ejecuta el bloque `DO`. Si `expres1` y/ó `expres2` y/ó `expres3` son expresiones que incluyen variables, el valor de éstas puede cambiarse dentro del bloque `DO` y esto no altera el número de interaciones (calculado con sus valores iniciales).

### 1.2.10. `DO` ilimitado, `EXIT` y `CYCLE`

El `DO` ilimitado tiene la siguiente sintaxis:

```
[nomb:] Do
  bloq
ENDDO [nomb]
```

Y la acción del mismo es que se repitan las sentencias `bloq` indefinidamente, pudiendo salir del mismo con las sentencias `EXIT` y `GOTO`.

La sentencia `EXIT[nomb]` dentro de un bloque `Do` transfiere el control a la primera sentencia ejecutable después del `ENDDO` a que se refiere, sino se indica el nombre `nomb`, transfiere el control al `ENDDO` del bloque más interior en el que está contenida.

La sentencia `CYCLE[nomb]` dentro de un bloque `Do` transfiere el control a la sentencia `ENDDO` a que se refiere, sino se indica el nombre `nomb`, transfiere el control al `ENDDO` del bloque más interior en el que está contenida.

## 1.3. Utilidades de programa. Procedimientos.

### 1.3.1. Programa principal

Un programa completo debe tener exactamente un programa principal. La forma es la siguiente:

```
PROGRAM [nombreprog]
  [sentencias de especificacion]
  [sentencias ejecutables]
CONTAINS
  [subprogramas internos  ]
END PROGRAM [nombreprog]
```

### 1.3.2. Subprogramas externos

Son llamados desde el programa principal o desde otros subprogramas. Pueden ser funciones o subrutinas. Ambas pueden ser recursivas, esto es, llamarse a sí mismas. No obstante esto es muy complejo, y su uso implica peor eficacia computacional.

### 1.3.3. Uso no recursivo de subprogramas `FUNCTION`

La sitaxis es la siguiente:

```
[tipo] FUNCTION nombfun ([argumentos ficticios])
  [sentencias de especificacion]
  [sentencias ejecutables]
END [FUNCTION [nombreprog]]
```

Con esto estaríamos definiendo el subprograma `FUNCTION nombfun`, invocándose con `nombfun([argumer actuales])`, y substituyendose los argumentos actuales en los ficticios y se evalúa la función. El valor asignado a `nombfun` es el valor devuelto a la función. Téngase en cuenta que:

- El término `tipo` es opcional. Si se omite se toma el tipo por defecto o el que haya sido establecido por sentencias `IMPLICIT`.
- La llamada puede formar parte de una expresión o sentencia más larga. Un subprograma `FUNCTION` puede contener cualquier sentencia excepto: `PROGRAM, FUNCTION, SUBROUTINE` y `BLOCK DATA`.
- La última sentencia tiene que ser `END`.
- Las variables y etiquetas en un subprograma `FUNCTION` son locales, esto es, independientes del programa principal y las de otros subprogramas.
- Los argumentos actuales deben coincidir en cantidad, orden, tipo y longitud con los argumentos ficticios. Puede no haber argumentos.
- Los argumentos actuales pueden modificarse: sin embargo, esta opción es especialmente desaconsejable.
- Una función no recursiva no puede llamarse a sí misma ni directa ni indirectamente, pero sí puede llamar a otros subprogramas.

#### Sentencia `RETURN` en Subprogramas `FUNCTION`

`RETURN` termina la ejecución de la función y devuelve el control a la unidad de programa que llamó a la función. Si la función no tiene sentencias `RETURN` su ejecución termina al llegar a la sentencia `END`. Puede ser una setntencia con etiqueta, y puede formar parte de una sentencia `IF`.

### 1.3.4. Uso no recursivo de Subprogramas `SUBROUTINE`

La sintaxis escalar

```
SUBROUTINE nombsubr ([argumentos ficticios])
  [sentencias de especificacion]
  [sentencias ejecutables]
END [SUBROUTINE [nombresubr]]
```

La llamada a la subrutina se realiza mediante la sentencia `CALL nombsubr [(argumentos actuales)]`, substituyendose los argumentos actuales en los ficticios. Las normas son las siguientes:

- La llamada puede formar parte de una expresión o sentencia más larga. Un subprograma `FUNCTION` puede contener cualquier sentencia excepto: `PROGRAM, FUNCTION, SUBROUTINE` y `BLOCK DATA`.
- La última sentencia tiene que ser `END`.

- Las variables y etiquetas en un subprograma `FUNCTION` son locales, esto es, independientes del programa principal y las de otros subprogramas.
- Los argumentos actuales deben coincidir en cantidad, orden, tipo y longitud con los argumentos ficticios. Puede no haber argumentos.
- Una función no recursiva no puede llamarse a sí misma ni directa ni indirectamente, pero sí puede llamar a otros subprogramas.

#### Sentencia `RETURN` en Subprogramas `SUBROUTINE`

`RETURN` termina la ejecución de la subrutina y devuelve el control a la unidad de programa que llamó a la función. Si la subrutina no tiene sentencias `RETURN` su ejecución termina al llegar a la sentencia `END`. Puede ser una setntencia con etiqueta, y puede formar parte de una sentencia `IF`.

### 1.3.5. Argumentos de subprogramsas externos

Los argumentos de un subprograma `FUNCTION` O `SUBROUTINE` pueden ser de naturaleza muy diversa: constantes o variables escalares, arrays o elementos de arrays, nombres de otros subprogramas, etc. Es necesario suministrar al compilador la información adecuada para identificar correctamente la naturaleza del argumento.

#### Propósito de los argumentos

Los argumentos ficticios pueden tener una declaración de propósito de entrada salida o entrada/salida. El propósito se declara con el atributo `INTENT`.

- `INTENT (IN)`: declara un argumento de entrada. No debe cambiarse su valor dentro del subprograma.
- `INTENT (OUT)`: declara un argumento de salida. El argumento actual debe ser una variable y se vuelve indefinida en entrada.
- `INTENT (IN/OUT)`: declara un argumento de entrada o salida. El argumento actual debe ser una variable.

Es recomendable declarar el propósito de los argumentos ficticios, lo cual ayuda a la documentación del programa y a las verificaciones durante la compilación.

### 1.3.6. Sentencia `EXTERNAL`

Se escribe como `EXTERNAL lista`, e identifica los nombres de `lista` como subprogramas (funciones o subrutinas) externos definidos por el usuario. Al ser una sentencia de especificación, debe preceder a las ejecutables y a las declaraciones de funciones. Cuando un argumento de un subprograma es el nombre de otro subprogarama, se debe declarar `EXTERNAl` en su unidad de llamada. Si una función intrínseca se declara `EXTERNAL` pierde su definición intrínseca en la unidad de programa asociada y se usa el subprograma del usuario.

```
1  EXTERNAL fun1, fun2, sin
2  x=1.5; n=3
3  CALL ameba (x, n, y1, fun1)    ! y1 = x**(5-n) = 2.25
4  CALL ameba (x, n, y2, fun2)    ! y2 = 3*x*(5-n) = 9
5  s = sin (y1, y2, 6.0, x)       ! s = y2/y1 + 6/x = 8
```

```
  PRINT*, y1, y2, s
  END

  SUBROUTINE ameba (x, n, y, f)
    y = f(x, 5, n)
  END

  FUNCTION fun1 (x, i, j)
    fun1 = x**(i-j)
  END

  FUNCTION fun2 (x, i, j)
    fun2 = 3*x*(i-j)
  END

  FUNCTION sin (a, b, c, d)
    sin = b/a + c/d
  END
```

### 1.3.7. Sentencia INTRINSIC

La sintaxis `INTRINSIC lista` declara los nombres de `lista` como funciones intrínsecas. Las normas son

- Los nombres de la `lista` deben ser funciones intrínsecas.
- Si un argumento de un subprograma es una función intrínseca se debe declarar `INTRINSIC` en la unidad de llamada.
- Si un nombre está en una sentencia `INTRINSIC` no puede estar en una sentencia `EXTERNAL`.

```
  INTRINSIC sin, cos, exp
  a=3.141592; b=-a
  r = fun (sin, cos, exp, a, b, 4)
  PRINT*, r
  END

  FUNCTION fun (f1, f2, f3, a, b, n) ! fun = (sin(a)+cos(b)+
  fun = (f1(a) + f2(b) + f3(a+b)) ** n ! exp(a+b))**n
  END
```

### 1.3.8. Subprogramas internos

Son subprogramas contenidos en el programa principal, en un subprograma externo o en un módulo. Su uso es adecuado, a efectos de organización, para subprogramas cortos (del orden de unas 20 líneas), que sólo se necesitan en un único programa, subprograma o módulo. La sintáxis es la siguiente:

```
  CONTAINS
  subprogramas internos
```

Y las normas son:

- Los subprogramas internos deben aparecer entre la sentencia `CONTAINS` y la sentencia `END` de la unidad de programa a la que pertenezcan.
- Un subprograma interno no puede contener a otro subprograma interno.

- Un subprograma interno sólo puede llamarse desde su host.
- Un host conoce todo acrca de la interface con sus subprogramas internos, por tanto no hace falta declarar el tipo en el host para una función interna.
- Un subprograma interno tiene acceso a las variables del host.
- El host no tiene acceso a las variables locales de los subprogramas internos.
- La sentencia `IMPLICIT NONE` en un host afecta al host y también a sus subprogramas internos.
- Las etiquetas son locales. Si una sentencia tiene etiqueta, ésta debe estar en la misma unidad de alcance que la sentencia que la referencia.

```
1  PROGRAM interno
2    CALL coefbinomial ! invoca una subrutina interna
3
4  CONTAINS
5
6    SUBROUTINE coefbinomial
7      INTEGER n, k
8      CALL leer(n) ! invoca una subrutina interna
9      DO k = 0, n
10       PRINT*, k, nsobrek(n,k) ! invoca una funcion interna
11     ENDDO
12   ENDSUBROUTINE coefbinomial
13
14   SUBROUTINE leer(n)
15     INTEGER n
16     PRINT*, ' Introducir el valor de n'
17     READ*, n
18   ENDSUBROUTINE leer
19
20   FUNCTION nsobrek(n,k)
21     INTEGER nsobrek, n, k
22     nsobrek = fact(n) / (fact(k)*fact(n-k)) ! invoca una funcion
23   ENDFUNCTION nsobrek ! interna 3 veces
24
25   FUNCTION fact(m)
26     REAL fact
27     INTEGER m, i
28     fact = 1
29     DO i = 2, m
30       fact = i*fact
31   ENDDO
32   ENDFUNCTION fact
33
34 ENDPROGRAM interno
```

Su principal utilidad es organizar mejor el código y permitir un mayor control sobre el ámbito de las variables, ya que los subprogramas internos solo pueden ser llamados desde el subprograma en el que están definidos. Las principales ventajas de los subprogramas internos son:

- Encapsulamiento: Permiten encapsular la lógica auxiliar o funciones específicas que solo tienen sentido dentro del contexto del subprograma principal. Esto ayuda a mantener el código más legible y modular.

- Ámbito de variables: Las variables locales del subprograma externo pueden ser utilizadas directamente dentro del subprograma interno, lo que evita la necesidad de pasarlas como argumentos. Esto simplifica el manejo de variables cuando son comunes a ambos subprogramas.

- Modularidad: Facilita la descomposición de tareas complejas en tareas más simples, dividiendo la lógica en partes más manejables, lo que mejora la mantenibilidad del código.

### 1.3.9. Módulos

Un módulo permite empaquetar definiciones de datos y compartir datos entre diferentes unidades de programas que pueden incluso compilarse por separado. Sirve, especialmente, para crear grandes librerías de software. En su uso sencillo:

- Ofrece posibilidades similares a `INCLUDE`.

- Permite compartir datos en ejecución.

- Sirve para inicializar variables.

La sitátxis es la siguiente:

```
MODULE nombmod
  [sentencias de especificacion]
ENDMODULE nombmod
```

Y las normas

- Se puede acceder a un módulo desde el programa principal, un subprograma u otro módulo. Se accede a las especificaciones y variables del módulo con los valores asignados (si los tienen). Las variables y datos de un módulo con los valores asignados si los tienen. Las variables y datos de un módulo tienen, por defecto, alcance **global**, en todas las unidades desde las que se acceden con `USE`.

- Desde un módulo se tiene acceso a las otras entidades del módulo incluyendo subprogramas.

- Puede contener sentencias `USE` para acceder a otros módulos.

- No debe acceder a sí mismo directamente o indirectamente a través de `USE`.

- El módulo debe compilarse antes que el programa que lo usa. En la sentencia compilación se crea un fichero `*.mod` que es el que lee la sentencia `USE`. Se recomienda que un módulo solo acceda a módulos anteriores a él.

### 1.3.10. Orden de las sentencias

Las diferentes sentencias que puede contener un programa de Fortran deben escribirse en el orden siguiente (tabla 1.1).

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, MODULE</td></tr>
<tr><td colspan="3">USE</td></tr>
<tr><td rowspan="4">FORMAT</td><td colspan="2">IMPLICIT NONE</td></tr>
<tr><td>PARAMETER</td><td>IMPLICIT</td></tr>
<tr><td>PARAMETER, DATA</td><td>Tipos derivados<br>Bloques INTERFACE<br>Declaración de tipos<br>Sentencias de especificación</td></tr>
<tr><td colspan="2">Sentencias ejecutables</td></tr>
<tr><td colspan="3">CONTAINS</td></tr>
<tr><td colspan="3">Subprogramas inteernos o subprogramas modulo</td></tr>
<tr><td colspan="3">END</td></tr>
</table>

Cuadro 1.1: orden de las sentencias.

## 1.4. Procedimientos intrínsecos

Los procedimientos intrínsecos son funciones y subrutinas que forman parte del lenguaje Fortran estándar, suministradas con el compilador. En fortar 95 hay 109 funciones y 6 subrutinas que pueden clasificarse en cuatro categorías de procedimientos intrínsecos:

- **Procedimientos elementales**: sus argumentos son escalares o arrayas. Si una función elemental se aplica a un array la función se aplica a cada elemento del array.
- **Funciones de interrogación**: devuelven propiedades de sus argumentos que no dependen de sus valores.
- **Funciones transformacionales**: suelen tener argumentos de arrays y resultados de arrays cuyos elementos dependen de muchos elementos del argumento.
- **Subrutinas no elementales**

Cada función devuelve un valor entero, real, complejo, lógico... de modo que tendremos que abreviar de algún modo el tipo de valor devuelto. En este caso usaremos que:

| | |
|---|---|
| I | Entero |
| R | Real |
| N | Numerico |
| L | Logico |
| CH | Carácter |

Cuadro 1.2: tipo de variable.

### 1.4.1. Funciones elementales que convierten tipos

| Nombre | Definición | Tipo argumentos | Tipo función |
|---|---|---|---|
| ABS(x) | Valor absoluto | I | I |
| ABS(x) | Valor absoluto | R | R |
| ABS(z) | Módulo complejo | C | R |
| AIMAG(z) | Parte imaginaria | C | R |
| AINT(x) | Quita decimales | R | R |
| ANINT(x) | Redondeo | R | R |
| CEILING(x) | Redondeo (por arriba) | R | I |
| CMPLX(x[,y]) | Pasa a complejo | N | C |
| FLOOR(x) | Redondeo (por abajo) | R | I |
| INT(x) | Pasa a entero | N | I |
| NINT(x) | Redondeo entero | R | I |
| REAL(x) | Pasa a real | N | R |

Cuadro 1.3: funciones elementales que pueden convertir tipos.

### 1.4.2. Funciones elementales que no convierten tipos

El resultado de las funciones elementales 1.4 es del tipo de su primer argumento.

| Nombre | Definición | Tipo argumentos | Tipo función |
|---|---|---|---|
| CONJG(z) | Conjugado complejo | C | C |
| DIM(x,y) | Diferencia positiva | (I,I) ó (R,R) | I ó R |
| MAX(x1,x2,[,x3,...]) | Máximo | (I,I,...) ó (R,R,...) | I ó R |
| MIN(x1,x2,[,x3,...]) | Mínimo | (I,I,...) ó (R,R,...) | I ó R |
| MOD(x,y) | Resto de x módulo y | (I,I) ó (R,R) | I ó R |
| MODULO(x,y) | x módulo y | (I,I) ó (R,R) | I ó R |
| SIGN(x,y) | Transferencia de signo | (I,I) ó (R,R) | I ó R |

Cuadro 1.4: funciones elementales que no pueden convertir tipos.

### 1.4.3. Funciones matemáticas elementales

El resultado de las funciones elementales 1.5 es del tipo de su primer argumento.

### 1.4.4. Operaciones con matrices y vectores

La función `DOT_PRODUCT(x,y)` requiere que x e y tengan una dimensión y el mismo tamaño. Si $x$ es entero o real devuelve $\sum x_i y_i$; si $x$ es complejo devuelve $\sum \overline{x}_i y_i$. Véase tabla 1.6.

La función `MATMUL(a,b)` devuelve un array de dos dimensiones en función de la forma de los dos arrays según la tabla 1.7.

| Nombre | Definición | Tipo argumentos | Tipo función |
|---|---|---|---|
| ACOS(x) | Arco Coseno | R / $\|x\| \leq 1$ | R en $[0,\pi]$ |
| ASIN(x) | Arco Seno | R / $\|x\| \leq 1$ | R en $[-\pi/2,\pi/2]$ |
| ATAN(x) | Arco Tangente | R | R en $[-\pi/2,\pi/2]$ |
| ATAN2(y,x) | Argumento número complejo | (R,R) | R en $(-\pi,\pi]$ |
| COS(x) | Coseno | R ó C | R ó C |
| COSH(x) | Coseno hiperbólico | R | R |
| EXP(x) | Exponencial | R ó C | R ó C |
| LOG(x) | Logaritmo neperiano | R ó C | R ó C |
| LOG10(x) | Logaritmo decimal | R x>0 | R |
| SIN(x) | Seno | R ó C | R ó C |
| SINH(x) | Seno hiperbólico | R | R |
| SQRT(x) | Raíz cuadrada | R ó C | R ó C |
| TAN(x) | Tangente | R | R |
| TANH(x) | Tangente hiperbólica | R | R |

Cuadro 1.5: funciones matemáticas elementales.

| Nombre | Definición | Tipo argumentos | Tipo función |
|---|---|---|---|
| DOT_PRODUCT(x,y) | Producto escalar real | (I ó R, I ó R) | I ó R |
| DOT_PRODUCT(z,y) | Producto escalar complejo | (C,I ó R) | C |
| MATMUL(a,b) | Producto matricial | (N,N) | N |
| TRANSPOSE(a) | Matriz traspuesta | N | N |

Cuadro 1.6:

| Operación | Forma de a | Forma de b | Forma de MATMUL(a,b) |
|---|---|---|---|
| Matriz x Matriz | (n,m) | (m,k) | (n,k) |
| Vector x Matriz | (m) | (m,k) | (k) |
| Matriz x Vector | (n,m) | (m) | (n) |

Cuadro 1.7:

| Nombre | Definición | Tipo argumentos | Tipo función |
|---|---|---|---|
| MAXVAL(x) | Máximo elemento | I ó R | I ó R |
| MINVAL(x) | Mínimo elemento | I ó R | I ó R |
| PRODUCT(x) | Producto de los elementos | I ó R | I ó R |
| SUM(a) | Suma de los elementos | I ó R | I ó R |

Cuadro 1.8:

### 1.4.5. Números aleatorios

La sintaxis para llamar a números aleatorios es la siguiente `CALL RANDOM_NUMBER ([HARVEST=]aleat)`, donde `aleat` es un argumento real (escalar o array), devolviendo números seudoaleatorios en `aleat` en el rango [0,1).

El otro tipo de forma de obtener un número aleatorio es usar `CALL RANDOM_SEED ([SIZE],[PUT],[GET]` donde los argumentos son:

- `SIZE`: variable escalar `INTEGER`. Variable de salida que contiene el tamaño `N` del array semilla.
- `PUT`: array `INTEGER` de dimensión (`N`). Variable de entrada utilizada para establecer la

semilla.

- `GET`: array `INTEGER` de dimensión (`N`). Variable de salida que contiene el valor actual de la semilla.

Si no se especifica una semmilla se establece una semilla que depende del procesador. Veamos un ejemplo de donde usamos los números aleatorios y el tiempo de cálculo:

```
1  PROGRAM aleatorio
2  INTEGER t1(8), t2(8)
3  INTEGER i, numrep, semilla(1)
4  REAL x, sx, tdif
5  CHARACTER(LEN=8) date1, date2
6  CHARACTER(LEN=10) time1, time2
7  CHARACTER(LEN=5) zona
8
9  PRINT*, ' numero de repeticiones'
10 READ*, numrep
11 PRINT*, ' semilla inicial'
12 READ*, semilla
13 CALL RANDOM_SEED (PUT=semilla)
14
15 CALL DATE_AND_TIME (VALUES=t1, DATE=date1, ZONE=zona, TIME=time1)
16 DO i = 1, numrep
17   CALL RANDOM_NUMBER (x)
18   sx = SIN(x)
19 ENDDO
20
21 CALL DATE_AND_TIME (VALUES=t2, TIME=time2, DATE=date2)
22
23 PRINT*, ' zona=', zona
24 PRINT*, ' date1=', date1, ' date2=', date2
25 PRINT*, ' time1=', time1, ' time2=', time2
26
27 tdif = 0.001*(t2(8)-t1(8)) + (t2(7)-t1(7)) + 60.*(t2(6)-t1(6)) + &
28   3600.*(t2(5)-t1(5))
29 PRINT*, ' tdif =', tdif
30
31 ENDPROGRAM aleatorio
```

## 1.5. Entrada y salida de datos. Ficheros. Formatos.

### 1.5.1. Elementos y clases de ficheros

Los conceptos fundamentales a considerar son: campos, registro y fichero.

- **Campo:** unidad de información que consta de varios caracteres que se tratan en conjunto.
- **Registro:** conjuto de campos, no necesariamente del mismo tipo.
- **Fichero:** conjunto de registros, no necesariamente con igual estructura.

Como ejemplo un fichero de personas podría contener un registro por cada persona y los campos podrían ser: nombre, DNI, dirección, edad, teléfono... En algunos casos, por ejemplo en las bases de datos, los registros de un fichero tienen la misma estrucutra, esto es, el mismo número y forma de los campos. Los tipos de **acceso** a un fichero en Fortran son secuencial o directo:

- **Secuencial:** para acceder a un registro hay que recorrer todo el fichero desde el principio hasta llegar a él.
- **Directo:** conociendo el número de orden de un registro en el dichero se puede acceder a él sin tener que recorrer los registros anteriores.

Los datos pueden almacenarse en **forma** formateada o no formateada:

- **Formateada**: la información se guarda como caracteres ASCII, legibles con la mayoría de los procesadores de texto.
- **No formateada**: un fichero es una serie de registros formados por "bloques físicos".

### 1.5.2. Lectura y escritura de datos

Internamente el ordenador representa los números y caracteres con cierta codificación. Para poder interpretar unos datos de entrada o mostrar unos datos de salida de forma legible se hacen conversiones entre la representación interna y la externa mediante especificaciones de formato.

Las entidades a leer o escribir se llaman listas de entrada/salida (lista I/O). En entrada se deben leer variables, en salida pueden escribirse expresiones. Si un array está en una lista I/O, se consideran todos los elementos del array en el orden de un almacenamiento del array. Una lista puede contener un DO implícito de variables.

Existen 3 formas de indicar el formato de los datos a leer o escribir:

- Sentencia `FORMAT` con etiqueta.
  - Sintaxis: `e FORMAT (codform)`
  - Acción: `codform` especifica los códigos de formato de lectura o escritura.
  - Normas: `e` es un número de etiqueta. Es una sentencia no ejecutable.
- Una expresión carácter que contiene el formato entre paréntesis.
- Un asteríctco * que indica formato libre (lista directa de entrada-salida).

Cada fichero **externo** (terminal, impresora, fichero en disco ó en cinta...) del qeu se lee o en el qeu se escribe lleva asociado un número de unidad no negativo, generalmente en el rango 1 a 99. Un número de unidad `u` asociado a un fichero externo puede ser:

- Una expresión entera con valor admisible (generalmente $1 \leq u \leq 99$).
- Un asterisco: entrada/salida estándar por defecto (generalmente teclado y pantalla).

Toda la sentencia de lectura o escritura en un fichero externo debe referirse explícitamente a su número de unidad asociado. Hay dos excepciones:

- Sentencia `READ` sin número de unidad.
  - Sintaxis: `READ` sin número de unidad.
  - Acción: lee datos del teclado (en modo interactivo). `fmt` indica el formato.
- Sentencia PRINT
  - Sintaxis: `PRINT fmt [,listavar]`
  - Escribe los datos en la pantalla (en modo interactivo) con el formato `fmt`.

## 1.6. Elaboración de programas

Es importante cuidar la elaboración del programa fuente y procurar satisfacer varios objetivos, entre otros: que sea claro y legible tanto para el autor del programa como para otros potenciales usuarios, que sea fácil de detectar errores, que sea eficiente en tiempo, precisión o memoria, que permita introducir cambios con facilidad, etc.

### 1.6.1. Estilo de programación

Algunos detalles que favorecen el estilo de programación son:

- Amplio uso de comentarios: incluir una breve descripción de algoritmos o procedimientos al principio de cada unidad de programa, en secciones de código diferencias, en límties de arrays, en sentencias que deberían cambiarse para ejecución con otros datos, etc.
- Descripción del significado de cada variable.
- Declaración organizada de variables (alfabética, por tipos, agrupada por similaridades, etc.).
- Líneas en blanco de separación entre secciones de código (blucles, bloques, `IF`,...) y entre subprogramas.
- Desplazamiento ("Identación") de las sentencais de estructuras (blucles, bloques, `IF`, `CASE`,`...`) unos espacios (6 espacios).

### 1.6.2. Depuración de errores

Los errores que pueden cometerse en la elaboración de un programa Fortran son de clase muy diversa: sintaxis, diseño de programa, programación, algorítmicos, instalación del software, errores de tamaño de memoria, etc. Una vez realizada una correcta instalción del software, los otros errores son imputables al usuario ó a limitaciones del software o hardware. La ley de MURPHY no falla cuando se aplica en programación. Algunos detalles que favorecen la detección y correción de errores son:

- Una redaccion clara con suficientes comentarios.
- Evitar estructuras de control, formatos y expresiones complicados.
- Si un programa es muy largo, conviene partirlo en subprogramas. Es difícil corregir un subprograma de más de unasd 300 líneas ejecutables.
- En primeras versiones de un programa, conviene incluir sentencias de escritura (a pantalla ó fichero) después de secciones diferenciadas de código con objeto de aislar posibles errores o comprobar el buen funcionamiento de partes de código.

### 1.6.3. Optimización de programas

Algunos detalles que afectan a la eficiencia de un programa son:

- Uso de la opción de compilación para optimizar la velocidad de ejecución.
- Potenciación `a**b`:

  - Si `b` es entero, tenemos que hasta `b`>5 la exponenciación se obtiene con *multiplicaciones*.
  - Si `b` es real, se calcula como `EXP(b*LOG(a))`.

- La raíz cuadrada es una operación rápida.
- Siempre que se pueda conviene ahorrar operaciones y simplificar fórmulas aunque las expresiones pueden ser numéricamente distintas.
- Si no hay peligro conviene reutilizar variables, vectores, y matrices. Si los elementos de un vector o matriz son conocidos, se pueden prescidir de ellos. El acceso a elementos de arrays consume tiempo.

# 2
# Introducción