

Notas Simulación en Física de materiales

Daniel Vazquez Lago

14 de septiembre de 2024

Índice general

Introducción	5
1. Introducción a Fortran	7
1.1. Estructura del programa	7
1.1.1. Formato código fuente	7
1.1.2. Tipos de datos	8
1.1.3. Operadores y expresiones	8
1.1.4. Entrada y salida estándar sin formato	9
1.1.5. Sentencias, PROGRAM, END	9
1.2. Sentencias de control	9
1.2.1. Sentencia CONTINUE	9
1.2.2. Sentencia STOP	9
1.2.3. Sentencia GOTO	10
1.2.4. Sentencia IF	10
1.2.5. Bloque IF-THEN-ENDIF	10
1.2.6. Bloque IF-THEN-ELSE-ENDIF	10
1.2.7. Bloque ELSE-IF	11
1.2.8. Selector SELECT CASE	11
1.2.9. Interacciones DO	12
1.2.10. DO ilimitado, EXIT y CYCLE	13
1.3. Utilidades de programa. Procedimientos.	13
1.3.1. Programa principal	13
1.3.2. Subprogramas externos	13
1.3.3. Uso no recursivo de subprogramas FUNCTION	13
1.3.4. Uso no recursivo de Subprogramas SUBROUTINE	14
1.3.5. Argumentos de subprogramas externos	15
1.3.6. Sentencia EXTERNAL	15
1.3.7. Sentencia INTRINSIC	16
1.3.8. Subprogramas internos	16
1.4. Procedimientos intrínsecos	16
1.5. Entrada y salida de datos. Ficheros. Formatos.	17
1.5.1. Elementos y clases de ficheros	17
1.6. Elaboración de programas	17
1.6.1. Estilo de programación	17
1.6.2. Depuración de errores	17
1.6.3. Optimización de programas	18
2. Introducción	19

Introducción

Usaremos $N = 500$ partículas 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.

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

“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 código fuente 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, algunas obligatorias, que todavía se mantienen, por lo que es importante mencionarlas. Estas son:

- Las sentencias de un programa se escriben en diferentes líneas.
- La posición de los caracteres 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 que 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 enteras y las demás 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:

<code>.EQ.</code>	<code>.NE.</code>	<code>.LT.</code>	<code>.LE.</code>	<code>.GT.</code>	<code>.GE.</code>
<code>==</code>	<code>/=</code>	<code><</code>	<code><=</code>	<code>></code>	<code>>=</code>

- Se pueden relacionar expresiones aritméticas con expresiones lógicas y expresiones de caracteres.
- Es recomendable utilizar paréntesis y/o 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. Sentencias de control

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

1.2.1. Sentencia 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. Sentencia 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 difícil 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:

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

La expresión **expres** debe ser escalar lógica. Si es verdadera se ejecutan 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 otros 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:

```
1  [nomb:] IF (expres) THEN
2      bloq1
3  ELSE [nomb]
4      bloq2
5  EDNDIF [nomb]
```

La expresión **expres** debe ser escalar lógica. Si es verdadera se ejecutan 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 otros nombres en la unidad de alcance en la que está el bloque **IF**.

1.2.7. Bloque ELSE-IF

La sitáxis es la siguiente:

```
1  [nomb:] IF (expres) THEN
2      bloq1
3  [ELSEIF (expres_i) THEN [nomb] b
4      bloq_i]...
5  [ELSE [nomb]
6      bloq2]
7  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 expresiones `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 este caso 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 puede 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.

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

1.2.8. Selector SELECT CASE

La sintaxis es la siguiente:

```
1  [nomb:] SELECT CASE (expres)
2  CASE (selector) [nomb]
3      bloq...
4  CASE DEFAULT [nomb]
5      bloq0...
6  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 que 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: $\text{MAX}\{ (\text{expres2} - \text{expres1} + \text{expres3}) / \text{expres3}, 0 \}$. Cuando $\{ \text{expres1} > \text{expres2} \text{ y } \text{expres3} > 0 \}$ ó $\{ \text{expres1} < \text{expres2} \text{ y } \text{expres3} < 0 \}$ 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 interacciones (calculado con sus valores iniciales).

1.2.10. DO ilimitado, EXIT y CYCLE

El **DO** ilimitado tiene la siguiente sintaxis:

```
1  [nomb:] Do
2      bloq
3  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:

```
1  PROGRAM [nombreprog]
2      [sentencias de especificacion]
3      [sentencias ejecutables]
4  CONTAINS
5      [subprogramas internos ]
6  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:

```

1  [tipo] FUNCTION nombfun ([argumentos ficticios])
2      [sentencias de especificacion]
3      [sentencias ejecutables]
4  END [FUNCTION [nombreprog]]

```

Con esto estaríamos definiendo el subprograma `FUNCTION nombfun`, invocándose con `nombfun([argumentos actuales])`, y substituyéndose 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

```

1  SUBROUTINE nombsubr ([argumentos ficticios])
2      [sentencias de especificacion]
3      [sentencias ejecutables]
4  END [SUBROUTINE [nombresubr]]

```

La llamada a la subrutina se realiza mediante la sentencia `CALL nombsubr [(argumentos actuales)]`, substituyéndose 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 subprogramas 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 subprograma, 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

```

```

6  PRINT*, y1, y2, s
7  END
8
9  SUBROUTINE ameba (x, n, y, f)
10     y = f(x, 5, n)
11 END
12
13 FUNCTION fun1 (x, i, j)
14     fun1 = x**(i-j)
15 END
16
17 FUNCTION fun2 (x, i, j)
18     fun2 = 3*x*(i-j)
19 END
20
21 FUNCTION sin (a, b, c, d)
22     sin = b/a + c/d
23 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.

```

1  INTRINSIC sin, cos, exp
2  a=3.141592; b=-a
3  r = fun (sin, cos, exp, a, b, 4)
4  PRINT*, r
5  END
6
7  FUNCTION fun (f1, f2, f3, a, b, n) ! fun = (sin(a)+cos(b)+
8  fun = (f1(a) + f2(b) + f3(a+b)) ** n ! exp(a+b))**n
9  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.

1.4. Procedimientos intrínsecos

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

- **Procedimientos elementales:** sus argumentos son escalares o arrays. 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**

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

1.5.1. Elementos y clases de ficheros

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 límites 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 similitudes, etc.).
- Líneas en blanco de separación entre secciones de código (bloques, bloques, IF,...) y entre subprogramas.
- Desplazamiento (“Identación”) de las sentencias de estructuras (bloques, 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 instalación 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 corrección de errores son:

- Una redacción 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 unas 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 $\text{EXP}(b * \text{LOG}(a))$.
- 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 prescindir de ellos. El acceso a elementos de arrays consume tiempo.

2

Introducción