Lecciones de `Fortran 90` para la asignatura Química Computacional

Curro Pérez Bernal <francisco.perez@dfaie.uhu.es>

$Id: clases_fortran.sgml,v 1.24 2013/07/02 09:38:58 curro Exp curro $

Resumen

El presente documento está basado en diversas referencias (ver 'Referencias' en la página 77) y es el esquema básico de las nociones de programación `Fortran` que se imparten en la asignatura optativa *Química Computacional* de cuarto curso del grado en Química en la Universidad de Huelva.

Índice general

Capítulo 1

Introducción

1.1. Objetivos

Los objetivos de esta clase son los siguientes:

1. dar una breve introducción a la programación y a las características de los lenguajes de programación.
2. indicar la importancia de poseer una idea clara y un esquema (diagrama de flujo) del algoritmo que se quiere programar.
3. dar una breve introducción a las características del lenguaje Fortran.
4. instalar el compilador Fortran de GNU (`gfortran`) en el ordenador. Indicar que unas opciones útiles para la compilación son "`gfortran -std=f95 -Wextra -Wall -pedantic`".
5. estudiar dos programas muy simples.
6. presentar posibles fuentes de información.

1.2. Puntos destacables.

Se trabaja con el editor `emacs` y con dos programas simples como ejemplo, 'Programa ejemplo_1_1.f90' en la página siguiente y 'Programa ejemplo_1_2.f90' en la página siguiente.

Se destaca, usando los ejemplos, las partes en las que se dividen un programa simple:

1. comienzo con la orden `PROGRAM` *nombre_programa*.
2. definición de variables.
3. cuerpo del programa con diferentes instrucciones y operaciones de `I/O` (entrada/salida).
4. fin del programa con la orden `END PROGRAM` *nombre_programa*.

Es importante hacer énfasis al explicar esto en los siguientes puntos

- la importancia de los comentarios (todo aquello en una línea tras el carácter `!`) y de un correcto sangrado para una fácil comprensión del programa. Indicar como ayuda en este apartado el uso del editor `emacs`.
- el porqué de la orden `IMPLICIT NONE`.
- Señalar las declaraciones de variables y, en su caso, la asignación inicial dada a las mismas.
- Señalar las operaciones de `I/O`.
- En el ejemplo 'Programa ejemplo_1_2.f90' en la página siguiente llamar la atención sobre la definición de la variable `N` y su asignación inicial `N = 3` como buena práctica de programación.

1.3. Programas usados como ejemplo.

1.3.1. Programa ejemplo_1_1.f90

```
PROGRAM ej_1_1
  !
  ! Este programa lee e imprime un nombre (cadena de caracteres)
  !
  IMPLICIT NONE
  CHARACTER(LEN=50) :: Nombre
  !
  PRINT *,' Escribe tu nombre entre comillas:'
  PRINT *,' (max 50 caracteres)'
  READ(*,*),Nombre
  PRINT *,Nombre
  !
END PROGRAM ej_1_1
```

1.3.2. Programa ejemplo_1_2.f90

```
PROGRAM ej_1_2
!
! Este programa lee tres números y calcula su suma y su media
!
  IMPLICIT NONE
  REAL :: N1, N2, N3, Average = 0.0, Total = 0.0
  INTEGER :: N = 3
  PRINT *,' Introduce tres numeros (separados por espacio o coma).'
  PRINT *,' '
  READ(*,*),N1,N2,N3
  Total =  N1 + N2 + N3
  Average = Total/N
  PRINT *,'La suma de los tres numeros es ',Total
  PRINT *,'Y su promedio es ',Average
END PROGRAM ej_1_2
```

Capítulo 2

Primeras operaciones básicas

2.1. Objetivos

Los objetivos de esta clase son los siguientes:

1 Enumerar las reglas básicas de la sintaxis de `Fortran` y el juego de caracteres usado en los programas.

2 La ejecución de operaciones básicas aritméticas y el orden en que se ejecutan los operadores aritméticos.

3 El uso de la orden `PARAMETER`.

4 Explicar los diferentes tipos que existen de variables numéricas y cómo definirlas.

2.2. Puntos destacables.

Reglas básicas de sintaxis:

- Número máximo de caracteres por línea de código: 132.
- Número máximo de caracteres en la definición de una variable: 31.
- '&' es el carácter de continuación de línea. Solo se coloca al final de la línea.[1]
- '!' es el carácter que marca inicio de comentario. Los comentarios tras el comando están permitidos.
- ';' es el carácter que permite separar dos (o más) comandos en una misma línea.

En la definición de variables puede usarse el guión bajo ('_') y mezclar números y letras, aunque el nombre de la variable no puede comenzar por un número.

Juego de caracteres usados en códigos `Fortran`

```
  A-Z  Letters              0-9  Digits
  _    Underscore                Blank
  =    Equal                +    Plus
  -    Minus                *    Asterisk
  /    Slash or oblique     '    Apostrophe
  (    Left parenthesis     )    Right parenthesis
  ,    Comma                .    Period or decimal point
  :    Colon                ;    Semicolon
  !    Exclamation mark     ""    Quotation mark
  %    Percent              &    Ampersand
  <    Less than            >    Greater than
```

[1] Salvo cuando se rompe en dos líneas una secuencia alfanumérica, tipo `CHARACTER`, en ese caso se coloca al final de la línea y al comienzo de la siguiente.

Jerarquía de operadores aritméticos:

- Operadores aritméticos: {+,−,*,/,**}.
- Jerarquía: (1) ** (dcha. a izqda.); (2) *,/ (depende del compilador); (3) +,− (depende del compilador).
- Atención al redondeo, a la mezcla de tipos diferentes de variables en funciones aritméticas. Se deben realizar las operaciones de modo que la pérdida de precisión se minimice.

  Cuando se realizan operaciones aritméticas, el procesador utiliza variables del mismo tipo, por lo que si las variables utilizadas son de diferente tipo una de ellas cambia de tipo para que sean ambos iguales. El orden de prioridad, de más bajo a más alto, es: `INTEGER`, `REAL`, `DOUBLE PRECISION` y `COMPLEX`, por lo que si se combina un número entero con uno real de doble precisión, el entero promociona a doble precisión y el resultado se da en doble precisión. Una vez terminada la operación u operaciones aritméticas, el tipo del resultado final se determina de acuerdo con el tipo de la variable a la que se asignará.
- Tamaño de enteros

  1 Entero 32 bits :: `(2**31)-1 = 2,147,483,647` (~ 10**9)

  2 Entero 64 bits :: `(2**63)-1 = 9,223,372,036,854,774,807` (~ 10**19)
- Tamaño y precisión de números reales

  1 Real 32 bits :: precisión = 6-9 ~ `0.3E38 - 1.7E38`

  2 Real 64 bits :: precisión = 15-18 ~ `0.5E308 - 0.8E308`
- Usando `PARAMETER` al definir una variable podemos definir constantes en los programas. En la definición pueden llevarse a cabo operaciones aritméticas simples. Ver 'Programa ejemplo_2_4.f90' en la página 6.
- (*) Presentar los diferentes tipos de enteros y de reales que existen en `Fortran` y las funciones intrínsecas [2] `KIND`, `EPSILON`, `PRECISION` y `HUGE` etc., así como la manera de definir variables dentro de los diferentes tipos que existen.

  En el caso de variables enteras, si queremos definir una variable llamada `i0` que pueda tomar, como mínimo, todos los valores posibles entre -999999 y 999999 debemos establecer una constante, llamada por ejemplo `ki`, usando la función intrínseca `SELECTED_INT_KIND()` y utilizarla en la definición de la variable

```
INTEGER, PARAMETER :: ki =  SELECTED_INT_KIND(6)
INTEGER(KIND=ki) :: i0
```

  La función intrínseca[3] `SELECTED_INT_KIND(X)` tiene como salida un entero que nos indica el tipo (*kind*) de una variable entera que es capaz de reproducir todos los números enteros en el rango `(-10E+X, 10E+X)` donde `X` es un escalar entero. De este modo si queremos que una constante entera en nuestro programa tenga esta precisión debemos hacer referencia a ella del siguiente modo:

```
-1234_ki
2354_ki
2_ki
```

  Si hubiera algún error al compilar con las órdenes anteriores y definir un entero, entonces la función `SELECTED_INT_KIND(X)` tendrá como salida `-1`.

  En el caso de variables reales hemos de usar una representación del número con decimales (*floating point representation*). Todas las que siguen son representaciones válidas de constantes reales en `Fortran`

```
-10.66E-20
0.66E10
1.
-0.4
1.32D-44
2E-12
3.141592653
```

[2] Ver 'Programa ejemplo_2_4.f90' en la página 6.

[3] La información acerca de la definición de funciones y las posibles funciones intrínsecas en `Fortran` puede encontrarse en 'Objetivos' en la página 39.

En este caso la orden para alterar el modo de almacenamiento de reales es usando la función `SELECTED_REAL_KIND(p=X,r=Y)`, con dos parámetros. Proporciona como salida un número entero asociado con el tipo (*kind*) de la variable definida. Permite definir a una variable real que cumpla con las siguientes condiciones:

- Que tenga una precisión que sea *al menos* `X` y con un rango de exponentes decimales que viene dado *al menos* por `Y`. Las etiquetas de los argumentos son opcionales.
- Si varios resultados son posibles se escogerá aquel con la menor precisión decimal.
- Al menos uno de los dos parámetros de entrada, `X` o `Y`, debe expresarse en la definición de la variable. Tanto `X` como `Y` son enteros. Si no existe una variable que se acomode a las condiciones exigidas la salida de `SELECTED_REAL_KIND(X,Y)` será `-1` si la precisión no alcanza el nivel impuesto, `-2` si es el exponente el que no lo alcanza y `-3` si ambos no son posibles.

Si por ejemplo queremos definir una variable real llamada `a0` que tenga una precisión de 15 decimales y con exponentes entre -306 y 307 lo podemos hacer con las ordenes

```
INTEGER, PARAMETER :: kr =  SELECTED_REAL_KIND(15,307)
REAL(KIND=kr) :: a0
```

Si queremos que los escalares reales que hemos mencionado anteriormente sean de este tipo haremos referencia a ellos como

```
-10.66E-20_kr
0.66E10_kr
142857._kr
-0.4_kr
2E-12_kr
3.141592653_kr
```

El programa 'Programa ejemplo_2_5.f90' en la página 7 permite ver cómo usar la orden `KIND`, que nos proporciona el valor de `KIND` asociado a un escalar o una variable dada. En este programa corto podemos ver el resultado por defecto de `KIND` obtenido para diferentes variables.

El programa 'Programa ejemplo_2_6.f90' en la página 7 se pueden encontrar ejemplos de los diferentes tipos, cómo definirlos y de las funciones que proporcionan información acerca de sus límites presentando las útiles funciones intrínsecas `KIND`, `DIGITS`, `EPSILON`, `TINY`, `HUGE`, `EXPONENT`, `MAXEXPONENT`, `MINEXPONENT`, `PRECISION` , `RADIX` y `RANGE`.

En este programa se definen las variables usando las funciones `SELECTED_INT_KIND` y `SELECTED_REAL_KIND` directamente en la definición. Es correcto aunque es mejor cuando se definen varias variables usar el procedimiento descrito en las notas.

Las funciones usadas dan la siguiente información:

1. `KIND(x)`: su salida es de tipo entero y proporciona el parámetro de tipo de la variable `x`.
2. `DIGITS(x)`: su salida es de tipo entero y proporciona el número de cifras significativas de `x`.
3. `EPSILON(x)`: si el argumento `x` es real su salida es otro real, con el mismo de tipo (*kind*) que `x`, y que es el número más pequeño de este tipo tal que `1.0 + EPSILON(X) >1`.
4. `TINY(x)`: si el argumento `x` es real su salida es un valor del mismo tipo que `x`, siendo el mínimo valor positivo definible para dicho tipo de variables.
5. `HUGE(x)`: si el argumento `x` es real o entero su salida es un valor del mismo tipo que `x`, siendo el máximo valor definible en dicho tipo.
6. `EXPONENT(x)`: exponente del valor `x`. Si `x = 0` entonces `EXPONENT(x)=0`.
7. `MAXEXPONENT(x)`: máximo exponente posible en el tipo del valor `x`.
8. `MINEXPONENT(x)`: mínimo exponente posible en el tipo del valor `x`.
9. `PRECISION(x)`: si el argumento `x` es real o complejo su salida es un valor entero que informa de la precisión decimal en el tipo de variable de `x`.
10. `RADIX(x)`: resultado entero e igual a la base del tipo de `x`.
11. `RANGE(x)`: resultado entero e igual al rango del exponente para el tipo de `x`.

- (*) Presentar como se pierde precisión al ejecutar cálculos y como indicar que una constante es de un tipo determinado para mejorar la precisión en los cálculos.

2.3. Programas usados como ejemplo.

2.3.1. Programa ejemplo_2_1.f90

```
PROGRAM ej_2_1
  IMPLICIT NONE
  !
  ! Ejemplo de programa que permite calcular la energía
  ! de un modo normal dados el número de cuantos (v), la
  ! frecuencia (w) y la anarmonicidad (wx) como
  !
  !  Ge(v) = we (v+1/2) - wexe (v+1/2)^2
  !
  !
  ! Definicion de variables
  REAL            :: energ_0, energ, delta_e ! deltae = energ-energ0
  REAL            :: we = 250.0, wexe = 0.25 ! Unidades: cm-1
  INTEGER         :: v = 0
  CHARACTER(LEN=60)    :: for_mol
  ! Operaciones I/O
  PRINT *,'Formula de la molecula : '
  READ(*,*), for_mol
  PRINT *,'Num. de quanta de excitacion : '
  READ(*,*), v
  ! Calculos
  energ = we*(v+0.5) - wexe*(v+0.5)**2
  energ_0 = we*(0.5) - wexe*(0.5)**2
  delta_e = energ - energ_0
  ! Operaciones I/O
  PRINT *
  PRINT *,'Especie molecular: ', for_mol
  PRINT *,'num. de quanta: ', v
  PRINT *,'energ = ',energ,'cm-1'
  PRINT *,'energ_0 = ',energ_0,'cm-1'
  PRINT *,'energ - energ_0 = ',delta_e,'cm-1'
END PROGRAM ej_2_1
```

2.3.2. Programa ejemplo_2_2.f90

```
PROGRAM ej_2_2
  IMPLICIT NONE
  REAL :: A,B,C
  INTEGER :: I
  A = 1.5
  B = 2.0
  C = A / B
  I = A / B
  PRINT *
  PRINT *, 'Caso (1), variable real'
  PRINT *,A,'/',B,' = ',C
  PRINT *, 'Caso (2), variable entera'
  PRINT *,A,'/',B,' = ',I
END PROGRAM ej_2_2
```

2.3.3. Programa ejemplo_2_3.f90

```
PROGRAM ej_2_3
  IMPLICIT NONE
  INTEGER :: I,J,K
  REAL :: Answer
  I = 5
  J = 2
  K = 4
  Answer = I / J * K
  PRINT *,'I = ',I
  PRINT *,'J = ',J
  PRINT *,'K = ',K
  PRINT *,'I / J * K = ',Answer
END PROGRAM ej_2_3
```

2.3.4. Programa ejemplo_2_4.f90

```
PROGRAM ej_2_4
  ! Programa para calcular el tiempo que tarda la luz en
  ! recorrer una distancia dada en unidades astronómicas.
```

```
  ! 1 UA = 1,50E11 m
  !
  !Definicion de variables
  IMPLICIT NONE
  ! u_a : unidad astronomica en km
  REAL , PARAMETER :: u_a=1.50*10.0**8
  ! a_luz : año luz --> distancia en km recorrida por la luz en un año.
  REAL , PARAMETER :: a_luz=9.46*10.0**12
  ! m_luz : minuto luz --> distancia en km recorrida por la luz en un minuto.
  REAL :: m_luz
  ! dist : distancia recorrida en UAs (INPUT)
  REAL :: dist
  ! t_min : tiempo en minutos necesario para recorrer la distancia Distance
  REAL :: t_min
  !
  ! min : parte entera de t_min
  ! seg : número de segundos (parte decimal de t_min)
  INTEGER :: min, seg
  !
  m_luz = a_luz/(365.25 * 24.0 * 60.0) ! Calculo del minuto-luz
  !
  PRINT *
  PRINT *,'Distancia en UAs'
  READ(*,*), dist
  PRINT *
  !
  t_min = (dist*u_a)/m_luz
  min = t_min; seg = (t_min - min) * 60
  !
  PRINT *,' La luz tarda ' , min,' minutos y ', seg,' segundos'
  Print *,' en recorrer una distancia de ',dist,' UA.'
END PROGRAM ej_2_4
```

2.3.5. Programa ejemplo_2_5.f90

```
PROGRAM ej_2_5
  INTEGER :: i
  REAL :: r
  CHARACTER :: c
  LOGICAL :: l
  COMPLEX :: cp
  PRINT *,' Integer ',KIND(i)
  PRINT *,' Real    ',KIND(r)
  PRINT *,' Char    ',KIND(c)
  PRINT *,' Logical ',KIND(l)
  PRINT *,' Complex ',KIND(cp)
END PROGRAM ej_2_5
```

2.3.6. Programa ejemplo_2_6.f90

```
PROGRAM ej_2_6
  ! From Program ch0806 of Chivers & Sleightholme
  !
  ! Examples of the use of the kind
  ! function and the numeric inquiry functions
  !
  ! Integer arithmetic
  !
  ! 32 bits is a common word size,
  ! and this leads quite cleanly
  ! to the following
  ! 8 bit integers
  ! -128 to 127 10**2
  ! 16 bit integers
  ! -32768 to 32767 10**4
  ! 32 bit integers
  ! -2147483648 to 2147483647 10**9
  !
  ! 64 bit integers are increasingly available.
  ! This leads to
  ! -9223372036854775808 to
  !  9223372036854775807 10**19
  !
  ! You may need to comment out some of the following
  ! depending on the hardware platform and compiler
  ! that you use.
  INTEGER                         :: I
  INTEGER ( SELECTED_INT_KIND( 2)) :: I1
  INTEGER ( SELECTED_INT_KIND( 4)) :: I2
  INTEGER ( SELECTED_INT_KIND( 8)) :: I3
  INTEGER ( SELECTED_INT_KIND(16)) :: I4
```

```
  ! Real arithmetic
  !
  ! 32 and 64 bit reals are normally available.
  !
  ! 32 bit reals 8 bit exponent, 24 bit mantissa
  !
  ! 64 bit reals 11 bit exponent 53 bit mantissa
  !
  REAL :: R = 1.0
  REAL ( SELECTED_REAL_KIND( 6, 37)) :: R1 = 1.0
  REAL ( SELECTED_REAL_KIND(15,307)) :: R2 = 1.0
  REAL ( SELECTED_REAL_KIND(18,310)) :: R3 = 1.0
  PRINT *,' '
  PRINT *,' Integer values'
  PRINT *,'         Kind       Huge'
  PRINT *,' '
  PRINT *,KIND(I ),' ',HUGE(I    )
  PRINT *,' '
  PRINT *,KIND(I1 ),' ',HUGE(I1  )
  PRINT *,KIND(I2 ),' ',HUGE(I2  )
  PRINT *,KIND(I3 ),' ',HUGE(I3  )
  PRINT *,KIND(I4 ),' ',HUGE(I4  )
  PRINT *,' '
  PRINT *,' ---------------------------------- '
  PRINT *,' '
  PRINT *,' Real values'
  !
  PRINT *,'         Kind     ', KIND(R ), '    Digits     ', DIGITS(R )
  PRINT *,'  Huge    = ',HUGE(R ), ' Tiny =', TINY(R)
  PRINT *,'  Epsilon = ',EPSILON(R),' Precision = ', PRECISION(R)
  PRINT *,'  Exponent = ',EXPONENT(R), 'MAXExponent = ', MAXEXPONENT(R), '  MINExponent = ',MINEXPONENT(R)
  PRINT *,'  Radix    = ', RADIX(R ), ' Range =', RANGE(R)
  PRINT *,' '
  !
  !
  PRINT *,'         Kind     ', KIND(R1 ), '    Digits     ', DIGITS(R1 )
  PRINT *,'  Huge    = ',HUGE(R1 ), ' Tiny =', TINY(R1)
  PRINT *,'  Epsilon = ',EPSILON(R1),' Precision = ', PRECISION(R1)
  PRINT *,'  Exponent = ',EXPONENT(R1), 'MAXExponent = ', MAXEXPONENT(R1), '  MINExponent = ',MINEXPONENT(R1)
  PRINT *,'  Radix    = ', RADIX(R1 ), ' Range =', RANGE(R1)
  PRINT *,' '
  !
  !
  PRINT *,'         Kind     ', KIND(R2 ), '    Digits     ', DIGITS(R2 )
  PRINT *,'  Huge    = ',HUGE(R2 ), ' Tiny =', TINY(R2)
  PRINT *,'  Epsilon = ',EPSILON(R2),' Precision = ', PRECISION(R2)
  PRINT *,'  Exponent = ',EXPONENT(R2), 'MAXExponent = ', MAXEXPONENT(R2), '  MINExponent = ',MINEXPONENT(R2)
  PRINT *,'  Radix    = ', RADIX(R2 ), ' Range =', RANGE(R2)
  PRINT *,' '
  !
  !
  PRINT *,'         Kind     ', KIND(R3 ), '    Digits     ', DIGITS(R3 )
  PRINT *,'  Huge    = ',HUGE(R3 ), ' Tiny =', TINY(R3)
  PRINT *,'  Epsilon = ',EPSILON(R3),' Precision = ', PRECISION(R3)
  PRINT *,'  Exponent = ',EXPONENT(R3), 'MAXExponent = ', MAXEXPONENT(R3), '  MINExponent = ',MINEXPONENT(R3)
  PRINT *,'  Radix    = ', RADIX(R3 ), ' Range =', RANGE(R3)
  PRINT *,' '
  !
END PROGRAM ej_2_6
```

Capítulo 3

Matrices o *Arrays* (Básico)

3.1. Objetivos

Los objetivos de esta clase son los siguientes:

1 presentar las matrices o arreglos de una y varias dimensiones como estructuras de datos en `Fortran`.

2 presentar cómo se define una matriz y cómo se accede a sus elementos.

3 presentar la sintaxis de los bucles `DO` como estructura de control. `DO` implícito. Uso de esta estructura con matrices.

4 Definición dinámica del tamaño de arreglos.

5 Arreglos multidimensionales.

3.2. Puntos destacables.

Conceptos previos:

1 rango *(rank)*: número de índices necesarios para indicar un elemento de la matriz.

2 límites *(bounds)*: valor superior e inferior del índice que indica los elementos de la matriz en cada dimensión.

3 extensión *(extent)*: número de elementos de la matriz para cada dimensión.

4 tamaño *(size)*: número total de elementos de la matriz.

5 Se dice que dos arreglos son "conformables" si tienen idéntico rango y extensión.

Es importante al explicar este material hacer énfasis en los siguientes puntos

- definición de una matriz monodimensional (vector) y uso de la estructura de control `DO` presentado en 'Programa ejemplo_3_1.f90' en la página siguiente. (ejercicio 2_1)
- es recomendable definir las dimensiones de las matrices (en caso de dimensionamiento estático) como variables (o parámetros) lo que facilita posteriores cambios.
- es importante siempre inicializar las matrices que se definan. La inicialización puede llevarse a cabo en el momento en el que la matriz se defina, mediante `READ` o usando asignaciones. Es muy simple inicializar todos los elementos de una matriz a un mismo valor: `vec = valor`. Una matriz también puede definirse e inicializarse usando los llamados *array constructors*. Por ejemplo, para definir una matriz entera `vec_int`con seis elementos puede usarse cualquiera de las tres opciones siguientes

```
do i = 0, 5
   vec_int(i) = 2*i
enddo
vec_int = (/(2*i, i = 0, 5)/)
vec_int = (/0,2,4,6,8,10/)
```

Las dos últimas opciones implican *array constructors* y pueden llevarse a cabo al definir la variable.

- como se define el tamaño de una matriz definida con el atributo `ALLOCATABLE` al correr el programa usando la función `ALLOCATE` tal y como se aplica en el ejemplo 'Programa ejemplo_3_2.f90' en esta página. En el caso de `ALLOCATE` es posible añadir una opción `STAT = var` tal que si la definición del arreglo ha tenido éxito entonces `var = 0`. Un ejemplo de esta opción se muestra en el 'Programa ejemplo_9_3.f90' en la página 51.
- presentar arreglos de datos con varias dimensiones (hasta un máximo de siete son posibles) y la forma del `DO` implícito. Ver ejemplo 'Programa ejemplo_3_3.f90' en la página siguiente.
- explicar la forma más general de la estructura `DO` y de la definición de matrices utilizando índices negativos. Ver ejemplo 'Programa ejemplo_3_4.f90' en la página siguiente.
- explicar como combinar el redireccionado de la `bash shell` con programas `Fortran`. Necesario para ejercicio 2, se explica en 'Matrices o *Arrays* (Avanzado)' en la página 13.

3.3. Programas usados como ejemplo.

3.3.1. Programa ejemplo_3_1.f90

```
PROGRAM ej_3_1
!
! DEFINICIÓN DE VARIABLES
  IMPLICIT NONE
  REAL :: Total=0.0, Promedio=0.0
  INTEGER, PARAMETER :: semana=7
  REAL , DIMENSION(1:semana) :: Horas_trab
  INTEGER :: dia
!
  PRINT *,' Introduzca las horas trabajadas'
  PRINT *,' por dia en una semana'
  DO dia=1,semana
     READ(*,*), Horas_trab(dia)
  ENDDO
!
  DO dia=1,semana
     Total = Total + Horas_trab(dia)
  ENDDO
  Promedio = Total / semana
!
  PRINT *,' Promedio de horas de trabajo semanales: '
  PRINT *, Promedio
END PROGRAM ej_3_1
```

3.3.2. Programa ejemplo_3_2.f90

```
PROGRAM ej_3_2
!
! DEFINICION DE VARIABLES
  IMPLICIT NONE
  REAL :: Total=0.0, Promedio=0.0
  REAL , DIMENSION(:), ALLOCATABLE :: Horas_trab
  INTEGER :: dia, num_dias
!
  PRINT *,' Introduzca el número de dias para los que '
  PRINT *,' se va a calcular el promedio de horas trabajadas.'
  READ(*,*), num_dias
!
  ALLOCATE(Horas_trab(1:num_dias))
!
  PRINT *,' Introduzca las horas trabajadas'
  PRINT *,' por dia en ', num_dias, 'dias.'
  DO dia=1, num_dias
     READ(*,*), Horas_trab(dia)
  ENDDO
!
```

```
  DO dia=1,num_dias
     Total = Total + Horas_trab(dia)
  ENDDO
  Promedio = Total / num_dias
!
  PRINT *,' Promedio de horas de trabajo en ',num_dias, ' dias : '
  PRINT *, Promedio
!
END PROGRAM ej_3_2
```

3.3.3. Programa ejemplo_3_3.f90

```
PROGRAM asistencia
  IMPLICIT NONE
  INTEGER , PARAMETER :: N_alum=4
  INTEGER , PARAMETER :: N_asig=3
  INTEGER , PARAMETER :: N_prac=3
  INTEGER :: alumno, asignatura, pract
  CHARACTER(LEN = 2) , DIMENSION(1:N_prac,1:N_asig,1:N_alum) :: asiste='NO'
  DO alumno = 1,N_alum
     DO asignatura = 1,N_asig
        READ(*,*),(asiste(pract,asignatura,alumno),pract = 1, N_prac)
     ENDDO
  ENDDO
  PRINT *,' Asistencia a practicas : '
  DO alumno=1, N_alum
     PRINT *,' Alumno = ', alumno
     DO asignatura = 1,N_asig
        PRINT *,'   Asignatura = ', asignatura, ' : ', (asiste(pract,asignatura,alumno),pract=1,N_prac)
    ENDDO
 ENDDO
END PROGRAM asistencia
```

3.3.4. Programa ejemplo_3_4.f90

```
PROGRAM longitude
  IMPLICIT NONE
  REAL , DIMENSION(-180:180) :: Time=0
  INTEGER :: Degree, Strip
  REAL :: Value
!
  DO Degree=-165,165,15
     Value=Degree/15
     DO Strip=-7,7
        Time(Degree+Strip)=Value
     ENDDO
  ENDDO
!
  DO Strip=0,7
     Time(-180 + Strip) = -180/15
     Time( 180 - Strip) = 180/15
  ENDDO
!
  DO Degree=-180,180
     PRINT *,Degree,' ',Time(Degree), 12 + Time(Degree)
  END DO
END PROGRAM longitude
```

Capítulo 4

Matrices o *Arrays* (Avanzado)

4.1. Objetivos

Los objetivos de esta clase son los siguientes:

1 presentar el orden natural en el que se almacenan los elementos en matrices multidimensionales.

2 presentar como se pueden manipular matrices completas y secciones de matrices en `Fortran`.

3 ver como se puede trabajar con parte de una matriz.

4 explicar la definición de matrices con la orden `WHERE`.

4.2. Puntos destacables.

- Orden de almacenamiento

  El orden en el que se almacenan los elementos de una matriz multidimensional en `Fortran` es tal que el primer subíndice varía más rápidamente, a continuación el segundo, a continuación el tercero y así sucesivamente. Es lo que se conoce como *column major order*.

  Por ejemplo, si definimos una matriz con dos subíndices y de dimensión 4x2 con la orden

  `REAL , DIMENSION(1:4,1:2) :: A,`

  Esta matriz `A` tiene ocho elementos que se guardan en memoria siguiendo el orden

  ```
  A(1,1), A(2,1), A(3,1), A(4,1), A(1,2), A(2,2), A(3,2), A(4,2)
  ```

  Por tanto para inicializar la matriz `A` podemos hacerlo de diferentes maneras. Supongamos que queremos inicializar la matriz de modo que cada elemento sea igual al número de su fila. Podemos usar dos bucles[1]

  ```
  DO I_col = 1, 2
       DO I_row = 1, 4
          A(I_row, I_col) = I_row
       ENDDO
  ENDDO
  ```

  Podemos también hacer uso de un *array constructor*, aunque la solución que parece más lógica, haciendo

  ```
  A = (/ 1, 2, 3, 4, 1, 2, 3, 4 /)
  ```

[1] Debido al almacenamiento en memoria de acuerdo con el *column major order* el orden de los bucles haciendo que las columnas corran antes que las filas puede mejorar el rendimiento del código en matrices de tamaño considerable.

no funciona. El *array constructor* produce un vector de dimensión ocho, y no una matriz 4x2. Este vector y la matriz `A` tiene idéntico tamaño, pero no son conformables. Para resolver este problema es preciso hacer uso de la orden `RESHAPE`. La sintaxis de esta orden es la siguiente

```
output_array = RESHAPE(array_1, array_2)
```

Donde *array_1* contiene es una matriz los datos que van a sufrir el cambio, *array_2* es un vector que describe las dimensiones de la nueva matriz *output_array*. El número de elementos, esto es, el tamaño, de las matrices *array_1* y *output_array* debe ser idéntico. En el caso anterior podemos por tanto hacer

```
A = RESHAPE( (/ 1, 2, 3, 4, 1, 2, 3, 4 /), (/ 4, 2 /) )
```

En 'Programa ejemplo_4_3.f90' en la página 17 se puede ver otro ejemplo. El comando `RESHAPE` puede usarse en la declaración del arreglo

```
INTEGER, DIMENSION(1:4,1:2) :: A = &
       RESHAPE( (/ 1, 2, 3, 4, 1, 2, 3, 4 /), (/ 4, 2 /) )
```

El orden seguido en el almacenamiento de los datos es especialmente importante en las operaciones de entrada y salida. Si ejecutamos

```
PRINT*, A
```

El resultado será

```
A(1,1), A(2,1), A(3,1), A(4,1), A(1,2), A(2,2), A(3,2), A(4,2)
```

Es preciso tener esto en cuenta al utilizar `READ` para leer los elementos de un arreglo en un fichero, haciendo `READ(unit,*) A`. Para cambiar el orden de lectura es posible hacer uso de *implicit* `DO`'s

```
READ(unit,*) ( ( A(row,col), col = 1, 2 ), row = 1, 4 )
```

- En `FORTRAN` es posible definir arreglos multidimensionales más allá de las matrices, siendo 7 el límite del número de índices y corriendo más rápidamente el primero, luego el segundo, y así sucesivamente hasta el último. En el ejemplo 'Programa ejemplo_4_2.f90' en la página 17 se caracteriza de forma completa un arreglo haciendo uso de varias de funciones de tipo. *inquiry* (ver 'Programa ejemplo_8_2.f90' en la página 41).
- Es importante al explicar el material de esta clase hacer énfasis en los siguientes puntos que afectan al manejo de matrices completas. Si por ejemplo definimos los vectores reales `V1`, `V2`, `V3` y `V4` como

```
REAL , DIMENSION(1:21) :: V1, V2
REAL , DIMENSION(-10:10) :: V3
REAL , DIMENSION(0:10) :: V4
```

Podemos trabajar de forma natural con estos vectores completos como en los siguientes ejemplos[2].

1 Asignación de un valor:

```
V1 = 0.5
```

Hacemos que todos los elementos de `V1` sean iguales a `0.5`.

2 Igualar matrices:

```
V1 = V2
```

Hacemos que cada elemento de `V1` sea igual al elemento equivalente de `V2`. Esta función sólo es aplicable a matrices *conformables*[3]. Por ejemplo, también es válido hacer

```
V3 = V2
```

pero *no* es válido

[2] Esta es una de las grandes ventajas de `Fortran90` frente a `Fortran77`.
[3] Se dice que dos matrices son "conformables" si tienen idéntico número de dimensiones (subíndices) y cada una de ellas tiene la misma longitud.

```
V1 = V4
```

3 Todas las operaciones aritméticas aplicables a escalares se pueden asimismo aplicar a matrices, siempre que estas sean conformables, aunque hay que tener claro qué implican pues es posible que matemáticamente no tengan sentido. Usando los mismos vectores que en el punto anterior podemos hacer

```
V1 = V2 + V3
V1 = V2*V3
```

En el primer caso `V1` es la suma de los dos vectores, pero en el segundo `V1` es un vector, cada uno de sus elementos es el producto de los elementos correspondientes de `V2` y `V3`. Esto *no* es el producto escalar de dos vectores. Como hemos indicado, este caso de nuevo las matrices han de ser conformables. Para matrices bidimensionales, por ejemplo, si definimos las matrices

```
REAL , DIMENSION(1:4,1:4) :: A, B, C
```

Pueden hacerse operaciones como las siguientes

```
A = A**0.5
C = A + B
C = A * B
```

Es importante tener en cuenta que la última operación no es el producto de dos matrices, sino otra matriz, de idéntica dimensión a las de `A` y `B`, y cada uno de sus elementos es el producto de los correspondientes elementos de `A` y `B`.

4 También pueden leerse todos los elementos de una matriz sin necesidad de un bucle `DO`, como en el ejemplo 'Programa ejemplo_4_1.f90' en la página siguiente. En este ejemplo también se presenta la función `SUM` que suma los elementos de un vector en la aplicación presentada, aunque tiene una aplicación más general.

- También es posible referirnos a una sección de una matriz, por ejemplo, usando las matrices definidas arriba podemos hacer

```
V1(1:10) = 0.5
B (1,1:4) = 100.0
```

En el primer caso se hacen iguales a `0.5` los diez primeros elementos de `V1`, mientras que en el segundo se hacen iguales a `100.0` los elementos de la primer fila de `A`. Puede verse una aplicación de este punto en el ejemplo 'Programa ejemplo_4_1.f90' en la página siguiente.

La forma más general como se define una sección de un arreglo de datos es como *`liminf:limsup:step`*: la sección comienza con el índice *liminf*, termina en *limsup* y *step* es el incremento en la variable que marca el índice. Cuando *step* no está presente, como en los casos anteriores, toma por defecto el valor 1. Algunos ejemplos:

```
V1(:)       ! todo el vector
V1(3:10)    ! elementos V1(3), V1(4), ... ,   V1(10)
V1(3:10:1)  !     ""      ""     ""             ""
V1(3:10:2)  !     ""     V1(3), V1(5), ... , V1(9)
V1(m:n)     ! elementos V1(m), V1(m+1), ... , V1(n)
V1(9:4:-2)  !     ""     V1(9), V1(7), V1(5)
V1(m:n:-k)  ! elementos V1(m), V1(m-k), ... , V1(n)
V1(::2)     !     ""     V1(1), V1(3), ... , V1(21)
V1(m:m)     ! Arreglo 1x1
V1(m)       ! Escalar
```

Es importante diferenciar los dos últimos casos.

- Al asignar valores a un arreglo puede utilizarse una *máscara lógica*, haciendo uso de la orden `WHERE`. El uso de una máscara lógica permite seleccionar los elementos del arreglo sobre los que se llevará a cabo una operación. Si, por ejemplo, queremos calcular la raíz cuadrada de los elementos de un arreglo real llamado `data_mat` y almacenarlos en el arreglo `sq_data_mat`, debemos evitar tomar raíces cuadradas de números negativos. Para ello podemos usar una combinación de bucles y condicionales

```
DO j_col = 1, dim_2
   DO i_row = 1, dim_1
         IF ( data_mat(i_row, j_col) >= 0.0 ) THEN
              sq_data_mat(i_row, j_col) = SQRT( data_mat(i_row, j_col) )
         ELSE
              sq_data_mat(i_row, j_col) = -99999.0
         ENDIF
   ENDDO
ENDDO
```

El uso de `WHERE` simplifica esta tarea. La forma general de esta orden es

```
[name:] WHERE (mask_expr_1)
   ....
   Array assignment block 1
   ....
ELSEWHERE (mask_expr_2) [name]
   ....
   Array assignment block 2
   ....
ELSEWHERE
   ....
   Array assignment block 3
   ....
ENDWHERE [name]
```

Donde *mask_expr_1* y *mask_expr_2* son arreglos lógicos conformables con la matriz que se asigna. El ejemplo anterior se simplifica usando `WHERE`

```
WHERE ( data_mat >= 0.0 )
            sq_data_mat = SQRT( data_mat )
ELSEWHERE
            sq_data_mat = -99999.0
ENDWHERE
```

- Estos puntos se tratan en los diferentes ejemplos proporcionados. En el ejemplo 'Programa ejemplo_4_3.f90' en la página siguiente se puede ver como se inicializan vectores y para matrices. En este último caso haciendo uso de la orden `RESHAPE`. Este ejemplo presenta también las funciones `Fortran DOT_PRODUCT` (producto escalar) y `MATMUL` (multiplicación de matrices).

  En el ejemplo 'Programa ejemplo_4_4.f90' en la página siguiente se presenta como puede asignarse una matrix usando `WHERE`, y una "máscara" lógica.

  Como se muestra en el ejemplo 'Programa ejemplo_4_5.f90' en la página 18 es importante no confundir los resultados esperados al vectorizar un programa eliminando los bucles `DO` presentes en él.

  En el ejemplo 'Programa ejemplo_4_6.f90' en la página 18 vemos como usar la orden `RESHAPE` en la definición de una matriz y como acceder a diferentes partes de la misma.

4.3. Programas usados como ejemplo.

4.3.1. Programa ejemplo_4_1.f90

```
PROGRAM ej_4_1
  !
  ! DEFINICION DE VARIABLES
  IMPLICIT NONE
  REAL :: Total=0.0, Promedio=0.0
  REAL , DIMENSION(:), ALLOCATABLE :: t_trab
  ! Factor que corrige el tiempo trabajado los dos últimos días
  REAL :: Fac_correc=1.05
  INTEGER :: dia, num_dias
  !
  PRINT *,' Introduzca el no. de dias para los que se va '
  PRINT *,' a calcular el promedio de horas y minutos trabajados.'
  READ(*,*), num_dias
  ! Dimensiona dinámicamente la matriz
  ALLOCATE(t_trab(1:num_dias))
  !
  PRINT *,' Introduzca las horas trabajadas'
  PRINT *,' por dia en ', num_dias, 'dias.'
  ! Lectura de datos
  READ(*,*), t_trab
  !
  t_trab(num_dias-1:num_dias) = Fac_correc*t_trab(num_dias-1:num_dias)
  !
  !
  Total = SUM(t_trab)
  !
  Promedio = Total / num_dias
  !
  PRINT *,' Horas de trabajo en ',num_dias, ' dias : '
  PRINT *, Promedio
  !
END PROGRAM ej_4_1
```

4.3.2. Programa ejemplo_4_2.f90

```
PROGRAM EJEMPLO_4_2
  !
  ! Program to characterize an array making use of inquiry functions
  !
  IMPLICIT NONE
  !
  REAL, DIMENSION(:,:), ALLOCATABLE :: X_grid
  INTEGER :: Ierr
  !
  !
  ALLOCATE(X_grid(-20:20,0:50), STAT = Ierr)
  IF (Ierr /= 0) THEN
     STOP 'X_grid allocation failed'
  ENDIF
  !
  WRITE(*, 100) SHAPE(X_grid)
100 FORMAT(1X, "Shape :        ", 7I7)
  !
  WRITE(*, 110) SIZE(X_grid)
110 FORMAT(1X, "Size :         ", I7)
  !
  WRITE(*, 120) LBOUND(X_grid)
120 FORMAT(1X, "Lower bounds : ", 7I6)
  !
  WRITE(*, 130) UBOUND(X_grid)
130 FORMAT(1X, "Upper bounds : ", 7I6)
  !
  DEALLOCATE(X_grid, STAT = Ierr)
  IF (Ierr /= 0) THEN
     STOP 'X_grid deallocation failed'
  ENDIF
  !
END PROGRAM EJEMPLO_4_2
```

4.3.3. Programa ejemplo_4_3.f90

```
PROGRAM ej_4_3
  !
  ! DEFINICION DE VARIABLES
  IMPLICIT NONE
  REAL, DIMENSION(1:5) :: VA = (/1.2,2.3,3.4,4.5,5.6/), PMAT
  INTEGER I
  INTEGER, DIMENSION(1:5) :: VB = (/(2*I,I=1,5)/)
  REAL :: PE
  REAL , DIMENSION(1:5,1:5) :: MC
  REAL , DIMENSION(25) :: VC = &
       (/ 0.0,0.0,0.0,0.0,1.0,0.5,2.0,3.2,0.0,0.0, &
          0.0,0.0,0.0,0.0,11.0,0.5,2.3,3.2,0.0,0.0, &
          1.0,3.0,-2.0,-2.0,-0.6 /)
  ! Producto escalar VA.VB
  PE = DOT_PRODUCT(VA,VB)
  !
  PRINT *, 'PRODUCTO ESCALAR (VA,VB) = ', PE
  !
  ! Producto de matrices VAxMC
  ! Haciendo RESHAPE de VC hacemos que sea una matriz 5 x 5
  MC = RESHAPE(VC,(/ 5, 5 /))
  PMAT = MATMUL(VA,MC)
  !
  PRINT *, 'PRODUCTO VAxMC = ', PMAT(1:5)
  !
END PROGRAM ej_4_3
```

4.3.4. Programa ejemplo_4_4.f90

```
PROGRAM long2
  IMPLICIT NONE
  REAL , DIMENSION(-180:180) :: Time=0
  INTEGER :: Degree, Strip
  REAL :: Value
  CHARACTER (LEN=1), DIMENSION(-180:180) :: LEW=' '
!
  DO Degree=-165,165,15
     Value=Degree/15
     DO Strip=-7,7
        Time(Degree+Strip)=Value
     ENDDO
  ENDDO
```

```
!
  DO Strip=0,7
     Time(-180 + Strip) = -180/15
     Time( 180 - Strip) = 180/15
  ENDDO
!
  DO Degree=-180,180
     PRINT *,Degree,' ',Time(Degree), 12 + Time(Degree)
  END DO
!
  WHERE (Time > 0)
     LEW='E'
  ELSEWHERE (Time < 0)
     LEW='W'
  ENDWHERE
!
  PRINT*, LEW
!
END PROGRAM long2
```

4.3.5. Programa ejemplo_4_5.f90

```
PROGRAM ej_4_5
  !
  ! DEFINICION DE VARIABLES
  IMPLICIT NONE
  REAL, DIMENSION(1:7) :: VA = (/1.2,2.3,3.4,4.5,5.6,6.7,7.8/)
  REAL, DIMENSION(1:7) :: VA1 = 0.0, VA2 = 0.0
  INTEGER I
  !
  VA1 = VA
  VA2 = VA
  !
  DO I = 2, 7
     VA1(I) = VA1(I) + VA1(I-1)
  ENDDO
  !
  VA2(2:7) = VA2(2:7) + VA2(1:6)
  !
  ! The previous two operations with VA1 and VA2 seem that
  ! should provide the same result. Which is not the case.
  PRINT*, VA1
  PRINT*, VA2
  !
  ! To obtain the same effect without an explicit DO loop we can do
  ! the following
  VA2 = VA
  VA2(2:7) = (/ (SUM(VA2(1:I)), I = 2,7) /)
  !
  PRINT*, VA1
  PRINT*, VA2
END PROGRAM ej_4_5
```

4.3.6. Programa ejemplo_4_6.f90

```
PROGRAM ej_4_6
  !
  ! DEFINICION DE VARIABLES
  IMPLICIT NONE
  INTEGER, DIMENSION(1:3,1:3) :: A = RESHAPE( (/ 1,2,3,4,5,6,7,8,9 /), (/ 3,3 /) )
  !
  !
  !      1  4  7
  !  A = 2  5  8
  !      3  6  9
  !
  PRINT*, "Elemento de la matriz", A(2,3)
  PRINT*, "Submatriz", A(1:2,2:3)
  PRINT*, "Submatriz", A(::2,::2)
  PRINT*, "Columna de la matriz", A(:,3)
  PRINT*, "Fila de la matriz", A(2,:)
  PRINT*, "Matriz completa", A
  PRINT*, "Matriz traspuesta", TRANSPOSE(A)
END PROGRAM ej_4_6
```

Capítulo 5

Estructuras de control

5.1. Objetivos

Los objetivos de esta clase son los siguientes:

1 presentar las diferentes estructuras de control condicionales en `Fortran` (*branching*).

2 presentar las diferentes formas de ejecutar bucles en programas `Fortran` (*loops*).

Estas estructuras de control son básicas para dominar el flujo de un programa, permitiendo que la ejecución del programa pueda depender de los datos implicados o de la decisiones del usuario que lo corra.

Es importante tener en cuenta que para cualquier proyecto medianamente complejo es preciso tener una idea clara del problema, de los inputs y outputs del mismo y del algoritmo que se utilizará, así como de la estructura del programa que se llevará a cabo como un diagrama de flujo o como pseudocódigo antes de comenzar a escribir el código `Fortran`.

Los problemas complejos deben dividirse en tareas más simples, y si es necesario estas se subdividirán en otras tareas aún más simples (*top-down design*) y cada una de estas tareas debe codificarse y comprobarse de forma independiente.

5.2. Puntos destacables.

Daremos de forma esquemática el formato que tienen las estructuras de control. En primer lugar las condicionales y en segundo lugar los bucles.

- Estructuras condicionales.

  Todas estas estructuras dependen de la evaluación de condiciones lógicas. Estas condiciones se establecen mediante los operadores relacionales siguientes:

  - == Ser igual a.
  - /= No ser igual a.
  - > Mayor que.
  - < Menor que.
  - >= Mayor o igual que.
  - <= Menor o igual que.

  También es posible utilizar operadores lógicos para combinar diferentes condiciones. Los operadores lógicos son

  - `.AND.` Operador Y lógico.
  - `.OR.` Operador O lógico.

- `.NOT.` Operador NO lógico.
- `.EQV.` Operador '==' lógico.
- `.NEQV.` Operador '/=' lógico.

Los diferentes condicionales posibles son los siguientes.

1 `IF THEN ENDIF`
La forma de este condicional es la siguiente

```
.
. codigo
.
IF (cond logica) THEN
        .
        . bloque_1
        .
ENDIF
.
. codigo
.
```

Solamente si se cumple la condición lógica impuesta se ejecutan las instrucciones en `bloque_1`, si no se cumple son ignoradas.
En caso de que sea únicamente una la instrucción ejecutada si se cumple la condición lógica se puede simplificar la orden eliminando `THEN` y `ENDIF` del siguiente modo

```
.
. codigo
.
IF (cond logica) instruccion
.
. codigo
.
```

2 `IF THEN ELSE ENDIF`
La forma de este condicional es la siguiente

```
.
. codigo
.
IF (cond logica) THEN
        .
        . bloque_1
        .
ELSE
        .
        . bloque_2
        .
ENDIF
.
. codigo
.
```

Si se cumple la condición lógica impuesta se ejecutan las instrucciones en `bloque_1`, si no se cumple se ejecuta el bloque `bloque_2`.

3 `IF THEN ELSE IF ENDIF`
La forma de este condicional es la siguiente

```
.
. codigo
.
IF (cond logica_1) THEN
        .
        . bloque_1
        .
ELSE IF (cond logica_2) THEN
        .
        . bloque_2
        .
ENDIF
.
. codigo
.
```

Si se cumple la condición lógica `cond logica_1` se ejecutan las instrucciones en `bloque_1`; si no se cumple, pero sí se cumple la condición lógica `cond logica_2`, entonces se ejecuta el bloque `bloque_2`.

4 `IF THEN ELSE IF ELSE ENDIF`

La forma de este condicional es la siguiente

```
.
. codigo
.
IF (cond logica_1) THEN
         .
         . bloque_1
         .
ELSE IF (cond logica_2) THEN
         .
         . bloque_2
         .
ELSE
         .
         . bloque_3
         .
ENDIF
.
. codigo
.
```

Si se cumple la condición lógica `cond logica_1` se ejecutan las instrucciones del `bloque_1`; si no se cumple, pero sí se cumple la condición lógica `cond logica_2`, entonces se ejecuta el bloque `bloque_2`. Si ninguna de las dos condiciones lógicas son ciertas, entonces se ejecuta el código en el bloque `bloque_3`.

5 `SELECT CASE`

La orden `CASE` permite escoger una opción entre diferentes posibilidades de forma más eficiente, clara y elegante que las estructuras de control anteriores.

La forma de este condicional es la siguiente

```
SELECT CASE (selector)
   CASE (label-1)
      bloque-1
   CASE (label-2)
      bloque-2
   CASE (label-3)
      bloque-3
     ............
   CASE (label-n)
      bloque-n
   CASE DEFAULT
      bloque-default
END SELECT
```

En esta orden `selector` es una variable o una expresión cuyo resultado es del tipo *entero*, *lógico* o *carácter*. En concreto no puede ser una variable ni una expresión que dé como resultado un número real.

En esta orden `label-1` hasta `label-n` son listas de etiquetas, separadas por comas, que deben tener una de las siguientes formas:

```
valor
valor-1 : valor-2
valor-1 :
 : valor-2
```

La primera indica el valor `valor`, la segunda un valor comprendido entre `valor-1` y `valor-2`, la tercer un valor mayor que `valor-1`y la última un valor menor que `valor-2`. Tanto `valor` como `valor-1` y `valor-2` deben ser constantes o alias definidos por una orden `PARAMETER`.

Como se ejecuta esta orden es evaluando `selector`. Se compara el resultado con los valores establecidos en cada una de las listas de etiquetas, ejecutando el bloque correspondiente a la primera de las comparaciones que resulte exitosa. Si ninguna comparación es exitosa se ejecuta -caso de que exista- el `bloque-default`. Tras ello se prosigue ejecutando las órdenes que existan tras el `END SELECT`.

Por ejemplo

```
SELECT CASE (I)
     CASE (1)
       PRINT*, "I = 1"
     CASE (2:9)
       PRINT*, "I in [2,9]"
     CASE (10:)
       PRINT*, "I in [10,INF]"
     CASE DEFAULT
       PRINT*, "I is negative"
END SELECT CASE
```

La orden `SELECT CASE` es más eficiente que una combinación de `IF`'s porque una sola expresión controla las diferentes alternativas.

Los condicionales == y /= no deben usarse con expresiones del tipo real, ya que estas son aproximadas. Así, si `A` y `B` son variables reales, no debe usarse

```
...
IF (A==B) same = .TRUE.
...
```

Para evitar esto debe definirse una tolerancia y utilizar algo como

```
REAL :: TOL = 0.00001
...
IF (ABS(A-B) < TOL) same = .TRUE.
...
```

Estas estructuras de control condicionales pueden anidarse en varios niveles. En este caso es interesante a veces etiquetar los bucles para lograr una mayor claridad en el código, como en el siguiente caso

```
primerif: IF (a == 0) THEN
            PRINT*,  "a is zero"
            IF (c /= 0) THEN
              PRINT*,  "a is zero and c is not zero"
            ELSE
              PRINT*,  "a and c are zero"
            ENDIF
          ELSEIF (a > 0) THEN primerif
            PRINT*,  "a is positive"
          ELSE primerif
            PRINT*,  "a is negative"
          ENDIF primerif
```

La etiqueta `primerif` simplemente trata de hacer el código más claro, sin jugar ningún otro papel. Si la etiqueta se define en el `IF`, entonces ha de estar presente en el `ENDIF`, aunque en las instrucciones `ELSE` y `ELSEIF` sea opcional. Puede haber tantos condicionales anidados como se desee.

En el ejemplo 'Programa ejemplo_5_1.f90' en la página siguiente puede verse una aplicación de la estructura `IF THEN ELSE IF ELSE ENDIF` y en el ejemplo 'Programa ejemplo_5_2.f90' en la página 24 como se lleva a cabo, aparentemente, la misma tarea con una aplicación de la estructura `CASE`.

- Bucles y comandos de utilidad en bucles

  1 Primera forma de bucle (*loop*) con el comando `DO`

  Ya conocemos una forma de formar bucles con la orden `DO`.

  ```
  DO Var = vinicial, vfinal, incremento
        bloque de instrucciones
  END DO
  ```

  Con esta forma la variable `Var` toma el valor inicial y se va incrementando en `inc` hasta que alcanza el valor `vfinal`. Cada vez se ejecuta el bloque de instrucciones limitado por el final del bloque marcado por el `ENDDO`. Por ejemplo, en el ejemplo 'Programa ejemplo_5_3.f90' en la página 24 se imprime el seno y coseno para una circunferencia completa dividida en octantes.

  2 Segunda forma de bucle (*loop*) con el comando `DO`

  Esta segunda forma, con el esquema siguiente, se conoce como un bucle `WHILE`

  ```
  DO WHILE (expresion logica)
       bloque de instrucciones
  ENDDO
  ```

  En este caso el bloque de instrucciones se ejecuta mientras que la expresión lógica en la cabecera del bucle sea cierta. Por ejemplo, en el programa 'Programa ejemplo_5_4.f90' en la página 24 se calculan aproximaciones sucesivas al `SIN(X)` usando la fórmula del desarrollo de Taylor de esta función, avanzando un grado en el desarrollo hasta que el usuario introduzca un cierto valor (cero en este caso).

  3 Tercera forma de bucle (*loop*) con el comando `DO`

  Esta forma es la conocida como un bucle del tipo `REPEAT UNTIL`. El esquema es el siguiente.

```
DO
      bloque de instrucciones
       IF (expresion logica) EXIT
END DO
```

El bloque de instrucciones se repite hasta que se cumpla la expresión lógica en el `IF`. En este caso, a diferencia de los dos anteriores, siempre se ejecuta al menos una vez el bloque de instrucciones. En los otros dos casos puede no llegar a ejecutarse el bloque de instrucciones ni una sola vez.

En este caso se hace uso de la orden `EXIT`. Cuando se ejecuta esta orden el flujo del programa abandona el bucle y continúa en la orden siguiente al `ENDDO` que marca el final del bucle. Otra orden interesante cuando se trabaja con bucles es la orden `CYCLE`. Cuando se ejecuta esta orden se vuelve al principio del bucle, y no se ejecutan las órdenes que se hallen entre su posición y el final del bucle en esa iteración. Como en el caso de los condicionales, los diferentes bucles anidados pueden etiquetarse, lo que permite indicar a qué bucle en concreto se refiere la orden `EXIT` o `CYCLE`. Por defecto, sin usar etiquetas, se refieren al último bucle.

En el 'Programa ejemplo_5_5.f90' en la página 25 se combinan varias de las instrucciones de `FORTRAN` descritas en esta sección. Este programa lee un fichero de datos (que se encuentra comentado más abajo del programa, basta con cortar, pegar y eliminar los '!'). Al abrir el fichero con `OPEN` el programa se asegura de que sea un fichero existente y solo de lectura (`STATUS = 'OLD'` y `ACTION='READ'`). Comienza a leer el fichero y salta una serie de lineas, en un bucle del tipo `REPEAT UNTIL`, hasta llegar a una línea en la que se proporciona el número de puntos en el fichero[1].Tras ello lee los puntos y los almacena en los vectores `X` e `Y`, calculando el máximo valor de `X` y el mínimo valor de `Y`. Para llevar a cabo este cálculo usa las funciones intrínsecas `MAXVAL` y `MINVAL` (ver 'Objetivos' en la página 39).

Existe una última orden, cuyo uso no se fomenta ya que no está de acuerdo con los principios de programación estructurada, la orden `GOTO`.

5.3. Programas usados como ejemplo.

5.3.1. Programa ejemplo_5_1.f90

```
PROGRAM EJEMPLO_5_1
  !
  IMPLICIT NONE
  !
  REAL :: NOTA
  CHARACTER (3), DIMENSION(1:5) :: NT, LISTNT=(/'S','A','N','Sob','MH'/)
  INTEGER :: IN
  ! READ NOTE
  PRINT *, "Nota del estudiante?"
  READ 50, Nota
50 FORMAT(F4.1)
  !
  IF (Nota>=0.0.AND.Nota<5.0) THEN
     IN=1
  ELSE IF (Nota>=5.0.AND.Nota<7.0) THEN
     IN=2
  ELSE IF (Nota>=7.0.AND.Nota<9.0) THEN
     IN=3
  ELSE IF (Nota>=9.0.AND.Nota<10.0) THEN
     IN=4
  ELSE IF (Nota==10.0) THEN
     IN=5
  ELSE
     IN=0
  ENDIF
!
  IF (IN==0) THEN
     PRINT *, "LA NOTA : ", Nota," NO ES UNA NOTA ACEPTABLE."
  ELSE
     PRINT 100,  Nota, LISTNT(IN)
  ENDIF
!
  100 FORMAT(1X,'LA NOTA DEL ALUMNO ES ',F4.1,' (',A3,')')
!
END PROGRAM EJEMPLO_5_1
```

[1]Para conseguir esto se hace uso de la opción `IERR` = *label* en la orden `READ`. Esta opción indica que si se ha producido un error de lectura el programa debe saltar a la línea marcada por *label*.

5.3.2. Programa ejemplo_5_2.f90

```
PROGRAM EJEMPLO_5_2
  IMPLICIT NONE
  REAL :: Nota
  INTEGER :: IN, Inota
  CHARACTER (3), DIMENSION(1:5) :: LISTNT=(/'S','A','N','Sob','MH'/)
  ! READ NOTE
  PRINT *, "Nota del estudiante?"
  READ(*,*), Nota
  !
  Inota = NINT(Nota)
  !
  SELECT CASE (Inota)
  CASE (0:4)
     IN = 1
  CASE (5,6)
     IN = 2
  CASE (7,8)
     IN = 3
  CASE (9)
     IN = 4
  CASE (10)
     IN = 5
  CASE DEFAULT
     IN = 0
  END SELECT
  !
  IF (IN==0) THEN
     PRINT *, "LA NOTA : ", Nota," NO ES UNA NOTA ACEPTABLE."
  ELSE
     PRINT 100,  Nota, LISTNT(IN)
  ENDIF
  !
100 FORMAT(1X,'LA NOTA DEL ALUMNO ES ',F4.1,' (',A3,')')
  !
END PROGRAM EJEMPLO_5_2
```

5.3.3. Programa ejemplo_5_3.f90

```
PROGRAM EJEMPLO_5_3
!
  IMPLICIT NONE
!
  REAL :: Pio2 = ASIN(1.0)
  REAL :: Angle1 = 0.0, Angle2 = 0.0
  INTEGER :: index
!
  DO index = 0, 16, 2
     Angle1 = index*Pio2/4.0
!
     WRITE(*,*)
     WRITE(*,*) 'Cos(',index/2,'Pi/4) = ',COS(Angle1),'; Cos(',index/2,'Pi/4) = ',COS(Angle2)
     WRITE(*,*) 'Sin(',index/2,'Pi/4) = ',SIN(Angle1),'; Sin(',index/2,'Pi/4) = ',SIN(Angle2)
     WRITE(*,*)
!
     Angle2 = Angle2 + Pio2/2.0
!
  ENDDO
END PROGRAM ejemplo_5_3
```

5.3.4. Programa ejemplo_5_4.f90

```
PROGRAM ejemplo_5_4
!
  IMPLICIT NONE
!
  REAL :: X_val = 0.0
  REAL :: X_app = 0.0, X_sum = 0.0
  INTEGER :: I_flag = 1, I_count = 0
!
! APROXIMACIONES SUCESIVAS A SIN(X) = X - X^3/3! + X^5/5! - X^7/7! + ...
  WRITE(*,*) "INTRODUZCA EL VALOR DEL ANGULO X (RAD) :"
  READ(*,*) X_val
!
  I_count = 1
  X_app = X_val
  X_sum = X_val
!
  WRITE(*,*) '         ORDEN     APROX    SIN(X)       APPROX-SIN(X)'
```

```
  DO WHILE (I_flag == 1)
     WRITE(*,*) I_count, X_app, SIN(X_val), X_app - SIN(X_val)
!
     X_sum = X_sum*(-1)*X_val*X_val/((I_count*2+1)*(I_count*2))
     X_app = X_app + X_sum
!
     I_count = I_count + 1
!
     WRITE(*,*) "STOP? (0 SI, 1 NO)"
     READ(*,*) I_flag
     IF (I_flag /= 1 .AND. I_flag /= 0) I_flag = 1
!
  ENDDO
END PROGRAM ejemplo_5_4
```

5.3.5. Programa ejemplo_5_5.f90

```
PROGRAM ej_5_5
  !
  IMPLICIT NONE
  REAL , DIMENSION(:), ALLOCATABLE :: X_vec, Y_vec
  INTEGER :: Index, Ierr, Numpoints = 0
  REAL :: Max_x, Min_y
  CHARACTER(LEN=64) :: Filename
  !
  ! READ FILENAME
  READ(5,*) Filename
  ! OPEN FILE
  OPEN( UNIT=10, FILE=Filename, STATUS='OLD', ACTION='READ' )
  !
  DO
     READ(UNIT=10, FMT=100, ERR=10) Numpoints
     IF (Numpoints /= 0) EXIT
10   READ (UNIT=10, FMT=*) ! JUMP ONE LINE
     CYCLE
  ENDDO
  !
  PRINT*, 'NUMPOINTS = ', Numpoints
  !
  ! ALLOCATE X, Y VECTORS
  ALLOCATE(X_vec(1:NUMPOINTS), STAT = IERR)
  IF (Ierr /= 0) STOP 'X_vec MEM ALLOCATION FAILED'
  ALLOCATE(Y_vec(1:NUMPOINTS), STAT = IERR)
  IF (Ierr /= 0) STOP 'Y_vec MEM ALLOCATION FAILED'
  !
  DO I = 1, Numpoints
     !
     READ(UNIT=10, FMT=110) X_vec(I), Y_vec(I)
     !
  ENDDO
  !
  Max_x = MAXVAL(X_vec)
  Min_y = MINVAL(Y_vec)
  !
  PRINT*, "MAXIMUM X VALUE = ", Max_x
  PRINT*, "MINIMUM Y VALUE = ", Min_y
  ! DEALLOCATE AND CLOSE FILE
  DEALLOCATE(X_vec, STAT = IERR)
  IF (Ierr /= 0) STOP 'X_vec MEM DEALLOCATION FAILED'
  DEALLOCATE(Y_vec, STAT = IERR)
  IF (Ierr /= 0) STOP 'Y_vec MEM DEALLOCATION FAILED'
  !
  CLOSE(10)
  ! FORMAT STATEMENTS
100 FORMAT(19X,I3)
110 FORMAT(F6.3,1X,F6.3)
  !
END PROGRAM ej_5_5
!# Remark 1
!# Remark 2
!Useless line 1
!Useless line 2
!Number of points = 4
!+1.300;-2.443
!+1.265;-1.453
!+1.345;-8.437
!+1.566;+4.455
```

Capítulo 6

Operaciones de Entrada/Salida (I/O) (I)

6.1. Objetivos

Los objetivos de esta clase son los siguientes:

1 presentar como es posible aprovechar el redireccionamiento de la entrada y salida estándar en UNIX para la lectura y escritura de datos en Fortran.

2 presentar la orden FORMAT, sus diferentes descriptores y cómo se define un formato que combina con las órdenes PRINT y WRITE.

3 adquirir conocimientos básicos acerca del uso de ficheros en Fortran con las órdenes OPEN, CLOSE, WRITE.

En esta clase, salvo en el primer punto mencionado, que es general, nos centramos en las operaciones de salida desde el programa Fortran, mientras que la siguiente clase se dedica a aplicar lo aprendido a la lectura de datos desde Fortran.

6.2. Puntos destacables.

- Redireccionamiento en UNIX

  El redireccionado de la entrada y salida estándar (STDIN/STDOUT) mediante los símbolos < y > permite, sin controlar en principio los formatos, desviar del teclado y de la pantalla la entrada y la salida estándar de un programa Fortran.

  Por ejemplo, los siguientes comandos hacen que al correr un programa llamado a.out su salida se dirija al fichero output.dat en el primer caso. En el segundo caso el programa lee su entrada de un fichero llamado output.dat mientras que en el tercer caso combinamos ambos procedimientos.

  ```
  a.out > output.dat
  a.out < input.dat
  a.out <input.dat > output.dat
  ```

  El ejercicio propuesto número 4 se facilita mucho si se tiene esto en cuenta.

  Puede redireccionarse también la salida de error (STDERR) como

  ```
  a.out 2> output.dat
  a.out 2>&1 input.dat
  ```

  En el segundo case se unen ambas salidas, STDERR y STDOUT

- Para que se tenga un mayor control del formato de las entradas y salidas se combinan las órdenes de entrada (READ) y salida (PRINT) con los llamados descriptores de formato. Hasta ahora hemos usado estas órdenes con los formatos por defecto, usando el llamado *formato libre*: READ(*,*) o READ* y PRINT*. Para especificar un formato dado a estos comandos se suelen usar en la forma PRINT *nlin*, *output_list*, donde *nlin* es una etiqueta que indica un comando FORMAT

que proporciona los necesarios descriptores de formato y *output_list* son las variables y constantes que deseamos imprimir. Ocurre de manera similar con la orden READ. Veremos diferentes ejemplos al explicar los diferentes descriptores de formato posibles. Veremos también que es posible incluir el formato directamente en la orden PRINT.

Los descriptores de formato en FORTRAN están influenciados por el hecho de que el dispositivo de salida habitual cuando se desarrolló este lenguaje era la impresora de línea. Por ello el primer carácter es un carácter de control de modo que si el primer carácter es

1. 0 : espaciado doble.
2. 1 : salta a nueva página.
3. + : sin espaciado, imprime la línea sobre la línea anterior.
4. blank : espaciado simple

Veremos los diferentes descriptores de formato: descriptores para fijar la posición vertical de una línea de texto, para alterar la posición horizontal de los caracteres en una línea, descriptores para mostrar adecuadamente datos enteros (I), reales (F y E), caracteres A y variables lógicas (L).

Usamos los siguientes símbolos

1. c : número de columna
2. d : número de dígitos tras el punto decimal (valores reales)
3. m : número mínimo de dígitos mostrados
4. n : número de espacios
5. r : número de veces que se repite un determinado descriptor
6. w : número de caracteres a los que afecta un determinado descriptor

Se explica en primer lugar el uso de los descriptores con operaciones de salida.

1. Descriptor de enteros I:
   Su forma general es rIw o rI$w.m$. Este descriptor indica que se van a leer o escribir r enteros que ocupa w caracteres o columnas. El número se justifica a la derecha, por lo que si es menor el número de dígitos que el previsto se rellena de espacios a la izquierda del número. Por ejemplo

   ```
   PRINT 100, I, I*I
   100 FORMAT(' ',I3, ' AL CUADRADO ES ', I6)
   ```

   imprime un espacio, tras él un número entero de como máximo tres dígitos. Luego imprime la cadena 'AL CUADRADO ES' y termina con el número al cuadrado, que se prevé que ocupará como máximo seis dígitos. Para ver un ejemplo de uso de este descriptor así como para ver qué ocurre si se supera el número máximo de dígitos permitidos puede usarse el ejemplo 'Programa ejemplo_6_1.f90' en la página 31. En este ejemplo incluimos el descriptor X, donde nX implica que se incluyan n espacios en la salida.
   Si en el ejemplo anterior se desea introducir el formato en la orden PRINT tendríamos

   ```
   PRINT "(' ',I3, ' AL CUADRADO ES ', I6)", I, I*I
   ```

   Como puede verse en la salida del ejemplo 'Programa ejemplo_6_1.f90' en la página 31 tenemos un desbordamiento aritmético de la variable big, ya que es una variable de 32 bits, tras lo cual la salida obtenida no tiene sentido alguno. Para resolver este problema podemos definir la variable como variable de 64 bits, como en el ejemplo 'Programa ejemplo_6_2.f90' en la página 31. En este ejemplo podemos ver lo que ocurre cuando el formato no cuenta con tamaño suficiente para acomodar el resultado del cálculo.
2. Descriptor de reales F:
   Su forma más general es rF$w.d$, siendo w el número total de columnas empleadas en la representación del número y d el número de dígitos tras el punto decimal. Por ejemplo F7.3 implica que el número tiene tres decimales, más el punto decimal, por lo que quedan tres dígitos no decimales, esto es, sería valido para números entre −99.999 y 999.999. Si se trunca la parte decimal se redondea adecuadamente. Es importante tener en cuenta que a veces el redondeo hará que el número tenga más cifras de las previstas y la salida se verá sustituida por una serie de asteriscos (*). En el ejemplo 'Programa ejemplo_6_3.f90' en la página 31 vemos el uso de este formato así como los posibles problemas que puede conllevar cuando las cifras implicadas no se adaptan adecuadamente al formato previsto.

3 Descriptor de reales E:

Permite el uso de la notación científica. Su forma general es *r*E*w.d*. El número que multiplica a la potencia de 10 toma valores entre 0.1 y 1.0. Este caso se distingue del anterior porque hay que reservar sitio para el exponente. De hecho es necesario un carácter para el punto decimal, otro para E (el indicador de exponente, que implica la potencia de diez por la que se multiplica el número), el signo del número completo, la magnitud del exponente y el signo del exponente. Por tanto, el tamaño mínimo en este caso es $w = d+7$. El ejemplo 'Programa ejemplo_6_4.f90' en la página 32 es idéntico al ejemplo 'Programa ejemplo_6_3.f90' en la página 31 sustituyendo los descriptores F por E. Como puede verse en este ejemplo, este formato es muy adecuado cuando se debe trabajar con valores que abarquen un gran rango de variación.

4 Descriptor de reales ES:

Permite el uso de la notación científica estándar. Su forma general es *r*ES*w.d*. El número que multiplica a la potencia de 10 toma valores entre 1.0 y 10.0. Aparte de esto, este caso es muy similar al anterior

5 Descriptor de datos lógicos L:

Este descriptor aparece en la forma *r*L*w*. Estos datos solo pueden tomar los valores TRUE y FALSE y la salida de este descriptor será T o F justificado a la derecha.

6 Descriptor de caracteres A:

Este descriptor aparece en la forma *r*A o *r*A*w*. En el primer lugar se indica que se repiten *r* veces formatos de caracteres con una anchura igual al número de caracteres de los que consta la variable. En el segundo caso la salida tiene una anchura fija dada por *w*. El ejemplo 'Programa ejemplo_6_5.f90' en la página 32 muestra el uso de este descriptor.

7 Descriptor X:

El descriptor *n*X controla el desplazamiento horizontal e indica que se escriban *n* espacios en la salida. El ejemplo 'Programa ejemplo_6_5.f90' en la página 32 muestra el uso de este descriptor combinado con los anteriores.

8 Descriptor T:

El descriptor T*c* controla el desplazamiento horizontal e indica que se salte directamente a la columna *c*.

9 Descriptor /:

El descriptor */c* envía la salida acumulada hasta el momento a STDOUT y comienza un nuevo registro con un retorno de carro. Este descriptor no necesita ser separado del resto mediante comas. Tras cada retorno de carro puede ubicarse un carácter de control.

10 Repetición:

En todos los casos descritos se puede indicar la repetición de formato usando un número a la derecha del símbolo que indica el formato. Por ejemplo,

```
100 FORMAT(1X, 2I6, 3F9.3)
```

es equivalente a

```
100 FORMAT(1X, I6, I6, F9.3, F9.3, F9.3)
```

y se puede simplificar a

```
100 FORMAT(1X, 3(I6, F9.3))
```

Es también necesario citar el significado especial que tiene un formato entre paréntesis en una orden FORMAT y lo que sucede si el número de datos proporcionado en la lista de variables de la orden WRITE es mayor o menor que el número de descriptores asociados en la correspondiente orden FORMAT.

- Fortran permite la manipulación de ficheros. En este caso vamos a concentrarnos en como se crean y se escriben ficheros. Para ello se utilizan las órdenes básicas OPEN, WRITE y CLOSE. Otras órdenes de interés son REWIND y BACKSPACE.

  La orden OPEN permite que se abra un fichero, en su forma más simple sería

```
OPEN(UNIT=int_unit,FILE='nombre_fichero')
```

  donde se indica el nombre del fichero creado y el número de unidad asociada a dicho fichero (número entero). Para operaciones de lectura y escritura el fichero queda asociado a este número. Un ejemplo de esto sería escribir algo en este fichero usaremos una orden como

```
OPEN(UNIT=33, FILE='output_program.dat')
WRITE(UNIT=33,FMT=100) lista_de_variables
```

lo que indica que envíe los datos en *lista_de_variables* al fichero asociado a la unidad 33, llamado `output_program.dat`, según el formato especificado en la línea etiquetada como `100`. Es posible usar la forma abreviada `WRITE(33,100)` e incluso `WRITE(33,*)` si queremos escribir en el fichero con el formato estándar dependiente de la lista de variables. Para escribir en la salida estándar `STDOUT`, en principio la pantalla, se puede usar `WRITE(6,formato)`, ya que esta salida está asociada a la unidad `6`. Por tanto `WRITE(6,*)` es equivalente a `PRINT*`, mientras que `STDIN` se asocia a la unidad `5`[1]. Para referirnos a `STDOUT` y `STDIN` es más conveniente usar el carácter `*` ya que los números anteriores pueden depender del procesador.

Una vez que el proceso de escritura ha terminado el fichero se cierra con la orden `CLOSE(UNIT=numero_unidad)`. En nuestro caso

```
CLOSE(UNIT=33)
```

El ejemplo 'Programa ejemplo_6_6.f90' en la página 32 muestra como se escribe a un fichero e introduce la función intrínseca de `Fortran` `CPU_TIME` que nos permite estimar el tiempo de cpu dedicado a cada parte del programa.

La orden `OPEN` puede especificarse de forma más detallada con los argumentos siguientes:

```
OPEN(UNIT=int_unit,FILE=char_nombre_fichero,STATUS=char_sta, ACTION=char_act, IOSTAT=int_stat)
```

Las nuevas opciones fijan los siguientes aspectos al abrir el fichero:

1 `STATUS=char_sta`
La constante o variable *char_sta* es de tipo carácter y puede tomar los siguientes valores estándar:
- 'OLD'
- 'NEW'
- 'REPLACE'
- 'SCRATCH'
- 'UNKNOWN'

2 `ACTION=char_act`
La constante o variable *char_act* es de tipo carácter y puede tomar los siguientes valores estándar:
- 'READ'
- 'WRITE'
- 'READWRITE'

Por defecto, si no se da otra opción, desde `Fortran`, los archivos se abren con los permisos activados tanto de lectura como de escritura.

3 `IOSTAT=int_var`
La constante o variable *int_stat* es de tipo entero y permite estimar si la apertura y el acceso al fichero ha sido exitosa. Si el valor que devuelve es `0` quiere decir que la apertura se ha resuelto sin problemas. Si ha habido problemas al abrir el fichero devuelve un valor diferente de cero.

Un ejemplo más completo que el anterior, creando un fichero de entrada (INPUT) sería

```
INTEGER ierror
OPEN(UNIT=33, FILE='input_program.dat', STATUS='OLD', ACTION='READ', IOSTAT=ierror)
```

Para crear un fichero de salida se hace

```
INTEGER ierror
OPEN(UNIT=33, FILE='output_program.dat', STATUS='NEW', ACTION='WRITE', IOSTAT=ierror)
```

- Es posible controlar de forma rudimentaria el acceso a los elementos almacenados en el fichero secuencial creado con la orden `OPEN`. Para ello existen dos comandos de interés.

```
BACKSPACE(UNIT = int_unit)
REWIND(UNIT = int_unit)
```

[1] `STDERR` se asocia a la unidad `0`.

Con BACKSPACE se retrocede un registro (una línea) en el fichero asociado a la unidad *int_unit*. Con REWIND se vuelve al primer registro del fichero.

- Hasta ahora hemos empleado la orden OPEN con ficheros con formato, siendo este el comportamiento por defecto al abrir un fichero. Por tanto, las órdenes

```
OPEN(UNIT=33,FILE='nombre_fichero')
OPEN(UNIT=33,FILE='nombre_fichero',FORM='FORMATTED')
```

son equivalentes. Los ficheros con formato tienen la ventaja de ser editables por el usuario, pero precisan de un mayor tiempo para su lectura y escritura que cuando se tratan los datos sin formato. Para ello se abren los ficheros con la opción FORM='UNFORMATTED' como en el ejemplo siguiente

```
OPEN(UNIT=33,FILE='nombre_fichero',FORM='UNFORMATTED')
```

Además, que los fichero no tengan formato permite almacenar números reales sin la pérdida de información que implica el cambio a un formato dado. En el Ejercicio 5 se pide que transforméis el ejemplo 'Programa ejemplo_6_6.f90' en la página siguiente para comprobar el tiempo que se tarda en la escritura y lectura sin formato. Al escribir en un fichero sin formato no se indica ningún formato en la orden, de manera que para escribir en el fichero sin formato que hemos abierto asociado con UNIT=33 haremos

```
WRITE(UNIT=33) lista-de-variables
```

6.3. Programas usados como ejemplo.

6.3.1. Programa ejemplo_6_1.f90

```
PROGRAM ej_6_1
!
  IMPLICIT NONE
  INTEGER :: i, big=10
!
  DO i=1,18
     PRINT 100, i, big
     100 FORMAT(1X, '10 elevado a ',I3,2X,'=',2X,I12)
     big=big*10
  END DO
END PROGRAM ej_6_1
```

6.3.2. Programa ejemplo_6_2.f90

```
PROGRAM ejemplo_6_2
!
  IMPLICIT NONE
!
  INTEGER, PARAMETER :: Long=SELECTED_INT_KIND(16) ! Tipo de entero de 64 bits
  INTEGER :: i
  INTEGER (KIND = Long) :: big=10
!
  DO i=1,18
!
     PRINT 100, i, big
     100 FORMAT(1X, '10 elevado a ',I3,2X,'=',2X,I16)
!
     big=big*10
!
  END DO
!
END PROGRAM ejemplo_6_2
```

6.3.3. Programa ejemplo_6_3.f90

```
PROGRAM ej_6_3
! Programa que hace patente problemas de desbordamiento
! aritmetico por exceso y por defecto.
  IMPLICIT NONE
  INTEGER :: I
```

```
  REAL     :: peq = 1.0
  REAL     :: gran    = 1.0
!
  DO i=1,45
     PRINT 100, I, peq, gran
100  FORMAT(' ',I3,' ',F9.4,' ',F9.4)
!
     peq=peq/10.0
     gran=gran*10.0
!
  END DO
END PROGRAM ej_6_3
```

6.3.4. Programa ejemplo_6_4.f90

```
PROGRAM ej_6_4
! Programa que hace patente problemas de desbordamiento
! aritmético por exceso y por defecto.
  IMPLICIT NONE
  INTEGER :: I
  REAL     :: peq = 1.0
  REAL     :: gran    = 1.0
!
  DO i=1,45
     PRINT 100, I, peq, gran
100  FORMAT(' ',I3,' ',E10.4,' ',E10.4)
!
     peq=peq/10.0
     gran=gran*10.0
!
  END DO
END PROGRAM ej_6_4
```

6.3.5. Programa ejemplo_6_5.f90

```
PROGRAM ej_6_5
  ! Programa que calcula el IMC o indice de Quetelet.
  ! Se hace según la expresión matemática:
  !    IMC=(peso (kg))/(talla^2 (m^2))
  !
  IMPLICIT NONE
  CHARACTER (LEN=25) :: NOMBRE
  INTEGER :: hcm = 0, peso = 0 ! altura en cm y peso en kg
  REAL     :: talla = 0.0 ! altura en metros
  REAL     :: IMC ! Indice de masa corporal
  !
  PRINT*, 'Introduzca su nombre de pila:'; READ*, NOMBRE
  !
  PRINT*, 'Introduzca su peso en kilogramos:'; READ*, peso
  !
  PRINT*, 'Introduzca su altura en centimetros:'; READ*, hcm
  !
  talla = hcm/100.0
  IMC = peso/(talla**2)
  !
  PRINT 100, NOMBRE, IMC, IMC
100  FORMAT(1X,'El IMC de ',A,' es ',F10.4,' o ',E10.4)
!
END PROGRAM ej_6_5
```

6.3.6. Programa ejemplo_6_6.f90

```
PROGRAM ej_6_6
!
  IMPLICIT NONE
  INTEGER , PARAMETER :: N=1000000
  INTEGER , DIMENSION(1:N) :: X
  REAL    , DIMENSION(1:N) :: Y
  INTEGER :: I
  REAL :: T
  REAL    , DIMENSION(1:5) :: TP
  CHARACTER(LEN = 10) :: COMMENT
!
  OPEN(UNIT=10,FILE='ej_6_6.txt')
!
  CALL CPU_TIME(T)
!
  TP(1)=T
```

```
  COMMENT=' T INICIAL '
  PRINT 100,COMMENT,TP(1)
!
  DO I=1,N
     X(I)=I
  END DO
!
  CALL CPU_TIME(T)
!
  TP(2)=T-TP(1)
  COMMENT = ' VAR ENTERA '
  PRINT 100,COMMENT,TP(2)
!
  Y=REAL(X)
!
  CALL CPU_TIME(T)
!
  TP(3)=T-TP(1)-TP(2)
  COMMENT = ' VAR REAL '
!
  PRINT 100,COMMENT,TP(3)
!
  DO I=1,N
     WRITE(10,200) X(I)
     200 FORMAT(1X,I10)
  END DO
!
  CALL CPU_TIME(T)
  TP(4)=T-TP(1)-TP(2)-TP(3)
!
  COMMENT = ' WRITE INTEG '
  PRINT 100,COMMENT,TP(4)
!
  DO I=1,N
     WRITE(10,300) Y(I)
300  FORMAT(1X,f10.0)
  END DO
!
  CALL CPU_TIME(T)
  TP(5)=T-TP(1)-TP(2)-TP(3)-TP(4)
!
  COMMENT = ' WRITE REAL '
  PRINT 100,COMMENT,TP(5)
  100 FORMAT(1X,A,2X,F7.3)
END PROGRAM ej_6_6
```

Capítulo 7

Operaciones I/O (II)

7.1. Objetivos

Los objetivos de esta clase son los siguientes:

1 presentar como se emplea la orden FORMAT, y sus diferentes descriptores con la orden READ.

2 adquirir conocimientos básicos acerca de la lectura de ficheros en Fortran.

3 dar como alternativas útiles para las operaciones de I/O los *here documents* y el formato de NAMELIST.

4 dar un ejemplo de como se puede trabajar con ficheros internos (*internal files*).

Son similares a los de la clase anterior, aunque en este caso nos centramos en la lectura de datos en Fortran. En general la lectura con formato se emplea al leer datos en ficheros y no al introducir datos con el teclado como entrada estándar.

7.2. Puntos destacables.

- La orden FORMAT se emplea de igual modo que en las operaciones de escritura tratadas en 'Operaciones de Entrada/Salida (I/O) (I)' en la página 27, teniendo los descriptores idéntico significado.
- Una opción útil cuando se emplea el comando READ es IOSTAT, que permite detectar si se ha llegado al final del archivo que se esté leyendo o hay un error en la lectura del dato. De este modo el comando quedaría como

```
READ(UNIT=int_unit, FMT=int_fmt, IOSTAT=int_var) input_list
```

  De este modo, si leemos un conjunto de datos (por ejemplo coordenadas de puntos (var_X,var_Y,var_Z) de un archivo y no sabemos el número total de los mismos podemos hacer algo como

```
num_data = 0
readloop: DO
   READ(UNIT=33, FMT=100, IOSTAT=io_status) var_X, var_Y, var_Z
   !
   ! Check reading
   IF (io_status /= 0) THEN
     ! Error in the input or EOF
     EXIT
   ENDIF
   num_data = num_dat + 1
   !   work with the coordinates
   !
      ......
   !
   ! Format statement
100 FORMAT(1X, 3F25.10)
   !
ENDDO readloop
```

En este fragmento de código la variable entera num_data es un contador que almacena el número de datos (puntos) leídos y la variable entera io_status controla que los datos se hayan leído de forma correcta.

- En el ejemplo 'Programa ejemplo_7_1.f90' en esta página se presenta como se lee un fichero de datos adjunto (notas.dat) usando secciones de matrices en el proceso de lectura. En dicho fichero se ordenan las notas obtenidas por una serie de alumnos en filas y columnas. Cada fila está asociada a un alumno y cada columna a una asignatura. Es conveniente que comprendáis cómo se lleva a cabo la lectura de los datos en este caso.

  Se plantea como ejercicio modificar 'Programa ejemplo_7_1.f90' en esta página para que lleve a cabo la misma tarea con el fichero notas_ej_6.dat, donde no se han llevado a cabo los promedios por alumno ni por asignatura. Partiendo del programa ejemplo debéis realizar los cambios necesarios para leer los datos, comprobar si hay errores de lectura, calcular los promedios necesarios, obtener la salida correcta y grabar un fichero donde aparezcan, como en notas.dat, los promedios calculados.

- Una forma bastante cómoda de leer datos en un programa Fortran es usando los llamados *here documents* de la shell. Esto consiste en hacer un breve *script*[1], en el que además de ejecutar el programa -compilándolo previamente si el programa no es demasiado extenso- se proporciona la entrada del programa, pudiendo comentarse dichas entradas. El ejemplo 'Programa ejemplo_7_2.f90' en la página siguiente es un programa que calcula las raíces de una ecuación de segundo grado y = A*x**2 + B*x + C y en el ejemplo 'Script ej_here_file' en la página siguiente, fichero ej_here_file encontrais una forma de aplicar los *here documents*. Para ejecutarlo tendréis que salvar el fichero y escribir

```
. ej_here_file
```

- Una tercera posibilidad es hacer uso del llamado namelist que consiste en una lista de valores asignados a variables etiquetadas con sus nombres. La forma de un comando NAMELIST es la siguiente

```
NAMELIST/var_group_name/ var1 [var2 var3 ... ]
```

  Este comando define las variables asociadas al grupo llamado *var_group_name* y debe aparecer en el programa antes de cualquier comando ejecutable. Para leer las variables en un NAMELIST se utiliza un comando READ donde en vez de especificar el formato de entrada con la opción FMT se utiliza la opción NML como sigue[2]

```
READ(UNIT=unit_number, NML=var_group_name, [...])
```

  El fichero NAMELIST con la información de entrada comienza con un carácter "&" seguido del nombre del grupo, *var_group_name*, y termina con el carácter "/". Los valores en el fichero pueden estar en líneas diferentes siempre que se hallen entre estos dos caracteres.

  El programa 'Programa ejemplo_7_3.f90' en la página siguiente es casi idéntico al programa 'Programa ejemplo_7_2.f90' en la página siguiente. Se ha modificado para hacer uso de un fichero de entrada en formato namelist, que llamamos sec_order.inp e incluimos como 'namelist input file' en la página 38.

- En 'Programa ejemplo_7_4.f90' en la página 38 se puede ver como se trabaja con un fichero interno (*internal file*) donde la operación de I/O tiene lugar en un buffer interno en lugar de en un archivo. Esto tiene utilidad en dos casos. En primer lugar para tratar con datos cuya estructura o formato no es bien conocida, por lo que se pueden leer en una variable de tipo CHARACTER y posteriormente ser tratados como un fichero interno. En segundo lugar, también permiten preparar datos que sean mezcla de datos de tipo carácter y de tipo numérico. Este segundo caso es el que se trata en 'Programa ejemplo_7_4.f90' en la página 38 donde se muestra como definir una serie de unidades de escritura asociadas a archivos que se van numerando sucesivamente.

  En este ejemplo se puede ver como se usa la función intrínseca TRIM para eliminar los espacios sobrantes de la variable pref al preparar el nombre del fichero.

7.3. Programas usados como ejemplo.

7.3.1. Programa ejemplo_7_1.f90

```
PROGRAM EJEMPLO_7_1
```

[1] From The Free On-line Dictionary of Computing (8 July 2008) [foldoc]: *script*: A program written in a scripting language.
[2] El formato NAMELIST también se puede utilizar con el comando WRITE para guardar variables etiquetadas en operaciones de salida.

```
  IMPLICIT NONE
  !Definicion de variables
  INTEGER , PARAMETER :: NROW=5
  INTEGER , PARAMETER :: NCOL=6
  REAL , DIMENSION(1:NROW,1:NCOL)   :: RESULT_EXAMS = 0.0
  REAL , DIMENSION(1:NROW)          :: MEDIA_ESTUD = 0.0
  REAL , DIMENSION(1:NCOL)          :: MEDIA_ASIGN = 0.0
  INTEGER :: R,C
  !
  ! Abrir fichero para lectura
  OPEN(UNIT=20,FILE='notas.dat',STATUS='OLD')
  !
  DO R=1,NROW
     READ(UNIT=20,FMT=100) RESULT_EXAMS(R,1:NCOL),MEDIA_ESTUD(R) ! Lectura de notas y luego de promedio
     100 FORMAT(6(2X,F4.1),2X,F5.2) ! Se leen 6 numeros seguidos y luego un septimo
  ENDDO
  READ (20,*) ! Saltamos una linea con esta orden
  READ (20,110) MEDIA_ASIGN(1:NCOL) !
110 FORMAT(6(2X,F4.1))
!
! IMPRESION DE LAS NOTAS EN LA SALIDA ESTANDAR
  DO R=1,NROW
     PRINT 200, RESULT_EXAMS(R,1:NCOL), MEDIA_ESTUD(R)
200  FORMAT(1X,6(1X,F5.1),' = ',F6.2)
  END DO
  PRINT *,'  ====  ====  ====  ====  ====  ==== '
  PRINT 210, MEDIA_ASIGN(1:NCOL)
210 FORMAT(1X,6(1X,F5.1))
END PROGRAM EJEMPLO_7_1
```

7.3.2. Programa ejemplo_7_2.f90

```
PROGRAM EJEMPLO_7_2
! PROGRAMA QUE CALCULA LAS SOLUCIONES DE UNA EC DE SEGUNDO GRADO
! y = A*x**2 + B*x + C
  IMPLICIT NONE
  !Definicion de variables
  REAL :: A = 0.0
  REAL :: B = 0.0
  REAL :: C = 0.0
  REAL, DIMENSION(2) :: SOL
  REAL :: TEMP
  INTEGER :: I
  !
  ! Lectura de coeficientes
  READ*, A
  READ*, B
  READ*, C
  ! print*, a,b,c
  !
  ! CALCULOS
  TEMP = SQRT(B*B-4.0*A*C)
  !
  SOL(1) = (-B+TEMP)/(2.0*A)
  SOL(2) = (-B-TEMP)/(2.0*A)
  !
  !
  ! IMPRESION DE LAS SOLUCIONES
  DO I=1, 2
     PRINT 200, I, SOL(I)
200  FORMAT(1X,'SOLUCION ', I2,' = ',F18.6)
  END DO
!
END PROGRAM EJEMPLO_7_2
```

7.3.3. Script `ej_here_file`

```
# Compilar
gfortran -o second_order  ejemplo_7_2.f90
# Ejecutar
./second_order <<eof
2.0     # A
1.0     # B
-4.0    # C
eof
```

7.3.4. Programa ejemplo_7_3.f90

```
PROGRAM EJEMPLO_7_3
```

```
! PROGRAMA QUE CALCULA LAS SOLUCIONES DE UNA EC DE SEGUNDO GRADO
! y = A*x**2 + B*x + C
  IMPLICIT NONE
  !Definicion de variables
  REAL :: A = 0.0
  REAL :: B = 0.0
  REAL :: C = 0.0
  REAL, DIMENSION(2) :: SOL
  REAL :: TEMP
  INTEGER :: I
  !
  !     NAMELIST DEFINITIONS
  NAMELIST/INP0/ A, B, C
  !     NAMELIST FILE
  OPEN(UNIT=10,FILE='sec_order.inp',STATUS='OLD')
  ! Lectura de coeficientes
  READ(10,INP0)
  !
  ! CALCULOS
  TEMP = SQRT(B*B-4.0*A*C)
  !
  SOL(1) = (-B+TEMP)/(2.0*A)
  SOL(2) = (-B-TEMP)/(2.0*A)
  !
  !
  ! IMPRESION DE LAS SOLUCIONES
  DO I=1, 2
     PRINT 200, I, SOL(I)
200  FORMAT(1X,'SOLUCION ', I2,' = ',F18.6)
  END DO
!
END PROGRAM EJEMPLO_7_3
```

7.3.5. `namelist` input file

```
#
#       INPUT FILE FOR EJEMPLO_7_3
#
 &INP0 A=2.0, B=1.0, C=-4.0 /
```

7.3.6. Programa ejemplo_7_4.f90

```
PROGRAM EJEMPLO_7_4
  !
  ! Internal file example
  !
  IMPLICIT NONE
  !Definicion de variables
  REAL :: x_var
  INTEGER :: unit_n, index_X
  CHARACTER(LEN=65) :: filename
  CHARACTER(LEN=56) :: pref
  !
  PRINT*, "Introduce file name preffix: "
  READ(*,*) pref
  !
  DO unit_n = 10, 20
     !
     WRITE(filename, '(A, "_", i2,".dat")') TRIM(pref), unit_n
     OPEN(UNIT = unit_n, FILE = filename, STATUS = "UNKNOWN", ACTION = "WRITE")
     !
     DO index_X = 0, 100
        x_var = REAL(index_X)*0.01
        WRITE(unit_n, '(1X,2ES14.6)') x_var, SIN(REAL(unit_n)*x_var)
     ENDDO
     !
     CLOSE(UNIT = unit_n)
     !
  ENDDO
!
END PROGRAM EJEMPLO_7_4
```

Capítulo 8

Subprogramas (I): funciones

8.1. Objetivos

Los objetivos de esta clase son los siguientes:

1 presentar las ventajas que ofrece el uso de funciones, subrutinas y módulos.

2 presentar el concepto de función en `Fortran`.

3 mostrar los diferentes tipos de funciones que pueden usarse: intrínsecas, genéricas, elementales, transformacionales e internas.

4 hacer posible la definición de funciones por el usuario.

5 mostrar la diferencia entre funciones externas y funciones contenidas en el programa principal.

El uso de estas estructuras permite desarrollar una programación más estructurada y eficaz ya que permiten

- el desarrollo y comprobación de diferentes subtareas de forma independiente.
- reciclar unidades en diferentes programas, disminuyendo el tiempo necesario para generar el código.
- aislar el código de efectos inesperados ya que están perfectamente diferenciadas las variables generales y las variables a las que tiene acceso cada subrutina o función.

8.2. Puntos destacables.

En primer lugar nos centramos en las funciones y, en la siguiente unidad, tratamos las subrutinas.

- Características generales de las funciones.

  Las principales características de una función son

  - Requieren el uso de uno o varios parámetros o argumentos al invocar la función.
  - Dichos argumentos pueden ser una expresión.
  - Generalmente una función da como resultado un valor, y dicho valor es función de los parámetros o argumentos con los que se ha invocado la función. El resultado de la evaluación de la función puede ser un escalar o una matriz de cualquier tipo.
  - Los argumentos pueden ser de diferente tipo.

En Fortran existen aproximadamente más de cien funciones predefinidas que pueden usarse de modo muy simple y directo. Por ejemplo, si necesitamos funciones trigonométricas podemos usar, siendo X, Y variables reales[1]:

- Y = SIN(X)
- Y = COS(X)
- Y = TAN(X)

Este tipo de funciones son llamadas *funciones intrínsecas*. En esta URL (http://gcc.gnu.org/onlinedocs/gfortran/Intrinsic-Procedures.html#Intrinsic-Procedures) puede encontrarse una lista completa de las funciones intrínsecas de que se dispone al usar el compilador gfortran. En general las funciones intrínsecas son *genéricas*, lo que quiere decir que admiten distintos tipos de argumentos (mejor dicho argumentos con diferente precisión). La excepción son las cuatro funciones no genéricas, LGE, LGT, LLE y LLT.

- Las funciones llamadas *elementales* admiten como argumento tanto escalares como matrices. Un ejemplo de esto se puede ver en los ejemplos 'Programa ejemplo_8_1.f90' en la página siguiente y 'Programa ejemplo_8_2.f90' en la página siguiente, donde se muestra el carácter elemental y genérico de algunas funciones. En el ejemplo 'Programa ejemplo_8_1.f90' en la página siguiente se muestra también la diferencia entre variables reales con diferente precisión. Estas funciones elementales pueden aplicarse tanto a variables escalares como a arreglos. En el último caso la función se aplica a cada elemento del arreglo.
- Existen también otro tipo de funciones. Entre ellas destacan las del tipo *inquiry* que dan información acerca de las características de un arreglo. Por ejemplo las funciones SIZE y ALLOCATED. Un ejemplo del uso de esta última función puede verse en el 'Programa ejemplo_5_5.f90' en la página 25 y en el 'Programa ejemplo_9_3.f90' en la página 51.

  Otro tipo de funciones son las llamadas "transformacionales", p.e. REAL y TRANSPOSE, que transforman entre diferentes tipos de datos, y funciones que implican medidas de tiempo como SYSTEM_CLOCK y DATE_AND_TIME.
- Conversión entre diferentes tipos de datos.
  - REAL(i): convierte un entero i en un real. El argumento i puede ser un entero, un real de doble precisión o un número complejo.
  - INT(x): convierte el real x en el entero equivalente, truncando la parte decimal, pudiendo ser x una variable real, real de doble precisión o compleja.
  - Las siguientes funciones son de interés para definir enteros a partir de reales.
    - CEILING(x): convierte un real x en el entero más pequeño mayor o igual a x.
    - FLOOR(x): convierte un real x en el mayor entero que sea menor o igual a x.
    - NINT(x): convierte un real x en el entero más próximo a x.
  - DBLE(a): convierte a a doble precisión. El argumento puede ser entero, real o complejo.
  - CMPLX(x) ó CMPLX(x,y): convierte en valores complejos (el segundo argumento es la parte imaginaria).
- Además de las funciones intrínsecas, pueden definirse funciones. La definición de una función implica por una parte la propia definición y la posterior llamada a la función desde un programa. La definición de una función sigue el siguiente esquema:

```
FUNCTION fun_name(argument_list)
  IMPLICIT NONE
  Declaration section (including arguments and fun_name)
  ....
  Local variables declaration
  ....
  fun_name = expr
  RETURN ! Optional
END FUNCTION fun_name
```

En el ejemplo 'Programa ejemplo_8_3.f90' en la página 42 se muestra como se define e invoca una función que calcula el máximo común divisor de dos números enteros. Es importante tener en cuenta lo siguiente:

- En este ejemplo podemos distinguir dos bloques. Un primer bloque con el programa principal y un segundo bloque donde se define la función. De hecho la función puede definirse en un fichero diferente al programa principal, y dar ambos ficheros al compilador para que prepare el programa ejecutable.

[1] Hay que tener en cuenta que en Fortran se supone que los ángulos en las funciones trigonométricas vienen expresados en radianes y no en grados.

- Es importante tener en cuenta que las variables definidas en la función tienen carácter local respecto a las variables que se definen en el programa.
- La función en este caso tiene como nombre `MCD` y su tipo es `INTEGER`. Por tanto, el programa espera que el valor que dé la función como resultado sea un entero.
- El atributo `INTENT(IN)` en la definición de las variables `A` y `B` de la función:

  ```
  INTEGER , INTENT(IN) :: A,B
  ```

  indica que dichas variables son variables de entrada y sus valores no pueden ser modificados por la función.

Todos los argumentos de una función deben tener este atributo para evitar que inadvertidamente sus valores se modifiquen al evaluar la función.

- Es posible definir funciones que sean *internas*, esto es, que se restrinjan a un determinado segmento de código, y no puedan ser llamadas desde otro punto del programa. Para ello se utiliza la orden `CONTAINS` como en los ejemplos 'Programa ejemplo_8_4.f90' en la página siguiente y 'Programa ejemplo_8_5.f90' en la página siguiente. El primero de estos dos programas define una función con la que calcular la energía de un nivel vibracional teniendo en cuenta la frecuencia *we* y la anarmonicidad *wexe*. El segundo, dado un número entero, calcula los factores primos de dicho número. En este ejemplo 'Programa ejemplo_8_5.f90' en la página siguiente podemos ver también el uso de un bucle del tipo *REPEAT UNTIL*.

8.3. Programas usados como ejemplo.

8.3.1. Programa ejemplo_8_1.f90

```
PROGRAM EJEMPLO_8_1
  IMPLICIT NONE
  !Definicion de variables
  INTEGER, PARAMETER :: Long=SELECTED_REAL_KIND(18,310)
  !
  REAL (KIND=Long), PARAMETER :: DPI = ACOS(-1.0_Long) ! Definimos el número Pi
  REAL (KIND=Long) :: DANGLE, DANGLERAD
  !
  REAL, PARAMETER :: PI = ACOS(-1.0) ! Definimos el número Pi
  REAL :: ANGLERAD
  !
  PRINT*, 'INTRODUZCA UN ANGULO (EN GRADOS)'
  READ*, DANGLE
  PRINT*
  ! PASO A RADIANES
  DANGLERAD = DPI*DANGLE/180.0_Long
  ANGLERAD = PI*DANGLE/180.0
  !
  PRINT 20, DANGLE, DANGLERAD
  PRINT 21, DANGLE, ANGLERAD
  PRINT*
  PRINT*
  !
  PRINT 22, DANGLERAD, SIN(DANGLERAD), COS(DANGLERAD), SIN(DANGLERAD)**2+COS(DANGLERAD)**2,&
       1.0_Long-(SIN(DANGLERAD)**2+COS(DANGLERAD)**2)
  PRINT*
  PRINT 22, ANGLERAD, SIN(ANGLERAD), COS(ANGLERAD), SIN(ANGLERAD)**2+COS(ANGLERAD)**2,1.0 - (SIN(ANGLERAD)**2+COS(ANGLERAD)**2)
  !
20 FORMAT (1X, 'UN ANGULO DE ',F14.8,' GRADOS = ', F14.8, ' RADIANES. (dp)')
21 FORMAT (1X, 'UN ANGULO DE ',F14.8,' GRADOS = ', F14.8, ' RADIANES. (sp)')
22 FORMAT (1X, 'ANGULO ',F14.8,', SIN = ', F13.9, ', COS =',F13.9,/'SIN**2+COS**2 = ', F16.12, ', 1 - SIN**2+COS**2 = ', F16.12)
END PROGRAM EJEMPLO_8_1
```

8.3.2. Programa ejemplo_8_2.f90

```
PROGRAM EJEMPLO_8_2
  IMPLICIT NONE
  !Definicion de variables
  INTEGER , PARAMETER :: NEL=5
  REAL, PARAMETER :: PI = ACOS(-1.0) ! Definimos el número Pi
  REAL , DIMENSION(1:NEL)   :: XR = (/ 0.0, PI/2.0, PI, 3.0*PI/2.0, 2.0*PI/)
  INTEGER , DIMENSION(1:NEL):: XI = (/ 0, 1, 2, 3, 4/)
  !
  PRINT*, 'SENO DE ', XR, ' = ', SIN(XR)
  PRINT*, 'LOG10 DE ', XR, ' = ', LOG10(XR)
  PRINT*, 'REAL ', XI, ' = ', REAL(XI)
END PROGRAM EJEMPLO_8_2
```

8.3.3. Programa ejemplo_8_3.f90

```
PROGRAM EJEMPLO_8_3
  IMPLICIT NONE
  INTEGER :: I,J,Result
  INTEGER :: MCD
  EXTERNAL MCD
  PRINT *,' INTRODUCE DOS NUMEROS ENTEROS:'
  READ(*,*),I,J
  RESULT = MCD(I,J)
  PRINT *,' EL MAX COMUN DIV DE ',I,' Y ',J,' ES ',RESULT
END PROGRAM EJEMPLO_8_3
!
INTEGER FUNCTION MCD(A,B)
IMPLICIT NONE
INTEGER , INTENT(IN) :: A,B
INTEGER :: Temp
  IF (A < B) THEN
     Temp=A
  ELSE
     Temp=B
  ENDIF
  DO WHILE ((MOD(A,Temp) /= 0) .OR. (MOD(B,Temp) /=0))
     Temp=Temp-1
  END DO
  MCD=Temp
END FUNCTION MCD
```

8.3.4. Programa ejemplo_8_4.f90

```
PROGRAM EJEMPLO_8_4
  IMPLICIT NONE
  ! Uso de una función interna en el cálculo de la fórmula
  ! E(v) = we (v+1/2) - wexe (v+1/2)**2.
  INTEGER :: V, VMAX
  REAL :: we, wexe, Energy
  PRINT *,' INTRODUCE EL VALOR DE Vmax:'
  READ(*,*), VMAX
  PRINT *,' INTRODUCE EL VALOR DE we Y DE wexe:'
  READ(*,*),we, wexe
  DO V = 0, VMAX
     Energy = FEN(V)
     PRINT 100, V, Energy
  ENDDO
100 FORMAT(1X,'E(',I3,') = ',F14.6)
CONTAINS
!
  REAL FUNCTION FEN(V)
    IMPLICIT NONE
    INTEGER , INTENT(IN) :: V
    FEN = we*(V+0.5)-wexe*(V+0.5)**2
  END FUNCTION FEN
!
END PROGRAM EJEMPLO_8_4
```

8.3.5. Programa ejemplo_8_5.f90

```
PROGRAM EJEMPLO_8_5
  !
  ! Simple program to compute the prime divisors of a given integer number.
  !
  IMPLICIT NONE
  INTEGER :: NUMVAL
  INTEGER :: NUM
  !
  READ(*,*), NUMVAL ! input
  !
  DO
     NUM = QUOT(NUMVAL)
     IF (NUM == NUMVAL) THEN
        PRINT*, NUM
        EXIT
     ELSE
        PRINT*, NUMVAL/NUM, NUM
        NUMVAL = NUM
     ENDIF
  ENDDO
  !
CONTAINS
  !
```

```
  INTEGER FUNCTION QUOT(NUM1)
    !
    INTEGER, INTENT(IN) :: NUM1
    INTEGER :: I
    !
    QUOT = NUM1
    !
    DO I = 2, NUM1-1
       IF (MOD(NUM1,I) == 0) THEN
          QUOT = NUM1/I
          EXIT
       ENDIF
    ENDDO
    !
  END FUNCTION QUOT
  !
END PROGRAM EJEMPLO_8_5
```

8.3.6. Programa ejemplo_8_6.f90

```
PROGRAM EJEMPLO_8_6
  !
  ! Program to evaluate a 1D potential function on grid points
  !
  IMPLICIT NONE
  !
  REAL, DIMENSION(:), ALLOCATABLE :: X_grid, Pot_grid
  !
  REAL :: X_min, X_max, Delta_X
  REAL :: V_0 = 10.0, a_val = 1.0
  INTEGER :: Index, X_dim
  INTEGER :: Ierr
  !
  !
  INTERFACE Potf
     ELEMENTAL FUNCTION Potf(Depth, Inv_length, X)
       !
       IMPLICIT NONE
       !
       REAL, INTENT(IN) :: Depth, Inv_length, X
       REAL :: Potf
       !
     END FUNCTION Potf
  END INTERFACE Potf
  !
  !
  READ(*,*), X_min, X_max, X_dim ! input minimum and maximum values of X and number of points
  !
  ALLOCATE(X_grid(1:X_dim), STAT = Ierr)
  IF (Ierr /= 0) THEN
     STOP 'X_grid allocation failed'
  ENDIF
  !
  ALLOCATE(Pot_grid(1:X_dim), STAT = Ierr)
  IF (Ierr /= 0) THEN
     STOP 'Pot_grid allocation failed'
  ENDIF
  !
  !
  Delta_X = (X_max - X_min)/REAL(X_dim - 1)
  !
  X_grid = (/ (Index, Index = 0 , X_dim - 1 ) /)
  X_grid = X_min + Delta_X*X_grid
  !
  Pot_grid = Potf(V_0, a_val, X_grid)
  !
  DO Index = 1, X_dim
     PRINT*, X_grid, Pot_grid
  ENDDO
  !
  DEALLOCATE(X_grid, STAT = Ierr)
  IF (Ierr /= 0) THEN
     STOP 'X_grid deallocation failed'
  ENDIF
  !
  DEALLOCATE(Pot_grid, STAT = Ierr)
  IF (Ierr /= 0) THEN
     STOP 'Pot_grid deallocation failed'
  ENDIF
  !
  !
END PROGRAM EJEMPLO_8_6
!
ELEMENTAL FUNCTION Potf(Depth, Inv_length, X)
```

```
    !
    IMPLICIT NONE
    !
    REAL, INTENT(IN) :: Depth, Inv_length, X
    !
    REAL :: Potf
    !
    Potf = -Depth/(COSH(Inv_length*X)**2)
    !
  END FUNCTION Potf
```

Capítulo 9

Subprogramas (II): subrutinas

9.1. Objetivos

Los objetivos de esta clase son los siguientes:

1 Considerar la diferencia entre funciones y subrutinas y por qué son precisas estas últimas.

2 Introducir los conceptos e ideas más útiles en la definición de subrutinas.

3 Argumentos de una subrutina.

4 Los comandos `CALL` e `INTERFACE`.

5 Alcance (*scope*) de las variables.

6 Variables locales y el atributo `SAVE`

7 Diferentes formas de transmitir matrices como argumentos a una subrutina.

8 Definición de matrices automáticas.

9.2. Puntos destacables.

1 El uso de subrutinas favorece una programación estructurada, mediante la definición de subtareas y su realización en las correspondientes subrutinas y evitando con su uso la duplicación innecesaria de código. Además hacen posible el uso de una extensa colección de librerías o bibliotecas de subrutinas programadas y extensamente probadas para una enorme cantidad de posibles aplicaciones.

2 Para explicar este punto vamos a usar un ejemplo práctico, como es el de la solución de una ecuación de segundo grado. Una posible forma de dividir este programa en subtareas es la siguiente:

   1 Programa principal.
   2 Input de los coeficientes de la ecuación por el usuario.
   3 Solución de la ecuación.
   4 Impresión de las soluciones.

   El programa 'Programa ejemplo_9_1.f90' en la página 48 se ajusta a este esquema usando dos subrutinas, llamadas `Interact` y `Solve`.

3 La definición de una subrutina tiene la siguiente estructura:

```
SUBROUTINE nombre_subrutina(lista de argumentos [opcional])
        IMPLICIT NONE
        Arguments (dummy variables) definition (INTENT)
        ...
        Local variables definition
        ...
        Execution Section
        ...
        [RETURN]
 END SUBROUTINE nombre_subrutina
```

Los argumentos se denominan *dummy arguments* porque su definición no implica la asignación de memoria alguna. Esta asignación se llevará a cabo de acuerdo con los valores que tomen los argumentos cuando se llame a la subrutina.

Cuando el compilador genera el ejecutable cada subrutina se compila de forma separada lo que permite el uso de *variables locales* con el mismo nombre en diferentes subrutinas, ya que cada subrutina tiene su particular alcance (*scope*).

En el programa 'Programa ejemplo_9_1.f90' en la página 48 se ve como este esquema se repite para las dos subrutinas empleadas.

4 Para invocar una subrutina se emplea el comando `CALL` de acuerdo con el esquema

```
CALL nombre_subroutina(argumentos [opcional])
```

Tras la ejecución de la subrutina invocada con la orden `CALL`, el flujo del programa retorna a la unidad de programa en la que se ha invocado a la subrutina y continúa en la orden siguiente al comando en el que se ha llamado la subrutina con `CALL`. Desde la subrutina se devuelve la ejecución con el comando `RETURN`. Si la subrutina llega a su fin también se devuelve el control al programa que la ha invocado, por lo que generalmente no se incluye el comando `RETURN` justo antes de `END SUBROUTINE`. Si es posible, las subrutinas deberían tener un solo punto de salida.

5 La subrutina y el programa principal se comunican a través de los argumentos (también llamados parámetros) de la subrutina. En la definición de la subrutina dichos argumentos son *dummies*, encerrados entre paréntesis y separados con comas tras el nombre de la subrutina. Dichos argumentos tienen un tipo asociado, pero *NO* se reserva ningún espacio para ellos en memoria. Por ejemplo, los argumentos `E`, `F` y `G` de la subrutina `Solve` en el ejemplo 'Programa ejemplo_9_1.f90' en la página 48 son del tipo `REAL`, pero no se reserva para ellos ningún espacio en memoria. Cuando la subrutina es invocada con el comando `CALL Solve(P,Q,R,Root1,Root2,IFail)` entonces los argumentos `E`, `F` y `G` pasan a ser reemplazados por unos punteros a las variables `P`, `Q` y `R`. Por tanto es muy importante que el tipo de los argumentos y el de las variables por las que se ven reemplazados coincidan, ya que cuando esto no sucede se producen frecuentes errores.

6 Alguno de los argumentos proporcionan una información de entrada (input) a la subrutina, mientras que otros proporcionan la salida de la subrutina (output). Por último, también es posible que los argumentos sean simultáneamente de entrada y salida.

Aquellos parámetros que solo sean de entrada es conveniente definirlos con el atributo `INTENT(IN)`. Este atributo ya lo vimos en 'Subprogramas (I): funciones' en la página 39 aplicándolo a funciones. Cuando un argumento posee este atributo el valor de entrada del parámetro se mantiene constante y no puede variar en la ejecución de la subrutina.

Si los parámetros solo son de salida es conveniente definirlos con el atributo `INTENT(OUT)`, para que se ignore el valor de entrada del parámetro y debe dársele uno durante la ejecución de la subrutina.

Si el parámetro tiene el atributo `INTENT(INOUT)`, entonces se considera el valor inicial del parámetro así como su posible modificación en la subrutina.

Hay ejemplos de los tres casos arriba citados en la subrutina `Solve` del ejemplo 'Programa ejemplo_9_1.f90' en la página 48. Es muy conveniente etiquetar con el atributo `INTENT` todos los argumentos.

7 De acuerdo con lo anterior es de vital importancia que no exista contradicción entre la declaración de variables en el programa que invoca a la subrutina y en la propia subrutina. Para facilitar este acuerdo entre ambas declaraciones existen los llamados *interface blocks*. En el programa 'Programa ejemplo_9_2.f90' en la página 49 podemos ver el programa 'Programa ejemplo_9_1.f90' en la página 48 al que se han añadido en el programa principal los *interface blocks* correspondientes a las subrutinas `Interact` y `Solve`.

8 Al igual que en el caso de las funciones, las variables declaradas en una subrutina que no sean parámetros o argumentos de la misma se consideran locales. Por ejemplo, en la subrutina `Interact` del 'Programa ejemplo_9_1.f90' en la página 48 la variable `IO_Status` es una variable local de la subrutina.

Generalmente las variables locales se crean al invocarse la subrutina y el valor que adquieren se pierde una vez que la subrutina se ha ejecutado. Sin embargo, usando el atributo `SAVE` es posible salvar el valor que adquiera la variable de una llamada a la subrutina hasta la siguiente llamada. Por ejemplo

```
INTEGER, SAVE:: It = 0
```

El valor que tome en este caso la variable `It` entre llamadas al subprograma en el que se haya declarado se conserva.

Como en el caso de las funciones, es posible hacer que el programa principal "conozca" las variables de las subrutinas que invoque mediante la orden `CONTAINS` y haciendo que de hecho las subrutinas formen parte del programa principal. Esta solución resulta difícil de escalar cuando crece la longitud del problema y no es recomendable.

9 Cuando el argumento de una subrutina no es una variable escalar (del tipo que fuera) sino una matriz (*array*) es necesario dar una información extra acerca de la matriz. El subprograma al que se pasa la matriz ha de conocer el tamaño de la matriz para no acceder a posiciones de memoria erróneas. Para conseguir esto hay tres posibles formas de especificar las dimensiones de una matriz que se halle en la lista de argumentos de una subrutina:

1 *explicit-shape approach*:

En este caso se incluyen como argumentos en la llamada a la subrutina las dimensiones de las matrices implicadas, declarando posteriormente las matrices haciendo uso de dichas dimensiones. Por ejemplo, si en una subrutina llamada `test_pass` se incluye un vector de entrada llamado `space_vec_in` y uno de salida`space_vec_out` con la misma dimensión, si hacemos uso del *explicit-shape approach* la subrutina comenzaría como

```
SUBROUTINE test_pass(space_vec_in, space_vec_out, dim_vec)
         IMPLICIT NONE
         INTEGER, INTENT(IN) :: dim_vec
         REAL, INTENT(IN), DIMENSION(1:dim_vec) :: space_vec_in
         REAL, INTENT(OUT), DIMENSION(1:dim_vec) :: space_vec_out
         ......
 END SUBROUTINE test_pass
```

2 *assumed-shape approach*:

En este caso es necesario incluir el correspondiente bloque `INTERFACE` en el subprograma que invoca la subrutina. Como veremos en el 'Subprogramas (III): módulos' en la página 55 esto se puede evitar incluyendo la subrutina en un módulo.

En el 'Programa ejemplo_9_3.f90' en la página 51 puede verse un programa en el que se calcula la media, la mediana[1], la varianza y la desviación estándar de un conjunto de números generados aleatoriamente. En el programa hemos marcado algunos de los puntos de interés que queremos explicar con detalle.

- *(1-3)* Hemos definido la matriz con dimensión variable, de forma que se dimensione mediante una orden `ALLOCATE`. En la orden que dimensiona a la matriz se indica que es un vector (`DIMENSION(:)`) y del mismo modo se hace en el *interface block*. El uso del *interface block* es recomendable, y en casos como este, con matrices definidas de este modo, resulta obligatorio. La orden *(3)*, `ALLOCATE(X(1:N), STAT = IERR)` hace que `X` pase a ser un vector N-dimensional. Usamos también el campo opcional `STAT` que nos permita saber si se ha podido dimensionar el arreglo solicitado. Solo si la salida (`IERR`) es cero la matriz se ha creado sin problemas. El uso de esta opción debe generalizarse.

```
REAL , ALLOCATABLE , DIMENSION(:) :: X  !! (1)

...

INTERFACE
   SUBROUTINE STATS(X,N,MEAN,STD_DEV,MEDIAN)
     IMPLICIT NONE

     ...

     REAL      , INTENT(IN) , DIMENSION(:)   ::    X  !!  (1)

     ...

   END SUBROUTINE STATS
END INTERFACE
```

Es importante tener en cuenta que se puede definir como `ALLOCATABLE` el argumento con el que se llama a una subrutina, así como a variables internas o locales de la subrutina, pero una variable *dummy* no puede tener este atributo.

A diferencia de en `FORTRAN77`, la forma recomendada de transmitir arreglos de datos entre un programa y una subrutina es como en el ejemplo, usando *assumed shape arguments* en los que no se da ninguna información acerca del tamaño del arreglo. Sí deben coincidir ambas variables en tipo, rango y clase (`KIND`).

[1] Se define la *mediana* de un conjunto de números como aquel valor de la lista tal que la mitad de los valores sean inferiores a él y la otra mitad sean superiores. Coincide con el valor medio en distribuciones simétricas. Para su cálculo es preciso ordenar previamente la lista de números.

- *(4)* y *(6)*: En estas órdenes se aprovecha la capacidad de `Fortran90` para trabajar con arreglos de variables, ya sean estos vectores o matrices. Por ejemplo, el comando `X=X*1000` multiplica todas las componentes del vector `X` por un escalar y el comando `SUMXI=SUM(X)` aprovecha la función `SUM` para sumar las componentes del vector. En estilo `Fortran 77` estas operaciones conllevarían un bucle `DO`, por ejemplo

```
SUMXI = 0.0
DO I = 1, N
   SUMXI = SUMXI + X(I)
ENDDO
```

- *(5)* En esta parte del programa se libera la memoria reservada para el vector `X` usando el comando `DEALLOCATE`. Este paso no es obligatorio en este programa, pero sí cuando la matriz del tipo `ALLOCATE` se ha definido en una función o subrutina y no tiene el atributo `SAVE`.
- *(7)* Aquí se aprovecha el comando `CONTAINS` para hacer que la subrutina de ordenamiento `SELECTION`, que como puede verse no posee argumentos, *conozca* las misma variables que la subrutina `STATS`, en la que está contenida. Por ello, en la subrutina `SELECTION` solo es preciso definir las variables locales. Esta subrutina se encarga de ordenar la lista de números según un algoritmo que consiste en buscar el número más pequeño de la lista y hacerlo el primer miembro. Se busca a continuación el más pequeño de los restantes que pasa a ser segundo, y así prosigue hasta tener ordenada la lista de números.

La definición de bloques `INTERFACE` se facilita con el uso de módulos, que describimos en la siguiente unidad.

3 *assumed-size approach*

En este caso no se da información a la subrutina acerca de las dimensiones de la matriz, es fácil caer en errores de difícil diagnóstico y se desaconseja su uso.

10 Arreglos multidimensionales. El 'Programa ejemplo_9_5.f90' en la página 52 es un ejemplo de como pasar como argumentos arreglos multidimensionales como *assumed shape arrays*. En él, tras que el usuario defina dos matrices, `A` y `B`, el programa calcula la matriz `C` solución del producto `AB` y tras ello calcula la matriz traspuesta de `A`. Se hace uso de las funciones de `Fortran 90` `MATMUL` y `TRANSPOSE`.

11 En las subrutinas pueden dimensionarse *automatic arrays*, que pueden depender de los argumentos de la subrutina. Estos arreglos son locales a la subrutina, no pueden tener el argumento `SAVE` y se crean cada vez que se invoca la subrutina, siendo destruidos al salir de ella. Esto hace que si no hay memoria suficiente para dimensionar el arreglo el programa no funcione. Para evitar esto deben definirse arreglos no automáticos, del tipo `ALLOCATABLE`.

12 Al pasar como argumento una variable de tipo `CHARACTER` dicho argumento se declara con una longitud `LEN = *` y cuando se llame a la subrutina la longitud de la variable pasa a ser la longitud de la variable en la llamada.

El 'Programa ejemplo_9_4.f90' en la página 52 muestra un programa en el que, al darle el nombre de un fichero y el número de datos almacenados en dicho fichero; el programa abre el fichero y lee dos columnas de valores que almacena en los vectores `X` e `Y`. En estos casos, dado que el tamaño de la variable `CHARACTER` es variable, es preciso usar un *interface block*.

El 'Programa ejemplo_9_6.f90' en la página 53 es un ejemplo donde se construyen dos vectores de números aleatorios de dimensión definida por el usuario usando el método *Box-Mueller*. Para ello se definen dos matrices de tipo `ALLOCATABLE`, `X` e `Y`, y en la subrutina interna `BOX_MULLER` se definen dos vectores de tipo automático: `RANDOM_u` y `RANDOM_v`.

Para calcular el valor medio, la desviación estándar y la mediana de los vectores `X` e `Y` se hace uso de la subrutina `STATS` del 'Programa ejemplo_9_3.f90' en la página 51. Se incluye el necesario `INTERFACE` en el programa principal y la subrutina se debe compilar en un fichero por separado. 'Programa ejemplo_9_6.f90' en la página 53

13 Sí es importante tener en cuenta que en el caso que se transfiera un *array* usando *assumed shape arguments* como en los ejemplos, el primer índice de la variable en la subrutina *se supone que comienza con el valor* `1`*, a menos que explícitamente se indique lo contrario*. En el ejemplo 'Programa ejemplo_9_7.f90' en la página 54 se muestra un caso simple donde es necesario indicar el índice inicial del vector cuando este no es cero. En este programa se calcula el factorial de los enteros entre `IMIN` e `IMAX` y se almacenan en un vector real. Se puede compilar y correr el programa haciendo `IMIN = 1` e `IMIN = 0` con y sin la definición del índice inicial en la subrutina, para ver la diferencia en las salidas.

9.3. Programas usados como ejemplo.

9.3.1. Programa ejemplo_9_1.f90

```
PROGRAM ejemplo_9_1
```

```
!
IMPLICIT NONE
! Ejemplo simple de un programa con dos subrutinas.
! subrutina (1):: Interact :: Obtiene los coeficientes de la ec. de seg. grado.
! subrutina (2):: Solve :: Resuelve la ec. de seg. grado.
!
! Definicion de variables
REAL :: P, Q, R, Root1, Root2
INTEGER :: IFail=0
LOGICAL :: OK=.TRUE.
!
   CALL Interact(P,Q,R,OK) ! Subrutina (1)
!
   IF (OK) THEN
!
      CALL Solve(P,Q,R,Root1,Root2,IFail) ! Subrutina (2)
!
      IF (IFail == 1) THEN
         PRINT *,' Complex roots'
         PRINT *,' calculation aborted'
      ELSE
         PRINT *,' Roots are ',Root1,' ',Root2
      ENDIF
!
   ELSE
!
      PRINT*,' Error in data input program ends'
!
   ENDIF
!
END PROGRAM ejemplo_9_1
!
!
SUBROUTINE Interact(A,B,C,OK)
   IMPLICIT NONE
   REAL , INTENT(OUT) :: A
   REAL , INTENT(OUT) :: B
   REAL , INTENT(OUT) :: C
   LOGICAL , INTENT(OUT) :: OK
   INTEGER :: IO_Status=0
   PRINT*,' Type in the coefficients A, B AND C'
   READ(UNIT=*,FMT=*,IOSTAT=IO_Status)A,B,C
   IF (IO_Status == 0) THEN
      OK=.TRUE.
   ELSE
      OK=.FALSE.
   ENDIF
END SUBROUTINE Interact
!
!
SUBROUTINE Solve(E,F,G,Root1,Root2,IFail)
   IMPLICIT NONE
   REAL , INTENT(IN) :: E
   REAL , INTENT(IN) :: F
   REAL , INTENT(IN) :: G
   REAL , INTENT(OUT) :: Root1
   REAL , INTENT(OUT) :: Root2
   INTEGER , INTENT(INOUT) :: IFail
! Local variables
   REAL :: Term
   REAL :: A2
   Term = F*F - 4.*E*G
   A2 = E*2.0
! if term < 0, roots are complex
   IF(Term < 0.0)THEN
      IFail=1
   ELSE
      Term = SQRT(Term)
      Root1 = (-F+Term)/A2
      Root2 = (-F-Term)/A2
   ENDIF
END SUBROUTINE Solve
```

9.3.2. Programa ejemplo_9_2.f90

```
PROGRAM ejemplo_9_2
  !
  IMPLICIT NONE
  ! Ejemplo simple de un programa con dos subrutinas.
  ! subrutina (1):: Interact :: Obtiene los coeficientes de la ec. de seg. grado.
  ! subrutina (2):: Solve :: Resuelve la ec. de seg. grado.
  !
  ! Interface blocks
  INTERFACE
```

```
      SUBROUTINE Interact(A,B,C,OK)
        IMPLICIT NONE
        REAL , INTENT(OUT) ::  A
        REAL , INTENT(OUT) ::  B
        REAL , INTENT(OUT) ::  C
        LOGICAL , INTENT(OUT) :: OK
      END SUBROUTINE Interact
      SUBROUTINE Solve(E,F,G,Root1,Root2,IFail)
        IMPLICIT NONE
        REAL , INTENT(IN) :: E
        REAL , INTENT(IN) :: F
        REAL , INTENT(IN) :: G
        REAL , INTENT(OUT) :: Root1
        REAL , INTENT(OUT) :: Root2
        INTEGER , INTENT(INOUT) :: IFail
      END SUBROUTINE Solve
    END INTERFACE
    ! Fin interface blocks
    !
    ! Definicion de variables
    REAL :: P, Q, R, Root1, Root2
    INTEGER :: IFail=0
    LOGICAL :: OK=.TRUE.
    !
    CALL Interact(P,Q,R,OK) ! Subrutina (1)
    !
    IF (OK) THEN
       !
       CALL Solve(P,Q,R,Root1,Root2,IFail) ! Subrutina (2)
       !
       IF (IFail == 1) THEN
          PRINT *,' Complex roots'
          PRINT *,' calculation aborted'
       ELSE
          PRINT *,' Roots are ',Root1,' ',Root2
       ENDIF
       !
    ELSE
       !
       PRINT*,' Error in data input program ends'
       !
    ENDIF
    !
  END PROGRAM ejemplo_9_2
  !
  !
  SUBROUTINE Interact(A,B,C,OK)
    IMPLICIT NONE
    REAL , INTENT(OUT) :: A
    REAL , INTENT(OUT) :: B
    REAL , INTENT(OUT) :: C
    LOGICAL , INTENT(OUT) :: OK
    INTEGER :: IO_Status=0
    PRINT*,' Type in the coefficients A, B AND C'
    READ(UNIT=*,FMT=*,IOSTAT=IO_Status)A,B,C
    IF (IO_Status == 0) THEN
       OK=.TRUE.
    ELSE
       OK=.FALSE.
    ENDIF
  END SUBROUTINE Interact
  !
  !
  SUBROUTINE Solve(E,F,G,Root1,Root2,IFail)
    IMPLICIT NONE
    REAL , INTENT(IN) :: E
    REAL , INTENT(IN) :: F
    REAL , INTENT(IN) :: G
    REAL , INTENT(OUT) :: Root1
    REAL , INTENT(OUT) :: Root2
    INTEGER , INTENT(INOUT) :: IFail
    ! Local variables
    REAL :: Term
    REAL :: A2
    Term = F*F - 4.*E*G
    A2 = E*2.0
    ! if term < 0, roots are complex
    IF(Term < 0.0)THEN
       IFail=1
    ELSE
       Term = SQRT(Term)
       Root1 = (-F+Term)/A2
       Root2 = (-F-Term)/A2
    ENDIF
  END SUBROUTINE Solve
```

9.3.3. Programa ejemplo_9_3.f90

```
PROGRAM ejemplo_9_3
  !
  IMPLICIT NONE
  !
  ! Definicion de variables
  INTEGER :: N
  REAL , ALLOCATABLE , DIMENSION(:) :: X  !! (1)
  REAL :: M,SD,MEDIAN
  INTEGER :: IERR
  !
  ! interface block   !! (2)
  INTERFACE
     SUBROUTINE STATS(VECTOR,N,MEAN,STD_DEV,MEDIAN)
       IMPLICIT NONE
       INTEGER , INTENT(IN)                    ::  N
       REAL      , INTENT(IN) , DIMENSION(:)   :: VECTOR  !!  (1)
       REAL      , INTENT(OUT)                 :: MEAN
       REAL      , INTENT(OUT)                 :: STD_DEV
       REAL      , INTENT(OUT)                 :: MEDIAN
     END SUBROUTINE STATS
  END INTERFACE
  PRINT *,' Cuántos valores vas a generar aleatoriamente ?'
  READ(*,*),N
  ALLOCATE(X(1:N), STAT = IERR)     !!  (3)
  IF (IERR /= 0) THEN
     PRINT*, "X allocation request denied."
     STOP
  ENDIF
  CALL RANDOM_NUMBER(X)
  X=X*1000             !!  (4)
  CALL STATS(X,N,M,SD,MEDIAN)
!
  PRINT *,' MEAN = ',M
  PRINT *,' STANDARD DEVIATION = ',SD
  PRINT *,' MEDIAN IS = ',MEDIAN
!
  IF (ALLOCATED(X)) DEALLOCATE(X, STAT = IERR)   !! (5)
  IF (IERR /= 0) THEN
     PRINT*, "X NON DEALLOCATED!"
     STOP
  ENDIF
END PROGRAM ejemplo_9_3
!
SUBROUTINE STATS(VECTOR,N,MEAN,STD_DEV,MEDIAN)
  IMPLICIT NONE
  ! Defincion de variables
  INTEGER , INTENT(IN)                    ::  N
  REAL      , INTENT(IN) , DIMENSION(:)    ::  VECTOR    !! (1)
  REAL      , INTENT(OUT)                   ::  MEAN
  REAL      , INTENT(OUT)                   ::  STD_DEV
  REAL      , INTENT(OUT)                   ::  MEDIAN
  REAL      , DIMENSION(1:N)               ::  Y
  REAL :: VARIANCE = 0.0
  REAL      :: SUMXI = 0.0, SUMXI2 = 0.0
  !
  SUMXI=SUM(VECTOR)       !! (6)
  SUMXI2=SUM(VECTOR*VECTOR)    !! (6)
  MEAN=SUMXI/N
  VARIANCE=(SUMXI2-SUMXI*SUMXI/N)/(N-1)
  STD_DEV = SQRT(VARIANCE)
  Y=VECTOR
  ! Ordena valores por proceso de seleccion
  CALL SELECTION
  IF (MOD(N,2) == 0) THEN
     MEDIAN=(Y(N/2)+Y((N/2)+1))/2
  ELSE
     MEDIAN=Y((N/2)+1)
  ENDIF
CONTAINS     !! (7)
  SUBROUTINE SELECTION
    IMPLICIT NONE
    INTEGER :: I,J,K
    REAL :: MINIMUM
    DO I=1,N-1
       K=I
       MINIMUM=Y(I)
       DO J=I+1,N
          IF (Y(J) < MINIMUM) THEN
             K=J
             MINIMUM=Y(K)
          END IF
       END DO
       Y(K)=Y(I)
       Y(I)=MINIMUM
```

```
    END DO
  END SUBROUTINE SELECTION
END SUBROUTINE STATS
```

9.3.4. Programa ejemplo_9_4.f90

```
PROGRAM ejemplo_9_4
  IMPLICIT NONE
  REAL,DIMENSION(1:100)::A,B
  INTEGER :: Nos,I
  CHARACTER(LEN=32)::Filename
  INTERFACE
     SUBROUTINE Readin(Name,X,Y,N)
       IMPLICIT NONE
       INTEGER , INTENT(IN) :: N
       REAL,DIMENSION(1:N),INTENT(OUT)::X,Y
       CHARACTER (LEN=*),INTENT(IN)::Name
     END SUBROUTINE Readin
  END INTERFACE
  PRINT *,' Type in the name of the data file'
  READ '(A)' , Filename
  PRINT *,' Input the number of items in the file'
  READ(*,*) , Nos
  CALL Readin(Filename,A,B,Nos)
  PRINT * , ' Data read in was'
  DO I=1,Nos
     PRINT *,' ',A(I),' ',B(I)
  ENDDO
END PROGRAM ejemplo_9_4
SUBROUTINE Readin(Name,X,Y,N)
  IMPLICIT NONE
  INTEGER , INTENT(IN) :: N
  REAL,DIMENSION(1:N),INTENT(OUT)::X,Y
  CHARACTER (LEN=*),INTENT(IN)::Name
  INTEGER::I
  OPEN(UNIT=10,STATUS='OLD',FILE=Name)
  DO I=1,N
     READ(10,*)X(I),Y(I)
  END DO
  CLOSE(UNIT=10)
END SUBROUTINE Readin
```

9.3.5. Programa ejemplo_9_5.f90

```
PROGRAM ejemplo_9_5
  IMPLICIT NONE
  REAL , ALLOCATABLE , DIMENSION &
       (:,:)::One,Two,Three,One_T
  INTEGER :: I,N
  INTERFACE
     SUBROUTINE Matrix_bits(A,B,C,A_T)
       IMPLICIT NONE
       REAL, DIMENSION (:,:), INTENT(IN) :: A,B
       REAL, DIMENSION (:,:), INTENT(OUT) :: C,A_T
     END SUBROUTINE Matrix_bits
  END INTERFACE
  PRINT *,'Dimensión de las matrices'
  READ*,N
  ALLOCATE(One(1:N,1:N))
  ALLOCATE(Two(1:N,1:N))
  ALLOCATE(Three(1:N,1:N))
  ALLOCATE(One_T(1:N,1:N))
  DO I=1,N
     PRINT*, 'Fila ', I,' de la primer matriz?'
     READ*,One(I,1:N)
  END DO
  DO I=1,N
     PRINT*, 'Fila ', I,' de la segunda matriz?'
     READ*,Two(I,1:N)
  END DO
  CALL Matrix_bits(One,Two,Three,One_T)
  PRINT*,' Resultado: Matriz Producto:'
  DO I=1,N
     PRINT *,Three(I,1:N)
  END DO
  PRINT *,' Matriz traspuesta A^T:'! Calcula la matriz transpuesta.
  DO I=1,N
     PRINT *,One_T(I,1:N)
  END DO
END PROGRAM ejemplo_9_5
!
```

```
SUBROUTINE Matrix_bits(A,B,C,A_T)
  IMPLICIT NONE
  REAL, DIMENSION (:,:), INTENT(IN) :: A,B
  REAL, DIMENSION (:,:), INTENT(OUT) :: C,A_T
  C=MATMUL(A,B)
  A_T=TRANSPOSE(A)
END SUBROUTINE Matrix_bits
```

9.3.6. Programa ejemplo_9_6.f90

```
PROGRAM ejemplo_9_6
  !
  IMPLICIT NONE
  !
  INTEGER :: I, IERR
  REAL, DIMENSION(:), ALLOCATABLE :: X, Y
  REAL :: M, SD, MEDIAN
  ! interface block
  INTERFACE
     SUBROUTINE STATS(VECTOR,N,MEAN,STD_DEV,MEDIAN)
       IMPLICIT NONE
       INTEGER , INTENT(IN)                    ::  N
       REAL      , INTENT(IN) , DIMENSION(:)   :: VECTOR
       REAL      , INTENT(OUT)                 :: MEAN
       REAL      , INTENT(OUT)                 :: STD_DEV
       REAL      , INTENT(OUT)                 :: MEDIAN
     END SUBROUTINE STATS
  END INTERFACE
  !
  READ*, I
  !
  ALLOCATE(X(1:I), STAT = IERR)
  IF (IERR /= 0) THEN
     PRINT*, "X allocation request denied."
     STOP
  ENDIF
  !
  ALLOCATE(Y(1:I), STAT = IERR)
  IF (IERR /= 0) THEN
     PRINT*, "Y allocation request denied."
     STOP
  ENDIF
  !
  CALL BOX_MULLER(I)
  !
  PRINT*, X
  CALL STATS(X,I,M,SD,MEDIAN)
  !
  PRINT *,' MEAN = ',M
  PRINT *,' STANDARD DEVIATION = ',SD
  PRINT *,' MEDIAN IS = ',MEDIAN
  !
  IF (ALLOCATED(X)) DEALLOCATE(X, STAT = IERR)
  IF (IERR /= 0) THEN
     PRINT*, "X NON DEALLOCATED!"
     STOP
  ENDIF
  PRINT*, Y
  CALL STATS(Y,I,M,SD,MEDIAN)
  !
  PRINT *,' MEAN = ',M
  PRINT *,' STANDARD DEVIATION = ',SD
  PRINT *,' MEDIAN IS = ',MEDIAN
  !
  IF (ALLOCATED(Y)) DEALLOCATE(Y, STAT = IERR)
  IF (IERR /= 0) THEN
     PRINT*, "Y NON DEALLOCATED!"
     STOP
  ENDIF
  !
CONTAINS
  !
  SUBROUTINE BOX_MULLER(dim)
    !
    ! Uses the Box-Muller method to create two normally distributed vectors
    !
    INTEGER, INTENT(IN) :: dim
    !
    REAL, PARAMETER :: PI = ACOS(-1.0)
    REAL, DIMENSION(dim) :: RANDOM_u, RANDOM_v ! Automatic arrays
    !
    CALL RANDOM_NUMBER(RANDOM_u)
    CALL RANDOM_NUMBER(RANDOM_v)
    !
```

```
      X = SQRT(-2.0*LOG(RANDOM_u))
      Y = X*SIN(2*PI*RANDOM_v)
      X = X*COS(2*PI*RANDOM_v)
      !
    END SUBROUTINE BOX_MULLER
    !
  END PROGRAM ejemplo_9_6
```

9.3.7. Programa ejemplo_9_7.f90

```
  PROGRAM EJEMPLO_9_7
    !
    IMPLICIT NONE
    !
    INTERFACE
       SUBROUTINE SUBEXAMPLE(IMIN, IMAX, FACT_MAT)
         INTEGER, INTENT(IN) :: IMIN, IMAX
         REAL, DIMENSION(IMIN:), INTENT(OUT) :: FACT_MAT
       END SUBROUTINE SUBEXAMPLE
    END INTERFACE
    !
    REAL, DIMENSION(:), ALLOCATABLE :: FACT_MAT
    INTEGER :: IMIN, IMAX, I
    !
    IMIN = 0
    IMAX = 5
    !
    ALLOCATE(FACT_MAT(IMIN:IMAX))
    !
    PRINT*, "MAIN", SIZE(FACT_MAT)
    !
    CALL SUBEXAMPLE(IMIN, IMAX, FACT_MAT)
    !
    DO I = IMIN, IMAX
       PRINT*, I, FACT_MAT(I)
    ENDDO
    !
  END PROGRAM EJEMPLO_9_7
  !!!!!!!!!!
  SUBROUTINE SUBEXAMPLE(IMIN, IMAX, FACT_MAT)
    !
    IMPLICIT NONE
    INTEGER, intent(in) :: IMIN, IMAX
    REAL, DIMENSION(IMIN:), intent(out) :: FACT_MAT
  ! The subroutine with the next line only would work for IMIN = 1
  !  REAL, DIMENSION(:), intent(out) :: FACT_MAT
    !
    INTEGER :: j,k
    !
    PRINT*, "SUB", SIZE(FACT_MAT)
    !
    DO j = imin, imax
       fact_mat(j) = 1.0
       do k = 2, j
          fact_mat(j) = k*fact_mat(j)
       enddo
    ENDDO
    !
    !
  END SUBROUTINE SUBEXAMPLE
```

Capítulo 10

Subprogramas (III): módulos

10.1. Objetivos

Los objetivos de esta clase son los siguientes:

1 Presentar los módulos y las ventajas que aportan.

2 Uso de módulos para la definición de variables. Reemplazo de bloques `COMMON`.

3 Uso de módulos para la definición de funciones y subrutinas.

4 Definición de variables públicas y privadas en módulos. Visibilidad en el módulo.

10.2. Puntos destacables.

1 La definición de módulos permite escribir código de forma más clara y flexible. En un módulo podemos encontrar

   1 Declaración global de variables.
   Reemplazan a las órdenes `COMMON` e `INCLUDE` de `FORTRAN 77`.
   2 Declaración de bloques `INTERFACE`.
   3 Declaración de funciones y subrutinas. La declaración de funciones y subrutinas en un módulo es conveniente para evitar la inclusión de los correspondientes `INTERFACE`, ya que estos están ya implícitos en el módulo.
   4 Control del acceso a los objetos, lo que permite que ciertos objetos tengan carácter público y otros privado.
   5 Los módulos permiten empaquetar tipos derivados, funciones, subrutinas para proveer de capacidades de programación orientada a objetos. Pueden también usarse para definir extensiones semánticas al lenguaje `FORTRAN`.

   La sintaxis para la declaración de un módulo es la siguiente:

```
MODULE module name
   IMPLICIT NONE
   SAVE
     declaraciones y especificaciones
   [ CONTAINS
     definición de subrutinas y funciones ]
END MODULE  module name
```

   La carga del módulo se hace mediante la orden `USE MODULE module name` que debe preceder al resto de órdenes de la unidad de programa en el que se incluya. Desde un módulo puede llamarse a otro módulo. A continuación desarrollamos brevemente estas ideas.

2 Una de las funciones de los módulos es permitir el intercambio de variables entre diferentes programas y subrutinas sin recurrir a los argumentos. La otra función principal es, haciendo uso de `CONTAINS`, definir funciones, subrutines y bloques `INTERFACE`.

La inclusión de estas unidades en un módulo hace que todos los detalles acerca de las subrutinas y funciones implicadas sean conocidas para el compilador lo que permite una más rápida detección de errores. Cuando una subrutina o una función se compila en un módulo y se hace accesible mediante `USE MODULE` se dice que tiene una interfaz explícita (*explicit interface*), mientras que en caso contrario se dice que tiene una interfaz implícita (*implicit interface*).

3 La definición de módulos favorece la llamada *encapsulación*, que consiste en definir secciones de código que resultan fácilmente aplicables en diferentes situaciones. En esto consiste la base de la llamada programación orientada a objetos. En el 'Programa ejemplo_10_1.f90' en la página 58 presentamos como se define un módulo (usando la orden `MODULE` en vez de `PROGRAM` para la definición de un *stack* de enteros. Es importante tener en cuenta como se definen en el módulo las variables `STACK_POS` y `STORE` con el atributo `SAVE`, para que su valor se conserve entre llamadas. Esto es especialmente importante cuando el módulo se llama desde una subrutina o función en vez de desde el programa principal.

4 Este módulo puede ser accedido por otra unidad de programa que lo cargue usando la orden `USE`. Debe compilarse previamente a la unidad de programa que lo cargue.

```
PROGRAM Uso_Stack
!
USE Stack     ! CARGA EL MODULO
!
IMPLICIT NONE
....
....
CALL POP(23); CAL PUSH(20)
....
....
END PROGRAM Uso_Stack
```

5 Como vemos en el 'Programa ejemplo_10_1.f90' en la página 58 las variables dentro de un módulo pueden definirse como variables privadas, con el atributo `PRIVATE`. Esto permite que no se pueda acceder a estas variables desde el código que usa el módulo. El programa que carga el módulo solo puede acceder a las subrutinas `POP` y `PUSH`. La visibilidad por defecto al definir una variable o procedimiento en un módulo es `PUBLIC`. Es posible añadir el atributo a la definición de las variables

```
INTEGER, PRIVATE, PARAMETER :: STACK_SIZE = 500
INTEGER, PRIVATE, SAVE :: STORE(STACK_SIZE) = 0, STACK_POS = 0
```

6 En ocasiones es posible que variables o procedimientos definidos en un módulo entren en conflicto con variables del programa que usa el módulo. Para evitar esto existe la posibilidad de renombrar las variables que carga el módulo, aunque esto solo debe hacerse cuando sea estrictamente necesario.

Si, por ejemplo, llamamos al módulo `Stack` desde un programa que ya tiene una variable llamada `PUSH` podemos renombrar el objeto `PUSH` del módulo a `STACK_PUSH` al invocar el módulo

```
USE Stack, STACK_PUSH => PUSH
```

Se pueden renombrar varios objetos, separándolos por comas.

7 Es posible hacer que solo algunos elementos del módulo sean accesibles desde el programa que lo invoca con la cláusula `ONLY`, donde también es posible renombrar los objetos si es necesario. Por ejemplo, con la llamada

```
USE Stack, ONLY: POP, STACK_PUSH => PUSH
```

Solamente se accede a `POP` y `PUSH`, y este último se renombra a `STACK_PUSH`.

8 Para definir variables comunes a diferentes partes de un programa se debe evitar el uso de variables en `COMMON` y, en vez de ello, se siguen los pasos siguientes.

1 Declarar las variables necesarias en un `MODULE`.

2 Otorgar a estas variables el atributo `SAVE`.

3 Cargar este módulo (`USE` *module_name*) desde aquellas unidades que necesiten acceso a estos datos globales.

Por ejemplo, si existen una serie de constantes físicas que utilizaremos en varios programas podemos definirlas en un módulo:

```
MODULE PHYS_CONST
  !
  IMPLICIT NONE
  !
  SAVE
  !
  REAL, PARAMETER :: Light_Speed = 2.99792458E08  ! m/s
  REAL, PARAMETER :: Newton_Ctnt = 6.67428E-11    ! m3 kg-1 s-2
  REAL, PARAMETER :: Planck_Ctnt = 4.13566733E-15 ! eV s
  !
  REAL :: Otra_variable
  !
END MODULE PHYS_CONST
```

En este módulo se definen tres constantes físicas (con el atributo `PARAMETER`, ya que son constantes) y una cuarta variable a la que se desea acceder que no permanece constante. En cualquier programa, función o subrutina que quieran usarse estas variables basta con cargar el módulo

```
PROGRAM CALCULUS
  !
  USE PHYS_CONST
  !
  IMPLICIT NONE
  !
  REAL DISTANCE, TIME
  !
  ...
  DISTANCE = Light_Speed*TIME
  ...
  !
END PROGRAM CALCULUS
```

9 El 'Programa ejemplo_10_2.f90' en la página 59 es un programa simple donde se utiliza el módulo para el manejo de un stack presentado para realizar operaciones (adición y substracción) con enteros en notación polaca inversa (RPN, reverse Polish notation).

Esta notación permite no usar paréntesis en las operaciones algebraicas y resulta más rápida que la notación usual. Si, por ejemplo, en el stack existen los números `(23, 10, 33)` y tenemos en cuenta que un stack se rige por el principio *last in, first out*, tendremos que si introducimos un número más (p.e. `5`) y realizamos las operaciones de suma (`plus`) y substracción (`minus`) tendremos lo siguiente

```
-       -         -              -
-       23        -              -
23      10        23             -
10      33        10             23
33   ->  5   ->   38 (=33+5) -> -28 (=10-38)

5      plus      minus
```

Para llevar a cabo esta tarea se carga el módulo `Stack` en *(1)*. Una vez cargado el módulo podemos acceder a las subrutinas `POP` y `PUSH` que nos permiten manejar el stack. En *(2)* comienza el bucle principal, con la etiqueta `inloop`, que termina cuando el usuario da como input `Q`, `q` o `quit`.

Para controlar este bucle se utiliza una estructura `SELECT CASE` que comienza en *(3)*. Esta estructura analiza cuatro casos posibles:

- *(4)*: salir del programa
- *(5)*: suma
- *(6)*: resta
- *(7)*: introduce número en el stack (`DEFAULT`)

En el último caso se transforma la variable de carácter leída en una variable entera para almacenarla en el stack.

Para compilar y correr este programa podemos hacerlo compilando previamente el módulo, si lo hemos salvado en el fichero `ejemplo_10_1_Stack.f90`

```
$ gfortran -c ejemplo_10_1_Stack.f90
$ gfortran -o ejemplo_10_2 ejemplo_10_2.f90 ejemplo_10_1_Stack.o
```

10 El uso de módulos también permite, de forma flexible, segura y fácil de modificar, controlar la precisión de los números reales (o enteros) en los cálculos que se lleven a cabo. Una posible forma de definir de forma portable la doble precisión es mediante un sencillo módulo, llamado `dble_prec`. Como vimos en el programa 'Programa ejemplo_2_6.f90' en la página 7 los números reales de doble precisión tienen un `KIND = 8`. Para hacer el código independiente de la plataforma donde compilemos podemos hacer

```
MODULE dble_prec
  IMPLICIT NONE
  INTEGER, PARAMETER :: dbl = KIND(1.0D0)
END MODULE dble_prec
```

Por tanto podemos definir esa precisión cargando este módulo, p.e.

```
PROGRAM TEST_MINUIT
  !
  USE dble_prec
  !
  IMPLICIT NONE
  !
  ! Variable Definition
  REAL(KIND=dbl), PARAMETER :: PI = 4.0_dbl*ATAN(1.0_dbl)
  REAL(KIND=dbl) :: ENERF
   ....
   ....
```

Esto favorece la portabilidad y reduce el riesgo de errores ya que para cambiar la precisión con la que se trabaja solamente es necesario editar el módulo. En el 'Programa ejemplo_10_3.f90' en la página 60 introducimos esta mejora en el programa 'Programa ejemplo_9_6.f90' en la página 53. Se almacena el módulo simple anteriormente descrito en un fichero llamado, p.e., `dble_prec.f90` y se compila previamente:

```
$ gfortran -c dble_prec.f90
$ gfortran -o ejemplo_10_3 ejemplo_10_3.f90 dble_prec.o
```

Un módulo más completo, donde se definen diferentes tipos de enteros y de reales es el dado en el programa 'Programa ejemplo_10_4.f90' en la página 61.

En un ejercicio se plantean al alumnos diferentes maneras de mejorar el programa simple 'Programa ejemplo_10_2.f90' en la página siguiente.

10.3. Programas usados como ejemplo.

10.3.1. Programa ejemplo_10_1.f90

```
MODULE Stack
  !
  ! MODULE THAT DEFINES A BASIC STACK
  !
  IMPLICIT NONE
  !
  SAVE
  !
  INTEGER, PARAMETER :: STACK_SIZE = 500
  INTEGER :: STORE(STACK_SIZE) = 0, STACK_POS = 0
  !
  PRIVATE :: STORE, STACK_POS, STACK_SIZE
  PUBLIC :: POP, PUSH
  !
  CONTAINS
    !
    SUBROUTINE PUSH(I)
      !
      INTEGER, INTENT(IN) :: I
      !
      IF (STACK_POS < STACK_SIZE) THEN
         !
         STACK_POS = STACK_POS + 1; STORE(STACK_POS) = I
         !
      ELSE
         !
         STOP "FULL STACK ERROR"
         !
```

```
      ENDIF
      !
    END SUBROUTINE PUSH
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    SUBROUTINE POP(I)
      !
      INTEGER, INTENT(OUT) :: I
      !
      IF (STACK_POS > 0) THEN
         !
         I = STORE(STACK_POS); STACK_POS = STACK_POS - 1
         !
      ELSE
         !
         STOP "EMPTY STACK ERROR"
         !
      ENDIF
      !
    END SUBROUTINE POP
    !
END MODULE Stack
```

10.3.2. Programa ejemplo_10_2.f90

```
PROGRAM RPN_CALC
  !
  ! SIMPLE INTEGER RPN CALCULATOR (ONLY SUM AND SUBSTRACT)
  !
  USE Stack                 !!        (1)
  !
  IMPLICIT NONE
  !
  INTEGER :: KEYB_DATA
  CHARACTER(LEN=10) :: INPDAT
  !
  INTEGER :: I, J, K, DATL, NUM, RES
  !
  !
  inloop: DO      !! MAIN LOOP        (2)
     !
     READ 100, INPDAT
     !
     SELECT CASE (INPDAT)   !!        (3)
        !
     CASE ('Q','q')  !! EXIT          (4)
        PRINT*, "End of program"
        EXIT inloop
     CASE ('plus','Plus','PLUS','+')   !! SUM              (5)
        CALL POP(J)
        CALL POP(K)
        RES = K + J
        PRINT 120, K, J, RES
        CALL PUSH(RES)
     CASE ('minus','Minus','MINUS','-')   !! SUBSTRACT        (6)
        CALL POP(J)
        CALL POP(K)
        RES = K - J
        PRINT 130, K, J, RES
        CALL PUSH(RES)
     CASE DEFAULT !! NUMBER TO STACK  (7)
        !
        DATL = LEN_TRIM(INPDAT)
        !
        RES = 0
        DO I = DATL, 1, -1
           NUM = IACHAR(INPDAT(I:I)) - 48
           RES = RES + NUM*10**(DATL-I)
        ENDDO
        !
        PRINT 110, RES
        CALL PUSH(RES)
     END SELECT
     !
  ENDDO inloop
  !
100 FORMAT(A10)
110 FORMAT(1X, I10)
120 FORMAT(1X, I10,' + ', I10,' = ', I20)
130 FORMAT(1X, I10,' - ', I10,' = ', I20)
END PROGRAM RPN_CALC
```

10.3.3. Programa ejemplo_10_3.f90

```
PROGRAM ejemplo_10_3
  !
  USE dble_prec
  !
  IMPLICIT NONE
  !
  INTEGER :: I, IERR
  REAL(KIND=dbl), DIMENSION(:), ALLOCATABLE :: X, Y
  REAL(KIND=dbl) :: M, SD, MEDIAN
  ! interface block
  INTERFACE
     SUBROUTINE STATS(VECTOR,N,MEAN,STD_DEV,MEDIAN)
       !
       USE dble_prec
       !
       IMPLICIT NONE
       INTEGER , INTENT(IN)                     ::  N
       REAL(KIND=dbl)      , INTENT(IN) , DIMENSION(:)   :: VECTOR
       REAL(KIND=dbl)      , INTENT(OUT)                 :: MEAN
       REAL(KIND=dbl)      , INTENT(OUT)                 :: STD_DEV
       REAL(KIND=dbl)      , INTENT(OUT)                 :: MEDIAN
     END SUBROUTINE STATS
  END INTERFACE
  !
  READ*, I
  !
  ALLOCATE(X(1:I), STAT = IERR)
  IF (IERR /= 0) THEN
     PRINT*, "X allocation request denied."
     STOP
  ENDIF
  !
  ALLOCATE(Y(1:I), STAT = IERR)
  IF (IERR /= 0) THEN
     PRINT*, "Y allocation request denied."
     STOP
  ENDIF
  !
  CALL BOX_MULLER(I)
  !
  PRINT*, X
  CALL STATS(X,I,M,SD,MEDIAN)
  !
  PRINT *,' MEAN = ',M
  PRINT *,' STANDARD DEVIATION = ',SD
  PRINT *,' MEDIAN IS = ',MEDIAN
  !
  IF (ALLOCATED(X)) DEALLOCATE(X, STAT = IERR)
  IF (IERR /= 0) THEN
     PRINT*, "X NON DEALLOCATED!"
     STOP
  ENDIF
  PRINT*, Y
  CALL STATS(Y,I,M,SD,MEDIAN)
  !
  PRINT *,' MEAN = ',M
  PRINT *,' STANDARD DEVIATION = ',SD
  PRINT *,' MEDIAN IS = ',MEDIAN
  !
  IF (ALLOCATED(Y)) DEALLOCATE(Y, STAT = IERR)
  IF (IERR /= 0) THEN
     PRINT*, "Y NON DEALLOCATED!"
     STOP
  ENDIF
  !
CONTAINS
  !
  SUBROUTINE BOX_MULLER(dim)
    !
    ! Uses the Box-Muller method to create two normally distributed vectors
    !
    INTEGER, INTENT(IN) :: dim
    !
    REAL(KIND=dbl), PARAMETER :: PI = ACOS(-1.0_dbl)
    REAL(KIND=dbl), DIMENSION(dim) :: RANDOM_u, RANDOM_v ! Automatic arrays
    !
    CALL RANDOM_NUMBER(RANDOM_u)
    CALL RANDOM_NUMBER(RANDOM_v)
    !
    X = SQRT(-2.0_dbl*LOG(RANDOM_u))
    Y = X*SIN(2.0_dbl*PI*RANDOM_v)
    X = X*COS(2.0_dbl*PI*RANDOM_v)
    !
  END SUBROUTINE BOX_MULLER
```

```
  !
END PROGRAM ejemplo_10_3
SUBROUTINE STATS(VECTOR,N,MEAN,STD_DEV,MEDIAN)
  USE dble_prec
  IMPLICIT NONE
  ! Definicion de variables
  INTEGER , INTENT(IN)                    ::  N
  REAL(KIND=dbl)      , INTENT(IN) , DIMENSION(:)    ::  VECTOR    !! (1)
  REAL(KIND=dbl)      , INTENT(OUT)                  ::  MEAN
  REAL(KIND=dbl)      , INTENT(OUT)                  ::  STD_DEV
  REAL(KIND=dbl)      , INTENT(OUT)                  ::  MEDIAN
  REAL(KIND=dbl)      , DIMENSION(1:N)              ::  Y
  REAL(KIND=dbl)      :: VARIANCE = 0.0_dbl
  REAL(KIND=dbl)      :: SUMXI = 0.0_dbl, SUMXI2 = 0.0_dbl
  !
  SUMXI=SUM(VECTOR)       !! (6)
  SUMXI2=SUM(VECTOR*VECTOR)    !! (6)
  MEAN=SUMXI/N
  VARIANCE=(SUMXI2-SUMXI*SUMXI/N)/(N-1)
  STD_DEV = SQRT(VARIANCE)
  Y=VECTOR
  ! Ordena valores por proceso de seleccion
  CALL SELECTION
  IF (MOD(N,2) == 0) THEN
     MEDIAN=(Y(N/2)+Y((N/2)+1))/2
  ELSE
     MEDIAN=Y((N/2)+1)
  ENDIF
CONTAINS     !! (7)
  SUBROUTINE SELECTION
    IMPLICIT NONE
    INTEGER :: I,J,K
    REAL :: MINIMUM
    DO I=1,N-1
       K=I
       MINIMUM=Y(I)
       DO J=I+1,N
          IF (Y(J) < MINIMUM) THEN
             K=J
             MINIMUM=Y(K)
          END IF
       END DO
       Y(K)=Y(I)
       Y(I)=MINIMUM
    END DO
  END SUBROUTINE SELECTION
END SUBROUTINE STATS
```

10.3.4. Programa ejemplo_10_4.f90

```
MODULE NUMERIC_KINDS
  ! 4, 2, AND 1 BYTE INTEGERS
  INTEGER, PARAMETER :: &
       i4b = SELECTED_INT_KIND(9), &
       i2b = SELECTED_INT_KIND(4), &
       i1b = SELECTED_INT_KIND(2)
  ! SINGLE, DOUBLE, AND QUADRUPLE PRECISION
  INTEGER, PARAMETER :: &
       sp = KIND(1.0), &
       dp = SELECTED_REAL_KIND(2*PRECISION(1.0_sp)), &
       qp = SELECTED_REAL_KIND(2*PRECISION(1.0_dp))
END MODULE NUMERIC_KINDS
```

Capítulo 11

Subprogramas (IV)

11.1. Objetivos

Los objetivos de esta clase son los siguientes:

1 Explicar como se deben gestionar los errores en la invocación de funciones y subrutinas.

2 Explicar como se pasa el nombre de una función o subrutina como argumento declarando las funciones o subrutinas implicadas con el atributo `EXTERNAL`.

3 Explicar como se pasa el nombre de una función o subrutina como argumento declarando las funciones o subrutinas en un módulo.

11.2. Puntos destacables.

1 Se debe evitar que un programa termine sin que una subprograma (función o subrutina) devuelva el control al programa que lo ha invocado. Por ello se debe no usar la orden `STOP` en el interior de subprogramas. La mejor forma de gestionar errores en una subrutina, sobre todo aquellos debidos a una incorrecta definición de los argumentos de entrada de la subrutina, es mediante el uso de varibles *flag* (bandera) que marquen que ha tenido lugar un error. En el siguiente ejemplo se calcula la raíz cuadrada de la diferencia entre dos números, y la variable `sta_flag` es cero si la subrutina se ejecuta sin problemas o uno si se trata de calcular la raíz cuadrada de un número negativo.

```
SUBROUTINE calc(a_1, a_2, result, sta_flag)
   IMPLICIT NONE
   REAL, INTENT(IN) :: a_1, a_2
   REAL, INTENT(OUT) :: result
   INTEGER, INTENT(OUT) :: sta_flag
   !
   REAL :: temp
   !
   temp = a_1 - a_2
   IF (temp >= 0) THEN
      result = SQRT(temp)
      sta_flag = 0
   ELSE
      result = 0.0
      sta_flag = 1
   ENDIF
END SUBROUTINE calc
```

Una vez ejecutada la subrutina se debe comprobar el valor de la variable `sta_flag` para informar si ha existido algún problema.

2 Al invocar una subrutina los argumentos pasan como una serie de punteros a ciertas posiciones de memoria. Eso permite que como argumento figure una función o subrutina.

3 En el caso de funciones, cuando se incluye el nombre de una función en la lista de argumentos se transforma en un puntero a dicha función. Para ello las funciones han de ser declaradas con el atributo `EXTERNAL`. Si, por ejemplo, desde un programa llamamos a una subrutina llamada `evaluate_func` para evaluar las funciones `fun_1` y `fun_2` podemos hacer algo como

```
PROGRAM test
  IMPLICIT NONE
  REAL :: fun_1, fun_2
  EXTERNAL fun_1, fun_2
  REAL :: x, y, output

  ......

  CALL evaluate_func(fun_1, x, y, output)
  CALL evaluate_func(fun_2, x, y, output)

  ......

END PROGRAM test

SUBROUTINE evaluate_func(fun, a, b, out)
   REAL, EXTERNAL :: fun
   REAL, INTENT(IN) :: a, b
   REAL, INTENT(OUT) :: out
   !
   out = fun(a,b)
END SUBROUTINE evaluate_func
```

En el 'Programa ejemplo_11_1.f90' en la página siguiente se muestra un ejemplo en el que se evalúa, dependiendo de la elección del usuario, el producto o el cociente entre dos números. Dependiendo de la elección se utiliza la subrutina `Eval_Func`, que acepta como uno de sus argumentos el nombre de la función que se va a evaluar, `prod_func` o `quot_func`. Debe indicarse el tipo de variable asociado a la función, pero no se puede especificar el atributo `INTENT`.

4 También pueden usarse nombres de subrutinas como argumentos. Para pasar el nombre de una subrutina como argumento dicha subrutina debe ser declarada con el atributo EXTERNAL. En el siguiente ejemplo una subrutina llamada `launch_sub` acepta como argumentos de entrada las variables `x_1` y `x_2` y el nombre de una subrutina a la que invoca con las variables anteriores como argumentos y tiene como argumento de salida la variable `result`.

```
SUBROUTINE launch_sub(x_1, x_2, sub_name, result)
  IMPLICIT NONE
  REAL, INTENT(IN) :: x_1, x_2
  EXTERNAL sub_name
  REAL, INTENT(OUT) :: result


  ......

  CALL sub_name(x_1, x_2, result)

  ......

END SUBROUTINE launch_sub
```

Como puede verse en este ejemplo, el argumento que indica la subrutina (`sub_name`) no lleva asociado el atributo `INTENT`. En el 'Programa ejemplo_11_2.f90' en la página 66 se muestra un ejemplo similar al anterior, en el que se evalúa dependiendo de la elección del usuario el producto o el cociente entre dos números. Dependiendo de la elección se utiliza la subrutina `Eval_Sub`, que acepta como uno de sus argumentos el nombre de la subrutina que se va a evaluar, `prod_sub` o `quot_sub`.

5 En el 'Programa ejemplo_11_3.f90' en la página 67 se muestra un ejemplo algo más complejo en el que se evalúa, dependiendo de la elección del usuario, una función entre tres posibles para un intervalo de la variable independiente. En este caso las funciones se declaran como `EXTERNAL` y se utiliza una subrutina interna para la definición del vector de la variable independiente, de acuerdo con la dimensión que proporciona el usuario, y la subrutina `Eval_Func` que acepta como uno de sus argumentos el nombre de la función que se evalué mostrando los resultados en pantalla.

6 Es posible también comunicar a un subprograma el nombre de una función o una subrutina mediante el uso de módulos. En el 'Programa ejemplo_11_4.f90' en la página 68 se muestra un programa similar al 'Programa ejemplo_11_3.f90' en la página 67 utilizando módulos. El módulo `Functions_11_4` debe compilarse en un fichero separado al del programa principal. Si, por ejemplo el módulo se llama `ejemplo_11_4_mod.f90` y el programa principal `ejemplo_11_4.f90` el procedimiento sería el siguiente

```
$ gfortran -c ejemplo_11_4_mod.f90
$ gfortran ejemplo_11_4.f90 ejemplo_11_4_mod.o
```

Como ocurría en el caso anterior, el o los argumentos que indican funciones o subrutinas no llevan el atributo `INTENT`.

11.3. Programas usados como ejemplo.

11.3.1. Programa ejemplo_11_1.f90

```
PROGRAM func_option
  !
  ! Select between funs to compute the product of the quotient of two quantities
  !
  IMPLICIT NONE
  !
  !
  REAL :: X_1, X_2
  INTEGER :: I_fun
  INTEGER :: I_exit
  !
  REAL, EXTERNAL :: prod_fun, quot_fun
  !
  I_exit = 1
  !
  DO WHILE (I_exit /= 0)
     !
     PRINT*, "X_1, X_2?"
     READ(UNIT = *, FMT = *) X_1, X_2
     !
     PRINT*, "function 1 = X_1 * X_2, 2 = X_1/X_2 ? (0 = exit)"
     READ(UNIT = *, FMT = *) I_fun
     !
     SELECT CASE (I_fun)
        !
     CASE (0)
        I_exit = 1
     CASE (1)
        CALL Eval_func(prod_fun, X_1, X_2)
     CASE (2)
        CALL Eval_func(quot_fun, X_1, X_2)
     CASE DEFAULT
        PRINT*, "Valid options : 0, 1, 2"
        !
     END SELECT
     !
     PRINT*, "Continue? (0 = exit)"
     READ(UNIT=*, FMT = *) I_exit
     !
     !
  ENDDO
  !
END PROGRAM func_option
!
SUBROUTINE Eval_Func(fun, X_1, X_2)
  !
  IMPLICIT NONE
  !
  REAL, INTENT(IN) :: X_1, X_2
  REAL, EXTERNAL :: fun
  !
  PRINT 10, fun(X_1, X_2)
  !
  10 FORMAT(1X, ES16.8)
  !
END SUBROUTINE Eval_Func
!
!
FUNCTION prod_fun(x1, x2)
  !
  IMPLICIT  NONE
  !
  REAL, INTENT(IN) :: x1, x2
  !
  REAL prod_fun
  !
  prod_fun = x1*x2
  !
END FUNCTION prod_fun
!
FUNCTION quot_fun(x1, x2)
```

```
  !
  IMPLICIT  NONE
  !
  REAL, INTENT(IN) :: x1, x2
  !
  REAL quot_fun
  !
  quot_fun = x1/x2
  !
END FUNCTION quot_fun
```

11.3.2. Programa ejemplo_11_2.f90

```
PROGRAM sub_option
  !
  ! Select between subs to compute the product or the quotient of two quantities
  !
  IMPLICIT NONE
  !
  !
  REAL :: X_1, X_2
  INTEGER :: I_sub
  INTEGER :: I_exit
  !
  EXTERNAL :: prod_sub, quot_sub
  !
  I_exit = 1
  !
  DO WHILE (I_exit /= 0)
     !
     PRINT*, "X_1, X_2?"
     READ(UNIT = *, FMT = *) X_1, X_2
     !
     PRINT*, "function 1 = X_1 * X_2, 2 = X_1/X_2 ? (0 = exit)"
     READ(UNIT = *, FMT = *) I_sub
     !
     SELECT CASE (I_sub)
        !
     CASE (0)
        I_exit = 0
     CASE (1)
        CALL Eval_Sub(prod_sub, X_1, X_2)
     CASE (2)
        CALL Eval_Sub(quot_sub, X_1, X_2)
     CASE DEFAULT
        PRINT*, "Valid options : 0, 1, 2"
        !
     END SELECT
     !
     PRINT*, "Continue? (0 = exit)"
     READ(UNIT=*, FMT = *) I_exit
     !
  ENDDO
  !
END PROGRAM sub_option
!
SUBROUTINE Eval_Sub(sub, X_1, X_2)
  !
  IMPLICIT NONE
  !
  EXTERNAL :: sub
  REAL, INTENT(IN) :: X_1, X_2
  !
  REAL :: res_sub
  !
  CALL sub(X_1, X_2, res_sub)
  PRINT 10, res_sub
  !
10 FORMAT(1X, ES16.8)
  !
END SUBROUTINE Eval_Sub
!
!
SUBROUTINE prod_sub(x1, x2, y)
  !
  IMPLICIT  NONE
  !
  REAL, INTENT(IN) :: x1, x2
  REAL, INTENT(OUT) :: y
  !
  y = x1*x2
  !
END SUBROUTINE prod_sub
!
```

```
!
SUBROUTINE quot_sub(x1, x2, y)
  !
  IMPLICIT  NONE
  !
  REAL, INTENT(IN) :: x1, x2
  REAL, INTENT(OUT) :: y
  !
  y = x1/x2
  !
END SUBROUTINE quot_sub
```

11.3.3. Programa ejemplo_11_3.f90

```
PROGRAM call_func
  !
  ! Select which curve is computed and saved in a given interval e.g. (-2 Pi, 2 Pi)
  !
  ! 1 ---> 10 x^2 cos(2x) exp(-x)
  ! 2 ---> 10 (-x^2 + x^4)exp(-x^2)
  ! 3 ---> 10 (-x^2 + cos(x)*x^4)exp(-x^2)
  !
  IMPLICIT NONE
  !
  !
  REAL, DIMENSION(:), ALLOCATABLE :: X_grid
  !
  REAL, PARAMETER :: pi = ACOS(-1.0)
  !
  REAL :: X_min, X_max, Delta_X
  INTEGER :: X_dim, I_fun
  INTEGER :: I_exit, Ierr
  !
  REAL, EXTERNAL :: fun1, fun2, fun3
  !
  X_min = -2*pi
  X_max = 2*pi
  !
  I_exit = 0
  !
  DO WHILE (I_exit /= 1)
     !
     PRINT*, "number of points? (0 = exit)"
     READ(UNIT=*, FMT = *) X_dim
     !
     IF (X_dim == 0) THEN
        !
        I_exit = 1
        !
     ELSE
        ALLOCATE(X_grid(1:X_dim), STAT = Ierr)
        IF (Ierr /= 0) THEN
           STOP 'X_grid allocation failed'
        ENDIF
        !
        CALL make_Grid(X_min, X_max, X_dim)
        !
        PRINT*, "function 1, 2, or 3? (0 = exit)"
        READ(UNIT = *, FMT = *) I_fun
        !
        SELECT CASE (I_fun)
           !
        CASE (0)
           I_exit = 1
        CASE (1)
           CALL Eval_func(fun1, X_dim, X_grid)
        CASE (2)
           CALL Eval_func(fun2, X_dim, X_grid)
        CASE (3)
           CALL Eval_func(fun3, X_dim, X_grid)
        CASE DEFAULT
           PRINT*, "Valid options : 0, 1, 2, 3"
           !
        END SELECT
        !
        DEALLOCATE(X_grid, STAT = Ierr)
        IF (Ierr /= 0) THEN
           STOP 'X_grid deallocation failed'
        ENDIF
        !
     ENDIF
     !
  ENDDO
  !
```

```
CONTAINS
  !
  SUBROUTINE make_Grid(X_min, X_max, X_dim)
    !
    REAL, INTENT(IN) :: X_min, X_max
    INTEGER, INTENT(IN) :: X_dim
    !
    INTEGER :: Index
    REAL :: Delta_X
    !
    !
    Delta_X = (X_max - X_min)/REAL(X_dim - 1)
    !
    X_grid = (/ (Index, Index = 0 , X_dim - 1 ) /)
    X_grid = X_min + Delta_X*X_grid
    !
  END SUBROUTINE make_Grid
  !
END PROGRAM call_func
!
SUBROUTINE Eval_Func(fun, dim, X_grid)
  !
  IMPLICIT NONE
  !
  INTEGER, INTENT(IN) :: dim
  REAL, DIMENSION(dim), INTENT(IN) :: X_grid
  REAL, EXTERNAL :: fun
  !
  INTEGER :: Index
  !
  DO Index = 1, dim
     PRINT 10, X_grid(Index), fun(X_grid(Index))
  ENDDO
  !
  10 FORMAT(1X, ES16.8,2X, ES16.8)
  !
END SUBROUTINE Eval_Func
!
!
FUNCTION fun1(x)
  !
  IMPLICIT  NONE
  !
  REAL, INTENT(IN) :: x
  !
  REAL fun1
  !
  fun1 = 10.0*x**2*cos(2.0*x)*exp(-x)
  !
END FUNCTION fun1
!
FUNCTION fun2(x)
  !
  IMPLICIT  NONE
  !
  REAL, INTENT(IN) :: x
  !
  REAL fun2
  !
  fun2 = 10.0*(-x**2 + x**4)*exp(-x**2)
  !
END FUNCTION fun2
!
FUNCTION fun3(x)
  !
  IMPLICIT  NONE
  !
  REAL, INTENT(IN) :: x
  !
  REAL fun3
  !
  fun3 = 10.0*(-x**2 + cos(x)*x**4)*exp(-x**2)
  !
END FUNCTION fun3
```

11.3.4. Programa ejemplo_11_4.f90

```
PROGRAM call_func
  !
  ! Select which curve is computed and saved in a given interval e.g. (-2 Pi, 2 Pi)
  !
  ! 1 ---> 10 x^2 cos(2x) exp(-x)
  ! 2 ---> 10 (-x^2 + x^4)exp(-x^2)
  ! 3 ---> 10 (-x^2 + cos(x)*x^4)exp(-x^2)
```

```
  !
  USE Functions_11_4
  !
  IMPLICIT NONE
  !
  !
  REAL, DIMENSION(:), ALLOCATABLE :: X_grid
  !
  REAL, PARAMETER :: pi = ACOS(-1.0)
  !
  REAL :: X_min, X_max, Delta_X
  INTEGER :: X_dim, I_fun
  INTEGER :: I_exit, Ierr
  !
  X_min = -2*pi
  X_max = 2*pi
  !
  I_exit = 0
  !
  DO WHILE (I_exit /= 1)
     !
     PRINT*, "number of points? (0 = exit)"
     READ(UNIT=*, FMT = *) X_dim
     !
     IF (X_dim == 0) THEN
        !
        I_exit = 1
        !
     ELSE
        ALLOCATE(X_grid(1:X_dim), STAT = Ierr)
        IF (Ierr /= 0) THEN
           STOP 'X_grid allocation failed'
        ENDIF
        !
        CALL make_Grid(X_min, X_max, X_dim)
        !
        PRINT*, "function 1, 2, or 3? (0 = exit)"
        READ(UNIT = *, FMT = *) I_fun
        !
        SELECT CASE (I_fun)
           !
        CASE (0)
           I_exit = 1
        CASE (1)
           CALL Eval_func(fun1, X_dim, X_grid)
        CASE (2)
           CALL Eval_func(fun2, X_dim, X_grid)
        CASE (3)
           CALL Eval_func(fun3, X_dim, X_grid)
        CASE DEFAULT
           PRINT*, "Valid options : 0, 1, 2, 3"
           !
        END SELECT
        !
        DEALLOCATE(X_grid, STAT = Ierr)
        IF (Ierr /= 0) THEN
           STOP 'X_grid deallocation failed'
        ENDIF
        !
     ENDIF
     !
  ENDDO
  !
CONTAINS
  !
  SUBROUTINE make_Grid(X_min, X_max, X_dim)
    !
    REAL, INTENT(IN) :: X_min, X_max
    INTEGER, INTENT(IN) :: X_dim
    !
    INTEGER :: Index
    REAL :: Delta_X
    !
    !
    Delta_X = (X_max - X_min)/REAL(X_dim - 1)
    !
    X_grid = (/ (Index, Index = 0 , X_dim - 1 ) /)
    X_grid = X_min + Delta_X*X_grid
    !
  END SUBROUTINE make_Grid
  !
END PROGRAM call_func
!
SUBROUTINE Eval_Func(fun, dim, X_grid)
  !
  USE Functions_11_4
```

```
    !
    IMPLICIT NONE
    !
    REAL :: fun
    INTEGER, INTENT(IN) :: dim
    REAL, DIMENSION(dim), INTENT(IN) :: X_grid
    !
    INTEGER :: Index
    !
    DO Index = 1, dim
       PRINT 10, X_grid(Index), fun(X_grid(Index))
    ENDDO
    !
    10 FORMAT(1X, ES16.8,2X, ES16.8)
    !
  END SUBROUTINE Eval_Func
  !
  MODULE Functions_11_4
    IMPLICIT NONE
    !
  CONTAINS
    !
    !
    FUNCTION fun1(x)
      !
      IMPLICIT  NONE
      !
      REAL, INTENT(IN) :: x
      !
      REAL fun1
      !
      fun1 = 10.0*x**2*cos(2.0*x)*exp(-x)
      !
    END FUNCTION fun1
    !
    FUNCTION fun2(x)
      !
      IMPLICIT  NONE
      !
      REAL, INTENT(IN) :: x
      !
      REAL fun2
      !
      fun2 = 10.0*(-x**2 + x**4)*exp(-x**2)
      !
    END FUNCTION fun2
    !
    FUNCTION fun3(x)
      !
      IMPLICIT  NONE
      !
      REAL, INTENT(IN) :: x
      !
      REAL fun3
      !
      fun3 = 10.0*(-x**2 + cos(x)*x**4)*exp(-x**2)
      !
    END FUNCTION fun3
  END MODULE Functions_11_4
```

Capítulo 12

Instalación y uso de las bibliotecas BLAS y LAPACK

12.1. Objetivos

Los objetivos de esta clase son los siguientes:

1 familiarizar el alumno con la compilación de programas y la instalación de librerías o bibliotecas usando el compilador `gfortran`.

2 Instalar las bibliotecas de interés científico `BLAS` y `LAPACK`.

3 Aprender a hacer uso de dichas bibliotecas.

Existe una gran cantidad de código `Fortran` accesible de forma abierta, ya sea como código fuente o en forma de biblioteca. En la presente clase el alumno se familiariza con la obtención, compilación, instalación y uso de dos bibliotecas de subrutinas de interés algebraico, `BLAS` y `LAPACK`.

12.2. Puntos destacables.

Indicaremos de forma escalonada los diferentes pasos que hay que seguir para la instalación de estas bibliotecas.

El código fuente de las bibliotecas `BLAS` y `LAPACK` puede descargarse de diferentes lugares, o instalarse a partir de paquetes de la distribución `Debian` o `Ubuntu` que se esté utilizando. En vez de ello las instalaremos compilándolas en nuestro ordenador.

- Descarga de código fuente de la biblioteca `BLAS`.

  Se puede descargar de la web de `NETLIB`[1], usando este enlace BLAS tgz (Netlib) (`http://www.netlib.org/blas/blas.tgz`).

- Una vez descargado el código fuente, se descomprime, se compila y se crea finalmente la librería.

```
tar xzf blas.tgz
cd BLAS
gfortran -O2 -c *.f
ar cr libblas.a *.o
```

  Con lo que se debe haber creado la librería estática `libblas.a`.

- A continuación se sitúa dicha librería en un lugar apropiado, por ejemplo con

[1] El repositorio de `Netlib` contiene software gratuito, documentación y bases de datos de interés para la comunidad científica en general y para aquellos interesados en la computación científica en particular. El repositorio es mantenido por los Laboratorios AT&T-Bell, la Universidad de Tennessee y el laboratorio nacional de Oak Ridge, con la ayuda de un gran número de colaboradores en todo el mundo. La web de `Netlib` es `http://www.netlib.org`.

```
sudo cp libblas.a /usr/local/lib
```

y se comprueba que tiene los permisos adecuados.

- Descarga del código fuente de la biblioteca LAPACK.

  Se puede descargar también de la web de NETLIB usando este enlace LAPACK tgz (Netlib) (http://www.netlib.org/lapack/lapack.tgz). Tras su descarga se desempaquetan los ficheros.

```
tar xzf lapack.tgz
cd lapack-3.2.1
```

- Esta biblioteca si tiene una serie de ficheros makefile para su compilación. Hemos de preparar un fichero make.inc adecuado, como el que hemos incluido en 'Ejemplo de fichero make.inc para LAPACK' en la página siguiente y que está disponible en Moodle en un fichero llamado make.inc.lapack.ubuntu (http://moodle.uhu.es/contenidos/file.php/245/src_fortran_clase/make.inc.lapack.ubuntu).

  Usando este fichero compilamos la librería haciendo

```
make
```

- Por último, instalamos la librería copiando los ficheros creados al lugar que creamos más adecuado para su ubicación.

```
sudo cp lapack_LINUX.a /usr/local/lib
sudo cp tmglib_LINUX.a /usr/local/lib
```

- Para terminar descargamos el código fuente de la biblioteca LAPACK95.

  Se puede descargar también de la web de NETLIB usando el enlace LAPACK95 tgz (Netlib) (http://www.netlib.org/lapack95/lapack95.tgz). Tras su descarga se desempaquetan los ficheros.

```
tar xzf lapack95.tgz
cd LAPACK95
```

- Esta biblioteca también tiene una serie de ficheros makefile para su compilación. Hemos de preparar de nuevo un fichero make.inc adecuado, como el que hemos incluido en 'Ejemplo de fichero make.inc para LAPACK95' en la página 74 y que está disponible en Moodle en un fichero llamado make.inc.lapack95.ubuntu (http://moodle.uhu.es/contenidos/file.php/245/src_fortran_clase/make.inc.lapack95.ubuntu).

  Usando este fichero compilamos la librería haciendo

```
cd SRC
make single_double_complex_dcomplex
```

  La opción escogida es la más general, pues general la librería para precisión simple, doble, compleja simple y compleja doble.

- Por último, instalamos la librería copiando los ficheros creados al lugar que creamos más adecuado para su ubicación.

```
sudo cp lapack95.a /usr/local/lib
sudo cp -r lapack95_modules /usr/local/lib
```

- En los directorios de ejemplos (LAPACK95/EXAMPLES1 y LAPACK95/EXAMPLES2) encontramos un gran número de ejemplos que podemos correr y comprobar las salidas obtenidas con las que se encuentran en Lapack95 User's guide (http://www.netlib.org/lapack95/lug95/node1.html). Las instrucciones para compilar y correr los ejemplos proporcionados pueden verse en el fichero README del directorio donde se encuentra el código fuente de los ejemplos.

- En el ejemplo 'Ejemplo de programa que invoca LAPACK95' en la página 74 se encuentra el código de un programa donde se recurre a la subrutina la_spsv para hallar la solución de un sistema lineal de ecuaciones, $Ax = B$, donde la matriz del sistema, A, es simétrica y se almacena de forma compacta y x, B son vectores. Es importante que comprenda como funciona este programa, así como que se sepa extraer de la documentación de LAPACK95 el significado de los argumentos de entrada y salida de la subrutina.

- Para correr este programa es necesario descargar el código ejemplo_la_spsv.f90 y los ficheros de datos spsv.ma y spsv.mb de la web del curso. Para compilar el programa se ejecuta la orden

```
gfortran -o ejemplo_la_spsv -I/usr/local/lib/lapack95_modules ejemplo_la_spsv.f90  /usr/local/lib/lapack95.a /usr/local/lib/tmgli
```

En esta orden de compilación se incluyen todas las librerías y módulos necesarios para que pueda crearse el ejecutable, haciendo uso de las librerías BLAS, LAPACK y LAPACK95 que hemos instalado.

- Para proyectos más complejos y evitar tener que escribir comandos de compilación tan complejos como el anterior es posible usar un fichero makefile como el que se proporciona en el ejemplo 'Ejemplo de makefile para compilar programas que invocan LAPACK95' en la página 75. Para usar este fichero en la compilación del ejemplo 'Ejemplo de programa que invoca LAPACK95' en la página siguiente es preciso copiar el fichero proporcionado o descargar el fichero makefile_lapack95 y ejecutar la orden

```
make -f makefile_lapack95 ejemplo_la_spsv
```

12.3. Programas usados como ejemplo.

12.3.1. Ejemplo de fichero make.inc para LAPACK

```
###################################################################
#  LAPACK make include file.                                      #
#  LAPACK, Version 3.2.1                                          #
#  MAY 2009                                                       #
#  Modified by Currix                                             #
###################################################################
#
SHELL = /bin/sh
#
#  The machine (platform) identifier to append to the library names
#
PLAT = _LINUX
#
#  Modify the FORTRAN and OPTS definitions to refer to the
#  compiler and desired compiler options for your machine.  NOOPT
#  refers to the compiler options desired when NO OPTIMIZATION is
#  selected.  Define LOADER and LOADOPTS to refer to the loader and
#  desired load options for your machine.
#
FORTRAN  = gfortran
OPTS     = -O2
DRVOPTS  = $(OPTS)
NOOPT    = -O0
LOADER   = gfortran
LOADOPTS =
#
# Timer for the SECOND and DSECND routines
#
# Default : SECOND and DSECND will use a call to the EXTERNAL FUNCTION ETIME
#TIMER    = EXT_ETIME
# For RS6K : SECOND and DSECND will use a call to the EXTERNAL FUNCTION ETIME_
# TIMER    = EXT_ETIME_
# For gfortran compiler: SECOND and DSECND will use a call to the INTERNAL FUNCTION ETIME
TIMER    = INT_ETIME
# If your Fortran compiler does not provide etime (like Nag Fortran Compiler, etc...)
# SECOND and DSECND will use a call to the INTERNAL FUNCTION CPU_TIME
# TIMER    = INT_CPU_TIME
# If neither of this works...you can use the NONE value... In that case, SECOND and DSECND will always return 0
# TIMER     = NONE
#
#  The archiver and the flag(s) to use when building archive (library)
#  If you system has no ranlib, set RANLIB = echo.
#
ARCH     = ar
ARCHFLAGS= cr
RANLIB   = ranlib
#
#  Location of the extended-precision BLAS (XBLAS) Fortran library
#  used for building and testing extended-precision routines.  The
#  relevant routines will be compiled and XBLAS will be linked only if
#  USEXBLAS is defined.
#
# USEXBLAS    = Yes
XBLASLIB     =
# XBLASLIB    = -lxblas
#
#  The location of the libraries to which you will link.  (The
#  machine-specific, optimized BLAS library should be used whenever
#  possible.)
```

```
#
#BLASLIB      = ../../blas$(PLAT).a
BLASLIB      = /usr/local/lib/libblas.a
LAPACKLIB    = lapack$(PLAT).a
TMGLIB       = tmglib$(PLAT).a
EIGSRCLIB    = eigsrc$(PLAT).a
LINSRCLIB    = linsrc$(PLAT).a
```

12.3.2. Ejemplo de fichero `make.inc` para `LAPACK95`

```
#
#  -- LAPACK95 interface driver routine (version 2.0) --
#     UNI-C, Denmark; Univ. of Tennessee, USA; NAG Ltd., UK
#     August 5, 2000
#
FC  = gfortran
FC1       = gfortran
# -dcfuns  Enable recognition of non-standard double
#          precision  complex intrinsic functions
# -dusty   Allows the compilation and execution of "legacy"
#          software by downgrading the category  of  common
#          errors  found  in  such software from "Error" to
# -ieee=full enables all IEEE arithmetic facilities
#          including non-stop arithmetic.

OPTS0    = -u -V -dcfuns -dusty -ieee=full
MODLIB   = -I./../lapack95_modules
OPTS1    = -c $(OPTS0)
OPTS3    = $(OPTS1) $(MODLIB)
OPTL     = -o
OPTLIB   =

LAPACK_PATH = /usr/local/lib/

LAPACK95 = ../lapack95.a
LAPACK77 = $(LAPACK_PATH)/lapack_LINUX.a
TMG77    = $(LAPACK_PATH)/tmglib_LINUX.a
BLAS     = $(LAPACK_PATH)/libblas.a

LIBS     = $(LAPACK95) $(TMG77) $(LAPACK77) $(BLAS)
SUF      = f90

XX = 'rm' -f $@; \
        'rm' -f $@.res; \
 $(FC) $(OPTS0) -o $@ $(MODLIB) $@.$(SUF) $(OPTLIB) $(LIBS); \
        $@ < $@.dat > $@.res; \
        'rm' -f $@

YY = $(FC) $(OPTS0) -o $@ $(MODLIB) $@.$(SUF) $(OPTLIB) $(LIBS)

.SUFFIXES: .f90 .f .o

.$(SUF).o:
 $(FC) $(OPTS3) $<

.f.o:
 $(FC1) $(OPTS3) $<
```

12.3.3. Ejemplo de programa que invoca `LAPACK95`

```
PROGRAM LA_SSPSV_EXAMPLE

  !  -- LAPACK95 EXAMPLE DRIVER ROUTINE (VERSION 1.0) --
  !     UNI-C, DENMARK
  !     DECEMBER, 1999
  !
  !  .. "Use Statements"
  USE LA_PRECISION, ONLY: WP => SP
  USE F95_LAPACK, ONLY: LA_SPSV
  !  .. "Implicit Statement" ..
  IMPLICIT NONE
  !  .. "Local Scalars" ..
  INTEGER :: I, N, NN, NRHS
  !  .. "Local Arrays" ..
  INTEGER, ALLOCATABLE :: IPIV(:)
  REAL(WP), ALLOCATABLE :: B(:,:), AP(:)
  !  .. "Executable Statements" ..
  WRITE (*,*) 'SSPSV Example Program Results.'
  N = 5; NRHS = 1
  WRITE(*,'(5H N = , I4, 9H; NRHS = , I4)') N, NRHS
  NN = N*(N+1)/2
```

```
  ALLOCATE ( AP(NN), B(N,NRHS), IPIV(N) )
  !
  OPEN(UNIT=21,FILE='spsv.ma',STATUS='UNKNOWN')
  DO I=1,NN
     READ(21,'(F3.0)') AP(I)
  ENDDO
  CLOSE(21)
  !
  WRITE(*,*)'Matrix AP :'
  DO I=1,NN; WRITE(*,"(15(I3,1X,1X),I3,1X))") INT(AP(I));
  ENDDO
  !
  OPEN(UNIT=21,FILE='spsv.mb',STATUS='UNKNOWN')
  DO I=1,N
     READ(21,'(F3.0)') B(I,1)
  ENDDO
  CLOSE(21)
  !
  WRITE(*,*)'Matrix B :'
  DO I=1,N; WRITE(*,"(10(I3,1X,1X),I3,1X)')") INT(B(I,1));
  ENDDO
  !
  WRITE(*,*)" CALL LA_SPSV( AP, B, 'L', IPIV )"
  !
  CALL LA_SPSV( AP, B, 'L', IPIV )
  !
  WRITE(*,*)'AP on exit: '
  DO I=1,NN; WRITE(*,"(15(E13.5))") AP(I);
  ENDDO
  !
  WRITE(*,*)'Matrix B on exit :'
  DO I=1,N; WRITE(*,"(F9.5)") B(I,1);
  ENDDO
  WRITE(*,*)'IPIV = ', IPIV
  !
END PROGRAM LA_SSPSV_EXAMPLE
```

12.3.4. Ejemplo de `makefile` para compilar programas que invocan `LAPACK95`

```
#
#  -- LAPACK95 makefile (version 1.0) --
#
FC  = gfortran
#
MODLIB   = -I/usr/local/lib/lapack95_modules
OPTS1    = -c
OPTS3    = $(OPTS1) $(MODLIB)
OPTL     = -o
OPTLIB   =

LAPACK_PATH = /usr/local/lib
LAPACK95_PATH = /usr/local/lib

LAPACK95 = $(LAPACK95_PATH)/lapack95.a
LAPACK77 = $(LAPACK_PATH)/lapack_LINUX.a
TMG77    = $(LAPACK_PATH)/tmglib_LINUX.a
BLAS     = $(LAPACK_PATH)/libblas.a

LIBS     = $(LAPACK95) $(TMG77) $(LAPACK77) $(BLAS)
SUF      = f90

YY = $(FC) -o $@ $(MODLIB) $@.$(SUF) $(OPTLIB) $(LIBS)

.SUFFIXES: .f90 .f .o

.$(SUF).o:
 $(FC) $(OPTS3) $<

ejemplo_la_spsv:
 $(YY)

clean:
 'rm' -f *.o *.mod core
```

Capítulo 13

Referencias

1 Stephen J. Chapman; *Fortran 95/2003 for Scientists and Engineers*, 3a Ed. Mc Graw Hill 2008.

2 Michael Metcalf, John Reid, and Malcolm Cohen; *Modern Fortran Explained*, Oxford University Press 2011.

3 Jeanne C. Adams *et al.*; *Fortran 95 Handbook*, MIT Press 1997.

4 Ian D. Chivers and Jane Sleightholme; *Introduction to Programming with Fortran*, Springer 2006.

5 An Interactive Fortran 90 Programming Course (http://www.liv.ac.uk/HPC/HTMLFrontPageF90.html)

6 gfortran: The GNU Fortran compiler (http://gcc.gnu.org/onlinedocs/gfortran)

7 Gfortran - GCC Wiki (http://gcc.gnu.org/wiki/GFortran)

8 USER NOTES ON FORTRAN PROGRAMMING (UNFP) (http://sunsite.informatik.rwth-aachen.de/fortran/)

9 Fortran 90 Tutoria by Michael Metcalf (http://wwwasdoc.web.cern.ch/wwwasdoc/WWW/f90/f90.html)