FORTRAN 2003 : Programación Multicapa para la Simulación de Sistemas Físicos

Juan A. Hernández
Mario A. Zamecnik

Departamento de Matemática Aplicada y Estadística
Escuela Técnica Superior de Ingenieros Aeronáuticos
Universidad Politécnica de Madrid

Portada:
Simulación numérica de la rotura de frentes de concentración de partículas fluidizadas mediante una corriente ascendente de gas (Juan A. Hernández).

Martín de los Heros, 67. 28008 Madrid
I.S.B.N. : 84-931805-1-3
Depósito Legal: M-3036-2001

Índice general

Prólogo

Este libro trata de enseñar, con una metodología orientada a la simulación de sistemas físicos, la implementación de proyectos software. Está orientado a estudiantes, profesionales e investigadores interesados en desarrollar simuladores de sistemas físicos de envergadura media a elevada, en donde el proceso que realiza el sistema sea más complejo que el sistema en sí.

Hasta hace no muchos años parecía que la programación estructurada y el FORTRAN estaban reñidos. Sin embargo, el FORTRAN 95 ha dotado al lenguaje no sólo de la potencialidad suficiente para poder escribir en FORTRAN como con cualquier otro lenguaje, sino que mediante la extensión de su biblioteca de funciones intrínsecas vectoriales pone al FORTRAN en un lugar privilegiado para la simulación numérica de aquellos problemas en los que el objeto a simular sea una estructura de datos vectorial o matricial. No obstante, el FORTRAN es un lenguaje antiguo, y debido a la compatibilidad de las sucesivas versiones ha heredado características no deseables u obsoletas. Este libro no es, ni mucho menos, un manual de referencia de FORTRAN 95. Las características obsoletas, redundantes y no necesarias para la simulación –a criterio de los autores– han sido eliminadas. Así, los cuatro primeros capítulos de este libro se pueden considerar el subconjunto del FORTRAN 95 elegido por los autores. En los tres primeros capítulos se explica la sintaxis y la potencialidad del lenguaje con las operaciones vectoriales y matriciales. Están estructurados por orden de complejidad y contienen numerosos ejemplos para que el lector pueda concretar lo aprendido. El capítulo 4 está íntegramente dedicado a la especificación de variables y procedimientos. Aunque a primera vista parece que este capítulo debería estar entre los primeros, los autores han considerado que, debido a la dificultad conceptual que entraña, un encuentro sin la asimilación de los ejemplos sencillos de los tres primeros capítulos sería demasiado fuerte. Una vez definidas las interfaces de conexión entre unidades de programa se empieza a construir el castillo. En el capítulo 5 se explican las características deseables de una unidad de programa, su conexión con otras unidades y algunas habilidades para cambiar el formato de los argumentos de entrada o salida de un procedimiento.

El capítulo 6 es el corazón y el objetivo de este libro. En este capítulo se desarrolla la metodología multicapa para la simulación de sistemas físicos, la cual se basa en la construcción de un programa de simulación mediante abstracciones físicas o matemáticas estructuradas por nivel de complejidad funcional. Las

abstracciones físicas o matemáticas permiten al programador escribir códigos con un lenguaje tan próximo a las matemáticas o a la física como él quiera. De esta forma, el desarrollo multicapa en uno o dos años de trabajo y en la misma área de trabajo crea un nivel de abstracción que puede ser considerado como un nuevo lenguaje de más alto nivel y específico para el área en cuestión. El coste de esta metodología reside en el equipo de trabajo, obligado a tener conocimientos avanzados del lenguaje. A medio y largo plazo se obtienen importantes beneficios: códigos que se validan muy rápidamente, puesto que se apoyan en capas ya validadas, y códigos reutilizables con una larga vida útil.

Aunque la potencialidad de la programación multicapa se empieza a ver en proyectos grandes de dificultad elevada, en el capítulo 6 se incluye un ejemplo pequeño en el que se pueden apreciar las diferencias entre la programación plana, la encapsulada y la multicapa. Por último, para que el lector pueda ejercitarse en la programación multicapa, en el capítulo 7 se exponen tres problemas sencillos de simulación de sistemas físicos.

Ahora quisiéramos expresar nuestro agradecimiento a todas aquellas personas que con sus comentarios han contribuido al desarrollo de este libro. En especial, a D. Jose Miguel Lozano, D. Alberto Lerma, D. Miguel Hermanns y a todos los alumnos de "Informática" y "Taller de Cálculo Numérico" de la Escuela Técnica Superior de Ingenieros Aeronáuticos de Madrid, que a lo largo de muchos cursos, con sus preguntas e inquietudes, nos han ayudado a depurar una metodología de programación para la simulación de sistemas físicos que venía gestándose desde hace ya quince años. A Dña. Cristina de Lorenzo por su minuciosidad en la lectura de este libro.

Por último ("last but not least"), a D. Pablo Ripollés por sus útiles y constructivos comentarios.

Juan A. Hernández
Mario A. Zamecnik
Madrid
Septiembre 2000

Capítulo 1

Algoritmos, programas y ordenadores

Un *algoritmo* es un conjunto finito de reglas que dan una secuencia de operaciones para resolver un problema específico. Un algoritmo, además de ser eficiente y estar bien definido, debe acabar en un conjunto finito de pasos. La entradas de las que partimos, así como los objetivos o salidas que queremos conseguir deben estar bien identificadas.

Un *programa* o *código* es la traducción de un algoritmo a un lenguaje de programación que, ejecutado en un ordenador, permite obtener la solución de un problema.

La *programación* estudia los fundamentos teóricos que nos permiten diseñar, desarrollar y mantener programas o códigos.

Actualmente, la programación juega un papel fundamental en el campo científico y en la ingeniería. Los problemas físicos en estudio requieren modelos matemáticos muy complejos que sólo pueden resolverse mediante el cálculo numérico y su posterior implementación en el ordenador. Por otra parte la *simulación* en ingeniería aporta experimentos computacionales valiosos que, sumados a los experimentos de laboratorio, permiten diseñar y desarrollar de forma más eficiente. Un ejemplo del uso de la simulación en la industria automotriz es el estudio de:

- Aerodinámica para reducir resistencia al avance, vibraciones y ruido.
- Alimentación y combustión del motor.

- Procesos de fundición para reducir pérdidas de material.
- Colisión y seguridad de los ocupantes del vehículo.

El ingeniero de simulación dispone, para programar, de lenguajes de programación de alto nivel. Un lenguaje de alto nivel es aquel que dispone de una sintaxis fácil de entender para el usuario; es un lenguaje próximo al programador. En contraposición existen los lenguajes de programación de bajo nivel o lenguaje máquina, que presentan una sintaxis fácil de entender para el ordenador. Ejemplos de lenguajes de programación de alto nivel son FORTRAN, C, PASCAL, COBOL, ADA, PL/I.

El objeto de este trabajo consiste en exponer y explicar las estructuras básicas de programación tomando el lenguaje FORTRAN como ejemplo de lenguaje. La aplicación a cualquier otro lenguaje de programación se puede hacer en el momento en que se entiendan y se sepan utilizar las estructuras básicas.

Los programas o aplicaciones de ordenador pueden dividirse en dos grandes grupos: aplicaciones de cálculo numérico o simulación numérica y aplicaciones para el procesamiento de datos o información. Antiguamente, el codificador o programador desarrollaba el programa que realizaba los cálculos necesarios para la resolución de un problema, enunciado y formulado por el interesado. Actualmente, debido a la potencia de cálculo de los ordenadores, se pueden hacer simulaciones muy complejas que implican algoritmos de programación muy elaborados. Para poder implementar estos algoritmos es necesario una persona que entienda a la perfección la formulación del problema. De aquí surge la necesidad de que el grupo de trabajo que modela el problema desarrolle sus propios códigos.

1.1. Programación por descomposición funcional

En el diseño de un código de simulación pueden identificarse dos métodos de trabajo: el método de diseño por *descomposición funcional* y el método de diseño *orientado a objetos*.

El método por descomposición funcional consiste en identificar las funciones más importantes de un sistema y refinarlas en componentes más y más pequeñas orientadas a la funcionalidad. De esta forma se generan distintas capas o niveles de funciones. Las capas altas en el diseño representan la abstracción algorítmica (el "qué"del proceso), y las capas bajas representan las operaciones primarias de las acciones de alto nivel. Esta forma de diseño junto con otras características da lugar a la *metodología de programación multicapa* que será desarrollada en el capítulo 6. En esta metodología es importante utilizar los principios de ocultación

de la información y de abstracción de datos y procedimientos. El diseño por descomposición funcional se usa normalmente en el desarrollo de sistemas FORTRAN en la simulación de sistemas físicos. Estos sistemas contienen un conjunto de operaciones matemáticas importante como, por ejemplo, problemas de dinámica de vuelo, mecánica de fluidos y dinámica estructural.

El método orientado a objetos consiste en identificar un conjunto de objetos abstractos con sus atributos, que modelan el sistema. A continuación se definen las operaciones que involucran dichos objetos y se establecen los enlaces entre los mismos. Normalmente, un objeto es una estructura de datos mas procedimientos. Este método concentra su atención en los objetos (las "cosas"del sistema) más que en las acciones que afectan a dichos objetos. El diseño orientado a objetos se usa normalmente en el desarrollo de sistemas C++ o ADA para la simulación de problemas con lógicas complicadas, como por ejemplo simuladores en tiempo real de centrales nucleares y simuladores de vuelo.

1.2. Lenguaje de programación FORTRAN

El lenguaje FORTRAN es un lenguaje muy próximo al lenguaje matemático y especialmente indicado para problemas susceptibles de descomposición funcional. Además, las operaciones vectoriales con matrices están ya incluidas en el lenguaje, lo cual permite al programador escribir basándose en bibliotecas con alto grado de abstracción numérico. Aunque el proceso paralelo es otra de las características importantes del FORTRAN 95, en este libro no analizamos su potencialidad.

El lenguaje FORTRAN fue desarrollado por un equipo de trabajo de IBM a mediados de los años cincuenta bajo el nombre: IBM Mathematical FORmula TRANslation System. Este lenguaje se concibió orientado para aplicaciones numéricas como su propio y definitivo nombre indica: FORmula + TRANslation.

Originalmente el FORTRAN comenzó como el estándar FORTRAN II para más adelante constituir el FORTRAN IV. Posteriormente se aprueba el FORTRAN 77 (Norma ANSI x3.9-1978: Programming Language FORTRAN) siendo este el lenguaje utilizado en cálculo numérico y simulación hasta mediados de los años 90. Las iniciales ANSI se corresponden con el "American National Standards Institute.ªcompañando a la evolución de las arquitecturas de los ordenadores, se aprueban el FORTRAN 90 (Norma ANSI x3.198-1992) cuyas prestaciones mejoran el procesamiento vectorial, y la versión HPF (High Performance FORTRAN) de aplicación en ordenadores de procesamiento paralelo con estructuras de memoria distribuida. Finalmente, la corrección de ciertas anomalías en el FORTRAN 90 sumadas a la versión HPF han dado lugar a una nueva versión denominada FORTRAN 95. En nuestros días el FORTRAN 95 (Norma ISO/IEC 1539-1:1997)

representa el estándar FORTRAN. Las iniciales ISO se corresponden con la "International Standards Organization".

Debido a su antigüedad y a la propiedad de este lenguaje de mantener la compatibilidad con versiones anteriores, el lenguaje posee ciertos vicios que pertenecen a su historia.

En los siguientes capítulos se explica la sintaxis del FORTRAN 95, sin pretender ser exhaustivos y sin hacer referencia a características antiguas u obsoletas del lenguaje. Por lo tanto, la sintaxis que aquí aparece no es la completa del FORTRAN 95 sino un subconjunto de palabras clave y procedimientos seleccionados por una metodología propia de la simulación de sistemas físicos.

1.3. Sistemas operativos

Los componentes físicos de un ordenador constituyen el hardware y el conjunto de programas disponibles constituyen el software.

Los programas más importantes que residen en un ordenador son el *sistema operativo* y el *compilador*. El sistema operativo es un programa que gestiona el uso de sistemas compartidos: procesador, memoria, dispositivos de entradas/salidas, utilidades software comunes. El compilador es un programa que realiza el análisis sintáctico y la traducción del lenguaje de alto nivel (FORTRAN) a lenguaje máquina. Posteriormente, el hardware ayudado del sistema operativo realizará secuencialmente las instrucciones almacenadas en la memoria principal, las cuales constituyen el programa en lenguaje máquina. Otra función del compilador es la de *optimizar* el programa de alto nivel para generar un programa en lenguaje máquina más rápido y más pequeño para una arquitectura dada. El concepto de *optimización* se desarrollará con más detalle en el capítulo 5.

1.4. Codificación de un programa

La secuencia de trabajo para codificar un programa en FORTRAN es la siguiente. Con un editor de texto se escribe el programa mediante palabras clave y etiquetas que controlan el flujo de datos. Una vez finalizado se obtiene el fichero *fuente* (fichero con extensión `.for` o `.f`), a continuación, se compila el fichero fuente y se obtiene el fichero *objeto* (fichero con extensión `.obj` o `.o`). En caso de tener varias unidades de programa en distintos ficheros objeto que forman parte del mismo programa, el compilador ensambla los mismos usando un "linker". Fi-

nalmente, el compilador genera el fichero *ejecutable* (fichero con extensión `.exe` o cualquier otra extensión).

Desde el punto de vista del programador, una variable es un *nombre* o *etiqueta* que identifica un dato. Los datos pueden ser números reales, enteros o cadenas de caracteres. El nombre o etiqueta de una variable es una secuencia de letras mayúsculas y minúsculas y números que deben empezar por una letra, pudiendo tener hasta treinta y una letras y números significativos.

El conjunto de caracteres admitidos está formado por:

- Las letras alfabéticas mayúsculas y minúsculas:
  `a, b, c, d, e, f, g, h, i, ,j, ,k, ,l, m,`
  `n, o, p, q, r, s, t, u, v, w, x, y, z.`
- Los dígitos numéricos: `0, 1, 2 ,3 ,4 ,5 ,6 ,7, 8, 9.`
- Los caracteres especiales: `_` (subguión) y `$` (dólar).

Por otra parte, el nombre de una variable no podrá coincidir en ningún caso con los nombres de las *palabras clave* del lenguaje. Estas palabras clave son reservadas para el lenguaje de programación y no pueden ser utilizadas para otros fines.

Además de los caracteres anteriores, existe un conjunto de caracteres reservado y con significado propio:

Significado	Símbolo	Nombre de carácter
suma	+	más
resta	-	menos
producto	*	asterisco
división	/	barra oblicua
menor que	<	menor
mayor que	>	mayor
paréntesis	(	paréntesis izquierdo
paréntesis	)	paréntesis derecho
separador	.	punto
separador	,	coma
separador	;	punto y coma
especificador	:	dos puntos
delimitador de caracteres	'	apóstrofe
comentarios	!	admiración
delimitador de caracteres	"	comillas
parte de una estructura	%	tanto por ciento
continuación	&	ampersand

Por otra parte, las variables que son cadenas de caracteres están formadas por todos los caracteres anteriores junto con el carácter `?` (interrogación). Si en un programa se utilizan otros caracteres distintos de este conjunto, los resultados pueden diferir de un ordenador a otro.

A continuación enumeramos una serie de reglas prácticas para la codificación de un programa en lenguaje FORTRAN:

1. Se pueden utilizar todas las columnas del editor que sean necesarias para escribir el programa.
2. Los comentarios en FORTRAN se indican con el carácter `!` y pueden colocarse en cualquier parte del fichero fuente. Todo lo que se encuentre a la derecha de este carácter hasta que termina la línea se considera un comentario, que sirve para documentar el código y que permite una comprensión fácil del mismo. Los comentarios no se procesan por el compilador.
3. Cuando una expresión es muy larga y se desea cortar, la podemos partir por donde queramos indicándoselo al compilador con el carácter de continuación `&` al final de la línea que se interrumpe.
4. Los espacios en blanco en un fichero fuente son ignorados por el compilador. Esto nos permite dejar blancos entre los símbolos para facilitar la lectura.
5. El carácter `;` (punto y coma) se puede utilizar como separador entre diferentes sentencias en una misma línea.
6. Tanto las palabras clave del lenguaje como los nombre de las variables que constituyen los datos se pueden escribir combinando mayúsculas con minúsculas para favorecer la lectura. Sin embargo, el compilador no diferencia las letras mayúsculas de las letras minúsculas.

1.5. Representación interna de los datos

El hardware de un ordenador está formado por los siguientes elementos básicos:

- Memoria principal.
- Unidad central de proceso (CPU Central Processing Unit).
- Dispositivos de entrada y salida.

La memoria principal consiste en millones de conjuntos elementales capaces de almacenar un cero o un uno. De esta forma, la memoria queda organizada en

elementos de información que son capaces de almacenar datos o instrucciones de un programa. La representación interna que utiliza un ordenador es base 2. La unidad mínima de información es el bit que posee dos estados: 0 y 1. Grupos de 8 bits constituyen un octeto o byte que posee 2^8 estados diferentes. Las memorias principales pueden ser RAM (Random Access Memory) y ROM (Read Only Memory). En la memoria RAM se puede leer y escribir, pero la información se pierde cuando no se recibe energía eléctrica. En las memorias ROM sólo se puede llevar a cabo la lectura. Sin embargo, la información reside en ella de forma permanente aunque no exista suministro de energía eléctrica.

El origen de las arquitecturas actuales se basa en la arquitectura creada por von Neumann entre los años 1946 y 1952. Esta arquitectura se basa en el concepto de *programa almacenado*. Esto significa que todo programa (instrucciones + datos) debe residir en la memoria. La unidad central de proceso debe traer la información de la memoria, operar y escribir el resultado en la memoria.

Los ordenadores trabajan de forma interna en lenguaje binario. Una variable desde el punto de vista del ordenador es una dirección de memoria, que normalmente se expresa en formato hexadecimal (base 16). Por ejemplo, si la memoria está organizada en grupos de 8 bits, una parte de la memoria con sus direcciones consecutivas se podría esquematizar de la siguiente forma:

Direcciones	Contenidos
`3F29`	1 0 1 0 1 1 1 0
`3F2A`	1 1 0 0 0 0 1 0
`3F2B`	0 0 0 0 0 0 0 0

En este ejemplo, una variable de tipo entero de un byte de tamaño especificada en un programa, podría ser la dirección de memoria `3F2A` y el dato o contenido de esa posición de memoria el número entero `194` en formato decimal y `1 1 0 0 0 0 1 0` en formato binario.

Finalmente, es importante hacer notar que el nombre de una variable es la etiqueta que el programador utiliza para identificarla y que el compilador asocia a una dirección de memoria. El contenido de la variable o la dirección de memoria es el dato, que puede ser un número entero o real, o una cadena de caracteres.

Las cadenas de caracteres también se representan usando un lenguaje binario y en este caso particular se utilizan *cadenas binarias*.

Existen varios códigos que convierten datos tipo carácter en cadenas binarias. La mayoría de los ordenadores utilizan el código EBCDIC (Extended Binary Co-

ded Decimal Interchange Code) o el código ASCII (American Standard Code for Information Interchange). En estos códigos, cada carácter está representado por una cadena binaria. Por ejemplo, el carácter `H` se representa en código EBCDIC con la cadena binaria `11001000` mientras que en código ASCII se representa con la cadena binaria `1001000`.

Los números enteros y reales se representan en lenguaje binario siguiendo un determinado modelo de representación. Generalmente, en FORTRAN 95 el modelo de representación para los datos es el estándar en coma flotante IEEE (IEEE Standard Binary Floating–Point Arithmetic, ANSI/IEEE Standard 754–1985). Este estándar define los formatos y los redondeos entre operaciones de datos reales o enteros.

Cuando se trabaja con distintos ordenadores compartiendo datos sin formato es importante saber el modelo de representación utilizado y convertir, si fuera necesario, los datos de un modelo a otro. Por otra parte, es importante hacer notar que las conversiones de tipo, aunque ocultadas al programador, requieren un gran esfuerzo para el compilador. Por ejemplo, mientras que el dato entero `3` se puede almacenar en el ordenador en un byte de información, el dato real `3.` puede ocupar 4 bytes. La asignación de una variable entera a una variable real requiere un conjunto grande de instrucciones, ocultadas al programador, para convertir el formato de representación de la variable entera a una variable real.

Capítulo 2

Primeros ejemplos de programas de cálculo y álgebra

2.1. to be included

elemental advance = no dummy versus actual assumed shape explicit shape

global, local and scope in modules

1 versus 1.

tab instead of blanks

brackets

mask in intrinsic functions

! and &

;

camel case versus underscore

overloading

forall parallel

enter matrices by row or columns

lower bound: upper bound

array operations C = A + B

public versus private

encapsulamiento y ocultación de datos

FORALL (I=1:N-1, J=1:N, J¿I) A(I,J) = A(J,I)

```
 x   = 1.23456789123456789123456789Q00
xd  = 1.23456789123456789123456789Q00
xdd = 1.23456789123456789123456789Q00
write(*, '(ES)' ) x  ! scientific notation 1.234 (first digit should be greater or equal tl
write(*, '(E)' ) x   ! exponential normalized  notation 0.1234 (first digit is zero )

write(*,*) " Single, double and quadruple precision "
write(*, '(E)' ) x
write(*, '(E)' ) xd
write(*, '(E)' ) xdd




!*******************************************************************************
!*
!*******************************************************************************
subroutine type_element




interface operator (+)
module procedure element
end interface

character(len=20) :: name
real (kind=4) :: x
real (kind=8) :: xd
real (kind=16) :: xdd
```

```
real :: a, b, c;
!  real :: a1, a2, a3;


type (person) :: father = person( "juan"), mother = person("cris")

associate ( name1 => father%name, name2=>mother%name )
name = trim(name1) // trim(name2)
end associate

a = 1; b = 1 ; c = 1;
associate ( a1 =>a, a2 =>b, a3=>c )
a3 = a1 + a2
write(*,*) " a3 = ", a3
end associate
write(*,*) " c = ", c
!    write(*,*) " a3 = ", a3

!   write(*,*) " father =", father % name
write(*,*) " element =", father + mother
write(*,*) " name =", name

x   = 1.23456789123456789123456789Q00
xd  = 1.23456789123456789123456789Q00
xdd = 1.23456789123456789123456789Q00
write(*, '(ES)' ) x  ! scientific notation 1.234 (first digit should be greater or equal tl
write(*, '(E)' ) x   ! exponential normalized  notation 0.1234 (first digit is zero )

write(*,*) " Single, double and quadruple precision "
write(*, '(E)' ) x
write(*, '(E)' ) xd
write(*, '(E)' ) xdd

write(*, '(3(E, :, ","))' ) x, xd, xdd

end subroutine



!*****************************************************************************
!*
!*****************************************************************************
subroutine Hello_world
```

```
write(*,'(a)', advance='no') " Hello world..... "
write(*,'(a)', advance='no') " press enter"
read(*,*)

end subroutine



!*****************************************************************************
!*
!*****************************************************************************
subroutine  commandline

character(len=256) :: line,  enval
integer :: i, iarg, stat, clen, len
integer :: estat, cstat

iarg = command_argument_count()
write(*,*) " iarg = ", iarg
do i=1,iarg
call get_command_argument(i,line,clen,stat)
write (*,'(I0,A,A)') i,': ',line(1:clen)
end do
call get_command(line,clen,stat)
write (*,'(A)') line(1:clen)

call get_environment_variable('HOSTNAME',enval,len,stat)
if (stat == 0) write (*,'(A,A)') 'Host=', enval(1:len)
call get_environment_variable('USER',enval,len,stat)
if (stat == 0) write (*,'(A,A)') 'User=', enval(1:len)


! call execute_command_line('ls -al', .TRUE., estat, cstat)
call execute_command_line('dir', .TRUE., estat, cstat)
if (estat==0) write (*,'(A)') "Command completed successfully"

end subroutine

!*****************************************************************************
!*
!*****************************************************************************
subroutine  allocate_characteristics

real, allocatable :: V(:), A(:, :)
character(:), allocatable :: S
```

```
integer :: i, j, N

real, pointer ::  B(:,:), Diagonal(:)
real, pointer :: memory(:)

real :: x, y, z
class(*), pointer :: p1(:)


N = 10
V = [ ( i/real(N), i=1, N ) ]  ! automatic allocation allocate( V(N) )
write(*,* ) " V = ", V

N = 2
A = reshape( [ (  ( (i/real(N))**j ,i=1, N ), j=1, N ) ], [N, N] )
do i=1, N
write(*,* ) " A = ", A(i,:)
end do

N = 4
A = reshape( [ (  ( (i/real(N))**j ,i=1, N ), j=1, N ) ], [N, N] )
do i=1, N
write(*,'(A, 100f6.2)' ) " A = ", A(i,:)
end do

S = "Hello world"
write(*,*) " S = ", S, len(S)

S = "Hello"
write(*,*) " S = ", S, len(S)


allocate( memory(1:N*N) )
B(1:N,1:N) => memory
memory = 0
forall(i=1:N) B(i,i) = 10.
do i=1, N
write(*,'(A, *(f6.2) )' ) " B = ", B(i,:)
end do
diagonal => memory(::N+1)
write(*,'(A, 100f6.2)' ) " diagonal = ", diagonal
write(*,'(A, 100f6.2)' ) " trace = ", sum(diagonal)
write(*,'(A, 100f6.2)' ) " trace = ", sum(memory(::N+1))

x = 1; y = 2; z = 3;
```

```
write(*,'("i=",I0,", REALs=",*(G0,1X),"....")') i, x, y, z  ! C++ style


allocate( integer :: p1(5) ) ! p1 is an array of integers

select type (p1)
type is (integer)
p1 = [( i, i=1, size(p1) ) ]
write(*,'(" p1 = ", *(I0, 1x) )')  p1

class default
stop 'Error in type selection'
end select

deallocate(p1)
allocate( real :: p1(3) ) ! now p1 is an array of reals

select type (p1)
type is (real)
p1 = [( i, i=1, size(p1) )  ]
write(*,'(" p1 = ", *(G0, 1x) )')  p1

class default
stop 'Error in type selection'
end select



end subroutine
```

2.2. Mi primer programa: "Hello world"

batch bsh

scripts

gfortran

2.3. Suma de una serie numérica

Dar el resultado de la suma de los 100 primeros términos de las siguientes series:

1. Serie de números naturales.
2. Serie de números naturales impares.
3. Serie numérica donde el término general de la serie es: $a_n = 1/n^2$ desde $n = 1$.
4. Serie numérica donde el término general de la serie es $a_n = 1/n!$ desde $n = 1$.
5. Serie numérica donde el término general de la serie es $a_n = (-1)^{n+1}/(2n-1)$ desde $n = 1$.

2.4. Operaciones con matrices y vectores

Considerar los vectores $V, W \in \mathbb{R}^N$ de componentes:

$$\{v_i = \frac{1}{i^2}, \quad i = 1 \dots N\},$$

$$\{w_i = \frac{(-1)^{i+1}}{2i-1}, \quad i = 1 \dots N\}.$$

Considerar la matriz $A \in \mathcal{M}_{N\times N}(\mathbb{R})$ donde su término genérico vale $a_{ij} = (i/N)^j$. Escribir un programa para calcular las operaciones siguientes con $N = 100$:

1. Suma de todas las componentes del vector V y del vector W.
2. Suma de todas las componentes de la matriz A.
3. Suma de las componentes del vector W mayores que cero.
4. Producto escalar de los vectores V y W.
5. Producto escalar del vector V y la columna N de la matriz A.
6. Suma de las componentes de vector que resulta de multiplicar la matriz A por el vector V.
7. Traza de la matriz A.

2.5. Asignación dinámica de memoria

Dada la matriz $A \in \mathcal{M}_{M\times M}(\mathbb{R})$ de término genérico

$$\{a_{ij} = (i/M)^j, \quad i = 0, \dots M-1, \quad j = 0, \dots M-1\}.$$

calcular las siguientes operaciones:

1. Calcular
$$\sum_{M=1}^{10} traza(A)$$
2. Calcular
$$\sum_{M=1}^{5} traza(A^2)$$
3. Calcular con $M = 4$
$$traza\left(\sum_{k=1}^{5} A^k\right)$$

2.6. Funciones definidas a trozos

Sean los vectores $X, F \in \mathbb{R}^{N+1}$. Las componentes de X almacenan los valores discretos del dominio de definición y F las imágenes correspondientes de la función $F : \mathbb{R} \to \mathbb{R}$ continua a trozos siguiente:

$$F(x) = \begin{cases} 1, & a \le x \le -\dfrac{\pi}{2}, \\ \cos(\pi x), & -\dfrac{\pi}{2} < x < \dfrac{\pi}{2}, \\ 0, & \dfrac{\pi}{2} \le x \le b. \end{cases}$$

Considerar una partición equiespaciada de la forma:

$$\{x_i = a + i\Delta x, \quad i = 0 \dots N\}, \qquad \Delta x = \frac{b-a}{N}, \qquad a < -\frac{\pi}{2}, \qquad b > \frac{\pi}{2}.$$

Se pide calcular la suma;

$$S_N = \sum_{i=0}^{N} F_i \Delta x$$

1. con $N = 10$
2. con $N = 20$
3. con $N = 100$

2.7. Series de funciones

Aproximar mediante un desarrollo en serie de potencias de la forma

$$f(x) = \sum_{k=0}^{M} a_k \; x^k, \qquad\qquad a_k = \frac{f^{(k)}(0)}{k!},$$

las funciones $F : \mathbb{R} \to \mathbb{R}$, siguientes:

1. $f(x) = e^x$ y calcular el valor $f(1)$ con $M = 5$.
2. $f(x) = \sin(x)$ y calcular el valor $f(\pi/2)$ con $M = 8$.
3. $f(x) = \cosh(x)$ y calcular el valor $f(1)$ con $M = 10$.
4. $f(x) = \dfrac{1}{1-x}$ y calcular el valor $f(0{,}9)$ con $M = 20$.
5. $f(x) = e^x$ y calcular el valor más preciso de $f(1)$ con doble precisión.
6. $f(x) = \sin(x)$ y calcular el valor más preciso $f(\pi/2)$ con doble precisión.
7. $f(x) = \cosh(x)$ y calcular el valor más preciso de $f(1)$ con doble precisión.
8. $f(x) = \dfrac{1}{1-x}$ y calcular el valor más preciso de $f(0{,}9)$ con doble precisión.

2.8. Lectura y escritura de ficheros

Crear los ficheros de datos ForTran con nombres `input_1.dat` e `input_2.dat` con la información siguiente:

Contenido del fichero de entrada `input_1.dat` :

```
1       Datos de entrada 1
2
3       1.2      3.4     6.2     -14.0     0.1
4       -25.2    -8.6    5.1      9.9     17.0
5       -1.0     -2.0    -5.4     -8.6     0.0
6       3.14     -11.9   -7.0     -12.1    9.2
7       6.66     5.32    0.001    0.2      0.001
```

Contenido del fichero de entrada `input_2.dat` :

```
1       Datos de entrada 2
2
3       1.2      3.4     6.2     -14.0     0.1       4.89    7.54
4       -25.2    -8.6    5.1     12.0      9.9       12.24   17.0
5       0.0      34.5    -1.0    -2.0      -43.04    -8.6     0.0
6       3.14     -11.9   71.0    7.0      17.0       -12.1    9.2
7       6.66     5.32    0.001    0.2      0.001     0.008   -0.027
8       54.0     77.1    -9.002  -13.2     0.017     65.53   -0.021
9       23.04    -51.98  -34.2    9.99     5.34      8.87     3.22
```

Escribir un programa que gestione los datos de los ficheros anteriores siguiendo los pasos siguientes:

Declarar las matrices $A \in \mathcal{M}_{N\times N}(\mathbb{R})$, $B \in \mathcal{M}_{N\times 3}(\mathbb{R})$, $C \in \mathcal{M}_{N\times 2}(\mathbb{R})$ y los vectores $U, V, W, T \in \mathbb{R}^N$.

Leer el fichero de entrada (`input_1.dat` o `input_2.dat`) de la forma siguiente:

1. Cargar el fichero completo en la matriz A.
2. Cargar las cuatro primeras columnas del fichero en los vectores U, V , W y T.
3. Cargar la primera columna en el vector T y las tres últimas columnas en la matriz B.
4. Cargar la segunda columna en el vector U y las dos últimas columnas en la matriz C.

5. Cargar las columnas 1, 2 y 4 en la matriz B.

Además, el programa debe crear el fichero de salida (`output_1.dat` o `output_2.dat`), donde se irán escribiendo las matrices y vectores de los apartados anteriores. El formato de escritura debe ser el de números reales con cinco decimales.

Para el enunciado anterior, escribir los programas siguientes:

1. Programa 1 : Asignación estática de memoria.
   Ejecutar el programa por separado para los ficheros `input_1.dat` e `input_2.dat`.
   Para ello modificar las dimensiones y en nombre de los ficheros en el programa fuente.
2. Programa 2 : Asignación dinámica de memoria.
   Ejecutar el programa una única vez para gestionar los datos de los ficheros de entrada `input_1.dat` e `input_2.dat`.

2.9. Sistemas lineales de ecuaciones

Implementar un módulo para la resolución de sistemas lineales de ecuaciones algebraicas. Los métodos de resolución propuestos son el de eliminación Gaussiana, factorización LU, factorización LU de la biblioteca *Numerical Recipes* y Jacobi.

Para cada método se pide:

- Validar los resultados con varios casos de prueba con dimensiones distintas.
- Evaluar tiempos de ejecución.
- Comparar resultados con los métodos restantes.

Aplicación : Estudiar el condicionamiento de sistemas lineales de ecuaciones para matrices aleatorias y de Vandermonde.

2.10. Sistemas no lineales de ecuaciones

Implementar un módulo para la resolución numérica de ecuaciones no lineales. Para funciones $F : \mathbb{R} \to \mathbb{R}$, los métodos de resolución propuestos son el de

la bisección y el de Newton-Raphson. Para funciones $F : \mathbb{R}^N \to \mathbb{R}^N$, se proponen el método de Newton-Raphson con matriz Jacobiana analítica y el método de Newton-Raphson con matriz Jacobiana numérica. Para la validación de los métodos propuestos, se pide implementar un módulo con al menos tres funciones $F : \mathbb{R} \to \mathbb{R}$ y al menos tres funciones $F : \mathbb{R}^N \to \mathbb{R}^N$. Este módulo debe contener las derivadas y matrices Jacobianas correspondientes de las funciones propuestas.

En el informe correspondiente, presentar tablas de soluciones numéricas en cada paso de iteración para las funciones de prueba propuestas.

2.11. Autovalores y autovectores

Implementar un módulo para el cálculo de autovalores y autovectores de una matriz. Los métodos de resolución propuestos son el método de la potencia y el método de la potencia inversa. Implementar el método de la potencia inversa a partir de la matriz inversa y resolviendo el sistema lineal correspondiente.

Para cada método se pide:

- Validar los resultados con varios casos de prueba con dimensiones distintas.
- Evaluar tiempos de ejecución. Comparar tiempos de ejecución del método de la potencia inversa mediante los dos algoritmos propuestos : matriz inversa y solución del sistema lineal.

Aplicación : Estudiar el condicionamiento de sistemas lineales de ecuaciones para matrices aleatorias y de Vandermonde. Calcular la relación $\lambda_{max}/\lambda_{min}$ de los casos de prueba presentados en el hito 1 y relacionar y discutir los resultados.

```
subroutine   power_method

integer :: i, j, k
integer, parameter :: PI = 4 * atan(1d0)
integer, parameter :: N = 20
real :: x(0:N), Vandermonde(0:N, 0:N), sigma
real :: a=-1, b=1
real V(0:N), V0(0:N)


x = [ ( a + (b-a)*i/N, i=0, N) ]
```

```
forall(i=0:N, j=0:N) Vandermonde(i,j) = x(i)**j

V = 1
V0 = 0
do while( abs(norm2(V)-norm2(V0)) > 1d-5 )
V0 = V
V = matmul( Vandermonde, V ) / norm2(V)
write(*,*) maxval(V)
end do
sigma = dot_product( V, V )
write(*,*) "sigma = ", sigma


end subroutine
```

2.12. Derivación numérica

1. Obtener las fórmulas de las derivadas numéricas primeras descentradas, con tres puntos equiespaciados a una distancia Δx.

2. A partir de la función $f(x) = \mathrm{e}^x$ en el punto $x = 0$, representar gráficamente el error total de las derivadas numéricas frente al valor de Δx en precisión simple y doble. En particular, representar gráficamente las derivadas primeras adelantada (definición de derivada), centrada y descentradas y la derivada segunda, con tres puntos equiespaciados a una distancia Δx. Discutir los resultados obtenidos.

3. Resolver los problemas de contorno en ecuaciones diferenciales ordinarias siguientes:

   - **Problema 1:**

     $$u'' + u = 0, \quad x \in [-1, 1], \qquad u(-1) = 1, \quad u(1) = 0,$$

   - **Problema 2:**

     $$u'' + u' - u = \sin(2\pi x), \quad x \in [-1, 1], \qquad u(-1) = 0, \quad u'(1) = 0.$$

   Para los problemas citados anteriormente se pide:

   a) A partir de las derivadas numéricas con tres puntos equiespaciados escribir el sistema de ecuaciones resultante.

   b) Obtener la solución numérica mediante la resolución de un sistema lineal de ecuaciones, con $N = 10$ y $N = 100$.

   c) Representar gráficamente los resultados obtenidos.

2.13. Integración numérica

Implementar un módulo para la resolución numérica de integrales definidas de funciones $F : \mathbb{R} \to \mathbb{R}$. Los métodos de resolución propuestos son las reglas del rectángulo, punto medio, trapecio y Simpson. Implementar un módulo de funciones $F : \mathbb{R} \to \mathbb{R}$ de prueba para validar los métodos numéricos propuestos. Este módulo debe contener al menos tres funciones con funciones primitivas conocidas y una función cuya función primitiva sea desconocida.

Evaluar el error de las soluciones numéricas para cada método propuesto y para distintos valores del incremento de la partición.

Capítulo 3

Unidades de programa y procedimientos

Un programa está formado por diferentes *unidades de programa* conectadas entre sí mediante un ejecutivo o unidad de programa principal, y que se comunican para dar lugar a un flujo de datos. Los diferentes tipos de unidades de programa son:

program **module**

Cada unidad de programa puede ser compilada de forma independiente de cualquier otra y tiene entidad propia. El programa principal (**program**) es la unidad de programa que determina la jerarquía funcional del programa y constituye el ejecutivo. Éste gobierna cuándo debe comenzar la ejecución de las instrucciones asociadas a las diferentes unidades que integran el programa principal. La unidad **module** puede consistir en una especificación e inicialización de variables y puede contener *procedimientos* o conjuntos de instrucciones para realizar procesos. Para que otras unidades de programa puedan acceder a estas variables o procedimientos, éstas deberán estar asociadas con dicha unidad **module** a través de la sentencia **use**. Generalmente, en un programa es necesario hacer cálculos repetitivos en diferentes partes del mismo, y para evitar el tener código duplicado dentro del mismo programa se diseña éste mediante un análisis funcional de las necesidades, estructurando las diferentes funciones necesarias e implementándolas en diferentes procedimientos que pueden ser:

subroutine **function**

La subrutina (**subroutine**) es un conjunto de instrucciones que, generalmente, se repite de forma cíclica y que a partir de unos datos de entrada permite obtener unos datos de salida. Los datos de salida y datos de entrada se denominan respectivamente *argumentos de salida* y *argumentos de entrada*. Los argumentos de entrada son las entradas de datos que la subrutina necesita para efectuar las funciones para las cuales ha sido diseñada. Los resultados de los cálculos de la subrutina se devuelven en los argumentos de salida. La función (**function**) es, también, un conjunto de instrucciones que ejecutan un determinado algoritmo. La única diferencia que tiene con la subrutina es que sus argumentos son todos de entrada, y el resultado de la función sólo puede ser cualquier tipo de dato que se devuelve en el propio nombre de la función. La utilidad específica de esta unidad de programa es la implementación de funciones utilizadas en matemática, donde las entradas son las variables independientes y los parámetros de la función y la salida es su valor.

Las estructuras de programa que aquí se explican utilizan una sintaxis basada en el estándar FORTRAN 95. Para explicar la sintaxis, se utilizan los siguientes criterios: las palabras clave del lenguaje se indican en negrita, las especificaciones y los nombres se indican en letra inclinada, y los elementos opcionales se representan encerrados entre corchetes. Esta notación es próxima a la notación BNF (Backus-Naur Form) utilizada en la norma ANSI/ISO para representar la sintaxis del lenguaje FORTRAN. Además, para facilitar la lectura, la sintaxis del lenguaje ha sido enmarcada en cuadros o cajas. Aunque la sintaxis de las especificaciones es objeto del capítulo 4, en este capítulo se darán ejemplos sencillos de unidades de programa.

3.1. Programa principal

La estructura y sintaxis del programa principal es la siguiente:

program *nombre*

[*sentencia use*]

[*especificación de variables*]

sentencias

end program [*nombre*]

El programa principal se puede asociar mediante la sentencia **use** a cualquier otra unidad **module**. La parte de *especificación de variables* tiene generalmente como función, reservar un espacio de memoria en cantidad y formato adecuados para permitir el trabajo de la unidad. Las *sentencias* (también llamada parte ejecutable) contienen un conjunto de tareas o instrucciones que son las que se ejecutan y para las cuales ha sido diseñada la unidad de programa. Es importante respetar el orden de estas dos partes: primero se especifican las variables y luego se escriben las sentencias. A continuación se presenta un ejemplo de unidad de programa **program**:

```
program suma

complex :: a, b, c

        a = (1e0, 3e0)
        b = (5e0, -4e0)
        c = a + b

        write (*,*) ' c = ', c

end program suma
```

La primera línea de código indica que se trata de una unidad de programa **program** de *nombre* `suma`. Las segunda línea constituye la declaración de variables que usará dicha unidad de programa. En ellas se declara que `a`, `b` y `c` son variables complejas. Las líneas que siguen son las sentencias que ejecutará el programa. En ellas se inicializan las partes real e imaginaria de las variables con datos y luego se suman. Es importante hacer notar que el compilador debe conocer el tipo de variables para poder sumarlas de forma correcta. En este caso el compilador identifica tres variables complejas y las opera con la suma definida en el cuerpo de los números complejos (suma de partes reales y suma de partes imaginarias). El resultado se imprime mediante la sentencia **write**. El primer asterisco `*` de la sentencia **write** indica que se imprime en la pantalla y el segundo asterisco que se imprime con los formatos adecuados para el texto `'c = '` y para el complejo

c. Finalmente, se cierra el programa con las palabras clave **end program suma**.

3.2. Subrutinas

La sintaxis de las subrutinas es la siguiente:

[**recursive**] **subroutine** *nombre* [(*argumento*$_1$, *argumento*$_2$, ...)]

[*sentencia use*]

[*especificación de argumentos*]

[*especificación de variables*]

sentencias

end subroutine [*nombre*]

La subrutina se puede asociar mediante la sentencia **use** a cualquier otra unidad **module**. En este caso, la *especificación de argumentos* no significa reservar espacio de memoria y sólo consiste en una especificación del formato y tipo de los argumentos de la subrutina. A continuación, *la especificación de variables* reserva un espacio de memoria para las variables de la subrutina. Las *sentencias* corresponderán a la tarea que realiza esa subrutina y que se repetirá cada vez que esta sea llamada desde otra unidad de programa.

A continuación se presenta un ejemplo de unidad de programa **subroutine**:

```
subroutine coeficientes (a, b, c)

real, intent (out) :: a, b, c

         real :: t1, t2

         write (*,*) ' t1, t2 = '
         read  (*,*)   t1, t2
         a = t1 + t2
         b = t1 * t2
         c = t2 / t1

end subroutine coeficientes
```

La primera línea de código indica que se trata de una unidad de programa **subroutine** de *nombre* `coeficientes` con los argumentos `a, b` y `c`. En la segunda línea se especifica que los argumentos `a, b` y `c` son todos variables reales y el atributo **intent** especifica que los tres argumentos son de salida de la subrutina. En la tercera línea se declaran las variables locales `t1` y `t2` como variables reales. En las siguientes *sentencias* se introducen datos en las variables `t1` y `t2` a través del teclado y se opera con ellas para calcular los coeficientes `a, b` y `c`. Finalmente, se cierra la subrutina con las palabras clave **end subroutine coeficientes**.

3.3. Funciones

La sintaxis de las funciones es la siguiente:

[**recursive**] **function** *nombre* [(*argumento*$_1$, *argumento*$_2$, ...)]

[*sentencia use*]

[*especificación de argumentos*]

especificación de nombre

[*especificación de variables*]

sentencias

nombre = *expresión*

end function [*nombre*]

La función se puede asociar mediante la sentencia **use** a cualquier otra unidad **module**. Como ocurre en la subrutina, en este caso, la *especificación de argumentos* no significa reservar espacio de memoria y sólo consiste en una declaración de tipo de esos argumentos. Sin embargo, la *especificación de variables* en sí reserva un espacio de memoria para las variables internas o locales de la función. A continuación, se ejecutan las operaciones para las cuales ha sido diseñada esta función y finalmente, en la variable que lleva el *nombre* de la función se devuelve el valor de la función.

La unidad de programa **function** ha sido concebida para la implementación de funciones matemáticas que necesitan ser utilizadas en distintas partes de un cálculo. Supóngase que la función:

$$f(x) = \frac{sen(ax)}{a^2} - \frac{x\ cos(ax)}{a}$$

se debe integrar y derivar numéricamente varias veces en un programa. La idea es implementarla en una unidad de programa **function** y llamarla cada vez que se necesita. Esta función matemática se podría implementar de la siguiente forma:

```
function f (x, a)

       real,  intent(in)     ::  x, a
       real,  intent(out)  ::  f

              f = sin(a*x)/a**2  -  x*cos(a*x)/a

end function f
```

Los argumentos de entrada son `x` y `a`, y la salida es el escalar `f` que es el nombre de la unidad de programa **function**.

En el caso de que queramos implementar un vector o una matriz de funciones, el nombre de la función se puede especificar como un vector o una matriz. En el siguiente ejemplo se muestra un vector de funciones, que puede ser el vector de posición de una partícula en el plano, en función del tiempo.

```
function Posicion ( t )

       real, intent(in)     :: t
       real, intent(out)  :: Posicion(2)


              Posicion(1) = t**2 + 1.
              Posicion(2) = t + 3.


end function Posicion
```

3.4. Módulos

La sintaxis es la siguiente:

module *nombre*

[*sentencia use*]

[*definición de nuevos tipos y operaciones*]

[*especificación de variables*]

[**contains**

subrutinas y funciones]

end module [*nombre*]

La unidad **module** se puede asociar mediante la sentencia **use** a cualquier otra unidad **module** diferente, pudiendo contener ésta variables y procedimientos. Así, estas variables y procedimientos externos son ahora conocidos por la primera unidad **module**. Además, se pueden definir nuevas operaciones y asignaciones y mediante la parte de *especificación de variables* se puede reservar espacio de memoria e inicializar con datos un conjunto de variables. También puede contener un conjunto de procedimientos que pueden ser funciones o subrutinas. El conjunto de variables que se especifica en la unidad **module** se consideran como variables externas y pueden ser compartidas las subrutinas y funciones de esta unidad y por unidades de programa que se asocien o comuniquen con esta unidad **module**.

A continuación se presenta un ejemplo de unidad de programa **module**:

```
module fisico

        real  ::  nu=0.7e0, rho=0.1e0, gamma=1.4e0
        logical  ::  init=.false.

end module fisico
```

La primera línea de código indica que se trata de una unidad de programa **module** de nombre `fisico`. Se declaran tres variables como reales y una variable lógica que son inicializadas en la propia declaración. Finalmente, se cierra el bloque de datos con las palabras clave **end module fisico**.

El siguiente ejemplo de unidad de programa pone de manifiesto el uso de un programa como el *ejecutivo* de un proceso.

```
program raices

  use segundo_grado

  real   ::   a, b, c
  complex  :: x1, x2


 !  *** raices de a x^2 + b x + c = 0

       read(*,*) a, b, c

       call raices_polinomio (a, b, c, x1, x2)

       write (*,*)  x1, x2

end program
```

Este programa calcula las raíces de una ecuación de segundo grado. La primera línea de código indica que se trata de una unidad de programa **program** de nombre `raices`. En la segunda línea se hace una asociación mediante la sentencia **use** al módulo `segundo_grado` que contiene la subrutina `raices_polinomio`. En las siguientes líneas se declaran los coeficientes de la ecuación como reales y las raíces como variables complejas. Las líneas que siguen son las sentencias que ejecutará el programa. Se asignan valores a los coeficientes de la ecuación a través del teclado y se calculan sus raíces imprimiendo los resultados en la pantalla del ordenador. Finalmente, se cierra el programa con las palabras clave **end program**. La unidad de programa `segundo_grado` contiene la subrutina `raices_polinomio`.

```
module segundo_grado

     subroutine  raices_polinomio (a, b, c, x1, x2)

          real,    intent (in)  :: a, b, c
          complex, intent (out) :: x1, x2

        x1 = ( - b + sqrt( b**2 - 4 * a * c ) ) /(2 * a)

        x2 = -b/a - x1

     end subroutine raices_polinomio

end module segundo_grado
```

Capítulo 4

Sentencias

Las *sentencias* o *parte ejecutable* contienen un conjunto de tareas o instrucciones que son las que se ejecutan y para las cuales ha sido diseñada la unidad de programa. Estas sentencias se pueden clasificar de la siguiente forma:

1. Asignaciones.
2. Sentencias para el control del flujo.
3. Operaciones de entrada y salida.
4. Llamadas a otras unidades de programa.

4.1. Asignaciones

Una asignación es una sentencia que asigna un dato en la posición de memoria de una variable. La estructura básica de una asignación es:

$$\boxed{\textit{nombre} = \textit{expresión}}$$

donde *expresión* puede ser una constante, una expresión matemática o una expresión lógica.

Es importante hacer notar que la asignación de un dato a una variable, que en FORTRAN se realiza mediante el signo `=`, no coincide con la igualdad matemática

a la que estamos acostumbrados. Estas estructuras tienen un sentido de funcionamiento de derecha a izquierda. Primero se evalúa lo que está a la derecha del signo `=` y al finalizar esta evaluación, la ejecución del signo igual es la que realiza la carga del dato en la posición de memoria. Por lo tanto, a la izquierda del signo igual siempre estará el nombre de una variable identificando una posición de memoria y nunca una constante o una expresión. La igualdad matemática `=` tiene su paralelismo en el lenguaje FORTRAN en una igualdad entre expresiones lógicas, que se representa por el símbolo `==`, y que analizaremos más adelante.

Las expresiones están formadas por variables y constantes relacionadas entre sí mediante operadores y paréntesis. Estos operadores clasificados según un orden de prioridad propio del FORTRAN son los siguientes: `**`, `*`, `/`, `+`, `-`, `<`, `>`, `<=`, `>=`, `==`, `/=`, `.and.`, `.or.` que corresponden respectivamente a la potencia, multiplicación, división, suma, resta, evaluación lógica menor que, mayor que, menor o igual que, mayor o igual que, igual que, no igual que, `and` lógico y `or` lógico. En el caso de no existir paréntesis en una expresión, la prioridad en las operaciones está determinada por el orden anterior. Sin embargo, la inclusión de paréntesis nos permite especificar en qué orden deben hacerse las operaciones. Pasamos a realizar una serie de ejercicios prácticos sobre asignaciones a variables. Estos ejemplos utilizan funciones intrínsecas propias del lenguaje que serán explicadas más adelante en este capítulo.

1. Determinar las raíces de la ecuación de segundo grado: $ax^2 + bx + c = 0$.

```
x1 = ( - b + sqrt( b*b - 4*a*c ) )  / (2*a)
x2 = ( - b - sqrt( b*b - 4*a*c ) )  / (2*a)
```

2. Evaluación de una asignación.

```
x = 3
x = x + ( x*x + 1 )/2
```

   El resultado x es igual a 8. Las asignaciones se hacen de derecha a izquierda. Primero se evalúa la expresión de la derecha y luego el valor resultante se le asigna a la variable al lado izquierdo de la igualdad. En primera línea de código a la variable x se le asigna un 3. En la segunda línea se evalúa la expresión $x + (x^2 + 1)/2$ y como resultado se obtiene 8, que es el valor que se asigna nuevamente a la variable x .

3. Determinación de π. Como sabemos que $tg(\pi/4) = 1$, entonces:

```
pi = 4. * atan ( 1.0 )
```

4. Escribir en FORTRAN las siguientes expresiones matemáticas:

   a) $p = \dfrac{\pi}{2}\sqrt{\dfrac{(l^2 + 10m^2)^{\frac{4}{5}}}{m\ n\ k}}$

```
p = pi/2 * sqrt (  (l*l + 10*m*m)**(4./5.)  / (m*n*k)  )
```

b) $z = 2\pi a \tan(\Phi)$

```
z = 2 * pi * a * tan ( Phi )
```

c) $x = \dfrac{ln(cy + d) - b}{a}$

```
x = ( log( c*y + d ) - b )/a
```

d) $z = x^y$

```
z = x**y
```

e) $z = e^{ax+bsen(x)}$

```
z = exp( a*x + b*sin(x) )
```

5. Evaluación de potencias. La evaluación de $z = x^4$ se puede escribir en FORTRAN de las dos formas siguientes:

```
z = x**4
z = x * x * x * x
```

Las dos líneas de código son equivalentes desde el punto de vista del resultado numérico. Sin embargo, generalmente, la primera es mucho más costosa desde el punto de vista de operaciones a realizar. Si el compilador no es muy inteligente, para evaluar x^4 lo que hace es calcular: $e^{4\ ln(x)}$, para lo cual necesita sumar una serie convergente para calcular $ln(x)$, el resultado multiplicarlo por 4 y volver a sumar una serie para evaluar $e^{4\ ln(x)}$. El número de operaciones necesario para llevar a cabo esta evaluación es del orden de diez veces mayor que el número de operaciones necesarias para evaluar $z = x\ x\ x\ x$. Ciertos compiladores inteligentes pueden distinguir si el exponente es un número entero o un número real y en consecuencia evaluar la expresión haciendo el mínimo número de operaciones.

6. Evaluación de una expresión lógica.

```
logical entrada
character (len=1) :: lectura_teclado

lectura_teclado = 'x'
entrada = ( (lectura_teclado == 'n') .or.  &
            (lectura_teclado == 's')        )
```

La primera línea declara la variable `entrada` como una variable lógica y en la segunda se declara `lectura_teclado` como una variable carácter de longitud uno. A continuación, se asigna el carácter `x` a la variable `lectura_teclado` y se evalúa una expresión lógica para saber si la variable `lectura_teclado` es igual a `s` o `n`. El resultado de la evaluación es `.false.`

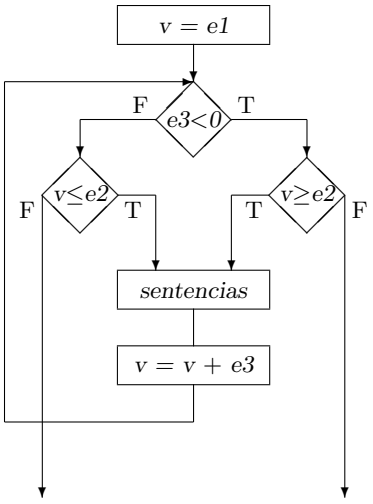
4.2. Control del flujo

Existen estructuras de programación que mediante el uso de palabras clave permiten controlar el flujo de datos a lo largo de una unidad de programa. Las principales sentencias para el control del flujo son las siguientes:

- Bucle **do**

  La estructura de un bucle **do** sirve para ejecutar cíclicamente sentencias. La sintaxis y el diagrama de flujo se representan a continuación:

```
do v = e1, e2, e3
        sentencias
end do
```



  El número de veces que se ejecutan sentencias es un número dado y que se determina a partir de *e1, e2* y *e3*, donde:

  - *v* es la variable o índice del bucle.
  - *e1* es la expresión aritmética que constituye el valor inicial al cual se inicializa *v*.
  - *e2* es una expresión aritmética que constituye el valor final que alcanza la variable *v*.
  - *e3* es una expresión aritmética que constituye el incremento. Si *e3* es omitida, se considera que el incremento es 1.

  Inicialmente, se evalúan las expresiones aritméticas enteras *e1*, *e2* y *e3*. Se asigna en *v* el valor de *e1* y se comprueba el signo de *e3*. En función del signo de *e3* se comprueba si *v* ha alcanzado su valor final. Si *v* no ha alcanzado el valor final *e2*, se ejecutan *sentencias* y se incrementa o decrementa el valor de *v*. Se vuelve a comprobar el valor de *v* y se siguen ejecutando *sentencias* una y otra vez hasta que *v* alcanza o supera el valor final. La estructura del

bucle se utiliza cuando se conoce explícitamente el número de iteraciones que hay que ejecutar *sentencias*. La variable del bucle *v* no se puede modificar por *sentencias* dentro del bucle. Por otra parte, la sentencia **exit** permite transferir el control desde dentro del bucle a la primera sentencia ejecutable siguiente al final del bucle. De esta forma, con la sentencia **exit** se abandona el bucle antes de que se alcance *e2*.

Pasamos a realizar una serie de ejercicios prácticos sobre las sentencias para el control del flujo.

1. Calcular el factorial de n.

```
f = 1

do i = n, 2, -1
       f = f * i
end do
```

2. Evaluar la función seno en n puntos equiespaciados en el intervalo $[x_0, x_f]$.

```
do i = 1, n
       x = x0 + ( xf -x0 ) * ( i - 1 ) / ( n - 1 )
       y(i) = sin ( x )
end do
```

3. Calcular la matriz b transpuesta de una matriz a de n filas y m columnas. La transpuesta de a es $b_{ji} = a_{ij}$.

```
do i = 1, n
   do j = 1, m
       b(j,i) = a(i,j)
   end do
end do
```

4. Calcular la matriz suma c de dos matrices a y b de n filas y m columnas. La matriz suma es $c_{ij} = a_{ij} + b_{ij}$.

```
do i = 1, n
   do j = 1, m
       c(i,j) = a(i,j) + b(i,j)
   end do
end do
```

5. Calcular el vector v como multiplicación de una matriz a de n filas y m columnas por un vector u de m filas. El vector v en notación de subíndices mudos es $v_i = a_{ij}u_j$.

```
do i = 1, n
   s = 0.0
   do j = 1, m
       s = s + a(i,j) * u(j)
   end do
   v(i) = s
end do
```

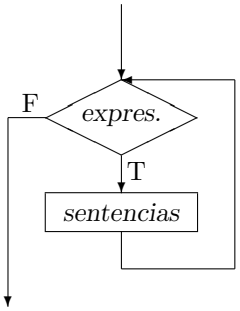
6. Calcular la matriz c como producto de una matriz a de n filas y m columnas y una matriz b de m filas y p columnas. La matriz resultante es $c_{ij} = a_{ik}b_{kj}$.

```
do i = 1, n
   do j = 1, p
              s = 0.0
              do k = 1, m
                s = s + a(i,k) * b(k,j)
              end do
              c(i,j) = s
   end do
end do
```

- Bucle **do while**

  Esta estructura se utiliza cuando no se conoce *a priori* el número de veces que es necesario ejecutar un bloque de sentencias o instrucciones. La sintaxis y el diagrama de flujo de esta estructura es la siguiente:

do while (*expresión*)
sentencias
end do



Esta estructura ejecuta *sentencias* cíclicamente un número indeterminado de veces. La salida de esta estructura se produce cuando el valor lógico de la *expresión* sea falso (F). Normalmente, en cada ciclo las variables que intervienen para evaluar *expresión* cambian, por lo que en algún momento *expresión* se hace falsa.

En el siguiente ejemplo se lee del teclado la variable entera `entrada_teclado` tantas veces como sea necesario hasta que la lectura sea 1 ó 2.

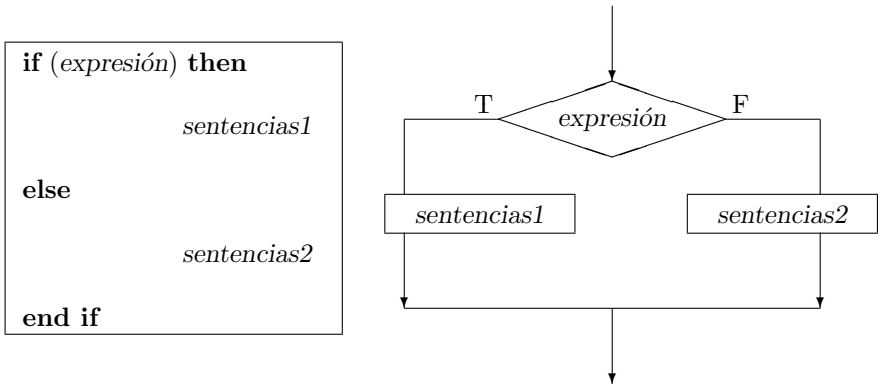
```
do while ( entrada_teclado \= 1 .or. entrada_teclado \= 2 )

   write(*,*) ' Elija su opción: 1, 2 '
   read(*,*)    entrada_teclado

end do
```

- **if** lógico

  Esta estructura permite hacer que el flujo de datos pase por una rama u otra dependiendo de una expresión lógica. La sintaxis y el diagrama de flujo de esta estructura es la siguiente:



  El flujo principal llega a esta estructura. Se evalúa la condición lógica *expresión*, y se obtiene un valor lógico verdadero o falso. Si es verdadero (T), el flujo principal discurre por la rama izquierda ejecutando *sentencias1*. Si es falso (F), el flujo discurre ejecutando *sentencias2*. A continuación, el flujo abandona esta estructura continuando con las instrucciones que siguen. La rama de la derecha es opcional. Con esta estructura la sintaxis queda definida de forma recursiva, es decir, las sentencias dentro de esta estructura pueden ser, a su vez, la misma estructura.

- Estructura de condiciones lógicas **if–elseif**

  La presente estructura permite combinar un conjunto de condiciones lógicas en una sola estructura. En realidad, esta estructura para el control del flujo ya está definida recursivamente en la estructura **if** lógico analizada anteriormente. Sin embargo, dada la importancia de esta estructura pasamos a analizarla por separado. El flujo principal entra en esta estructura evaluando la expresión lógica *exp1*. Si *exp1* es verdadera, se ejecutan *sentencias1* y se abandona la estructura, volviendo al flujo principal para continuar a la siguiente instrucción. Si *exp1* es falsa, se evalúa *exp2*. Si *exp2* es verdadera,

se ejecutan *sentencias2*, abandonándose posteriormente el flujo. Si *exp2* es falsa, se procede de forma similar evaluando *exp3*, *exp4*,...

if (*exp1*) **then**

sentencias1

elseif (*exp2*) **then**

sentencias2

elseif (*exp3*) **then**

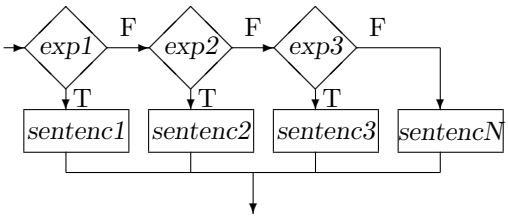
sentencias3

.....

else

sentenciasN

end if



En el siguiente ejemplo se define una función por trozos representada por la siguiente expresión:

$$y(x) = \begin{cases} -y_{max}, & -x_0 \geq x, \\ y_{max}\dfrac{x}{x_0}, & -x_0 < x < x_0, \\ y_{max}, & x \geq x_0. \end{cases}$$

```
if ( x <= - x0 ) then
                          y = - ymax
elseif ( x < x0 ) then
                          y = ymax * x / x0
else
                          y = ymax
end if
```

- Estructura **case**

  Una construcción **case** ejecuta condicionalmente un bloque de sentencias dependiendo del valor de una expresión de tipo entero o carácter. La sintaxis de la construcción **case** es la siguiente:

```
select case (expresión)
        case (rango1)
                sentencias1
        case (rango2)
                sentencias2
        case (rango3)
                sentencias3
        .....
        case default
                sentenciasN
end select
```

  donde *rango1*, *rango2*, ... son expresiones de tipo entero o carácter que representan un rango de valores de la forma:

  exp–inferior [: *exp–superior*], ...

  Cuando el flujo llega a esta construcción, se ejecutan las sentencias que pertenezcan al rango de valores dentro del cual se encuentra el resultado de *expresión*.

  En el siguiente ejemplo se hace uso de una construcción **case** para construir un menú de procedimientos.

```
write (*,*) ' Introduzca su opción: 1, 2 , 3 '
read  (*,*)  i

select case (i)

         case (1)
                  call procedimiento1

         case (2)
                  call procedimiento2

         case (3)
                  call procedimiento3

         case default

                  write(*,*) ' Procedimiento no implementado '

end select
```

- Parada **stop**

  Interrumpe el flujo del programa y la ejecución se acaba devolviendo el control al sistema operativo. La sentencia **stop** se suele usar para abortar la ejecución por causas anormales o inesperadas.

4.3. Operaciones de entrada y salida

El objeto principal de las operaciones de entrada y salida es transferir o recibir datos desde un medio externo. La lectura de datos de medios externos tales como una cinta, disco magnético u óptico o un teclado es una operación de entrada y la escritura en medios tales como una cinta, disco magnético, óptico o monitor es una operación de salida. Cuando escribimos o leemos desde un dispositivo externo, el acceso es por defecto secuencial. Si queremos leer o escribir una determinada información, debemos recorrer todo el contenido de la información. El proceso es el mismo que deberíamos realizar cuando uno trata de buscar una pieza musical en una cinta de cassette en un equipo de música.

Lo primero que tenemos que hacer para acceder al dispositivo externo es abrirlo. Esta operación se realiza mediante la sentencia **open** cuya sintaxis se detalla a continuación:

open (*especificación–dispositivo*)

donde *especificación del dispositivo* es una lista formada principalmente por:

- [**unit** =] *unidad*
- [**file** =] '*nombre*'
- [**status** = '**new**', '**old**', '**unknown**', '**replace**', '**scratch**']
- [**form** = '**formatted**', '**unformatted**']
- [**err** = *etiqueta*]

Esta instrucción abre o asocia un número entero *unidad* a un fichero o dispositivo externo identificado por *nombre*. Este fichero puede tener existencia previa ('**old**') o ser de nueva creación ('**new**'). En las circunstancias donde queramos que el fichero tenga una existencia temporal, la especificación '**scratch**' eliminará el fichero al final de la ejecución del programa. Si existe algún error al intentar abrir un fichero, la sentencia cede el control de flujo a la sentencia *etiqueta*. El fichero puede estar formado por un conjunto de caracteres ASCII ('**formatted**') o por un conjunto de datos en su representación binaria ('**unformatted**'). Cuando el volumen de datos del fichero es muy grande, la escritura de datos sin formato tiene la ventaja de ahorrar mucho espacio. Un número real con doble precisión en su representación binaria ocupa 8 bytes mientras que en formato ASCII ocupa aproximadamente el triple. Una vez que hemos abierto o creado un fichero podemos escribir o leer haciendo referencia al número de la unidad mediante las sentencias **write** y **read**.

write (*especificación-entrada–salida*) [*objetos–salida*]

read (*especificación-entrada–salida*) [*objetos–entrada*]

La especificación de entrada y salida constituye una lista formada por:

- [**unit** =] *unidad*
- [**err** = *etiqueta1*]
- [**end** = *etiqueta2*]
- [**fmt** =] '(*lista-formatos*)'

La sentencia **write** escribe las variables de la lista de objetos de salida en un dispositivo externo como puede ser un fichero del disco del ordenador. La sentencia **read** lee desde el dispositivo externo los objetos de entrada que estarán caracterizados por una lista de nombres de variables. El proceso de lectura o escritura se realiza siempre que no se detecte un final de fichero (**end**) o un error (**err**) en cuyo caso el control del flujo se cede a las etiquetas prefijadas (*etiqueta1, etiqueta2*). La lista de formatos está formada por los siguientes componentes dependiendo del tipo de dato que queramos leer o escribir:

- Variable en formato hexadecimal: `Zw`
- Variable entera: `Iw`
- Variable real en formato coma flotante: `Fw.d`
- Variable real en notación exponencial: `Ew.d`
- Variable lógica: `Lw`
- Variable tipo carácter: `Aw`

donde `w` es el número total de dígitos que se reserva para la representación de la variable. Mientras que en el formato en coma flotante `d` representa el número de decimales, en el formato de notación exponencial representa los dígitos reservados para la mantisa. El formato libre se representa por * e indica que la variable se escribe o se lee de acuerdo a la declaración de tipo que posea.

Existen algunas sentencias adicionales como la sentencia:

rewind (**unit** = *unidad*)

que permite rebobinar la unidad al principio del fichero o unidad, la sentencia:

backspace (**unit** = *unidad*)

que permite posicionar la lectura o la escritura en la línea anterior, y por último, la sentencia:

close (**unit** = *unidad*)

que desconecta o cierra el dispositivo referenciado por el entero *unidad*.

Pasamos a realizar una serie de ejercicios prácticos sobre las operaciones de entrada y salida.

1. Crear un fichero que se llame `matriz.dat`. El fichero tiene que ser editable por cualquier editor de textos.

```
open (UNIT=3, FILE='matriz.dat', STATUS='new', FORM='formatted')
```

2. Escribir por pantalla tres números reales con tres decimales.

```
write (*, '(3F8.3)' ) x, y, z
```

3. Presentar en la pantalla un dato entero con diferentes formatos.

```
integer :: n = -34787

      write (*, '(i5)' )  n
      write (*, '(i6)' )  n
      write (*, '(i10)')  n
```

   Los resultados en pantalla son los siguientes:

```
*****
-34787
    -34787
```

4. Escribir un dato real en doble precisión en distintos formatos en coma flotante.

```
 real (8) :: pi
 pi = -4e0*atan(1e0)
    write (*,  *         )  pi
    write (*, '(f2.1)'   )  pi
    write (*, '(f10.6)'  )  pi
    write (*, '(f19.15)' )  pi
```

   La correspondiente salida en pantalla es la siguiente:

```
  -3.14159265358979
 **
  -3.141593
  -3.141592653589793
```

5. Escribir un dato real en doble precisión con distintos formatos exponenciales.

```
real (8) :: pi
pi = -4e0*atan(1e0)
   write (*, '(e2.1)'   )  pi
   write (*, '(e10.6)'  )  pi
   write (*, '(e14.8)'  )  pi
   write (*, '(e25.15)' )  pi
```

   La correspondiente salida en pantalla es la siguiente:

```
**
**********
-.31415927E+01
-0.314159265358979E+01
```

6. Escribir un dato tipo carácter con distintos formatos.

```
character (20) :: nombre
nombre = 'programacion'

       write (*,  *      )  nombre
       write (*, '(a5)'  )  nombre
       write (*, '(a20)' )  nombre
```

   La correspondiente salida en pantalla es la siguiente:

```
 programacion
progr
programacion
```

7. Escribir un programa que pregunte por el valor de x, que calcule el $sen(x)$ y que lo imprima en la pantalla.

```
write (*,*) 'Introducir el valor de x ='
read (*,*) x
y = sin ( x )
write (*,*) 'El valor del seno de x: ', x, 'vale: ', y
```

8. Leer de un fichero de datos llamado `vector.dat`, ya existente, 100 variables de doble precisión. Considerar que los elementos del fichero han sido grabados sin formato.

```
real  v(100)

open (UNIT=3, FILE='vector.dat', STATUS='old', FORM='unformatted')
read(3) v(1:100)
```

En ciertas ocasiones en las que se quiera leer o escribir de una vez todo un conjunto de objetos se puede utilizar de sentencia **namelist** con la siguiente sintaxis:

namelist / *nombre* / *lista de variables*

La sentencia **namelist** agrupa *lista de variables* bajo la etiqueta *nombre*, permitiendo la escritura y la lectura de todas las variables que agrupe de una forma compacta. En el siguiente ejemplo se utiliza para dar los parámetros de inicialización específicos a un determinado programa ya compilado, mediante un fichero de inicialización.

```
program simula

            real  :: tf
            integer :: n
            namelist / inicia / tf,  n

             open (3, file='parametros.ini')
             read (3, inicia)

             call integra(n, tf)

end program
```

En este ejemplo se lee desde un fichero llamado `parametros.ini` los datos `tf` y `n`. El fichero `parametros.ini` es un fichero ASCII que se puede modificar con cualquier editor de textos y debe tener el siguiente formato:

```
&inicia

             tf  =  13.0     ! tiempo total de integracion
             n   =  200      ! numero de pasos

/
```

De esta forma, los ficheros de inicialización pueden modificarse para correr diferentes casos sin necesidad de modificar el programa.

4.4. Funciones intrínsecas

Los compiladores FORTRAN disponen de bibliotecas de *funciones intrínsecas* para facilitar la tarea del programador. Algunas de las funciones intrínsecas más importantes clasificadas por su funcionalidad son:

- Funciones elementales.

Función nombre	Definición	Tipo argumento	Tipo función
abs	valor absoluto	int, real, cmplx	int, real
exp	exponencial	real, complex	real, complex
log	log neperiano	real, complex	real, complex
log10	log en base 10	real	real
sqrt	raíz cuadrada	real, complex	real, complex
max (a, b,...)	máximo de arg.	int, real	int, real
min (a, b,...)	mínimo de arg.	int, real	int, real

- Funciones de variable compleja.

Función nombre	Definición	Tipo argumento	Tipo función
imag	parte imaginaria	complex	real
real	parte real	complex	real
conjg	conjugado	complex	complex

- Funciones trigonométricas.

Función nombre	Definición	Tipo argumento	Tipo función
sin	seno	real, complex	real, complex
cos	coseno	real, complex	real, complex
tan	tangente	real, complex	real, complex
asin	arco seno	real	real
acos	arco coseno	real	real
atan	arco tangente	real	real
atan2	arco tangente a/b	real	real
cotan	cotangente	real	real

- Funciones hiperbólicas.

Función nombre	Definición	Tipo argumento	Tipo función
sinh	seno	real	real
cosh	coseno	real	real
tanh	tangente	real	real

- Conversión de identificador de tipo.

Función nombre	Definición	Tipo argumento	Tipo función
int	conversión	int, real, cmplx	int
int2	conversión	int, real, cmplx	integer(2)
real	conversión	int, real, cmplx	real(4)
float	conversión	integer	real(4)
cmplx	conversión	int, real, cmplx	complex
char	conversión	integer	character
sngl	conversión	real(8)	real(4)
dble	conversión	int, real, cmplx	real(8)

Debido a que la misma función puede ser llamada con variables de diferentes tipos, el compilador reconoce el tipo de la variable de entrada y devuelve el resultado de la función con el mismo tipo de la variable de entrada. Para el cálculo de estas funciones intrínsecas el compilador utiliza internamente desarrollos en serie. Por lo tanto, en un mismo ordenador y con el mismo programa dos compiladores pueden dar tiempos de ejecución totalmente diferentes dependiendo de la velocidad de convergencia de sus funciones intrínsecas.

4.5. Operaciones con matrices

Existen tres formas de hacer asignaciones y operaciones con matrices:

1. Mediante los nombres de matrices con la misma dimensión tratadas como un conjunto de elementos.
2. Mediante los elementos de las matrices utilizando índices.
3. Mediante sectores de las matrices con la misma dimensión.

Para cualquiera de estas tres formas, se pueden aplicar todos los operadores matemáticos que se usan para variables escalares. A continuación se muestran ejemplos para estas tres formas de operaciones con matrices. En el primer ejemplo se opera con tres matrices identificadas por su nombre `A`, `B` y `C`.

```
program operaciones_1

  real  ::  A(4,5), B(4,5), C(4,5)

    ! *** inicializacion de A
          A(1,:) =   (/ 8e0, 7e0, 4e0, 9e0, 1e0 /)
          A(2,:) =   (/ 5e0, 4e0, 2e0, 3e0, 7e0 /)
          A(3,:) =   (/ 2e0, 3e0, 9e0, 1e0, 6e0 /)
          A(4,:) =   (/ 7e0, 5e0, 8e0, 2e0, 4e0 /)

    ! *** se asigna a todos los elementos de B el valor 3
          B = 3e0

    ! *** se asigna a todos los elementos de C el valor 0
          C = 0e0

    ! *** operaciones  escalares para todos los elementos

          ! *** operaciones definidas matematicamente
                C = A + B;    C = 5 * A + 2 * B

          ! *** operaciones escalares que no coinciden
          !     con la operacion matricial
                C = A*B;  C = A/B;   C = exp(A); C = A**B;
                C = B**3; C = sin(A) - cos(A);  C = sqrt(B*A);

end program Operaciones_1
```

Estas operaciones se realizan mediante el procesamiento vectorial propio del FORTRAN 95, siendo ésta la implementación óptima. En las primeras líneas se ini-

cializan todos los elementos de la matrices `A`, `B` y `C`. En concreto, la matriz `A` se inicializa mediante la construcción de vectores filas. Las siguientes líneas constituyen operaciones entre matrices como son la suma, el producto de un escalar por una matriz, el producto de dos matrices, la exponencial de una matriz, etc. Estas operaciones entre matrices representan un bucle implícito en los dos índices de las matrices que operan de forma ordinaria uno por uno todos los elementos de las matrices. El producto `A * B` en FORTRAN 95 no coincide con la definición matemática del producto de dos matrices que en FORTRAN 95 es **matmul** como veremos más adelante. De igual forma la potencia `A**3` no coincide con la potencia matemática de una matriz $\mathbf{A}^3$. La exponencial de una matriz es otro ejemplo de este funcionamiento anómalo del FORTRAN 95.

En el segundo ejemplo se operan los elementos de las matrices a través de sus índices. Las asignaciones y operaciones se realizan indicando el *nombre* de la matriz y los índices para cada dimensión. Estos índices apuntan al elemento escalar con el que se quiere operar. Su sintaxis es la siguiente:

nombre (*índice*$_1$, *índice*$_2$, ...)

```
program operaciones_2

  integer ::  i, j
  real :: A(4,5), B(4,5), C(4,5)

       ! *** inicialización de las matrices A y B  :
             do i=1, 4
                     do j=1, 5
                             A(i,j) = 3e0; B(i,j) = 3e0 * i + j;
                     end do
             end do

       ! *** equivalente a C = A / B en FORTRAN 95.
             do i=1, 4
                     do j=1, 5
                             C(i,j) = A(i,j) / B(i,j)
                     end do
             end do

       ! *** suma de elementos de diferente posición
             C (2,2) = A (1,4) + B (3,5)

end program Operaciones_2
```

En las primeras líneas se asigna la constante 3 a los elementos de la matriz **A** y una expresión a los elementos de la matriz **B**. En las siguientes líneas se divi-

den los elementos de la matriz **A** por los elementos correspondientes de la matriz **B** y el resultado se asigna a **C**. Este bucle es equivalente a ejecutar la sentencia $\mathbf{C} = \mathbf{A}/\mathbf{B}$ del ejercicio anterior. Finalmente, en la última línea se suma el elemento escalar $\mathbf{A}_{1,4}$ con el elemento escalar $\mathbf{B}_{3,5}$ y el resultado se asigna al elemento escalar $\mathbf{C}_{2,2}$.

En el tercer ejemplo se operan sectores de matrices. Las asignaciones y operaciones se realizan indicando el *nombre* de la matriz y para cada dimensión se especifican los valores inicial y final del sector de matriz, pudiéndose indicar opcionalmente un incremento. Si se omiten los valores iniciales o finales, se consideran los valores correspondientes a la declaración de la variable. Su sintaxis es la siguiente:

nombre ([*inicio*$_1$] : [*fín*$_1$] [: *incremento*$_1$], [*inicio*$_2$] : [*fín*$_2$] [: *incremento*$_2$], ...)

```
program operaciones_3

   integer ::  i, j
   real :: A(4,5), B(4,5), C(4,5)

        ! *** inicializacion de A
             A(1,:) =   (/ 8e0, 7e0, 4e0, 9e0, 1e0 /)
             A(2,:) =   (/ 5e0, 4e0, 2e0, 3e0, 7e0 /)
             A(3,:) =   (/ 2e0, 3e0, 9e0, 1e0, 6e0 /)
             A(4,:) =   (/ 7e0, 5e0, 8e0, 2e0, 4e0 /)

        ! *** inicializacion de B y C
             B = 3e0;   C = 0e0

        ! *** linea 1
             C (1,:) = 2e0

        ! *** linea 2
             C (3,:) = A (1,:) + B (4,:)

        ! *** linea 3
             C (1:3, 4)  = A (1, 1:5:2) + A (3, 1:3)

        ! *** linea 4
             B (2:4, 1:5:3) = 5e-1


 end program Operaciones_3
```

La línea 1 del ejemplo asigna a todos los elementos de la fila 1 la constante 2:

$$\mathbf{C} = \begin{pmatrix} \boxed{2} & \boxed{2} & \boxed{2} & \boxed{2} & \boxed{2} \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

La línea 2 suma todos los elementos de la fila 1 de **A** con los correspondientes de la fila 4 de **B** y se asignan a todos los elementos de la fila 3 de **C**:

$$\begin{pmatrix} 2 & 2 & 2 & 2 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ \boxed{11} & \boxed{10} & \boxed{7} & \boxed{12} & \boxed{4} \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} =$$

$$= \begin{pmatrix} \boxed{8} & \boxed{7} & \boxed{4} & \boxed{9} & \boxed{1} \\ 5 & 4 & 2 & 3 & 7 \\ 2 & 3 & 9 & 1 & 6 \\ 7 & 5 & 8 & 2 & 4 \end{pmatrix} + \begin{pmatrix} 3 & 3 & 3 & 3 & 3 \\ 3 & 3 & 3 & 3 & 3 \\ 3 & 3 & 3 & 3 & 3 \\ \boxed{3} & \boxed{3} & \boxed{3} & \boxed{3} & \boxed{3} \end{pmatrix}$$

La línea 3 suma los elementos de la fila 1 entre las columnas 1 y 5 con incrementos de 2 de **A** con los elementos de la fila 3 entre las columnas 1 y 3 con incrementos de 1 de **A** y se asignan a los elementos de la columna 4 entre las filas 1 y 3 con incrementos de 1 de **C**:

$$\begin{pmatrix} 2 & 2 & 2 & \boxed{10} & 2 \\ 0 & 0 & 0 & \boxed{7} & 0 \\ 11 & 10 & 7 & \boxed{10} & 4 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} =$$

$$= \begin{pmatrix} \boxed{8} & 7 & \boxed{4} & 9 & \boxed{1} \\ 5 & 4 & 2 & 3 & 7 \\ 2 & 3 & 9 & 1 & 6 \\ 7 & 5 & 8 & 2 & 4 \end{pmatrix} + \begin{pmatrix} 8 & 7 & 4 & 9 & 1 \\ 5 & 4 & 2 & 3 & 7 \\ \boxed{2} & \boxed{3} & \boxed{9} & 1 & 6 \\ 7 & 5 & 8 & 2 & 4 \end{pmatrix}$$

Finalmente, la línea 4 asigna la constante 0,5 a los elementos comprendidos entre las filas 2 y 4 con incrementos de 1 y las columnas 1 a 5 con incrementos de 3 de **B**:

$$\mathbf{B} = \begin{pmatrix} 3 & 3 & 3 & 3 & 3 \\ \boxed{0.5} & 3 & 3 & \boxed{0.5} & 3 \\ \boxed{0.5} & 3 & 3 & \boxed{0.5} & 3 \\ \boxed{0.5} & 3 & 3 & \boxed{0.5} & 3 \end{pmatrix}$$

Para operar con sectores de matrices, estos sectores deben tener igual número de dimensiones e igual número de elementos por dimensión.

Para facilitar y optimizar las operaciones con matrices, el FORTRAN 95 proporciona las siguientes funciones intrínsecas:

- **dot_product**: Esta función calcula el producto escalar de dos vectores y su sintaxis es la siguiente:

  resultado = **dot_product** (*vector* a, *vector b*)

  Las entradas de esta función son los argumentos *vector a* y *vector b*, y la salida es un escalar en el nombre de la función **dot_product**. Los argumentos *vector a* y *vector b* deben ser vectores de igual tamaño, y pueden tener identificador de tipo **integer**, **real** o **complex**. Si los identificadores de tipo son **integer** o **real** el resultado es la suma de los productos de sus componentes correspondientes. En cambio, si los identificadores de tipo son **complex** el resultado es la suma de los productos de las componentes conjugadas de *vector a* por las componentes correspondientes de *vector b*. En cualquier caso, **dot_ product** representa el producto escalar en un espacio vectorial euclídeo de elementos reales o complejos, es decir:

  $$\textbf{dot\_product}\ (\textit{vector a}, \textit{vector b}) = \textit{vector a}^* \ \cdot \ \textit{vector b}\ ,$$

  donde *vector* a^* representa el vector conjugado transpuesto de *vector a*.

  El siguiente ejemplo calcula el producto escalar de dos vectores de componentes reales.

```
program producto_vectores_1

        integer, parameter :: n = 3
        real :: a(n), b(n)
              a = (/ 1., 2., 3. /)
              b = (/ 2., 5., 4. /)
              write (*,*) dot_product (a, b)

end program producto_vectores_1
```

Al ejecutar este ejemplo, el resultado es: `24.00000`.

El ejemplo anterior es equivalente al siguiente programa donde, para obtener el producto escalar, se realizan operaciones escalares sobre las componentes de los vectores:

```
program producto_vectores_2

        integer, parameter :: n=3
        real  :: a(n), b(n), s
              a = (/ 1., 2., 3. /)
              b = (/ 2., 5., 4. /)
              s = a(1)*b(1) + a(2)*b(2) + a(3)*b(3)
              write (*,*) s

end program producto_vectores_2
```

Es importante destacar que el resultado de esta función intrínseca no es equivalente a realizar `a*b`, que correspondería al producto de las componentes correspondientes y que se asignaría a otro vector de igual tamaño.

```
program producto_vectores_3

    integer, parameter :: n=3
    real :: a(n), b(n), c(n)

     a = (/ 1., 2., 3. /)
     b = (/ 2., 5., 4. /)

     !  operacion escalar
        do i = 1, n
                  c(i) = a(i)*b(i)
        end do

     !  operacion vectorial  (mismo resultado )
        c = a*b


end program producto_vectores_3
```

Finalmente, el siguiente ejemplo calcula el producto escalar de dos vectores de componentes complejas.

```
program producto_vectores_4

       integer, parameter :: n=2
       complex :: a(n), b(n)

            a = (/ (2e0, 3e0), (1e0, 5e0) /)
            b = (/ (6e0,-9e0), (2e0, 4e0) /)

            write (*,*) dot_product (a, b)

end program producto_vectores_4
```

cuyo resultado es:

```
(7.00000000000000,-42.0000000000000).
```

- **matmul**: Esta función calcula el producto de dos matrices *matriz a* y *matriz b* de elementos enteros, reales o complejos y su sintaxis es la siguiente:

resultado = **matmul** (*matriz* a, *matriz b*)

- **transpose**: Esta función calcula la matriz traspuesta de una matriz dada y su sintaxis es la siguiente:

resultado = **transpose** (*matriz*)

- **shape**: Esta función calcula la forma de una matriz y su sintaxis es la siguiente:

resultado = **shape** (*matriz*)

  El *resultado* es un vector de enteros de tantas componentes como dimensiones tenga la variable *matriz*. El valor de cada componente es la extensión o el tamaño de cada índice de *matriz*. Si *matriz* tiene dos índices, las componentes de *resultado* son las filas y las columnas de *matriz*.

- **reshape**: Esta función construye una matriz con una forma diferente de la forma de la matriz dada. Su sintaxis es la siguiente:

resultado = **reshape** (*matriz*, *nueva forma* [, **order**=*orden*])

  donde *nueva forma* es un vector de enteros de hasta 7 componentes que define la forma de la matriz *resultado*. El valor de cada componente de *nueva forma* es la extensión o el tamaño de cada índice del nuevo *resultado*. El siguiente argumento de la función (*orden*) es opcional y es un vector de enteros con el mismo número de componentes que *nueva forma* y que indica en qué secuencia se rellena *resultado* a partir de los datos *matriz*.

  En el siguiente ejemplo se transforma un vector en una matriz. Mediante el vector *orden* podemos rellenar la matriz resultado por filas o por columnas.

```
program transforma

    real :: v(6), A(3,2)

    v = (/ 5., 3., 0., 2., 4.,  6. /)

    A = reshape(v, (/3,2/), ORDER=(/2,1/) ) ! rellena por filas

    do i=1,3;  write(*,*) A(i,1:2);  end do ! 5.  3.
    write(*,*)                              ! 0.  2.
                                            ! 4.  6.

    A = reshape(v, (/3,2/), ORDER=(/1,2/) ) ! rellena por columnas

    do i=1,3; write(*,*) A(i,1:2); end do   ! 5.  2.
    write(*,*)                              ! 3.  4.
                                            ! 0.  6.
end program
```

- *operador*: La sintaxis de esta función intrínseca es la siguiente:

operador (*matriz* [, [**dim**=] *índice*] [, [**mask**=] *expr–lógica*])

donde el argumento de entrada es una *matriz* y *operador* puede ser la palabra clave **sum, product, maxloc, maxval, minloc, minval** que se corresponde con las operaciones suma, producto, posición del máximo, valor máximo, posición del mínimo y valor mínimo respectivamente. El argumento **dim** es el escalar entero *índice* que puede valer entre 1 y 7 (número máximo de índices de una matriz). Este argumento, que es opcional, indica el índice sobre el cual se realiza la operación (sumatorio, producto, búsqueda del máximo o mínimo). Por ejemplo, si `A` es una matriz de tres índices, el valor máximo con **dim** igual a 2 es una matriz de dos índices, que contiene los valores máximos de `A` cuando variamos su segundo índice. La operación sólo se realiza sobre los elementos que cumplen la *expresión lógica* representada en **mask.** En general, el resultado de la función intrínseca es una matriz con una dimensión menor que la matriz argumento de entrada. En concreto, si la matriz es de dos dimensiones, el operador es **sum** y **dim**=1, entonces el resultado es un vector cuyas componentes representan el sumatorio de cada una de las columnas de la matriz argumento de entrada. Si el argumento **dim** no está presente, el resultado es un escalar que representa la suma de todos los elementos de la matriz. Para los operadores **sum** y **product** los elementos de la matriz pueden ser enteros, reales o complejos y para **maxval, maxloc, minval, minloc** los elementos sólo pueden ser enteros o reales.

En el siguiente ejemplo se pone de manifiesto el uso de la función **maxloc** con máscara para buscar e interpolar en una tabla unidimensional.

```
program interpola

    integer, parameter :: n = 6
    real :: x(n), y(n), x0, y0
    integer ::  i(1), j


     x = (/ -4., -2., 0., 2., 4.,  6. /)
     y = (/ -0.202, -0.050, 0.108, 0.264, 0.421, 0.573 /)


     write(*,*) ' Interpolacion para  x ='
     read(*,*)  x0

     i = maxloc(x, x < x0);   j = i(1)

     y0 = y(j) + ( x0-x(j) )/( x(j+1)-x(j) )*( y(j+1) - y(j) )

     write(*,*) ' Valor interpolado y0 = ', y0

end program interpola
```

- **forall**: Asignación a elementos de una matriz que verifiquen una determinada condición.

  forall (*rango* [, *expresión-lógica*])

  asignación–matrices

  [*construcciones forall*]

  [**end forall**]

  donde *rango* es el rango de valores de los índices de la matriz que se recorren para asignar valores a la matriz y *expresión lógica* es la condición que se debe verificar para que se realice la asignación.

En los siguientes ejemplos se combinan operaciones vectoriales y escalares con funciones intrínsecas.

1. Funciones intrínsecas aplicadas a toda la matriz o vector.

```
      program Operaciones_4

      integer, parameter:: n=4
      real :: A(n,n), U(n), AxU(n), U_A(n,n)
      real :: trA, At(n,n), diagonal(n)
      real :: U(n) = (/3,    12,     5,    7 /)


!     *** inicializa la matriz A
          A(1,1:n) = (/   5.,   0.2,     4.,   8.   /)
          A(2,1:n) = (/  -7.,   -6.,    -9.,   42.  /)
          A(3,1:n) = (/   4.,    5.,     7., -0.1   /)
          A(n,1:n) = (/  -7.,    42.,    9.,   12.  /)


!     *** producto de matriz por vector
          AxU =  matmul(A, U)

          write(*,*) ' AxU = ', AxU
          write(*,*)

!     *** carga los elementos positivos
          U_A = 0.0
          forall (i=1:n, j=1:n, A(i,j)>0) U_A(i,j) = A(i,j)

          write(*,*) ' U_A = '
          do i=1,n;   write(*,*)  U_A(i,1:n);  end do
```

```
!      *** traza de matriz
           forall (i=1:n) diagonal(i) = A(i,i)
           trA = sum(diagonal)

           write(*,*)
           write(*,*) ' Traza de A = ', trA

!      *** matriz traspuesta
           At = transpose (A)

         end
```

2. Funciones intrínsecas sobre sectores de matrices o vectores.

```
         program Operaciones_5

         integer, parameter:: n=4, m=6
         real :: A(n,m)


          A(1,1:m) = (/  0.7,   8.0,   -4.0,   0.1,   0.2, -5.0 /)
          A(2,1:m) = (/  7.0,   1.0,    4.0,  -22.0, 12.0,  6.0 /)
          A(3,1:m) = (/  5.0,  -2.0,    1.0,   13.0,  8.5,  6.0 /)
          A(4,1:m) = (/  9.0,  13.0,  -22.0,    8.0,  3.0,  5.0 /)

!       *** producto escalar de la columna 1 con la 2
           write(*,*) dot_product( A(1:n,1), A(1:n,2) )

!       *** producto escalar de la fila 2 con la 4
           write(*,*) dot_product( A(2,1:m), A(4,1:m) )

         end
```

Capítulo 5

Especificaciones

La parte de especificación de variables es aquella en la que se detallan los tipos, atributos y tamaño de las variables. La especificación de un procedimiento describe el interfaz de llamada del procedimiento. En este interfaz se detallan los tipos, atributos y tamaño de los argumentos.

Las variables de un programa se pueden clasificar en: variables locales y variables externas. Las variables locales son las variables que están declaradas de forma local a un procedimiento y son invisibles para el resto de las unidades de programa y procedimientos, por lo que su uso queda restringido al procedimiento considerado. Las variables externas son variables que, habiendo sido especificadas en unidades de programa **module**, se pueden ver en las unidades de programa que se asocien a esta unidad mediante la sentencia **use**, o en los procedimientos contenidos en la unidad de programa donde hayan sido especificadas. En consecuencia, las unidades de programa o procedimientos que compartan variables externas podrán leer y escribir sobre éstas de una forma indiscriminada.

5.1. Sentencia use

Una variable declarada en un módulo determinado se puede ver por otra unidad de programa o convertir en variable externa mediante la sentencia **use**. La sintaxis de una llamada a una unidad **module** declarada con *nombre* es la siguiente:

use *nombre–modulo* [, **only** : [*nuevo–nombre* =>] *nombre, ...*]

Cuando se especifica **only** solo las variables o procedimientos indicados serán compartidas para la unidad de programa que hace la asociación. En el siguiente ejemplo se crea una unidad de programa **module** de nombre `fisico` que contiene las variables reales x, y, z y se accede a él mediante la sentencia **use**.

```
module fisico
                  real ::    x=1, y=2, z=3
end module fisico

program asociacion
                       use fisico, only : w => x, y
                       write(*,*) w, y
end program asociacion
```

5.2. Identificador de tipo

Todas estas especificaciones tienen una componente común que es el identificador de tipo que pasamos a describir. De la misma forma que en el lenguaje matemático se trabaja con números enteros, reales y complejos, en programación se puede operar con variables de distinto tipo. El *identificador de tipo* especifica el tipo de dato que puede almacenar una variable. Los identificadores de tipo que existen en FORTRAN son los siguientes:

1. **integer** [([**kind** =] *tamaño*)] especifica que la variable es de tipo entero. Un valor entero tiene una representación exacta de su valor en el ordenador. La palabra clave **kind** indica la cantidad de memoria necesaria para guardar la variable entera y, en consecuencia, el número máximo y mínimo que puede albergar la variable entera. Es importante indicar que este elemento depende fundamentalmente de cada procesador y que las declaraciones dadas a continuación son orientativas.

   ```
   integer (kind=1) :: i;     integer (kind=2) :: j;
   integer (kind=4) :: k;     integer (kind=8) :: l
   ```

   La variable entera `i` ocupa 1 byte de memoria y en ella se puede guardar datos enteros comprendidos entre - 128 y 127. La variable `j` ocupa 2 bytes de memoria y puede albergar datos enteros comprendidos entre - 32 768 y 32 767. La variable `k` ocupa 4 bytes de memoria y su rango de valores está comprendido entre - 2 147 483 648 y 2 147 483 647. La variable `l` ocupa 8 bytes de memoria. Este identificador de clase no es habitual en ordenadores personales pero sí está presente en estaciones de trabajo y ordenadores de gran potencia de cálculo. Para asegurar la portabilidad del programa desde el punto de vista de su rango, se puede definir su tamaño mediante la

función intrínseca **selected_int_kind**. Si no se especifica el identificador de clase, el compilador asume [**kind=**] 4.

2. **real** [([**kind** =] *tamaño*)] especifica que la variable es de tipo real. Un valor real no tiene una representación exacta en el ordenador y el error que se comete cuando se introduce en el ordenador un número real se denomina *pérdida de precisión* o *round-off*. A continuación se pone un ejemplo de declaración de variables reales.

```
real  (kind=4) :: a;     real (8) :: b;     real (16) :: c;
```

   La variable `a` ocupa 4 bytes de memoria. El número más pequeño que se puede representar es 1,18 `x` 10^{-38}. El número más grande que se puede representar es 3,40 `x` 10^{38}. La precisión en coma flotante es 1,19 `x` 10^{-7}. La variable `b` ocupa 8 bytes de memoria. El número más pequeño que se puede representar 2,23 `x` 10^{-308}. El número más grande que se puede representar 1,79 `x` 10^{308}. La precisión en coma flotante es 2,22 `x` 10^{-16}. La variable `c` ocupa 16 bytes de memoria. Este identificador de clase no es habitual en ordenadores personales pero sí está presente en estaciones de trabajo y ordenadores de gran potencia de cálculo. La precisión puede ser variable y se define usando la función intrínseca **selected_real_kind**. Esta opción asegura la portabilidad del programa desde el punto de vista de su precisión de variables reales. Si no se especifica el identificador de clase, el compilador asume [**kind=**] 4.

3. **complex** [([**kind** =] *tamaño*)] especifica que la variable es de tipo complejo con parte real y parte imaginaria y también en este caso no tiene una representación exacta en el ordenador. Para este tipo de dato se especifican los mismos identificadores de clase que se definen para variables tipo **real**, y se aplican tanto a la parte real como a la parte imaginaria. Por lo tanto, una variable de tipo **complex** con identificador de clase 8, ocupará 16 bytes de memoria, 8 bytes para cada parte real e imaginaria. Si no se especifica el identificador de clase, el compilador asume [**kind=**] 4. Cuando en un programa, todas las variables reales y complejas se declaran con identificador de clase [**kind=**] 4, se dice que está programado en *simple precisión*. Por el contrario, si todas las variables reales y complejas se declaran con identificador de clase [**kind=**] 8, se dice que está programado en *doble precisión*.

4. **character** [([**len** =] *tamaño*)] especifica que la variable es del tipo carácter. Los caracteres posibles son los caracteres imprimibles ASCII. En variables tipo carácter, el identificador de clase no especifica precisión sino que define la longitud de la cadena de caracteres. En este caso, se reemplaza la palabra clave **kind** con la palabra clave **len**. La declaración:

```
character  (len=10)  :: nombre
```

especifica que la variable `nombre` esta formada por 10 caracteres ASCII.

5. **logical** [([**kind** =] *tamaño*)] especifica que la variable es del tipo lógico. Una variable lógica tiene dos valores posibles: verdadero (`.true.`) y falso (`.false.`). Si no se especifica el identificador de clase, el compilador asume [**kind=**] 4 y la variable ocupa 4 bytes de memoria.

6. **type** (*nombre tipo*) especifica una estructura de datos que previamente ha sido definida. La definición de nuevos tipos se describe en la sección 5.5.

Es importante hacer notar que las operaciones entre datos son función del tipo de dato. Cuando una constante está representada por caracteres numéricos sin puntos, el compilador asume que es un dato de tipo entero. Si escribimos en FORTRAN `x=3/4`, dividimos dos números enteros, y el resultado es un número entero, que se asigna a la variable `x`. Por tanto, el valor de `x` será 0 (parte entera de 3/4). Para evitar que ocurran asignaciones indeseadas, se puede declarar explícitamente que el 3 y el 4 son variables reales escribiendo `3.0` y `4.0` o `3e0` y `4e0`. De esta forma, el valor de `y` es 0.75. Para especificar que el 3 y el 4 son variables reales con doble precisión, tenemos que escribir `3d0` y `4d0`.

Intentar ejecutar el siguiente programa en el ordenador y explicar los resultados obtenidos.

```
program reals    !     perdida de precision

    write(*,*) ' *** Diferentes resultados al calcular 1.1/2 '
    write(*, '(a20, f17.15)') ' 1.1   / 2.          ',  1.1/2.
    write(*, '(a20, f17.15)') ' 1.1   / 2           ',  1/2
    write(*, '(a20, f17.15)') ' 1.1   / 2d0         ',  1.1/2d0
    write(*, '(a20, f17.15)') ' 1.1d0 / 2d0         ',  1.1d0/2d0


    write(*,*) ' *** Diferentes resultados al calcular 1/3  '
    write(*,'(a20, f17.15)') ' 1   / 3d0          ',  1/3d0
    write(*,'(a20, f17.15)') ' 1.  / 3d0          ',  1./3e0
    write(*,'(a20, f17.15)') ' 1d0 / 3d0          ',  1d0/3d0
    write(*,'(a20, f17.15)') ' 1d0 / 3.           ',  1d0/3.
    write(*,'(a20, f17.15)') ' 1.  / 3.           ',  1./3.
    write(*,'(a20, f17.15)') ' 1   / 3            ',  1/3
    write(*,'(a20, f17.15)') ' 1.  / 3            ',  1./3

end program reals
```

Debido a la precisión finita del ordenador, todos los números reales no pueden estar representados de forma exacta. De esta forma, un dato real se guarda como

el número más próximo (en precisión doble o simple) al dato que en realidad queremos guardar. En consecuencia, el dato se introduce con un error de redondeo que puede tener consecuencias no deseadas. En el siguiente ejemplo se calculan mediante funciones intrínsecas los parámetros de precisión del ordenador. Es importante conocer dichos parámetros para poder predecir el comportamiento de los resultados numéricos como en el ejemplo anterior.

```
program parametros_de_maquina

      real(4) :: x
      real(8) :: y


      write(*,*)  ' Valor maximo           ',   huge(x)
      write(*,*)  ' Valor minimo           ',   tiny(x)
      write(*,*)  ' Round--off             ',   epsilon(x)
      write(*,*)  ' Digitos significativos ',   precision(x)

      write(*,*)

      write(*,*)  ' Valor maximo           ',   huge(y)
      write(*,*)  ' Valor minimo           ',   tiny(y)
      write(*,*)  ' Round--off             ',   epsilon(y)
      write(*,*)  ' Digitos significativos ',   precision(y)


end parametros_de_maquina
```

5.3. Especificación de variables

La especificación de variables tiene como función reservar un espacio de memoria en cantidad y formato adecuados para permitir el trabajo de la unidad de programa o procedimiento. La cantidad de memoria se indica en unidades de bytes y el formato dependerá del tipo de variable como se verá a continuación. Una especificación de variables puede ser *estática* o *dinámica*. Es estática cuando el espacio de memoria queda determinado en *tiempo de compilación*. En este caso, se fija la memoria y el programa en *tiempo de ejecución* sólo modifica los datos de esa memoria. La declaración de variables es dinámica cuando se puede reservar memoria en tiempo de ejecución, según las necesidades del programa. La especificación de variables tiene la siguiente estructura:

id–tipo [,*atributos*] :: *nombre–objeto* [(*dimensión*)] [= *inicialización*]

A continuación se describen cada uno de los elementos indicados en la estructura anterior. Los *atributos* se escriben en una lista que comienza con una coma y finaliza con :: y especifican características de las variables relacionadas con la inicialización de datos, tipo de asignación de memoria, dimensiones de matrices y punteros. Los atributos más corrientemente utilizados son los siguientes:

1. **parameter**: Permite inicializar la variable con un dato en tiempo de compilación. El dato se especifica en la lista de variables mediante la *inicialización*. El dato de una variable declarada con este atributo no se puede modificar en tiempo de ejecución.

2. **save**: La variable especificada con este atributo guarda su valor después de abandonar el procedimiento. Al llamar nuevamente a este procedimieto, la variable tiene su valor anterior.

3. **allocatable**: La variable declarada con este atributo admite asignación dinámica de memoria.

4. **pointer**: Especifica que un objeto es un puntero o una variable dinámica. Un puntero no contiene datos pero apunta a una variable escalar o vectorial donde se almacenan los datos. La variable declarada con este atributo admite asignación dinámica de memoria.

Si no se especifican los atributos **allocatable** o **pointer**, la asignación de memoria es estática. Es decir, la especificación reserva espacio en memoria para las variables sin estos atributos. Mientras que las variables pueden tener atributos **allocatable** o **pointer**, los argumentos de un procedimiento sólo pueden tener atributo **pointer**. De esta forma, la asignación de memoria de cualquier variable con el atributo **allocatable** debe ser realizada allí donde se encuentre su especificación. Sin embargo, si la variable se especifica con atributo **pointer**, la asignación de memoria puede realizarse en cualquier otra función o subrutina.

Por último, la especificación de variables permite mediante el símbolo igual (=) inicializar la variable a la expresión *inicialización* que está formada por constantes. Mientras que una asignación a una variable dentro de un procedimiento se realiza siempre que el procedimiento sea llamado, la inicialización de la especificación de variables se produce una sola vez en tiempo de compilación. Por ejemplo, si una variable x con atributo **save** se inicializa con el valor 11 y durante la ejecución del proceso donde está especificada se hace $x = 33$, al volver a llamar a ese procedimiento el valor inicial de x es 33 y no 11.

La sintaxis de *dimensión* es la siguiente:

índice–inferior : *índice–superior*, ...

Tanto el *índice–inferior* como el *índice–superior* pueden ser constantes enteras o expresiones enteras que involucren variables con atributo **parameter**.

En los siguientes ejemplos se especifica explícitamente la forma y el tamaño de una variable.

1. Declarar x, y, z como variables reales de precisión doble y un vector v de dimensión 10 de variables enteras, cuyo primer índice sea el cero.

```
real(8) x, y, z

integer, parameter :: n=10
integer ::    v (0:n-1)
```

2. Declarar una matriz a de 10 filas y 30 columnas de números complejos de doble precisión.

```
integer, parameter ::  n=10, m=30
complex(8)  a(n, m)
```

3. Declarar un vector v de dimensión 30 cuyos elementos sean cadenas de caracteres de 10 caracteres.

```
integer, parameter :: m=30
character(10)  :: v(m)
```

Si la asignación de memoria de una variable se realiza dinámicamente, su atributo es **allocatable** o **pointer** y, entonces, el *índice–inferior* y el *índice–superior* se deben suprimir de la especificación. La petición de memoria se hará de forma dinámica mediante la sentencia:

allocate (*nombre–objeto* (*dimensión*), ...)

en donde en *dimensión* se especifica el tamaño y la forma de la variable. Una vez que hayamos utilizado la variable y ésta ya no sea necesaria, la sentencia:

deallocate (*nombre–objeto*, ...)

permitirá liberar la memoria que ocupa la variable en cuestión.

En el siguiente ejemplo se define un vector de complejos con atributo **allocatable** y una matriz cuadrada de complejos con atributo **pointer** cuya dimensión se introduce por teclado.

```
program especificacion_dinamica

     use memoria
     complex, allocatable  ::  vector(:)
     complex, pointer      ::  matriz(:, :)
     integer ::  m

        write(*,*) ' Introduzca la dimension = '
        read(*,*) m

   ! *** asignacion de memoria

   !     se debe realizar en este programa  para vector(:)
         allocate( vector(1:m) )

   !     se puede realizar en otro sitio para el puntero matriz(:,:)
         call asigna_memoria( m, matriz )

        ..........................

        deallocate(vector, matriz)

end program

module memoria

  contains

     subroutine asigna_memoria(m, A)
                  integer ::  m
                  real, pointer :: A(:,:)

        allocate(A(1:m,1:m))

     end subroutine asigna_memoria

end module memoria
```

5.4. Especificación de argumentos

La especificación de argumentos ("dummy arguments") tiene como función especificar los tipos, las dimensiones y los atributos de las etiquetas de los argumentos. Es importante hacer notar que no se trata de una especificación de variables y, por lo tanto, no se reserva memoria para los argumentos. Sólo cuando se hace una llamada a una función o una subrutina se asocian las variables con

que se llama ("actual arguments") con los argumentos de la subrutina o función ("dummy arguments"). Mientras que la especificación de variables obliga a tener un espacio físico de memoria donde albergar el objeto, la especificación de argumentos constituye la definición que deben tener los objetos en las llamadas al procedimiento en cuestión. La forma básica de especificación de una lista de argumentos tiene la siguiente estructura:

id–tipo [,*atributos*] :: *nombre–argumento* [(*dimensión*)]

especificación de interfaces

Aunque la estructura es muy similar a la de especificación de variables, existen tres diferencias esenciales: el tipo de atributos, el formato de los índices y la especificación de interfaces para argumentos que pueden ser procedimientos. Los *atributos* de argumentos más corrientemente utilizados son los siguientes:

1. **intent(in)**: Especifica que el parámetro es de entrada. Es decir, los argumentos con este atributo constituyen la información necesaria para que el procedimiento elabore su proceso.
2. **intent(out)**: Especifica que el argumento es de salida. Los argumentos con este atributo constituyen las salidas o los cálculos del procedimiento.
3. **intent(inout)**: Especifica que el parámetro puede ser de entrada y de salida. Salvo que se justifique, no es una buena metodología declarar un parámetro como de entrada y salida a la vez.
4. **pointer**: Especifica que el argumento es puntero. La descripción del atributo **pointer** ya ha sido detallada en la especificación de variables.
5. **optional**: Especifica que el parámetro es opcional. Esto quiere decir que podemos omitir parte de la funcionalidad del procedimiento restringiendo la lista de objetos en la llamada al procedimiento. La presencia del argumento opcional en la llamada al procedimiento la podemos detectar mediante la sentencia:

   result = **present** (*nombre–argumento*)

   cuyo resultado es una variable de tipo **logical** (.true. o .false.).

La sintaxis de *dimensión* es muy similar a la especificación de variables:

índice–inferior: *índice–superior*, ...

con las siguientes diferencias. Tanto el *índice–inferior* como el *índice–superior* pueden ser constantes enteras o expresiones enteras que involucren variables con atributo **parameter** o argumentos enteros. En este caso diremos que la especificación se hace de forma *explícita*.

En el siguiente ejemplo se especifica explícitamente los argumentos de una subrutina que calcula la potencia n–ésima de una matriz.

```
subroutine potencia_e (n, m, A, B)

      integer,  intent(in)::  n, m      !  potencia y dimension de A
      real, intent(in) ::  A(m,m)
      real, intent(out)::  B(m,m)       !  B = potencia n-esima de A

      integer ::  i

            B = A
            do i=1, n-1
                        B = matmul(A,B)
            end do

end subroutine potencia_e
```

En ciertas ocasiones, es conveniente no especificar la dimensión de la matriz. En estos casos el *índice–inferior* y el *índice–superior* se pueden suprimir de la especificación y diremos que estamos ante una especificación de *forma importada* o *asumida*. Este tipo de especificación permite importar la dimensión del objeto o variable que ocupa la lista de argumentos en el momento en que el procedimiento es llamado. La misma subrutina anterior mediante una especificación importada o asumida y con el último parámetro opcional queda:

```
subroutine potencia (n, A, B)

    integer         ::  n           !  potencia n
    real            ::  A(:, :)     !  si B no esta, A = A^n
    real, optional  ::  B(:, :)     !  B =  A^n

      integer ::  i

      if (present(B)) then
                           B = A
                           do i=1, n-1;  B = matmul (A,B);  end do
      else
                           do i=1, n-1;  A = matmul (A,A);  end do
      end if

end subroutine potencia
```

La sintaxis de las funciones también nos permite escribir vectores o matrices de funciones. En el siguiente ejemplo se escribe una función que a partir de una matriz devuelve su potencia n–ésima.

```
function potencia (n, A)

    integer            ::  n                  ! potencia n
    real, intent(in)   ::  A(:, :)
    real, pointer      ::  potencia(:, :)     ! potencia =  A^n


                integer ::  i

                potencia = A

                do i=1, n-1;  potencia = matmul (potencia,A);  end do

end  function potencia
```

Cuando los argumentos son procedimientos, es necesario especificar el interfaz del procedimiento mediante la *especificación de interfaces* que tiene la siguiente sintaxis:

interface

subroutine *nombre* (*lista–argumentos*)

[*sentencia use*]

especificación de argumentos

end subroutine

function *nombre* (*lista–argumentos*)

[*sentencia use*]

especificación de argumentos

especificación de nombre

end function

end interface

En el siguiente programa calculamos la potencia n–ésima de una matriz mediante la subrutina `potencia` de los ejemplos anteriores. Donde la matriz de entrada `A` tiene especificación importada y la matriz de salida `B` tiene especificación importada y opcional. El programa se asocia mediante la sentencia **use** a una unidad **module** que contiene a la subrutina `potencia`.

```
program llamada_potencia

       use funciones

       real :: matriz(2,2)          !   especificacion explicita

         matriz = 3                 !   inicialización

         call potencia(3, matriz)   !   matriz^3

         write(*,*) matriz(1,1:2)   !   imprime la primera fila

end program llamada_potencia
```

En el siguiente ejemplo se crea una fusión que calcula la integral definida entre $[x_0, x_f]$ de una función real de variable real. Los dos primeros argumentos de `integral` son los límites de integración y el tercer argumento es la función que se quiere integrar.

```
      function Integral (x0,  xf,  f)

                        interface
                                  function f(x)
                                             real :: x
                                             real :: f
                                  end function
                        end interface

                        real :: x0,  xf
                        real :: Integral

              ..........

      end function
```

Las siguientes funciones calculan la norma euclídea y norma del supremo de un vector. Todas tienen la misma funcionalidad pero diferentes especificaciones de parámetros.

```
module normas

 contains

    function norma_L2( vector )      !  Norma Euclidea de un vector

               real :: vector(:)     !  especificacion importada
               real :: norma_L2

                 norma_L2 =  sqrt(   sum( vector**2 )  )

    end  function norma_L2

    function norma_L2e( n, vector )  ! Norma Euclidea de un vector

             integer :: n
             real :: vector(1:n)     ! especificacion  explicita
             real :: norma_L2e

                norma_L2e =  sqrt(   sum( vector**2 )  )

    end function norma_L2e


    function norma_sup(vector)       ! Norma del supremo de un vector

               real :: vector(:)     ! especificacion importada
               real :: norma_sup

                 norma_sup =  maxval(  abs(vector)  )

    end function norma_sup

end module normas
```

Para poner de manifiesto las diferencias en la llamada de funciones con diferentes especificaciones de argumentos calculamos la norma euclídea de un vector fila de una matriz, y la norma del supremo del mismo vector fila y de la matriz completa. Todas las diferentes funciones de norma Euclídea o norma del supremo dan el mismo resultado. Es importante hacer notar que, en la especificación de tamaño asumido o importado, la dimensión no es parámetro ya que se asume la dimensión de la variable que aparece en la llamada. Este tipo de especificación es útil cuando la dimensión explícita de la matriz o vector no es necesaria para escribir la funcionalidad del procedimiento.

```
      program especificacion_argumentos

      use normas                    ! especificacion del interface

      real, allocatable :: A(:,:)

        allocate(A(1:2,1:2))        ! asignacion de memoria dinamica

!  *** inicializacion de la matriz A
       A(1,1:2) = (/ 1.1, 2.2 /)
       A(2,1:2) = (/ 3.3, 4.4 /)


!  *** norma Euclidea del vector: primera fila de A

       write(*,*) 'Norma Euclidea  ', norma_L2( A(1,1:2) )
       write(*,*) 'Norma Euclidea  ', norma_L2e( 2, A(1,1:2) )

!  *** norma del supremo de la primera fila de A y de la matriz A

       write(*,*) 'Norma supremo de A ',norma_sup( A(1,1:2) )
       write(*,*) 'Norma supremo de A ',norma_sup( reshape(A, (/4/)) )

      end
```

5.5. Definición de nuevos tipos y operaciones

Además de los tipos intrínsecos (**real, integer, complex, character, logical**), en FORTRAN se pueden definir tipos derivados a partir de tipos intrínsecos y definir nuevas operaciones entre tipos intrínsecos y tipos derivados. Toda nueva definición de nuevos tipos u operaciones se debe ubicar en unidades de programa **module** tal y como se indica en la sintaxis de las unidades module dada en la sección 3.4. La definición de tipos derivados tiene la siguiente sintaxis:

type *nombre*

especificación de componentes

end type

en donde en *especificación de componentes* se detalla las partes que integran el nuevo tipo que puede estar formado por: tipos intrínsecos y tipos previamente definidos.

En el siguiente ejemplo definimos un nuevo tipo llamado `triangulo` que contiene el área y la posición en coordenadas cartesianas (x, y) de sus tres vértices. Además, en el programa principal creamos un vector de 100 triángulos, inicializamos sus áreas y las sumamos.

```
module nuevos_tipos

          type triangulo
                       real :: area
                       real :: x(3)
                       real :: y(3)
          end type

end module nuevos_tipos


program triangulos
                   use nuevos_tipos

                   type (triangulo) v(100)

                   integer ::  i

                   forall(i=1:100)  v(i)%area  =  0.1

                   write(*,*)  ' Area total ',  sum( v%area )

end program triangulos
```

Como se observa en el ejemplo, la referencia a una parte o componente de una variable de nuevo tipo se hace mediante el carácter `%`. Otra forma de asignar componentes escalares que son estructuras previamente definidas es mediante una asignación con la siguiente sintaxis:

objeto = *nombre de tipo* (*lista de expresiones*)

donde *nombre de tipo* es el tipo de la variable escalar *objeto* y *lista de expresiones* es un conjunto de valores separados por comas que constituyen las componentes de estructura *objeto*. En la siguiente línea asignamos a la componente 10 de un vector de triángulos un área de 0.5 y las coordenadas de sus tres vértices $(0, 0), (1, 0)$ y $(0, 1)$.

```
v(10) = triangulo(  0.5,  (/0., 1., 0./),  (/0., 0., 1./) )
```

Además de crear nuevos tipos, mediante la especificación de interfaces podemos definir:

1. Nuevas operación con variables de tipos intrínsecos o tipos derivados.

2. Nuevas asignaciones `nombre = expresion` cuando nombre y expresión tienen tipos intrínsecos o derivados diferentes.

La sintaxis de las nuevas operaciones o extensión a nuevos tipos de operaciones ordinarias es la siguiente:

interface operator (*nombre–operador*)

function *nombre* (*lista–argumentos*)

[*sentencia use*]

especificación de argumentos

especificación de nombre

end function

end interface

donde *nombre-operador* es cualquier operador predefinido o un nuevo nombre de operador de la forma *nombre-operador*.

En el siguiente ejemplo definimos en producto vectorial entre dos vectores de $\mathbb{R}^3$. Primero, definimos el tipo vector de reales de tres componentes:

```
module nuevos_tipos

        type vector_R3
                      real :: c(3)
        end type

end module nuevos_tipos
```

Después, definimos lo que es el producto vectorial:

```
function producto_vectorial(u, v)    !  producto vectorial en R3

      use nuevos_tipos
      type (vector_R3), intent(in)  :: u, v
      type (vector_R3), intent(out) :: producto_vectorial

  producto_vectorial%c(1) = u%c(2) * v%c(3) - u%c(3) * v%c(2)
  producto_vectorial%c(2) = u%c(1) * v%c(3) - u%c(3) * v%c(1)
  producto_vectorial%c(3) = u%c(1) * v%c(2) - u%c(2) * v%c(1)

end function
```

Y creamos el operador `.x.` basado en la función anterior `producto_vectorial`:

```
   module nuevas_operaciones
       interface operator (.x.) !  operacion producto vectorial en  R3

           function producto_vectorial(u, v)

                  use nuevos_tipos
                  type (vector_R3), intent(in)  :: u, v
                  type (vector_R3), intent(out) :: producto_vectorial

           end function producto_vectorial

       end interface
   end module
```

Como ejemplo de aplicación creamos un programa principal con dos vectores de $\mathbb{R}^3$ y calculamos su producto vectorial.

```
   program operaciones

    use nuevos_tipos
    use nuevas_operaciones

   ! *** u y v son dos vectores de reales
         type (vector_R3) :: u = vector_R3( (/1., 2., 3./) )
         type (vector_R3) :: v = vector_R3( (/ 4.,5.,6. /) )

   ! *** calcula el producto vectorial de dos vectores de R3
         write(*,*) ' Vector u = ', u
         write(*,*) ' Vector v = ', v
         write(*,*) ' Producto vectorial u x v = ',   (u.x.v)

   end
```

Es importante hacer notar el nivel de abstracción que se puede conseguir al definir nuevos operadores. La simple e inocente expresión `(u.x.v)` calcula el producto

vectorial en $\mathbb{R}^3$. Operadores mucho más complejos se pueden definir permitiendo al programador escribir en una capa de abstracción más próxima al lenguaje matemático.

La especificación de interfaces además de servir para especificar los argumentos de una subrutina o función, también sirve para definir nuevas operaciones y asignaciones con diferentes tipos de datos. La sintaxis de la definición de las nuevas asignaciones es:

interface assignment (=)

subroutine *conversor–tipos* (*nombre*, *expresión*)

[*sentencia use*]

id–tipo, **intent(out)** :: *nombre* [(*dimensión*)]

id–tipo, **intent(in)** :: *expresión* [(*dimensión*)]

end subroutine

end interface

donde *conversor–tipos* es el nombre de una subrutina que realiza la asignación *nombre* = *expresión*. Es importante hacer notar que el objeto de este interface es extender la asignación a situaciones en las que *expresión* y *nombre* tienen diferentes tipos. Esta subrutina toma el dato de *expresión*, lo manipula y lo carga en *nombre*.

En el siguiente ejemplo pretendemos convertir un complejo dado en forma cartesiana a un complejo en forma polar mediante una asignación y luego calcular el producto de dos complejos dados en su forma polar. Para ello definimos un nuevo tipo que es un número complejo en forma polar:

```
module nuevos_tipos

    type polar                  !  define un complejo en forma polar
              real :: rho       !  modulo
              real :: theta     !  argumento

    end type

end module nuevos_tipos
```

Definimos una subrutina que pasa el complejo de forma cartesiana a polar:

```
subroutine cartesiana_polar(z_polar, z_cartesiana)
          use nuevos_tipos
          type (polar), intent(out) :: z_polar
          complex,      intent(in)  :: z_cartesiana

      real :: Pi

      Pi = 4*atan(1.0)

 !  modulo
    z_polar%rho = abs(z_cartesiana)

 !  argumento
    z_polar%theta = atan2( real(z_cartesiana), &
                           imag(z_cartesiana) )* 360 /(2*Pi)

end subroutine
```

Y una subrutina que multiplica complejos a partir de su forma polar:

```
 function producto(z, w)  ! Calcula el producto de dos
                          ! complejos en forma polar
          use nuevos_tipos
          type (polar),  intent(in) :: z,  w
          type (polar)              ::  producto

         producto%rho =  z%rho * w%rho        !  multiplica los modulos
         producto%theta = z%theta + w%theta   !  suma los argumentos

end function
```

Especificamos el interfaz en un módulo de asignaciones y operaciones extendidas:

```
module asignaciones_y_operaciones_extendidas

    interface assignment (=)    !   convierte un complejo de
                                !   forma cartesiana a forma polar

        subroutine cartesiana_polar(z_polar,  z_cartesiana)

                use nuevos_tipos
                type (polar),  intent(out)  :: z_polar
                complex, intent(in)         :: z_cartesiana

        end subroutine

    end interface
```

```
    interface operator (*)          !   multiplica en forma polar
         function producto(z, w)

                 use nuevos_tipos
                 type (polar), intent(in):: z, w
                 type (polar)            :: producto

        end function
    end interface

end module asignaciones_y_operaciones_extendidas
```

Por último, creamos un programa principal donde se especifican dos complejos uno en forma cartesiana y otro en forma polar:

```
program asignaciones

 use nuevos_tipos
 use asignaciones_y_operaciones_extendidas

   complex :: z = (1.0, 1.0)         ! z  complejo en forma cartesiana
   type (polar) w                    ! w  complejo en forma polar

   write(*,*) ' Forma cartesiana z=', z  ! imprime forma cartesiana
   w = z                                 ! convierte cartesiana a polar
   write(*,*) ' Forma polar =', w

   write(*,*) ' Producto en forma polar = ', w*w

end program asignaciones
```

La asignación `w = z` es equivalente a la llamada `call asigna(w,z)` que permite no solo extender las asignaciones entre tipos diferentes sino hacer el programa más fácil de entender.

Capítulo 6

Procedimientos

6.1. Diseño y desarrollo de un procedimiento

Aunque el diseño software es algo muy personal y que el propio programador va adquiriendo con la experiencia, es posible dar ciertas directrices para un desarrollo eficiente. Generalmente, estas directrices se engloban en una determinada metodología de trabajo.

En primer lugar, un procedimiento debe contener una cabecera de especificación en la que se indique:

- Descripción breve de la funcionalidad del procedimiento.
- Asociaciones a unidades de programa mediante la sentencia **use** que permiten acceder a variables externas, usar nuevos tipos de variables, utilizar operaciones, asignaciones y procedimientos externos.
- Especificación de los interfaces de los argumentos que son procedimientos.
- Especificación de los argumentos que se asocian con variables. Descripción detallada de los argumentos y su clasificación en argumentos de entrada y salida.
- Autor y fecha.

Esta cabecera permite al programador tratar el procedimiento como una caja negra que procesa unas entradas para obtener unas salidas utilizando ayudas o procedimientos externos a la unidad.

A continuación se especifican las variables del procedimiento, prestando especial cuidado en su inicialización. Variables locales no inicializadas pueden producir fenómenos históricos, que se manifiestan en su ejecución como problemas no deterministas. Es decir, si una variable local se utiliza en una expresión sin haber sido inicializada, el resultado de la expresión dependerá del valor aleatorio que tenga en ese momento la variable. Puede ser que sea cero, o cualquier otro valor desconocido y, en general, diferente cada vez que el procedimiento es llamado. De esta forma, no podremos asegurar la funcionalidad del procedimiento. El siguiente ejemplo almacena en la variable `contador` el número de veces que ha sido llamada la unidad de programa; cuando la llamamos por primera vez, inicializamos todas las variables necesarias.

```
integer, save :: contador= 0;       !   contador a  0

   contador =  contador + 1         !   numero de llamadas

   if (contador  == 1) then         !   primera vez que se llama

                    ......
   end if
```

Aunque uno pudiera pensar que en un procedimiento los nombres de las variables o de los argumentos no son relevantes, y que lo importante es el algoritmo, nada más lejos de la realidad. El nombrar correctamente las variables y los argumentos puede reducir drásticamente el volumen de comentarios necesarios para entender el procedimiento. Tanto los procedimientos como los argumentos y variables deben estar nombrados de forma suficientemente clara como para que permitan identificar su función sin necesidad de recurrir constantemente a documentación adicional. En ciertas ocasiones, se hace necesario una determinada metodología para nombrar a las tareas que pertenecen a un proyecto software. Las variables y los argumentos pueden tener muchos caracteres para identificar sus objetos sin necesidad de grandes comentarios. Sin embargo, si existe un modelo matemático previo a partir del cual se realiza la implementación es imprescindible que la notación sea estrictamente igual a la del modelo matemático. De esta forma, se facilitará por una parte la compresión del código para un futuro programador, y por otra parte cualquier modificación a partir del modelado matemático. Los comentarios en una unidad de programa son imprescindibles pero no se deben convertir en una novela acerca del mismo.

El tamaño de las unidades de programa depende mucho del estilo de programación, pero como regla general una unidad está formada al menos por una o dos sentencias y a lo sumo por 100 sentencias. Unidades de programa grandes siempre pueden y deben partirse en otras unidades que realizan funciones más simples.

6.2. Optimización

Generalmente, una vez desarrollada y probada la unidad de programa es necesario reducir los tiempos de ejecución o el tamaño de memoria requerida por limitaciones de nuestra arquitectura. Surge así una necesidad en el desarrollo de la unidad que es la de la *optimización*, y que consiste en hacer un código más eficiente.

La tarea de optimización se realiza en dos etapas. En la primera etapa *el compilador optimiza* el código fuente para generar un ejecutable más eficiente. Dependiendo de la habilidad del compilador la tarea de optimización será más o menos efectiva. Es importante en esta etapa conocer las opciones de optimización que ofrece nuestro compilador.

Si, una vez superada esta primera etapa, el grado de eficiencia alcanzado por el compilador no satisface al programador, se inicia la segunda etapa donde *el programador optimiza*. Para llevar a cabo esta etapa es importante que el programador conozca a la perfección el compilador y la arquitectura donde se ejecutará el código. A partir de estos conocimientos es posible reescribir algunos algoritmos adaptándolos a la arquitectura. Sin embargo, modificar la programación adaptándola a una arquitectura dada sacrifica la claridad del programa y los resultados no suelen ser impresionantes. Además, un cambio de arquitectura obligaría a reescribir las partes de código modificadas. Cuando la optimización del compilador no es suficiente, es preferible que el programador se centre en cuestiones de algoritmia más que en cuestiones de lenguaje. El cambio de algoritmo para resolver un determinado problema puede dividir por 10 tanto las necesidades de memoria como el tiempo de ejecución.

A continuación daremos algunas optimizaciones que realizan automáticamente la mayoría de los compiladores:

- Constantes en tiempo de compilación. El compilador en tiempo de compilación detecta operaciones cuyos resultados no variarán en tiempo de ejecución y los transforma en constantes. Ejemplo:

```
integer ::  i
...........
...........
i = 2 * (5 + 2)
```

  El compilador transforma y ejecuta:

```
integer ::  i
...........
...........
i = 14
```

- Simplificación de expresiones. El compilador detecta expresiones que pueden ser transformadas a otras con menor número de operaciones. Ejemplo:

```
...........
y = 2 * sin(x) + 3 * sin(x)
...........
```

El compilador transforma y ejecuta:

```
...........
y = 5 * sin(x)
...........
```

- Eliminación de subexpresiones comunes. El compilador detecta subexpresiones que son comunes en varias líneas de código y las almacena en una variable temporal reduciendo la cantidad de operaciones. Ejemplo:

```
integer ::  i, j, k, l, n
...........
i = (l + k) * 2
j = (l + k) + n
```

El compilador transforma y ejecuta:

```
integer ::  temp
...........
temp = (l + k)
i = temp * 2
j = temp + n
```

- Ordenamiento de instrucciones. Al evaluar una instrucción el compilador realiza el análisis de la misma de izquierda a derecha. Por ejemplo, al evaluar la expresión:

```
x = a * b * (y + z)
```

el compilador realiza las siguientes operaciones internas:

```
-trae de la memoria el valor de a
-le multiplica el valor b
-el resultado lo almacena en una variable temporal t1
-trae de la memoria el valor de y
-le suma el valor de z
-el resultado lo multiplica por el valor de t1
```

En este caso se han necesitado seis operaciones internas. En este caso el compilador reordena la expresión original a la forma:

```
x = (y + z) * a * b
```

que requiere sólo cuatro operaciones internas:

```
-trae de la memoria el valor de y
-le suma el valor de z
-le multiplica el valor de a
-le multiplica el valor de b
```

- Reducción a operaciones menos costosas. Dentro de las operaciones básicas existen algunas que ejecutan más rápido que otras. Por ejemplo, en la mayoría de los compiladores la suma y la resta tienen un coste en tiempo similar. Mayor coste tiene una multiplicación y más aún la división. El compilador intentará reducir cualquier operación a otra de menor coste. Por ejemplo, las siguientes operaciones:

```
real(4)  x, y, a, b
x = y * 2
a = b**2
```

el compilador las transforma y ejecuta:

```
real(4)  x, y, a, b
x = y + y
a = b * b
```

- Reducción a funciones intrínsecas más eficientes. Existen funciones de la biblioteca de FORTRAN que son más eficientes que otras. El compilador reducirá cuando sea posible a aquella que ejecute más rápido. Ejemplo:

```
real(4)  x, y
y = x**0.5
```

El compilador transforma y ejecuta:

```
real(4)  x, y
y = sqrt (x)
```

La función raíz cuadrada ejecuta más rápido que la función exponencial.

- Optimización de bucles: expresiones independientes. El compilador analiza las expresiones que se encuentran dentro de un bucle. Si existe alguna que no depende del mismo lo retira fuera del bucle para reducir tiempo de ejecución. Ejemplo:

```
do i=1, n
     .....
     a = 2.
     .....
end do
```

El compilador transforma y ejecuta:

```
a = 2.
do i=1, n
     .....
     .....
end do
```

La optimización por medio del compilador de las operaciones vectoriales es una de las características más relevantes de FORTRAN 95 frente a otros lenguajes de programación. El uso de funciones vectoriales intrínsecas, así como la automática recolocación en memoria de los datos de una matriz por medio del compilador, hacen que sea mucho más eficiente la labor del compilador que la que pueda realizar el programador.

6.3. Llamadas a procedimientos

Generalmente, una aplicación realiza tareas repetitivas a lo largo de su proceso. Por ejemplo, si nuestra aplicación calcula valores de integrales definidas más de una vez, conviene abstraer el proceso de la integración en una subrutina o función. El uso de subrutinas y funciones permite realizar abstracciones de procesos evitando la duplicación de código. Si existe código duplicado o redundante, cualquier modificación en el algoritmo de cálculo requiere modificaciones en todas las partes del programa donde aparezcan los segmentos redundantes. La redundancia

hace poco transparente y poco estructurado el código, y alarga innecesariamente el programa.

La sintaxis de la llamada a una subrutina es de la forma:

call *nombre* ([*nombre–argumento* =] *nombre–objeto–llamada*, ...)

Si el procedimiento es una función, la llamada es igual salvo que se suprime la palabra clave **call**. Cuando un programa llama a un procedimiento, la ejecución del programa que hace la llamada se para mientras se realiza el procedimiento. Al finalizar el procedimiento, el programa que hace la llamada reanuda su ejecución. Cuando el procedimiento se llama, los argumentos del procedimiento ("dummy arguments") se asocian con las variables con las que se llama ("actual arguments"). Si existen argumentos opcionales, el número de objetos asociados en la llamada puede diferir del número de argumentos especificados en el procedimiento.

El puntero de una variable es la señal o la dirección en la cual se encuentra el contenido de la variable, y el dato que contiene la variable es el contenido del puntero o de la dirección. El nombre de una variable constituye la etiqueta que el programador tiene para distinguirla del resto de variables. En FORTRAN, la asociación que se produce en la llamada de un procedimiento consiste en pasar a cada argumento del procedimiento la dirección de la variable y no su valor. De esta forma, mediante la llamada a un procedimiento se pueden manejar los punteros.

La comunicación entre unidades de programa o procedimientos sólo se puede realizar mediante variables externas o mediante argumentos de subrutinas o funciones. Si la comunicación se realiza a través de variables externas, no es fácil analizar el flujo de datos de un programa. Se requieren herramientas específicas para analizar el flujo y una gran dosis de paciencia. Es una buena metodología restringir el uso de las variables externas para constantes físicas universales, constantes matemáticas o constantes numéricas propias del cálculo numérico, pero nunca como canal dinámico de comunicación entre unidades de programa. Es decir, las variables externas deberán ser –siempre que sea posible– constantes estáticas y que no constituyan variables dinámicas en el programa principal. La comunicación entre procedimientos se puede realizar a través de sus argumentos que pueden ser:

1. Argumentos escalares de tipos intrínsecos o derivados.
2. Procedimientos como argumentos.
3. Argumentos vectoriales o matriciales de tipos intrínsecos o derivados.

El caso más sencillo de comunicación es cuando los argumentos son escalares. En el siguiente ejemplo se crea una subrutina que calcula la hipotenusa de un triángulo rectángulo.

```
module procedimientos

contains

      subroutine hipotenusa (c1,  c2,  h)

                  real, intent(in)   ::  c1,  c2    !   catetos
                  real, intent(out)  ::  h          !   hipotenusa

                  h  =  sqrt( c1**2 + c2**2  )

      end subroutine hipotenusa

end module procedimientos
```

Se especifica que los tres argumentos son escalares reales. Se crea un programa principal desde el que se la llama tres veces a la subrutina anterior para calcular las hipotenusas de tres triángulos. Cada vez que se llama a la subrutina que calcula la hipotenusa colocamos en la llamada variables escalares diferentes que son las componentes de tres vectores.

```
      program triangulos   ! calcula las hipotenusas de tres triangulos

           use procedimientos

           real  ::  a(3) = (/ 1e0, 2e0,   3e0   /)   ! catetos a y b
           real  ::  b(3) = (/ 2e0, 3.5e0, 4.5e0 /)
           real  ::  h(3)                             ! hipotenusa

           integer ::  i      ! triangulo i

           do i = 1, 3
                       call hipotenusa ( a(i), b(i), h(i) )
                       write(*,*) ' La hipotenusa es: ', h(i)
           end do

      end program triangulos
```

Un caso un poco más complicado de comunicación entre unidades de programa es cuando uno de los argumentos es una función o subrutina. En el siguiente ejemplo creamos una función que devuelve la integral definida entre dos puntos de una función dada.

```
!  Calcula la integral definida de una funcion f
   function Integral(a, b, f)
                 real, intent(in) :: a, b  ! intervalo de integracion
                 interface                 ! interfaz de f(x)
                     function f(x)
                       real, intent(in) :: x
                       real, intent(out):: f
                     end function
                 end interface
                 real, intent(out) :: Integral

     integer :: i, n=100;     ! discretizacion en n subintervalos
     real    :: h, x, s=0e0

           h = (b-a) / (n-1)

           do i = 1, n
                x = a + h * ( i - 1 )
                s = s + h * f(x)
           end do

           Integral = s

   end function Integral
```

Nos interesa que la función $f(x)$ sea un argumento para que la unidad Integral sirva para calcular la integral de cualquier función real de variable real como se especifica en el interfaz de la función. Como ejemplo de uso de esta unidad calculamos la integral de $R(x) = sen(x)/x$ entre 0 y 5. Definimos la función:

```
function R(x)     ! definicion de R(x)= sen(x) / x
    real, intent(in)   :: x
    real, intent(out)  :: R

        if (x == 0e0) then
                        R = 1e0
        else
                        R = sin(x) / x
        end if
end function R
```

Creamos una unidad **module** de nombre `integracion` donde colocamos las dos funciones `R` y `Integral` y calculamos la integral.

```
program Integrar

   use integracion

 !  ** Calcula la integral de sen(x) / x entre 0 y 5

       write(*,*) 'Integral =', Integral(0e0, 5e0, R)

end program Integrar
```

Por último, la comunicación entre unidades de programa a través de argumentos que son variables vectoriales o matriciales es, probablemente, la mayor potencialidad del lenguaje y sin duda una de las tareas más difíciles de aprender. FORTRAN 95 permite pasar a través de los argumentos de una subrutina sectores de matrices con la misma sintaxis que en las operaciones con matrices ya vistas.

En el siguiente ejemplo se hacen llamadas a procedimientos y se asocian argumentos con sectores de matrices. Consideramos el siguiente sistema de ecuaciones de segundo orden,

$$A(x)y'' + B(x)y' + C(x)y = 0, \tag{6.1}$$

donde $y(x)$ es un vector columna de dimensión m, $A(x), B(x)$ y $C(x)$ son matrices de $m \times m$. El sistema anterior junto a ciertas condiciones en los extremos de intervalo $[0, 1]$ constituye un problema de contorno.

Cuando se discretiza el intervalo $[0, 1]$ en n puntos equiespaciados

$$\{x_j = (j-1)/(n-1), \qquad j = 1, \ldots n\}, \tag{6.2}$$

se obtiene el siguiente sistema lineal:

$$D_{j,j+1}y_{j+1} + D_{j,j}y_j + D_{j,j-1}y_{j-1} = b_j, \tag{6.3}$$

donde la matriz del sistema es tridiagonal de $n \times n$, b_j es el término independiente que aparece al discretizar las condiciones de contorno y $D_{j,j+1}, D_{j,j}, D_{j,j-1}$ son submatrices de tamaño $m \times m$ definidas como:

$$D_{j,j+1} = A(x_j) + \frac{\Delta x}{2}B(x_j), \tag{6.4}$$

$$D_{j,j} = -2A(x_j) + \Delta x^2 C(x_j), \tag{6.5}$$

$$D_{j,j-1} = A(x_j) - \frac{\Delta x}{2}B(x_j). \tag{6.6}$$

Primero, escribimos un programa que inicializa cada una de las submatrices $A(x), B(x)$, y $C(x)$:

```
subroutine Sistema (x,  A, B, C)         ! sistema de ecuaciones

 real , intent(in) :: x                  ! variable independiente
 real , intent(out) :: A(2,2), B(2,2), C(2,2)


           A(1,1:2) =  (/ x**2, 0e0 /)
           A(2,1:2) =  (/ 1e0,  0e0 /)

           B(1,1:2) = (/ 0e0, - 1e0 /)
           B(2,1:2) = (/ 1e0,   0e0 /)


           C(1,1:2) = (/ 1e0, 0e0 /)
           C(2,1:2) = (/ 0e0, 1e0 /)

end subroutine Sistema
```

Segundo, creamos una subrutina que escriba por filas una matriz cuadrada:

```
subroutine imprime(n, A)

           integer, intent(in) ::  n
           real,    intent(in) ::  A(n,n)  ! especificacion explicita

    do i=1, n

        write(*,'(20f7.3)') A(i,1:n)   ! imprime la matriz por filas

    end do

end subroutine imprime
```

Tercero y último, agrupamos los procedimientos anteriores en una unidad **module** de nombre `agrupa` y escribimos el programa principal que crea e imprime la matriz de submatrices teniendo en cuenta que los elementos de la diagonal principal, superior e inferior están dados por (6.4)–(6.6).

```
program Matriz_de_submatrices

   use agrupa

   integer, parameter :: n=7, m=2  ! discretizacion y # variables
   real ::  D(m, n, m, n)          ! matriz de submatrices

   real ::  f(m*n), y(m*n)         ! termino independiente y solucion

   real ::  A(m,m), B(m,m), C(m,m) ! matrices del sistema
   real ::  dx, x                  ! paso espacial

     integer ::   j

!    *** inicializa a ceros
        D = 0e0
        dx = 1e0/(n-1)


!    *** barre los nodos de la malla x_j , j=2, n-1
        do j=2, n-1

!       *** Determina A, B y C

            x = dx*(j-1)
            call Sistema (x, A, B, C)

!       *** Carga los valores en la diagonal superior,
!           inferior y diagonal principal.

            D(1:m, j, 1:m,  j+1) = A(1:m,1:m) + dx/2 * B(1:m,1:m)
            D(1:m, j, 1:m,  j-1) = A(1:m,1:m) - dx/2 * B(1:m,1:m)
            D(1:m, j, 1:m,  j  ) = -2*A(1:m,1:m) + dx**2 * C(1:m,1:m)

        end do

!    *** j=1 y j=n son las condiciones de contorno
!        Imponer las condiciones de contorno:

!    *** imprime la matriz del sistema As(m*n, m*n)
!        elimina la condiciones de contorno j=1, j=n

         call imprime ( (n-2)*m, D(1:m, 2:n-1, 1:m, 2:n-1 ) )
end program
```

En el bucle del programa se recorren cada uno de los nodos de la malla calculando los valores de las matrices $A(x), B(x)$ y $C(x)$. Estos valores se cargan en la matriz de submatrices mediante una asignación de sectores de matrices. En la llamada `imprime` pasamos un sector de la matriz de submatrices. Es importante hacer

notar que mientras la matriz de submatrices D es de cuatro índices, el parámetro matriz de la subrutina `imprime` tiene dos índices. Es decir, la llamada a `imprime` reformatea un sector de una matriz de cuatro índices a una matriz de dos índices. En FORTRAN a diferencia de otros lenguajes el reformateo se hace por columnas. Si el orden de los índices en la especificación de la matriz de submatrices es el adecuado, podremos obtener el resultado perseguido. Sea la matriz del sistema:

$$D = \begin{pmatrix} D_{11} & D_{12} & \dots & D_{1n} \\ D_{21} & D_{22} & \dots & D_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ D_{n1} & D_{n2} & \dots & D_{nn} \end{pmatrix}$$

donde los elementos D_{ij} son las siguientes submatrices de $m \times m$:

$$D_{ij} = \begin{pmatrix} d_{11} & d_{12} & \dots & d_{1m} \\ d_{21} & d_{22} & \dots & d_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ d_{m1} & d_{m2} & \dots & d_{mm} \end{pmatrix}$$

Si queremos trabajar con D como si fuera una matriz de $n\ m$ filas y $n\ m$ columnas, debemos reformatear a dos índices. La forma más fácil de reformatear es a través de una llamada a una subrutina o mediante una sentencia **reshape**. En este programa hemos reformateado mediante una llamada a la subrutina `imprime`. Como el orden por defecto en el reformateo es por columnas, el orden de los índices para conseguir el resultado deseado debe ser: primero la dimensión del sistema m, segundo el número de puntos de la discretización n, tercero m y cuarto n.

Otro ejemplo interesante de reformateo de matrices y vectores es el que se suele hacer cuando se simula el comportamiento de un sistema descrito por un sistema de ecuaciones en derivadas parciales. Toda integración de un problema de evolución descrito por un conjunto de ecuaciones en derivadas parciales requiere una discretización espacial y una discretización temporal. Para fijar ideas, podríamos pensar en un problema de evolución en mecánica de fluidos en un espacio tridimensional caracterizado por la presión y dos componentes de velocidad. El objetivo es poder tratar con cierto nivel de abstracción las variables de evolución del fluido. Para integrar el problema discretizado espacialmente es conveniente tratarlo como un sistema de ecuaciones diferenciales ordinarias. De esta forma, se

puede elegir un esquema temporal estándar para integrar el problema. La discretización espacial exigirá reformatear el vector de estado en tres campos escalares que en cada instante son funciones de un punto genérico (x, y, z). Así, el cálculo de derivadas espaciales o de gradientes se puede hacer de forma cómoda.

En FORTRAN 95 se pueden utilizar dos procedimientos para reformatear un vector o una matriz:

1. Mediante la función intrínseca **reshape**.

2. Mediante la llamada a procedimientos y asociación de argumentos y variables con diferente especificación.

La forma más sencilla consiste en reformatear una variable mediante la sentencia **reshape**. El problema es que la nueva variable ocupa la misma cantidad de memoria que la variable original. Esto puede llegar a ser un problema insalvable cuando trabajamos con problemas que tienen vectores de estado muy grandes (millones de datos).

Para salvar esta limitación podemos llamar a procedimientos y asociar argumentos y variables con diferentes dimensiones. Aunque la regla de oro en la asociación entre variables y argumentos es que ambos deben tener el mismo tipo, la misma clase y el mismo número de dimensiones (Type Kind Rank rule), en ciertas ocasiones esta regla se puede ignorar a sabiendas del programador. Cuando la especificación de los argumentos de un procedimiento es importada o asumida (`A(:,:,:)`), no nos podemos saltar esta regla puesto que al hacer la asociación entre los argumentos y las variables, el compilador busca las dimensiones y los límites de los índices para asignárselos a los argumentos. Si las dimensiones no coinciden el compilador dará un error alertándonos del problema. Sin embargo, si la especificación de los argumentos es explícita, como por ejemplo:

```
subroutine Opera( n, A )

     integer, intent(in) :: n
     real,    intent(in) :: A(n,n)

     .........
```

la llamada se puede hacer con una variable que no tenga las mismas dimensiones que el argumento, como se observa en el siguiente ejemplo:

```
program Puntero

      integer, parameter :: n=10
      real ::   U(n*n)

      call Opera(n, U)

     .........
```

Se recuerda que en una llamada la asociación entre argumentos y variables consiste en pasar la dirección del primer elemento de la variable de llamada. De esta forma, pasamos un puntero a `Opera` que apunta a `U(1)`. Dentro de `Opera` se manejan $n \times n$ datos a partir de esa dirección en forma de una matriz de n filas y n columnas. El problema que tiene esta forma de operar es que si el programador no está seguro de lo que hace las consecuencias pueden ser desastrosas.

En el siguiente ejemplo hacemos asociaciones entre argumentos y variables con diferentes dimensiones, es decir, saltándonos la regla TKR (Type Kind Rank). De esta forma, podemos trabajar sobre las mismas direcciones de memoria pensado que a veces la variable dependiente es un vector de estado y otras veces la misma variable está formada por tres campos escalares tridimensionales. Lo mismo se podría haber conseguido con la función **reshape** y cumpliendo la regla TKR pero con el coste de haber duplicado la necesidades de memoria.

```
program Punteros

 use Fisica

 real , allocatable :: U(:), F(:)
 integer :: n

    Nx = 10;  Ny = 200;  Nz = 5; n = (Nx+1)*(Ny+1)*(Nz+1)

    write(*,*) ' Dimension del vector de estado ', n


    allocate(U(1:3*n), F(1:3*n))  ! tres variables de dimension n

    U( 1 : 3*n )  = 1             ! inicializa a  1

    call Funcion(U, F)            ! calcula la F


end program Punteros
```

```
module Fisica

     integer  :: Nx       ! puntos a lo largo eje x
     integer  :: Ny       ! puntos a lo largo eje y
     integer  :: Nz       ! puntos a lo largo eje z

contains

subroutine Funcion (U, F)
                real, intent(in)  ::  U(:)     ! vector de estado
                real, intent(out) ::  F(:)     ! derivada de U

    integer ::  dim
    dim = (Nx+1) * (Ny+1) * (Nz+1)

 ! dummy arguments and actual arguments have different shapes
 ! problem :  no TKR rule  ( Type, Kind and Rank )
 !            programmer must be careful (no bounds are checked)

   call Fluidos (N1 = Nx, N2 = Ny, N3 = Nz,          &
                 P  = U( 1 ),                        &
                 v  = U( dim+1 ),                    &
                 w  = U( 2*dim+1 ),                  &
                 Fp = F( 1 ),                        &
                 Fv = F( dim+1 ),                    &
                 Fw = F( 2*dim+1 )                    )

end subroutine Funcion


subroutine Fluidos (N1, N2, N3, P, v, w, Fp, Fv, Fw)

         integer, intent(in)  ::  N1, N2, N3
         real, intent(in)  ::  P(0:N1, 0:N2, 0:N3)  ! presion
         real, intent(in)  ::  v(0:N1, 0:N2, 0:N3)  ! velocidad u
         real, intent(in)  ::  w(0:N1, 0:N2, 0:N3)  ! velocidad v

         real, intent(out) ::  Fp(0:N1, 0:N2, 0:N3) ! derivada de la P
         real, intent(out) ::  Fv(0:N1, 0:N2, 0:N3) ! derivada de la u
         real, intent(out) ::  Fw(0:N1, 0:N2, 0:N3) ! derivada de la v

!    *** sentencias
         Fp = P + v; Fv = v + w; Fw = v - w

end subroutine Fluidos

end module Fisica
```

Capítulo 7

Proyecto software multicapa

7.1. Fases en el desarrollo

En este capítulo analizaremos las fases para el desarrollo de un programa o un proyecto software de tamaño pequeño (dos años hombre) y expondremos la metodología multicapa vinculada a la fase de la *implementación*. El ciclo de vida completo de un proyecto software de gran tamaño (más de veinte años hombre) no es objeto del presente documento.

La primera fase del proyecto consiste en el análisis de las necesidades, lo cual origina el enunciado del problema que queremos resolver. Esta fase es de vital importancia para fijar desde un principio este problema. Una ambigua definición del enunciado implicará inevitablemente ambigüedades en otras fases, que traerán como consecuencia una modificación continua de las mismas. Lamentablemente, la definición del enunciado del problema no se puede hacer tan estricta como uno quisiera, y siempre es necesario iterar con las diferentes fases del desarrollo para dejar el enunciado del problema perfectamente definido.

La segunda fase del proyecto consiste en la formulación del modelo matemático del problema físico a resolver, así como la determinación de los algoritmos numéricos necesarios para la resolución. Dependiendo del énfasis que hagamos en la modelización del problema y la complejidad de éste, el esfuerzo de esta fase será mayor o menor.

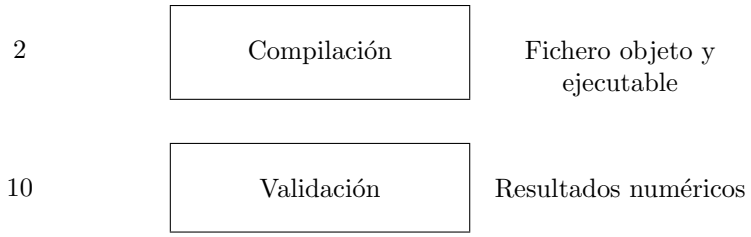
La tercera fase consiste en la implementación de los algoritmos numéricos en un lenguaje de programación. La escritura de los programas en una determinada arquitectura se realiza mediante un editor de textos y el esfuerzo necesario para concluir esta fase es pequeño frente al de otras fases. El resultado de esta fase es un fichero ASCII que contiene lo que se conoce como código fuente.

La cuarta fase es la compilación del código fuente en una determinada arquitectura "hardware". Generalmente, la compilación consiste en un análisis sintáctico del código fuente y una posterior generación de código ejecutable para un determinado microprocesador o CPU. En esta fase el programador, con los mensajes de error y alertas del compilador, debe eliminar los posibles errores sintácticos del código fuente. Como resultado de esta fase se obtiene un fichero que llamamos ejecutable, y que puede correr o ejecutarse en una determinada arquitectura. Es importante hacer notar que mientras que el código fuente es independiente de la arquitectura, el código ejecutable es específico de una arquitectura "hardware".

La quinta y última fase consiste en la validación de los resultados numéricos que se obtienen al correr el fichero ejecutable de la fase anterior. Esta fase queda concluida cuando existe una cierta seguridad de que los resultados numéricos son correctos. Dependiendo de la complejidad del problema, la seguridad al cien por cien no es generalmente algo alcanzable.

Para agilizar y conseguir superar con éxito cada una de las fases anteriores se suele utilizar un *entorno de desarrollo* que es un conjunto integrado de herramientas que permiten el desarrollo de un proyecto software de forma eficiente. En la siguiente tabla hemos representado esquemáticamente las fases del proyecto software, mostrando el esfuerzo necesario en una escala de uno a diez y el resultado de cada una de las fases.

ESFUERZO NECESARIO	FASE	RESULTADOS
1	Análisis de las necesidades	Enunciado del problema
8	Estudio del problema	Modelo y algoritmos
2	Implementación	Fichero fuente



7.2. Metodología de programación

En los comienzos de la informática el estilo de programación fue un aspecto que dependía de las características personales y habilidades del programador. Con la evolución de la informática y su creciente aplicación en disciplinas tales como las ciencias y la ingeniería, los programas se hicieron más complejos y comenzaron a ser tratados por equipos multidisciplinarios de programadores. Además, la vida útil de los programas se incrementó y comenzó a ser éste un aspecto importante en el estilo de programación.

Durante el mantenimiento de un código de simulación muchos ingenieros trabajan sobre el mismo corrigiendo errores, perfeccionando o completando los modelos y adaptando códigos existentes a nuevas arquitecturas. Surge así la necesidad de crear una metodología que recoja la experiencia de los ingenieros y programadores en la construcción de grandes códigos. Es decir, una colección de normas para orientar sobre la forma más eficiente de hacer un programa.

Es comúnmente aceptado que los programadores con muchos años en el desarrollo de códigos, convergen todos a una metodología de programación muy similar. Sin embargo, el precio que tienen que pagar es alto. Muchos códigos desarrollados, muchas horas perdidas y una relación de eficiencia baja. Por esta razón se hace necesario que exista una metodología de programación específica para cada proyecto software. De esta forma, los programadores inexpertos guiados por una metodología, no se chocarán con tantas paredes como el que tuvo que aprender a fuerza de golpes: algo muy común en la programación. Por otra parte, los rendimientos que se consiguen desarrollando con metodología de programación son mucho más elevados y el producto final goza de una vida útil mucho más larga.

Es importante hacer notar que el perfecto conocimiento del lenguaje de programación es condición necesaria pero no suficiente para la correcta implementación de un proyecto software. El resto de conocimientos y aspectos teóricos de la programación deben estar contenidos en una metodología de programación que es necesario aprender y aplicar permanentemente. Atendiendo a la metodología de programación, podemos clasificar la programación en:

- Programación anárquica.
- Programación estructurada.
  - estructurada plana.
  - estructurada encapsulada.
- Programación multicapa.

La programación *anárquica* se identifica por un conjunto de líneas de código escritas sin orden ni metodología alguna. El programador posee solamente conocimientos mínimos del lenguaje de programación y todo está permitido, incluso el uso de la sentencia GO TO. Seguir el flujo de datos en este tipo de programación es como seguir la pista de un espagueti en un plato de pasta.

La programación *estructurada plana* se identifica por un conjunto de líneas de código escritas con orden y metodología, aunque no existe una división de tareas. No existe una estructura jerárquica que organice y aísle las diferentes tareas. Una misma unidad de programa puede contener más de una tarea. Las estructuras básicas que se deben utilizar son: el bucle repetitivo, la construcción **if-then-else** y las asignaciones de cadenas de operaciones. A partir de éstas, se pueden generar estructuras más complicadas. Dichas estructuras deben ser construidas de forma tal que puedan ser tratadas como verdaderas cajas negras. La comunicación se realiza, generalmente, a través de un diccionario de datos ("datapool") y no a través de los argumentos de los procedimientos. Para poder implementar esta metodología el programador debe poseer conocimientos básicos de programación (capítulos 3 y 4).

La programación *estructurada encapsulada* se identifica por un conjunto de líneas de código escritas con orden y metodología donde existe una división de tareas. Las funciones grandes se subdividen en subfunciones pero no existe una estructura jerárquica que las organice. Podemos entender este estilo de programación como una nube de procedimientos conectados entre sí sin estructura funcional alguna. El programador, que debe poseer conocimientos medios o elevados de programación (capítulos 3–6), utiliza las mismas estructuras básicas que en la programación estructurada plana e incorpora la división de tareas en distintos procedimietos con argumentos.

Las diferencias entre la metodología plana y encapsulada residen principalmente en el flujo de datos. En la programación estructurada plana los procedimientos, generalmente, no tienen argumentos y constituyen un conjunto de sentencias que realizan un determinado proceso. Puesto que el procedimiento actúa sobre variables externas, se produce una abstracción del proceso pero no del dato. En la programación estructurada encapsulada los procedimientos tienen argumentos de entrada y de salida y se encapsulan las funciones en procedimientos. Como el procedimiento actúa sobre los argumentos de entrada, se consigue una abstracción

del proceso y también del dato. En efecto, los argumentos de entrada no tienen existencia como variables hasta que no se realiza la llamada al procedimiento. De esta forma, podemos clasificar los procedimientos en dos categorías límites:

1. Aquellos en los que sus entradas y salidas son variables externas.
2. Aquellos en los que sus entradas y salidas son argumentos.

Mientras que en los primeros el flujo de datos está enmascarado por los diferentes procedimientos, en los segundos el flujo de datos viene determinado de forma muy clara por la estructura funcional del código. Por ejemplo, si la velocidad de un fluido es una variable externa que se modifica por varios procedimientos, seguir la pista de su evolución requiere el uso de diferentes herramientas específicas que analicen la secuencia de sus modificaciones.

Por último, la programación multicapa es con mucho la metodología más eficiente en los códigos de simulación y será el objeto de análisis de los próximos apartados. En esta metodología existe una estructura funcional entre las diferentes unidades de programa y procedimientos.

7.3. Implementación

La *implementación* es la forma de poner en práctica un algoritmo de acuerdo a una determinada metodología. Las metodologías más evolucionadas permiten mayor libertad de implementación. Existen tres formas en las que se puede llevar a cabo la fase de implementación en un proyecto software:

- Implementación de arriba hacia abajo ("top-down").
- Implementación por prototipos.
- Procedimiento híbrido.

La implementación *de arriba hacia abajo* consiste en partir de un conjunto de especificaciones muy estrictas definidas en la fase de diseño. En dichas especificaciones las funciones del programa son clasificadas jerárquicamente en funciones ejecutivas o de alto nivel y en funciones físico-matemáticas o de bajo nivel. La implementación "top-downçomienza codificando los módulos de las funciones ejecutivas; luego se baja en el nivel de jerarquía y se codifican los módulos correspondientes a las funciones físico-matemáticas. Esta forma de implementar exige respetar estrictamente las especificaciones iniciales y codificar todo el software de

una vez sin realizar pruebas intermedias de funcionamiento. Este método tiene el inconveniente de que no se tiene información de cómo funciona el programa hasta el final de la codificación. Además, resulta evidente que las especificaciones deben ser estudiadas y definidas con mucha precisión, lo cual en la práctica es muy difícil de lograr. Es casi imposible conocer de antemano todas las funciones que nuestro código tendrá que contener.

La implementación por *prototipos* parte de un conjunto de especificaciones más generales y poco estrictas. De esta forma, se comienza codificando cualquiera de los módulos en base a prototipos que define el programador inicialmente. A lo largo de esta fase los prototipos se codifican y, casi simultáneamente, se realizan sobre ellos pruebas de funcionamiento. En caso de errores los prototipos se corrigen sobre la marcha hasta que se llega a la solución. Para las pruebas de funcionamiento, y en el caso que nuestro módulo requiera de otros prototipos que aún no han sido implementados, es necesaria la construcción de emuladores que faciliten las funciones de éstos. Con este procedimiento, las especificaciones se corrigen y amplían durante la fase de implementación.

El procedimiento *híbrido* consiste en combinar los dos procedimientos anteriores. Se parte de un conjunto de especificaciones lo más riguroso posible. Se codifica inicialmente usando la implementación "top-down"para mantener un cierto orden, aunque cada módulo puede ser codificado usando prototipos y admitiendo simultáneamente pruebas de funcionamiento con emuladores. Dependiendo de la metodología de programación usada, este procedimiento de implementación puede ser el más apropiado.

Las herramientas más significativas de la fase de implementación son las siguientes:

- Editores sensibles al lenguaje de programación.
- Sistemas de compilación.
- Analizador estático de código.

Los *editores sensibles al lenguaje de programación* son aplicaciones específicas para cada lenguaje que permiten visualizar y compilar un código, visualizar un diagnóstico de mensajes, alarmas y errores y corregir de forma ágil los errores; todo dentro de una única sesión de edición. La visualización de código se puede facilitar utilizando distintos colores para identificar palabras clave del lenguaje, líneas con comentarios y líneas con errores de compilación.

Los *sistemas de compilación* automatizan y simplifican el proceso de *construcción* de códigos de simulación complejos. Esta herramienta accede a los ficheros

fuentes en la biblioteca en su versión actualizada, compila cada unidad de programa y realiza una secuencia de dependencias entre los ficheros fuentes para construir de forma automática un *ejecutable*. Esta herramienta permite reconstruir una biblioteca de unidades de programa usando solamente aquellas unidades que han sido modificadas posteriormente a la anterior construcción.

El *analizador estático de código* es una herramienta que permite realizar una *referencia cruzada* entre distintas unidades de programa. De esta forma el programador puede localizar unidades de programa, variables y datos dentro de un conjunto grande y complejo de líneas de código, y responder a preguntas tales como a qué unidad de programa corresponde la variable `alpha`, dónde es llamado el procedimiento `RK_4` o a qué procedimientos llama la subrutina `discret_espacial`. Esta herramienta incluye utilidades gráficas para visualizar estructuras o árboles de llamadas y extraer información de diseño. Otra tarea importante del analizador estático de código es identificar y trabajar con un diccionario de variables externas.

Otras utilidades de un entorno de desarrollo son un sistema de comparación de código fuente para detectar modificaciones entre distintas versiones de una unidad de programa, utilidades para localizar, copiar y visualizar código, y sistemas de transferencia de información como unidades de cinta y sistemas de impresión.

7.4. Programación multicapa

La programación multicapa se identifica por un conjunto de líneas de código escritas con una metodología donde existe una división de funciones grandes en subfunciones. Estas funciones se estructuran de forma piramidal y es en esta estructura donde se definen capas o niveles de abstracción.

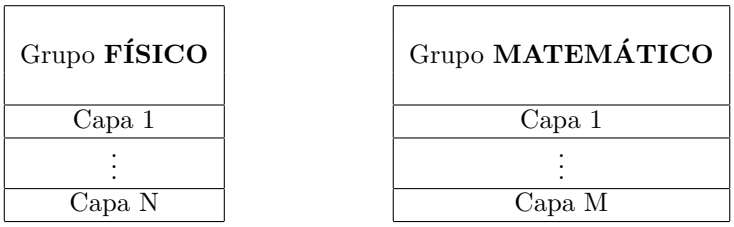
El enunciado del problema que queremos resolver nos permite determinar las funciones u operaciones de máximo nivel de abstracción. La capa de aplicación o ejecutivo es una secuencia de las operaciones anteriormente identificadas que, generalmente, se repiten de forma cíclica. La clasificación de estas funciones en físicas y matemáticas permite definir el *grupo físico* y el *grupo matemático*.

Las diferentes funciones que constituyen un grupo se jerarquizan en *capas* o niveles de abstracción. A su vez, cada capa puede estar compuesta por diferentes abstracciones. Así, la capa i representa un mayor nivel de abstracción que la capa $i+1$. Por lo tanto, las abstracciones de la capa $i+1$ son más elementales que las abstracciones de la capa i.

El grupo físico estará formado por las funciones que tengan que ver con el modelado del sistema, tales como las ecuaciones, condiciones de contorno y condiciones iniciales. Las funciones del grupo físico se basan o apoyan en abstracciones matemáticas o en abstracciones físicas más elementales. Es decir, la determinación de los diferentes términos de las ecuaciones se construye o bien con abstracciones matemáticas, o bien con abstracciones físicas más elementales.

El grupo matemático estará formado por las funciones que tengan que ver con operadores matemáticos, tales como funciones que resuelven sistemas de ecuaciones lineales o no lineales, y operadores en diferencias que aproximan operadores diferenciales. Las funciones del grupo matemático se basan o apoyan en abstracciones matemáticas más elementales. Es importante hacer notar que en ningún momento las funciones del grupo matemático se basan o apoyan en abstracciones físicas. Es decir, la funcionalidad de los operadores matemáticos se construye mediante abstracciones matemáticas más elementales y no mediante abstracciones físicas.

Todo código de simulación se puede dividir en los siguientes grupos:



Para poder implementar esta metodología, es necesario un conocimiento profundo del lenguaje de programación. En concreto, la determinación de los canales de comunicación o interfaces entre capas, las llamadas a procedimientos con matrices que cambian su formato en la llamada y la creación de niveles de abstracción son algunas de las tareas más difíciles.

La metodología multicapa está muy ligada a la implementación "top–down". A continuación se presentan los conceptos que fundamentan el estilo de programación multicapa, dividiendo el conjunto de normas en las siguientes partes que facilitan una clara implementación "top–down":

1. Definición de grupos.
    a) Identificar las funciones físicas y matemáticas necesarias del código de simulación.

b) Crear el grupo físico y el grupo matemático con las funciones anteriormente identificadas clasificadas por su carácter físico o su crácter matemático.

2. Grupo matemático y estructura multicapa.

   a) A partir de las funciones de alto nivel de abstracción, identificar las funciones matemáticas de más bajo nivel. Por ejemplo, una interpolación polinómica bidimensional se puede basar en una interpolación polinómica unidimensional.

   b) Agrupar las funciones matemáticas en capas estableciendo una jerarquía funcional asociada al nivel de abstracción. Es decir, el nivel de abstracción de una función matemática de la capa i se basa en niveles de abstracción más elementales de la capa $i + 1$.

   c) Construir los procedimientos generalizando suficientemente la función como para poder reutilizar código. Cada procedimiento debe constituir una única función u operador matemático.

   d) Las capas matemáticas deben estar ligadas a funciones u operaciones bien definidas dentro de las matemáticas. El conocimiento del análisis matemático y de los métodos matemáticos ayudarán al programador a identificar las capas. Generalmente, en simulación trabajamos en espacios vectoriales de dimensión finita donde se definen operadores en diferencias sobre los elementos del espacio vectorial. Por ejemplo, un producto vectorial en $\mathbb{R}^3$, un operador en diferencias que resulta de discretizar el operador diferencial laplaciana, etc.

3. Grupo físico y estructura multicapa.

   a) Si la física que queremos simular está formada por diferentes medios con diferentes modelos o ecuaciones matemáticas, el grupo físico puede estar formado por un conjunto de elementos o unidades de programa que se corresponden con cada medio o modelo físico.

   b) El grupo físico o la formulación del problema, a diferencia de lo que ocurre en el grupo matemático, puede estar basado en abstracciones físicas o matemáticas. En concreto, las ecuaciones diferenciales están formadas por un conjunto de términos que, generalmente, son funciones u operadores matemáticos de alto nivel de abstracción que actúan sobre campos escalares o vectoriales. Sin embargo, un término de una ecuación puede estar determinado por el comportamiento físico de otro sistema (problemas termoelásticos, problemas de interacción fluido–estructura).

4. Interfaces de los procedimientos, funciones u operadores.

   a) Las definiciones y las especificaciones de los interfaces de las nuevas operaciones o procedimientos de cada capa deben agruparse en unidades de programa **module**. De esta forma, aunque el procedimiento

queda oculto en su unidad de programa, los interfaces de comunicación entre procedimientos son conocidos por todas las unidades de programa o procedimientos que se asocian a ésta mediante la sentencia **use**. Se pueden crear procedmientos de mayor nivel de abstracción conociendo exclusivamente los interfaces de los de menor nivel sin necesidad de conocer su algoritmo o proceso.

b) Los argumentos de los procedimientos deberán ser de entrada o salida y no de entrada y salida salvo que esté estrictamente justificado.

c) La especificación de los argumentos matrices o vectores puede ser importada o explícita. Si la especificación es importada, las especificaciones de los argumentos y las variables asociadas en la llamada deben coincidir.

Sólo mediante una especificación explícita, las dimensiones de los argumentos y las variables asociadas en la llamada pueden diferir. Esto puede ser útil cuando sea necesario reformatear las dimensiones de variables matrices o vectores. Es decir, la variable con que se llama a un procedimiento puede ser un vector y, sin embargo, el procedimiento internamente trabaja con una matriz de dos índices. El reformateo de variables o datos entre capas permite trabajar con comodidad dentro de cada capa o función. Por ejemplo, un procedimiento puede integrar en el tiempo un vector que, en el grupo físico, está formado por las componentes ordenadas de una matriz de tres índices o un campo escalar de $\mathbb{R}^3$.

d) Si los argumentos son procedimientos, su interfaz debe ser especificada explícitamente.

5. Variables externas a un procedimiento.

a) Las variables especificadas en una unidad **module** se pueden usar por cualquier procedimiento mediante la sentencia **use**, prestando atención a que su empleo esté en concordancia con la estructura funcional multicapa. Por ejemplo, los procedimientos del grupo matemático no deben utilizar variables del grupo físico. Los niveles de abstracción más elementales no deben usar variables de los niveles de abstracción más complejos.

b) Las variables especificadas en una unidad de programa **module** pueden ser usadas por todos los procedimientos que contiene esta unidad de programa o abstracción.

c) El caso más claro de variables externas lo constituyen datos de partida o constantes del problema que deben ser inicializadas, si es posible, mediante la propia inicialización de su especificación. Si los valores de estas supuestas constantes, especificadas como variables externas, se modifican a lo largo de la ejecución de un programa, la funcionalidad de los procesos cambia a lo largo de la ejecución del programa. Por

ejemplo, si el número π cambia de valor, la funcionalidad de una subrutina que calcule el área de un círculo mediante la variable externa π se modificará.

6. Especificación de variables y nuevos tipos

   a) Especificar siempre todas las variables.

   b) Asignar memoria dinámicamente en función del tamaño del problema.

   c) Las definiciones de nuevos tipos o estructuras de datos y nuevas operaciones comunes a una capa deben estar confinadas en una unidad de programa **module**. De esta forma, estos nuevos tipos se pueden considerar como tipos externos para las funciones que los necesiten sin necesidad de redefinirlos.

7.5. Ejemplos de programación

Para poner de manifiesto las ventajas e inconvenientes de las diferentes metodologías de programación, vamos a codificar la integración de una ecuación en derivadas parciales mediante un programa con estructura plana, encapsulada y multicapa. Consideramos la siguiente ecuación en derivadas parciales:

$$\frac{\partial u}{\partial t} + u_0 \frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2}$$

con condiciones de contorno periódicas entre $x = 0$ y $x = 1$ y con la condición inicial $u(x, 0) = sen(2\pi x)$. El esquema numérico que vamos a utilizar para la discretización espacial es el siguiente:

$$\frac{\partial u}{\partial x}(x_j, t_n) = \frac{u^n_{j+1} - u^n_{j-1}}{2\Delta x},$$

$$\frac{\partial^2 u}{\partial x^2}(x_j, t_n) = \frac{u^n_{j+1} - 2u^n_j + u^n_{j-1}}{(\Delta x)^2}.$$

Este esquema constituye una aproximación mediante diferencias finitas centradas de segundo orden para la derivada primera y la derivada segunda. u^n_j representa el valor de la función u en el instante t_n y en la posición x_j. Para el integrador temporal, utilizaremos un Adams–Bashforth de segundo orden cuyo esquema numérico es el siguiente:

$$u^{n+1} = u^n + \frac{3\Delta t}{2} f^n - \frac{\Delta t}{2} f^{n-1},$$

donde Δt es el paso temporal en la integración.

7.5.1. Programación plana

En el siguiente programa integramos la ecuación en derivadas parciales mediante un programa con estructura plana.

```
      program EDPS  !  Integracion EDP mediante estructura plana

!     *** declaracion de variables
          integer, parameter :: nmax=1000
          real  :: dt, x_j, pi, dx, u(nmax), f(nmax), uo(nmax)
          real  :: nu = 0.2e0, u0 = 1e0
          integer ::  nt, nx, i, j

!     *** paso temporal, malla espacial, pasos temporales
          nx = 150
          dx = 1e0/(nx-1)
          dt = 1e0/( (nx-1)*(nx-1) )
          nt = 1e0/dt

!     *** condicion inicial
          pi = 4e0 * atan(1e0)
          do j=1, nx
              x_j = (j-1)*dx
              u(j) = sin( 2e0*pi*x_j)
          enddo


!     *** el primer paso temporal lo damos con un Euler
          i=1
!         *** condiciones de contorno
              u(1)  = 1e0/2e0 * ( u(2) + u(nx-1) )
              u(nx) = 1e0/2e0 * ( u(2) + u(nx-1) )
              f(1)  = 0e0
              f(nx) = 0e0

!         *** calcula las derivadas parciales en x
              do j=2, nx-1
                f(j) =    (nu/(dx*dx) - u0/(2e0*dx))*u(j+1) +  &
                          (nu/(dx*dx) + u0/(2e0*dx))*u(j-1)    &
                           - 2e0*nu/(dx*dx) * u(j)
              enddo

!         *** actualiza la funcion y guarda el valor anterior
              do j=1, nx
                  uo(j) = u(j)
                  u(j)  = u(j) + dt * f(j)
              enddo
```

```
!     *** bucle para la integracion temporal (Adams-Bashforth)
          do i=2, nt

!         *** condiciones de contorno
              u(1)  = 1e0/2e0 * ( u(2) + u(nx-1) )
              u(nx) = 1e0/2e0 * ( u(2) + u(nx-1) )

!         *** calcula las derivadas parciales en x
              do j=2, nx-1
                f(j) =  3e0/2e0*(                                  &
                           (nu/(dx*dx) - u0/(2e0*dx))*u(j+1) +     &
                           (nu/(dx*dx) + u0/(2e0*dx))*u(j-1)       &
                            - 2e0*nu/(dx*dx) * u(j)                &
                               )                                   &
                      -1e0/2e0 *(                                  &
                           (nu/(dx*dx) - u0/(2e0*dx))*uo(j+1) +    &
                           (nu/(dx*dx) + u0/(2e0*dx))*uo(j-1)      &
                            - 2e0*nu/(dx*dx) * uo(j)               &
                               )
              enddo


!         *** actualiza la funcion y guarda el valor anterior
              do j=2, nx-1
                uo(j) = u(j)
                u(j)  = u(j) + dt * f(j)
              enddo
              uo(1)  = u(1)
              uo(nx) = u(nx)

          enddo

!         *** imprime resultados para i=nt
              write(*,*) 'Paso nt=', nt, ' Instante t=', dt*i
              do j=1, nx
                  write(*,*) ' j, u(j)=', j, u(j)
              enddo

      end program EDPS
```

El programa es limpio, está indentado, separadas las funciones en bloques y, en resumen, está bien escrito. Sin embargo, las funciones más generales no están encapsuladas, lo que origina los dos inconvenientes principales de la metodología plana: *(i)* la imposibilidad de reutilizar funciones o procedimientos del programa para otro problema diferente, y *(ii)* la necesidad de validar el programa en su conjunto. No se puede asegurar ni probar el correcto funcionamiento de partes del programa.

7.5.2. Programación encapsulada

El mismo problema anterior se podría haber codificado mediante un programa en el que las funciones básicas se encapsulen y exista flujo de datos entre unidades de programa. En este caso el programa principal se convierte en un ejecutivo que llama a las diferentes subrutinas que ocultan un procedimiento.

```
      program EDPS    ! progamacion encapsulada

!     *** procedimientos externos
          use procedimientos

!     *** declaracion de variables
          integer, parameter :: nmax=1000
          real :: t, dt, u(nmax), f(nmax)
          integer ::  nt, ns, i, j

!     *** Condicion inicial, pasos y puntos en x
          call CI_edp (nt, ns, u, dt)

!     *** bucle para la integracion temporal
          t = 0
          do i=1, nt


!         *** pasos temporales con Adams-Bashforth
              call AdamsB2  (Sistema_EDP, t, dt, ns, u, f)

          enddo

!         *** imprime resultados para i=nt
              open(2,file='edps.txt')
              write(*,*) 'Paso nt=', nt, ' Instante t=', dt*i
              do j=1, ns
                   write(*,*) j, u(j)
                   write(2,*) j, u(j)
              enddo

      end program EDPS
```

La primera sentencia del programa es una asociación a la unidad de programa `module procedimientos` que contine todas las subrutinas que se utilizan en el programa (`CI_edp`, `AdamsB2`, `Sistema_EDP`). Después de la especificación de variables, el programa principal calcula la condición inicial mediante la subrutina:

```
      subroutine CI_edp (nt, ns, u, dt)

            use fisico_num   ! especificaciones externas

            integer, intent(out) ::  nt   ! saltos temporales
            integer, intent(out) ::  ns   ! puntos en x
            real   , intent(out) ::  dt   ! paso temporal
            real   , intent(out) ::  u(:) ! condicion inicial u(x,0)

          real  :: x_j, pi, tf
          integer ::  j


!     *** parametros fisicos
          u0 = 1e0; nu = 0.1e0; pi = 4e0*atan(1e0)

!     *** parametros numericos espaciales
          nx = 150;  dx = 1e0/(nx-1)

!     *** parametros numericos temporales
          dt = dx**2; tf = 0.25e0; nt = tf/dt; dt = tf/nt

!     *** numero de ecuaciones
          ns = nx

!     *** condicion inicial
          do j=1, nx
              x_j  = dx*(j-1)
              u(j) = sin (2*pi*x_j)
          enddo

      end subroutine
```

En esta subrutina se especifican las variables externas mediante la sentencia:

```
          use fisico_num
```

El nombre `fisico_num` es la siguiente unidad de programa en donde se crean variables físicas y numéricas.

```
module fisico_num   ! especificación de variables externas

     real  ::  u0   ! velocidad
     real  ::  nu   ! viscosidad
     real  ::  nx   ! numero de puntos en x
     real  ::  dx   ! paso de la malla espacial

end  module
```

De esta forma, quedan encapsuladas la condición inicial y la inicialización de parámetros. Después, en el programa principal los pasos temporales se dan mediante la llamada al integrador temporal:

```
subroutine AdamsB2(Sistema, t, dt, n, u, F) ! Esquema Adams-Bashforth

     interface                    ! interfaz de  F(U,t)
          subroutine Sistema(U,t,F)
                    real, intent(in)  ::  U(:), t
                    real, intent(out) ::  F(:)
          end subroutine
     end interface

     integer, intent(in) ::   n    !  dimension del sistema
     real,    intent(in) ::   t    !  tiempo actual
     real,    intent(in) ::  dt    !  paso temporal
     real,    intent(inout):: U(n) !  vector de estado u(t)
     real,    intent(out)::  F(n)  !  derivada de u(t) en t y t-dt

     real, save, allocatable :: F2(:)

    call Sistema(U, t, F)              ! sistema de ecuaciones

    if (t == 0e0 ) then
                         U = U + dt * F  ! primer paso con Euler
                         allocate( F2(n) )
                    else
                         U = U + dt/2*( 3e0 * F - F2 )
                    endif
    F2 = F

    t = t + dt

end subroutine
```

Esta subrutina implementa un esquema Adams–Bashforth de segundo orden y paso de tiempo constante. Es importante hacer notar que uno de los argumentos de entrada es una subrutina donde se deben calcular las derivadas del vector de

estado. Por último, el operador espacial discreto que calcula las derivadas del vector de estado está encapsulado en:

```
subroutine Sistema_EDP (u, t, f)

  use fisico_num             ! especificaciones externas

  real, intent(in) ::  u(:)  ! valores discretos  u(x_j, t_n)
  real, intent(in) ::  t     ! tiempo actual
  real, intent(out)::  f(:)  ! operador espacial discreto f(x_j, t_n)

       integer ::  j

!     *** condiciones de contorno
       u(1)  = 0.5e0 * ( u(2) + u(nx-1) )
       u(nx) = 0.5e0 * ( u(2) + u(nx-1) )

!     *** operador espacial discreto
       dx = 1e0/(nx-1)
       do j=2, nx-1
          f(j) =  - u0/(2e0*dx) * ( u(j+1) - u(j-1) )           &
                  + nu/(dx*dx)  * ( u(j+1) - 2e0*u(j) + u(j-1) )
       enddo

  end subroutine
```

7.5.3. Programación multicapa

El mismo problema podría haber sido implementado con metodología multicapa. En este ejemplo presentamos un código multicapa analizando el cumplimiento de los diferentes puntos de la metodología.

En este caso la aplicación es desarrollar un programa que integre un problema de evolución, concretamente la ecuación de Burgers con condiciones iniciales y de contorno, mediante un esquema Adams–Bashforth de paso de tiempo constante. El análisis del enunciado nos permite identificar las siguientes funciones: integrador del problema de evolución, modelo de la ecuación de Burgers con sus condiciones iniciales y de contorno, esquema temporal Adams-Bashforth y unidad que determina el paso de tiempo constante. Podemos definir el grupo matemático formado por los elementos: integrador del problema de evolución y esquema temporal Adams–Bashforth. El grupo físico estará formado por la ecuación semidiscretizada espacialmente con las condiciones de contorno y las condiciones iniciales. El programa principal se puede escribir como:

```
program Simula

  use Problema_Cauchy          ! grupo matematico
  use Esquemas_temporales      ! grupo matematico
  use Problema_Burgers         ! grupo fisico


  call Problema_Evolucion( Sistema = Burgers,              &
                           Condicion_Inicial = CI_Burgers, &
                           Paso_temporal = dt_Burgers,     &
                           Esquema_temporal = AdamsB2       )

end program Simula
```

El programa principal `Simula` se basa en las abstracciones:

`Problema_Cauchy` `Esquemas_temporales` `Problema_Burgers`

que son unidades de programa **module**. El módulo `Problema_Cauchy` contiene la subrutina `Problema_Evolucion` que integra el siguiente problema de Cauchy:

$$\frac{dU}{dt} = F(U, t; p_1, \ldots p_n), \qquad U(0) = U^0$$

donde t es la variable independiente, $p_1, \ldots p_n$ son los parámetros del problema y U^0 es la condición inicial.

```
module Problema_Cauchy
contains

subroutine Problema_Evolucion( Sistema, Condicion_Inicial,  &
                               Paso_temporal, Esquema_temporal )

     interface    !  especificacion de argumentos que son procedimentos

        subroutine  Sistema(U, t, F)            ! Calcula F(U,t)
              real, intent(in)  :: U(:), t
              real, intent(out) :: F(:)
        end subroutine Sistema

        subroutine  Condicion_Inicial(tf, U, F) ! condicion inicial
                  real, intent(in) :: tf
                  real, pointer    :: U(:), F(:)
        end subroutine Condicion_Inicial
```

```
          subroutine Paso_temporal(Sistema, t, dt, U, F) ! Salidas y dt
                 interface
                      subroutine Sistema(U, t, F)
                           real, intent(in)  :: U(:), t
                           real, intent(out) :: F(:)
                      end subroutine Sistema
                 end interface
                 real, intent(in) :: t, U(:), F(:)
                 real, intent(out) :: dt
          end subroutine Paso_temporal


          subroutine Esquema_temporal(Sistema, t, dt, U, F)
                  interface
                        subroutine Sistema(U, t, F)  ! Calcula F(U,t)
                             real, intent(in)  :: U(:), t
                             real, intent(out) :: F(:)
                        end subroutine Sistema
                  end interface
                  real, intent(in)    :: dt
                  real, intent(out)   :: t, F(:)
                  real, intent(inout) :: U(:)
          end subroutine Esquema_temporal

        end interface

!     *** especificacion de  variables
          real ::  tf, dt, t
          real, pointer :: U(:), F(:)

!     *** calculo de la condicion inicial
          call Condicion_Inicial ( tf, U, F )

!     *** bucle para la integracion temporal
          t = 0e0
          do while( t < tf )
             call Paso_temporal    ( Sistema, t, dt, U, F )
             call Esquema_temporal ( Sistema, t, dt, U, F )
          enddo

   end subroutine
end module
```

Un esquema temporal explícito para obtener la solución U^{n+1} en el instante t_{n+1} a partir del instante t_n se puede escribir de forma general como:

$$U^{n+1} = G(U^n, U^{n-1}, \ldots, \Delta t, F)$$

donde G representa el esquema temporal que se formula en función de pasos anteriores y de los valores que toma la función del problema de Cauchy F en di-

ferentes valores. En concreto, para escribir una subrutina que implemente un esquema Adams–Bashforth, necesitamos conocer exclusivamente la especificación de sus argumentos y nada de la física que queramos resolver. La unidad `Esquemas_temporales` contiene la subrutina `AdamsB2` que implementa el esquema Adams–Bashforth.

```
module Esquemas_temporales
contains

    subroutine AdamsB2(Sistema, t, dt, U, F)

           interface
                 subroutine Sistema(U, t, F)  ! Calcula F(U,t)
                      real, intent(in)  :: U(:), t
                      real, intent(out) :: F(:)
                end subroutine Sistema
           end interface
           real, intent(in)    :: dt
           real, intent(out)   :: t, F(:)
           real, intent(inout) :: U(:)

        integer :: n                      ! dimension del problema
        real, save, allocatable :: F2(:)  ! F( U^{n-1}, t_{n-1} )


        call Sistema(U, t, F)

 !      El primer paso esquema  Euler
        if (t == 0e0 ) then

                 n = size(U)
                 U = U + dt * F
                 allocate( F2(n) )

 !      los demas pasos esquema Adams-Bashforth
        else

                 U = U + dt/2 * ( 3 * F - F2 )
        endif

        F2 = F           !  F^{n-1} <-  F^n

        t = t + dt

    end subroutine

end module
```

Nos falta por último definir la abstracción física `Problema_Burgers`.

```
module Problema_Burgers


              integer ::  nx             ! numero de puntos en x
              real :: dx

              real, parameter:: nu=0.1   ! parametros fisicos
              real, parameter:: u0=1.0

contains


    subroutine Burgers (U, t, F)            ! F del problema de Cauchy

          real, intent(in) :: U(:), t
          real, intent(out):: F(:)

        integer ::  j

        U(1) =  0.5 * ( U(2) + U(nx-1) );  F(1)  = 0.0  ! cc contorno
        U(nx) = 0.5 * ( U(2) + U(nx-1) );  F(nx) = 0.0


        do j=2, nx-1
              F(j) =  - u0/(2*dx) * ( U(j+1) - U(j-1) ) &
                      + nu/dx**2  * ( U(j+1) - 2*U(j) + U(j-1) )
        enddo

    end subroutine


    subroutine CI_Burgers (tf, U, F)       ! Condicion incial

          real, intent(out) :: tf
          real, pointer :: U(:), F(:)

         real :: Pi
         Pi = 4*atan(1.0)

         tf = 1.0                      ! tiempo total integracion
         nx = 100 ;  dx = 1.0/(nx-1)   ! malla
         allocate( U(nx), F(nx) )      ! asignacion memoria

         forall(j=1:nx)  U(j) = sin ( 2* Pi * dx * (j-1)  )

    end subroutine
```

```
        subroutine dt_burgers (Sistema, t, dt, U, F)

             interface
                   subroutine Sistema(U, t, F)  ! Calcula F(U,t)
                        real, intent(in)  :: U(:), t
                        real, intent(out) :: F(:)
                  end subroutine Sistema
            end interface
            real, intent(in) t, U(:), F(:)
            real, intent(out) dt

               integer ::  j
               integer, save:: paso=0

            dt = 0.2*dx**2/nu             ! paso temporal
            paso = paso + 1

            if (mod(paso,10)==0) then     ! guarda datos cada 10 pasos

                 open(2, file='pepe.dat')
                 do j = 1, nx
                   write(2,*)  dx*(j-1), U(j)
                 enddo
                 close(2)
                 write(*,*) ' tiempo  = ', t
                 read(*,*)

            endif

        end subroutine

    end module
```

Esta abstracción de nuevo es un módulo donde el número de puntos de la discretización espacial `nx`, el paso espacial `dx` y los parámetros físicos del problema se especifican como variables externas a todas las unidades de `Problema_Burgers`. Este módulo contiene la determinación de la función $F(U, t; p_1, \ldots p_n)$ del problema de Cauchy oculta en `Burgers`, el valor de la condición inicial U^0 oculta `CI_Burgers` y la determinación del paso temporal calculado `dt_Burgers`. La subrutina `Burgers` realiza la semidiscretización espacial. En `CI_Burgers` se decide la dimensión del problema, el tiempo final de integración `tf`, se asigna memoria dinámicamente y se da la condición inicial. Y `dt_Burgers` además de determinar el paso temporal, y puesto que es llamada en cada paso de integración, se utiliza para escribir en disco los resultados de la integración.

Como las subrutinas están encapsuladas en unidades de programa **module**, en todo punto en donde se realice una asociación a esa abstracción mediante la sentencia **use**, los interfaces serán explícitos sin necesidad de reescribirlos. Única-

mente, es necesario hacer una especificación de interfaz para los argumentos que son procedimientos.

Aunque los tres programas desarrollados con las metodologías: plana, encapsulada y multicapa obtienen los mismos resultados, existen diferencias desde el punto de vista de la validación, vida útil del programa y reutilidad.

Como puede comprobar el lector, el primer programa escrito con una metodología plana requiere menos conocimiento del lenguaje de programación para su implementación, lo cual constituye una ventaja para una iniciación en la programación. Por otra parte su extensión es menor, lo cual en principio parece otra ventaja. Como inconvenientes podemos señalar que la discretización espacial, la física y la discretización temporal están en el mismo sitio y es difícil –a veces imposible– conocer el integrador utilizado o las ecuaciones que se integran. Por otra parte, al no existir abstracción no se puede reutilizar código para futuros proyectos.

Sin embargo, el segundo programa escrito con encapsulación supera con creces en claridad al primero. Con un simple vistazo el lector sabe dónde se calcula la condición inicial, qué tipo de integrador utiliza y cual es el sistema de ecuaciones a integrar. Es importante darse cuenta que en este segundo programa el integrador espacial está desacoplado del integrador espacial, y se puede modificar cualesquiera de los dos integradores sin tocar el sistema de ecuaciones o el resto del programa. Este programa tiene un mantenimiento mucho menos laborioso incluso para la misma persona que lo ha desarrollado.

Por último, el tercer programa escrito con metodología multicapa basada en niveles de abstracción requiere un conocimiento profundo del lenguaje. Aunque en principio el programa parece más largo, la integración del problema de Cauchy y los diferentes esquemas temporales son abstracciones que una vez validadas no se vuelven a tocar durante muchos años. El programador, en un proyecto software de simulación de un problema de evolución, sólo tendrá que realizar esfuerzo en el desarrollo del grupo físico. Por otra parte, si el proyecto hiciera énfasis en diferentes esquemas para la discretización espacial, la discretización de las derivadas podría pertenecer a un nivel de abstracción del grupo matemático y dejar la semidiscretización espacial desacoplada de la física del problema. Por ejemplo, la unidad `Burgers` podría escribrise de la siguiente forma:

```
subroutine Burgers (U, t, F)             ! F del problema de Cauchy
       real, intent(inout) U(:)
       real, intent(in) ::  t
       real, intent(out):: F(:)

      U(1) =  0.5 * ( U(2) + U(nx-1) )   ! cc contorno
      U(nx) = 0.5 * ( U(2) + U(nx-1) )
      call Burgersh(nx, U, F, F(2*nx+1:3*nx), F(3*nx+1:4*nx) )

end subroutine

subroutine Burgersh (n, U, F, DU, D2U)
      integer, intent(in) ::  n
      real, intent(in) :: U(:)
      real, intent(out):: F(:), DU(:), D2U(:)

     call Deriva ( n, U, DU, D2U )   ! calcula Ux y Uxx
     F =  - u0 * DU + nu * D2U       ! ecuacion de Burgers

end subroutine
```

donde la unidad de programa `Deriva` pertenece al grupo matemático que calcula mediante diferencias finitas la derivada primera y segunda.

```
subroutine Deriva (n, U, DU, D2U)
            integer, intent(in) ::  n
            real, intent(in)  :: U(:)
            real, intent(out) :: DU(:), D2U(:)

  integer ::  j
  real :: dx
  dx = 1e0/n

   do j=1, n
     if (j==1) then
                     DU(1)  = ( -3*U(1) + 4*U(2) - U(3) )/(2*dx)
                     D2U(1) = ( U(1) - 2*U(2) + U(3) )/dx**2
     elseif (j==n) then
                     DU(n) = ( 3*U(n) - 4*U(n-1) + U(n-2) )/(2*dx)
                     D2U(n) = ( U(n) - 2*U(n-1) + U(n-2) )/dx**2
     else
                     DU(j)  = ( U(j+1) - U(j-1) )/(2*dx)
                     D2U(j) = ( U(j+1) - 2*U(j) + U(j-1) )/dx**2
     endif
   enddo

end subroutine
```

De esta forma, cambiar la física, el esquema espacial o temporal consiste simplemente en establecer asociaciones entre argumentos y procedimientos externos.

Por ejemplo, para simular la oscilación de un péndulo con un esquema Adams–Bashforth, escribiríamos:

```
program Simula

   use Esquemas_temporales
   use Problema_Cauchy
   use Problema_Oscilador

   call Problema_Evolucion( Sistema = Oscilador,               &
                            Condicion_Inicial = CI_Oscilador,  &
                            Paso_temporal = dt_Oscilador,      &
                            Esquema_temporal = AdamsB2          )
end program Simula
```

En este caso las abstracciones del grupo matemático ya están escritas y validadas y sólo tenemos que escribir las ecuaciones del péndulo, sus parámetros y su condición inicial.

```
module Problema_Oscilador

 contains

    subroutine Oscilador (U, t, F)           !   y'' + y = 0
          real, intent(in)  :: U(:), t
          real, intent(out) :: F(:)

         F(1) =   U(2)          !   y  = U(1)
         F(2) = - U(1)          !   y' = U(2)

   end subroutine

   subroutine CI_Oscilador (tf,  U, F)
           real, intent(out) :: tf
           real, pointer :: U(:), F(:)

           tf = 200.0

           allocate( U(2), F(2) )  ! asignacion memoria

           U(1) = 0.0              ! condicion inicial
           U(2) = 1.0

   end subroutine
```

```
   subroutine dt_oscilador (Sistema, t, dt, U, F )

         interface
                 subroutine Sistema(U, t, F)
                       real, intent(in)  :: U(:), t
                       real, intent(out) :: F(:)
                end subroutine Sistema
         end interface
         real, intent(in) t, U(:), F(:)
         real, intent(out) dt

        integer , save :: paso = 0

        paso = paso + 1

        if (paso==1) then;  open(2, file='pepe.dat');  endif

        dt = 0.1                   ! paso de tiempo

        write(2,*)  U(1), U(2)     ! guarda la posicion y velocidad

   end subroutine

end module
```

7.6. Validación

La validación de códigos es una de las tareas más laboriosas de un proyecto software. Desde el punto de vista de la programación la validación se centra en la codificación y no en la validación de los modelos o algoritmos utilizados, lo cual es objeto de otra disciplina. Los recursos que se pueden utilizar para la validación son:

- Herramientas gráficas de validación.
- Analizador de prestaciones.

El *Analizador de prestaciones* ayuda al programador a examinar el comportamiento del código de simulación en tiempo de ejecución e identificar código no eficiente y cuellos de botella. Esta herramienta recoge datos y hace estadística durante el tiempo de ejecución. Además, permite generar histogramas y tablas de llamadas a partir de estos datos. También, genera información de tiempos de ejecución parciales y totales, memoria utilizada y contabiliza las operaciones de

entrada y de salida. Una función importante del analizador de prestaciones es localizar partes del código que no han sido ejecutadas durante las pruebas y por lo tanto no validadas.

Cuanto mejor esté escrito el código, más fácilmente y rápidamente se lleva a cabo la validación. En muchas ocasiones puede costar meses descubrir ciertos errores ("bugs") y a veces es necesario reescribir ciertas partes del código que puedan dar errores. Dependiendo de la complejidad del código, el esfuerzo de reescritura para eliminar errores fantasmas puede ser inferior al esfuerzo necesario para la modificación de códigos mal escritos.

Sin embargo, generalmente no es posible validar al cien por cien los diferentes resultados que puede dar un código de simulación. Cuando el código es grande y la simulación comprende una gran cantidad de funciones, resulta imposible diseñar un conjunto de pruebas de funcionamiento que asegure la validación de cada línea de código con todas sus posibles combinaciones. Dependiendo de la fiabilidad exigida al proyecto, la validación se hace con un determinado margen de confianza. Es decir, no se prueban los infinitos casos test que serían necesarios para asegurar un funcionamiento correcto del programa al cien por cien. Admitido que la tarea de validación es una tarea que se realiza tan solo en un tanto por ciento, es fácil imaginarse que los programas generalmente se consideran validados cuando no presentan problemas o fallos en el cumplimiento de sus requisitos.

Si el código de simulación debe entregarse a un cliente, se procede a diseñar un conjunto de pruebas de *aceptación*. La aceptación del código de simulación pasa por el estricto cumplimiento de estas pruebas, que verifican determinados aspectos puntuales y globales de la simulación. En ningún momento se cubre toda la globalidad. Una vez entregado el software, todo error que pudiera aparecer se corrige en la denominada fase de mantenimiento del código de simulación.

Es comúnmente aceptado que códigos grandes (50000–1000000 líneas) bien estructurados y con un diagrama jerárquico bien definido presentan muchos menos problemas fantasma o fallos que una programación plana no jerarquizada. Los códigos bien escritos son más robustos en su funcionamiento, frente a los diferentes casos para los que están diseñados, que los códigos que incluyen programación plana. La depuración de pequeñas secuelas o errores durante su vida útil se hace mucho más fácil en programas bien jerarquizados y estructurados. Como regla general se puede afirmar que la solución de un error detectado en un programa sin jerarquía ni estructura pasa por la reescritura completa de esa parte de código antes que por intentar entender cómo está implementado y modificarlo.

Sin embargo, como hemos dicho con anterioridad, el uso de una estructura jerárquica elaborada debe limitarse a códigos que por su extensión y complejidad lo requieran y, por otra parte, a programadores con un nivel avanzado de programación.

Se hace por tanto necesario utilizar una metodología específica de validación. En el caso concreto de intentar validar códigos desarrollados para la simulación numérica, las siguientes técnicas pueden ser útiles:

- Cambiar el paso temporal o la malla espacial de la integración numérica. Representando los resultados para diferentes pasos temporales de integración o para diferentes mallas, y conociendo la precisión de nuestro esquema numérico, podremos saber si los resultados tienen la precisión numérica de nuestro esquema. Por otra parte, es fácil analizar la convergencia de los resultados numéricos para pasos temporales suficientemente pequeños.
- Comparar los resultados numéricos con una solución analítica conocida.

7.7. Características de un código de simulación

Después de haber analizado ciertos aspectos de un proyecto software podemos enunciar cuales son las características deseables en un código de simulación:

- Portabilidad.

  Generalmente, cuando desarrollamos un código no sabemos en qué arquitectura específica va a correr nuestro programa. Es más, nos gustaría que nuestro programa funcionara en cualquier arquitectura hardware. En la mayoría de los casos, no es necesario ejecutar el código en una máquina ("target") específica, pero aunque así fuera, deberemos escribir el código hasta donde podamos independiente de la máquina, y sólo una pequeña parte, y desacoplada del resto, propia de la arquitectura en cuestión. De esta forma, si no existe una dependencia de la arquitectura específica en nuestro código, éste será automáticamente portado a otra máquina, compilado y ejecutado sin necesidad de modificación alguna. Por otra parte, si existiera una cierta dependencia con la arquitectura, y siempre que ésta haya sido aislada convenientemente, el esfuerzo de reescritura para una nueva arquitectura es mínimo. Hoy en día un desarrollo software es, en general, mucho más costoso que un ordenador. En consecuencia, parece necesario que los códigos sigan funcionando sin problemas durante su vida útil aunque las arquitecturas cambien.

- Robustez.

  Durante la vida útil de un código, se hacen necesarias numerosas modificaciones. Independientemente de la pericia del programador, las modificaciones no deberían manifestar grandes problemas. Sin embargo, hay ciertos

códigos en los que cualquier modificación se puede convertir en un verdadero infierno. Por otra parte, es deseable que un programa no dé errores de ejecución para un rango amplio de los argumentos de entrada para los que está diseñado. Aquellos códigos en los que las modificaciones sean fáciles y cuyo rango de funcionamiento sea amplio los consideraremos como robustos.

- Reutilidad.

  Es fácil entender que diferentes proyectos software pueden tener necesidades similares: inversión de matrices, resolución de sistemas lineales, integradores temporales, etc. Por esta razón, es necesario escribir software lo suficientemente general y parametrizado como para que en posteriores proyectos software pueda ser utilizado sin modificación alguna.

- Claridad.

  Implica que el código sea fácil de entender y de seguir. De esta forma, el código puede ser interpretado por otros programadores y sus errores pueden ser detectados con mayor facilidad. Dicha característica se logra con una escritura tan próxima al lenguaje matemático como sea posible utilizando la misma notación que en el *Manual de Modelado Matemático* del proyecto software (si existe). Además, se deben indentar las distintas estructuras para que puedan ser facilmente identificadas. Otro aspecto importante es comentar las líneas de código en forma precisa y clara sin ser excesiva.

- Eficiencia.

  Un código de simulación *eficiente* es aquel cuyo código en lenguaje máquina correspondiente presenta un *tiempo de ejecución* reducido y consume un espacio de *memoria* pequeño.

  El concepto de eficiencia tiene sentido en grandes códigos de simulación donde dependiendo del problema físico que se trata y de la arquitectura que se dispone, los tiempos de ejecución suelen ser altos, del orden de la decena de horas y las estructuras de datos usadas ocupan gran cantidad de memoria. La tarea de optimización apunta a obtener códigos más eficientes.

- Reducido coste de validación.

  La validación de los resultados es, como ya hemos comentado, una de las fases que más esfuerzo requiere. Es por lo tanto imprescindible que el programa esté implementado con una metodología que permita validar procedimientos de forma independiente.

- Reducido mantenimiento.

  Hoy en día, los códigos de simulación pueden tener una vida útil muy larga. A lo largo de su vida, éstos se suelen modificar para contemplar nuevos casos, nuevos modelos físicos o diferentes algoritmos. Cada vez que exista una modificación, habrá que validar el programa. Por lo tanto, las modificaciones tendrán un reducido coste asociado siempre que el programa permita la validación independiente de procedimientos.

El logro de estas características depende fundamentalmente de la metodología de programación que adoptemos. A continuación se resumen las ventajas e inconvenientes de cada metodología de programación:

- Programación anárquica.

  Esta programación genera como resultado un código con un flujo de datos muy complicado e imposible de entender. Además, presenta todas las desventajas posibles: son difíciles de validar, imposibles en la corrección de errores, tediosos de modificar, y laboriosos de optimizar y mantener.

- Programación estructurada plana.

  El desarrollo de pequeños programas es más rápido que intentar escribir de una forma estructurada encapsulada. Por otra parte, la velocidad del código se ve incrementada con respecto a códigos estructurados encapsulados. El incremento relativo en la velocidad de ejecución es función del tipo de programación y el tipo de compiladores utilizado. Con respecto a la programación anárquica son más fáciles de entender y de seguir.

  Por el contrario, los inconvenientes superan en gran medida a las ventajas. En el momento en que los programas superan las 500 líneas de código la depuración de éstos se hace muy laboriosa y la validación se convierte en un verdadero infierno. Son complicados de optimizar y de mantener y es imposible reutilizar código.

- Programación estructurada encapsulada.

  Respecto de la programación plana, los procedimientos encapsulados presentan la ventaja de que pueden ser reutilizados aunque requiere un análisis previo del flujo de datos asociado. Las desventajas siguen siendo significativas, son difíciles de optimizar y de validar y el mantenimiento sigue siendo costoso.

- Programación multicapa.

  Los inconvenientes que presenta esta metodología de programación es que los códigos son más lentos que los códigos planos y que consumen más memoria. Sin embargo, las ventajas superan a los inconvenientes. Como ventajas podemos destacar que esta metodología da lugar a códigos muy robustos, fácilmente depurables y con un mantenimiento durante su vida útil muy económico. Por otra parte, la estructuración por capas permite el abordaje de un proyecto software mediante un equipo multidisciplinar de personas.

  En la metodología multicapa la implementación es "top-downçomenzando por la aplicación, luego con el grupo físico y por último codificando el grupo matemático. A su vez, en las capas de cada grupo es posible implementar utilizando prototipos. Esto nos permite ir validando las interfaces entre capas en la fase de implementación.

La validación en este tipo de códigos es sumamente ordenada. Se comienza validando el grupo matemático. Cada integrador y algoritmo matemático se valida con soluciones obtenidas de la literatura. Una vez que se tiene la seguridad del correcto funcionamiento del grupo matemático se valida el grupo físico utilizando ya el grupo matemático validado. Finalizadas las pruebas del grupo físico se procede a validar la aplicación. El estricto orden de este procedimiento asegura que hemos probado casi todas las líneas de código y el porcentaje de validación es casi el cien por cien. Con otras metodologías es más difícil asegurar que las pruebas de validación hayan pasado por la mayoría de las líneas de código.

Capítulo 8

Ejemplos de simulación

8.1. Fases de un proyecto de simulación

El objetivo principal de un proyecto de simulación de un sistema físico mediante ecuaciones diferenciales es desarrollar un código específico para la predicción del comportamiento del sistema. En el desarrollo de un proyecto software de simulación se deben abordar las siguientes tareas clasificadas por fase de desarrollo.

En la primera fase (*análisis de necesidades*) se debe definir, con la mayor precisión, el sistema físico que se pretende simular y los objetivos de la simulación.

En la segunda fase (*estudio del problema*) se deben abordar las siguientes tareas:

1. Determinación del modelo matemático. Generalmente, los modelos matemáticos de sistemas físicos constituyen un conjunto de ecuaciones diferenciales ordinarias o en derivadas parciales. En este apartado se describe el problema físico así como las hipótesis necesarias para que el modelo matemático sea válido. Un estudio cualitativo del comportamiento de la solución, así como cuestiones básicas acerca de existencia y unicidad son necesarias antes de abordar numéricamente el problema. El estudio de la regularidad de la solución y de las posibles singularidades nos permitirá elegir la formulación diferencial, integral o variacional del problema que mejor se adapte al tipo de solución.

2. Esquema numérico para la discretización espacial. Dependiendo de la complejidad geométrica del dominio espacial donde esté definida la solución

objeto del estudio, se elige el tipo de mallado: estructurado o no estructurado. La presencia en la solución de ondas de choque o de capas límites nos define la posición de los puntos de colocación en la malla. Si la solución es suave, se utiliza una malla equiespaciada y para soluciones con gradientes fuertes, una malla no equiespaciada o refinada. La elección de diferencias finitas, volúmenes finitos, métodos espectrales o elementos finitos depende principalmente del tipo de las condiciones de contorno y de la formulación elegida en el apartado anterior. Si la formulación es la diferencial, las diferencias finitas son la elección más simple; los métodos espectrales son otra opción. Los esquemas en volúmenes finitos están ligados a la formulación integral y los elementos finitos a la formulación variacional. Por último, la elección del orden de precisión del esquema espacial empleado depende del tipo de soluciones que esperemos. El tipo de mallado así como las variables físicas que definen el problema nos permiten definir las estructuras de datos que se utilizarán en el esquema numérico.

3. Análisis de estabilidad lineal del problema diferencial y del problema semidiscretizado espacialmente. En este apartado analizamos la estabilidad de una solución estacionaria, si existe, linealizando el sistema. También, estudiamos la estabilidad del problema semidiscretizado espacialmente para comparar ambos caracteres de estabilidad. Generalmente, el objetivo es que la semidiscretización espacial preserve el carácter de estabilidad.

4. Esquema numérico para la discretización temporal. Mediante el análisis de estabilidad del apartado anterior podemos elegir el esquema numérico para avanzar en el tiempo. La forma de los autovalores de estabilidad nos permitirá elegir el esquema temporal con una región de estabilidad absoluta que mejor se adapte al problema objeto de la simulación y determinar el paso de integración temporal que preserva el carácter de estabilidad del problema discretizado espacial y temporalmente. Teniendo en cuenta la relación entre el paso de integración temporal y el paso espacial, se elige el orden de precisión del esquema temporal para que el error espacial y temporal tengan el mismo orden de magnitud.

5. Coste computacional: tiempo de ejecución y memoria asignada. Conocido el modelo matemático objeto de estudio y el esquema numérico a utilizar, es fácil estimar el orden de magnitud del número de operaciones necesario por paso de tiempo. Las estructuras de datos definidas al elegir el tipo de mallado nos permiten estimar la cantidad de memoria necesaria para llevar a cabo la simulación.

En la tercera fase (*implementación*) se puede desarrollar un código de simulación empleando estructura multicapa. En esta fase se escribe el código fuente en el que se implementa el esquema numérico anteriormente elegido y se realizan las siguientes tareas:

1. Estructura funcional con jerarquía. El primer paso es diseñar una estructura de bloques con una jerarquía funcional en la que se designan las principales funciones: integrador temporal, condiciones de contorno, ecuaciones, discretización espacial, condiciones iniciales.

2. Especificación de cada unidad funcional. Una vez definido el diagrama de bloques anterior, es importante especificar la funcionalidad de cada bloque y los argumentos de entrada y argumentos de salida.

3. Implementación. Estamos preparados para escribir el código fuente que responde a las especificaciones anteriormente descritas.

En la cuarta fase (*compilación*) se compila el código fuente hasta dejar éste libre errores.

Por último, la quinta fase (*validación*) es una de la tareas más laboriosas en el desarrollo de un proyecto software. El éxito y la celeridad en la validación de un código dependen principalmente, de que se hayan seguido con una cierta metodología las tareas anteriores. Se realizan las siguientes tareas:

1. Representación gráfica de la solución. El primer paso para validar la solución es representar gráficamente los resultados. La presencia en los resultados de rizados, gradientes fuertes que no deberían aparecer, o comportamientos anómalos en el contorno nos indican de forma rápida y certera los errores en la implementación del código. Cuando revisado el código no se encuentra la fuente de error, los comportamientos anómalos pueden ser debidos a una defectuosa elección del esquema o a un mal tratamiento de las condiciones de contorno. En este caso, es necesario volver a revisar todos y cada uno de los apartados anteriores.

2. Estimación del error. Cuando los resultados gráficos tienen visos de verosimilitud, procedemos a realizar un estudio de convergencia con el paso de la malla y con el paso temporal. De esta forma, calculamos numéricamente el orden de precisión del esquema numérico utilizado.

3. Discusión de los resultados. La discusión de los resultados desde el punto de vista físico suele ser un herramienta útil para validar el código numérico.

Uno de los objetivos del problema es la reducción del tiempo de ejecución de la simulación y del consumo de recursos, es decir, la optimización. Pensemos que el código objeto de estudio se puede utilizar para simular en tiempo real el flujo alrededor de un perfil y que el tiempo de proceso puede ser crítico. Si después de la validación alguno de estos dos objetivos no se ha cumplido, es necesario optimizar el código de forma que se minimicen los tiempos y los recursos. Las tareas que se deben realizar dentro de la optimización son las siguientes:

1. Análisis cuantitativo del coste computacional de las diferentes partes del código. Se analiza el tiempo de ejecución de cada componente funcional. En aquellas componentes en donde resida el grueso del tiempo de ejecución se pone especial énfasis en la optimización.

2. Esfuerzo computacional sin optimizar el código.

3. Esfuerzo computacional con optimización. El tiempo de ejecución de un código sin optimizar puede suponer diez veces el de un código optimizado. Las necesidades de memoria de un código sin optimizar pueden llegar a ser el doble o el triple que las del mismo optimizado. Esto hace pensar en la importancia de la optimización del producto final. Dependiendo de la metodología de programación, del esquema numérico utilizado y del tipo de código, las opciones de compilación que minimizan los tiempos de ejecución pueden variar de unos códigos a otros. Es por tanto necesario probar sistemáticamente las opciones de optimización del compilador para dar con la combinación óptima que minimiza los tiempos de ejecución.

8.2. Flujo de calor con convección

El flujo de calor en régimen estacionario $\dot{Q}$ entre un fluido incompresible a temperatura T_∞ que se sopla sobre un cuerpo sólido a temperatura T_c se puede expresar como,

$$\dot{Q} = Sh_c(T_\infty - T_c), \tag{8.1}$$

donde h_c es la constante de convección y S es la superficie del cuerpo. El objetivo de este problema consiste en obtener mediante simulación numérica la constante h_c para una geometría dada.

El problema de la transmisión de calor en el fluido se puede modelar mediante la siguiente ecuación convectivo–difusiva,

$$\rho c \left(\frac{\partial T}{\partial t} + \mathbf{v} \cdot \nabla T \right) = k \nabla^2 T, \tag{8.2}$$

donde T es la temperatura del fluido, $\mathbf{v}$ es la velocidad del fluido que se considera conocida, ρ es su densidad, c es su calor específico y k es su conductividad. Esta ecuación completada con las condiciones iniciales y de contorno permite obtener la transmisión de energía térmica entre el fluido y el sólido.

Consideramos el problema en un dominio infinito bidimensional (x, y) con una condición inicial $T(x, y, 0)$ constante para todo el campo e imponemos las condiciones de contorno en el infinito y en el contorno sólido.

Las condiciones de contorno en el infinito dependen de que el flujo sea entrante o saliente. Si por una parte del contorno del infinito el flujo entra ($\mathbf{v} \cdot \mathbf{n} < 0$), podemos suponer que la temperatura del fluido está sin perturbar y vale la del infinito T_∞. Sin embargo, si el flujo sale por alguna parte del contorno del infinito ($\mathbf{v} \cdot \mathbf{n} > 0$), no se puede imponer la condición de contorno de campo de temperaturas del infinito. En este caso el campo de temperaturas está modificado por la presencia del contorno sólido aguas arriba de ese punto. Puesto que no se puede imponer la condición de contorno, la temperatura en esas partes del contorno deberá ser extrapolada.

Las condiciones de contorno en la superficie sólida dependen del propio sólido en cuestión. En concreto, si el sólido tiene una conductividad muy alta y es capaz de evacuar rápidamente el flujo de calor comunicado por el fluido manteniendo su temperatura constante, una buena hipótesis es considerar que la temperatura en el contorno sólido es la misma para todo el contorno y constante en el tiempo. Si el sólido no es capaz de evacuar el calor comunicado por el fluido, aún teniendo una conductividad elevada, será necesario integrar una ecuación diferencial adicional para conocer la evolución temporal de la temperatura del sólido y, en consecuencia, de la temperatura del contorno. En otras ocasiones, se conoce el flujo de calor en la superficie del sólido y la condición de contorno se introduce en las derivadas del campo de temperaturas (condición de contorno tipo Neumann) y no en la temperatura en sí (condición de contorno tipo Dirichlet). En general, si no conocemos la distribución de temperaturas del sólido, será necesario integrar de forma acoplada con el fluido la ecuación de la transmisión de calor por conducción en el sólido. La condición de contorno en este caso desaparece y se convierte en un punto interior más del dominio de integración.

La discretización espacial del problema de la transmisión de calor se puede abordar mediante dos esquemas diferentes: diferencias finitas asociadas a la formulación diferencial y volúmenes finitos asociados a la formulación integral del problema. Generalmente, cuando la condición de contorno es la temperatura, los esquemas en diferencias finitas son mucho más fáciles de implementar que los esquemas en volúmenes finitos. Cuando se ataca directamente la formulación diferencial (8.2) es necesario obtener esquemas que aproximen las derivadas primeras y segundas que aparecen en (8.2) en los puntos nodales o de colocación de una malla. Si la malla es estructurada, estos esquemas se pueden obtener mediante diferencias finitas de los valores nodales. En el caso más sencillo podemos considerar una malla estructurada cartesiana para un dominio cuadrado $[0,1] \times [0,1]$ en donde su nodos están dados por las siguientes expresiones:

$$x_{ij} = \frac{i}{N}, \qquad y_{ij} = \frac{j}{N}, \qquad i = 0 \dots N, \quad j = 0 \dots N,$$

donde $N+1$ es el número de puntos con que se discretiza en cada dirección. Tanto las derivadas primeras como las derivadas segundas las podemos calcular mediante un esquema centrado de tres puntos con orden dos de aproximación.

Las derivadas en la dirección x o en la dirección y se expresan de forma similar,

$$\left(\frac{\partial T}{\partial x}\right)_{ij} = \frac{1}{2\Delta x}\left(T_{i+1,j} - T_{i-1,j}\right), \qquad \left(\frac{\partial^2 T}{\partial x^2}\right)_{ij} = \frac{1}{\Delta x^2}\left(T_{i+1,j} - 2T_{ij} + T_{i-1,j}\right).$$

Las expresiones anteriores son válidas para todos los puntos interiores

$$\{(x_{ij}, y_{ij}), j = 1 \ldots N-1, i = 1 \ldots N-1\}.$$

Si las condiciones de contorno se dan en los valores de las temperaturas del contorno, al plantear las derivadas discretizadas en los puntos próximos al contorno las condiciones de contorno se introducen de forma automática. Para los puntos del contorno en donde la temperatura se deba extrapolar, la temperatura de los puntos del contorno se puede poner en función de la temperatura de los puntos interiores,

$$T_0 = \frac{2T_1}{3} + \frac{T_2}{3}. \tag{8.3}$$

En la expresión anterior, T_0 es la temperatura en el contorno y T_1, T_2 son las temperaturas interiores en la línea perpendicular al contorno. Es importante hacer notar que los valores de las magnitudes nodales son funciones del tiempo, y que las expresiones de los valores del contorno en función de los puntos interiores son válidas para todo instante. De esta forma, siempre podemos plantear un conjunto de $(N-1) \times (N-1)$ ecuaciones diferenciales ordinarias para la evolución de las magnitudes en los puntos interiores. Los valores de las magnitudes en el contorno, que han sido previamente despejadas en función de los puntos interiores, aparecen en las ecuaciones de puntos interiores próximos al contorno.

Si el problema de transmisión de calor está compuesto por diferentes medios con diferentes conductividades, es de esperar la presencia de discontinuidades en las derivadas del campo de temperaturas. En este caso, la formulación integral permite de forma automática la conservación de los flujos de calor y permite obtener resultados fiables. La formulación integral de la transmisión de calor se puede expresar de la siguiente forma:

$$\frac{d}{dt}\int_{\Omega} \rho c T d\Omega + \int_{\partial\Omega} (\rho c T \mathbf{v} - k\nabla T) \cdot \mathbf{n} ds = 0, \tag{8.4}$$

donde Ω es un volumen de control genérico.

Cuando se ataca la formulación integral (8.4) se debe definir un conjunto de volúmenes de control. Para el caso sencillo de una malla cartesiana estructurada, los volúmenes de control se pueden elegir como el espacio comprendido entre cuatro nodos de la malla. En este caso, mediante el teorema del valor medio podemos realizar las integrales extendidas a los volúmenes de control Ω_{ij} de la siguiente manera:

$$\frac{d}{dt}\int_{\Omega_{ij}} \rho c T d\Omega = \rho c \frac{d\bar{T}_{ij}}{dt}\Omega_{ij},$$

donde $\bar{T}_{ij}$ es el valor medio en el dominio Ω_{ij}. Además, para conocer la variación de la temperatura media del volumen Ω_{ij} necesitamos calcular los flujos de $\mathbf{F} = \rho c T \mathbf{v} - k\nabla T$ a través de las caras del volumen Ω_{ij}. Si suponemos que los flujos a lo largo de los lados tienen una variación lineal (esquema de segundo orden), el flujo Φ_l a lo largo de cada lado l se puede aproximar mediante,

$$\Phi_l = \frac{1}{2}\left(\mathbf{F}_1 + \mathbf{F}_2\right) \cdot \mathbf{n},$$

donde $\mathbf{F}_1$ y $\mathbf{F}_2$ son los valores del flujo en los extremos de los lados o vértices de la malla. Estos flujos dependen tanto de los valores de las variables como de sus derivadas en los vértices. Para calcular las derivadas podemos utilizar un esquema clásico de diferencias finitas. Puesto que el sistema de ecuaciones diferenciales lo vamos a plantear para la evolución de los valores medios, necesitamos unas relaciones algebraicas que nos relacionen los valores medios con los valores nodales. Para un esquema de segundo orden, estas relaciones se obtienen integrando para un conjunto consecutivo de cuatro volúmenes de control,

$$\Omega_t = \Omega_{ij} + \Omega_{i+1j} + \Omega_{ij+1} + \Omega_{i+1j+1},$$

obteniéndose,

$$\Omega_t T_{ij} = \Omega_{ij}\bar{T}_{ij} + \Omega_{i+1j}\bar{T}_{i+1j} + \Omega_{ij+1}\bar{T}_{ij+1} + \Omega_{i+1j+1}\bar{T}_{i+1j+1}.$$

Las relaciones anteriores son válidas para todos los vértices interiores. Los valores de las magnitudes en los vértices del contorno los podemos extrapolar mediante los vértices vecinos.

Los flujos extendidos a las caras los podemos clasificar en flujos en lados interiores y flujos en lados del contorno. Para los lados del contorno sólido no permeable $\mathbf{v} \cdot \mathbf{n}$ es nulo, y la única componente de flujo que queda es la del calor por conducción. Este flujo puede ser conocido directamente a partir de la condición de contorno. Si no fuera ésta la situación, el cálculo de la derivada normal se podría llevar a cabo mediante un esquema clásico descentrado en diferencias finitas que involucrara nodos próximos al contorno.

8.3. Transmisión de ondas de densidad

El objetivo de este problema consiste en determinar la eficiencia de una pantalla acústica creada para protegernos del ruido de una carretera. Se considera que el aire no es absorbente y que no existe ni viento ni gradientes de temperatura. Con estas hipótesis el sistema no disipativo de ecuaciones que gobierna las ondas de presión sonora son:

$$\frac{\partial P}{\partial t} + \rho_0 c^2 \nabla \cdot \mathbf{v} = 0, \tag{8.5}$$

$$\rho_0 \frac{\partial \mathbf{v}}{\partial t} = -\nabla P, \tag{8.6}$$

donde ρ_0 es la densidad del aire, c es la velocidad del sonido en el aire, P es la perturbación de presión y $\mathbf{v}$ la correspondiente perturbación de velocidad de la onda de presión. Para eliminar la velocidad de estas ecuaciones, derivamos la primera con respecto al tiempo y tomamos la divergencia de la segunda,

$$\frac{\partial^2 P}{\partial t^2} = \nabla^2 P. \tag{8.7}$$

Para transformar este problema en un problema de evolución de primer orden, definimos $\Phi = \partial P/\partial t$ y el sistema queda:

$$\frac{\partial \Phi}{\partial t} = c^2 \nabla^2 P, \tag{8.8}$$

$$\frac{\partial P}{\partial t} = \Phi. \tag{8.9}$$

Como condiciones iniciales consideraremos que no existe perturbación inicial, es decir:

$$P(x, y, 0) = 0, \qquad \Phi(x, y, 0) = 0. \tag{8.10}$$

La condición de contorno depende del tipo de contorno tratado. Consideramos que el suelo o cualquier superficie existente es reflexiva pura o, lo que es lo mismo, que la perturbación de velocidad $\mathbf{v}$ es nula. En las superficies del infinito las ondas salen sin ningún efecto de reflexión y, en consecuencia, no podemos imponer condiciones de contorno. Por último, tendremos superficies que se encuentren vibrando o que sean precisamente los focos de ruido. Para estas superficies, la componente normal de la velocidad a la superficie está dada por: $\mathbf{v} \cdot \mathbf{n} = f(t)$. Imponer estas condiciones de contorno mediante la formulación diferencial es complicado. Por esta razón, planteamos la formulación integral de (8.8)–(8.9) extendida a un rectángulo de la malla cartesiana:

$$\frac{d\bar{\Phi}_{ij}}{dt} = \frac{c^2}{A_{ij}} \sum_{l=1}^{4} F_l, \tag{8.11}$$

$$\frac{\bar{P}_{ij}}{dt} = \bar{\Phi}_{ij}, \tag{8.12}$$

donde A_{ij} representa el área del volumen de control, $\bar{P}_{ij}$ el valor medio de su presión y F_l representa el siguiente flujo a través del lado l:

$$F_l = \int_l \nabla P \cdot \mathbf{n} dl. \tag{8.13}$$

Para evaluar numéricamente esta integral, podemos hacer una integral con n puntos gausianos. En concreto, para un esquema de segundo orden nos basta con el valor del integrando en el punto medio del lado,

$$F_l = \Delta L \left(\nabla P \cdot \mathbf{n}\right)_{pm}, \tag{8.14}$$

donde ΔL es la longitud del lado, $\mathbf{n}$ su normal y pm representa su punto medio. Cuando el lado en cuestión no sea un lado del contorno, la derivada en la dirección de la normal que aparece en la ecuación anterior se calcular mediante la información de los valores medios de los volúmenes de control adyacentes a ese lado. Si ese lado pertenece al contorno, entonces tendremos que especificar el tipo de contorno:

1. Lado del infinito.
2. Lado reflexivo.
3. Lado vibrante o foco emisor.

En el caso de que se trate de un lado del infinito no se puede imponer condición de contorno y (8.14) deberá ser calculado descentrando la derivada direccional o tomando información de volúmenes existentes en el dominio de integración. En el caso de que se trate de un lado reflexivo, la velocidad normal a ese lado es nula o, lo que es lo mismo, $F_l = 0$. Por último, en el caso de que el lado pertenezca a un foco vibrante, $\mathbf{v}\cdot\mathbf{n} = f(t)$ es conocido. Para transformar esa condición de contorno en velocidad a una condición de contorno en flujo, multiplicamos la ecuación (8.6) escalarmente por la normal y derivamos con respecto al tiempo la condición de contorno. De esta forma, la condición de contorno impuesta en flujos queda:

$$F_l = -\rho_0 \frac{df}{dt}. \tag{8.15}$$

8.4. Transmisión de ondas de flexión

La transmisión de las ondas en los sólidos elásticos es mucho más complicada que en los fluidos. Mientras que en los fluidos las únicas ondas que existen son las de densidad, en los sólidos elásticos existen cinco tipos de ondas que transmiten información: ondas de flexión, ondas de densidad, ondas transversales, ondas de torsión, ondas de dilatación y ondas de Rayleigh.

En este apartado vamos a considerar exclusivamente las ondas lineales de flexión que aparecen en las vigas. Cuando una viga se ve sometida a cargas constantes o no constantes perpendiculares al eje de la viga y contenidas en el plano de simetría de la sección, si existe, la viga flexa en el plano de la carga y las únicas deformaciones importantes son las que se producen en la dirección de la carga. Las tensiones o esfuerzos que aparecen en el interior de la viga se pueden expresar en función de la flecha vertical de las diferentes secciones. De esta forma, la transmisión de ondas cuando una viga vibra sometida a cargas gravitatorias o

cualquier otro tipo de cargas, se puede modelar a través de la siguiente ecuación diferencial:

$$\rho A \frac{\partial^2 u}{\partial t^2} = -\frac{\partial^2}{\partial x^2}\left(EI\frac{\partial^2 u}{\partial x^2}\right) + g\,(x,t)\,, \tag{8.16}$$

donde ρ representa la densidad de la viga, $A(x)$ el área de su sección, $u(x,t)$ el desplazamiento vertical de las diferentes secciones definidas por su coordenada x, $g(x,t)$ la función de cargas y $EI(x)$ la rigidez a flexión de la viga.

Para transformar este problema en un problema de evolución de primer orden, definimos la variable $w(x,t)$ mediante,

$$\frac{\partial w}{\partial t} = -EI(x)\frac{\partial^2 u}{\partial x^2}, \tag{8.17}$$

que junto con la ecuación para la flecha $u(x,t)$,

$$\rho A(x)\frac{\partial u}{\partial t} = \frac{\partial^2 w}{\partial x^2} + \int_0^t g(x,\tau)d\tau, \tag{8.18}$$

constituyen un sistema de dos ecuaciones de primer orden para determinar las vibraciones de la viga.

Las ecuaciones (8.17) y (8.18) se deben completar con condiciones iniciales para $u(x,t)$ y para $w(x,t)$ y condiciones de contorno. En el caso de que la viga parta de una condición de reposo sin carga, la flecha es nula $u(x,0) = 0$ y su velocidad también es nula$\dot{u}(x,0) = 0$. Mediante la ecuación (8.18), esta última condición es equivalente a:

$$\frac{\partial^2 w}{\partial x^2}(x,0) = 0, \tag{8.19}$$

que obliga a que la condición inicial para w sea de la forma $w(x,0) = A + Bx$ con A y B constantes. Sin pérdida de generalidad, elegimos $A = 0$ y $B = 0$ y la condición inicial para w queda: $w(x,0) = 0$.

Con respecto a las condiciones de contorno, tenemos que imponer el tipo de vinculación de los extremos de la viga que pueden ser:

1. Empotramiento. Si un extremo $x = L$ de la viga está empotrado, entonces la flecha y el giro del extremo son cero, es decir:

$$u(L,t) = 0, \qquad \frac{\partial u}{\partial x}(L,0) = 0.$$

2. Articulación. Si un extremo $x = L$ de la viga está articulado, entonces la flecha y el momento flector del extremo son cero

$$u(L,t) = 0, \qquad \frac{\partial^2 u}{\partial x^2}(L,0) = 0.$$

3. Extremo libre. Cuando el extremo de la viga está libre, entonces el momento flector y el esfuerzo cortante son cero, es decir:

$$\frac{\partial^2 u}{\partial x^2}(L,0) = 0, \qquad \frac{\partial^3 u}{\partial x^3}(L,0) = 0.$$

Independientemente del tipo de vinculación con el terreno, podemos imponer dos condiciones de contorno para $u(x,t)$ en un extremo y otras dos en el otro.

Si abordamos el problema (8.17)–(8.18) mediante diferencias finitas, podemos discretizar la viga mediante una serie de nodos equiespaciados $\{x_j, j = 0, \dots N\}$ e imponer las ecuaciones (8.17) y (8.18) en los nodos interiores $j = 2, \dots N-2$ para determinar la evolución de la flecha de estas secciones. La flecha de las secciones $j = 0, 1, N-1, N$ es función de la flecha de los nodos interiores para que se satisfagan las condiciones de contorno. Como las condiciones de contorno son derivadas de la flecha $u(x,t)$, la discretización descentrada en $x = 0$ de sus dos condiciones de contorno constituye un sistema de dos ecuaciones en diferencias que involucran la flecha de los nodos $j = 0, 1, 2, \dots$. Siempre podemos despejar de este sistema u_0 y u_1 y ponerlo en función de los valores $u_2, u_3, \dots$. De igual forma, al imponer la condición de contorno en $x = L$, la flecha de los nodos $j = N-1$ y $j = N$ está determinada en función de las flechas de los nodos interiores. De esta forma, el sistema discretizado (8.6)–(8.18) queda:

$$\frac{dw_j}{dt} = F(u_0, u_1, \dots u_j, \dots u_N), \qquad j = 0, \dots N \tag{8.20}$$

$$\rho A(x_j)\frac{du_j}{dt} = G(w_1, \dots w_{N-1}) + \int_0^t g(x_j, \tau)d\tau, \qquad j = 2, \dots N-2 \tag{8.21}$$

con las condiciones de contorno en $x = 0$ siguientes:

$$u_0 = f_1(u_2, \dots u_{N-2}), \qquad u_1 = f_2(u_2, \dots u_{N-2}) \tag{8.22}$$

y las condiciones en $x = L$

$$u_N = f_3(u_2, \dots u_{N-2}), \qquad u_{N-1} = f_4(u_2, \dots u_{N-2}) \tag{8.23}$$

Para realizar la evolución temporal, partimos de la condición inicial

$$w_j = 0, \qquad u_j = 0, \qquad j = 0, \dots N$$

y mediante un esquema temporal integramos (8.20)–(8.21). Conocida la solución en un instante t_n se determinan los valores de la flecha u_0, u_1, u_{N-1} y u_N mediante (8.22)–(8.23). A continuación, mediante (8.20) y (8.21) se calcula la solución en el instante t_{n+1} y así sucesivamente.

Bibliografía

[1] Bishop, P. (1989) Conceptos de Informática. Anaya Multimedia.

[2] Borse, G. J. (1989) Programación en FORTRAN 77 con Aplicaciones de Cálculo Numérico en Ciencias e Ingeniería. Anaya Multimedia.

[3] Cuevas, G. (1991) Ingeniería del software: práctica de la programación. RA–MA.

[4] Dowd, K. (1993) High performance computing. O'Reilly & Associates, Inc.

[5] ESA.(1991) ESA Software Engineering Standards. PSS-05-0. Issue 2.

[6] Etter, Delores M. (1997) Structured FORTRAN 77 for engineers and scientists. Addison–Wesley.

[7] García Merayo, F.. (1990) Programación en FORTRAN 77. Paraninfo.

[8] Golden, James T. (1976) FORTRAN IV. Programación y cálculo. Urmo, S.A. de ediciones.

[9] Joyanes, L. (1987) Metodología de la programación; diagramas de flujo, algoritmos y programación estructurada. Mc Graw–Hill.

[10] Kernighan, B. W. y Ritchie, D. M. (1988) The C programming language. Prentice–Hall.

[11] Knuth, Donald E. (1997) The art of computing programming. Volume 1: Fundamental algorithms. Volume 2: Seminumerical algorithms. Addison–Wesley.

[12] Metcalf, M. y Reid, J. (1999) FORTRAN 90/95 explained. OXFORD. University Press.

[13] NASA. (1992) Recommended Approach to Software Development. Revision 3. Software Engineering Laboratory Series. SEL–81–305.

[14] Piattini, Mario G. y Daryanani, Sunil N. (1995) Elementos y herramientas en el desarrollo de sistemas de información. Una visión actual de la tecnología CASE. RA–MA.

[15] Pressman, Roger S. (1997) Ingeniería del software. Un enfoque práctico. Mc. Graw–Hill.

[16] Schofield, C. F. (1989) Optimising fortran programs. John Wiley & Sons.

[17] Wirth, N. (1980) Algoritmos + Estructuras de Datos = Programas. Ediciones del Castillo.

[18] Yourdon, E. (1989) Managing the structrured techniques. Prentice–Hall.

Índice alfabético