Tema 2

Datos y variables

Versión: 16 de febrero de 2009

2.1 Tipos de datos

Se denomina **dato** a cualquier objeto manipulable por el ordenador. Un dato puede ser un carácter leído de un teclado, información almacenada en un disco, un número que se encuentra en la memoria central, etc.

Los distintos tipos de datos se representan en diferentes formas en el ordenador: por ejemplo, no se almacena internamente de la misma manera un número entero que un carácter. Aunque los lenguajes de alto nivel permiten en alguna medida ignorar la representación interna de los datos, es preciso conocer algunos conceptos mínimos.

A nivel de máquina todos los datos se representan utilizando una secuencia **finita** de bits. De este hecho ya se deduce que **no todos los datos son representables en un ordenador**. La definición de un **tipo de dato** incluye la definición del conjunto de valores permitidos y las operaciones que se pueden llevar a cabo sobre estos valores.

Cuando se utiliza un dato en un programa es preciso que esté determinado su tipo para que el traductor sepa cómo debe tratarlo y almacenarlo. Dependiendo del lenguaje puede o no ser preciso declarar expresamente en el programa el tipo de cada dato. No todos los tipos de datos existen en todos los lenguajes de programación. Hay lenguajes más ricos que otros en este sentido. Los tipos de datos básicos más usuales son:

- **Enteros:** números pertenecientes a un subconjunto finito de los números enteros.

```
EJEMPLO:
     5      22    -1
```

- **Reales:** números pertenecientes a un subconjunto finito de los números reales (constan de una parte entera y una parte decimal).

```
EJEMPLO:
     0.09     -31.423     3.0
```

- **Lógicos:** los dos valores lógicos, VERDADERO (true) o FALSO (false).
- **Caracteres:** un conjunto finito de caracteres reconocidos por un ordenador.

```
EJEMPLO:
      alfabéticos:  A  B  C  ...  Z  a  b  c  ...  z
      numéricos:    1  2  3  ...  9  0
      especiales:   +    -    *    /    ^    .    ;    <    >    $
```

2.2 Estructuras de datos

Los tipos de datos vistos en la sección anterior se suelen denominar **elementales**. Una **estructura de datos** o **tipo de datos estructurado** es un tipo de dato construido a partir de otros tipos de datos. Como ejemplo se tienen los siguientes:

- **Complejos:** son datos formados por un par de datos reales y sirven para tratar números complejos.

```
EJEMPLO:
     2+3i    -3+i (i es la unidad imaginaria)
```

- **Cadenas de caracteres:** (tambien llamadas **string**) son una sucesión de caracteres delimitados por una comilla (apóstrofo) o dobles comillas, según el tipo de lenguajes de programación.

```
EJEMPLO:
     'Esto es una cadena de caracteres'    'string'    '123abc'
```

- **Matrices:** son conjuntos de datos numéricos organizados para formar una matriz o un vector.

```
EJEMPLO: vector fila de dimensión 4
     [1, 0, 3, 4]
```

2.3 Operaciones aritméticas

Las operaciones aritméticas habituales se representan normalmente mediante los símbolos siguientes:

Descripción	Símbolo
Exponenciación	^
Suma	+
Resta	−
Multiplicación	*
División	/

Tabla 2.1: Operadores aritméticos elementales

2.3.1 Reglas de prioridad

Las operaciones aritméticas NO se efectúan siempre en el orden en que están escritas. El orden viene determinado por las reglas siguientes:

1. Exponenciaciones.
2. Multiplicaciones y divisiones.
3. Sumas y restas.
4. Dentro de cada grupo, de izquierda a derecha.

   Para cambiar este orden se usan los paréntesis.

5. Si hay paréntesis, su contenido se calcula antes que el resto.
6. Si hay paréntesis anidados, se efectúan primero los más internos.

EJEMPLO:

$2 + 3 * 4 = 2 + \mathbf{12} = 14$

$(2 + 3) * 4 = \mathbf{5} * 4 = 20$

$1/3 * 2 = \mathbf{0.3333}\ldots * 2 = 0.6666$

$1/(3 * 2) = 1/\mathbf{6} = 0.166666\ldots$

$2 + 3\text{^}4/2 = 2 + \mathbf{81}/2 = 2 + \mathbf{40.5} = 42.5$

$4\text{^}3\text{^}2 = (4\text{^}3)\text{^}2 = \mathbf{64}\text{^}2 = 4096$

2.4 Operaciones de comparación

Imprescindibles para verificar **condiciones** son las **expresiones lógicas**, es decir, expresiones cuya evaluación produce un **valor lógico**. Las más simples son aquéllas en las que se comparan dos datos. Los **operadores de comparación** actúan entre dos datos, que tienen que ser del mismo tipo, y producen un resultado lógico: **true** o **false** (en MATLAB se equiparan con **1** y **0** respectivamente).

Los operadores de comparación se representan de distintas formas según el lenguaje. Los que se muestran en la tabla siguiente son los de MATLAB:

Descripción	Símbolo
Igual a	==
No igual a	~=
Menor que	<
Mayor que	>
Menor o igual que	<=
Mayor o igual que	>=

Tabla 2.2: Operadores de comparación

```
EJEMPLO:
      3<6        true
      0>=1       false
      'A'=='B'   false
```

2.5 Operadores lógicos

Son los que actúan entre dos operandos de tipo lógico. Permiten construir expresiones que representen condiciones más complicadas, como que se verifiquen varias condiciones a la vez, que se verifique una entre varias condiciones, etc.

La representación de los operadores lógicos varía bastante de un lenguaje a otro. Es estas notas se representarán como en MATLAB:

Descripción	Símbolo
Negación	~
Conjunción	&
Disyunción	|

Tabla 2.3: Operadores lógicos

El primero de ellos, el operador de negación lógica ~, es un operador unario, es decir, actúa sobre un solo operando lógico, dando como resultado el opuesto de su valor, como muestra la siguiente tabla:

A	~ A
true	false
false	true

Tabla 2.4: Resultados del operador ~

```
EJEMPLO:
     (6>10) false       ~ (6>10) true
```

Los otros dos operadores lógicos actúan siempre entre dos operandos. Los resultados de su aplicación se muestran en la tabla siguiente:

A	B	A & B	A | B
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Tabla 2.5: Resultados de los operadores & y |.

```
EJEMPLO:
     (1<5)&(5<10)  true
     (0<5)|(0>5)  false
```

En una expresión pueden aparecer varios operadores lógicos. En ese caso, el orden en que se evalúan, es el siguiente:

1. la negación lógica $\sim$
2. la conjunción y disyunción **&** y |
3. En el caso de igual precedencia, se evalúan de izquierda a derecha.

```
EJEMPLO:
     ~(5>0)&(5>4)  false
```

2.6 Funciones intrínsecas

La mayoría de los lenguajes de programación de alto nivel dispone de una serie de funciones pre-programadas para evaluar las funciones elementales que se usan en Matemáticas (coseno, seno, arco-tangente, logaritmo, ...) y otras utilitarias. Se les conoce como **funciones intrínsecas**. La utilización de estas funciones :

```
nombre_de_la_función(argumento/s)
```

donde `argumento/s` pueden ser: un número o expresión, variable o expresión de variables (ver la Sección 2.8).

```
EJEMPLO:
     sqrt(2)  % para calcular √2
     log(cos(x/2))   % para calcular ln(cos(x/2))
```

2.7 Orden general de evaluación de expresiones

En una expresión general pueden aparecer operadores de tipo aritmético, de comparación y lógicos, así como funciones. El orden de evaluación es el que sigue:

- Si en una expresión hay paréntesis, lo primero que se evalúa es su contenido. Si hay paréntesis anidados, se comienza por los más internos. Si hay varios grupos de paréntesis disjuntos, se comienza por el que esté más a la izquierda.
- En una expresión sin paréntesis de agrupamiento, el orden de evaluación es el siguiente:
  1. Las funciones. Si el argumento de la función es una expresión, se le aplican estas reglas. Si hay varias funciones, se comienza por la de la izquierda.
  2. Los operadores aritméticos, en el orden ya indicado.
  3. Los operadores de comparación.
  4. Los operadores lógicos, en el orden antes mencionado.

2.8 Variables

Una **variable** es un nombre simbólico que identifica una parte de la memoria en la que se pueden guardar números u otro tipo de datos. Es un "sitio" en la memoria del ordenador para "guardar" datos. El contenido de una variable se puede recuperar y modificar cuantas veces se quiera durante la ejecución de un programa (o a lo largo de una sesión de trabajo de MATLAB).

Una variable, en general, se identifica por su **nombre** y su **tipo**. El nombre debe estar formado por letras y números y comenzar por una letra, aunque normalmente también se admite el uso de ciertos caracteres especiales. Una variable está asociada a un tipo de datos, el cual determina la cantidad de bytes que usa la variable (ver Sección 2.9).

En la mayoría de los lenguajes de programación (por ejemplo FORTRAN o C) es necesario especificar el tipo de dato que va a contener una variable antes de usarla, declarándolo con las ordenes específicas.

En el lenguaje de programación de MATLAB las variables no necesitan ningún tipo de declaración y pueden almacenar sucesivamente distintos tipos de datos: enteros, reales, escalares, matriciales, caracteres, etc. Se crean, simplemente, **asignándoles un valor**.

2.8.1 Instrucción de asignación

Las **instrucciones de asignación** sirven para almacenar un valor en una variable. La sintaxis más habitual de una operación de asignación es:

```
VARIABLE  =  EXPRESIÓN
```

que debe ser interpretada como: evaluar (si es preciso) el resultado de la `EXPRESIÓN` y almacenarlo en la dirección de memoria correspondiente a `VARIABLE`.

El signo = significa "ALMACENAR EN".

La acción de almacenar un valor en una variable hace que se pierda el valor que, eventualmente, tuviera dicha variable previamente.

```
EJEMPLO:
      a=2                  % guardar en la variable a el valor 2
      b=-4                 % guardar en la variable b el valor -4
      raiz=sqrt(2*b+8*a)   % guardar en la variable raiz el valor √8
      a=a+1                % sumar 1 al contenido de a (guardar 3 en a)
```

2.9 Representación de números en el ordenador

El **bit** es la unidad mínima de información empleada en informática o en cualquier dispositivo digital. Un bit es un (diminuto) dispositivo electrónico capaz de tener dos estados: "apagado", que se asimila al cero y "encendido", que se asimila al uno.

Así, con un bit, sólo pueden representarse dos valores. Para representar más cantidad de valores es necesario usar más bits. Si se usan 2 bits se pueden representar $4 = 2^2$ valores distintos: 00, 01, 10, 11. Si se usan 4 bits se pueden representar $16 = 2^4$ valores distintos: 0000, 0001, 0010, 0011, ...etc. En general, con n bits se pueden representar 2^n valores distintos.

Los bits se suelen agrupar en grupos de 8, formando un **byte**. Para almacenar datos, a su vez, se suelen considerar agrupaciones de 4 u 8 bytes (32 ó 64 bits), a las que se llama **palabra**. Estas

agrupaciones determinan el conjunto de valores que se pueden almacenar en ellas: en 4 bytes (32 bits) se pueden almacenar 2^{32}=4.294$_1$967.296 valores distintos, mientras que en 8 bytes (64 bits) se pueden almacenar $2^{64} > 18 \times 10^{18}$ valores distintos.

Como consecuencia de lo anterior es claro que el tamaño de la palabra determina el cardinal (número de elementos) del conjunto de datos de un determinado tipo. Por ejemplo, no se podrán almacenar en el ordenador más de 2^{32} números enteros distintos: **el conjunto de números con los que se puede trabajar en un ordenador es finito**. A los números (enteros o reales) que sí se pueden representar en el ordenador se les suele llamar **números-máquina**.

Debido a que cada bit sólo dispone de dos estados posibles, los ordenadores utilizan para representar datos numéricos el **sistema binario** de numeración, en el que sólo hay dos **dígitos**: **0** y **1**. Esto significa, básicamente, que los números se representan en el ordenador en su expresión **en base 2**. La forma concreta de codificarlos depende del tipo de dato: entero, real,...

2.9.1 Representación de números enteros

Para un número entero N se tiene:

- **Representación decimal:**

$$\begin{aligned} N_{(10)} &= a_k\, a_{k-1} \ldots a_1\, a_0 \\ &= a_k \cdot 10^k + a_{k-1} \cdot 10^{k-1} + \cdots + a_1 \cdot 10^1 + a_0 \cdot 10^0, \quad a_i \in \{0, 1, 2, \ldots, 9\} \quad \forall i \end{aligned}$$

EJEMPLO:
$N = 3459 = 3 \cdot 10^3 + 4 \cdot 10^2 + 5 \cdot 10^1 + 9 \cdot 10^0$

- **Representación binaria:**

$$\begin{aligned} N_{(2)} &= b_q\, b_{q-1} \ldots b_1\, b_0 \\ &= b_q \cdot 2^q + b_{q-1} \cdot 2^{q-1} + \cdots + b_1 \cdot 2^1 + b_0 \cdot 2^0, \quad b_i \in \{0, 1\} \quad \forall i \end{aligned}$$

EJEMPLO:
$N_{(2)} = 10111 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 23_{(10)}$

Para pasar de un número entero N del sistema decimal a binario se dividen sucesivamente dicho número y los cocientes sucesivos por 2, hasta llegar a un cociente igual a 1. Entonces se tiene $N_{(2)} = C_n R_n R_{n-1} \ldots R_2 R_1$, con $C_n = 1$, siendo R_k el resto de la k-ésima división, como se muestra en la Figura 2.1.

:2	N	
	C_1	R_1
	C_2	R_2
	C_3	R_3
	...	...
	C_n	R_n

Figura 2.1: Representación binaria de un número entero N.

EJEMPLO:

$N = 77_{(10)} = 1001101_{(2)}$

:2	77	
	38	1
	19	0
	9	1
	4	1
	2	0
	1	0

2.9.2 Representación de números reales

La parte entera de un número real se representa en base 2 igual que cualquier número entero. Para la parte fraccionaria se tiene:

- **Representación decimal:**

$$\begin{aligned} R_{(10)} &= 0.d_1\, d_2\, d_3\, \ldots\, d_n \ldots \\ &= d_1 \cdot 10^{-1} + d_2 \cdot 10^{-2} + \cdots + d_n \cdot 10^{-n} + \ldots \\ &= 0.d_1 + 0.0d_2 + \cdots + 0.00 \ldots d_n + \ldots, \quad d_i \in \{0, 1, \ldots, 9\} \quad \forall i \end{aligned}$$

EJEMPLO:

$R_{(10)} = 0.325 = 3 \cdot 10^{-1} + 2 \cdot 10^{-2} + 5 \cdot 10^{-3}$

- **Representación binaria:**

$$\begin{aligned} R_{(2)} &= 0.c_1 c_2 c_3 \ldots c_n c_{n+1} \ldots \\ &= c_1 \cdot 2^{-1} + c_2 \cdot 2^{-2} + \cdots + c_n \cdot 2^{-n} + \ldots, \quad c_i \in \{0, 1\} \quad \forall i \end{aligned}$$

EJEMPLO:

$$R_{(2)} = 0.111 = 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} = 0.875_{(10)}$$

El procedimiento (iterativo) para obtener la representación binaria de un número con parte fraccionaria es:

(a) Poner $R_0 = R$. Calcular $2R_0$ y tomar $c_1 = [2R_0]$ (parte entera de $2R_0$).

(b) Poner $R_1 = 2R_0 - c_1$. Calcular $2R_1$ y tomar $c_2 = [2R_1]$ (parte entera de $2R_1$).

(c) ...

(d) Poner $R_k = 2R_{k-1} - c_k$. Calcular $2R_k$ y tomar $c_{k+1} = [2R_k]$ (parte entera de $2R_k$), $\quad k \geq 1$.

EJEMPLO:

$R = 0.23_{(10)} = 0.0011101\ldots_{(2)}$

$$\begin{array}{lll}
R_0 = 0.23\ ; & 2R_0 = 0.46\ ; & c_1 = 0 \\
R_1 = 2R_0 - c_1 = 0.46\ ; & 2R_1 = 0.92\ ; & c_2 = 0 \\
R_2 = 2R_1 - c_2 = 0.92\ ; & 2R_2 = 1.84\ ; & c_3 = 1 \\
R_3 = 2R_2 - c_3 = 0.84\ ; & 2R_3 = 1.68\ ; & c_4 = 1 \\
R_4 = 2R_3 - c_4 = 0.68\ ; & 2R_4 = 1.36\ ; & c_5 = 1 \\
R_5 = 2R_4 - c_5 = 0.36\ ; & 2R_5 = 0.62\ ; & c_6 = 0 \\
R_6 = 2R_5 - c_6 = 0.62\ ; & 2R_6 = 1.24\ ; & c_7 = 1
\end{array}$$

Obsérvese en el ejemplo cómo un número decimal con parte fraccionaria finita en base 10 puede tener una parte fraccionaria infinita en base 2.

Lo contrario no es verdad: si $0.c_1c_2c_3\ldots c_n$ es la expresión en base 2 de un número fraccionario, se tiene:

$$0.c_1c_2c_3\ldots c_n = c_1 \cdot 2^{-1} + c_2 \cdot 2^{-2} + \cdots + c_n \cdot 2^{-n} = \frac{c_1}{2} + \frac{c_2}{2^2} + \cdots + \frac{c_n}{2^n}$$

donde los c_i valen 0 ó 1. Por tanto la expresión anterior no es más que una suma **finita** de términos de la forma $\frac{1}{2^k} = (\frac{1}{2})^k = 0.5^k$, que tiene siempre un número finito de cifras decimales.

2.9.3 Almacenamiento de números enteros

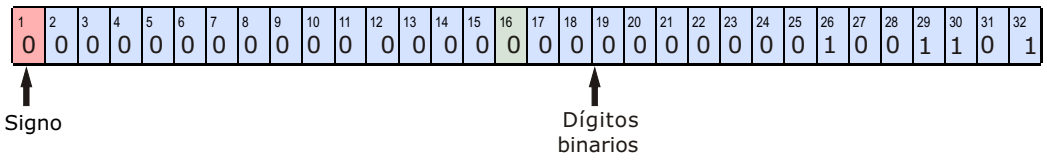
Los números enteros se almacenan en palabras de 32 bits, distribuidos de la siguiente manera [1]:

- 1 bit para el signo del número (0 si es positivo, 1 si es negativo)
- los 31 bits restantes son para almacenar los dígitos binarios del número sin signo.

[1] Esta explicación está simplificada. La codificación interna real es algo más complicada.

EJEMPLO:

$N = 77_{(10)} = 1001101_{(2)}$



En consecuencia, se tiene lo siguiente:

- El mayor entero positivo que se puede almacenar con este sistema es:

$$\underbrace{11\ldots11}_{31} = 2^{30} + 2^{29} + \cdots + 2^1 + 2^0 = 2^{31} - 1 = 2.147.483.647$$

- El menor entero negativo es: $-2.147.483.647$.
- El total de números enteros distintos representables: (doble representación para el cero) es:

$$2 \times 2^{31} - 1 = 2^{32} - 1 = 4.294.967.295.$$

2.9.4 Almacenamiento de números reales

Los números reales, habitualmente se representan en **coma flotante** (en inglés, *floating point*). Se trata de un modo "normalizado" de representación de números con parte fraccionaria que se adapta al orden de magnitud de los mismos, permitiendo así ampliar el rango de números representables.

Consiste en trasladar la coma (o punto) decimal hasta la posición de la primera cifra significativa (la primera, comenzando por la izquierda, que no es nula) a la vez que se multiplica el número por la potencia adecuada de la base de numeración.

EJEMPLO:

$17.02625 = 0.1702625 \times 10^2$ (en numeración decimal)

$0.0000000234 = 0.234 \times 10^{-7}$ (en numeración decimal)

$101.11000101 = 0.10111000101 \times 2^3$ (en numeración en base 2)

Sea, pues, un número R dado por la siguiente representación binaria en coma flotante:

$$R = (\pm)0.m_1m_2m_3\ldots m_k \cdot 2^{\pm e},$$

donde $m_1m_2m_3\ldots m_k$ se llama la **mantisa** y el número e (que escribimos aquí en forma de entero decimal) es el **exponente**, es decir, el número de lugares que hay que trasladar la coma a derecha o izquierda para obtener el número inicial. Obsérvese que m_1 tiene que ser siempre igual a 1, ya que es, por definición, la primera cifra no nula empezando por la izquierda.

EJEMPLO:

$R = 5.625_{(10)} = 101.101_{(2)} = 0.101101 \cdot 2^3$

Para almacenar dicho número, los 32 bits de una palabra se distribuyen [2] de la siguiente manera:

[2] Esta distribución puede variar ligeramente de un ordenador a otro.

- 1 bit para el signo del número (0 si es positivo, 1 si es negativo).
- 23 bits para la mantisa. Como m_1 es siempre igual a 1 se utiliza el pequeño " truco" de no almacenarlo. Así, aunque se almacenan 23 cifras binarias se "conocen" en realidad 24.
- 8 bits para el exponente con su signo (se guarda codificado como un número entero).

EJEMPLO:

$R = -5.625_{(10)} = -0.101101 \cdot 2^3, \ 3_{10} = 11_2$

Se tiene:

- El mayor exponente positivo que se puede almacenar: $1111111 = 2^6+2^5+\cdots+2^0 = 2^7-1 = 127$.
- El menor exponente negativo: -127.
- El número de mayor magnitud: $\approx 2^{127} \approx 10^{38}$.
- El número de menor magnitud: $\approx 2^{-127} \approx 10^{-38}$.

2.9.5 Overflow y underflow

Es claro que usando palabras de 32 bits sólo podemos representar una cantidad finita de números, en consecuencia podemos encontrar los siguientes fenómenos:

- **Overflow**: fenómeno que se produce cuando el resultado de un cálculo es un número real con exponente demasiado grande (en el ejemplo anterior > 127).
  - Cuando se produce un overflow durante la ejecución de un programa, los ordenadores modernos generan un "evento" de error y como resultado devuelven el símbolo `Inf`.
- **Underflow**: fenómeno que se produce cuando el resultado de un cálculo es un número real con exponente demasiado pequeño (en los ejemplos arriba mencionados < -127).
  - Cuando se produce durante la ejecución de un programa, normalmente se sustituye el número por cero.

2.9.6 Errores

Al realizar cálculos en el ordenador, es inevitable cometer errores. Grosso modo, estos errores pueden ser de tres tipos:

1. **Errores en los datos de entrada**: vienen causados por los errores al realizar mediciones de magnitudes físicas.

2. **Errores de truncamiento (o discretización)**: En la mayoría de las ocasiones, los algoritmos matemáticos que se utilizan para calcular un resultado no son capaces de calcular su valor exacto, limitándose a calcular una **aproximación** del mismo. Este error es inherente al algoritmo de cálculo utilizado.

3. **Errores de redondeo**: aparecen debido a la cantidad finita de números que podemos representar en un ordenador. Por ejemplo, consideramos las dos siguientes mantisas-máquina consecutivas que podemos almacenar en 23 bits (recordar que el primer dígito es siempre 1 y por tanto no se guarda):

$$R_1 = 0.\underbrace{10000000000000000000000}_{24 \text{ dígitos}}$$

$$R_2 = 0.\underbrace{10000000000000000000001}_{24 \text{ dígitos}}$$

Ningún número real entre R_1 y R_2 es representable en un ordenador que utilice 23 bits para la mantisa. Tendrá que ser aproximado bien por R_1, bien por R_2, cometiendo con ello un error E que verifica:

$$E \leq \frac{|R_1 - R_2|}{2} = \frac{2^{-24}}{2} = 2^{-25} \simeq 10^{-7}$$

puesto que $|R_1 - R_2| = 0.00000000000000000000001 = 0.1 \times 2^{-23} = 2^{-24}$. De esta forma, en el almacenamiento de la mantisa se comete un error menor o igual que 10^{-7} o, dicho de otra forma, sólo los 7 primeros dígitos significativos de un número decimal almacenado en el ordenador se pueden considerar exactos.

EJEMPLO:

$$\begin{aligned} R = 0.1_{(10)} &= 0.\widehat{00011}_{(2)} \\ &= 0.00011001100110011\ldots \\ &= = 0.\underbrace{11001100110011001100}_{24}1100\cdots \times 2^{-3} \end{aligned}$$

R no es un número-máquina (su mantisa binaria tiene infinitos dígitos).

La aproximación de R por redondeo es:

$$\overline{R} = 0.\underbrace{11001100110011001101}_{24 \text{ dígitos}} \times 2^{-3}.$$

Aritmética en coma flotante

Para llevar a cabo una operación aritmética en un ordenador, hay que tener en cuenta que los números con los que trabajamos pueden no ser exactamente los de partida, sino sus aproximaciones

por redondeo. Incluso aunque los números sean números-máquina (representables en palabras de 32 bits), los resultados que obtengamos puede que no lo sean. A continuación vamos a ver algunos ejemplos de cálculos efectuados en la aritmética de coma flotante, en el caso de un ordenador (hipotético) que trabaje con números reales en base 10, con 7 dígitos para la mantisa, y con aproximación por redondeo.

EJEMPLO:
Para sumar dos números en coma flotante, primero se representan con el mismo exponente, luego se suman y el resultado se redondea:

$$\begin{aligned} x_1 &= 0.1234567 \cdot 10^3 = 0.123456700000 \cdot 10^3 \quad (= 123.4567) \\ x_2 &= 0.3232323 \cdot 10^{-2} = 0.000003232323 \cdot 10^3 \quad (= 0.003232323) \end{aligned}$$

$$\begin{array}{rll} & 0.123456700000 & \cdot 10^3 \\ + & 0.000003232323 & \cdot 10^3 \\ \hline & 0.123459932323 & \cdot 10^3 \quad \text{(sólo se almacenan los 7 primeros dígitos)} \\ & 0.1234599 & \cdot 10^3 \quad \text{(resultado-máquina de } x_1 + x_2\text{)} \end{array}$$

Obsérvese que las 5 últimas cifras de x_2 no contribuyen a la suma.

EJEMPLO:
Para multiplicar (la división se hace por algoritmos más complicados) dos números en coma flotante se multiplican las mantisas y se suman los exponentes. El resultado se normaliza:

$$\begin{aligned} x_1 &= 0.4734612 \cdot 10^3 \\ x_2 &= 0.5417242 \cdot 10^5 \end{aligned}$$

$$\begin{array}{rll} & 0.4734612 & \cdot 10^3 \\ \times & 0.5417242 & \cdot 10^5 \\ \hline & 0.256485389801 04 & \cdot 10^8 \\ & 0.2564854 & \cdot 10^8 \quad \text{(resultado-máquina de } x_1 \cdot x_2\text{)} \end{array}$$