Capítulo 1

Aritmética del ordenador.

1. Sistemas de numeración.
2. Definiciones.
3. Almacenamiento de números enteros.
4. Almacenamiento de números reales.
5. Problemas con los números reales.

1.1. Sistemas de numeración.

Los ordenadores trabajan almacenando la información en el sistema binario. Sin embargo, la interacción con el ordenador se lleva a cabo en el sistema decimal. La transformación de la información entre ambos sistemas se realiza internamente en el ordenador y el usuario, en principio, no debe preocuparse. No obstante, es importante tener conciencia de los procedimientos particulares implicados por los problemas que podrían aparecer y que más adelante discutiremos.

El significado de los números es bien conocido. En la base decimal se tiene:

$$(714.72)_{10} = 7 \times 10^2 + 1 \times 10^1 + 4 \times 10^0 + 7 \times 10^{-1} + 2 \times 10^{-2}$$

Análogamente, en la base binaria podemos escribir:

$$\begin{aligned}(11001.001101)_2 &= 1\times 2^4 + 1\times 2^3 + 0\times 2^2 + 0\times 2^1 + 1\times 2^0 + 0\times 2^{-1}\\ &\quad + 0\times 2^{-2} + 1\times 2^{-3} + 1\times 2^{-4} + 0\times 2^{-5} + 1\times 2^{-6}\\ &= (25.203125)_{10}\end{aligned}$$

Este ejemplo nos indica cómo se realiza la transformación de binario a decimal.

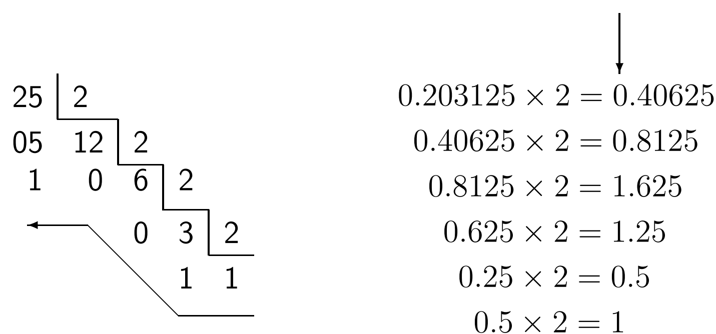
La transformación de un número decimal, $(N)_{10}$, a binario viene dada por

$$d_k = \operatorname{int}\left[\frac{(N)_{10}}{2^k} - \sum_{j=k+1}^{\infty} d_j\, 2^{j-k}\right],$$

donde d_k es el dígito correspondiente a la potencia 2^k. El mismo resultado puede obtenerse realizando sucesivas divisiones por 2 para la parte entera y sucesivas multiplicaciones por 2 para la parte decimal.

Ejemplo:
Encontrar la representación en binario del número $(25.203125)_{10}$

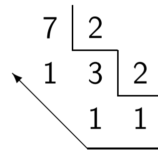
$(25.203125)_{10} = (11001.001101)_2$



Ejemplo:
Encontrar la representación en binario del número $(7.68436)_{10}$

$$(7.68436)_{10} = (111.101011110011\cdots)_2$$

- parte entera: $(7)_{10} = (111)_2$



- parte decimal: $(0.68436)_{10} = (0.101011110011\cdots)_2$

$$
\begin{aligned}
0.68436 \times 2 &= 1.36872\\
0.36872 \times 2 &= 0.73744\\
0.73744 \times 2 &= 1.47488\\
0.47488 \times 2 &= 0.94976\\
0.94976 \times 2 &= 1.89952\\
0.89952 \times 2 &= 1.79904\\
0.79904 \times 2 &= 1.59808\\
0.59808 \times 2 &= 1.19616\\
0.19616 \times 2 &= 0.39232\\
0.39232 \times 2 &= 0.78464\\
0.78464 \times 2 &= 1.56928\\
0.56928 \times 2 &= 1.13856\\
\vdots \quad & \quad \vdots
\end{aligned}
$$

En general, la transformación de decimal a binario puede producir números con un número de decimales mayor que el número decimal original o incluso infinito, como ocurre en el último de los ejemplos que acabamos de ver. Es evidente, por otro lado, que el ordenador no puede manejar un número infinito o demasiado grande de dígitos, ya que tiene limitaciones asociadas a la memoria que utiliza. Nótese que no sería ninguna solución trabajar directamente en el sistema decimal ya que también se presentarían situaciones similares con infinitos dígitos. Por tanto, en general se presentarán distintos problemas (redondeo, acotación, etc.) que habrá que tener presentes al calcular.

1.2. Definiciones.

1. **bit** (*binary digit*)
   Es el elemento básico de memoria. Puede tomar dos valores: 0 ó 1.

2. **byte**
   Es una agrupación de 8 bits y constituye la unidad básica de información.

3. **palabra** (word)
   Es el conjunto de bytes que utiliza el ordenador para almacenar un dato, por ejemplo, un número. El tamaño de la palabra caracteriza el sistema operativo (que es el conjunto de instrucciones que permiten interactuar con el ordenador). El antiguo MS-DOS tenía palabras de 2 bytes (16 bits). El Windows-XP tiene palabras de 4 bytes (32 bits). Existen versiones de Linux con esa misma longitud de palabra. Los modernos sistemas operativos están basados en palabras de 8 bytes (64 bits). Evidentemente la capacidad del sistema operativo debe estar en consonancia con el procesador del ordenador.

1.3. Almacenamiento de números enteros.

En primer lugar vamos a analizar el rango de números enteros que es posible manejar utilizando únicamente 1 byte de memoria. Como los números enteros tienen signo, hemos de reservar la memoria correspondiente para el

mismo. Para ello necesitamos sólo 1 bit. Si tenemos en cuenta que

$$(-1)^0 = +1 \qquad (-1)^1 = -1\,,$$

podemos utilizar el valor 0 del bit para representar el signo "+" y el valor 1 para el "−".

Los restantes 7 bits del byte los utilizamos para representar el valor absoluto del número entero. El menor número que podemos representar es, obviamente, el 0, mientras que el mayor número es

$$(1111111)_2 = (10000000 - 1)_2 = (2^7 - 1)_{10} = (127)_{10}$$

Por lo tanto, el rango de números posible con los 7 bits es

$$[0000000, 1111111]_2 = [0, 127]_{10}\ ,$$

con lo que 1 byte nos permite representar los números enteros en el rango

$$[-127, +127]_{10}\ .$$

Hay que señalar aquí que, con esta norma tenemos una doble representación para el 0: ± 0, lo que nos hace "perder" un número. Para solucionar esta deficiencia se pueden llevar a cabo distintos procedimientos.

Uno de los más utilizados sigue tomando el bit #1 para representar el signo del número, mientras que los siete bits restantes se emplean para representar el valor absoluto del número, si este es positivo, y el "complemento a dos" de ese valor absoluto si el número es negativo.[1] En la tabla 1.1 puede verse el resultado de este procedimiento que permite representar los números del rango

$$[-128, +127]_{10}\ ,$$

sin que el 0 tenga doble representación (hemos cambiado el -0 por el -128).

Otra posibilidad es sumar a los números un "sesgo" convenientemente elegido. En este caso se puede sumar a los números el sesgo $S = 2^7 - 1$. De esta forma todos los números en el rango $[-127, +127]$ serían positivos y no sería necesario almacenar el signo. Tendríamos disponibles 8 bits, lo que nos permitiría representar 2^8 números, igual que antes, pero ahora tendríamos el rango

$$\left[0, 2^8 - 1\right]_{10} - S = [0, 255]_{10} - S = [-127, +128]_{10}\ .$$

[1]Para determinar el "complemento a dos" de un número binario se cambian los "0" por "1", y viceversa, y al número binario resultante se le suma 1.

Tabla 1.1: Almacenamiento de números enteros.

Número decimal	Tipo de representación		
	signo y magnitud	complemento a 2	sesgada
+128	–	–	11111111
+127	01111111	01111111	11111110
+126	01111110	01111110	11111101
...	...	...	...
+3	00000011	00000011	10000010
+2	00000010	00000010	10000001
+1	00000001	00000001	10000000
+0	00000000	00000000	01111111
−0	10000000	–	–
−1	10000001	11111111	01111110
−2	10000010	11111110	01111101
−3	10000011	11111101	01111100
...	...	...	...
−126	11111110	10000010	00000001
−127	11111111	10000001	00000000
−128	–	10000000	–

Ahora hemos cambiado el -0 por el 128.

Las diferencias entre las representaciones pueden comprobarse en la tabla 1.1. La representación complemento a dos tiene la ventaja de permitir una mayor facilidad para la realización de algunas operaciones aritméticas (sumas y restas). La representación sesgada se utiliza para almacenar los exponentes en el caso de números reales.

Lo que hemos visto para estas representaciones puede extenderse a la utilización de un mayor número de bytes para representar números enteros. El rango de números que podrán representarse es

$$\begin{aligned}\left[-2^{15}, 2^{15}-1\right]_{10} &= [-32768, +32767]_{10} \text{ para 2 bytes y} \\ \left[-2^{31}, 2^{31}-1\right]_{10} &= [-2147483648, +2147483647]_{10} \text{ para 4 bytes,}\end{aligned}$$

si se utiliza la representación complemento a dos, y

$$\begin{aligned}\left[-2^{15}+1, 2^{15}\right]_{10} &= [-32767, +32768]_{10} \text{ para 2 bytes y} \\ \left[-2^{31}+1, 2^{31}\right]_{10} &= [-2147483647, +2147483648]_{10} \text{ para 4 bytes,}\end{aligned}$$

si se utiliza la representación sesgada.

En general se utilizan 2 ó 4 bytes para representar los números enteros.

1.4. Almacenamiento de números reales.

1.4.1. Notación científica normalizada.

Es común utilizar la denominada notación científica, notación exponencial o notación en coma flotante[2] para escribir los números reales:

$$\begin{aligned}714.72 &= 0.71472 \cdot 10^3 = 7147.2 \cdot 10^{-1} \\ -0.00622 &= -0.622 \cdot 10^{-2} = -62.2 \cdot 10^{-4}\end{aligned}$$

Se denomina notación científica normalizada a aquélla en la que los números reales se representan en la forma

$$x = \pm m \cdot 10^n ,$$

donde m verifica que $1 \leq m < 10$, con $m = 0$ si $x = 0$, y n es un entero (positivo, negativo o cero). Los dos ejemplos anteriores en notación normalizada serían $7.1472 \cdot 10^2$ y $-6.22 \cdot 10^{-3}$, respectivamente. De manera análoga, en el sistema binario podemos escribir los números reales en notación normalizada como

$$x = \pm m \cdot 2^n ,$$

donde m verifica que $1 \leq m < 2$, con $m = 0$ si $x = 0$, y n es un entero. El valor m se denomina mantisa y n es el exponente. Evidentemente, en el caso de los números en binario ambos deberán estar en dicho sistema de numeración.

[2]Se denomina así porque la coma decimal parece "flotar" según varíe el exponente.

1.4.2. Norma IEEE-754.

Simple precisión.

Vamos a analizar qué indica esta norma para almacenar los números reales y consideraremos inicialmente que tenemos disponibles 4 bytes para esta tarea. Es lo que se denomina "simple precisión".

Partiendo de la notación normalizada, necesitamos representar la mantisa y el exponente con sus respectivos signos (la base de numeración ya está predeterminada: 2). Es decir, necesitamos 2 bits para los signos y el resto puede emplearse para las magnitudes de ambas cantidades. Podemos emplear el bit #1 para el signo de la mantisa (el del número, de hecho), el bit #2 para el signo del exponente, los bits #3 al #9 para el valor absoluto del exponente y los bits #10 al #32 para la magnitud de la mantisa. Por tanto, disponemos de 7 bits para el exponente y 23 para la mantisa.

Sin embargo, y como ya hemos visto antes, en lugar de utilizar 1 bit para el signo del exponente y 7 para su valor absoluto, podemos usar la representación sesgada y tomar los 8 bits (del bit #2 al #9) para el exponente. En este caso consideramos un sesgo $S = 2^7 - 1$ y podemos representar los exponentes en el rango

$$[-127, +128] \; .$$

El signo de la mantisa se almacena como 0 (número positivo) o 1 (número negativo) en el bit #1.

Para el almacenamiento de la mantisa, lo primero que hay que tener en cuenta es que al verificar que $1 \leq m < 2$, el primer dígito significativo de la mantisa es siempre "1", seguido de "." y, por tanto, no hay que almacenarlo (aunque sí tenerlo en cuenta, claro): sólo hemos de almacenar la parte decimal de la mantisa. El almacenamiento se lleva a cabo en orden desde el punto decimal, de manera que en el bit #10 se almacena el dígito correspondiente a 2^{-1} y en el bit #32 el dígito correspondiente a 2^{-23}.

Este último dígito es el que nos va a marcar la precisión de los números reales almacenados en el ordenador. Como $2^{-23} = 1.2 \cdot 10^{-7}$ los números tendrán una precisión de sólo seis cifras decimales. Cualesquiera números reales con un número de cifras decimales superior a seis serán objeto de redondeo.

Evidentemente, si cambiamos la norma dando un bit más al exponente y uno menos a la mantisa, aumentamos el rango de números reales que podemos almacenar, pero a costa de perder precisión en los mismos.

El número de números reales distintos que podemos representar en simple precisión es de

$$2^{32} = 4.294967296 \cdot 10^9$$

únicamente. ¿Cuál es el mayor de ellos? Evidentemente será el representado como

$$0 \; 11111111 \; 11111111111111111111111\,.$$

El primer 0 nos indica que es un número positivo. El exponente es

$$(11111111)_2 - (127)_{10} = (2^8 - 1)_{10} - (127)_{10} = (255 - 127)_{10} = (128)_{10}\,,$$

como ya sabíamos. Por último, la mantisa viene dada por (hay que recordar el "1." implícito que no se almacena)

$$(1.11111111111111111111111)_2 = (2 - 2^{-23})_{10}\,.$$

Por tanto, el número buscado es

$$[(2 - 2^{-23}) \cdot 2^{128}]_{10} = (6.8056469 \cdot 10^{38})_{10}\,.$$

¿Cuál es el número real positivo más pequeño que puede representarse con 4 bytes según esta norma? Ese número es

$$0 \; 00000000 \; 00000000000000000000000\,.$$

De nuevo el primer 0 indica que el número es positivo. El exponente es, como ya vimos, -127, y la mantisa vale

$$(1.00000000000000000000000)_2 = (1.0)_{10}\,.$$

El número buscado es

$$(1.0 \cdot 2^{-127})_{10} = (5.8774718 \cdot 10^{-39})_{10}\,.$$

Sin embargo, en estas circunstancias, el 0 no estaría incluido entre los valores posibles. Por ello, muchos compiladores asignan el valor 0 cuando tanto el exponente como la parte decimal de la mantisa almancenados tienen 0 en todos sus dígitos. El número más pequeño que puede almacenarse no es exactamente el que acabamos de calcular, sino que es el siguiente:

$$(1.00000000000000000000001)_2 = (5.8774725 \cdot 10^{-39})_{10}\,.$$

Los números reales representados en simple precisión en el ordenador pertenecen al rango

$$[-6.8056469\cdot10^{38}\,,\,-5.8774725\cdot10^{-39}]+0+[5.8774725\cdot10^{-39}\,,\,6.8056469\cdot10^{38}]$$

Cuando en el desarrollo de un cálculo aparece un número cuya magnitud es inferior al valor de $5.9 \cdot 10^{-39}$ se produce un desbordamiento por defecto o *underflow* asignándose automáticamente el valor 0. Si lo que aparece es un número superior al valor máximo $6.8 \cdot 10^{38}$ entonces se produce un desbordamiento por exceso u *overflow* que da lugar a un aviso de error y la interrupción del cálculo.

Ejemplo:
Determinar el número decimal cuya representación es

0 10100011 10101000000000000000000

Ésta es una representación en simple precisión. El exponente y la mantisa son, respectivamente,

$$(10100011)_2 - (127)_{10} = (2^7 + 2^5 + 2^1 + 2^0 - 127)_{10} = (36)_{10}\,,$$
$$(1.10101)_2 = (1 + 2^{-1} + 2^{-3} + 2^{-5})_{10} = (1.65625)_{10}\,.$$

Por tanto

$$(x)_{10} = (1.65625 \cdot 2^{36})_{10} = (113816633344)_{10}$$

Ejemplo:
Determinar la representación en simple precisión de

$$x = (-28)_{10}$$

En binario tenemos

$$(28)_{10} = (11100)_2 = (1.11)_2 \cdot 2^4\,.$$

El exponente será

$$(4)_{10} + (127)_{10} = (100)_2 + (1111111)_2 = (10000011)_2$$

Por tanto la representación buscada es

$$1\ 10000011\ 11000000000000000000000\,,$$

donde el bit #1 se ha puesto a 1 por ser el número negativo.

Doble precisión.

En el caso de necesitar mayor precisión o números que estén fuera de las fronteras establecidas por la simple precisión, es necesario utilizar un número de bytes mayor para representar los números reales. La doble precisón utiliza 8 bytes en los que el almacenamiento se lleva a cabo de forma que el primer bit se sigue utilizando para el signo del número, los 11 siguientes para el exponente y los últimos 52 para la mantisa. Se sigue asumiendo el "1." implícito como en el caso de simple precisión. El exponente se almacena con sesgo. En este caso $S = 2^{10} - 1 = 1023$. El rango de exponentes es por tanto $[-1023, 1024]$.

El número de números reales que se pueden almacenar es ahora

$$2^{64} = 1.84467440737096 \cdot 10^{19}$$

El mayor de los números reales positivos que podemos representar es

$$[(2 - 2^{-52}) \cdot 2^{1024}]_{10} \approx [2 \cdot (2^{256})^4]_{10} = (3.5953862697246 \cdot 10^{308})_{10}\,,$$

mientras que el más pequeño es

$$\begin{aligned}(1.0 \cdot 2^{-1023})_{10} &= (2 \cdot 2^{-1024})_{10} = [2 \cdot (2^{256})^4]_{10} \\ &= [2 \cdot (8.63616855509444 \cdot 10^{-78})^4]_{10} \\ &= (2 \cdot 8.63616855509444^4 \cdot 10^{-312})_{10} \\ &= (1.11253692711512 \cdot 10^{-308})_{10}\,.\end{aligned}$$

Por último, como $2^{-52} = 2.22 \cdot 10^{-16}$ podemos almacenar números reales con hasta 15 cifras decimales exactas.

1.4.3. Números máquina.

Los números máquina son aquellos números reales que pueden almacenarse de forma exacta en el ordenador. Este conjunto de números es finito y por lo tanto siempre se producirán errores al introducir datos en un ordenador.

Ejemplo:
Determinar la representación en simple precisión de $x = (0.2)_{10}$.

$$\begin{aligned}(0.2)_{10} &= (1)_{10}/(5)_{10} = (1)_2/(101)_2 \\ &= (0.00110011001100110011001100110011001100\cdots)_2 \\ &= (1.1001100110011001100110011001100\cdots)_2 \cdot 2^{-3}\,.\end{aligned}$$

La representación buscada no es posible sin cometer algún error ya que la mantisa tiene más de los 23 dígitos (o los 52 en caso de doble precisión) reservados para ella (de hecho tiene infinitos).

En general un número en binario lo podemos escribir como

$$x = m \cdot 2^n = (1.d_1 d_2 d_3 \cdots d_{22} d_{23} d_{24} d_{25} \cdots)_2 \cdot 2^n .$$

Y nos preguntamos por los números máquina más próximos a x. Uno de ellos es

$$x' = (1.d_1 d_2 d_3 \cdots d_{22} d_{23})_2 \cdot 2^n ,$$

que es el que se obtiene mediante el procedimiento denominado truncamiento. Como vemos descartamos los dígitos decimales por encima del 23. O lo que es lo mismo, en la mantisa descartamos los bits por encima del 23. Evidentemente, $x > x'$.

El otro número máquina próximo a x es el que se obtiene mediante redondeo por exceso

$$x'' = (1.d_1 d_2 d_3 \cdots d_{22} d_{23})_2 \cdot 2^n + 1 \cdot 2^{-23} \cdot 2^n .$$

Este segundo número máquina cumple que $x < x''$ y

$$x'' = x' + 1 \cdot 2^{-23} \cdot 2^n ,$$

con lo que

$$x'' - x' = 2^{n-23} .$$

El ordenador elegirá entre ambos números máquina de forma que almacenará el más próximo a x de los dos, que denominaremos $\mathrm{fl}(x)$. Podemos escribir que

$$|\mathrm{fl}(x) - x| = \min\left(|x - x'|, |x'' - x|\right) \leq \frac{1}{2}\,|x'' - x'| = \frac{1}{2}\, 2^{n-23} = 2^{n-24} ,$$

y el error relativo es

$$|\delta| = \left|\frac{\mathrm{fl}(x) - x}{x}\right| \leq \frac{2^{n-24}}{m \cdot 2^n} \leq 2^{-24} ,$$

ya que $m \geq 1$. La cantidad $\varepsilon = 2^{-24}$ se denomina error de redondeo unitario para la representación en simple precisión.

Ejemplo:
Determinar la representación en simple precisión de $x = (0.2)_{10}$.
Tenemos

$$x = (0.2)_{10} = (1.100110011001\cdots)_2 \cdot 2^{-1}\,.$$

Los números máquina más proximos a x son

$$\begin{aligned} x' &= (1.10011001100110011001100)_2 \cdot 2^{-3} \\ x'' &= (1.10011001100110011001101)_2 \cdot 2^{-3} \end{aligned}$$

A partir de aquí resulta que

$$x - x' = (1.1001001\cdots)_2 \cdot 2^{-3} \cdot 2^{-24} = 0.2 \cdot 2^{-24}\,.$$

Por otra parte

$$x'' - x = (x'' - x') - (x - x') = 2^{-24} - 0.2 \cdot 2^{-24} = 0.8 \cdot 2^{-24}\,.$$

Por tanto, $\mathrm{fl}(x) = x'$. El error absoluto es

$$|\mathrm{fl}(x) - x| = 0.2 \cdot 2^{-24} \approx 1 \cdot 10^{-8}$$

y el error relativo valdrá

$$\frac{|\mathrm{fl}(x) - x|}{x} = \frac{1 \cdot 10^{-8}}{0.2} = 2^{-24} \approx 6 \cdot 10^{-8}\,.$$

Finalmente, la representación según la norma IEEE-754 será

$$0\ 01111100\ 10011001100110011001100\,,$$

ya que el exponente (-3) se expresa sesgado resultando

$$(124)_{10} = (1111100)_2\,.$$

El número máquina utilizado para representar el número x verifica

$$\mathrm{fl}(x) = x(1 + \delta)\,, \quad |\delta| \le \varepsilon\,.$$

Si $\odot$ representa cualquiera de las cuatro operaciones básicas $(+, -, *, /)$ y x e y son dos números máquina, entonces

$$\text{fl}(x \odot y) = (x \odot y)(1 + \delta)\,, \quad |\delta| \leq \varepsilon\,.$$

Por tanto ε proporciona una cota para el error relativo en cualquier operación básica. Pero si los números x e y no son números máquina resulta que

$$\begin{aligned}
\text{fl}(x \odot y) &= [x(1 + \delta_1) \odot y(1 + \delta_2)](1 + \delta_3) \\
&= (x \odot y)(1 + \delta_1)(1 + \delta_2)(1 + \delta_3) \\
&\sim (x \odot y)(1 + \delta_1 + \delta_2 + \delta_3) = (x \odot y)(1 + \delta)\,,
\end{aligned}$$

con $|\delta| \leq 3\epsilon$ y donde hemos tenido en cuenta que los productos de las δ son mucho menores que cada una de ellas individualmente.

Si calculamos $x(y+z)$ teniendo en cuenta que los tres son números máquina, entonces

$$\begin{aligned}
\text{fl}[x(y + z)] &= [x\text{fl}(y + z)](1 + \delta_1)\,, \quad |\delta_1| \leq \varepsilon \\
&= [x(y + z)(1 + \delta_2)](1 + \delta_1)\,, \quad |\delta_1| \leq \varepsilon\,, |\delta_2| \leq \varepsilon \\
&= x(y + z)(1 + \delta_2 + \delta_1 + \delta_2\delta_1) \\
&\approx x(y + z)(1 + \delta_1 + \delta_2) \\
&= x(y + z)(1 + \delta)\,, \quad |\delta| \leq 2\varepsilon\,,
\end{aligned}$$

donde hemos tenido en cuenta que el producto $\delta_1\delta_2$ es despreciable frente a cada uno de los dos factores.

Para minimizar los errores de este tipo, muchos ordenadores realizan los cálculos utilizando registros temporales especiales que pueden utilizar más bits que los asignados a los números máquina usuales. Los bits extra se denominan bits de protección y permiten obtener el resultado de la operación con una precisión adicional. En cualquier caso, al final se debe llevar a cabo el correspondiente redondeo para almacenar el resultado en un registro normal.

1.5. Problemas con los números reales.

Los errores de redondeo que acabamos de discutir son inevitables y, en algunas ocasiones, difíciles de controlar. Sin embargo, existen otro tipo de errores y problemas que sí es posible evitar. Vamos a ver aquí algunos de ellos.

1.5.1. Pérdida de dígitos significativos.

La pérdida de dígitos significativos es uno de estos problemas. Ocurre cuando se restan dos números próximos. Para entenderlo fácilmente, supongamos que queremos calcular la diferencia entre los números $x = 0.5450344923$ e $y = 0.5450276544$. Evidentemente, $x - y = 0.0000068379$. Supongamos que hiciéramos los cálculos con un ordenador que trabajara con números decimales con 6 cifras decimales en la mantisa. Está claro que, en tal caso, se calcularía la diferencia entre $\mathrm{fl}(x) = 0.545034$ y $\mathrm{fl}(y) = 0.545028$ que vale $\mathrm{fl}(x) - \mathrm{fl}(y) = 0.000006$. El error relativo que se comete es

$$\left|\frac{x - y - [\mathrm{fl}(x) - \mathrm{fl}(y)]}{x - y}\right| = \left|\frac{0.0000068379 - 0.000006}{0.0000068379}\right| \approx 12.3\,\%\,,$$

que resulta ser considerable. Por tanto hay que evitar las situaciones en las que se pueda dar lugar al cálculo de diferencias entre número parecidos.

Ejemplo:
Supongamos que $y = \sqrt{1+x^2} - 1$ y que queremos calcular y para valores de $x \sim 0$.
Si se programa la expresión tal y como está indicada se producirá una pérdida de precisión por pérdida de dígitos significativos ya que estaríamos calculando diferencias entre números próximos. El problema se puede evitar sencillamente de la forma siguiente:

$$y = \sqrt{1+x^2} - 1 = (\sqrt{1+x^2} - 1)\frac{\sqrt{1+x^2}+1}{\sqrt{1+x^2}+1} = \frac{x^2}{1+\sqrt{1+x^2}}\,,$$

donde no aparece ninguna resta.

Ejemplo:
Supongamos que $y = x - \operatorname{sen} x$ y que queremos calcular y para valores de $x \sim 0$.
Como sabemos, cuando $x \sim 0$, $\operatorname{sen} x \sim x$ y de nuevo tenemos un problema de pérdida de precisión por pérdida de dígitos significativos ya que estaríamos restando números próximos. La solución ahora la podemos encontrar sin más que tener en cuenta el desarrollo en serie de Taylor de la función seno alrededor de $x = 0$:

$$\operatorname{sen} x = \sum_{k=0}^{n} \frac{(-1)^k}{(2k+1)!} x^{2k+1} + E_{2n+2}(x)\,,$$

con

$$E_{2n+2} = \frac{x^{2n+2}}{(2n+2)!} \operatorname{sen}^{(2n+2)} \xi \,.$$

La idea es tomar n suficientemente grande para que $E_{2n+2}(x)$ sea despreciable. Como $\operatorname{sen}^{(2n+2)} \xi = (-1)^{n+1} \operatorname{sen} \xi$,

$$|E_{2n+2}(x)| = \left| \frac{x^{2n+2} \operatorname{sen} \xi}{(2n+2)!} \right| \leq \frac{x^{2n+2}}{(2n+2)!} \,.$$

Podemos entonces sustituir la diferencia directa $x - \operatorname{sen} x$ por

$$y = x - \operatorname{sen} x \approx \frac{x^3}{3!} - \frac{x^5}{5!} + \cdots + (-1)^{n+1} \frac{x^{2n+1}}{(2n+1)!}$$

y cortamos la serie de manera que el error cometido esté por debajo del límite que deseemos.

Desarrollo en serie de Taylor.

El desarrollo en serie de Taylor de la función $f(x)$ alrededor de un punto $x = x_0$ viene dado por

$$f(x) = \sum_{k=0}^{n} \frac{1}{k!} (x - x_0)^k f^{(k)}(x_0) + E_n(x)$$

donde

$$E_n(x) = \frac{1}{(k+1)!} f^{(n+1)}(\xi)(x - x_0)^{n+1} \,, \;\; \xi \in [x_0, x]$$

y $f^{(n)}(x_0)$ hace referencia a la derivada n-ésima de f evaluada en x_0.

1.5.2. Inestabilidad numérica.

Se dice que un cálculo numérico es inestable cuando los pequeños errores que se producen en alguna de las etapas del mismo se propagan a las etapas siguientes agrandándose y degradando la exactitud de los resultados. Este problema es típico de las evaluaciones que involucran relaciones de recurrencia.

Ejemplo:

Considerar la sucesión de números reales definida por

$$x_0 = x^0 = 1 \,, \quad x_1 = x^1 = x \,, \quad x_{n+1} = \frac{10}{3} x_n - x_{n-1} \,, \;\; n \geq 1 \,,$$

cuyo término general es $x_n = x^n$ en el caso en que $x = 3$ o $x = 1/3$.
Supongamos, en primer lugar que los valores iniciales se eligen de manera que $x_0 = 1$ y $x_1 = 3$. En la tabla podemos ver los valores obtenidos con la regla de recurrencia, los que proporciona el término general y el error relativo entre ambos resultados. Como vemos este error es nulo en todos los casos y, por tanto, la regla de recurrencia proporciona los resultados correctos.

n	x_n	3^n	error relativo
0	1.00000000E+00	1.00000000E+00	0.00000000E+00
1	3.00000000E+00	3.00000000E+00	0.00000000E+00
2	9.00000000E+00	9.00000000E+00	0.00000000E+00
3	2.70000000E+01	2.70000000E+01	0.00000000E+00
4	8.10000000E+01	8.10000000E+01	0.00000000E+00
5	2.43000000E+02	2.43000000E+02	0.00000000E+00
6	7.29000000E+02	7.29000000E+02	0.00000000E+00
7	2.18700000E+03	2.18700000E+03	0.00000000E+00
8	6.56100000E+03	6.56100000E+03	0.00000000E+00
9	1.96830000E+04	1.96830000E+04	0.00000000E+00
10	5.90490000E+04	5.90490000E+04	0.00000000E+00
11	1.77147000E+05	1.77147000E+05	0.00000000E+00
12	5.31441000E+05	5.31441000E+05	0.00000000E+00
13	1.59432300E+06	1.59432300E+06	0.00000000E+00
14	4.78296900E+06	4.78296900E+06	0.00000000E+00
15	1.43489060E+07	1.43489070E+07	6.96917226E−08
16	4.30467160E+07	4.30467200E+07	9.29222921E−08
17	1.29140152E+08	1.29140160E+08	6.19481995E−08
18	3.87420448E+08	3.87420480E+08	8.25975945E−08
19	1.16226138E+09	1.16226150E+09	1.10130124E−07
20	3.48678426E+09	3.48678451E+09	7.34200825E−08

Pero si lo valores iniciales los elegimos como $x_0 = 1$ y $x_1 = 1/3$, los resultados obtenidos con la regla de recurrencia y con la expresión del término general de la sucesión discrepan notablemente, tal y como indica el error relativo cometido. A partir de $n = 9$ se obtienen valores que no coinciden en ningún dígito y cabe decir pues que el resultado proporcionado por la regla de recurrencia es inaceptable. El problema es que los pequeños errores en la determinación de uno de los valores se propagan a los otros, incrementándose en cada paso. Además se producen pérdidas de precisión al calcular diferencias entre números próximos.

n	x_n	$(1/3)^n$	error relativo
0	1.00000000E+00	1.00000000E+00	0.00000000E+00
1	3.33333343E−01	3.33333343E−01	0.00000000E+00
2	1.11111164E−01	1.11111112E−01	4.69386578E−07
3	3.70371938E−02	3.70370373E−02	4.22447920E−06
4	1.23461485E−02	1.23456791E−02	3.80203128E−05
5	4.11663577E−03	4.11522621E−03	3.42522282E−04
6	1.37597043E−03	1.37174211E−03	3.08244606E−03
7	4.69932333E−04	4.57247370E−04	2.77420133E−02
8	1.90470717E−04	1.52415785E−04	2.49678418E−01
9	1.64970057E−04	5.08052617E−05	2.24710584E+00
10	3.59429454E−04	1.69350878E−05	2.02239494E+01
11	1.03312812E−03	5.64502943E−06	1.82015549E+02
12	3.08433105E−03	1.88167644E−06	1.63813989E+03
13	9.24797542E−03	6.27225461E−07	1.47432607E+04
14	2.77422518E−02	2.09075154E−07	1.32689328E+05
15	8.32261965E−02	6.96917155E−08	1.19420400E+06
16	2.49678418E−01	2.32305712E−08	1.07478370E+07
17	7.49035180E−01	7.74352404E−09	9.67305280E+07
18	2.24710560E+00	2.58117461E−09	8.70574784E+08
19	6.74131680E+00	8.60391536E−10	7.83517338E+09
20	2.02239494E+01	2.86797169E−10	7.05165558E+10

Demostración del término general de la sucesión.

Queremos demostrar que el término general de la sucesión del problema es $x_n = x^n$ cuando $x = 3$ o $x = 1/3$. Como vemos se cumple para $n = 0$ y $n = 1$. Para que se cumpla para $n = 2$, x debe tomar los valores 3 o $1/3$. Supongamos que es cierto para todos los valores $m \leq n$ y vamos a demostrarlo para $n + 1$:

$$\begin{aligned} x_{n+1} &= \frac{10}{3}x_n - x_{n-1} = \frac{10}{3}x^n - x^{n-1} \\ &= x^{n-1}\left(\frac{10}{3}x - 1\right) = x^{n-1}x_2 = x^{n-1}x^2 = x^{n+1}\,. \end{aligned}$$

donde, como debe cumplirse para x_2, limita los valores a los anteriormente citados.

Ejemplo:
Considerar la sucesión de números reales definida por

$$y_n = \int_0^1 \mathrm{d}x\, x^n\, e^x \,, \quad n \geq 0 \,.$$

De aquí resulta que $y_0 = e - 1$. Para $n \geq 1$, integramos por partes con lo que

$$y_n = x^n\, e^x\big|_0^1 - n \int_0^1 \mathrm{d}x\, x^{n-1}\, e^x = e - n y_{n-1} \,, \quad n \geq 1 \,.$$

Los valores que se obtienen a partir de la expresión anterior se muestran en la segunda columna de la tabla. Estos resultados son incorrectos. En efecto, es fácil comprobar que

$$\lim_{n\to\infty} y_n = 0 \,.$$

Pero como vemos, a partir de $n = 10$ los valores empiezan a crecer en valor absoluto y a tener un signo alternado. De nuevo el problema está originado por el cálculo de diferencias entre números próximos y la propagación de los errores.

n	y_n	α_n	error relativo
0	1.718282E+00	1.718282E+00	0.000000E+00
1	1.000000E+00	1.000000E+00	0.000000E+00
2	7.182818E−01	7.182819E−01	8.298225E−08
3	5.634364E−01	5.634363E−01	1.057877E−07
4	4.645363E−01	4.645365E−01	5.132397E−07
5	3.956004E−01	3.955995E−01	2.260037E−06
6	3.446792E−01	3.446846E−01	1.556327E−05
7	3.055274E−01	3.054900E−01	1.223350E−04
8	2.740627E−01	2.743615E−01	1.089067E−03
9	2.517173E−01	2.490280E−01	1.079919E−02
10	2.011085E−01	2.280015E−01	1.179509E−01
11	5.060878E−01	2.102652E−01	1.406903E+00
12	−3.354772E+00	1.950999E−01	1.819515E+01
13	4.633032E+01	1.819827E−01	2.535863E+02
14	−6.459063E+02	1.705237E−01	3.788780E+03
15	9.691313E+03	1.604263E−01	6.040875E+04
16	−1.550583E+05	1.514609E−01	1.023752E+06
17	2.635994E+06	1.434468E−01	1.837611E+07
18	−4.744788E+07	1.362399E−01	3.482671E+08
19	9.015097E+08	1.297239E−01	6.949450E+09
20	−1.803019E+10	1.238038E−01	1.456352E+11

La solución del problema es utilizar el desarrollo en serie de la función exponencial. Tenemos entonces

$$\alpha_n = \int_0^1 \mathrm{d}x\, x^n \sum_{k=0}^{\infty} \frac{x^k}{k!} = \sum_{k=0}^{\infty} \frac{1}{k!} \int_0^1 \mathrm{d}x\, x^{n+k} = \sum_{k=0}^{\infty} \frac{1}{(n+k+1)\, k!}\,.$$

Los valores obtenidos de esta forma se muestran en la tabla (tercer columna). En la suma hemos considerado hasta $k = 30$ y vemos cómo verifican el comportamiento en el límite antes indicado. En la última columna de la tabla se relacionan los errores relativos cometidos al utilizar la regla de recurrencia.