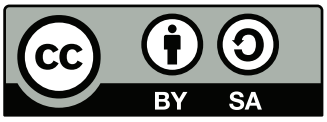
Universidad Complutense de Madrid

Laboratorio de Computación Científica
Laboratory for Scientific Computing
Aritmética del Computador ※ Computer Arithmetic

Juan Jiménez
Héctor García de Marina
Lía García

2 de septiembre de 2024



Índice general

Índice de figuras

Índice de cuadros

Capítulo/Chapter 1

Aritmética del Computador y Fuentes de error
Computer Arithmetics and Errors

> "What's the use?" a few hundred rivets chattered. "We've given - we've given, and the sooner we confess that we can't keep the ship together and go off our little heads, the easier it will be. Not rivet forged can stand this strain".
> "No one rivet was ever meant to. Share it among you", the Steam answered.
>
> Rudyard Kipling, The Ship that Found Herself

En el capítulo 1, introdujimos la representación binaria de números así como la conversión de binario a decimal y decimal a binario. A lo largo de este capítulo vamos a profundizar más en el modo en que el ordenador representa y opera con los números así como en una de sus consecuencias inmediatas: la imprecisión de los resultados.

1.1. Representación binaria y decimal

Los números reales pueden representarse empleando para ello una recta que se extiende entre $-\infty$ y $+\infty$. La mayoría de ellos no

In chapter one, we introduced the binary representation of numbers and the conversion from decimal to binary and from binary to decimal. In this chapter, we will get deep into how a computer represents and operates numbers. We also discuss one of its direct consequences: the imprecision of the results.

1.1. Binary and denary representations

Real numbers can be represented on a line extending from $-\infty$ to ∞. Most do not have exact numerical representations because they have infinite decimals.

admiten una representación numérica exacta, puesto que poseen infinitos decimales.

Los números enteros, $1, -1, 2, -2, 3, -3, \cdots$ admiten una representación numérica exacta. En cualquier caso, rara vez manejamos números enteros con una cantidad grande de dígitos, habitualmente, los aproximamos, expresándolos en notación científica, multiplicándolos por una potencia de 10. Tomemos un ejemplo de la química: la cantidad de átomos o moléculas que constituyen un mol de una sustancia se expresa por el número de Avogadro, $N_A = 6.02214179 \times 10^{23}$, dicho numero, –que debería ser un entero– se expresa empleando tan solo 9 de sus 23 cifras significativas (a veces tan solo se dan las tres primeras).

Cuando truncamos un número, es decir, despreciamos todos sus dígitos hacia la derecha a partir de uno dado, estamos aproximando dicho número por un número racional, es decir, por un número que puede expresarse como el cociente entre dos números enteros. $1/2, 3/4, \cdots$

Algunos números racionales se reducen a su vez a números enteros, 6/3, otros dan lugar a números cuya representación exige un número finito de dígitos:

The integer numbers $1 - 1, 2, -2, 3 - 3, \cdots$ can be represented in scientific notation. However, we rarely deal with numbers containing a large amount of digits. Usually, we take an approximate value, written then in scientific notation, by multiplying the number by a power of 10. Let's take an example from chemistry: the number of atoms or molecules that make up a mole of a chemical substance is defined as Avogadros's number $N_A = 6.02214179 \times 10^{23}$. This number should be an integer, but we represent it in scientific notation given just a few, nine in our example, of its 23 digits.

We truncate a number when we miss off digits past a certain point in the number. Then, we approximate the truncated number by a rational number, i.e., a number that the quotient of two integers can represent. $1/2, 3/4, cdots$

Some fractional numbers reduce, in turn, to integer numbers 6/3. Others are numbers that need an infinity number of digits to be represented:

$$11/2 = (5.5)_{10} = 5 \times 10^0 + 5 \times 10^{-1}$$

Si representamos el mismo número en base 2 obtenemos,

If we represent the same number in base 2 we obtain,

$$11/2 = (101.1)_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1}$$

Sin embargo, el que un número racional admita una representación finita, depende de la base en que se representa. Por ejemplo: $1/10 = (0.1)_{10}$ no admite representación finita en base 2.

However, A rational number may or may not have a finite representation, depending on the base used to represent it. For example, $1/10 = (0.1)_{10}$ has no finite representation in base 2.

$$1/10 = (0.0001100110011\cdots)_2 = 1 \times 2^{-4} + 1 \times 2^{-5} + 0 \times 2^{-6} + 0 \times 2^{-7} + 1 \times 2^{-8} + \cdots$$

En este último caso, se trata de una representación que, aunque no termina, es repetitiva o periódica; tras el primer cero decimal se repite indefinidamente la secuencia: 0011. Los números racionales admiten siempre una representación finita ó infinita periódica.

El número racional $1/3 = (0.333\cdots)_{10} =$

For this case, the number representation is infinite but repetitive or periodic; following the first zero after the decimal point, the sequence 0011 repeats endlessly. Rational numbers always have a finite representation or a periodic infinite one.

$(0.010101\cdots)_2$ solo admite representación infinita periódica tanto en base 10 como en base 2. Sin embargo, admite representación finita si lo representamos en base 3: $1/3 = (0.1)_3 = 0 \times 3^0 + 1 \times 3^{-1}$.

Habitualmente, en la vida ordinaria, representamos los números en base 10, sin embargo, como hemos visto en el capítulo 1 el ordenador emplea una representación binaria. Como acabamos de ver, una primera consecuencia de esta diferencia, es que al pasar números de una representación a otra estemos alterando la precisión con la que manejamos dichos números. Esta diferencia de representación supone ya una primera fuente de errores, que es preciso tener en cuenta cuando lo que se pretende es hacer cálculo científico con un ordenador. Como iremos viendo a lo largo de este capítulo, no será la única.

1.2. Representación de números en el ordenador

Enteros positivos La forma mas fácil de representar números sería empleando directamente los registros de memoria para guardar números en binario. Si suponemos un ordenador que tenga registros de 16 bits. Podríamos guardar en ellos números comprendidos entre el 0 y el $2^{16} - 1 = 65535$. Este último se representaría con un 1 en cada una de las posiciones del registro, en total 16 unos. Se trata del entero más grande que podríamos representar con un registro de 16 bits:

posición	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
valor	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Hasta ahora, nuestro sistema de representación solo admite números enteros positivos. El menor número que podemos representar es el cero, y el mayor —dependerá del tamaño n de los registros del ordenador—, $2^n - 1$. Es evidente que este sistema de representación resulta insuficiente.

Enteros positivos y negativos. Un primera mejora consistiría en ampliar el sistema de representación para que admita también números negativos. Una posibilidad sería

The rational number $1/3 = (0.333\cdots)_{10} = (0.010101\cdots)_2$ has a periodic infinite representation both in basis ten and in basis 2. However, it has a finite representation in base 3: $1/3 = (0.1)_3 = 0 \times 3^0 + 1 \times 3^{-1}$.

Usually, in ordinary life, we represent the number using base-ten. However, as we saw in chapter **??**, computers use a binary representation. The first consequence of this difference is that we alter number precision when we change them from one representation to another. Thus, this representation difference is, in fact, the first cause of errors that we must take into account when we want to use a computer to perform scientific computing. In this chapter, we will find more causes.

1.2. Computer number representation

positive integers. The most straightforward method for representing numbers would be using the memory registers directly to store the numbers in binary format. Considering a computer with 16-bit size registers, we can save numbers between 0 and $2^{16} - 1 = 65535$. This maximum number would be represented by filling each register position with a 1, so 16 ones in total. This number is the largest one we can represent using 16 bits:

Our current representation system only accepts positive integer numbers. The lowest number we can represent is zero, and the largest one –which will depend on the computer register size–, would be $2^n - 1$. It's clear that this representation system is insufficient.

Positive and negative integers. A first improvement would be to expand the system so that it also admits negative numbers. One option would be to reserve a bit of the register to represent the sign, using the remaining re-

reservar uno de los bits del array para representar el signo , y usar los restantes para representar el número. En nuestro ejemplo de un registro de 16 bits, el número más grande sería ahora $2^{15} - 1$ y el más pequeño sería $-2^{15} + 1$. El cero tendría una doble representación, una de ellas con signo más y la otra con signo menos.

Una representación alternativa, cuyo uso está muy extendido en las máquinas, es la representación conocida con el nombre de *Complemento a 2* . Supongamos un registro de n bits:

- Un entero x no negativo se almacena empleando directamente su representación binaria. el intervalo de enteros no negativos representables es: $0 \leq x \leq 2^{(n-1)} - 1$.
- Un entero negativo $-x$, donde $1 \leq x \leq 2^{(n-1)}$, se almacena como la representación binaria del entero positivo: $2^n - x$
- El número así construido, recibe el nombre de complemento a 2 de x.

En general, dado un número entero $N < 2^n$ su complemento a dos en una representación binaria de n dígitos se define como:

$$C_2^N = 2^n - N$$

En la tabla 1.1 se muestra un ejemplo de esta representación, para el caso de un registro de 4 bits:

La representación en complemento a 2 tiene las siguientes características:

- La representación de los números positivos coincide con su valor binario
- La representación del número 0 es única.
- El rango de valores representables en una representación de N bits es $-2^{N-1} \leq x \leq 2^{N-1} - 1$.
- Los números negativos se obtienen como el complemento a dos, cambiado de signo, del valor binario que los representa.

gister to represent the (absolute value of the) number. Returning to our example of a 16-bit register, the largest number representable would now be $2^{15} - 1$ and the lowest $-2^{15} + 1$. Zero would have a double representation, one with a plus sign and another with a minus sign.

An alternative representation, frequently used by computers, is the *2's complement* representation. We will describe the representation for a generic n-bits register:

- A non-negative integer is stored using directly its binary representation. The range of representable non-negative numbers is $0 \leq x \leq 2^{(n-1)} - 1$.
- A negative integer $-x$ where $1 \leq x \leq 2^{(n-1)}$, is stored as the binary representation of the positive integer $2^n - x$.
- The number built following the procedure just described is called the 2's complement of x.

In general, we define the 2's complement of an integer $N < 2^n$ in a binary representation of n digits as:

$$C_2^N = 2^n - N$$

Table 1.1 shows an example of the 2's complement representation for a 4-bit register.

2's complement representation has the following properties:

- For a positive integer, its 2's complement representation fits its standard binary representation.
- We get a single representation for the number zero.
- The range of representable values in an N-bit representation is $-2^{N-1} \leq x \leq 2^{N-1} - 1$.
- We obtain the negative number by computing the 2's complement of their bi-

Decimal	(4 bits)	$C_2 = 2^n - x$	n. represent-ado-ed
15	1111	$(16-1) = 15$	-1
14	1110	$(16-2) = 14$	-2
13	1101	$(16-3) = 13$	-3
12	1100	$(16-4) = 12$	-4
11	1011	$(16-5) = 11$	-5
10	1010	$(16-6) = 10$	-6
9	1001	$(16-7) = 9$	-7
8	1000	$(16-8) = 8$	-8
7	0111		7
6	0110		6
5	0101		5
4	0100		4
3	0011		3
2	0010		2
1	0001		1
0	0000		0

Tabla 1.1: Representación en complemento a dos para un registro de 4 bits
Table 1.1: 2's complement representation for a 4-bit register

Una forma práctica de obtener el complemento a dos de un número binario es copiar el número original de derecha a izquierda hasta que aparezca el primer 1. Luego de copiar el primer 1, se complementan el resto de los bits del número (si es 0 se pone un 1 y si es 1 se pone un 0). Por ejemplo el complemento a dos del número 2, (0010 en una representación de 4 bits) sería 1110,

nary representation and changing the sign of the result

A practical method for computing the 2's complement of a binary number is to copy the number from right to left till we reach the first 1. Once we have copied the first 1, we *complement* the remaining number bits. This means changing the zeros to ones and the ones to zeros. For instance, the 2'complemet of the number 2 (0010 in a -bit representation) will be 1110,

$$
\begin{aligned}
001\mathbf{0} &\xrightarrow[\textit{copy the right end bit}]{\textit{copia bit derecha}} \cdots 0 \\
00\mathbf{1}0 &\xrightarrow[\textit{copy first bit=1}]{\textit{copia primer bit=1}} \cdot\cdot 10 \\
0\mathbf{0}10 &\xrightarrow[\textit{complement bit 0}]{\textit{complemento bit 0}} \cdot 110 \\
\mathbf{0}010 &\xrightarrow[\textit{complement bit 0}]{\textit{complemento bit 0}} 1110
\end{aligned}
$$

Una propiedad importante de la representación de complemento a dos, es que la diferencia de dos números puede realizarse directamente sumando al primero el complemento a dos del segundo. Para verlo podemos usar los dos números del ejemplo anterior: 0010 es

An important property of 2's complement representation is that we can compute the difference between two numbers just by adding the first number to the 2's complement of the second one. We can see it using the two numbers of the previous example: 0010 is the bi-

la representación binaria del número 2, si empleamos un registro de cuatro bits. Su complemento a dos, 1110 representa al número −2. Si los sumamos [1]:

[1]Sumar en binario es como sumar en decimal. Se procede de bit a bit, de derecha a izquierda y cuando se suman 1 y 1, el resultado es 10 (base 2), de modo que el bit resultante se coloca en 0 y se lleva el 1 hacia el siguiente bit de la izquierda.

nary representation of the number 2 if we use a 4-bit representation. it 2's complement, 1110 represent the number −2. If we add them [1]:

[1]the addition of binary numbers is like the addition of decimal numbers. We add then, bitwise, from right to left, and when we add 1 and 1, the result is 10 (base 2). So, the resulting bit is set to zero, and we carry on the 1 to the next left bit.

	0	0	1	0
	1	1	1	0
1	0	0	0	0
	0	0	0	0

El resultado de la suma nos da un número cuyos primeros cuatro bits son 0. El bit distinto de cero, no puede ser almacenado en un registro de cuatro bits, se dice que el resultado de la operación *desborda* el tamaño del registro. Ese bit que no puede almacenarse se descarta al realizar la operación de suma, con lo que el resultado sería cero, como corresponde a la suma de 2+(−2). Esta es la motivación fundamental para emplear una representación en complemente a dos: No hace falta emplear circuitos especiales para calcular la diferencia entre dos números, ya que ésta se representa como la suma del minuendo con el complemento a dos del sustraendo.

1.2.1. Números no enteros: Representación en punto fijo y en punto flotante

La representación de números no enteros se emplea para representar de modo aproximado números reales. Como hemos dicho antes, los números racionales periódicos y los números irracionales no pueden representarse de forma exacta mediante un número finito de decimales. Por esta razón, su representación es siempre aproximada. Para los números racionales no periódicos, la representación será exacta o aproximada dependiendo del número que se trate, el tamaño de los registros del ordenador y el sistema empleado para la representación. Los dos sistemas de representación más conocidos son la representación en

The result of the addition yields a number with its first four bits as 0. The bit with value one cannot be accommodated in a 4-bit register. We refer to the result as an *overflow* of the register size. This bit that can not be stored is discarded after the addition. Thus, the result will be zero, as can be expected after adding 2 + (−2). This is a fundamental motivation to use the 2's complement representation: we do not need any additional special circuits to compute the difference between two numbers because we can represent it straightforwardly by adding the minuend plus the 2's complement of the subtrahend.

1.2.1. Non-integer numbers: fixed-point and floating-point representations.

We use the non-integer representation to approximately represent real numbers. As we have seen before, periodic rational numbers and irrational numbers cannot be exactly represented using a finite number of decimals. This is the reason why their representation is always an approximation. In the case of non-periodic rational numbers, the representation will be exact or approximated depending on the number itself, the size of the computer registers and the representation system we use. The two most common representation systems are the fixed-point and the floating-point representations.

punto fijo y, de especial interés para el cálculo científico, la representación en punto flotante.

Representación en punto fijo. En la representación en punto fijo, el registro empleado para almacenar un número se divide en tres partes:

- 1 bit para almacenar el signo del número.
- Un campo de bits de tamaño fijo para representar la parte entera del número.
- Un campo para almacenar la parte decimal del número

Por ejemplo, si tenemos un registro de 16 bits podemos reservar 1 bit para el signo, 7 para la parte entera y 8 para la parte decimal. La representación en punto fijo del número binario -10111.0011101101 sería:

s	p. entera/ integer p.							p. decimal/ decimal p.							
1	0	0	1	0	1	1	1	0	0	1	1	1	0	1	1

Es interesante notar cómo para representar la parten entera, nos sobran dos bits en el registro, ya que la parte entera solo tiene cinco cifras y tenemos 7 bits para representarla. Sin embargo, la parte decimal tiene 10 cifras; como solo tenemos 8 bits para representar la parte decimal, la representación trunca el número eliminando los dos últimos decimales.

Si asociamos cada bit con una potencia de dos, en orden creciente de derecha a izquierda, podemos convertir directamente el número representado de binario a decimal,

s	p. entera / integer p.							p. decimal / decimal p.							
1	0	0	1	0	1	1	1	0	0	1	1	1	0	1	1
-	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}

Por tanto el número representado sería,

$$-(0\cdot 2^6+0\cdot 2^5+1\cdot 2^4+0\cdot 2^3+1\cdot 2^2+1\cdot 2^1+1\cdot 2^0+0\cdot 2^{-1}+0\cdot 2^{-2}+1\cdot 2^{-3}+1\cdot 2^{-4}+$$
$$+1\cdot 2^{-5}+0\cdot 2^{-6}+1\cdot 2^{-7}+1\cdot -8)=-23.23046875$$

De modo análogo podemos calcular cual sería el número más grande representable,

fixed-point representation. To represent a number using a fixed-point representation, we divide the register into three parts:

- We set aside one bit to store the number sign.
- We represent the integer part of the number using a fixed-size field of bits.
- We represent the decimal part of the number using also a fixed-size field of bits.

For instance, if we are using a 16-bit register we can take a bit to represent the number sign, seven bits for the integer part and eight bits for the decimal part. following this register sharing, the representation of the number -10111.0011101101 would be:

It is interesting to notice how we have two extra bits to represent the integer part of the number because the integer part has only five digits and we have seven bits to represent it. However, to represent the decimal part of the number, we have ten digits and only eight bits to represent it. So, the representation truncates the number cutting out the last two decimals.

If we associate each bit with a power of two, in increasing order from right to left, we can straightforwardly convert the number from its binary representation to its decimal form.

Then, the number represented will be,

Similarly, we can calculate the largest representable number,

s	p. entera/ integer p.							p. decimal/decimal p.							
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}

$$1\cdot 2^6+1\cdot 2^5+1\cdot 2^4+1\cdot 2^3+1\cdot 2^2+1\cdot 2^1+1\cdot 2^0+1\cdot 2^{-1}+1\cdot 2^{-2}+1\cdot 2^{-3}+1\cdot 2^{-4}+$$
$$+1\cdot 2^{-5}+1\cdot 2^{-6}+1\cdot 2^{-7}+1\cdot 2^{-8}=127.99609375$$

El número menor representable,

The lowest representable number,

s	p. entera / integer p.							p. decimal/ decimal p.							
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
-	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}

$$-(1\cdot 2^6+1\cdot 2^5+1\cdot 2^4+1\cdot 2^3+1\cdot 2^2+1\cdot 2^1+1\cdot 2^0+1\cdot 2^{-1}+1\cdot 2^{-2}+1\cdot 2^{-3}+1\cdot 2^{-4}+$$
$$+1\cdot 2^{-5}+1\cdot 2^{-6}+1\cdot 2^{-7}+1\cdot 2^{-8})=-127.99609375$$

El número entero más grande,

The largest integer,

s	p. entera / integer p.							p. decimal/ decimal p.							
0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}

$$1\cdot 2^6+1\cdot 2^5+1\cdot 2^4+1\cdot 2^3+1\cdot 2^2+1\cdot 2^1+1\cdot 2^0+0\cdot 2^{-1}+0\cdot 2^{-2}+0\cdot 2^{-3}+0\cdot 2^{-4}+$$
$$+0\cdot 2^{-5}+0\cdot 2^{-6}+0\cdot 2^{-7}+0\cdot 2^{-8}=127$$

El número decimal positivo más próximo a 1,

The positive decimal number closest to 1,

s	p. entera / integer p.							p. decimal/ decimal p.							
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}

$$0\cdot 2^6+0\cdot 2^5+0\cdot 2^4+0\cdot 2^3+0\cdot 2^2+0\cdot 2^1+0\cdot 2^0+1\cdot 2^{-1}+1\cdot 2^{-2}+1\cdot 2^{-3}+1\cdot 2^{-4}+$$
$$+1\cdot 2^{-5}+1\cdot 2^{-6}+1\cdot 2^{-7}+1\cdot 2^{-8}=0.99609375$$

O el número decimal positivo más próximo a 0,

Or the positive decimal number closest to 0,

s	p. entera / integer p.							p. decimal/ decimal p.							
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}

$$0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} + \\ + 0 \cdot 2^{-5} + 0 \cdot 2^{-6} + 0 \cdot 2^{-7} + 1 \cdot 2^{-8} = 0.00390625$$

La representación en punto fijo fue la más usada en los primeros computadores. Sin embargo, es posible con el mismo tamaño de registro representar un espectro mucho más amplio de números empleando la representación en punto flotante.

Representación en punto flotante. La representación de números en el ordenador en formato de punto flotante, es similar a la forma de representar los números en la llamada notación científica. En notación científica los números se representan como el producto de una cantidad por una potencia de diez:

The fixed point representation was widely used in the first computers. However, using the same register size, it is possible to represent a much wider spectrum of numbers using the floating-point representation.

floating-point representation. The floating-point representation of numbers in a computer is similar to the numeric representation known as scientific notation. In scientific notation, number are represented as the product of a quantity by a power of ten:

$$234.25 \equiv 2.345 \times 10^2,\ 0.000234 \equiv 2.34 \times 10^{-5},\ -56.238 \equiv -5.6238 \times 10^1.$$

La idea es desplazar el punto decimal hasta que solo quede una cifra significativa en la parte entera del número y multiplicar el resultado por 10 elevado a la potencia adecuada para recuperar el número original.

Muchas calculadoras y programas de cálculo científico presentan los resultados por pantalla en formato científico. Habitualmente lo hacen con la siguiente notación,

We shift the decimal point till just a single figure remains at its left, i.e, only a figure is use to represent the integer part of the number. We need to multiply the result for the power of ten that allows us to recovery the original number.

Many calculator a scientific computing software show the computed results on the screen using the scientific format. Usually they utilize the following notation to do it,

$$-5.3572 \times 10^{-3} \xrightarrow[calculator]{calculadora} -5.3572e-03$$

Es decir, en primer lugar se representa el signo del número si es negativo (si es positivo lo habitual es omitirlo). A continuación la parte significativa, 5.3572, que recibe el nombre de mantisa . Y por último se representa el valor del exponente de la potencia de 10, -3, precedido por la letra e, —e de exponente—. En notación científica se asume que el exponente corresponde siempre a una potencia de diez. Es evidente que tenemos el número perfectamente descrito sin más que indicar los valores de su signo, mantisa y exponente,

First, if the number is negative, we represent the sign. (Usually, if the number is positive we omit the sign symbol). Then, the part of the number 5.3572 which, when multiplied by the power of ten, retrieves the number represented, this part is called the mantissa of the number. Eventually, we represent the value of the (10) power's exponent, -3, preceded by the letter e, which stand for exponent. Scientific notation always assumes that the exponent belongs to a power of ten. It is clear that we have the number thoroughly described if we supply its sing, mantissa and exponent.

numero number	r. científica scientific r.	r. calculadora calculator r.	signo sign	mantisa mantissa	exponente exponent
-327.43	-3.2743×10^2	$-3.2743e2$	$-$	3.2743	2

La representación binaria en punto flotante sigue exactamente la misma representación que acabamos de describir para la notación científica. La única diferencia es que en lugar de trabajar con potencias de diez, se trabaja con potencias de dos, que son las que corresponden a la representación de números en binario.
Así, el número binario -1101.00101 se representaría como -1.10100101×2^3, y el número 0.001011101 se representaría como 1.011101×2^{-3}.

El término punto flotante viene del hecho de que el punto decimal de la mantisa, no separa la parte entera del número de su parte decimal. Para poder separar la parte entera de la parte decimal del número es preciso emplear el valor del exponente. Así, para exponente 0, el punto decimal de la mantisa estaría en el sitio que corresponde al número representado, para exponente 1 habría que desplazarlo una posición a la izquierda, para exponente -1 habría que desplazarlo una posición a la derecha, para exponente 2, dos posiciones a la izquierda, para exponente -2, dos posiciones a las derecha, etc.

¿Cómo representar números binarios en punto flotante empleando los registros de un computador? Una solución inmediata es dividir el registro en tres partes. Una para el signo, otra para la mantisa y otra para el exponente. Supongamos, como en el caso de la representación en punto fijo, que contamos con registros de 16 bits. Podemos dividir el registro en tres zonas, la primera, de un solo bit la empleamos para guardar el signo. La segunda de, por ejemplo 11 bits, la empleamos para guardar la mantisa del número y por último los cuatro bits restantes los empleamos para guardar el exponente en binario. Podríamos entonces representar el número -1.10100101×2^3 como,

The floating-point follows exactly the same representation that we just described for the scientific notation. The only difference is that we now use powers of two, which are the proper powers of binary representation, instead of powers of ten.
So, we would represent the binary number -1101.00101 as -1.10100101×2^3, and the binary number 0.001011101 as 1.011101×2^{-3}.

The term floating-point means that the mantissa decimal point does not split the integer part of the number from its decimal part. We need to use the exponent value to split the integer and decimal part of the number. So, for an exponent equal to zero, the mantissa decimal point is in the correct position to represent the number. For an exponent equal to 1, we must move the decimal point one position towards the left. For an exponent equal to -1, one position towards the right. For exponent 2, two positions towards the left. For exponent -2, two positions towards the right. And so on.

How can we represent floating points binary numbers using a computer register? A straightforward solution is to divide the register into three parts, One for the sign, another to store the mantissa and the third part for the exponent. Suppose that, as in the case of the fixed point representation, we have got 16-bit registers. We can divide the register in three zones. The first one, with a single bit to save the number sign. A second one of, let say 11 bits, to save the number mantissa and we reserve the remaining four bits to hold the exponent in binary representation. We could, then, represent the number -1.10100101×2^3 as,

sign.	mantissa											exponent.			
1	1	1	0	1	0	0	1	0	1	0	0	0	0	1	1
-	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^3	2^2	2^1	2^0

Podríamos entonces representar el número -1.10100101×2^3 como,

Si bien la representación que acabamos de ver sirve para el número representado, es evi-

Although the representation just described suits well the represented number, it is clear

dente que no podemos representar así un número con exponente negativo como por ejemplo 1.011101×2^{-3}. Además la división que hemos hecho del registro de 16 bits es arbitraria; podríamos haber elegido un reparto distinto de bits entre mantisa y exponente. De hecho, hasta la década de los noventa del siglo XX, cada ordenador adoptaba su propia representación de números en punto flotante. La consecuencia era que los mismos programas, ejecutados en máquinas distintas daban diferentes soluciones.

Ya en 1976 John Palmer, de la compañía INTEL, promueve la creación de un estándar, es decir, de una representación en punto flotante que fuera adoptada por todos los fabricantes de computadoras. El primer borrador del estándar fue elaborado por William Kahan (Intel), Jerome Coonen (Berkeley) y Harold Stone (Berkeley): documento K-C-S. Inspirado en el trabajo de Kahan en el IBM 7094 (1968). El proceso de estandarización fue bastante lento. La primera versión no fue operativa hasta 1985. El resultado de estos trabajos fue el estándar actual. Conocido como IEEE 754. IEEE son la siglas de Institute of Electrical and Electronics Engineers. Se trata de una asociación que nació en Estados Unidos en el siglo XIX y que goza actualmente de carácter internacional. El IEEE se ocupa, entre otras cosas, de establecer estándares industriales en los campos de la electricidad, la electrónica y la computación. Veamos en detalle el estándar IEEE 754.

El estándar IEEE 754. En primer lugar, el estándar se estableció pensando en dos tamaños de registro el primero de ellos emplea 32 bits para guardar los números en punto flotante y recibe el nombre de estándar de precisión simple. El segundo emplea 64 bits y se conoce como estándar de precisión doble.

Simple precisión. Empecemos por describir el estándar de precisión simple. En primer lugar, se reserva un bit para contener el signo. Si el bit vale 1, se trata de un número negativo, y si vale 0 de un número positivo. De los restantes 31 bits, 23 se emplean para

that we cannot represent with it a negative number like, for example, 1.011101×2^{-3}. Furthermore, we have arbitrarily divided the 16-bit register; we could have chosen a different allocation of bits between the mantissa and exponent. In fact, until the 1990s, each computer adopted its own representation for floating-point numbers. As a consequence, the same programs run on different computers yield different results.

already in 1974, John Palmer from INTEL company, promote the creation of a standard, that is, of a floating-point representation which might be adopted for all computers manufactures. The first draft of the standard, inspired in Kahan's work for the IBM 7094 computer (1968), was prepare by Willian Kahan (Intel), Jerome Coonen (Berkeley) and Harold Stone (Berkeley): K-C-S document. The standardisation process was slow. The first version of the standard was no ready till the year 1985. The result of these works was the current standard, known as the IEEE 754 standard. IEEE are the acronym of Institute of Electrical and Electronic Engineers. The Institute is an association born in the United States in the XIX century, and it has evolved to become an international. The IEEE is devoted, among other things, to stablish international industrial standards on the electricity, electronics and computer fields. Let's see in some detail the IEEE 754 standard.

the IEEE 745 standard. The standard was established thinking in register of two different size. The first one uses 32 bits to save floating-point numbers. It is known as the simple precision standard. The second one uses 64 bits and it is Known as the double precision standard.

Simple precision. We will start describing the simple precision standard. First we take a bit to save the sign. If the bits is one, then the number is negative. If the bits is zero the number is positive. We use 23 bits of the remaining 32, to represent the number mantissa and the other last 8 bits to represent the exponent.

representar la mantisa y 8 para representar el exponente.

Si analizamos como es la mantisa de un número binario en representación de punto flotante, nos damos cuenta de que la parte entera siempre debería valer 1. Por tanto, podemos guardar en la mantisa solo la parte del número que está a la derecha del punto decimal, y considerar que el 1 correspondiente a la parte entera está implícito. Por ejemplo, si tenemos el número binario 1.010111×2^3 solo guardaríamos la cifra 010111. El 1 de la parte entera sabemos que está ahí pero no lo representamos. A este tipo de mantisa la llamaremos normalizada, más adelante veremos por qué.

En cuanto al exponente, es preciso buscar una forma de representar exponentes positivos y negativos. El estándar establece para ello lo que se llama la representación en *exceso*. Con ocho bits podemos representar hasta $2^8 = 256$ números distintos. Éstos irían desde el $[00000000] = 0$ hasta el $[11111111] = 255$.

Si partimos nuestra colección de números en dos, podríamos emplear la primera mitad para representar números negativos, reservando el mayor de ellos para representar el cero, y la segunda para representar números positivos. Para obtener el valor de un número basta restar al contenido del exponente el número reservado para representar el cero. Se dice entonces que la representación se realiza *en exceso* a dicho número.

En el caso del estándar de simple precisión la primera mitad de los números disponibles iría del 0 al 127 y la segunda mitad del 128 al 255. La representación se realiza entonces en exceso a 127. La tabla 1.2 muestra unos cuanto ejemplos de cálculo de la representación *en exceso*

Tenemos por tanto que el exponente de 8 bits del estándar de precisión simple permite representar todos los exponentes enteros desde el -127 al 128.

Ya tenemos casi completa la descripción del éstandar. Solo faltan unos detalles –aunque muy importantes– relativos a la representación de números muy grandes y muy pequeños. Empecemos con los números muy grandes.

¿Cúal es el número más grande que podría-

If we examine a binary number's mantissa when using the floating-point representation, we realize that it must always take a value equal to one. Therefore, we only need to save in the chunk of the register reserved for the mantissa the decimal part, i.e., the digits located on the right of the decimal point, and consider the leading 1, belonging to the integer part of the number, implicit. For instance, if we have the binary number 1.010111×2^3 we only save the figure 010111. We Know the leading 1 of integer part is there, but we do not represent it. We will call this kind of mantissa a normalised mantissa, we shall see why later on.

We need a way to represent positive and negative exponents. The standard prescribes the use of representation known as biased exponent representation. Using 8 bits, we can represent up to $2^8 = 256$ different numbers. These numbers expand from $[00000000] = 0$ to $[11111111] = 255$.

We divide our numbers into two halves and take the lower half to represent negative numbers, using the largest number to represent the zero. Of course, we use the upper half to represent positive numbers. It is enough to subtract the number reserved to represent the zero to recover the *real* exponent of the number. So we can consider that our representation of the exponent is *biased* by the number used to represent the zero.

For the simple precision standard, the first half of available numbers ranges from 0 to 127 and the second half from 128 to 255. Then, the representation is biased by 127. Table 1.2 shows several examples on how to calculate this *biased* representation.

In conclusion: the 8 bits of the simple precision standard allows us to represent all integer exponents from -127 to 128.

We have almost ready the description of the IEEE standard. However, some essential details about representing very large and very small numbers remain. Let's start with the very large numbers. What is the most significant number we could represent using the simple precision standard? We could suppose

Bits de exponente Exponent's bits	equivalente decimal base-ten equivalent	Exceso a 127 127 biased Exponente representado represented exponent
00000000	0	$0 - 127 = -127$
00000001	1	$1 - 127 = -126$
00000010	2	$2 - 127 = -125$
⋮	⋮	⋮
01111110	126	$126 - 127 = -1$
01111111	127	$127 - 127 = 0$
10000000	128	$128 - 127 = 1$
⋮	⋮	⋮
11111110	254	$254 - 127 = 127$
11111111	255	$255 - 127 = 128$

Tabla 1.2: Representación *en exceso a* 127, para un exponente de 8 bits.
Table 1.2: 127-*biased* representation for a 8-bit exponent

mos representar en estándar de simple precisión? En principio cabría suponer que el correspondiente a escribir un 1 en todos los bits correspondientes a la mantisa (la mayor mantisa posible) y escribir también un 1 en todos los bits correspondientes al exponente (el mayor exponente posible). Sin embargo, el estándar está pensado para proteger las operaciones del computador de los errores de desbordamiento (ver sección 1.3 más adelante). Para ello reserva el valor más alto del exponente. Así cuando el exponente contenido en un registro es 128, dicho valor no se considera propiamente un número, de hecho es preciso mirar el contenido de la mantisa para saber qué es lo que representa el registro:

- Si los bits de la mantisa son todos cero, el registro contiene la representación del infinito ∞. El estándar específica el resultado de algunas operaciones en el infinito: $1/\infty = 0$, $\arctan(\infty) = \pi/2$.
- Si por el contrario, el contenido de la mantisa es distinto de cero, se considera el contenido del registro como no numérico (NaN, abreviatura en ingles de *Not a Number*). Este tipo de resultados, considerados no numéricos, surgen en algunas operaciones que carecen de sentido matémático: $0/0,\ \infty \times 0,\ \infty - \infty$.

that, in principle, the number with all its mantissa values set to one (the most significant possible mantissa) and all its exponent values set to one (the largest possible exponent). However, the standard was thought to protect the computer operations against overflow errors (see section 1.3 below). That protection is carried out by reserving the largest possible value of the exponent. Therefore, when a register holds an exponent equal to 128, this value is not considered a proper number. It is necessary to see the mantissa to know what the register does represent:

- If the mantissa bits are all zero, the register holds the representation of infinite ∞. The standard specifies the results of some operations on infinite: $1/\infty = 0, arctan(\infty) = \pi/2$.
- On the contrary, if the content of the mantissa is not zero, the content of the register is consider as non-numeric (NaN, acronym of Not a Number). Some computations, which lack of mathematical sense like $0/ =, \infty \times 0, \infty - \infty$, yields this NaN kind of results.

Therefore, the standard limits to 127 the maximum exponent which can be use to represent a finite number. We can build as an

Todo esto hace que el exponente mayor que realmente representa un número sea el 127. Como ejemplo podemos construir el número más grande que podemos representar distinto de infinito. Se trata del número con la mantisa más grande posible, una mantisa formada exclusivamente por unos, seguida por el exponente más grande posible. El exponente mas grande en exceso es 127, que corresponde a un exponente en binario $127 + 127 = 254 \equiv [11111110]$. Por tanto el número se representaría como,

sig.	←mantisa, 23 bits →	← exponente, 8 bits →
0	11111111111111111111111	11111110

En base 10 el número representado sería, $(\mathbf{1 \cdot 2^0} + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdots + 1 \cdot 2^{-23}) \times 2^{127} \approx 3.402823466385289 \times 10^{38}$. Donde hemos representado en negrita el 1 implícito, no presente en la mantisa.

Veamos ahora los números mas pequeños, de acuerdo con lo dicho, el número más próximo a cero que podríamos construir sería aquel que tuviera la mantisa más pequeña y el exponente más pequeño, es decir una mantisa formada solo por ceros y un exponente formado solo por ceros. Ese número sería $(1 \cdot 2^0) \cdot 2^{-127}$ (se debe recordar que la mantisa en la representación del estándar esta normalizada; lleva un 1 entero implícito).

Es evidente que si no fuera por el 1 implícito de la mantisa sería posible representar números aún más pequeños. Por otro lado, si consideramos siempre la mantisa como normalizada, es imposible representar el número 0. Para permitir la representación de números más pequeños, incluido el cero, los desarrolladores del estándar decidieron añadir la siguiente regla: *Si los bits del exponente de un número son todos ceros, es decir, si el exponente representado es -127, se considera que la mantisa de ese número* ***no*** *lleva un 1 entero implícito. Además el exponente del número se toma como -126.* Los números así representados reciben el nombre de números desnormalizados. Veamos algunos ejemplos.

example the largest number we can represent, different to infinity. This number should have the largest possible mantissa, i.e. a mantissa with all values set to one, and the largest possible exponent, 127, using the biasing representation of the simple precision standard. This exponent correspond to a binary exponent $127 + 127 = 254 \equiv [11111110]$. Thus, the largest number for simple precision representation would be,

This number in base-ten will take the form, $(\mathbf{1 \cdot 2^0} + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdots + 1 \cdot 2^{-23}) \times 2^{127} \approx 3.402823466385289 \times 10^{38}$. Where we have write in bold letter the implicit 1, which is not present in the matissa.

Let's turn now to the smallest numbers, according to the method we are using to represent the numbers, the number closest to zero we can build would be the number with the smallest mantissa and the smallest exponent. That is, a mantissa with all its bits set to zero and also an exponent with only zeros. This number will be, $(1 \cdot 2^0) \cdot 2^{-127}$ (remember that the standard uses a normalised mantissa, hence it has a implicit integer 1).

If the standard would not use a mantissa with an implicit one, we could represent smaller numbers. On the other hand, using a normalised mantissa, it is impossible to represent the number zero. The standard developers decided to add the following rule to allow representing small numbers, including the zero: *If the exponent bits of a number are all equal to zero, that is, if the exponent of the number is -127, the standard considers that the number's mantissa has* ***not*** *got an integer 1 implicit. In addition, the exponent of the number is taken as -126.* This special numbers are called denormalised numbers. Let's see some examples.

sig.	←mantisa, 23 bits →	← exponent., 8 bits →
0	10100000000000000000000	00000000

El exponente del número representado en la tabla anterior es 0. Por tanto, en exceso a 127 el exponente sería $0 - 127 = -127$. Este exponente corresponde a un número desnormalizado por tanto el número expresado en base 10 sería,

The exponent of the number represented in the table above is 0. Therefore, using a 127 biased representation the exponent is $0 - 127 = -127$. Then, this exponent belongs to a denormalised and so, the number represented in base-ten will be,

$$(1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdots) \times 2^{-126}$$

Nótese como el exponente ha sido reducido en una unidad (−126) a la hora de calcular la representación decimal del número.

El número más próximo a cero que podemos representar mediante números desnormalizados, será aquel que tenga la menor mantisa posible distinta de cero,

Notice how the exponent has been reduced one unit (-126) to calculate the decimal representation of the number.

The closest number to zero we can represent using denormalised number, will be that with the minimum non-zero mantissa,

ssig.	←mantissa, 23 bits →	← exponent., 8 bits →
0	00000000000000000001	00000000

que representado en base 10 sería el número: $(0 \cdot 2^{-1} + 0 \cdots + 1 \cdot 2^{-23}) \times 2^{-126} = 2^{-149} \approx 1.401298464324817 \times 10^{-45}$. Podemos calcular para comparar cual sería el número normalizado más próximo a cero, se trata del número que tiene una mantisa formada solo por ceros y el exponente más pequeño distinto de cero (un exponente igual a cero, implica automáticamente un número desnormalizado),

Which represented in base-ten will be the number: $(0 \cdot 2^{-1} + 0 \cdots + 1 \cdot 2^{-23}) \times 2^{-126} = 2^{-149} \approx 1.401298464324817 \times 10^{-45}$. We can compare this number with the closest number to zero we can represent using a normalised mantissa. This number has all zeros mantissa and the smallest possible non-zero exponent (Recall; an all-zero exponent automaticaly implies a denormalised number),

sig.	←mantissa, 23 bits →	← exponent., 8 bits →
0	00000000000000000000	00000001

Si lo representamos en formato decimal obtenemos: $(\mathbf{1 \cdot 2^0} + 0 \cdot 2^{-1} + 0 \cdots + 0 \cdot 2^{-23}) \times 2^{-126} = 2^{-126} \approx 1.175494350822288 \times 10^{-38}$

Como un último ejemplo, podemos calcular cual es el número desnormalizado más grande. Debería tener la mantisa más grande posible (todo unos) con un exponente formado exclusivamente por ceros,

If we now represent the number in denary system we obtain: $(\mathbf{1 \cdot 2^0} + 0 \cdot 2^{-1} + 0 \cdots + 0 \cdot 2^{-23}) \times 2^{-126} = 2^{-126} \approx 1.175494350822288 \times 10^{-38}$.

We will calculate, as a final example, the largest number we can represented using denormalised numbers. In this case, the number should have the largest possible mantissa (all ones) and a all-zeros exponent,

sig.	←mantissa, 23 bits →	← exponent., 8 bits →
0	11111111111111111111	00000000

En formato decimal, el número sería: $(1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdots + 1 \cdot 2^{-23}) \times 2^{-126} \approx 1.175494210692441 \times 10^{-38}$. Los dos últimos números representados, son muy próximos entre sí. De hecho, hemos calculado ya su diferencia, al obtener el número desnomarlizado más próximo a cero, $(\mathbf{1 \cdot 2^0} + 0 \cdot 2^{-1} + 0 \cdots + 0 \cdot$

In base-ten representation the number would be: $(1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdots + 1 \cdot 2^{-23}) \times 2^{-126} \approx 1.175494210692441 \times 10^{-38}$. These last two represented number are really near one to another. In fact, we have already calculate their difference, when we got the denormalised number closest to zero, $(\mathbf{1 \cdot 2^0} +$

$2^{-23})\times 2^{-126} - (1\cdot 2^{-1} + 1\cdot 2^{-2} + 1 \cdots + 1\cdot 2^{-23})\times 2^{-126} = (0\cdot 2^{-1} + 0 \cdots + 1\cdot 2^{-23}) \times 2^{-126} = 2^{-149} \approx 1.401298464324817 \times 10^{-45}$.

Doble precisión La diferencia entre los estándares de simple y doble precisión está en el tamaño de los registros empleados para representar los números. En doble precisión se emplea un registro de 64 bits. Es decir, el tamaño de los registros es el doble que el empleado en simple precisión. El resto del estándar se adapta al tamaño de la representación; se emplea 1 bit para el signo del número, 52 bits para la mantisa y 11 bits para el exponente.

En este caso el exponente puede almacenar $2^{11} = 2048$ números (desde el 0 al 2047). Como en el caso de estándar de simple precisión se dividen los números por la mitad, con lo que los exponentes se representan ahora en exceso a 1023. por tanto el exponente 0 representa el valor $0-1023 = -1023$ y el exponente 2047 representa el valor $2047 - 1023 = 1024$.

De nuevo, el valor mayor del exponente, 1024 se emplea para representar el infinito ∞, si va acompañado de una mantisa formada exclusivamente por ceros. En caso de que acompañe a una mantisa no nula, el contenido del registro se considera que no representa un número NaN (error de desbordamiento) . Podemos, como ejemplo, obtener para el estándar de doble precisión el número más grande representable,

$0\cdot 2^{-1} + 0 \cdots + 0\cdot 2^{-23}) \times 2^{-126} - (1\cdot 2^{-1} + 1\cdot 2^{-2} + 1 \cdots + 1\cdot 2^{-23})\times 2^{-126} = (0\cdot 2^{-1} + 0 \cdots + 1\cdot 2^{-23})\times 2^{-126} = 2^{-149} \approx 1.401298464324817 \times 10^{-45}$.

Double precision The difference between the single and the double precision standards is in the size of the registers we use to represent the numbers. When working with double precision, we use 64-bits size registers. Therefore, the size of the register is double than the size we use for simple precision. The standard is just fit to the larger size of the registers; we reserve a bit for the number's sign, 52 bits for the mantissa and 11 bits for the exponent.

In this case the exponent can store $2^{11} = 2048$ different numbers (from 0 to 2047). Very much alike the simple precision case, we split the exponent numbers into two halves, representing the values with a 1023 bias. Therefore, the an exponent equal to zero represent the value $0 - 1023 = -1023$ and the exponent 2047 represent the value, $2047 - 1023 = 1024$.

Again, we use the largest exponent value 1024 to represent the infinite ∞ with an all-zeros companying mantissa. Otherwise, if the companying mantissa has any non-zero bit, then the register does not represent a number NaN (overflow error). As an example, We can obtain the largest representable number using the double precision standard,

sig.	←mantissa, 52 bits →	← exponent., 11 bits →
0	1111111111111111111111111111111111111111111111111111	11111111110

El número representado toma en base 10 el valor: $(\mathbf{1\cdot 2^0} + 1\cdot 2^{-1} + 1\cdot 2^{-2} + 1 \cdots + 1\cdot 2^{-52}) \times 2^{1023} \approx 1.797693134862316 \times 10^{308}$

Por último, para exponente -1023 es decir, un exponente formado exclusivamente por ceros, el número representado se considera desnormalizado; se elimina el 1 entero implícito de la representación de la mantisa y se aumenta en una unidad el exponente, que pasa así a valer -1022. Como ejemplo, podemos calcular el número más próximo a cero representable en doble precisión,

The value in base-ten of the represented number is: $(\mathbf{1\cdot 2^0} + 1\cdot 2^{-1} + 1\cdot 2^{-2} + 1 \cdots + 1\cdot 2^{-52}) \times 2^{1023} \approx 1.797693134862316 \times 10^{308}$

Lastly, numbers with an exponent -1023, that is, numbers with an all-zero exponent, represent denormalised numbers. We remove the implicit one from the mantissa and increase the exponent on one unit, which takes the value -1022. For instance, we can compute the closest to zero number using the double precision representation,

sig.	←mantissa, 52 bits →	← exponent., 11 bits →
0	0000000000000000000000000000000000000000000000000001	00000000000

El número representado toma el base 10 el valor: $(0{\cdot}2^{-1}{+}0{\cdot}2^{-2}{+}0\cdots{+}1{\cdot}2^{-52}){\times}2^{-1022} \approx 4.940656458412465 \times 10^{-324}$

La tabla 1.3 resume y compara las características de los dos estándares vistos,

The number represented takes in base-ten the value: $(0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdots + 1 \cdot 2^{-52}) \times 2^{-1022} \approx 4.940656458412465 \times 10^{-324}$

Table 1.3 summarises and compares the characteristics of both standards we have seen,

Precisión simple. Registro de 32 bits. Exponente exceso a 127
Simple precision. 32-bits Register. Biased exponent a 127

Mantissa (23 bits)	exponent. (8 bits) (-exceso)(-bias)	número representado/represented number
0	$0-127=-127$	$(-1)^{bs}\cdot(0\cdot 2^{-1}+\cdots+0\cdot 2^{-23})\times 2^{0-127}=0$
$\neq 0$	$0-127\equiv\mathbf{-126}$	$(-1)^{bs}\cdot(m_1\cdot 2^{-1}+\cdots+m_{23}\cdot 2^{-23})\times 2^{0-127\equiv\mathbf{-126}}$
$\forall$	$1-127=-126$ hasta/until $254-127=127$	$(-1)^{bs}\cdot(\mathbf{1}+m_1\cdot 2^{-1}+\cdots+m_{23}\cdot 2^{-23})\times 2^{(e_8\cdot 2^8+\cdots+e_1\cdot 2^0)-127}$
0	$255-127=128$	$(-1)^{bs}\cdot(0\cdot 2^{-1}+\cdots+0\cdot 2^{-23})\times 2^{255-127}\equiv\infty$
$\neq 0$	$255-127=128$	$(-1)^{bs}\cdot(m_1\cdot 2^{-1}+\cdots+m_{23}\cdot 2^{-23})\times 2^{255-127}\equiv$ NaN

Precisión doble. Registro de 64 bits. Exponente exceso a 1023
Double precision. 64-bits register. Biased exponent 1023

Mantissa (52 bits)	exponent. (11 bits) (-exceso)(-bias)	número representado/represented number
0	$0-1023=-1023$	$(-1)^{bs}\cdot(0\cdot 2^{-1}+\cdots+0\cdot 2^{-52})\times 2^{0-1023}=0$
$\neq 0$	$0-1023\equiv\mathbf{-1022}$	$(-1)^{bs}\cdot(m_1\cdot 2^{-1}+\cdots+m_{23}\cdot 2^{-52})\times 2^{0-1023\equiv\mathbf{-1022}}$
$\forall$	$1-1023=-1022$ hasta $2046-1023=1023$	$(-1)^{bs}\cdot(\mathbf{1}+m_1\cdot 2^{-1}+\cdots+m_{52}\cdot 2^{-52})\times 2^{(e_{11}\cdot 2^{10}+\cdots+e_1\cdot 2^0)-1023}$
0	$2047-1023=1024$	$(-1)^{bs}\cdot(0\cdot 2^{-1}+\cdots+0\cdot 2^{-52})\times 2^{2047-1023}\equiv\infty$
$\neq 0$	$2047-1023=1024$	$(-1)^{bs}\cdot(m_1\cdot 2^{-1}+\cdots+m_{52}\cdot 2^{-52})\times 2^{2047-1023}\equiv$ NaN

Tabla 1.3: Comparación entre los estándares del IEEE para la representación en punto flotante. (bs bit de signo, m_i bit de mantisa, e_i bit de exponente)
Table 1.3: Comparison between the IEEE standards for floating point representation. (bs sign bit, m_i mantissa bit, e_i exponent bit)

Para terminar veamos algunos ejemplos de representación de números en el estándar IEEE 754:

1. ¿Cúal es el valor decimal del siguiente número expresado en el estándar del IEEE 754 de simple precisión precisión?

We will finish with some examples of number representation using the IEEE 754 standard:

1. Which is the base-ten representation of the following number represented in the simple precision IEEE 754 standard?

sig.	←mantissa, 23 bits →	← exponent., 8 bits →
1	11000000000000000000000	01111100

- El bit de signo es 1, por lo tanto el número es negativo.
- El exponente sería, $0\cdot 2^7+1\cdot 2^6+1\cdot 2^5+1\cdot 2^4+1\cdot 2^3+1\cdot 2^2+0\cdot 2^1+0\cdot 2^0-127=$

- The bit sign is 1 therefore, it is a negative number.
- The exponent is $0{\cdot}2^7{+}1{\cdot}2^6{+}1{\cdot}2^5{+}1{\cdot}2^4{+}1{\cdot}2^3{+}1{\cdot}2^2{+}0{\cdot}2^1{+}0{\cdot}2^0{-}127=124{-}127=-3$

$124 - 127 = -3$

- A la vista del exponente, la mantisa está normalizada, $\mathbf{1} + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} = 1.75$
- Por tanto el número en representación decimal es: $(-1)^1 \times (1.75) \times 2^{-3} = -0.21875$

2. ¿Cúal es el valor decimal del siguiente número expresado en el estándar del IEEE 754 de simple precisión?

sig.	←mantissa, 23 bits →	← exponent., 8 bits →
0	01000000000000000000000	10000001

- El bit de signo es 0, por lo tanto el número es positivo.
- El exponente sería, $1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot ^1 + 1 \cdot 2^0 - 127 = 129 - 127 = 2$
- A la vista del exponente, la mantisa está normalizada, $\mathbf{1} \cdot \mathbf{2^0} + 1 \cdot 2^{-2} = 0.25$
- Por tanto el número en representación decimal es: $(-1)^0 \times (0.25) \times 2^2 = +5.0$

3. ¿Cúal es la representación en el estándar IEEE 754 de simple precisión del número: 347.625?

- Convertimos el número a binario, $347.625 = 101011011.101$
- Representamos el número en formato de coma flotante, 1.01011011101×2^8
- mantisa: 01011011101 (normalizada)
- exponente: $8 \xrightarrow{exceso\ 127} 127 + 8 = 135 \xrightarrow{binario\ 8\ bits} 10000111$
- signo: 0 (positivo)

Con lo cual la representación de 347.625 es,

sig.	←mantissa, 23 bits →	← exponent., 8 bits →
0	01011011101000000000000	10000111

4. ¿cúal el la representación en el estándar IEEE 754 de simple precisión del número: $\frac{5}{3}$

- $\frac{5}{3} = 1.66666\cdots$
- Pasamos la parte entera a binario: $1 = 1 \cdot 2^0$
- Pasamos la parte decimal a binario:

- looking at the exponent we realise that the number has a normalised mantissa, $\mathbf{1} + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} = 1.75$
- Thus, the number in denary representation is: $(-1)^1 \times (1.75) \times 2^{-3} = -0,21875$

2. Which is the base-ten representation of the following number represented in the simple precision IEEE 754 standard?

sig.	←mantissa, 23 bits →	← exponent., 8 bits →
0	01000000000000000000000	10000001

- The sign bit is 0, therefore is a positive number.
- The exponent is, $1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot ^1 + 1 \cdot 2^0 - 127 = 129 - 127 = 2$
- looking at the exponent we see that it is a normalised number, $\mathbf{1} \cdot \mathbf{2^0} + 1 \cdot 2^{-2} = 0.25$
- Then, the denary representation of the number is: $(-1)^0 \times (0.25) \times 2^2 = +5.0$

3. Which is the simple precision standard IEEE 754 representation of the number 347.625?

- We change the number to binary representation, $347.625 = 101011011.101$
- We get the floating point representation of the number, 1.01011011101×2^8
- mantissa: 01011011101 (normalised)
- exponent: $8 \xrightarrow{bias\ 127} 127+8 = 135 \xrightarrow{binary\ 8\ bits} 10000111$
- sign: 0 (positive)

Then the representation of 347.625 is,

sig.	←mantissa, 23 bits →	← exponent., 8 bits →
0	01011011101000000000000	10000111

4. What is the standard IEEE 754 simple precision representation of the number?: $\frac{5}{3}$

- $\frac{5}{3} = 1.66666\cdots$
- We pass the integer part to binary: $1 = 1 \cdot 2^0$
- We pass the decimal part to binary:

$0.666666\cdots \times 2 = 1.333333\cdots$
$0.333333\cdots \times 2 = 0.666666\cdots$
A partir de aquí se repite el periodo $\widehat{10}$ indefinidamente: $1.666666 \xrightarrow{binario} 1.101010\cdots$

- Mantisa: el número en binario quedaría: $1.101010\cdots$, con la misma representación en punto flotante, $1.101010\cdots \times 2^0$. La mantisa será, 10101010101010101010

- Exponente:
$0 \xrightarrow{exceso\ 127} 127 + 0 = 127 \xrightarrow{binario\ 8\ bits}$ 01111111

- Signo: 0 (positivo)

Con lo cual la representación de $\frac{5}{3}$ es,

$0.666666\cdots \times 2 = 1.333333\cdots$
$0.333333\cdots \times 2 = 0.666666\cdots$
from this point on the period $\widehat{10}$ repeats indefinitely: $1.666666 \xrightarrow{binario} 1.101010\cdots$

- Mantissa: the number representation in binary is: $1.101010\cdots$, and the same floating-point representation, $1.101010\cdots \times 2^0$. The mantissa is, 10101010101010101010

- Exponent:
$0 \xrightarrow{exceso\ 127} 127 + 0 = 127 \xrightarrow{binario\ 8\ bits}$ 01111111

- Sign: 0 (positive)

therefore, $\frac{5}{3}$ binary representation is,

sig.	←mantisa, 23 bits →	← exponente, 8 bits →
0	10101010101010101010	01111111

1.3. Errores en la representación numérica.

Como se ha indicado anteriormente, cualquier representación numérica que empleemos con el ordenador, está sometida a errores derivados del tamaño finito de los registros empleados. Vamos a centrarnos en el estudio de los errores cometidos cuando representamos números empleando el formato de punto flotante del estándar del IEEE754.

En primer lugar, solo hay una cantidad finita de números que admiten una representación exacta, son los que se obtienen directamente de los valores –unos y ceros– contenidos en un registro al interpretarlos de acuerdo con las especificaciones del estándar. Estos números reciben el nombre de números máquina.

Un número real no será un número máquina si,

1. Una vez representado en formato de punto flotante su exponente esta fuera del rango admitido para los exponentes: es demasiado grande o demasiado pequeño.

2. Una vez representado en formato de punto flotante su mantisa contiene más dígitos de los bits que puede almacenar la mantisas del estándar.

1.3. Numerical representations Errors.

As we have already stated before, whatever numerical representation we use with a computer is subject to errors due to the computer register's finite size. We are going to focus on the mistakes we make when using the IEEE754 standard for floating point representation to represent real numbers.

First, there is only a finite set of numbers with an exact representation. These number are those directly obtained from the values —ones and zeros— contained in the register and interpreted according to the standard specifications. These numbers are called machine numbers.

A real number will not be a machine number if,

1. Once we have represented it in floating point format, the number exponent is out of the admitted range for exponents: it is too large or too small.

2. Once we have represented it in floating point format, the number mantissa has got more digits than bits can store the IEEE 754 standard mantissa.

Si el exponente se sale del rango admitido, se produce un error de desbordamiento. Si se trata de un valor demasiado grande, el error de desbordamiento se produce por exceso (*overflow*). El ordenador asignará al número el valor $\pm\infty$. Si el exponente es demasiado pequeño , entonces el desbordamiento se produce por defecto (*underflow*) y el ordenador asignará al número el valor cero.

Si el tamaño de la mantisa del número excede el de la mantisas representables, la mantisa se trunca al tamaño adecuado para que sea representable. Es decir, se sustituye el número por un número máquina cercano, este proceso se conoce como redondeo y el error cometido en la representación como error de redondeo.

1.3.1. Error de redondeo unitario

Supongamos que tenemos un número no máquina $x = (1.a_1a_2\cdots a_{23}a_{24}\cdots)\times 2^{\exp}$.

Aproximación por truncamiento. Si queremos representarlo empleando el estándar de simple precisión, solo podremos representar 23 bits de los que componen su mantisa. Una solución es truncar el número, eliminando directamente todos los bits de la mantisa más allá del 23 $x \approx x_T = (1.a_1a_2\cdots a_{23})\times 2^{\exp}$. Como hemos eliminado algunos bits, el número máquina x_T, por el que hemos aproximado nuestro número, es menor que él.

Aproximación por exceso. Otra opción sería aproximarlo empleando el número máquina inmediatamente superior. Esto sería equivalente a eliminar todos los bits de la mantisa más allá del 23 y sumar un bit en la posición 23 de la mantisa $x \approx x_E = (1.a_1a_2\cdots a_{23} + 2^{-23})\times 2^{\exp}$. En este caso, estamos aproximando por un número máquina mayor que el número aproximado.

Redondeo Es evidente que cualquier número real que admita una representación aproximada en el formato del estándar estará comprendido entre dos números máquina: $x_T \leq x \leq x_E$.

If the exponent is beyond the admissible range, it lead to overflow or underflow. Overflow happens when the exponent is too large, while underflow occurs when the exponent is too small. In the first case (overflow) the computer will assign the number the value $pm\infty$. In the second case, (underflow) the computer will assign the number the value zero.

If a number mantissa size exceeds the size of a representable mantissa, it is cut off to a suitable size to make it representable. In other words, we replace the number with a nearby machine number. this process is know as rounding-off and the error we make as rounding error.

1.3.1. round-off error

Suppose we have a non-machine number $x = (1.a_1a_2\cdots a_{23}a_{24}\cdots)\times 2^{\exp}$.

Approximation by Truncation. If we want to represent the number using the simple precision standard, we can only represent the first 23 bit of its mantissa. A possible solution is to truncate the number, eliminating all mantissa bits beyond the 23rd, $x \approx x_E = (1.a_1a_2\cdots a_{23}+ 2^{-23})\times 2^{\exp}$. Because we have eliminated some bits, the machine number x_T we have taken to approximate our number is less than it.

Approximation by Excess. Another option is to approximate the number using the next upper machine number. This is the same that eliminate all mantissa bits beyond the 23th and then, and add bit to the 23th mantissa position value, $x \approx x_E = (1.a_1a_2\cdots a_{23} + 2^{-23})\times 2^{\exp}$. Notice that now, we are using a machine number greater than the number we are approximating.

Rounding. Whatever number admits an approximate representation using the standard format will be comprised of two machine numbers: $x_T \leq x \leq x_E$.

We will follow, as a general criterion, to approximate each real number, either by truncation or excess, always using the nearest number. Whenever we round a number, we are get-

En general, el criterio que se sigue es aproximar cada número real, por truncamiento o exceso, empleando en cada caso el número máquina más cercano. Siempre que redondeamos un número cometemos un error, que podemos definir como el valor absoluto de la diferencia entre el valor real del número y su aproximación. Este error recibe el nombre de error absoluto,

ting an error that we can define as the absolute value of the difference between the actual valor of the number and the machine number chosen to represent it. This error is called absolute error.

$$\text{Error absoluto/absolute error} = |x - x_r|$$

Donde $x_r = x_T$ si se redondeó por truncamiento o $x_r = x_E$ si se redondeó por exceso.

El intervalo entre dos números máquina consecutivos puede obtenerse restando el menor del mayor,

Where $x_r = x_T$, if we rounded the number by truncation or $x_r = x_E$ if we rounded the number by excess.

We can obtain the interval between two consecutive machine numbers by subtracting the lower from the larger.

$$x_E - x_T = (1.a_1a_2\cdots a_{23} + 2^{-23}) \times 2^{\exp} - (1.a_1a_2\cdots a_{23}) \times 2^{\exp} = 2^{-23} \times 2^{\exp}$$

Si aproximamos ahora cualquier número real comprendido en el intervalo x_T y x_E por el más cercano de estos dos, el error absoluto que cometemos, sera siempre menor o como mucho igual que la mitad del intervalo,

If we approximate any real number inside the interval (x_T, x_E) by the nearest of this last two, the absolute error we get is always less or as much equal to half the interval,

$$|x - x_r| \leq \frac{1}{2}|x_E - x_T| = \frac{1}{2} \cdot 2^{-23} \cdot 2^{\exp}$$

Este resultado se ilustra gráficamente en la figura 1.1

This result is graphically illustrated in figure 1.1

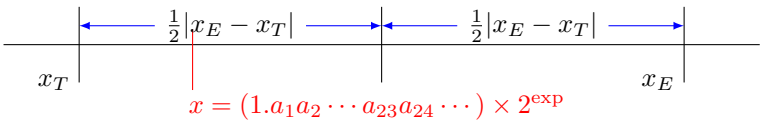


Figura 1.1: Posición relativa de un número no máquina x y su redondeo a número máquina por truncamiento x_T y por exceso x_E. Si redondeamos al más próximo de los dos, el error es siempre menor o igual a la mitad del intervalo $x_E - x_T$.
Figure 1.1: Location of a non-machine number in relation to its rounding to a machine number through truncation (x_T) and excess (x_E). When we round the number to the nearest of the two, the error is always less than half the interval $x_E - x_T$.

Examinando con un poco de detalle el resultado anterior, vemos que consta de tres términos. El término $\frac{1}{2}$ surge de aproximar un número real por su número máquina más cercano, El termino $2^{\exp}$ depende del tamaño del número. Para números grandes este factor será grande y para números pequeños será un factor

If we look at the previous result in more detail, we can see that is the product of three factors: the term $\frac{1}{2}$ comes from approximating a real number by its nearest machine number. The term 2^{exp} depends on the size of the number. For large numbers, this factor will be large, for small numbers, small. The

pequeño. Por último queda el factor 2^{-23}; este factor está directamente relacionado con la mantisa empleada en la representación. Efectivamente, si hubiéramos representado el número en el estándar de doble precisión, es fácil demostrar que el error absoluto cometido habría quedado acotado como,

last term is a fixed factor 2^{-23} and it is directly related with the mantissa size used in the representation. Indeed, if we had used the double precision standard, the absolute error we would have gotten would be limited as,

$$|x - x_r| \leq \frac{1}{2}|x_E - x_T| = \frac{1}{2} \cdot 2^{-52} \cdot 2^{\exp}$$

Es decir, el único factor que cambia en el error es precisamente el término relacionado con el tamaño de la mantisa. Este término recibe el nombre de precision del computador o *epsilon del computador* (eps). Y vale siempre 2 elevado a menos $(-)$ el número de bits de la mantisa. Por tanto, podemos generalizar la expresión parar la cota del error absoluto como,

That is, the only factor which changes is precisely the term related with the mantise size. This term is known as the machine precision or the *machine epsilon* (eps) and it takes always the value 2 raises to minus (-) the mantissa number-of-bits. Therfore, we can generalise the upper limit of the absolute error as,

$$|x - x_r| \leq \frac{1}{2}|x_E - x_T| = \frac{1}{2} \cdot \ \text{eps} \cdot 2^{\exp}$$

El significado del *epsilon* del computador queda aún más claro si definimos el error relativo,

The machine *epsilon* meaning is still easy to understand if we define relative error,

$$\text{Error relativo/Relative error} = \frac{|x - x_r|}{|x|} \leq \frac{1}{2}\frac{|x_E - x_T|}{|x|} = \frac{1}{2} \cdot \frac{\text{eps} \cdot 2^{\exp}}{(1.a_1a_2 \cdots a_{23}a_{24} \cdots) \times 2^{\exp}} \leq \frac{1}{2}\text{eps}$$

El error relativo pondera el valor del error cometido con la magnitud del número representado. Un ejemplo sencillo ayudará a entender mejor su significado. Imaginemos que tuviéramos un sistema de representación que solo nos permitiera representar número enteros. Si queremos representar los números 1.5 y 1000000.5 su representación sería 1 y 1000000 en ambos casos hemos cometido un error absoluto de 0.5. Sin embargo si comparamos con los números representados, en el primer caso el error vale la mitad del número mientras que en el segundo no llega a una millonésima parte.

En el caso de la representación que estamos estudiando, el error relativo cometido para cualquier número representable es siempre más pequeño que la mitad del *epsilon* del computador,

The relative error weighs the value of the gotten error wit the magnitude of the represented number. A simple example may help to better understand it meaning. Consider a representation system that only allows as ti represent integer numbers. If we what to represent the numbers 1.5 and 1000000.5 their representations would be 1 and 1000000. In both cases we get an absolute error equal to 0.5. But if we compare this results with the represented numbers, on the first case the error is half the number represented while in the second case this error does not reach the millionth part of the number.

For the representations we are studying, the relative error we get for whatever representable number is always least than half the machine *epsilon*,

$$x_r = x \cdot (1 + \delta); |\delta| \leq \frac{1}{2} \cdot eps$$

Un último comentario sobre el *epsilon* del computador, entendido como precisión. La diferencia entre dos números máquina consecutivos está estrechamente relacionada con el *epsilon*. Si tenemos un número máquina y queremos incrementarlo en la cantidad más pequeña posible, dicha cantidad es precisamente el *epsilon*, multiplicado por 2 elevado al exponente del número. La razón de que esto sea así está relacionada con el modo en que se suman dos números en la representación en punto flotante. Supongamos que tenemos un número cualquiera representado en el estándar de precisión simple,

A last comment on the machine *epsilon* considering it from the machine precision point of view. The difference between two consecutive machine numbers is tightly related to the *epsilon*. If we have a machine number and we want to increase it by the least possible quantity, such quantity is precisely the *epsilon* times two rises to the number exponent. The reason for this is the way we add numbers when using the numbers floating point representation. Suppose we have a number whatsoever represented using the simple precision standard,

sig.	←mantissa, 23 bits →	← exponent., 8 bits →
0	11110000000000000000000	10000000

El número representado tiene de exponente $2^7 - 127 = 1$. Supongamos ahora que quisieramos sumar a este número la cantidad 2^{-22} su representación en el estándar emplearía una mantisa de ceros (recordar el 1 ímplicito) y un exponente $-22 + 127 = 105$. Sería por tanto,

The represented number has exponent $2^7 - 127 = 1$. Suppose now that we want to add to this number the quantity 2^{-22} Its representation in the standard should use an all-zeros mantissa (recall the implicit 1) and an exponent $-22 - 127 = 105$. So it should be,

sig.	←mantissa, 23 bits →	← exponent., 8 bits →
0	00000000000000000000000	01101001

Para sumar dos números en notación científica es imprescindible que los dos estén representados con el mismo exponente, para entonces poder sumar directamente las mantisas. Disminuir el exponente del mayor de ellos hasta que coincida con el del menor no es posible, ya que eso supondría añadir dígitos a la parte entera de la mantisa, pero no hay bits disponibles para ello entre los asignados a la mantisa. Por tanto, la solución es aumentar el exponente del menor de los números, hasta que alcance el valor del exponente del mayor, y disminuir el valor de la mantisa desnormalizándola, es decir sin considerar el 1 implícito. Por tanto en nuestro ejemplo, debemos representar 2^{-22} empleando un exponente 1, $2^{-22} \to 2^{-23} \cdot 2^1$,

To add two numbers using scientific notation, we have to represent both numbers using the same exponent and then directly add their mantissa. We cannot decrease the exponent of the larger number to reach the exponent of the lower one because this means adding numbers to the mantissa integer part. However, this is impossible because there are no more bits available to enlarge the mantissa integer part. Therefore, the only solution available is to increase the exponent of the lower number till it reaches the value of the larger number exponent and to diminish the value of the mantissa, de-normalising it; that is, no longer considering the implicit 1. So, in our example, we must represent 2^{-22} using a exponent equal to 1, $2^{-22} \to 2^{-23} \cdot 2^1$,

sig.	←mantisa **(desnorm.)**, 23 bits →	← exponente, 8 bits →
0	00000000000000000000001	100000000

La suma de ambos números se obtiene sumando directamente las mantisas,

We can now get the sum the numbers, just adding their mantissas.

sig.	←mantisa, 23 bits →	← exponente, 8 bits →
0	11110000000000000000001	10000000

¿Qué pasa si tratamos de sumar un número más pequeño,por ejemplo 2^{-23}? Al repre-

What happen if we sum a smaller number, for example 2^{-23}? When we try to represent

sentarlo con exponente 1 para poder sumarlo el número tomaría la forma, $2^{-23} \rightarrow 2^{-24} \cdot 2^1$. Es fácil ver el problema, con una mantisa de 23 bits nos se puede representar el número 2^{-24} porque ya no hay *hueco* para él. La mantisa sería cero y –dado que se trata de una representación desnormalizada–, el número resultante sería cero. Por tanto, al sumarlo con el número inicial nos daría este mismo número.

Esto nos lleva a que la precisión del computador no es igual para todos los números representables, sino que depende de su magnitud. La precisión, tomada en función de la distancia entre dos números consecutivos, es: precisión = eps$\cdot 2^{exp}$ y su valor se duplica cada vez que aumentamos el exponente en una unidad. La figura 1.2 muestra esquemáticamente este fenómeno.

the number with exponent 1, the number will takes the form $2^{-23} \rightarrow 2^{-24} \cdot 2^1$. It is easy to see the problem, with a 23-bits mantissa it is not possible to represent the number 2^{-24} because there is not *place* for it. the mantissa would be equal to zero and, due to we are using a denormalised representation, the resulting number is zero. Therefore, when wi try to add it to the other number we obtain this same number again.

This lead us to realise that the precision is not equal for every representable number but it depends on the number size. If we associate the precision with the distance between to consecutive machine numbers we may write: precision = eps $\cdot\, 2^{exp}$ and its values duplicates each time we increase the exponent in one unit. Figure 1.2 depicts this phenomenon.

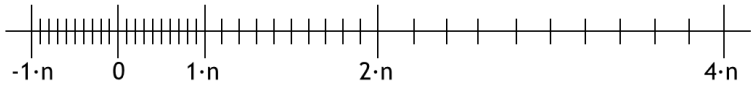


Figura 1.2: Ilustración del cambio de precisión con la magnitud de los números representados.
Figure 1.2: Graphic illustration of precision change with the magnitude of represented numbers

1.3.2. Errores de desbordamiento

En el apartado anterior hemos visto una de las limitaciones de la representación numérica de los ordenadores; el hecho de usar una mantisa finita hace que solo unos pocos números tengan representación exacta, el resto hay que aproximarlos cometiendo errores de redondeo. En este apartado nos centraremos en estudiar las limitaciones debidas al hecho de usar un exponente finito; solo podemos representar un rango limitado de valores.

La figura 1.3 muestra esquemáticamente el rango de números representables. El número negativo más pequeño representable, viene definido, por un bit de signo 1, para indicar que se trata de un número negativo y la mantisa y el exponente más grandes que, dentro de las especificaciones del estándar, todavía representan un número finito. Cualquier número negativo menor que éste, produce un error de desbordamiento conocido en la literatura

1.3.2. Underflow and Overflow errors

In previous sections we have seen one of the computer limitations to represent numbers; a finite mantissa makes that only a few numbers has an exact representation. We need to approximate the remaining numbers, getting in the process round-off errors. In this section, we focus on the limitations derived from the use of finite exponents, which the range of number we can represent using a computer.

Figure 1.3 shows a schematic view of the representable numbers range. The lowest negative representable number is define using a sign bit equal to 1, to indicate that it is a negative number, an the largest mantissa and exponent which, according to the standard specification, still represent a finite number. Any lowest negative number will produce an error technically known as negative overflow error. The negative number closest to zero that we can represent will be that which has a sign bit 1, the (denormalised) smallest possible man-

técnica con el nombre de *overflow* negativo. El número negativo más próximo a cero, que se puede representar será aquel que tenga bit de signo 1, la mantisa (desnormalizada) más pequeña posible y el exponente más pequeño posible. Cualquier número más próximo a cero que este, será representado por el ordenador como cero. Se ha producido en este caso un error de desbordamiento conocido como *underflow* negativo.

De modo análogo a como hemos definido el número negativo más pequeño representable, podemos definir el número positivo más grande representable. La única diferencia será que en este caso el bit de signo toma valor cero para indicar que es un número positivo. Cualquier número mayor que éste que queramos representar, provocará un error de desbordamiento (*overflow* positivo.) Por último, el número positivo más próximo a cero representable coincide con el correspondiente negativo, de nuevo salvo en el bit de signo, que ahora deberá ser cero. En el caso de tratar de representar un número más pequeño el ordenador no lo distinguirá de cero, produciéndose un desbordamiento denominado *underflow* positivo.

tissa and the lowest possible exponent. Whatever number closer than it to zero will be represented as zero by the computer. In this case we get an error known as negative underflow error.

Similar to how we have defined the lower negative number we can represent, we can also define the largest positive representable number. The only difference with the lower number case will be the sign bit, which now is equal to zero, to indicate that the represented number is positive. If we try to represent a number greater than largest representable, we will get a positive overflow error. Lastly, the positive number closest to zero we can represent is equal to the negative one except for the sign bit, which must now be equal to zero. If we try to represent a smaller number than the smallest representable, the computer cannot tell one from the other, producing a positive underflow and taking zero as the representation of the number.

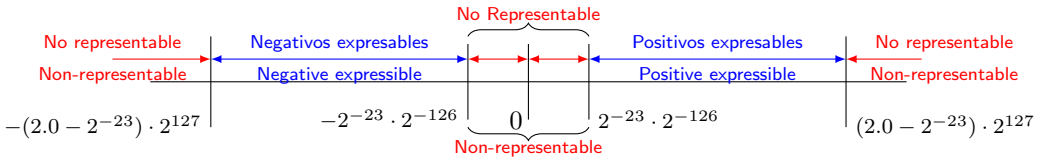


Figura 1.3: Números representables y desbordamientos en el estándar IEEE 754 de precisión simple.
Figure 1.3: Representable numbers and over/under-flows in the IEEE 754 simple precision standard.

1.4. Errores derivados de las operaciones aritméticas

Hasta ahora, nos hemos centrado siempre en el problema de la representación numérica del computador. En esta sección vamos a introducir un nuevo tipo de errores que tienen si cabe todavía más importancia desde el punto de vista del cálculo científico. Se trata de los

1.4. Errors derived from arithmetic operation

So far, we have focused on the numerical representation problem. In this section we will introduce a new kind of errors we are still more critical from the point of view of scientific computing; these are errors derived from the arithmetic operations.

errores derivados de las operaciones aritméticas.

Indirectamente ya se introdujo el problema en la sección anterior al hablar de la precisión del computador y de la necesidad de igualar los exponentes de los sumandos al mayor de ellos antes de poder sumar en formato de punto flotante. Veamos en más detalle algunas consecuencias de la aritmética en punto flotante.

1.4.1. Acumulación de errores de redondeo

Para empezar, se ilustrará el proceso de la suma de dos números representados en base 10 en formato de punto flotante. Supongamos que queremos sumar los números 99.99 y 0.161. Supongamos además que seguimos una representación en punto flotante con las siguientes limitaciones la mantisa es de cuatro dígitos, el exponente es de dos dígitos.

Si sumamos los números, tal y como los hemos escrito más arriba, sin seguir formato de punto flotante, el resultado sería,

We partially introduced the problem in the previous section when we spoke about computer precision and how we need to match the exponents of the addends, making them equal to the largest one, before performing the addition in floating-point representation. Let see in more detail some consequences of performing floating point arithmetic operations.

1.4.1. Accumulation of round-off errors.

We start looking at the addition process of two base-ten numbers represented on floating-point format. Suppose we want to add number 99.99 and 0.161. Besides, suppose we use the floating-point representation with the following two limitation: the mantissa is four digit size and the exponent has only two digits.

If we add the number as they are written above, without use the floating-point format the result would be,

$$99.99 + 0.161 = 100.151$$

Supongamos que los representamos ahora en el formato de punto flotante descrito más arriba, $99.99 = 9.999 \times 10^1$, $0.161 = 1.610 \times 10^{-1}$. Vamos a descomponer el proceso de sumar estos dos números en cuatro pasos, que reflejan, esquemáticamente, el proceso que seguiría un computador.

1. **Alineamiento.** Consiste en representar el número más pequeño empleando el exponente del mayor. Para ellos se desplazan hacia la derecha los dígitos del número más pequeño tantas posiciones como indique la diferencia de los exponentes de los dos números y se cambiar el exponente del número mas pequeño por el del más grande.

Suppose now that we represent the number following the floating-point format previously described, $99.99 = 9,999 \times 10^1$, $0.161 = 1.610 \times 10^{-1}$. We will divided the addition process of this two numbers in four steps which, schematically, shows the procedure the computer will carry out.

1. **Alignment.** This first step involves representing the smaller number using the exponent of the larger one: We shift the smaller number digits rightwards so many positions as the difference between the exponents of the numbers, and we equal the exponent of the smaller number to the larger one.

$$1.610 \times 10^{-1} \rightarrow 0.016 \times 10^1$$

Como cabía esperar, el desplazamiento hacia la derecha de la mantisa produce la pérdida de los últimos dígitos del número. Tenemos aquí un primer error de redondeo en el proceso

As it can be expected, the rightwards shift of the mantissa digits causes the lost of the last digits of the number. So we find here a first rounding error in the alignment procedu-

de alineamiento.

2. **Operación.** Una vez que los operandos están alineados, se puede realizar la operación. Si la operación es suma y los signos son iguales o si la operación es resta y los signos son diferentes, se suman las mantisa. En otro caso se restan.

Es importante comprobar tras realizar la operación si se ha producido desbordamiento de la mantisa; en el ejemplo propuesto:

re.

2. **Operation.** Once the addends have been aligned, we can carry out the operation. Whether the operation is an addition and the number signs are equal or whether the operation is a subtraction and the signs are different, it is enough to add the mantissas. Otherwise, we subtract them.

We must check after perform the computation if the mantissa has overflowed. In our example,

$$\begin{array}{rr} & 9.999 \cdot 10^1 \\ + & 0.016 \cdot 10^1 \\ \hline & 10.015 \cdot 10^1 \end{array}$$

No es posible emplear directamente el resultado de la suma de las mantisas de los sumandos como mantisa de la suma ya que requeriría emplear un dígito más. El resultado obtenido desborda el tamaño de la mantisa.

3. **Normalización.** Si se ha producido desbordamiento de la mantisa, es preciso volver a normalizarla,

We cannot use directly the result of the addends mantissas sum as the mantissa of the result because it need an extra digit, i.e., this addition yields a result which overflows the size of the mantissa.

3. **Normalisation.** If the mantissa has overflowed then, it is necessary to normalise it again,

$$10.015 \times 10^1 = 1.0015 \times 10^2$$

4. **Redondeo.** El desplazamiento de la mantisa hace que no quepan todo los dígitos, por lo que es necesario redondearla (por truncamiento o por exceso),

4. **Rounding.** The shifting of the digits makes that not all the digits can be accommodate inside the mantissa. Therefore, it is necessary to round it (by truncation or excess).

$$1.0015 \times 10^2 = 1.002 \times 10^2$$

Por último, se comprueba si las operaciones realizadas han producido desbordamiento del exponente, en cuyo caso el resultado sería un número no representable: ∞ ó 0.

5. **Renormalización.** Para números en representación binaria es preciso a veces volver a normalizar la mantisa después del redondeo. Supongamos, como en el ejemplo anterior, una mantisa de cuatro bits, —ahora con números representados en binario— y que tras realizar una operación suma el resultado que se obtiene es 11.111×2^1. La mantisa ha desbordado y hay que normalizarla $11.111 \times 2^1 \rightarrow 1.1111 \times 2^2$. Tenemos que redondear la mantisa y lo normal en este caso, dado que el número estaría exactamente en la mitad del intervalo entre dos números representables, es hacerlo

Eventually, we check if the operations we have carried out have produce an exponent overflow, because in this case the number would be not representable: ∞ or 0.

5. **Renormalisation.** For number in binary representation is sometimes necessary to normalise again the mantissa after carried out the rounding. Suppose as in the previous example that we have a four-bit size mantissa, but now we numbers in binary representation, and that after performing an addition operation it yield the result 11.111×2^1. The mantissa has overflowed and then we have to normalised it $11.111 \times 2^1 \rightarrow 1.1111 \times 2^2$. We have to round the mantissa and, due to the number is just on the interval half between two machine number, we round it by excess:

por exceso: $1.1111 \times 2^2 \rightarrow 10.000 \times 2^2$. La operación de redondeo por exceso ha vuelto a desbordar la mantisa y, por tanto, hay que volver a normalizarla: $10.000 \times 2^2 \rightarrow 1.000 \times 2^3$.

Analicemos el error cometido para la suma empleada en los ejemplos anteriores. En aritmética exacta, el número obtenido tras la suma es 100.152. Empleando la representación en punto flotante, con una mantisa de cuatro dígitos el resultado es $1.002 \times 10^2 = 100.2$. por tanto, el error absoluto cometido es,

$1.1111 \times 2^2 \rightarrow 10.000 \times 2^2$. The round-by-excess process has overflowed the mantissa one more time and, thus, we have to nomarlise it again $10.000 \times 2^2 \rightarrow 1.000 \times 2^3$.

If we analise the error we get after perform the addition in our previous example, In exact arithmetic, the sum of the numbers yields 100.152. When we use floating-point representation with a four-bit mantissa the result is $1.002 \times 10^2 = 100.2$. Then, the absolute error we get is,

$$|100.151 - 100.2| = 0.049$$

y el error relativo,

and the relative error,

$$\left|\frac{100.151 - 100.2}{100.152}\right| \approx 0.000489$$

En general puede comprobarse que para cualquier operación aritmética básica $\odot$ (suma, resta, multiplicación, división) y dos números máquina x, y se cumple,

In general, it is possible to demonstrate that for any basic arithmetic operation $\odot$ (addition, subtraction, multiplication, division) and two machine numbers x, y, it fulfils that,

$$\text{flotante}(x \odot y) = (x \odot y) \cdot (1 + \delta) \quad |\delta| \leq eps$$

Este enunciado se conoce como el axioma fundamental de la aritmética en punto flotante: *El eps del computador es la cota superior del error relativo en cualquier operación aritmética básica realizada en punto flotante entre números máquina.*

El axioma fundamental de la aritmética en punto flotante establece una cota superior para el error. En la práctica, es frecuente que se encadenen un gran número de operaciones aritméticas elementales. Al encadenar operaciones, los errores cometidos en cada una de ellas se acumulan.

Supongamos por ejemplo que queremos realizar la operación $x \cdot (y + z)$ en aritmética flotante. La operación podríamos describirla como,

This assertion is know as the fundamental axiom of floating-point arithmetic: *the machine eps is the upper bound of the relative error in any basic floating-point arithmetic operation performed between to machine numbers.*

The fundamental axiom of floating-point arithmetic sets an error upper bound. In practice, when computing, we link many basic arithmetical operations. This operation linking will lead to an accumulation of the errors we get in each basic operation.

for instance, suppose we want to carry out the operation, $x \cdot (y+z)$ in floating-point arithmetic. We may describe the operation as follows,

$$\begin{aligned} \text{flotante}\,(x \cdot (y + z)) &= (x \cdot \text{flotante}(y + z)) \cdot (1 + \delta_1) \\ &= (x \cdot (y + z)) \cdot (1 + \delta_2) \cdot (1 + \delta_1) \\ &\approx (x \cdot (y + z)) \cdot (1 + 2\delta) \end{aligned}$$

Donde δ_1 representa el error relativo cometido en el producto y δ_2 el error relativo come-

Where δ_1 represents the relative error after performing the product and δ_2 the error deri-

tido en la suma. Ambos errores están acotados por el *eps* del ordenador ($\delta_1,\ \delta_2 \leq eps$). El valor δ se puede obtener como,

ved from the addition. Both errors are bounded by the machine *eps*. ($\delta_1,\ \delta_2 \leq eps$). Notice that δ can be approximated as,

$$(1+\delta_1)\cdot(1+\delta_2) = 1+\delta_1\delta_2+\delta_1+\delta_2 \approx 1+2\delta,\ \delta = \text{máx}(\delta_1,\delta_2)$$

Podemos concluir que, en este caso, el error de redondeo relativo duplica al de una operación aritmética sencilla. En general, el error tenderá a multiplicarse con el número de operaciones aritméticas encadenadas.

Hay situaciones en las cuales los errores de redondeo que se producen durante una operación aritmética son considerables. Por ejemplo, cuando se suman cantidades grandes con cantidades pequeñas,

We may conclude that, in this case, the rounding error doubles the error of a simple arithmetical operation. An, in general, the error tends to multiply with the number of linked operations.

In some situations, the rounding errors derived from an arithmetical operation are notable. For example, when we add small quantities with large ones,

$$\text{flotante}(1.5\cdot 10^{38} + 1.0\cdot 10^{0}) = 1.5\cdot 10^{38} + 0$$

En este caso, durante el proceso de alineamiento, es preciso desplazar la mantisa del número pequeño 38 posiciones decimales para poder sumarlo. Con cualquier mantisa que tenga menos de 38 dígitos el resultado es equivalente a convertir el segundo sumando en cero.

Otro ejemplo es la pérdida de la propiedad asociativa,

In this case, on the alignment process, it is necessary to shift the small number mantissa 38 decimal places to be able to sum it to the large one. Using any mantissa with less than 38 digits is as much as converting the second addend into zero.

Another example is the lost of the associative property,

$$\left.\begin{aligned} x &= 1.5\cdot 10^{38} \\ y &= -1.5\cdot 10^{38} \end{aligned}\right\} \Rightarrow (x+y)+1 \neq x+(y+1) \begin{cases} (x+y)+1 = 1 \\ x+(y+1) = 0 \end{cases}$$

Los resultados pueden estar sometidos a errores muy grandes cuando la operación aritmética es la sustracción de cantidades muy parecidas. Si, por ejemplo, queremos realizar la operación $100.1 - 99.35 = 0.75$ y suponemos que estamos empleando una representación en punto flotante con una mantisa de cuatro dígitos y los números representados en base 10 ($100.1 = 1.001\cdot 10^2$, $99.35 = 9.935\cdot 10^1$)

1. **Alineamiento**

The results may be subject to very large errors when the arithmetical operation is the subtraction of two very similar quantities. Suppose we want to carry out the operation $100.1 - 99.35 = 0.75$ using a floating-point representation with 4-digits mantissa and the numbers represented in base-10. ($100.1 = 1.001\cdot 10^2$, $99.35 = 9.935\cdot 10^1$)

1. **Alignment**

$$9.935\cdot 10^1 \rightarrow 0.994\cdot 10^2$$

2. **Operación**

2. **Operation**

$$\begin{array}{rr} & 1.001\cdot 10^2 \\ - & 0.994\cdot 10^2 \\ \hline & 0.007\cdot 10^2 \end{array}$$

3. **Normalización**

3. **Normalisation**

$$0.007\cdot 10^2 \rightarrow 7.000\cdot 10^{-1}$$

Los pasos 4 y 5 no son necesarios en este ejemplo. Si calculamos ahora el error absoluto de redondeo cometido,

Steps 4 and 5 are not necessary in this example If we now calculate the absolute rounding error we get,

$$|0.75 - 0.7| = 0.05$$

Y el error relativo,

And the relative error,

$$\left|\frac{0.75 - 0.7}{0.75}\right| \approx 0.0666$$

Es decir, se comete un error de un $6,7\,\%$. El problema en este caso surge porque en el proceso de alineamiento perdemos un dígito significativo.

En un sistema de representación en punto flotante en que los números se representan en base β y el tamaño de la mantisa es p, Si las sustracciones se realizan empleando p dígitos, el error de redondeo relativo puede llegar a ser $\beta - 1$.

Veamos un ejemplo para números en base 10 ($\beta = 10$). Supongamos que empleamos una mantisa de cuatro dígitos y que queremos realizar la operación $1.000 \cdot 10^0 - 9.999 \cdot 10^{-1}$. El resultado exacto sería, $1 - 0.9999 = 0.0001$. Sin embargo, en el proceso de alineamiento,

Hence, we have approximately a $6.7\,\%$ error. In this case the problem arises during the alignment process, because we lost a significant digit.

In a floating-point representation system, where we represent the number in base-β and the mantissa size is p, if we perform differentiation using p digits, we can expect a relative rounding error that can reach a maximum value of $\beta - 1$.

Let's see an example for numbers written in base-10 ($\beta = 10$). Suppose we are using a four-digits mantissa and we want to carry out the operation $1.000 \cdot 10^0 - 9.999 \cdot 10^{-1}$. The exact result would be, $1 - 0.9999 = 0.0001$. Nevertheless, in the alignment process,

$$9.999 \cdot 10^{-1} \rightarrow 0.9999 \cdot 10^0$$

Si redondeamos el número por exceso, el resultado de la sustracción sería cero. Si lo redondeamos por defecto,

If we round the number by excess the result of the substraction will be zero. If we round the number by truncation,

$$\begin{array}{rr} & 1.000 \cdot 10^0 \\ - & 0.999 \cdot 10^0 \\ \hline & 0.001 \cdot 10^0 \end{array}$$

y el error relativo cometido sería,

and the relative error we get would be,

$$\left|\frac{1.000 \cdot 10^{-4} - 1.000 \cdot 10^{-3}}{1.000 \cdot 10^{-4}}\right| = \frac{10^{-4}(10-1)}{10^{-4}} = 10 - 1 \Rightarrow \beta - 1$$

Para paliar estos problemas, el estándar IEEE 754 establece que las operaciones aritméticas deben realizarse siempre empleando dos dígitos extra para guardar los resultados intermedios. Estos dos dígitos extra reciben el nombre de dígitos de protección o dígitos de guarda (*guard digits*). Si repetimos la sustracción, $100.1 - 99.35$, empleando los dos dígitos de guarda,

1. **Alineamiento**

To alleviate these problems, the IEEE 754 standard establishes that arithmetical operation should be performed always using two extra digits to save the intermediate results. These two extra bits are called protection digit or guard digits. If we repeat the subtraction, $100.1 - 99.35$, but using now the two guard digits,

1. **Alignment.**

$$9.93500 \cdot = 10^1 \rightarrow 0.99350 \cdot 10^2$$

2. **Operación**

$$\begin{array}{rr} & 1.00100 \cdot 10^2 \\ - & 0.99350 \cdot 10^2 \\ \hline & 0.00750 \cdot 10^2 \end{array}$$

3. **normalización**

$$0.00750 \cdot 10^2 \rightarrow 7.500 \cdot 10^{-1}$$

En este caso, obtenemos el resultado exacto. En general, empleando dos bits de guarda para realizar las sustracciones el error de redondeo es siempre menor que el *eps* del computador.

1.4.2. Anulación catastrófica

La anulación catastrófica se produce cuando en una operación aritmética, típicamente la sustracción, los dígitos más significativos de los operandos, que no están afectados por el redondeo, se cancelan entre sí. El resultado contendrá fundamentalmente dígitos que sí están afectados por errores de redondeo. Veamos un ejemplo.

Supongamos que queremos realizar la operación $b^2 - 4 \cdot a \cdot c$ empleando números en base 10, y una mantisa de 5 dígitos. Supongamos que los números empleados son: $b = 3.3357 \cdot 10^0$, $a = 1.2200 \cdot 10^0$, $c = 2.2800 \cdot 10^0$. El resultado exacto de la operación es,

In this case we obtain the exact result. In general, using two guards bit to perform subtractions, the rounding error is always less than the machine *epsilon*.

1.4.2. Catastrophic cancellation.

Catastrophic cancellation takes place when performing an arithmetical operation –typically the subtraction– the more significant digits that are not affected by the rounding cancel between them. The result will then contain digits that are indeed affected by the rounding. Let's see an example,

Suppose we can carry out the operation $b^2 - 4 \cdot a \cdot c$ using base-10 numbers and a five-digit mantissa. Suppose also that we employ the numbers: $b = 3.3357 \cdot 10^0$, $a = 1.2200 \cdot 10^0$, $c = 2.2800 \cdot 10^0$. The exact result of the operation is,

$$b^2 - 4 \cdot a \cdot c = 4.944 \cdot 10^{-4} \tag{1.1}$$

Si realizamos las operaciones en la representación pedida,

If we perform the operations in the requested representation,

$$\begin{aligned} b^2 &= 1.112689 \cdot 10^1 \\ 4 \cdot a \cdot c &= 1.112640 \cdot 10^1 \\ b^2 - 4 \cdot a \cdot c &= 5.0000 \cdot 10^{-4} \end{aligned}$$

El error relativo cometido es aproximadamente un 1 %. Como se han utilizado dos bits de guarda en las operaciones intermedias, la sustracción no comente en este caso error alguno. El error se debe a los redondeos de los resultados anteriores. Una vez realizada la sustracción, el resultado solo contiene los dígitos sometidos a redondeo ya que los anteriores se han anulado en la operación.

We get a relative error of around a 1 %. We used two guard bits in the intermediate operations, so the subtraction makes no mistake. The error is due to the rounding of previous results. Once the subtraction is performed, the result only holds digits affected by the previous rounding because the first (more significant) digits have been cancelled during the operation.

Como regla práctica se debe evitar al realizar operaciones aritméticas aquellas situaciones en las que se sustraen cantidades casi iguales. Veamos otro ejemplo, para ilustrar esta idea. Supongamos que queremos realizar la operación,

As a thumb rule, we have to avoid, when performing arithmetical operation, the subtraction of of almost equal quantities. Let's see another example, to clarify this idea. Suppose we whant to carry out the operation,

$$y = \sqrt{x^2 + 1} - 1$$

Para valores pequeños de x, esta operación implica una anulación con pérdida de dígitos significativos. Una posible solución es utilizar una forma alternativa de construir la ecuación. Si multiplicamos y dividimos por el conjugado,

For small values of x, this operation implies a cancellation with the loss of significant digits. A possible solution is to write the equation using an alternative form. If we multiply and divide by the conjugate,

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

1.4.3. Errores de desbordamiento

Al, hablar del rango finito de los números representables, se indicó cómo trata el éstandar IEEE 754 los números que desbordan los límites de representación. Al realizar operaciones aritméticas, es posibles llegar a situaciones en las que el resultado sea un número no representable bien por se demasiado grande en magnitud, (*Overflow* positivo o negativo) o por ser demasiado pequeño, (*underflow* positivo o negativo). Un ejemplo que nos permite ilustrar este fenómeno; es el del cálculo del módulo de un vector,

1.4.3. Overflow and underflow errors

When we spoke above the representable numbers finite rank, we showed how the IEEE 754 standard deals with the numbers that overflows the representation boundaries. When we perform arithmetic operations it is possible to arrive to situations where the operation result be a non-representable number because its magnitude is too large (positive or negative overflow) or because its magnitude is too small (positive or negative underflow). We will present an example that allows us to show this phenomenon; the computing of a vector module,

$$||\vec{v}|| = ||(v_1, v_2 \cdots v_n|| = \sqrt{\sum_{i=1}^{n} v_i^2}$$

El siguiente módulo de python contiene dos funciones para calcular la norma de un vector. Vamos a fijarnos en primer lugar en la funcion `norma_naive`.

The following Python module includes two functions to calculate a vector norm. Let's first examine the function `norma_naive`.

norm_naive.py

```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4 Created on Wed Jul 17 14:43:50 2024
5 This
```

```
@author: juan
"""

import numpy as np

def norma_naive(x):
    """
    This program (naive) calculates the norm of a vector
    Parameters
    ----------
    x : TYPE Real Vector
        DESCRIPTION. imput vector  whose norm we wamma get

    Returns
    -------
    m: Type Real number
       Description. The vector norm
    """


    m = 0.
    for e in x:
        m = m +e**2

    m =np.sqrt(m)
    return m

def norma_robust(x):
        """
       This program calculates the norm of a vector
       Parameters
       ----------
       x : TYPE Real Vector
           DESCRIPTION. imput vector  whose norm we wamma get

       Returns
       -------
       m: Type Real number
          Description. The vector norm
       """
       l =x.shape[0]
       if l==1:
           m = np.abs(x[0])
       else:
           mayor  = 0
           mscalado = -1
           for e in x:
               if e == 0:
                   continue
               modxi = np.abs(e)
               if mayor<modxi:
                   mscalado = 1 + mscalado*(mayor/modxi)**2
```

```
58                  mayor = modxi
59              else:
60                  mscalado = mscalado+(modxi/mayor)**2
61              m = mayor*np.sqrt(mscalado)
62      return m
63
```

`norm_naive` provocará un error de desbordamiento incluso para n=1, si introducimos un número cuyo cuadrado sea mayor que el mayor número representable. Si introducimos el número, $2^{1024/2} = 1.340780792994260e + 154$ en nuestro programa, Python devolverá como solución *inf*.

`norm_naive` will generate an overflow error even for n=1, provide we introduce a number whose square is greater that the largest representable number. If we introduce the number $2^{1024/2} = 1.340780792994260e + 154$, our function will return *inf* and we also get a warning from Python.

```
In [77]: x= np.array([2**(1024/2)])

In [78]: m = norma(x)
/home/juan/LCC_Python/codigos/aritmetica_del_computador/norm_naive.py:28:
RuntimeWarning: overflow encountered in scalar power
m = m +e**2

In [79]: m
Out[79]: inf
```

Es decir, el resultado produce un error de desbordamiento. El problema se produce en el bucle, al calcular el cuadrado del número,

So, the result we get is an overflow error. The problem takes place in the loop when it calculate the square of the number,

$$\left(2^{1024/2}\right)^2 = 2^{1024} > (2 - 2^{-52}) \cdot 2^{1023}$$

Una vez producido el desbordamiento el resto de las operaciones quedan invalidadas. Como en casos anteriores, la solución a este tipo de problemas exige modificar la forma en que se realizan los cálculos. Un primer paso sería igualar el módulo al valor absoluto del número cuando n=1. De este modo, el modulo del número propuesto en el ejemplo anterior se podría calcular correctamente.

Todavía es posible mejorar el programa, y ampliar el rango de vectores para los que es posible calcular el módulo. Para ello es suficiente recurrir a un pequeño artificio: dividir los elementos del vector por el elemento de mayor tamaño en valor absoluto, calcular el módulo del vector resultante, y multiplicar el resultado de nuevo por el elemento de mayor tamaño,

once the programs produces the overflow, the remaining operation are invalid. Like in previous cases, we can solve this kind of problems modifying the method we follow to compute the result. A first step would be to make the norm equal to the absolute value of the number in case n=1. This way we can correctly compute the norm of the number proposed in the previous example.

However, if we can widen the rank of the vector for which we want to compute the norm, more than this fix is needed. We can improve our code using a small trick: we divide the elements of the vector for the largest element in absolute value, then calculate the module of the vector so obtained, and eventually multiply again the result for the largest number.

$$\sqrt{\sum_{i=1}^{n} v_i^2} = |\operatorname{máx} v_i| \cdot \sqrt{\sum_{i=1}^{n} \left(\frac{v_i}{|\operatorname{máx} v_i|}\right)^2}$$

La segunda función incluida en el anterior script the python, **norm_robust**, calcula la norma de un vector empleando el procedimiento que acabamos de describir.

The second function included in the python script above, **norm_robust**, computes the norm of a vector using the procedure just described,

```
In [109]: x = np.array([0,0,0,0,0,2**(1024/2),2**(1024/2)])

In [110]: x
Out[110]:
array([0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
0.00000000e+000, 1.34078079e+154, 1.34078079e+154])

In [111]: m = norma_naive(x)
/home/juan/LCC_Python/codigos/aritmetica_del_computador/norm_naive.py:28:
RuntimeWarning: overflow encountered in scalar power
m = m +e**2

In [112]: m
Out[112]: inf

In [113]: m = norma_robust(x)

In [114]: m
Out[114]: 1.8961503816218355e+154
```

Si aplicamos este programa a obtener la norma del vector,

If we apply this program to obtaint the norm of the vector,

$$x = [2^{1024/2}\ 2^{1024/2}]$$

obtenemos como resultado,

We get the folloing result,

$$m = 1.8961503816218355 \cdot 10^{154}$$

En lugar de un error de desbordamiento (*inf*).

Instead an overflow error (*inf*)

Índice alfabético

Alphabetic Index