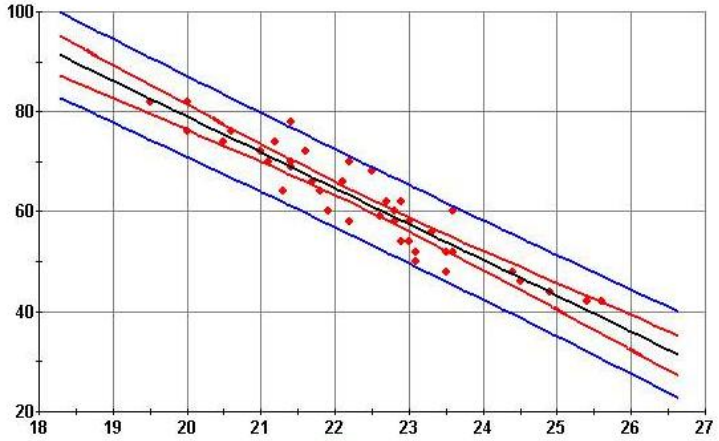
Aritmética del Computador

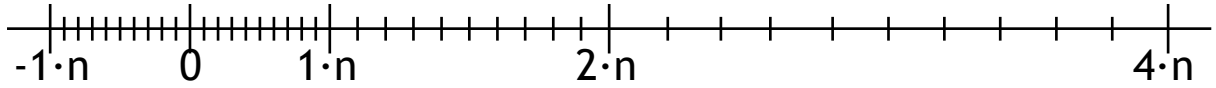
Prof. MauriZio Mattesini

Posibles fuentes de Error

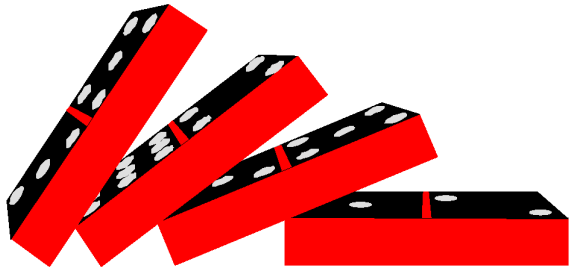
1. Datos de entrada
   - Experimental
   - Cálculos previos



2. Representación de los números :
   - Redondeo
   - Desbordamiento



3. Cálculos:
   - Acumulación de errores de redondeo
   - Anulación catastrófica
   - Desbordamiento



4. Algoritmo :
   - Discretización / Truncamiento
   - Estabilidad del algoritmo

$$\frac{dx}{dt} \approx \frac{x_n - x_{n-1}}{T}$$

Representación binaria y decimal

Los **números reales** pueden representarse convenientemente mediante una recta que se extiende hacia el infinito en ambas direcciones. Los **enteros** son los números 0, 1, -1, 2, -2, 3, -3,...

Los **números racionales** son aquéllos que consisten en el cociente de dos enteros, como 1/2, 2/3, 6/3; algunos de éstos, por ejemplo 6/3, son enteros. Las expansiones de números reales no enteros pueden ser **finitas** o **no terminar**. Por ejemplo, en el caso de **11/2** ambas expansiones terminan:

$11/2=(5.5)_{10}=5x10^{0}+5x10^{-1}$
$11/2=(101.1)_{2}=1x2^{2}+0x2^{1}+1x2^{0}+1x2^{-1}$

Sin embargo, el número **1/10**, el cual obviamente tiene la representación decimal finita $(0.1)_{10}$, no tiene una representación binaria finita, sino una expansión que no termina:

$1/10=(0.0001100110011...)_{2}=1x2^{-4}+1x2^{-5}+0x2^{-6}+0x2^{-7}+1x2^{-8}+1x2^{-9}+0x2^{-10}+...$

Nótese que esta representación, aunque no termina, es **repetitiva** o **periódica**. La fracción **1/3** tiene una expansión interminable tanto en binario como en decimal:

$1/3=(0.333...)_{10}=(0.010101...)_{2}$

Los **números racionales** siempre tienen expansiones **finitas** o **repetitivas**. Por ejemplo, **1/7**:

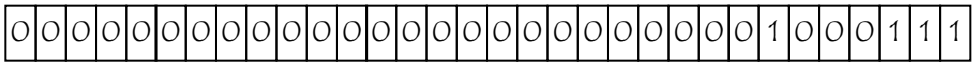
$1/7=(0.142857142857...)_{10}$

Los **números irracionales** son los reales que no son racionales. Ejemplos familiares son: $\Phi=(1.6180...)_{10}$, **sqrt(2)**$=(1.414213...)_{10}$, $\pi=(3.141592...)_{10}$, $e=(2.71828182845...)_{10}$. Los números irracionales siempre tienen expansiones **interminables** y **no repetitivas**.

Representación de números en la computadora

¿Cuál es la mejor manera de representar números en la computadora?

Comencemos por considerar los **números enteros positivos**. Típicamente, los enteros se almacenan en una **palabra de 32 bits** y si nos concernieran sólo enteros no negativos, la representación sería fácil. Por ejemplo, el **entero 71** se almacenaría como:



De esta manera se pueden representar los enteros no negativos desde **0** (una cadena de bits de 32 ceros) a $\mathbf{2^{32}-1}$ (una cadena de bits de 32 unos). El número 2^{32} es demasiado grande, dado que su representación binaria consiste en un uno seguido por 32 ceros.

Necesitamos ser capaces de representar **enteros negativos** (p.e., **-71**), además de los enteros positivos y 0. La idea más obvia es **signo-y-módulo**: usar uno de los 32 bits para representar el signo, y los restantes 31 bits para almacenar la magnitud del entero, que entonces podría variar entre **0** y $\mathbf{2^{31}-1}$.

Sin embargo, prácticamente todas las máquinas usan una representación más ingeniosa, llamada **complemento a 2** (opcional).

Representación de punto fijo (primeros computadores)

Para la mayoría de propósitos de cómputo numérico, los **números reales**, sean racionales o irracionales, se almacenan aproximadamente usando la representación binaria del número. Hay dos métodos posibles, llamados **punto fijo** y **punto flotante**.

En la representación de **punto fijo**, la palabra puede verse como dividida en tres campos:

1. Un campo de bit para el signo del número
2. Un campo de bits para la representación binaria del número anterior al punto binario
3. Un campo de bits para la representación binaria posterior al punto binario.

Por ejemplo, en una palabra de **32 bits** con **anchos de campo** de 15 y 16 bits respectivamente, el número $(11/2)_{10}=(101.1)_2$ se almacenaría como:

0	000000000000101	1000000000000000

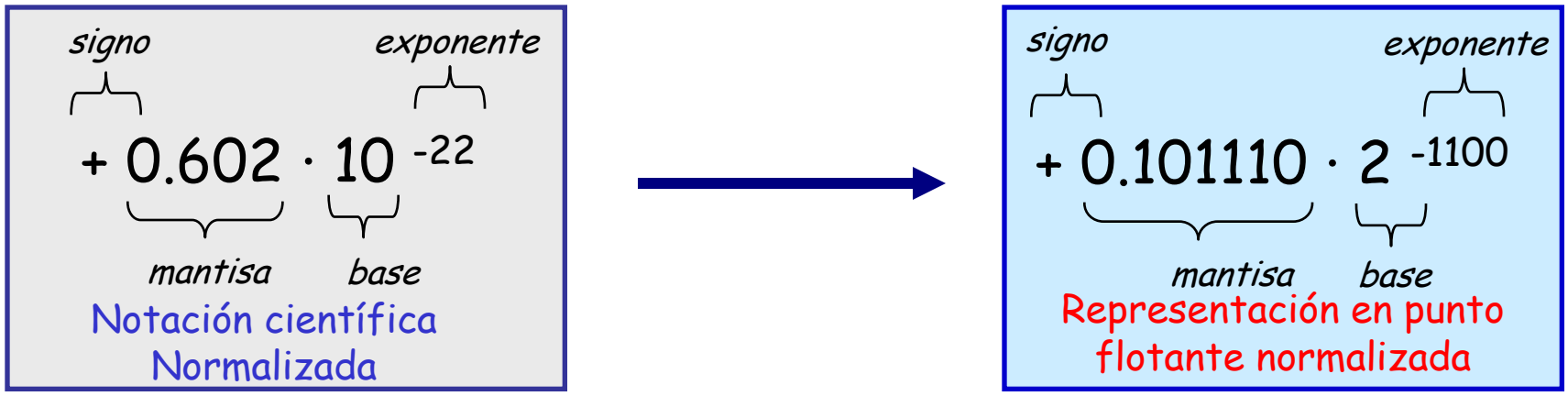
mientras el número $(1/10)_{10}=(0.0001100110011\ldots)_2$ se almacenaría aproximadamente como:

0	000000000000000	0001100110011001

El sistema de punto fijo está severamente limitado por el tamaño de los números que pueden almacenarse. En el ejemplo anterior, podrían almacenarse sólo números que varíen en tamaño desde 2^{-16} (1.5259e-05) hasta 2^{15} (32768). Por consiguiente, la representación de punto fijo es **raramente usada para computación científica!**

Representación en Punto Flotante (I)

Los computadores representan los números reales utilizando una notación semejante a la conocida notación científica normalizada:



En todo número en punto flotante se distinguen cuatro componentes:

- ★ **Signo:** indica el signo del número (**0**=positivo, **1**=negativo)
- ★ **Mantisa:** contiene la magnitud del número (en binario puro)
- ★ **Exponente:** contiene el valor de la potencia de la base (notación exceso)
- ★ **Base:** queda implícita y es común a todos los números (habitual base 2)

Otras alternativas menos extendidas para la representaciones de los números reales:

- ✓ Notación punto fijo (primeros computadores)
- ✓ Ciertas representaciones involucran el almacenamiento de los logaritmo de un número (las multiplicaciones se pueden implementar mediante sumas)
- ✓ Notación racional : se utilizan dos números enteros (a,b) para representar la fracción a/b

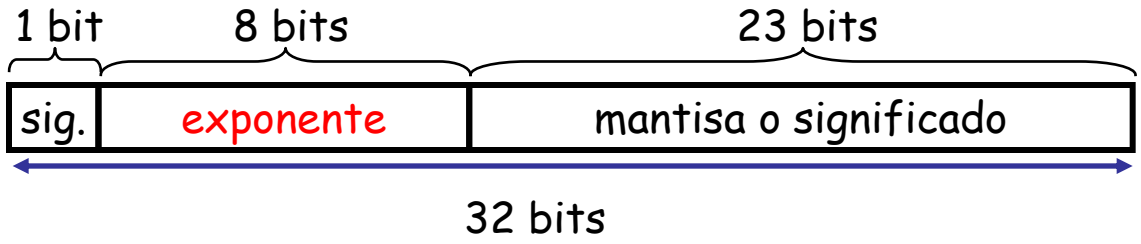
Representación en Punto Flotante (II)

Un mismo número puede tener **varias representaciones** ($0.110 \cdot 2^5 = 110 \cdot 2^2 = 0.0110 \cdot 2^6$)
Los número suelen estar normalizados. Un número está normalizado en base dos si tiene la forma:

0.1 xxxxxxxx	2^{xxx}
1. xxxxxxxx	2^{xxx-1}

→ Tienen siempre un 1 a la izquierda: se puede almacenar de forma implícita (**bit fantasma**).

Ejemplo: Formato de punto flotante de 32 bits



Si el número real decimal se puede representar con **exponente** y **mantisa** ocupando los bits asignados, entonces el número real tendrá una representación en punto flotante exacta denominándose **número de máquina**.

El rango de valores representable por cada uno de los campos es:

- **Exponente** (8 bits en exceso a la mitad del exp. max: 254/2=**127**) : indica el **rango** que el computador entiende como números reales.

  Exp_min: $(00000001)_2=(1)_{10}$ -> *emin*=$(\text{exponente-127})_{10}=(1-127)_{10}=\mathbf{(-126)_{10}}$

  Exp_max: $(11111110)_2=(254)_{10}$ -> *emax*=$(\text{exponente-127})_{10}=(254-127)_{10}=\mathbf{(+127)_{10}}$

  El rango de valores representables está entre -126 ($2^{-126}\approx\mathbf{10^{-38}}$) y +127 ($2^{127}\approx\mathbf{10^{38}}$).

- **Mantisa** (23 bits normalizados 1.M) : indica la **precisión** o el **número de decimales** que el computador asigna para un número real dado. Los valores binarios representables oscilan entre 1.00...0 y 1.11...1 (1 y $2-2^{-23}$).

El formato simple IEEE

32 bits

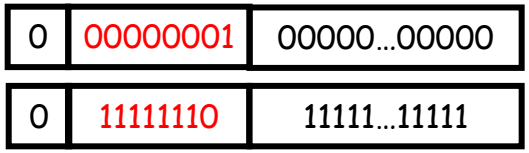
Valores del exponente ($a_1 \ldots a_8$)	Valore numérico representado
$(00000000)_2=(0)_{10}$	$\pm(\mathbf{0}.b_1b_2b_3\ldots b_{23})_2 \cdot 2^{-126}$ → Número subnormal
$(00000001)_2=(1)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \cdot 2^{(1-127)=-126}$ N_{min}
$(00000010)_2=(2)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \cdot 2^{-125}$
$(00000011)_2=(3)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \cdot 2^{-124}$
↓	↓
$(01111111)_2=(127)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \cdot 2^{(127-127)=0}$
$(10000000)_2=(128)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \cdot 2^{(128-127)=1}$
↓	↓
$(11111100)_2=(252)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \cdot 2^{125}$
$(11111101)_2=(253)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \cdot 2^{126}$
$(11111110)_2=(254)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{23})_2 \cdot 2^{(254-127)=+127}$ N_{max}
$(11111111)_2=(255)_{10}$	$\pm\infty$ si $b_1=\ldots=b_{23}=0$; si no, NaN

254 → exceso (254/2)=127

Épsilon de máquina: $\varepsilon=2^{-23}\approx 1.2\times10^{-7}$

$$N_{min}=(1.000\ldots0)_2\times2^{-126}=2^{-126}\approx1.2\times10^{-38}$$

$$N_{max}=(1.111\ldots1)_2\times2^{127}=(2-2^{-23})\times2^{127}\approx2^{128}\approx3.4\times10^{38}$$



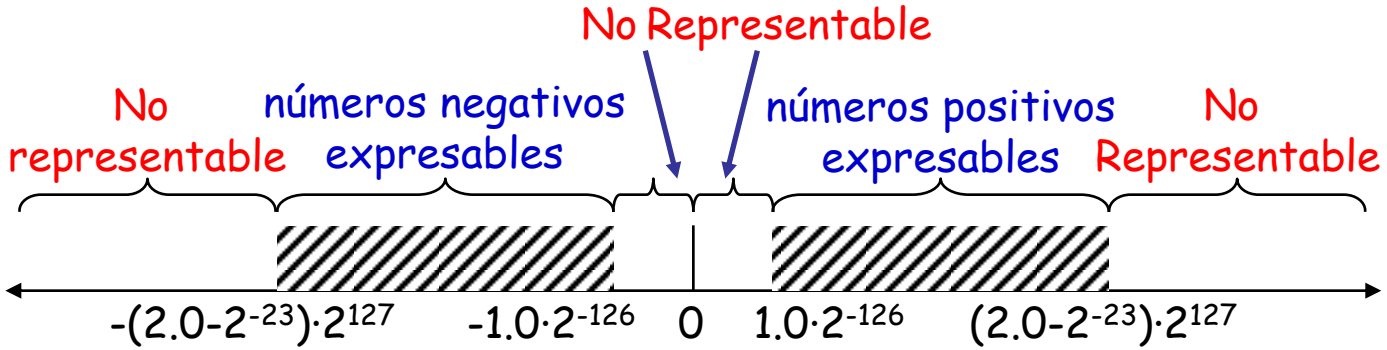
UCM

Representación en Punto Flotante (IV)

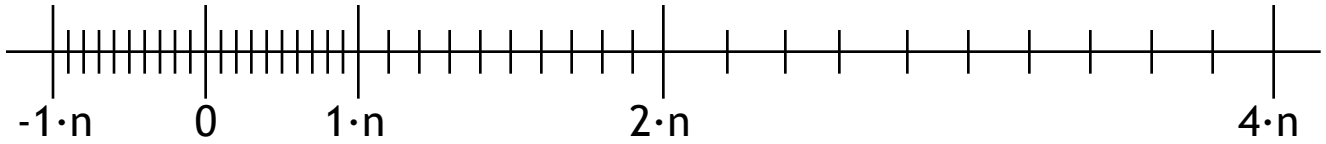
Un número x arbitrario se representará de la forma:

$$(x)_2 = (-1)^{(sm)_2} \times (1.m_1 \ldots m_{23}) \times 2^{(-1)^{(se)_2}(e_7 \ldots e_1)_2 - (127)_{10}}$$

✓Rango Representable: en coma flotante con mantisa normalizada nos encontramos con un hueco alrededor del cero de números reales que no se pueden representar. El cero tampoco se podrá representar salvo que se le asigne una representación especifica.



✓La cantidad de números representables es 2^{32}. El espacio de representación no es uniforme : Mayor densidad cerca del 0 !!



✓Precisión: Depende del número de bits de la mantisa. Con 23 bits es imposible distinguir entre 2 números que se diferencien en $2^{-23} \approx 10^{-7}$. Precisión 23 bits, precisión 7 dígitos decimales, épsilon de máquina (eps) del computador: 2^{-23} (indica el espacio vacío entre 1 y el siguiente número de punto flotante).

IEEE 754 (I)

- ✓ Hasta la década de los 90 cada computador utilizaba su propio formato en punto flotante (numero de bits para exponente y mantinsa, como se realiza el redondeo, acciones tomadas en condiciones excepcionales) :
  - ★ En algunas máquinas (Crays, IBM System 370) era necesario utilizar ciertos trucos para obtener resultados correctos.

    Ejemplo: En algunos computadores, algunos números próximos a cero sólo se consideraban distintos de cero en comparaciones y sumas (se trataban como ceros en multiplicaciones y divisiones). Consecuencia: Para poder utilizar de forma segura una variable como denominador debía multiplicarse por 1.0 antes de comparar con cero!!
  - ★ **Era imposible escribir programas portables que produjesen los mismos resultados en máquinas diferentes** (Sólo el pentágono, AT&T... Podían permitírselo)
- ✓ John Palmer (Intel) promueve la creación de un estándar para punto flotante (1976) antes de comenzar el diseño del coprocesador matemático del i8086/8 y del i432. Primeras reuniones comenzaron en 1977
- ✓ El primer borrador 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)
- ✓ Proceso de estandarización bastante lento. El estándar no fue introducido hasta 1985
- ✓ El i8087 (coprocesador) fue la primera implementación comercial importante (1981)
- ✓ El i486 fue el primer μProcesador que implementaba el estándar (1989)

IEEE 754 (II)

Precisión simple (32 bits): Exponente Exceso 127

Exponente (8)	Mantisa (23)	Representa
-127 (combinación binaria 0)	0	$(-1)^{(sm)_2} \cdot (0.0000...0)_2 \cdot 2^{\underbrace{(00000000)_2-(127)_{10}}_{2^{(-127)_{10}}}} = (\pm 0)_{10}$
-127 (combinación binaria 0)	$\neq 0$	$(-1)^{(sm)_2} \cdot (0.m_{23}...m_1)_2 \cdot 2^{(-126)_{10}}$
[-126, 127] ($[1, 254]_{10}$)	$\forall$	$(-1)^{(sm)_2} \cdot (1.m_{23}...m_1)_2 \cdot 2^{(e_8...e_1)_2-(127)_{10}}$
128 (combinación binaria 255)	0	$(-1)^{(sm)_2} \cdot (0.0000...0)_2 \cdot 2^{\underbrace{(1111111)_2-(127)_{10}}_{2^{(255-127)_{10}=2^{(128)_{10}}}}} = (\pm \infty)_{10}$
128 (combinación binaria 255)	$\neq$	$(-1)^{(sm)_2} \cdot (0.m_{23}...m_1)_2 \cdot 2^{(128)_{10}} = (NaN)_{10}$

Precisión doble (64 bits): Exponente Exceso 1023

Exponente (11)	Mantisa (52)	Representa
-1023 (comb. binaria 0)	0	$(-1)^{(sm)_2} \cdot (0.0000...0)_2 \cdot 2^{\underbrace{(00000000000)_2-(1023)_{10}}_{2^{(-1023)_{10}}}} = \pm(0)_{10}$
-1023	$\neq 0$	$(-1)^{(sm)_2} \cdot (0.m_{52}...m_1)_2 \cdot 2^{(-1022)_2}$
[-1022, 1023] ($[1, 2046]_{10}$)	$\forall$	$(-1)^{(sm)_2} \cdot (1.m_{52}...m_1)_2 \cdot 2^{(e_{11}.....e_1)_2-(1023)_{10}}$
1024 (comb. binaria 2047)	0	$(-1)^{(sm)_2} \cdot (0.0000...0)_2 \cdot 2^{\underbrace{(11111111111)_2-(1023)_{10}}_{2^{(2047-1023)_{10}=2^{(1024)_{10}}}}} = \pm(\infty)_{10}$
1024	$\neq$	$(-1)^{(sm)_2} \cdot (0.m_{52}...m_1)_2 \cdot 2^{(1024)_{10}} = (NaN)_{10}$

Codificaciones con significado especial (IV)

e=11111111
(Combinación binaria 255)

m=0 (23 bits =0) ⟶ **+∞ o - ∞**

Representan cualquier valor de la región de overflow. El estándar especifica aritmética en el infinito. Ejemplo: **1/ ∞=0, atan(∞)=π/2.**

m>0 ⟶ **NaN (resultado sin sentido)**

Se obtiene n como resultado de operaciones inválidas. El estándar especifica que cuando el argumento de una operación es un NaN, el resultado es NaN.
Ejemplo: **0 x ∞, 0/0 y ∞- ∞.** Estas operaciones no tienen sentido matemático.

e=00000000
(Combinación binaria 0)

m=0000…0 ⟶ **0 decimal**

El cero se representa como número no normalizado.

m>0 ⟶ **Números desnormalizados**

Número sin normalizar cuyo bit implícito se supone que es 0. Permiten representar número en las regiones de underflow (desbordamiento a cero gradual). Es decir, sirve para rellenar el hueco existente en torno al cero (véase los números subnormales).

El formato doble IEEE

64 bits

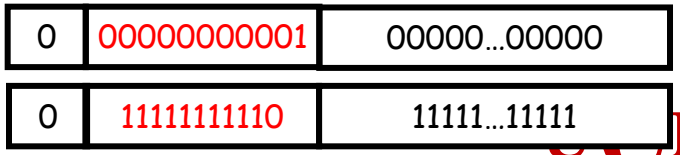
Valores del exponente ($a_1 \ldots a_{11}$)	Valore numérico representado	
$(00000000000)_2=(0)_{10}$	$\pm(0.b_1b_2b_3\ldots b_{52})_2 \cdot 2^{-1022}$	→ Número subnormal
$(00000000001)_2=(1)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \cdot 2^{(1-1023)=-1022}$	N_{min}
$(00000000010)_2=(2)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \cdot 2^{-1021}$	
$(00000000011)_2=(3)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \cdot 2^{-1020}$	
↓	↓	
$(01111111111)_2=(1023)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \cdot 2^{0}$	
$(10000000000)_2=(1024)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \cdot 2^{1}$	
↓	↓	
$(11111111100)_2=(2044)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \cdot 2^{1021}$	
$(11111111101)_2=(2045)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \cdot 2^{1022}$	
$(11111111110)_2=(2046)_{10}$	$\pm(1.b_1b_2b_3\ldots b_{52})_2 \cdot 2^{(2046-1023)=+1023}$	N_{max}
$(11111111111)_2=(2047)_{10}$	$\pm\infty$ si $b_1=\ldots=b_{52}=0$; si no, NaN	

2046→exceso (2046/2)=1023

Épsilon de máquina: $\varepsilon=2^{-52}\approx 2.2\times10^{-16}$

$$N_{min}=(1.000\ldots0)_2\times2^{-1022}=2^{-1022}\approx2.2\times10^{-308}$$

$$N_{max}=(1.111\ldots1)_2\times2^{1023}=(2-2^{-23})\times2^{1023}\approx2^{1024}\approx1.8\times10^{308}$$



IEEE 754 (IV).Ejemplos

$$(x)_{10} = (-1)^{(sm)_2} \times (1 + \textit{fración}) \times 2^{(Exponente - Exceso)_{10}}$$

Ejemplo 1: ¿ Cuál es el valor decimal de:
1 01111100 11000000000000000000000?

- ✓ El bit de signo es 1 : **número negativo**
- ✓ El exponente contiene 01111100 = $(124)_{10}$
- ✓ La mantisa es 0.11000… = $(0.75)_{10}$

El valor es : $(-1)^1 \times (1 + 0.75) \times (2^{(124-127)}) = -1.75 \times (2^{(-3)}) =$ - 0.21875

Ejemplo 2: ¿ Cuál es el valor decimal de:
0 10000001 01000000000000000000000?

- ✓ El bit de signo es 0 : **número positivo**
- ✓ El exponente contiene 10000001 = $(129)_{10}$
- ✓ La mantisa es $0x2^{-1} + 1x2^{-2} = (0.25)_{10}$

El valor es : $(-1)^0 \times (1 + 0.25) \times (2^{(129-127)}) = 1.25 \times (2^{(2)}) =$ + 5.0

IEEE 754 (V).Ejemplos

Ejemplo 3: ¿ Cuál es la representación en simple precisión de : 347.625 ?

1- Convertir a binario: 347.625 = 101011011.101
2-Normalizar el número (mover el punto decimal hasta que haya un solo 1 a la izquierda) 101011011.101 =1.01011011101 x (2 ^ (8))
3- mantisa: 01011011101
4- exponente: 8 en exceso 127 → 8=(exponente-exceso 127) → exponente=(127+8) = $(135)_{10}$ = 10000111
5- El número es positivo: bit de signo 0

Resultado : 0 10000111 01011011101000000000000

Ejemplo 4: *Representar el número decimal 5/3 en precisión simple.*

$(5/3)_{10}$= 1.6666... x 10^0

Mantisa: 1.66666....

⇒ Parte entera: 1 x 2^0=1

⇒ Parte decimal: 0.666... x 2=1.333...

0.333... x 2=0.666... (Es cíclico. Infinitos decimales)

Por tanto $(0.66...)_{10}$=$(0.101010...)_2$

Luego: $(1.101010)_2$ binario puro sin signo =$(1.66...)_{10}$

Exponente: exp-127=$(0)_{10}$ ⇒ exp= $(127)_{10}$= $(01111111)_2$

El número representado como 5/3 será el número máquina más próximo:

0 01111111 10101010101010101010101

Errores en la representación de números reales

Número real que no sea un número máquina

Se puede deber a:

1. Que el exponente, una vez normalizado, esté fuera del rango admitido (demasiado grande o pequeño).
2. Que la mantisa, una vez normalizada, tenga más de 23 bits (más bits de los que puede almacenar).

¿Cómo resolverlo?

1. Si el exponente se sale del rango admitido se produce el fenómeno de **desbordamiento**, bien por exceso (*overflow*) bien por defecto (*underflow*).
   Desbordamiento por exceso: toma, p.e., el valor $\pm\infty$ (según sea el número positivo o negativo), y sigue trabajando siempre y cuando tengan sentido las operaciones siguientes (p.e. $1/\infty=0$). Normalmente es el resultado de un error de cálculo que no ha considerado el programador.
   Desbordamiento por defecto: se asigna, por ejemplo, el valor cero y continua el cálculo reemplazando el resultado por cero o se desnormaliza el número para poder tener en número máquina próximo en el entorno cero.

2. Cuando se necesita trabajar con números no máquina que no producen desbordamiento lo que se hace es aproximarlos por números máquina cercanos (**redondeo**).

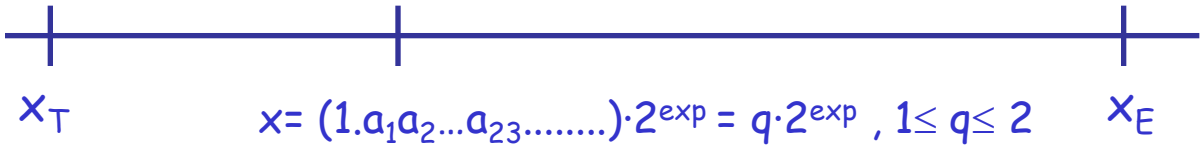
Error de Redondeo Unitario (I)

Representación en punto flotante de x-> $fl(x)=(1.a_1a_2\ldots a_{23})\cdot 2^{exp}$ (suponemos 23 bits de mantisa)

2^{-23}

Error de redondeo unitario: Depende del nº de bits de la mantisa
épsilon (*eps*) o precisión del computador

- TRUNCAMIENTO: $fl(x)=x_T=(1.a_1a_2\ldots a_{23})*2^{exp}$
  Se descartan el resto de bits
- EXCESO: $fl(x)=x_E=((1.a_1a_2\ldots a_{23})+2^{-23})*2^{exp}$
  El último bit se incrementa en uno
- REDONDEO: $fl(x)=\begin{cases} x_T \\ x_E \end{cases}$ Se usa el más cercano



Error absoluto$=|x-x_T| \leq 1/2|x_E-x_T| = (1/2)\cdot 2^{-23}\cdot 2^{exp} = 2^{exp-24}$

Error relativo$=|x-x_T|/|x| = 2^{exp-24}/(q\cdot 2^{exp}) = 2^{-24}/q \leq 2^{-24}$

Error de Redondeo Unitario (II)

Formato n Bits $eps=2^{-n}$

"Axioma de la Representación en punto Flotante"

$$fl(x) = x_{redondeado} = x\cdot(1+\delta); \qquad |\delta| \le eps$$

Precisión del sistema: Número de bits de la mantisa (incluido el bit escondido)

Matlab:

$$eps=2^{-(p-1)}=2^{-52}=2.22044\ e-016$$

Si está en efecto el modo de redondeo al más cercano:

$$|\delta| \le \frac{1}{2} eps = 2^{-p}$$

Ejemplo: Error al representar $(2/3)_{10}$ en formato simple IEEE754

Calcular el error de truncado absoluto y relativo
Cual es la representación final

Error de Redondeo Unitario (III)

Ejemplo: Error al representar $(2/3)_{10}$ en formato simple IEEE754

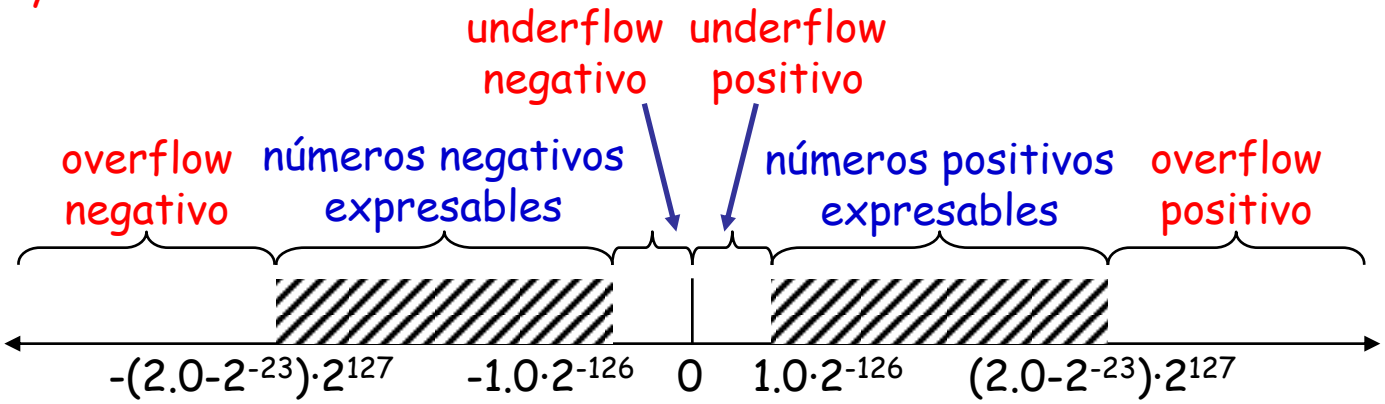
Calcular el error de truncado absoluto y relativo
Cual es la representación final

- Se convierte en binario:
  $(2/3)_{10}=(0.1010\ldots)_2=(1.010101\ldots)_2 x2^{-1}$
- Los números de maquinas más cercanos son:
  $x_-=(1.0101\ldots010)_2 x2^{-1}$ ← Obtenido truncando
  $x_+=(1.0101\ldots011)_2 x2^{-1}$ ← Obtenido redondeando
- Determinamos el más cercano calculando $(x-x_-)$ y (x_+-x)
  $(x-x_-)=(0.1010\ldots)_2 x2^{-24}=(2/3)_{10} x2^{-24}$
  $(x_+-x)=(x_+-x_-)-(x-x_-)=2^{-24}-(2/3)_{10} x2^{-24}=\boxed{(1/3)x2^{-24}}$ ← $fl(x)=x_+$
- Error de truncado absoluto:
  $|fl(x)-x|=(1/3)x2^{-24}$
- Error de truncado relativo:
  $|fl(x)-x|/|x|=(1/3x2^{-24})/(2/3)_{10}=2^{-25}$
- Representación final:
  $(0.666\ldots)_{10}=(0.101010\ldots)_2=(\mathbf{1}.01010\ldots)_2 x2^{-1}$
  Exponente: $exp-127=(-1)_{10} \Rightarrow exp=(126)_{10}=(01111111)_2$
  El número máquina más próximo será:
  0 01111111 01010101010101010101011

Error de Desbordamiento (I)

- ✓ Solo se puede representar un rango limitado de valores
  1. Principalmente dependen del nº de bits asignados al exponente

✓Overflow y Underflow :



- ✓ Proteger las conversiones de tipos utilizando los límites representables (**realmax**, **intmax**...)

realmax=1.79769313486231e+308

intmax= 2147483647

- ✓ Utilizar códigos especiales para marcar que se ha producido desbordamiento, *+-inf*

If a>intmax then a=intmax

If a~=0 then c=b/a
else c=b/eps

Ejemplo de desbordamiento en Matlab:
Cálculo de distancias $(x^2+y^2)^{1/2}$ en Doble Precisión

✓ si $x > 2^{Emax/2} \Rightarrow$ No podemos aplicar la fórmula.

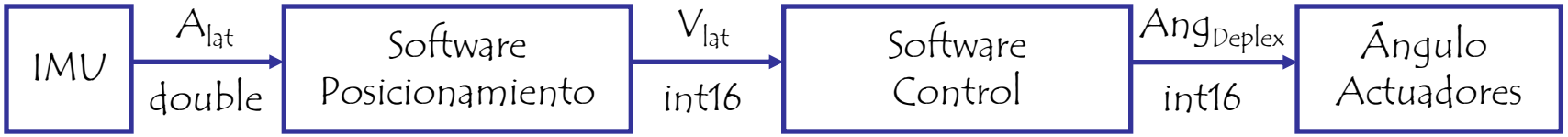
```
x = 2 ^((1023/2)+1)
y = 2
d = sqrt((x^2)+(y^2))
d = Inf  ←
```

Error de Desbordamiento (II)

Cuidado con las Conversiones (punto flotante → entero)

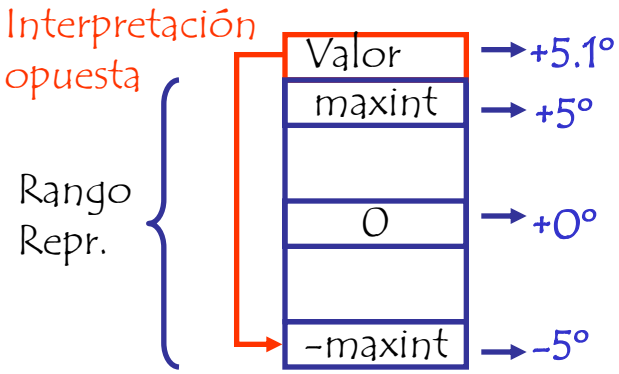
Guayana Francesa, 4 Junio 1996. Un Cohete **Ariane V** de la Agencia Espacial Europea estalla a los 40 segundos del despegue. Era el primer lanzamiento del Ariane V, después de 10 años de desarrollo (7 billones de $). El cohete y su carga estaban valorados en 800 millones de $.

El error fue debido a un desbordamiento en el software de Posicionamiento. Cuando se presentó la operación invalida, el programa cerró todo el sistema de guía y el cohete se autodestruyó.



La velocidad horizontal del cohete con respecto a la plataforma se convertía a un entero (con signo) de 16 bits.

El número era superior a +32768 (2^{15}, es decir, el mayor entero con signo representable con 16 bits) y la conversión fallo.



Acumulación de Errores de Redondeo (I)

Ejemplo : **Suma en punto flotante** (99.99 + 0.161 = 100.151). Suponemos que:

- ✓ La mantisa puede ser de **4** dígitos decimales
- ✓ El exponente puede tener **2** dígitos

•Extraer signos, exponentes y magnitudes.
•Tratar operandos especiales (por ejemplo, alguno de ellos a cero)
•Desplazar la mantisa del número más pequeño a la derecha $|e_1-e_2|$ bits
•Fijar el exponente del resultado al máximo de los exponentes

1. Alineamiento

$9.999 \cdot 10^{1}$
$1.610 \cdot 10^{-1}$

$9.999 \cdot 10^{1}$
$+ \quad 0.0161 \cdot 10^{1}$

10.015×10^{1}

3. normalización

$1.0015 \cdot 10^{2}$

$1.002 \cdot 10^{2}$

4. redondeo

2. Operación

- Si la operación es suma y los signos son iguales, o si la operación es resta y los signos son diferentes, sumar las mantisas. En otro caso restarlas
- Detectar Overflow de la mantisa

- Redondear el resultado y renormalizar la mantisa si es necesario (IEE754 ultimo par si hay duda)
- Corregir el exponente en función de los desplazamientos realizados sobre la mantisa.
- Detectar overflow o underflow del exponente

Error de Redondeo Relativo: 0.0005/1.002≈0.049 %

Error Rel. Operación: 0.049/100.151≈0.0489 %

5. Re-normalización: A veces es necesario volver a normalizar (9.9999 → 10.000 →...)

Acumulación de Errores de Redondeo (II)

Hay **SITUACIONES EXTREMAS** en las que no se puede evitar considerables errores de redondeo: ERRORES DEBIDO A LA PERDIDA DE DIGITOS SIGNIFICATIVOS

Ejemplo 1: Suma de cantidades grandes a números pequeños

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

Alineamiento : es necesario desplazar el punto decimal 38 posiciones a la izquierda

Con 16 dígitos decimales de mantisa ese desplazamiento supone convertir al segundo sumando en 0!

No hay Propiedad Conmutativa !!!!!

$x = 1.5 \cdot 10^{38}$
$y = -1.5 \cdot 10^{38}$

$(x + y) + 1 \neq x + (y + 1)$

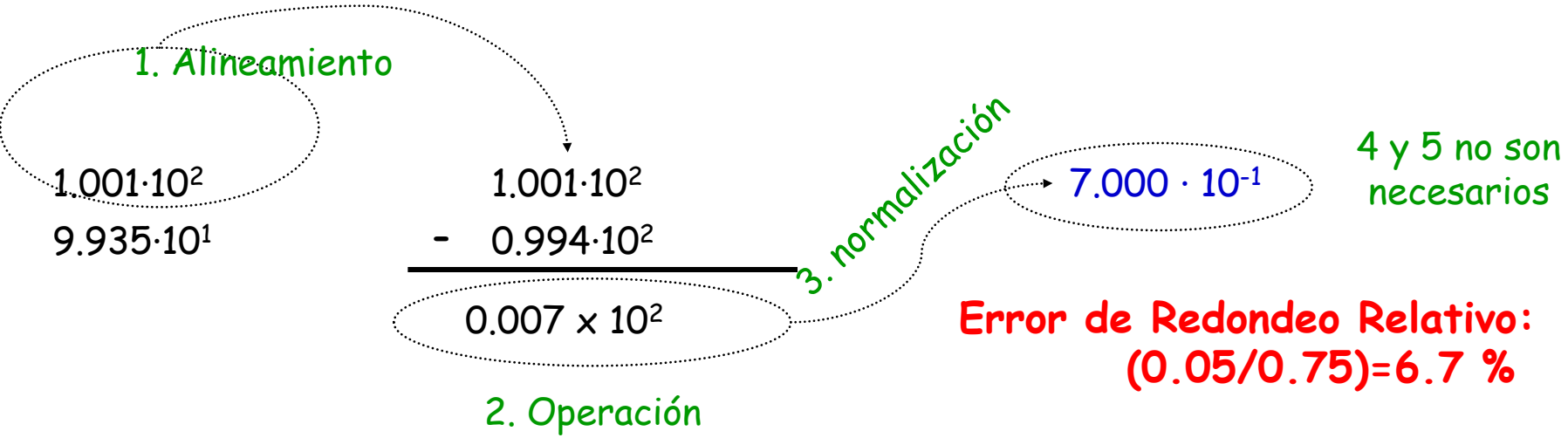
$(x + y) + 1 \longrightarrow 1$

$x + (y + 1) \longrightarrow 0$

Acumulación de Errores de Redondeo (III)

Ejemplo 2 : **Sustracción de cantidades casi iguales** (100.1 - 99.35 = 0.75).

- ✓ La mantisa puede ser de **4** dígitos decimales
- ✓ El exponente puede tener **2** dígitos

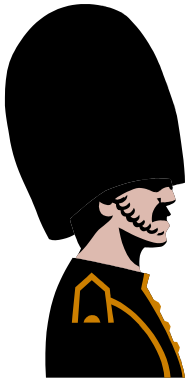


Error de Redondeo Relativo: (0.05/0.75)=6.7 %

Problema: En el Alineamiento se pierde 1 dígito significativo

En un sistema de representación en punto flotante binario, las sustracciones pueden cometer un error de redondeo relativo del **50%**

Acumulación de Errores de Redondeo (IV)



Serie 360 de IBM (1968)

Dígitos de Protección (*guard digits*) :

Para minimizar este problema, el estándar IEEE 754 especifica que los cálculos intermedios deben realizarse utilizando 2 dígitos de protección.

Ejemplo 2 revisado

1. Alineamiento

$1.0010 \cdot 10^2$

$9.9350 \cdot 10^1$

$1.0010 \cdot 10^2$

$- \; 0.9935 \cdot 10^2$

2. Operación

0.0075×10^2

3. normalización

$7.5000 \cdot 10^{-1}$

4 y 5 no son necesarios

Resultado Correcto

Si utilizamos 2 dígito de guarda para realizar las substracciones el error de redondeo relativo será siempre menor que eps

Acumulación de Errores de Redondeo (V)

Supongamos un computador decimal de **5** dígitos **(epsilon del computador = 10^{-5})** en el que se utiliza formato extendido para cálculos intermedios y dos números maquina x e y:

$x = 3.1426 \cdot 10^{2}$ $y = 9.2577 \cdot 10^{4}$

OPERACIÓN	RESULTADO NORMALIZADO	REDONDEO	ERROR RELATIVO
Multiplicación	$2.909324802 \cdot 10^{7}$	$2.9093 \cdot 10^{7}$	$8.5 \cdot 10^{-6}$
Suma	$9.289126 \cdot 10^{4}$	$9.2891 \cdot 10^{4}$	$2.3 \cdot 10^{-6}$
Resta	$-9.2262740 \cdot 10^{4}$	$-9.2263 \cdot 10^{4}$	$2.8 \cdot 10^{-6}$
División	$3.394579647 \cdot 10^{-3}$	$3.3946 \cdot 10^{-3}$	$6.0 \cdot 10^{-6}$

- ✓ x e y números máquina (es decir, son representables de forma exacta)
- ✓ $\odot$ operación aritmética básica (+, -, $\times$, $\div$)

Axioma fundamental de la aritmética en punto flotante:

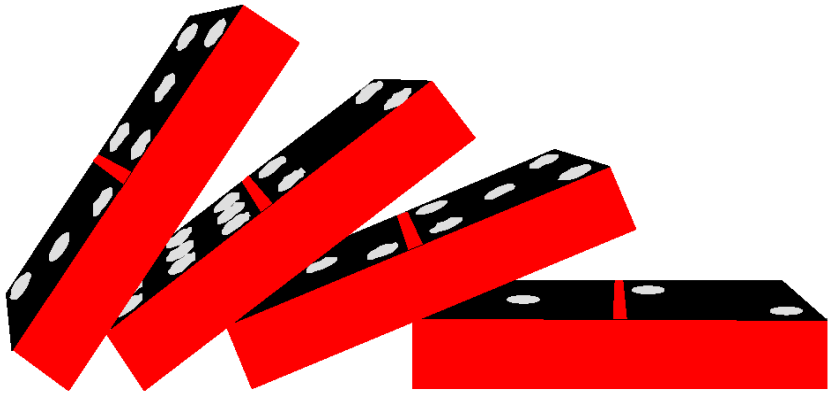
$$\text{Flotante}(x \odot y) = (x \odot y)(1+\delta) \qquad |\delta| \leq eps$$

Epsilon del computador (salvo posibles discrepancias): Cota Superior del error de redondeo Relativo en cualquier aritmética básica.

Acumulación de Errores de Redondeo (VI)

Cómo se acumulan los errores al encadenar varias operaciones?

A pesar de que el error de redondeo al realizar una operación aritmética básica pueda ser pequeño, al encadenar operaciones dichos errores pueden magnificarse (acumularse) considerablemente.



Ejemplo: x, y, z números máquina, se desea realizar la operación $(x \cdot (y+z))$. Siendo $|\delta_1|$, $|\delta_2|$ y $|\delta| \leq eps$.

$$\begin{aligned} Flotante\,(x \cdot (y+z)) &= (x \cdot Flotante\,(y+z))\,(1+\delta_1) && |\delta_1| \leq eps \\ &= (x \cdot (y+z))\,(1+\delta_1)\cdot(1+\delta_2) && |\delta_2| \leq eps \\ &\approx (x \cdot (y+z))\,(1+2\delta) && |\delta| \leq eps \end{aligned}$$

El error de redondeo relativo se multiplica por 2. **Teorema**: Sean *n* números máquina positivos. El error de redondeo relativo al calcular la suma de dichos números es aproximadamente igual a *n·eps.*

Acumulación de Errores de Redondeo(VII). Ejemplo

Guerra del Golfo, 25 Febrero 1991. Una batería de antimisiles Patriot en Dharan (Arabia Saudita), no consiguió interceptar un misil Scud iraquí, que impactó contra un barracón de la armada americana, muriendo 28 soldados.

La predicción de la "puerta de alcance" (*range gate*) se calcula en función de:

- ✓ Velocidad del Scud (dato conocido : 1676 m/s)
- ✓ Tiempo transcurrido desde que el Scud fue detectado por el radar por última vez

✓ Para las mediciones de tiempo, el sistema utiliza un reloj interno con una precisión de décimas de segundo. El tiempo transcurrido desde el *boot* se almacenaba en un registro como un valor entero.

Acumulación de Errores de Redondeo(VIII). Ejemplo

El tiempo en segundo se almacena en un registro de 24 bits. Aunque el reloj interno daba décimas de segundo, para hacer los cálculos, el número entero con las décimas de segundo transcurridas se convertía a punto flotante, multiplicándose el valor por 0.1 para obtener el tiempo en segundos. Es decir, **el reloj interno de la computadora que controlaba el sistema de defensa almacenaba el tiempo como un valor entero en unidades de décimas de segundo**, y el programa de la computadora lo convertía a un valor de punto flotante en unidades de segundos, redondeando consiguientemente la expansión.

¿Qué pasa con el 0.1?

¿ Cuál es la representación en <u>simple precisión</u> de: $(0.10)_{10}$?

1- Convertir a binario: 0.10 = 0.0001100110011001100110011001100 ...
2- Normalizar el número : $1.10011001100110011001100 \ldots \times 2^{-4}$
3- Truncar el número: 23 bits de mantisa!!! $1.10011001100110011001100 \times 2^{-4}$

0.10 no se puede escribir como una suma finita de potencias de 2 !!!

0.10 se representa en precisión simple por el número 0.099999940395352
El error cometido es : $5.960464483090178 \times 10^{-09}$

```
A=2^-4; v=[10011001100110011001100];
for i=1:23
A=A+v(i)*1/(2^(i+4));
end
error=0.1-A
```

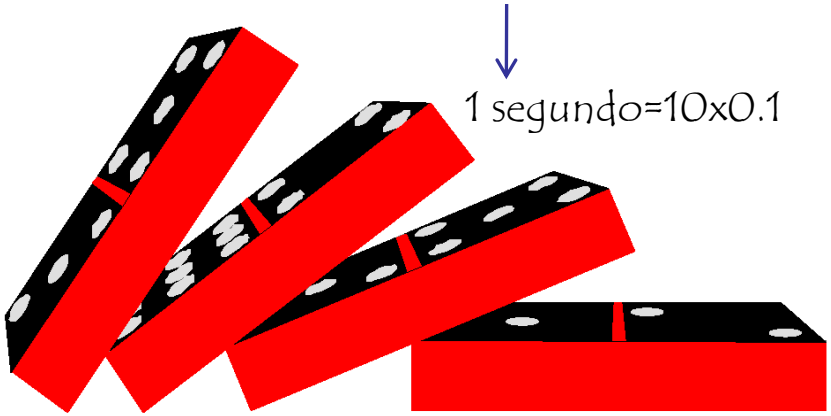
Acumulación de Errores de Redondeo(IX). Ejemplo

Batería Antimisiles Patriot:El error de redondeo cometido al representar 0.1 es ≈ de 5.9×10^{-9}.

0001100110011001100110011

La batería llevaba unas 100 horas en funcionamiento antes de producirse el fallo. **El error cometido al representar 0.1 SE HABIA ACUMULADO**

$5.9 \times 10^{-9} \times (100 \times 60 \times 60 \times 10) =$ **0.0212 segundos**



En ese intervalo, un misil *Scud* recorre **35.60 m**. Este error solo se tenía en cuenta en algunas de las partes del software **(LOS ERRORES NO SE CANCELABAN!!!!)**

Fuente: General Accounting Office, GAO/IMTEC-92-26 " Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia"

Anulación Catastrófica (II)

Regla Práctica: Se debe evitar situaciones en las que se restan cantidades casi iguales

Ejemplo 1

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

Implica una cancelación por sustracción y pérdida de dígitos significativos para valores **pequeños de x**

¿Cómo evitarlo?

Solución: utilizar una forma alternativa de asignación

Multiplicando y Dividiendo por el Conjugado

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

Anulación Catastrófica (IV)

Ejemplo 3

$$y \leftarrow x - \sin(x)$$ Implica una cancelación para valores pequeños de x

¿Cómo evitarlo? Desarrollo en serie de Taylor para sin(x)

$$\begin{aligned} y &= x - \sin(x) \\ &= x - \left(x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots \right) \\ &= \frac{x^3}{3!} - \frac{x^5}{5!} + \frac{x^7}{7!} + \cdots \end{aligned}$$

Si x está cercano a cero, se puede utilizar una serie truncada

$$y \leftarrow \frac{x^3}{6}\left\{1 - \frac{x^2}{20}\left[1 - \frac{x^2}{42}\left(1 - \frac{x^2}{72}\right)\right]\right\}$$

Si se necesitase un rango amplio de valores para x, lo más conveniente es utilizar ambas expresiones (el error cometido al utilizar la serie truncada crece x), cada uno con su propio rango

Cálculos Estables e Inestables (I)

Proceso/Algoritmo Numérico Inestable (Informal) :

Los pequeños errores que se producen en alguna de sus etapas se agrandan en etapas posteriores y degradan seriamente la exactitud del resultado final, a pesar de que en aritmética exacta el algoritmo sea correcto.

Algoritmo Estable *(backward stability)* :

Si alg(x) representa la versión discreta de un algoritmo ALG(x), en el que se incluye el efecto del error de redondeo, se dice que alg(x) es estable si para todo los datos de entrada x existe un pequeño δx tal que $alg(x)=ALG(x+\delta x)$

backward error

EJEMPLO 1: sucesión de números reales

$$\begin{cases} x_0 = 1 \\ x_1 = \frac{1}{3} \\ x_{n+2} = \frac{13}{3}x_{n+1} - \frac{4}{3}x_n \end{cases}$$

Por inducción, se puede demostrar que es equivalente

$$x'_n = \left(\frac{1}{3}\right)^n = \frac{1}{3}x'_{n-1}$$

Vamos a suponer que x'_n son los valores exactos de la sucesión y con ellos vamos a estimar el error de redondeo de x_n.

Cálculos Estables e Inestables (II)

# iter	Valor Inestable x_n	Valor Estable x'_n	Error Absoluto $\lvert x_n - x'_n \rvert$	Error Relativo $\left\lvert \frac{x_n - x'_n}{x_n} \right\rvert$
	1.0000e+000	1.0000e+000	0	0
	3.3333e-001	3.3333e-001	0	0
1	1.1111e-001	1.1111e-001	1.6653e-016	1.4988e-015
2	3.7037e-002	3.7037e-002	7.7716e-016	2.0983e-014
3	1.2346e-002	1.2346e-002	3.1641e-015	2.5629e-013
4	4.1152e-003	4.1152e-003	1.2675e-014	3.0800e-012
5	1.3717e-003	1.3717e-003	5.0707e-014	3.6966e-011
6	4.5725e-004	4.5725e-004	2.0283e-013	4.4359e-010
7	1.5242e-004	1.5242e-004	8.1133e-013	5.3231e-009
8	5.0805e-005	5.0805e-005	3.2453e-012	6.3878e-008
9	1.6935e-005	1.6935e-005	1.2981e-011	7.6653e-007
10	5.6450e-006	5.6450e-006	5.1925e-011	9.1984e-006
11	1.8815e-006	1.8817e-006	2.0770e-010	1.1038e-004
	6.2639e-007	6.2723e-007	8.3080e-010	1.3246e-003
	2.0575e-007	2.0908e-007	3.3232e-009	1.5895e-002
	5.6399e-008	6.9692e-008	1.3293e-008	1.9074e-001
	-2.9941e-008	2.3231e-008	5.3171e-008	2.2889e+000
	-2.0494e-007	7.7435e-009	2.1269e-007	2.7466e+001
	-8.4816e-007	2.5812e-009	8.5074e-007	3.2959e+002
	-2.9941e-008	2.3231e-008	5.3171e-008	2.2889e+000
	-2.0494e-007	7.7435e-009	2.1269e-007	2.7466e+001
	-8.4816e-007	2.5812e-009	8.5074e-007	3.2959e+002
	-3.4021e-006	8.6039e-010	3.4030e-006	3.9551e+003

- Error absoluto : se multiplica por ≈13/3 en cada iteración, en 10 iteraciones:

$$(13/3)^{10}\ 10^{-16} \approx\ 10^{-10}$$

- x_n es una sucesión monótona decreciente

El error Relativo crece considerablemente

Si, en cambio, la solución creciera, el error relativo sería menor y la solución tolerable