Los misteriosos flotantes

Recordemos lo que sabemos del entorno de Programación Científica Octave. Dijimos que es un entorno de trabajo **interpretado** para el **cálculo científico** y la **visualización de datos**. En consecuencia, Octave utiliza un **lenguaje de alto nivel** que permite concentrarnos en la **resolución de problemas** complejos.

Pero vimos que existe una limitación: la **resolución numérica** de problemas, a diferencia de la resolución matemática simbólica, introduce inevitablemente **errores de aproximación** (también llamados "errores de redondeo").

Discutiremos ahora de **dónde** vienen estos errores. La respuesta es, esencialmente, de la **representación de los valores numéricos** en la arquitectura de los computadores.

El estándar IEEE 754

Vimos, a principios del semestre, que los **números no enteros** normalmente se codifican con el estándar **IEEE 754**, que permite representar este tipo de valores y algunos símbolos numéricos separándolos en tres componentes: **signo**, **mantisa** y **exponente**.

Para entenderlos, dijimos que los computadores usan una representación que se parece a la **notación científica**. Por ejemplo, el número 137.5625 puede representarse como el número $+1.375625 \times 10^2$. Pero esta expresión está en **base decimal**, y se cumple la siguiente relación de los dígitos, la posición que usan y potencias de 10 correspondientes:

$$\begin{aligned}
137.5625 &= 1\cdot 100 + 3\cdot 10 + 7\cdot 1 + 5\cdot 1/10 + 6\cdot 1/100 + 2\cdot 1/1.000 + 5\cdot 1/10.000 \\
&= 1\cdot 10^{2} + 3\cdot 10^{1} + 7\cdot 10^{0} + 5\cdot 10^{-1} + 6\cdot 10^{-2} + 2\cdot 10^{-3} + 5\cdot 10^{-4} \\
&= (1\cdot 10^{0} + 3\cdot 10^{-1} + 7\cdot 10^{-2} + 5\cdot 10^{-3} + 6\cdot 10^{-4} + 2\cdot 10^{-5} + 5\cdot 10^{-6})\cdot 10^{2} \\
&= +1.375625 \times 10^{2}
\end{aligned}$$

Esta notación es cómoda para los seres humanos, pero no para los computadores, que usan una **base binaria** (***bits***). El computador entonces, tiene que representar el valor con unos y ceros. La clave está en representar el valor con potencias de 2, positivas o negativas.

$$\begin{aligned}
137.5625_{(10)} &= 1\cdot 2^{7} + 0\cdot 2^{6} + 0\cdot 2^{5} + 0\cdot 2^{4} + 1\cdot 2^{3} + 0\cdot 2^{2} + 0\cdot 2^{1} + 1\cdot 2^{0} + 1\cdot 2^{-1} + 0\cdot 2^{-2} + 0\cdot 2^{-3} + 1\cdot 2^{-4} \\
&= (1\cdot 2^{0} + 0\cdot 2^{-1} + 0\cdot 2^{-2} + 0\cdot 2^{-3} + 1\cdot 2^{-4} + 0\cdot 2^{-5} + 0\cdot 2^{-6} + 1\cdot 2^{-7} + 1\cdot 2^{-8} + 0\cdot 2^{-9} + 0\cdot 2^{-10} + 1\cdot 2^{-11})\cdot 2^{7} \\
&= +(1.00010011001)_{(2)} \times 2^{7}
\end{aligned}$$

Notemos entonces que los dígitos o *bits* antes del **punto decimal** (coma decimal en castellano) se asocian a **potencias con exponente positivo**, mientras que los dígitos o *bits* después del punto decimal, se asocian a **potencias con exponentes positivos**. El valor del exponente de cada potencia corresponde a la **distancia** que el dígito o *bit* tiene respecto de la **primera posición a la izquierda** del punto decimal. Cuando normalizamos, dejamos **sólo un dígito o bit** a la izquierda del punto decimal y **factorizamos** por una potencia común para todas las posiciones.

Teniendo el número en notación científica de base binaria, sólo queda decidir cómo representarlos en la arquitectura del computador. El estándar IEEE 754 ofrece muchas posibilidades, pero generalmente se usan dos de ellas. La primera utiliza **32 *bits*** y se le conoce como **precisión simple**. La Figura 1 muestra un esquema de esta representación.

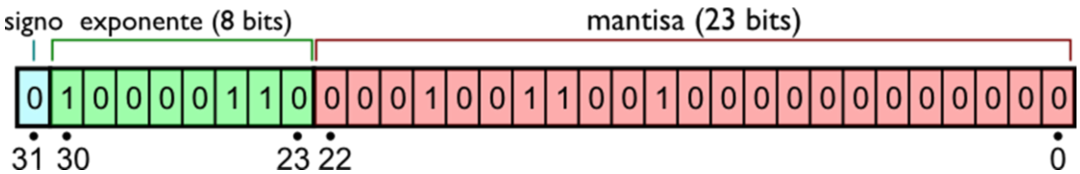

Figura 1. Representación del número $137.5625_{(10)}$ con precisión simple en el estándar IEEE 754.

Recordemos que queremos representar el número no entero $+(1.00010011001)_{(2)} \times 2^7$. Podemos ver que **el primer bit** (31 en la figura) se usa para representar el **signo del valor**. Un **cero** indica que el valor es **positivo** y un **uno** que el valor es **negativo**. Los siguientes **ocho bits** (30 al 23 en la figura) se usan para representar **el exponente** de la potencia común a todas las posiciones. Para facilitar el manejo de exponentes negativos, el valor original es **desplazado en 127**. En el ejemplo, $7_{(10)} + 127_{(10)} = 00000111_{(2)} + 01111111_{(2)} = 10000110_{(2)}$. Los últimos **23 *bits*** (22 al 0 en la figura) se usan para representar la **mantisa** del valor no entero. Como el único *bit* a la izquierda del punto decimal es siempre uno, éste **se omite**. Los otros *bits* ocupan las posiciones **en el orden que aparecen** y **se completa** la totalidad de los *bits* con **ceros**. El valor no entero en base decimal se obtiene con la siguiente fórmula:

$$(-1)^{\text{signo}} \cdot \left(1 + \sum_{i=1}^{23} (b_{23-i} \cdot 2^{-i})\right) \cdot 2^{(\text{exponente}-127)}$$

En el ejemplo, esto sería $(-1)^0 \cdot (1 + 2^{-4} + 2^{-7} + 2^{-8} + 2^{-11}) \cdot 2^{(134-127)} = 1 \cdot 1.07470703125 \cdot 2^7 = 137.5625$, que efectivamente es el valor original.

La otra representación del estándar IEEE 754 que se utiliza representa un valor no entero con **64 *bits*** y se le conoce como **precisión doble**. La Figura 2 muestra el esquema de esta representación con el ejemplo que estamos utilizando.

Figura 2. Representación del número $137.5625_{(10)}$ con precisión doble en el estándar IEEE 754.

En la precisión doble, **el primer bit** (63 en la figura) también se usa para representar el **signo**. Análogamente a la precisión simple, los siguientes **once bits** (62 al 52 en la figura) representan **el exponente** de la potencia común. También existe un desplazamiento de este valor, que en este caso es de **1.023**. Así, el exponente $7_{(10)}$ del ejemplo se convierte en $00000000111_{(2)} + 01111111111_{(2)} = 10000000110_{(2)}$. Los últimos ***52 bits*** (51 al 0 en la figura) se usan para la **mantisa**. También se omite el único *bit* a la izquierda del punto decimal. Luego se almacenan los otros bits (00010011001) y se completa con ceros. El valor no entero en base decimal se obtiene con la siguiente fórmula:

$$(-1)^{\text{signo}} \cdot \left(1 + \sum_{i=1}^{52} (b_{52-i} \cdot 2^{-i})\right) \cdot 2^{(\text{exponente}-1023)}$$

Siguiendo nuestro ejemplo, nuevamente volvemos al valor original aplicando esta fórmula: $(-1)^0 \cdot (1 + 2^{-4} + 2^{-7} + 2^{-8} + 2^{-11}) \cdot 2^{(1030-1023)} = 1 \cdot 1.07470703125 \cdot 2^7 = 137.5625.$

Pregunta 1

Con tu grupo de trabajo, responde la primera pregunta de la actividad.

Para nuestro ejemplo, usar precisión simple o doble no hizo diferencia alguna. Pero esto no simpre es así. Ambas representaciones estándares entregan un **número limitado**, **predeterminado**, de *bits* para representar un valor. Pero, al igual que en la base decimal, existen valores que requieren un número infinito de cifras para escribirlos. Por ejemplo, 0,142857142857142847142857 es sólo una aproximación del número racional $1/_7 = 0,\overline{142857}$. Análogamente, podemos representar una aproximación del número binario racional $0,0\overline{0011}$:

Precisión simple	:	Binario: 00111101110011001100110011001101 Equivalente decimal: 0.100000001490116119384765625
Precisión doble	:	Binario: 0011111110111001100110011001100110011001100110011001100110011010 Equivalente decimal: 0.1000000000000000055511151231257

Estos son los **valores más cercanos** al racional decimal $1/_{10}$ que podemos tener con el estándar IEEE 754. Aquí vemos una diferencia entre la precisión simple y la precisión doble: efectivamente la doble precisión representa un valor más cercano al 0.1 que queremos almacenar. Es una diferencia pequeña, pero puede hacerse importante en cálculos que requieren **iteraciones**. El Ejemplo 1 muestra la salida que entrega el programa `unDecimo.m` que compara los resultados de sumar 500.000 veces el valor $1/_{10}$ usando precisión simple y precisión doble.

unDecimo.m

```
simple = single(0);
doble = double(0);

for i = 1:500000
   simple = simple + single(1/10);
   doble = doble + double(1/10);
end

format long;
disp(simple)
disp(doble)
```

Ejemplo 1

```
octave:1> source('c:/Desktop/unDecimo.m')
 50177.0976562500
 49999.9999995529
octave:2>
```

El programa del Ejemplo 1 utiliza las funciones `single()` y `double()` para indicar **explícitamente** al intérprete de Octave qué representación usar para las variables y operaciones. También podemos ver la sintaxis para **construir iteraciones fijas**: comienzan con la palabra `for` y terminan con la palabra `end`; la variable `i` va tomando, en cada iteración, los valores del vector generado por el operador `:` que le aparece asignado. Al finalizar el ciclo, el programa solicita desplegar los valores, mediante la función `disp()`, con más decimales que lo que normalmente se despliegan.

Podemos ver que hay una diferencia notoria en el resultado final, y que con doble precisión nos acercamos bastante más al verdadero valor (50.000). El **impacto** que pueden tener estos **errores de aproximación** dependerá de la **aplicación** en el mundo real que estemos considerando.

Un ejemplo trágico ocurrió el 25 de febrero de 1991, durante la Primera Guerra del Golfo Pérsico. Las bases militares de EE.UU. estaban protegidas por el **sistema antimisiles Patriot**, que debía interceptar cualquier misil Scub que los iraquíes pudieran lanzarles.

Para evitar falsas alarmas, el radar del sistema Patriot debía detectar un objeto volador con las características de un Scud en **dos puntos distintos**. La posición del segundo punto, de confirmación, se calculaba a partir del primer lugar donde se detectaba el objeto, asumiendo que éste era un misil Scub. Si se le volvía a detectar en el punto de confirmación, el sistema Patriot lanzaba un misil para interceptarlo en pleno vuelo. Pero si no se le hallaba en ese punto, el sistema no lo consideraba un misil Scub y no entraba en acción.

Para calcular la posición de confirmación, se usaba el **tiempo del sistema**, que almacenaba el número de *ticks* de 0,1 segundos desde la puesta en marcha. Como vimos, en un computador, el valor 0,1 es aproximado, muy cerca, pero no exactamente 0,1. Al minuto del ataque, el sistema llevaba funcionando **100 horas**, o 3.600.000 *ticks* del reloj. La suma de estas pequeñas diferencias significó una desviación final de 0,3433 segundos. Un misil Scud recorre **687 metros en este tiempo**, por lo que el sistema Patriot no lo encontró una segunda vez. El misil ya había pasado, un tercio de segundo antes, y llegó a su destino matando a 28 soldados e hiriendo a otros 100.

Podemos ver que lo esencial que resulta que tengamos **conciencia** que para un computador el número 0,1 **no existe**. No importa lo que digan los manuales de programación, **los computadores no manejan números reales**.

Los flotantes

Técnicamente, los computadores manejan **números de punto flotante**, o simplemente **flotantes**, definidos por el estándar IEEE 754. Aunque muchos lenguajes de programación y paquetes de software especializados, entre ellos Octave, llamen a estos números "reales", el conjunto de los flotantes ($\mathbb{F}$) es **distinto** al conjunto de los reales:

$$\mathbb{F} \neq \mathbb{R}$$

En consecuencia, **no todas las propiedades** que son válidas para $\mathbb{R}$ son también válidas para $\mathbb{F}$. Octave hace preferencia por flotantes de doble precisión. En este caso, $\mathbb{F}$ es (2, 53, -1022, 1023), es decir, los números en base 2 con mantisa de 53 *bits* y exponente variando entre -1.022 y 1.023. Eso significa que el **error relativo** entre un número real x y su reemplazante $fl(x) \in \mathbb{F}$ está dado por:

$$\frac{|x - fl(x)|}{|x|} \leq \frac{1}{2} 2^{\text{exponente}-52}$$

El valor $2^{\text{exponente}-52}$ es conocido como eps (por *machine epsilon*) y corresponde a la diferencia entre $fl(x)$ y el flotante más cercano que es mayor que $fl(x)$. A diferencia de $\mathbb{R}$, en donde entre un real y otro siempre hay infinitos valores intermedios, en $\mathbb{F}$ no existe valor alguno entre $fl(x)$ y $fl(x)$ +eps.

En Octave existe la constante predefinida eps que almacena la diferencia entre el valor uno y el siguiente flotante, mayor que uno, que puede representarse. Este valor corresponde a $\text{eps}(1) = 2^{-52} \cong 2{,}22045 \times 10^{-16}$. Podemos ver que este error relativo es bastante pequeño.

```
octave:1> eps
ans = 2.22044604925031e-016
octave:2>
```

```
octave:1> realmin, realmax
ans =  2.22507385850720e-308
ans =  1.79769313486232e+308
octave:2>
```

Como existe un número finito de *bits* para la mantisa y existe un rango fijo para el exponente, hay una **cota superior** y una **cota inferior** para los valores (absoluto) que pertenecen a $\mathbb{F}$: $x_{\text{mín}} = 2^{-1022}$ y $x_{\text{máx}} = 2^{1023}(2 - 2^{-52})$. En Octave, las constantes predefinidas `realmin` y `realmax` contienen estos valores.

```
octave:1> realmax + eps(realmax)
ans = Inf
octave:2> realmin / realmax
ans = 0
octave:3>
```

En consecuencia, tratar de representar un número más pequeño que $x_{\text{mín}}$ o más grande que $x_{\text{máx}}$ resultan respectivamente en errores de **desborde inferior** (*underflow*) y **desborde superior** (*overflow*). Un *underflow* puede identificarse porque el resultado es cero, cuando no debiera serlo, y un *overflow* porque su resultado es infinito. Notemos que la función `eps()` calcula el *machine epsilon* de un flotante cualquiera.

Los errores de desborde también pueden ser nefastos. Un *overflow* en el computador a bordo del cohete Ariane V de la Agencia Espacial Europea causó que éste explosionara en su vuelo inaugural en 1996, convirtiendo en humo y chatarra los US$370 millones que había costado su construcción.

Mirando más detenidamente la función `eps()`, podremos notar que el valor del *machine epsilon* cambia para diferentes flotantes. Ésta es otra consecuencia de la representación interna de los flotantes: El conjunto $\mathbb{F}$ es **más denso** alrededor de $x_{\text{mín}}$ y **más disperso** a medida que se acercan a $x_{\text{máx}}$. De hecho, el valor (absoluto) que inmediatamente sigue a `realmin` en Octave está a una distancia de alrededor de 5×10^{-323} (0,000…317 ceros…005). Por otro lado, el número que inmediatamente precede a `realmax` está a una distancia de alrededor de 2×10^{292} (2000…287 ceros…00). Si bien la **distancia relativa** es pequeña en ambos casos, nos podemos llevar algunas sorpresas. Por ejemplo, para un número real x grande, puede darse perfectamente que $fl(x) + 100.000 = fl(x)$, puesto que el flotante más cercano a $x + 100.000$ sea $fl(x)$ y no $fl(x) +$`eps`$(fl(x))$.

```
octave:1> eps(realmin)
ans =  4.94065645841247e-324
octave:2> eps(realmax)
ans =  1.99584030953472e+292
octave:3> 1000000000000000000000
ans = 1.00000000000000e+021
octave:4> 1000000000000000000000 + 100000
ans = 1.00000000000000e+021
octave:5>
```

Otra fuente de sorpresas es la pérdida de algunas las propiedades de $\mathbb{R}$. En particular, la **conmutatividad** de la **suma** y la **multiplicación** se **mantienen en $\mathbb{F}$**, pero la **asociatividad** y la **distributividad no están aseguradas**. El Ejemplo 2 muestra que es peligroso usar la **igualdad de flotantes** para decidir bifurcaciones en un programa.

sorpresas.m

```
x = 1e-16 + 1 - 1e-16;
y = 1e-16 - 1e-16 + 1;
if(x == y)
   disp('PRUEBA 1: IGUAL')
else
   disp('PRUEBA 1: DISTINTO')
end

fraccion = 58/40 - 1;
fraccionEquivalente = 18/40;
if(fraccion == fraccionEquivalente)
   disp('PRUEBA 2: IGUAL')
else
   disp('PRUEBA 2: DISTINTO')
end

raizDos = sqrt(2);
if(raizDos * raizDos ~= 2)
   disp('PRUEBA 3: DISTINTO')
else
   disp('PRUEBA 3: IGUAL')
end

senoPi = sin(pi);
if(senoPi ~= 0)
   disp('PRUEBA 4: DISTINTO')
else
   disp('PRUEBA 4: IGUAL')
end
```

Ejemplo 2

```
octave:1> source('c:/Desktop/sorpresa.m')
PRUEBA 1: DISTINTO
PRUEBA 2: DISTINTO
PRUEBA 3: DISTINTO
PRUEBA 4: DISTINTO
octave:2>
```

En el programa `sorpresas.m` podemos ver la sintaxis para construir **bifurcaciones**, que en Octave toman la forma de construcciones `if-end`, `if-else-end` y `if-elseif-else-end`. También podemos observar dos de los **operadores de comparación**: `==` es el comparador de **igualdad** y `~=` es el comparador de **desigualdad** ("es distinto que"). También vemos las funciones nativas `sqrt()`, que devuelve la raíz cuadrada de un número, y `sin()` que corresponde a la función seno.

El Ejemplo 2 nos debe dejar una gran lección: tomar decisiones basándonos en la igualdad, o desigualdad, de flotantes puede llevarnos a **comportamientos inesperados**. Por ejemplo, un programa diseñado para que estudiantes de enseñanza media practiquen funciones trigonométricas, podría perfectamente preguntar por "el seno de π" y discutir que la respuesta "0" es incorrecta porque no es igual a su cálculo de `sin(pi)`.

Incluso, a veces ni siquiera podemos confiar en los otros operadores de comparación. Por ejemplo, ¿cuántas iteraciones hace el programa de la derecha? Este programa muestro la sintaxis para construir ciclos `while-end` en Octave. Como en Python, el cuerpo del ciclo se ejecuta mientras la condición se cumpla.

```
suma = 0.0;
while(suma < 1.0)
   suma = suma + 1/10
end
```

Pregunta 2

Ahora responde la segunda pregunta de la actividad, trabajando con tu equipo.

Incluso si evitamos usar flotantes en las condiciones de bifurcaciones y ciclos de nuestros programas, los flotantes nos pueden dar problemas con nuestros cálculos, debido principalmente porque los **errores de aproximación pueden acumularse**. De hecho, existe toda una sub-disciplina de la matemática, conocida como **análisis numérico** o **cálculo numérico**, dedicada al estudio de algoritmos robustos y estables para resolver problemas matemáticos numéricamente. En la mayoría de las ingenierías, este curso está más adelante en el currículum. Este curso será importantísimo para trabajar con operaciones complejas con números flotantes. Por ahora, un resultado del cálculo numérico que ejemplifica estos riesgos:

Archimedes propuso **aproximar el valor de π** calculando los perímetros de polígonos inscribiendo y circunscribiendo un círculo, comenzando con hexágonos, y duplicando el número de lados en cada iteración. Esta idea lleva naturalmente a la siguiente ecuación de recurrencia:

$$t_0 = \frac{1}{\sqrt{3}}, \quad t_{n+1} = \frac{\sqrt{(t_n)^2 + 1} - 1}{t_n}$$

Está demostrado que $3 \cdot 2^{n+1} \cdot t_n$ converge a π a medida que n aproxima infinito.

Pero este algoritmo es **inestable numéricamente**, es decir, cuando se usa en un computador, la fórmula comienza a converger a π por algunas iteraciones, pero luego los errores de aproximación de las operaciones con flotantes hacen que el resultado comience a alejarse.

Es más, esto se resuelve usando la siguiente ecuación recurrente, matemáticamente equivalente, pero **numéricamente estable**.

$$t_0 = \frac{1}{\sqrt{3}}, \quad t_{n+1} = \frac{t_n}{\sqrt{(t_n)^2 + 1} + 1}$$

Pregunta 3

Responde ahora, junto a tu grupo de trabajo, la tercera pregunta de la actividad.

Finalmente, debemos notar que en los ejemplos anteriores han aparecido **valores especiales**. Estos, y varios otros que no discutiremos, tienen una representación especial en el estándar IEEE 754, que no sigue la idea de la notación científica. Aunque no se trata de manera especial por los lenguajes de programación, el **valor cero** es especial porque $\mathbb{F}$ contiene dos representaciones para él, ¡una positiva y otra negativa!

Vimos que está definido el valor **infinito** (`Inf`), positivo y negativo, que usualmente aparece al dividir por valores muy cercanos a cero. Otro valor especial es `NaN` (del inglés *not-a-number*) que aparece cuando una **operación no tiene sentido**. Por último, debemos mencionar el valor `NA` (del inglés *not-available*) que se usa para indicar que un **valor es desconocido**.

Por esta razón, Octave provee funciones nativas para verificar si alguno de nuestros flotantes a tomado un valor especial. Entre ellas: `isinf()`, `isnan()`, `isna()`, y la muy útil `isfinite()`, que retorna verdadero si un valor es finito, es decir no es `Inf`, ni `NaN`, ni `NA`.

Ejemplo 3

```
octave:1> 1/0, 1/-0
warning: division by zero
ans = Inf
warning: division by zero
ans = -Inf
octave:2> 0/0
warning: division by zero
ans = NaN
octave3:> x = [2 Inf 2 NaN 4 NA 4];
octave:4> mean(x)
ans = NA
octave:5> isfinite(x)
ans =

   1   0   1   0   1   0   1

octave:6> mean( x( isfinite(x) ) )
ans =  3
octave:6> NaN == NaN, NA == NA, Inf == Inf
ans = 0
ans = 0
ans = 1
octave:7> NaN - NaN == 0, NA - NA == 0, Inf - Inf == 0
ans = 0
ans = 0
ans = 0
octave:8>
```

En el Ejemplo 3 pueden verse algunos de los valores especiales. Primero notemos que debemos tener cuidado cuando operamos con ellos, puesto que los resultados no son siempre muy intuitivos. Por otro lado, podemos ver que las funciones para detectar valores especiales, y en general **la gran mayoría de las funciones** en Octave, **funcionan** no sólo para valores escalares, sino que también para **vectores y matrices**.