Aritmética finita y análisis de error

Contenidos

Sistemas decimal y binario

En el **sistema decimal** el número 107.625 significa:

$$107.625 = 1 \cdot 10^2 + 7 \cdot 10^0 + 6 \cdot 10^{-1} + 2 \cdot 10^{-2} + 5 \cdot 10^{-3}$$

Es decir, es una suma cuyos términos son potencias de 10 y utiliza 10 dígitos $0, 1, \ldots, 9$.

Los ordenadores usan el **sistema binario** porque así los únicos dígitos que se almacenan son el 0 y el 1. En sistema binario los números se representan como sumas de potencias de 2:

$$(107.625)_{10} = 2^6 + 2^5 + 2^3 + 2^1 + 2^0 + 2^{-1} + 2^{-3} = (1101011.101)_2 \qquad (1)$$

CONVERSIÓN DECIMAL A BINARIO

Para convertir la parte entera, dividimos por 2 de forma repetida. Los restos de estas divisiones son los dígitos en base 2, de menos a más significativos.

Cocientes	107	53	26	13	6	3	1	0
Restos		1	1	0	1	0	1	1

El número $(107)_{10}$ en sistema binario es $(1101011)_2$.

Nota Algunas versiones de Matlab usan la función de2bi para realizar esta conversión.

Ejercicio 1 Escribir una función `de2bi_a` que transforme la parte entera de un número decimal a binario con `N` cifras. Aplicarla al número `x = 107.625` con `N = 8`.

NOTA: pueden ser útiles las siguientes funciones Matlab

- `fix(x)`: redondea `x` hacia `0`.
- `rem(x,n)`: da el resto de dividir `x` entre `n`.
- `zeros(1,m)`: crea una vector que contiene `m` ceros.

```
x=107.625;
N=8;
b=de2bi_a(x,N)
```

```
b =
     0     1     1     0     1     0     1     1
```

Para convertir a binario la parte decimal, multiplicamos por 2, le quitamos la parte entera, que será nuestro dígito binario y repetimos el proceso.

$$\begin{array}{lcccc} \textit{Decimal} & 0.625 & 0.25 & 0.5 & 0 \\ \textit{Entera} & & 1 & 0 & 1 \end{array}$$

Ejercicio 2 Escribir una función `de2bi_b` que transforme la parte decimal de un número en base 10 a base 2 y almacene un número dado de cifras. Aplicarla al número `x = 107.625` para obtener `N = 5` cifras binarias.

```
x=107.625;
N=5;
b=de2bi_b(x,N)
```

```
b =
     1     0     1     0     0
```

Ejercicio 3 Usar la función anterior para calcular la representación binaria de $(0.1)_{10}$ con 30 cifras decimales.

```
Ejercicio3
```

```
b =
  Columns 1 through 13
     0     0     0     1     1     0     0     1     1     0     0     1     1
  Columns 14 through 26
     0     0     1     1     0     0     1     1     0     0     1     1     0
  Columns 27 through 30
     0     1     1     0
```

Y tenemos que $(0.1)_{10} = (0.00011001100110011001100110011\ldots)_2$ y en binario es periódico. Por lo tanto si usamos un número finito de dígitos no lo podremos representar de forma exacta.

CONVERSIÓN BINARIO A DECIMAL

De acuerdo con (1) para convertir $(1101011.101)_2$ a base 10

```
1*(2^6)+1*(2^5)+0*(2^4)+1*(2^3)+0*(2^2)+1*(2^1)+1*(2^0)+1*(2^-1)+0*(1^-2)+1*(2^-3)
```

```
ans =
  107.6250
```

Representación de enteros

Si tenemos m dígitos o bits de memoria podemos almacenar 2^m números diferentes en binario.

ENTEROS POSITIVOS

Si sólo tomamos enteros positivos podríamos representar enteros con valores entre $(00\ldots00)_2 = (0)_{10}$ y $(11\ldots11)_2 = (2^m - 1)_{10}$. Por ejemplo, para $m = 3$ bits podríamos representar los enteros del 0 al 7.

$Número_{10}$	$Número_2$
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Ejercicio 4 Realizar un programa `Ejercicio4.m` que calcule cómo se almacenará $(80)_{10}$ si usamos representación binaria con una precisión de 8-bits sin signo.

```
Ejercicio4
```

```
b =
  Columns 1 through 6
           0           1           0           1           0           0
  Columns 7 through 8
           0           0
```

Es decir, con esta representación $(80)_{10} = (01010000)_2$

ENTEROS CON SIGNO

Si queremos representar enteros con signo reservaremos el primer bit para el signo y entonces tendremos espacio para $m - 1$ cifras significativas. Es decir $2^{m-1} - 1$ números positivos, otros tantos negativos y dos ceros, uno con signo

positivo y otro negativo. El rango de números representados sería de $[-2^{m-1}+1, 2^{m-1}-1]$. Si $m=3$ podremos representar números de -3 a 3.

$Número_{10}$	$Número_2$
-3	111
-2	110
-1	101
-0	100
$+0$	000
$+1$	001
$+2$	010
$+3$	011

El primer bit es el signo, 0 para los positivos y 1 para los negativos.

Ejemplo Calcular cómo se almacenará $(-80)_{10}$ si usamos representación binaria con una precisión de 8-bits con signo.

Solución: como es un número negativo, la representación es la misma que en el caso anterior pero el primer bit es el signo y como es negativo, es 1. Es decir, $(-80)_{10} = (11010000)_2$

ENTEROS CON SIGNO: REPRESENTACIÓN AL COMPLEMENTO DE DOS

Una forma de evitar la doble representación del cero es usar la representación al complemento de dos: los números negativos se construyen tomando los dígitos de los positivos y cambiando los 0 a 1 y los 1 a 0 y sumándole 1. De esta manera la suma de un número y su opuesto es siempre 0.

$Número_{10}$	$Número_2$
-4	100
-3	101
-2	110
-1	111
0	000
1	001
2	010
3	011

Ejemplo Para representar $(-2)_{10}$ empezamos convirtiendo $(2)_{10}$ a binario $(010)_2$

- invertimos sus dígitos $\rightarrow (101)_2$
- le sumamos 001 (tener en cuenta que en binario

$$0+0=0 \quad 0+1=1 \quad 1+1=10) \rightarrow (110)_2$$

Si sumamos $(010)_2 + (110)_2 = (000)_2$

En este caso, el primer bit de los negativos es 1 y el de los positivos es 0. Hemos representado enteros en el rango de $[-2^{m-1}, 2^{m-1}-1]$.

Ejemplo Calcular cómo se almacenará $(-80)_{10}$ si usamos representación binaria representación al complemento de dos con una precisión de 8-bits.

Solución: como $(80)_{10} = (01010000)_2$

- en el número anterior cambiamos los 0 por 1 y viceversa $\rightarrow 10101111$
- y sumándole 1 ya tenemos la representación $(-80)_{10} = (1011000)_2$

ENTEROS CON SIGNO: REPRESENTACIÓN SESGADA

Otra forma es la representación sesgada. Los números negativos se representan con valores positivos consecutivos, empezando por el menor negativo, y los positivos toman los valores siguientes. La representación de los números la obtenemos añadiendo el *sesgo* 2^{m-1} al número $x_r = x + 2^{m-1} \in [0, 2^m - 1]$.

$Número_{10}$	$Número_2$
-4	000
-3	001
-2	010
-1	011
0	100
1	101
2	110
3	111

Ejercicio 5 Calcular cómo se almacenará $(80)_{10}$ si usamos representación binaria representación sesgada con una precisión de 8-bits.

Nota: hay que sumarle el sesgo $x_r = x + 2^{m-1}$.

```
Ejercicio5
```

```
b =
     1     1     0     1     0     0     0     0
```

REPRESENTACIÓN DE ENTEROS EN MATLAB

Matlab, por defecto, almacena los datos numéricos como reales en punto flotante con doble precisión. Para usar un dato como entero hay que usar una función que lo convierta a entero (Mathworks-enteros). Matlab tiene 4 clases de enteros con signo y 4 sin signo, con 1, 2, 3 y 4 bytes de almacenamiento.

¿De cual de las dos útimas formas almacena Matlab los enteros? Depende. Almacena los enteros con la representación del *complemento al dos* y los exponentes de los números en punto flotante, que son enteros, con representación *sesgada*. ¿Por qué? Hay varios motivos. Uno de ellos es que los números con representación *sesgada* son más fáciles de comparar. Los códigos Matlab comparan frecuentemente números reales, y Matlab empieza comparando el exponente y sólo si son iguales sigue con la mantisa.

Ejercicio 6 Completar el cuadro del rango de valores para cada una de las clases anteriores.

Clase	Rango
Con signo-8 bits	$[-2^7, 2^7-1] = [-128, 127]$
Con signo-16 bits	$[-2^{15}, 2^{15}-1] = \cdots$
Con signo-32 bits	
Con signo-64 bits	
Sin signo-8 bits	$[0, 2^8-1] = \cdots$
Sin signo-16 bits	
Sin signo-32 bits	
Sin signo-64 bits	

(Solución en Mathworks-enteros)

Cuando intentamos convertir un real a entero y este es mayor que el admitido por el rango de la clase, matlab le da el valor máximo de la clase:

```
x=int8(300)
```

```
x =
  127
```

Análogamente con el mínimo

```
x=int8(-300)
```

```
x =
 -128
```

Representación de los números reales

En el caso de los números reales se utiliza una representación en **punto flotante** en **base** $\beta = 2$.

$$x_r = (-1)^s \cdot m \cdot \beta^e = (-1)^s \cdot (a_1.a_2 \ldots a_t) \cdot \beta^e$$

donde:

- s es el **signo**. Se almacena un 1 si el número es negativo y 0 si es positivo.
- m es la **mantisa** que si está *normalizada* tiene un valor $1 \leq m < \beta$ y el primer dígito ha de ser distinto de 0. Este dígito en base dos solo puede ser 1 y por lo tanto, en una mantisa normalizada siempre es $a_1 = 1$, no es necesario almacenarlo y ganamos un bit. Esto se llama *técnica del bit escondido*.
- e es el **exponente** que es un entero con signo y utiliza *representación binaria sesgada*.

Los números se almacenan en "palabras" de 32 bits (precisión sencilla), 64 bits (doble precisión), o incluso 128 bits (precisión cuádruple). La precisión por defecto en Matlab es la *doble precisión*. En el caso de *doble precisión* los bits se reparten:

- 1 bit para el signo.
- 52 bits para la mantisa.
- 11 bits para el exponente.

Si tenemos 11 bits para el exponente con signo, significa que tenemos espacio para $2^{11} = 2048$ números en formato binario, $0 < E < 2047$. El primer valor 00000000000 lo reservamos para cero y números desnormalizados y el último 11111111111 para `Inf` y `NaN`. Así que el exponente tomará valores $1 < E < 2046$. Y como el sesgo es 1023 estos valores representan a los exponentes $-1022 < E - 1023 < 1023$. Por lo tanto el exponente máximo es $E_{max} = 1023$, y el mínimo $E_{min} = -1022$. Si dividimos por el valor mínimo:

$$\frac{1}{x_{min}} = \frac{1}{m\beta^{E_{min}}} = \frac{1}{m\beta^{-1022}} = \frac{1}{m}\beta^{1022} < \frac{1}{m}\beta^{1023}$$

y no se supera el valor máximo (overflow).

Ejercicio 7 El mayor número normalizado que puede representar Matlab en doble precisión será, en representación binaria

$$(+1) \cdot (1.11 \ldots 11) \cdot 2^{1023}$$

Como habíamos dicho, el primer 1 no hace falta guardarlo y nos quedan 52 bits donde almacenamos los unos de $0.11\ldots11$. Por lo tanto, en decimal este número será:

$$(1+1\cdot 2^{-1}+1\cdot 2^{-2}+1\cdot 2^{-3}+\cdots+1\cdot 2^{-52})\cdot 2^{1023}$$

Hacer un programa `Ejercicio7` que calcule esta suma. Sumar los elementos de la serie de más pequeños a mayores (más adelante veremos por qué). Su valor tiene que coincidir con la constante de Matlab `realmax`

```
format long % Cambia a formato de 16 cifras
realmax     % Valor máximo para números con representación normalizada en
            % punto flotante
Ejercicio7  % El cálculo con nuestro programa
```

```
ans =
    1.797693134862316e+308
ans =
    1.797693134862316e+308
```

Ejercicio 8 El mayor entero consecutivo, utilizando el formato en punto flotante, del que podemos almacenar todos los dígitos significativos y que, por tanto, podemos almacenar de forma exacta en binario podría ser:

$$(+1)\cdot(1.11\ldots11)\cdot 2^{52}$$

Es decir, tenemos 53 cifras de precisión en binario. El entero siguiente:

$$(+1)\cdot(1.00\ldots00)\cdot 2^{53}$$

también podemos almacenarlo de forma exacta. Pero el siguiente a este ya no porque necesitaríamos un bit más a la derecha en la mantisa.

Hacer un programa `Ejercicio8.m` que calcule el mayor entero que podemos almacenar en punto flotante.

```
format rat    % Cambiamos a formato rat (racional) para ver mejor el entero
Ejercicio8    % Calculamos el entero
```

```
ans =
9007199254740992
```

Todos los enteros menores se pueden almacenar de forma exacta. Si contamos, vemos que tiene 16 dígitos. Por lo tanto, podemos almacenar todos los enteros con 15 dígitos decimales y casi todos los enteros con 16 dígitos de forma exacta.

Pero, podría decirse, el número

$$(+1)\cdot(1.11\ldots11)\cdot 2^{62}$$

es más grande y también es entero. Sin embargo sus 10 últimas cifras serían forzosamente 0 y desde el punto de vista de la precisión no cuentan porque no les podemos dar el valor que queramos.

Ejercicio 9 Hacer un razonamiento similar para calcular `realmin` y calcularlo.

$$(+1)\cdot(1.00\ldots00)\cdot 2^{-1022}$$

Su valor tiene que coincidir con la constante de Matlab `realmin`

```
format long % Volvemos a formato long (estábamos usando formato rat)
realmin     % Valor mínimo para números con representación normalizada en
            % punto flotante
Ejercicio9  % el cálculo con nuestro programa
```

```
ans =
    2.225073858507201e-308
ans =
    2.225073858507201e-308
```

¿Qué sucede con los valores de los exponentes fuera del rango $[-1022, 1023]$?

Si el exponente es menor que -1022 , el número es desnormalizado o cero y estamos usando para los bits correspondientes al exponente el valor especial 00000000000. Entonces ya no se asume que la mantisa es 1 sino 0.

```
x=0.5*2^-1023   % desnormalizado
1/x
```

```
x =
    5.562684646268003e-309
ans =
   Inf
```

```
2^-1080   % Underflow
```

```
ans =
     0
```

El menor valor desnormalizado es

$$(+1)(0.000\ldots01)\times 2^{-1022}$$

es decir, $2^{-1022-52}$

```
num_min=2^(-1022-52)
2^(-1022-53)          % Underflow
```

```
num_min =
    4.940656458412465e-324
ans =
     0
```

Si el exponente es mayor que 1023 devuelve infinito

```
2^1024
```

```
ans =
   Inf
```

```
-2^1025
```

```
ans =
  -Inf
```

PRECISIÓN DE LA MÁQUINA

Vamos a calcular el número más pequeño que se puede sumar a 1 utilizando la representación binaria en punto flotante con doble precisión.

1 en representación en punto flotante de doble precisión normalizada es

$$(+1) \cdot (1.00\ldots00) \cdot 2^0$$

con 52 ceros en la mantisa. El número más pequeño que se le puede sumar representado en punto flotante, pero no normalizado es:

$$(+1) \cdot (0.00\ldots01) \cdot 2^0$$

que pasado a decimal es

```
1*2^(-52)
```

```
ans =
     2.220446049250313e-16
```

o lo que es lo mismo

```
eps
```

```
ans =
     2.220446049250313e-16
```

Este valor, el menor número ϵ tal que $1 + \epsilon > 1$ se llama *precisión de la*

máquina y nos da la precisión de la representación en punto flotante.

Como, en doble precisión, $\texttt{eps} \approx 2.22 \cdot 10^{-16}$ se puede decir que corresponde, aproximadamente, a 16 cifras decimales.

Entre

$$1 = (+1) \cdot (1.00\ldots00) \cdot 2^0$$

y

$$1 + eps = (+1) \cdot (1.00\ldots01) \cdot 2^0$$

no podemos representar, de forma exacta y en punto flotante, ningún otro número real.

Si queremos el número más pequeño comparable con x

```
x=10;
eps(x)
```

```
ans =
     1.776356839400250e-15
```

```
x=100;
eps(x)
```

```
ans =
     1.421085471520200e-14
```

```
x=1000;
eps(x)
```

```
ans =
     1.136868377216160e-13
```

Y vemos que crece con el valor absoluto de x. Esto quiere decir, que la diferencia entre dos números consecutivos representables de forma exacta en punto flotante, aumenta conforme nos alejamos del cero. Por ello, la densidad de números representados en punto flotante no es uniforme. Es mayor cerca del cero y menor conforme nos alejamos. Recordar que el sistema en punto flotante sólo representa números racionales.

RESULTADOS ESPECIALES

Algunas operaciones dan resultados especiales

```
3/0
```

```
ans =
   Inf
```

```
-3/0
```

```
ans =
  -Inf
```

```
0/0
```

```
ans =
   NaN
```

`NaN` significa "Not a Number". Su aparición significa, en general, que algo ha ido mal con el programa y/o que se ha realizado una operación no válida.

Overflow. Ocurre cuando el resultado de la operación realizada es finita pero el resultado supera el rango de reales que pueden ser representados.

```
x=1e300
x^2
```

```
x =
    1.000000000000000e+300
ans =
   Inf
```

Underflow. Ocurre cuando el resultado de la operación es menor que el menor número desnormalizado que puede ser almacenado.

```
1/x^2
```

```
ans =
     0
```

Aproximación y error de redondeo

Empecemos con un juego

Descarga y ejecuta, tras borrar todas las variables, el programa rectángulo. Una pista sobre lo que sucede la obtendrás ejecutando el programa pista.

Redondeo

A menudo, para un número real x, no exite una representación exacta en punto flotante y cae entre dos números consecutivos $x^- < x < x^+$. Como representación de x elegiremos uno de los dos dependiendo del *método de redondeo* usado. IEEE reconoce 4 sistemas de redondeo:

1. Hacia arriba.
2. Hacia abajo.
3. Hacia cero ("truncamiento").
4. Hacia el más cercano.

Matlab utiliza este último método de redondeo.

PÉRDIDA DE DÍGITOS.

Supongamos que estamos usando un sistema de representación decimal que almacena 3 cifras significativas y queremos sumar los números $a = 123$ y $b = 1.25$. El resultado es $s = 124.25$ pero como solo podemos almacenar tres cifras redondeamos, por ejemplo, a $s_r = 124$. Hemos perdido cifras y estamos cometiendo un error. En general, cualquier operación aritmética conlleva errores de redondeo.

Ejemplo Si realizamos la operación $3 \times 0.1 \times 100$:

```
3*0.1*100
```

```
ans =
  30.000000000000004
```

El resultado no es 30 debido al error de redondeo. Estos se pueden propagar y afectar hasta invalidar los resultados. En el caso anterior el error se debe a que, como vimos al principio, el número $x = 0.1$ en base dos es periódico y por ello su representación en el ordenador no es exacta.

Ejemplo Si calculamos

$$\sum_{k=1}^{10000} 0.1$$

```
suma=0;
for i=1:10000
    suma=suma+0.1;
end
disp(suma)
```

```
     1.000000000000159e+03
```

y el error absoluto es

```
abs(suma-1000)
```

```
ans =
     1.588205122970976e-10
```

Ejemplo Si sumamos o restamos dos números muy diferentes en magnitud se pierde exactitud debido a los errores de redondeo. Por ejemplo

```
a=1e+9;
epsilon=1e-8;
suma=a;
for n=1:10000
    suma=suma+epsilon;
end
suma
```

```
suma =
     1.000000000000000e+09
```

Sin embargo el resultado debería haber sido

```
suma=a+10000*epsilon
```

```
suma =
     1.000000000000100e+09
```

Ejemplo Vamos a sumar los N primeros términos de la serie harmónica

$$\sum_{n=1}^{N} \frac{1}{n}$$

Usaremos precisión sencilla para que el efecto sea más evidente. Podemos pasar un número de doble precisión a precisión sencilla usando el comando `single`.

El resultado teórico para 1000 términos sería

```
N=1000;
suma=single((psi(N+1)-psi(1)))
```

```
suma =
   7.4854708
```

Empezando la suma por el término $n = 1$

```
suma1=single(0.0);
for n=1:N
    suma1=suma1+single(1)/single(n);
end
suma1
error1=abs(suma1-suma)
```

```
suma1 =
   7.4854784
error1 =
   7.6293945e-06
```

Empezando por el término final

```
suma2=single(0.0);
for n=N:-1:1
    suma2=suma2+single(1)/single(n);
end
suma2
error2=abs(suma2-suma)
```

```
suma2 =
   7.4854717
error2 =
   9.5367432e-07
```

Es mayor el error en el caso primero porque los términos de la sucesión son cada vez más pequeños mientras que la suma es cada vez más grande. Por lo tanto, conforme avanzamos en la suma se producirá pérdida de dígitos por sumar números de magnitud muy distinta.

ERROR DE CANCELACIÓN

Aparece al restar números grandes de valor parecido. Por ejemplo

$$\sqrt{x^2+\epsilon^2}-x$$

```
x=1000000;ep=0.0001;
a=sqrt(x^2+ep)-x
```

```
a =
     1.164153218269348e-10
```

En este caso el error de calcelación se puede evitar utilizando una *expresión matemática equivalente*

$$\sqrt{x^2+\epsilon^2}-x=\frac{(\sqrt{x^2+\epsilon^2}-x)(\sqrt{x^2+\epsilon^2}+x)}{\sqrt{x^2+\epsilon^2}+x}=\frac{\epsilon^2}{\sqrt{x^2+\epsilon^2}+x}$$

```
b=ep/(sqrt(x^2+ep)+x)
```

```
b =
     5.000000000000000e-11
```

El error relativo cometido al calcularlo con la primera expresión es

```
Er=abs((a-b)/b)
```

```
Er =
   1.328306436538696
```

Este ejemplo muestra que *expresiones matemáticamente equivalentes no son, necesariamente, equivalentes desde el punto de vista computacional*.

Ejercicio 10 Al resolver la ecuación de segundo grado

$$x^2 + 10^8 x + 1 = 0$$

utilizando la fórmula

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

una de las soluciones es:

```
a=1;b=10^8;c=1;
x1=(-b+sqrt(b^2-4*a*c))/(2*a)
```

```
x1 =
    -7.450580596923828e-09
```

Pero al sustituir en el polinomio para calcular el residual

```
pol=a*x1^2+b*x1+c
```

```
pol =
   0.254941940307617
```

es relativamente grande. Encontrar una expresión matemática equivalente que dé una solución con menor residual y calcular de nuevo la solución y su residual. Escribir para ello un programa llamado `Ejercicio10.m`

```
Ejercicio10
```

```
x1 =
```

```
    -1.000000000000000e-08
pol =
     1.110223024625157e-16
```

Ejercicio 11 Al resolver la ecuación de segundo grado

$$x^2 - 10^8 x + 1 = 0$$

utilizando la fórmula

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

una de las soluciones es:

```
a=1;b=-10^8;c=1;
x2=(-b-sqrt(b^2-4*a*c))/(2*a)
```

```
x2 =
     7.450580596923828e-09
```

Pero al sustituir en el polinomio

```
pol=a*x2^2+b*x2+c
```

```
pol =
   0.254941940307617
```

es relativamente grande. Encontrar una expresión matemática equivalente que dé una solución con menor residual y calcular de nuevo la solución y su residual. Escribir para ello un programa llamado `Ejercicio11.m`

```
Ejercicio11
```

```
x2 =
     1.000000000000000e-08
pol =
     1.110223024625157e-16
```

Ejemplo Teniendo en cuenta que

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

calcular el número de términos necesarios para calcular e^1 con el menor error posible.

El menor número comparable con e^1 es

```
Ea=eps(exp(1))
```

```
Ea =
     4.440892098500626e-16
```

Y el número de iteraciones necesarias es

```
n=0;
suma=1;
%
itermax=100;
error=abs(exp(1)-suma);
%
while( error > Ea && n < itermax )
    n=n+1;
    suma=suma+1/factorial(n);
    error=abs(exp(1)-suma);
end
n
error
```

```
n =
    17
error =
     0
```

Y aunque sabemos que con 17 iteraciones todavía cometemos error, este ya no es apreciable por la máquina.

Ejercicio 12 Teniendo en cuenta que

$$\cos x = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!}$$

calcular el número de términos necesarios para calcular $\cos 30^o$ con el menor error posible (recordar que los argumentos de funciones trigonométricas han de darse en radianes).

```
Ejercicio12
```

```
n =
     7
error =
     1.110223024625157e-16
```

Referencias

- Michael L. Overton, 1996. Floating point representation.
- B. Parhami, 2001. Number Representation and Computer Arithmetic.
- Adrian Sandu, 2008. Computer Representation of Numbers and Computer Arithmetic.

Published with MATLAB® 7.14