ANSI/IEEE Standard 754-1985

Matlab use floating-point arithmetic, which involves a finite set of numbers with finite precision. This leads to the phenomena of roundoff, underflow, and overflow.

Most nonzero numbers are normalized

binary64 double precision

$x = \pm(1.f)2^e$

±: **1** sign bit

1.*f* = 1 + *f*   (0 <= *f* < 1)

f: **52** significand (mantissa) bits   (+ 1 that is implicit)

$f = \frac{(\text{integer } < 2^{52})}{2^{52}}$

While the exponent *e* can be positive or negative, in binary formats it is stored as an unsigned number *be* that has a fixed "bias" added to it (to avoid storing a sign bit for the exponent). The sign of *e* is accommodated by storing *be*=*e*+1023.

be: **11** exponent bits (0 <= *be* <=$2^{11} - 1$= 2047)

e = *be* - 1023   (-1023 <= *e* <= 1024)

$x = \pm(1.f)2^{\text{be}-1023}$

The 2 extreme values for the exponent field *be*, 0 and 2047, are reserved for exceptional floating-point numbers.

Values of all 0s in this field are reserved for zero (*f*=0) and subnormal floating point numbers (*f*>0).

Values of all 1s are reserved for Inf (f=0) and NaN (f>0).

```
% 64 bits: s be1be2 ... be11 f1f2 ... f52
% s / be = 3 caracteres hexadecimales (12 bits)
format hex
unoHex = 1      % s=0, be= 0 followed by 10 ones(e=0), and f=0
```

```
unoHex =
   3ff0000000000000
```

```
dosHex = 2
```

```
dosHex =
   4000000000000000
```

```
infHex = Inf    % s=0, be=11 ones, and f=0
```

```
infHex =
   7ff0000000000000
```

```
nanHex = NaN    % s=1, be=11 ones, and f>0
```

```
nanHex =
   fff8000000000000
```

```
ceroHex = 0      % s=0, be=0, and f=0
```

```
ceroHex =
   0000000000000000
```

```
ceroHex =-0
```

```
ceroHex =
   8000000000000000
```

The finiteness of *e* is a limitation on range.

The finiteness of *f* is a limitation on precision.

Floating point numbers have a maximum,a minimum, and discrete spacing.

Any numbers that don't meet these limitations must be approximated by ones that do.

Finite Range

```
format long
maxBE = (2^11)-1               % 2047
```

```
maxBE =
        2047
```

```
exponentBias = (2^10)-1        % 1023, e = be - exponentBias
```

```
exponentBias =
        1023
```

```
minExponente = 1 - exponentBias % -1022, 0 is reserved
```

```
minExponente =
       -1022
```

```
maxExponente = (maxBE - 1) - exponentBias      % 1023, 2047 is reserved
```

```
maxExponente =
        1023
```

```
minF = 0
```

```
minF =
     0
```

```
maxF = 1-2^-52
```

```
maxF =
   1.000000000000000
```

¿Cuál es el menor número positivo normalizado?

```
% Smallest value
%1.f*2^(minExponente)
```

```
%el minimo valor de f sería f=0x00
smallestValue=(1+minF) * 2^(minExponente)
```

```
smallestValue =
    2.225073858507201e-308
```

¿Cuál es el mayor número positivo?

```
% Largest value
largestValue=(1+maxF) *2^(maxExponente)
```

```
largestValue =
    1.797693134862316e+308
```

Matlab calls this numbers realmin and realmax

```
minMatlab = realmin                     % normalized
```

```
minMatlab =
    2.225073858507201e-308
```

```
maxMatlab = realmax
```

```
maxMatlab =
    1.797693134862316e+308
```

If any computation tries to produce a value larger than realmax, it is said to **overflow**. The result is an exceptional floating-point value called infinity or Inf. It is represented by taking e=2047 and f=0 and satisfies relations like 1/Inf = 0 and Inf+Inf = Inf.

If any computation tries to produce a value that is undefined even in the real number system, the result is an exceptional value known as Not-a-Number, or NaN. Examples include 0/0 and Inf-Inf. NaN is represented by taking e=2047 and f nonzero.

Denormal numbers

If any computation tries to produce a value smaller than realmin, it is said to **underflow**.

Matlab allows exceptional denormal or subnormal floating-point numbers in the interval (realmin , eps*realmin]. Denormal numbers are represented by taking *be*=0 (and *f*>0).

```
menorRealmin = 1/realmax % < realmin
```

```
menorRealmin =
    5.562684646268003e-309
```

```
% 64 bits: s e1e2 ... e11 f1f2 ... f52
% s / be = 3 caracteres hexadecimales (12 bits)
format hex
menorRealmin     % s=0, be=0, and f>0=0100 12 zeroes => 0.25 × 2^-1022
```

```
menorRealmin =
   0004000000000000
```

The special exponent 0, meaning be1be2 . . . be11 = 000 0000 0000, denotes a departure from the standard floating point form. In this case the machine number is interpreted as the non-normalized floating point number ±0.b1b2 . . . b52 × 2^{-1022}. Note that the left-most bit is no longer assumed to be 1.

1022 is the smallest (normalized) exponent (1 - exponentBias).

Smallest normalized positive value = +1.0000 . . . 0000 × 2^{-1022}

Largest positive denormal value =   +0.1111 ... 1111 × 2^{-1022}

Smallest positive denormal value =   +0.0000 . . . 0001 × 2^{-1022}

```
format long
realmin
```

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

```
largestDenormal = hex2num('000fffffffffffff')   % ver num2hex
```

```
largestDenormal =
    2.225073858507201e-308
```

```
realmin - largestDenormal
```

```
ans =
    4.940656458412465e-324
```

```
menorRealmin
```

```
menorRealmin =
    5.562684646268003e-309
```

```
% Smallest non zero
% 0 00000000000 0000000000000000000000000000000000000000000000000001
smallestNonZero = (2^-52)*(2^-1022)  % 2^-1074
```

```
smallestNonZero =
    4.940656458412465e-324
```

```
eps*realmin
```

```
ans =
    4.940656458412465e-324
```

Double precision numbers below 2^{-1074} cannot be represented at all. Any results smaller than this are set to 0.

Many numbers below machine epsilon are machine representable, even though adding them to 1 may have no effect.

Finite Precision

Machine epsilon is the smallest number that, when added to one (1.0), yields a result different from one. It is the distance from 1 to the next larger floating point number

```
% eps(1)
epsilon=1;
while 1+epsilon~=1
    epsilon=epsilon/2;
end
epsilon=epsilon*2
```

```
epsilon =
    2.220446049250313e-16
```

Matlab epsilon equals the value of the unit in the last place relative to 1 (2^{-52})

```
epsBits = 2^(-52);
epsMatlab = eps(1);   % epsilon eps(1.0)
```

The distance between two adjacent floating-point numbers is not constant, but it is smaller for smaller values, and larger for larger values.

```
% eps(2)
epsilon2=1;
while 2+epsilon2~=2
    epsilon2=epsilon2/2;
end
epsilon2=epsilon2*2
```

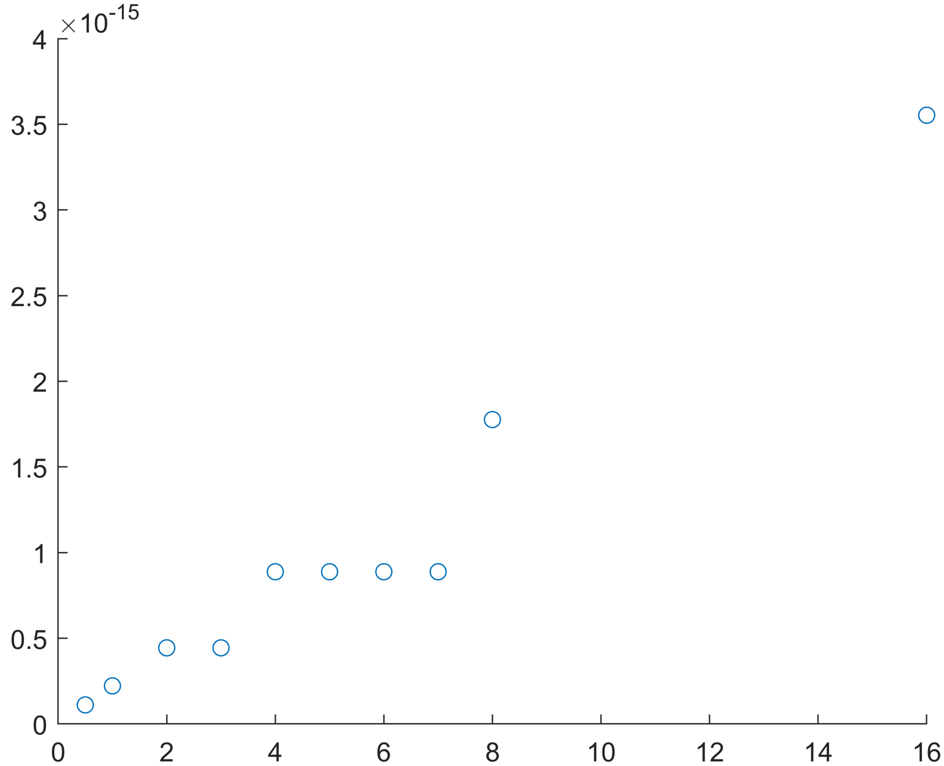
```
epsilon2 =
    4.440892098500626e-16
```

Within each binary interval 2^e <= x < 2^{e+1}, the numbers are equally spaced with an increment of 2^{e-52}.

The spacing changes at the numbers that are perfect powers of 2; the spacing on the side of larger magnitude is 2 times larger than the spacing on the side of smaller magnitude. The distribution in each binary interval is the same.

Usa la función eps(x) para desplegar el espaciamiento entre números para x=1/2,1,2,3,4,5,6,7,8,16

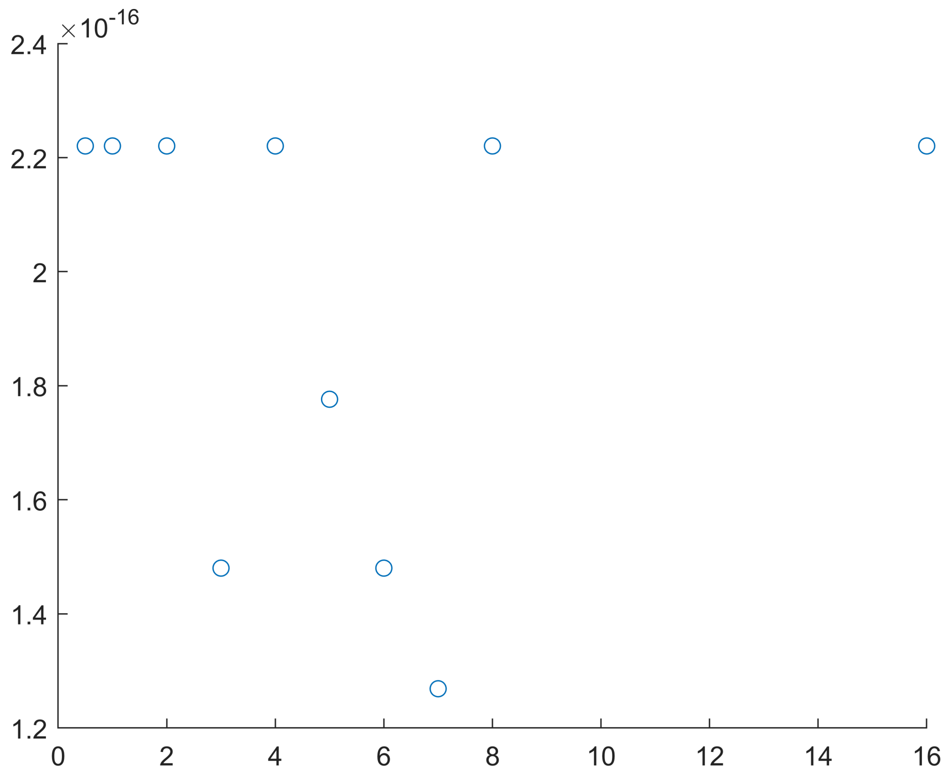
```
% espaciamiento
%Por la gráfica desplegada podemos observar que la distancia entre los
%números cambia cada potencia de 2. Por eso, no observamos un cambio en la
%distancia para las x=2,3 y para x=4,5,6,7
x=[1/2,1,2,3,4,5,6,7,8,16];
y=eps(x);
scatter(x,y);
```



Machine epsilon is an upper bound in the relative error (floating-point relative accuracy).

Calcula el espaciamiento relativo eps(x)/x para x=1/2,1,2,3,4,5,6,7,8,16

```
% espaciamiento relativo
%Podemos notar una discrepancia en el numero 3,5,6,7. Los numeros que no
%son potencia de 2. Era de esperarse sabiendo que las distancias se
%mantienen iguales en las no potencias de 2.
x=[1/2,1,2,3,4,5,6,7,8,16];
y=eps(x)./x;
scatter(x,y);
```



Rounding to the nearest at the last place.

Rounding to Nearest Value Rule - if the number falls midway, it is rounded to the nearest value with an even least significant digit.

The maximum relative error incurred when the result of an arithmetic operation is **rounded** to the nearest floating-point number is eps/2. The maximum relative spacing between numbers is eps. In either case, the roundoff level is about 16 decimal digits.

Floating-point arithmetic

There are various kinds of errors that we encounter when using a computer for computation.

- Truncation Error: Caused by adding up to a finite number of terms, while we should add infinitely many terms to get the exact answer in theory.
- Errors depending on the numerical algorithms, step size, and so on.
- Round-off: Caused by representing/storing numeric data in finite bits.
- Overflow/Underflow: Caused by too large or too small numbers to be represented/stored properly in finite bits—more specifically, the numbers having absolute values larger/smaller than the maximum (fmax)/ minimum(fmin) number that can be represented in MATLAB.
- Negligible Addition: Caused by adding two numbers of magnitudes differing by over 52 bits.
- Loss of Significance: Caused by a “bad subtraction,” which means a subtraction of a number from another one that is almost equal in value.

- Error Magnification: Caused and magnified/propagated by multiplying/dividing a number containing a small error by a large/small number.

It is important to realize that computer arithmetic, because of the truncation and rounding that it carries out, can sometimes give surprising results.

Round-Off or What You Get Is Not What You Expect

Calcula 1 - 3*(4/3 - 1)

```
format long
%Yo apostaría a que el error viene de querer representar numeros con
%decimales infinitos como 1/3 y realizar operaciones con ellos, se espera
%tener un "error" en el resultado
res=1 - 3*(4/3 - 1)
```

```
res =
     2.220446049250313e-16
```

```
resEsperado= 0
```

```
resEsperado =
     0
```

```
diferencia=abs(res-resEsperado)
```

```
diferencia =
     2.220446049250313e-16
```

Suma 10 veces 0.1

```
% De nuevo, me parece que el unico error que se me ocurriría es cómo se
% representa el 0.1 en el formato que maneja Matlab. En efecto, el valor de
% 0.1 en hexa es 3fb999999999999a, no es el valor exacto de 0.1 sino una
% aproximación.

res=0;
for i=1:10
    res=res+0.1;
end
res
```

```
res =
   1.000000000000000
```

```
resEsperado=1
```

```
resEsperado =
     1
```

```
diferencia=abs(res-resEsperado)
```

```
diferencia =
     1.110223024625157e-16
```

¿A qué es igual 7/100*100 - 7?

```
% La representación de fracciones en el formato de matlab parece tener un
% error de aproximación mínimo
res=7/100*100 - 7
```

```
res =
     8.881784197001252e-16
```

```
resEsperado=0
```

```
resEsperado =
     0
```

```
diferencia=abs(res-resEsperado)
```

```
diferencia =
     8.881784197001252e-16
```

Calcula el seno de π

```
% La representación del numero irracional pi, muestra un problema a la hora
% de pasarse a binario. El calculo del seno se hace mediante cálculos
% numericos desde matlab. Es imposible llegar a 0 si no se tiene con
% exactitud el valor de pi.
res=sin(pi)
```

```
res =
     1.224646799147353e-16
```

```
resEsperado=0
```

```
resEsperado =
     0
```

```
diferencia=abs(res-resEsperado)
```

```
diferencia =
     1.224646799147353e-16
```

¿Importa el orden de las operaciones?

1e-16 + 1 - 1e-16 == 1e-16 - 1e-16 + 1

```
% Importa porque Matlab realiza la operación, hace su aproximación con el
% formato y luego realiza la siguiente operación. Por eso hay una
% discrepancia en el resultado cuando no debería tenerlo.
res1=1e-16 + 1 - 1e-16
```

```
res1 =
   1.000000000000000
```

```
res2=1e-16 - 1e-16 + 1
```

```
res2 =
     1
```

```
diferencia=abs(res1-res2)
```

```
diferencia =
     1.110223024625157e-16
```

Representación en bits

```
% https://www.h-schmidt.net/FloatConverter/IEEE754.html

% Which familiar real numbers are approximated by floating-point numbers
% that display the following values with format hex?


% 4059000000000000
hex2num('4059000000000000')
```

```
ans =
   100
```

```
%010000000101100100000000000000000000000000000000
%signo positivo
%be=10000000101=1029
be=1029-1023;
%f=.1001=0.5625
f=0.5625;
res=+(1+f)*2^(be)
```

```
res =
   100
```

```



% 3f847ae147ae147b
hex2num('3f847ae147ae147b')
```

```
ans =
   0.010000000000000
```

```
%0011111110000100011110101110000101000111101011100001010001111011
%signo positivo
%be=01111111000=1016
be=1016-1023;
%f=.0100011110101110000101000111101011100001010001111011=0.28000000000000002665
f=0.28000000000000002665;
res=+(1+f)*2^(be)
```

```
res =
   0.010000000000000
```

```
% Let F be the set of all IEEE double-precision floating-point numbers,
% except NaNs and Infs, which have biased exponent 7ff (hex),
% and denormals, which have biased exponent 000 (hex).
% How many elements are there in F?
```

```
%Cada binario dentro del formato double64 representa un número, por lo
%tanto necesitamos saber cuántos números representamos con 64 bits y
%restarle los valores de NaN, Inf y subnormales

res=2^64-1 %Todos los valores posibles con 64 bits
```

```
res =
     1.844674407370955e+19
```

```
%infHex = Inf    % s=0, be=11 ones, and f=0
%nanHex = NaN    % s=1, be=11 ones, and f>0
%Inf es solo un valor, pero nan tiene 52 bits para definirse, todos los de
%f, entonces NaN serían 2^52-1
%respecto a los subnormales, son los numeros que empiezan con 000 base 16,
%entonces serían igual 2^52-1 numeros en total subnormales.

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

```
res =
     1.843773687445481e+19
```