# ANSI/IEEE Standard 754

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.

In 1985, the IEEE Standards Board and the American National Standards Institute adopted the ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic.

**Table of Contents**



## Most nonzero numbers are normalized

IEEE 754 - 2008  binary64 double precision

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

±: **1** sign bit

1.*f* = 1 + *f*  $(0 \le f < 1)$

*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 $(0 \le be \le 2^{11} - 1 = 2047)$

*e* = *be* - 1023  $(-1023 \le e \le 1024)$

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

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

- Values of all 0s 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
```

```
unoPunto5Hex = 1.5
```

```
unoPunto5Hex =
   3ff8000000000000
```

```
unoPunto75Hex = 1.75
```

```
unoPunto75Hex =
   3ffc000000000000
```

```
diezHex = 10
```

```
diezHex =
   4024000000000000
```

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

```
maxExponente =
        1023
```

```
minF = 0
```

```
minF =
     0
```

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

```
maxF =
   1.000000000000000
```

The smallest positive normalized floating-point number has: *f*=0 and *e*=-1022.

```
% Smallest value = +1.0000 . . . 0000 × 2^-1022
minBits = (1+minF)*(2^minExponente)                % 1 * 2^(-1022)
```

```
minBits =
    2.225073858507201e-308
```

The largest floating-point number has: *f* a little less than 1 and *e* = 1023.

```
% Largest value = +1.1111 . . . 1111 × 2^(1023)
maxBits = (1+maxF)*(2^maxExponente)      % (2-eps)*2^1023
```

```
maxBits =
    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

```
epsCalculated = 1.0;
i=0;
while (1.0 + epsCalculated/2) ~= 1.0
    epsCalculated = epsCalculated/2;
    i=i+1;
end
epsCalculated       % eps(1) = eps
```

```
epsCalculated =
    2.220446049250313e-16
```

```
iteraciones=i;      % 52
diferente = 1+epsCalculated == 1;
igual = 1+epsCalculated/2==1;
```

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

```
epsBits = 2^(-52)
```

```
epsBits =
    2.220446049250313e-16
```

```
epsMatlab = eps   % epsilon eps(1.0)
```

```
epsMatlab =
    2.220446049250313e-16
```

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)
eps2Calculated = 2.0;
i=0;
while (2.0 + eps2Calculated/2) ~= 2.0
    eps2Calculated = eps2Calculated/2;
    i=i+1;
end
```

```
eps2Calculated       % eps(2) = 2*eps
```

```
eps2Calculated =
     4.440892098500626e-16
```

```
iteraciones=i       % 52
```

```
iteraciones =
    52
```

```
diferente = 2+eps2Calculated == 2
```

```
diferente = logical
   0
```

```
igual = 2+eps2Calculated/2==2
```

```
igual = logical
   1
```

Within each binary interval $2^e \leq x < 2^{e+1}$, 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.

```
% If e=0, the spacing of the numbers between 1 and 2 is 2^(-52).
spacing1 = 2^(0-52)    % 1=2^0
```

```
spacing1 =
     2.220446049250313e-16
```

```
eps1=eps(1.0)
```

```
eps1 =
     2.220446049250313e-16
```

```
spacing2 = 2^(1-52)    % 2=2^1
```

```
spacing2 =
     4.440892098500626e-16
```

```
eps2=eps(2)
```

```
eps2 =
     4.440892098500626e-16
```

```
spacing4 = 2^(2-52)    % 4=2^2
```

```
spacing4 =
     8.881784197001252e-16
```

```
eps4=eps(4)
```

```
eps4 =
     8.881784197001252e-16
```

```
spacing8 = 2^(3-52)  % = 2^2*2^(1-52) = 4*2^(1-52) = 4*eps(2)
```

```
spacing8 =
     1.776356839400250e-15
```

```
eps8=eps(8)
```

```
eps8 =
     1.776356839400250e-15
```

```
4*eps(2)
```

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

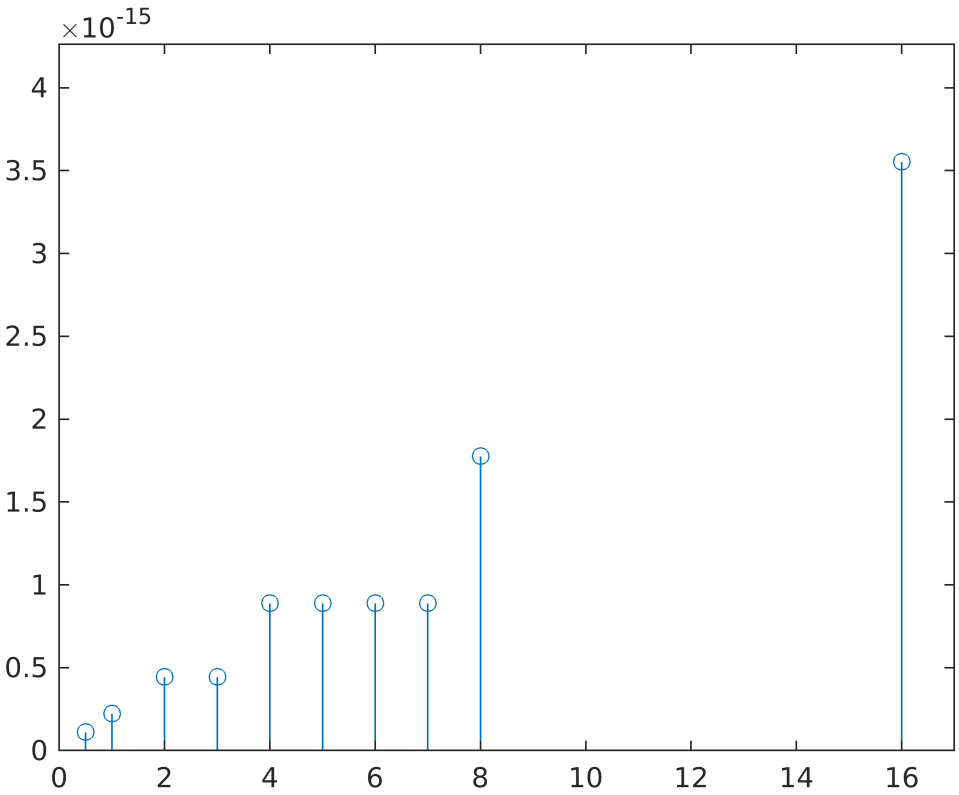
```
x=[1/2,1,2,3,4,5,6,7,8,16]
```

```
x = 1×10
   0.500000000000000   1.000000000000000   2.000000000000000   3.000000000000000 ···
```

```
spacing=eps(x)
```

```
spacing = 1×10
10^-14 ×
   0.011102230246252   0.022204460492503   0.044408920985006   0.044408920985006 ···
```

```
stem(x,spacing);
axis([0,x(end)+1,0,1.2*eps(x(end))]);
```



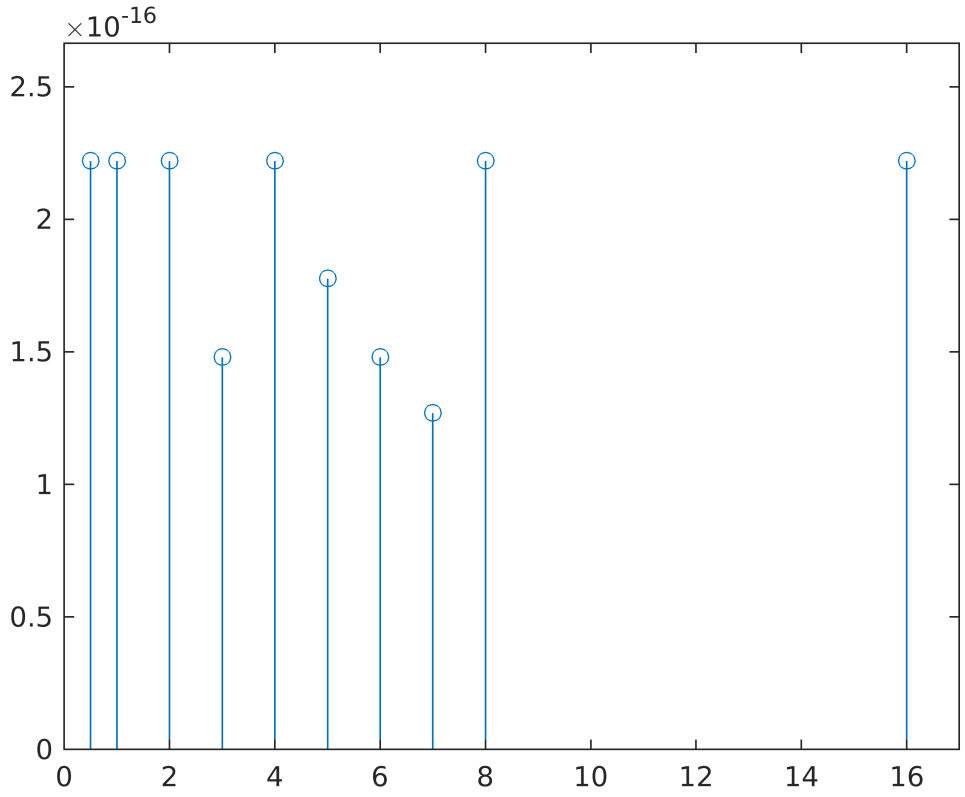

Machine epsilon is an upper bound in the relative error (floating-point relative accuracy).

```
% eps(1/2)/(1/2) = eps(1)/1 = eps(2)/2 = eps(2^n)/2^n = eps
```

```
% eps >= eps(n)/n
x=[1/2,1,2,3,4,5,6,7,8,16];
rSpacing=eps(x)./x
```

```
rSpacing = 1×10
10^-15 ×
   0.222044604925031   0.222044604925031   0.222044604925031   0.148029736616688 ···
```

```
stem(x,rSpacing);
axis([0,x(end)+1,0,1.2*eps(x(end))/x(end)]);
```




Find largest positive integer that is exact

```
lpie = 1;
i=0;
while eps(lpie) <= 1
    lpie = lpie*2;
    i=i+1;
end
i
```

```
i =
    53
```

```
format long
lpie
```

```
lpie =
```

```
     9.007199254740992e+15
```

```
log2(lpie)
```

```
ans =
    53
```

```
format hex
lpie
```

```
lpie =
   4340000000000000
```

```
format long
lpie = 1.0*2^53        % 53 = 4*16^2+3*16+4 - 1023
```

```
lpie =
     9.007199254740992e+15
```

```
eps(lpie)
```

```
ans =
     2
```

```
lpie-1
```

```
ans =
     9.007199254740991e+15
```

```
lpie == lpie-1
```

```
ans = logical
   0
```

```
spine= lpie+1  % smallest positive integer not exact
```

```
spine =
     9.007199254740992e+15
```

```
lpie == spine
```

```
ans = logical
   1
```

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.

**Example 1**

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

```
% The decimal number 4/3 is not exactly representable as a binary fraction
% For this reason, the following calculation does not give zero,
% but rather reveals the quantity eps(1).
eps1 = 1 - 3*(4/3 - 1)
```

```
eps1 =
     2.220446049250313e-16
```

```
% Similarly, 0.1 is not exactly representable as a binary number.
% 0.1 = 2^-4 * (1 + 9/16 + 9/16^2 + 9/16^3 + ...)
t=0.1;
format hex
t
```

```
t =
   3fb999999999999a
```

```
% The first three characters, 3fb, give the hexadecimal representation of
% decimal 1019, which is the biased exponent e+1023 if e is -4.
% The other 13 characters are the hexadecimal representation of the
% fraction f.
format long
% Thus, you get the following nonintuitive behavior:
a = 0.0;
for i = 1:10
  a = a + 0.1;
end
a
```

```
a =
   1.000000000000000
```

```
a = sum(0.1*ones(1,10))
```

```
a =
   1.000000000000000
```

```
isequal(a,1)   % 0
```

```
ans = logical
   0
```

```
s = [0:0.1:1];
isequal(s(11),1)
```

```
ans = logical
   1
```

```
num2hex(0.1) % 3fb9999999999a redondeo
```

```
ans =
'3fb999999999999a'
```

```
% 7/100
% How do you compare the absolute value of this answer with the resolution
% of the range to which 7 belongs?
res = 7/100*100 - 7
```

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

```
isequal(res,eps(7))  % eps(7)
```

```
ans = logical
   1
```

```
% If Matlab is asked to store 9.4, then subtract 9,and then subtract 0.4,
% the result will be something other than zero! What happens is that:
% First, 9.4 is stored as fl(9.4) = 9.4 + 0.2×2^-49. When 9 is subtracted
% (9 can be represented with no error), the result is 0.4 + 0.2×2^-49.
% Now, asking the computer to subtract 0.4 results in subtracting the
% machine number fl(0.4) = 0.4 + 0.1×2^-52, which will leave
% 0.2×2^-49 - 0.1×2^-52 = 0.1×2^-52(2^4 - 1) = 3 × 2^-53 instead of zero.
x=9.4;
format hex
x;
format
y=x-9;
z=y-0.4;
3*2^(-53);

cero = 1 - 0.9 - 0.1;           % 2.7756e-17
cero = 0.1 + 0.2 - 0.3;         % 5.5511e-17
cero = 0.2 - 0.3 + 0.1;         % 2.7756e-17

% Since pi is not really pi, it is not surprising that
% sin(pi) is not exactly zero:
```

```
e = sin(pi);                    % % 1.1102e-16

% There are gaps between floating-point numbers.
% As the numbers get larger, so do the gaps, as evidenced by:
d = (2^53 + 1) - 2^53   % gap > 1
```

```
d = 0
```

```
% while
d = (2^52 + 1) - 2^52
```

```
d = 1
```

**Example 2**

Catastrophic Cancellation (of leading digits). Loss of significance

```
% When subtractions are performed with nearly equal operands, sometimes
% cancellation can occur unexpectedly (or loss of significant digits).
% Cancellation can be explained by noting that if both x and y are
% accurate to k digits, and if they agree in the first kf digits,
% then their difference will contain only about k-kf significant digits,
% the first kf digits cancel each other out.
% This observation is the reason for the well-known adage of numerical
% computing - that one should avoid taking the difference of two similar
% numbers if at all possible.
% When we subtract two numbers that are nearly equal, the leading digits
% of the numbers cancel, leaving a result close to the rounding error.
% In other words, the rounding error dominates the difference.

% in order to prevent  loss of significance', it is important to avoid a 'bad subtracti
%  – that is, a subtraction of a number from another number having almost equal value.
% Let us consider a simple problem of finding the roots of a second-order equation
% ax2 + bx + c = 0 by using the quadratic formula.
% Let |4ac| < b2. Then, depending on the sign of b, a "bad subtraction" may be
% encountered when we try to find x1 or x2, which is the smaller one of the two
% roots. This implies that it is safe from the "loss of significance" to compute the
% root having the larger absolute value first and then obtain the other root by using
% the relation (between the roots and the coefficients) x1x2 = c/a.

% Adding a large and a small number
% Cuando se suman dos números, la mantisa del menor se modifica de tal
% manera que los exponentes sean los mismos. Esto tiene el efecto de
% alinear los puntos decimales.
% The following is an example of a cancellation caused by swamping
% (loss of precision that makes the addition insignificant).
s1 = 1e-16;
s = s1 + 1;         % 1
s2 = s - 1;
s1==s2              % 0
```

```
ans = logical
   0
```

```
% In the process of adding the two numbers, an alignment is made so that the
% two exponents in their 64-bit representations equal each other; and it will kick
```

```
% out the part smaller by more than 52 bits, causing some numerical error.
% For example, adding 2^−23 to 2^30 does not make any difference,
% while adding 2−^22 to 2^30 does.

% Note that the order of operations can matter in the computation:
b = 1e-16 + 1 - 1e-16;          % 1.0000
1e-16 + 1
```

```
ans = 1
```

```
c = 1e-16 - 1e-16 + 1;          % 1
isequal(b,c)
```

```
ans = logical
   0
```

```
c-b                            % 1.1102e-16
```

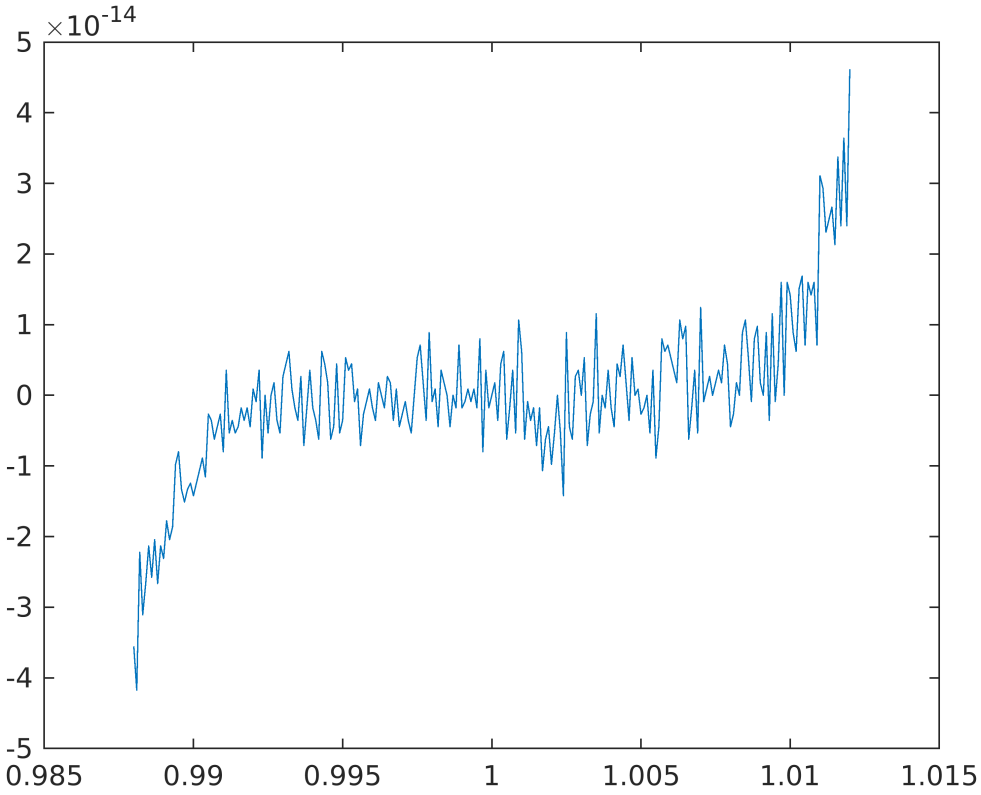
```
ans = 1.1102e-16
```

```
eps(0.5)
```

```
ans = 1.1102e-16
```

```


% Another example plots a seventh-degree polynomial.
% The resulting plot doesn't look anything like a polynomial. It isn't
% smooth. You are seeing roundoff error in action. The y-axis scale factor
% is tiny, 10^-14. The tiny values of y are being computed by taking sums
% and differences of numbers as large as 35 x (1.012)^4.
% There is severe subtractive cancellation.
x = 0.988:.0001:1.012;
y = x.^7-7*x.^6+21*x.^5-35*x.^4+35*x.^3-21*x.^2+7*x-1;
plot(x,y);
```




**Example 3**

Floating-Point Operations and Linear Algebra

```
% Round-off, cancellation, and other traits of floating-point arithmetic
% combine to produce startling computations when solving the problems of
% linear algebra.
% MATLAB warns that the following matrix A is ill-conditioned,
% and therefore the system Ax = b may be sensitive to small perturbations:
A = diag([2 eps]);
b = [2; eps];
y = A\b;
```

```
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =
1.110223e-16.
```

```
% A problem is said to be well conditioned if its solution is not affected
% greatly by small perturbations to the data that define the problem.
% Otherwise, it is said to be ill conditioned.

% Condition number of a matrix is the ratio of the largest singular value
% of that matrix to the smallest singular value.
% For a square matrix A, the square roots of the eigenvalues of A'A,
% where A' is the conjugate transpose, are called singular values
% The condition number of a matrix measures the sensitivity of the
% solution of a system of linear equations (x) to errors in the data (b).
```

```
% It gives an indication of the accuracy of the results from
% matrix inversion and the linear equation solution.
% Let e be the error in b. The error in the solution A^-1*b is A^-1*e.
% The ratio of the relative error in x to the relative error in b is
% norm(A^-1*b)/norm(A^-1*e) / norm(e)/norm(b)
% Returns an estimate for the reciprocal condition of A in 1-norm.
% If A is well conditioned, rcond(A) is near 1.0.
% If A is badly conditioned, rcond(A) is near 0.

C = rcond(A); % reciprocal condition number

% If the condition number of A is large, then the system A*x=b is
% ill-conditioned and a small residual r=b-A*x does not imply that
% x is close to the exact solution
```

## Ejercicios

```
% https://www.h-schmidt.net/FloatConverter/IEEE754.html
% signo e f (1 11 52)
% 3ff in base 16 is 3x16^2 + 15x16 + 15 = 1023 in decimal. So e=0.
% Any floating-point number between 1.0 and 2.0 has e=0, so its hex output
% begins with 3ff. 1 = 3ff0000000000000
% First 3 hex = 400 => be=1024, e=1, numbers between 2.0 and 4.0

% Which familiar real numbers are approximated by floating-point numbers
% that display the following values with format hex?
% 4059000000000000
% 3f847ae147ae147b
% 3fe921fb54442d18

% 4059000000000000
hex2num('4059000000000000')
```

```
ans = 100
```

```
% The first hex digit, 4, is 0100 in binary. The first bit is the sign of
% the floating-point number; 0 is positive, 1 is negative.
% So the number is positive.
signo = 1;
% The remaining bits of the first three hex digits contain be=e+1023.
% be=405 in base 16 is 4x16^2 + 5 = 1029 in decimal.
coeficientes = [4,0,5]
```

```
coeficientes = 1×3
     4     0     5
```

```
exponentes = [2,1,0]
```

```
exponentes = 1×3
     2     1     0
```

```
16.^exponentes
```

```
ans = 1×3
   256    16     1
```

```
coeficientes.*(16.^exponentes)
```

```
ans = 1×3
        1024           0           5
```

```
be = sum(coeficientes.*(16.^exponentes))
```

```
be = 1029
```

```
e = be - 1023
```

```
e = 6
```

```
% So e=6 and 2^6=64 and 64<x<128
% The other 13 hex digits contain f
% f=9000000000000
coeficientes = [9,0,0,0,0,0,0,0,0,0,0,0,0]
```

```
coeficientes = 1×13
     9     0     0     0     0     0     0     0     0     0     0     0     0
```

```
exponentes = 1:1:13
```

```
exponentes = 1×13
     1     2     3     4     5     6     7     8     9    10    11    12    13
```

```
16.^exponentes
```

```
ans = 1×13
10^15 ×
    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000 ···
```

```
f = sum(coeficientes.*(1./(16.^exponentes)))
```

```
f = 0.5625
```

```
uno = signo*(1+f)*2^e          % 100 decimal
```

```
uno = 100
```

```
uno = hex2num('4059000000000000')
```

```
uno = 100
```

```
format hex
dos=1/100
```

```
dos =
   3f847ae147ae147b
```

```
format long

% 3fe921fb54442d18
% The first hex digit, 3, is 0011 in binary. The first bit is the sign of
% the floating-point number; 0 is positive, 1 is negative.
% So the number is positive.
% The remaining bits of the first three hex digits contain e+1023.
```

```
% In this example, 3fe in base 16 is 3x16^2+15x16+14 = 768+240+14 =
% 1022 in decimal. So e=-1 and 2^-1=0.5 and 0.5<x<1
% The other 13 hex digits contain f = 921fb54442d18
tresH = '3fe921fb54442d18';
tresHv = tresH';
sbeH = tresHv(1:3);
sbeD = hex2dec(sbeH);
fH = tresHv(4:16);
fD = hex2dec(fH);
i=2:-1:0;
sbe = sum(sbeD'.*16.^i);
negativo = 8*16^2;
if sbe >= negativo
    signo = -1;
    be = sbe - negativo;
else
    signo = 1;
    be = sbe;
end;
bias = 2^10-1
```

```
bias =
        1023
```

```
e = be - bias
```

```
e =
    -1
```

```
i=1:13;
f=sum(fD'.*16.^-i)
```

```
f =
   0.570796326794897
```

```
tres=(2^e)*(1+f);
format hex
pi/4
```

```
ans =
   3fe921fb54442d18
```

```
format long
tres
```

```
tres =
   0.785398163397448
```

```
% 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).
% (a) How many elements are there in F?
2*(2^11-2)*2^52 + 1   % including negatives and 1 zero
```

```
ans =
     1.842872967520007e+19
```

```
% (b) What fraction of the elements of F are in the interval 1<=x<2?
% (c) What fraction of the elements of F are in the interval 1/64<=x<1/32?
1/(2^11-2)
```

```
ans =
     4.887585532746823e-04
```

```
% Use the classic formula in Matlab to compute both roots for
a=1;
b=-100000000;
c=1;
format long
x1=(-b+sqrt(b^2-4*a*c))/(2*a);
x2=(-b-sqrt(b^2-4*a*c))/(2*a);
roots([a b c]);
% You should find that the classic formula is good for computing one root,
% but not the other. So use it to compute one root accurately and then
% use the fact that x1x2 = c/a to compute the other.
```

```
% Problems Clever 1.34
% x=1; while 1+x > 1, x=x/2, pause(.02), end    % 2*eps
% x=1; while x+x > x, x=x/2, pause(.02), end    % 0
% x=1; while x+x > x, x=x*2, pause(.02), end    % Inf
```