

Floating-Point Numbers

On this page...

Double-Precision Floating Point

Single-Precision Floating Point

Creating Floating-Point Data

Arithmetic Operations on Floating-Point Numbers

Largest and Smallest Values for Floating-Point Classes

Accuracy of Floating-Point Data

Avoiding Common Problems with Floating-Point Arithmetic

Floating-Point Functions

References

MATLAB® represents floating-point numbers in either double-precision or single-precision format. The default is double precision, but you can make any number single precision with a simple conversion function.

Double-Precision Floating Point

MATLAB constructs the double-precision (or `double`) data type according to IEEE® Standard 754 for double precision. Any value stored as a `double` requires 64 bits, formatted as shown in the table below:

Bits	Usage
63	Sign (0 = positive, 1 = negative)
62 to 52	Exponent, biased by 1023
51 to 0	Fraction f of the number $1.f$

Single-Precision Floating Point

MATLAB constructs the single-precision (or `single`) data type according to IEEE Standard 754 for single precision. Any value stored as a `single` requires 32 bits, formatted as shown in the table below:

Bits	Usage
31	Sign (0 = positive, 1 = negative)
30 to 23	Exponent, biased by 127
22 to 0	Fraction f of the number $1.f$

Because MATLAB stores numbers of type `single` using 32 bits, they require less memory than numbers of type `double`, which use 64 bits. However, because they are stored with fewer bits, numbers of type `single` are represented to less precision than numbers of type `double`.

Creating Floating-Point Data

Use double-precision to store values greater than approximately 3.4×10^{38} or less than approximately -3.4×10^{38} . For numbers that lie between these two limits, you can use either double- or single-precision, but single requires less memory.

Creating Double-Precision Data

Because the default numeric type for MATLAB is `double`, you can create a `double` with a simple assignment statement:

```
x = 25.783;
```

The `whos` function shows that MATLAB has created a 1-by-1 array of type `double` for the value you just stored in `x`:

```
whos x
      Name      Size      Bytes  Class
      x         1x1         8      double
```

Use `isfloat` if you just want to verify that `x` is a floating-point number. This function returns logical 1 (`true`) if the input is a floating-point number, and logical 0 (`false`) otherwise:

```
isfloat(x)
ans =
     1
```

You can convert other numeric data, characters or strings, and logical data to double precision using the MATLAB function, `double`. This example converts a signed integer to double-precision floating point:

```
y = int64(-589324077574);           % Create a 64-bit integer

x = double(y)                       % Convert to double
x =
    -5.8932e+11
```

Creating Single-Precision Data

Because MATLAB stores numeric data as a `double` by default, you need to use the `single` conversion function to create a single-precision number:

```
x = single(25.783);
```

The `whos` function returns the attributes of variable `x` in a structure. The `bytes` field of this structure shows that when `x` is stored as a single, it requires just 4 bytes compared with the 8 bytes to store it as a `double`:

```
xAttrib = whos('x');
xAttrib.bytes
ans =
     4
```

You can convert other numeric data, characters or strings, and logical data to single precision using the `single` function. This example converts a signed integer to single-precision floating point:

```
y = int64(-589324077574);           % Create a 64-bit integer

x = single(y)                       % Convert to single
x =
    -5.8932e+11
```

Arithmetic Operations on Floating-Point Numbers

This section describes which classes you can use in arithmetic operations with floating-point numbers.

Double-Precision Operations

You can perform basic arithmetic operations with `double` and any of the following other classes. When one or more operands is an integer (scalar or array), the `double` operand must be a scalar. The result is of type `double`, except where noted otherwise:

- `single` — The result is of type `single`
- `double`
- `int*` or `uint*` — The result has the same data type as the integer operand
- `char`
- `logical`

This example performs arithmetic on data of types `char` and `double`. The result is of type `double`:

```
c = 'uppercase' - 32;
```

```
class(c)
ans =
    double
```

```
char(c)
ans =
    UPPERCASE
```

Single-Precision Operations

You can perform basic arithmetic operations with `single` and any of the following other classes. The result is always `single`:

- `single`
- `double`
- `char`
- `logical`

In this example, `7.5` defaults to type `double`, and the result is of type `single`:

```
x = single([1.32 3.47 5.28]) .* 7.5;
```

```
class(x)
ans =
    single
```

Largest and Smallest Values for Floating-Point Classes

For the `double` and `single` classes, there is a largest and smallest number that you can represent with that type.

Largest and Smallest Double-Precision Values

The MATLAB functions `realmax` and `realmin` return the maximum and minimum values that you can represent with

the `double` data type:

```
str = 'The range for double is:\n\t%g to %g and\n\t %g to %g';  
sprintf(str, -realmax, -realmin, realmin, realmax)
```

```
ans =
```

```
The range for double is:
```

```
-1.79769e+308 to -2.22507e-308 and  
2.22507e-308 to 1.79769e+308
```

Numbers larger than `realmax` or smaller than `-realmax` are assigned the values of positive and negative infinity, respectively:

```
realmax + .0001e+308
```

```
ans =
```

```
Inf
```

```
-realmax - .0001e+308
```

```
ans =
```

```
-Inf
```

Largest and Smallest Single-Precision Values

The MATLAB functions `realmax` and `realmin`, when called with the argument `'single'`, return the maximum and minimum values that you can represent with the `single` data type:

```
str = 'The range for single is:\n\t%g to %g and\n\t %g to %g';  
sprintf(str, -realmax('single'), -realmin('single'), ...  
        realmin('single'), realmax('single'))
```

```
ans =
```

```
The range for single is:
```

```
-3.40282e+38 to -1.17549e-38 and  
1.17549e-38 to 3.40282e+38
```

Numbers larger than `realmax('single')` or smaller than `-realmax('single')` are assigned the values of positive and negative infinity, respectively:

```
realmax('single') + .0001e+038
```

```
ans =
```

```
Inf
```

```
-realmax('single') - .0001e+038
```

```
ans =
```

```
-Inf
```

Accuracy of Floating-Point Data

If the result of a floating-point arithmetic computation is not as precise as you had expected, it is likely caused by the limitations of your computer's hardware. Probably, your result was a little less exact because the hardware had

insufficient bits to represent the result with perfect accuracy; therefore, it truncated the resulting value.

Double-Precision Accuracy

Because there are only a finite number of double-precision numbers, you cannot represent all numbers in double-precision storage. On any computer, there is a small gap between each double-precision number and the next larger double-precision number. You can determine the size of this gap, which limits the precision of your results, using the `eps` function. For example, to find the distance between 5 and the next larger double-precision number, enter

```
format long

eps(5)
ans =
    8.881784197001252e-16
```

This tells you that there are no double-precision numbers between 5 and $5 + \text{eps}(5)$. If a double-precision computation returns the answer 5, the result is only accurate to within $\text{eps}(5)$.

The value of $\text{eps}(x)$ depends on x . This example shows that, as x gets larger, so does $\text{eps}(x)$:

```
eps(50)
ans =
    7.105427357601002e-15
```

If you enter `eps` with no input argument, MATLAB returns the value of $\text{eps}(1)$, the distance from 1 to the next larger double-precision number.

Single-Precision Accuracy

Similarly, there are gaps between any two single-precision numbers. If x has type `single`, $\text{eps}(x)$ returns the distance between x and the next larger single-precision number. For example,

```
x = single(5);
eps(x)
```

returns

```
ans =
    4.7684e-07
```

Note that this result is larger than $\text{eps}(5)$. Because there are fewer single-precision numbers than double-precision numbers, the gaps between the single-precision numbers are larger than the gaps between double-precision numbers. This means that results in single-precision arithmetic are less precise than in double-precision arithmetic.

For a number x of type `double`, $\text{eps}(\text{single}(x))$ gives you an upper bound for the amount that x is rounded when you convert it from `double` to `single`. For example, when you convert the double-precision number 3.14 to `single`, it is rounded by

```
double(single(3.14) - 3.14)
ans =
    1.0490e-07
```

The amount that 3.14 is rounded is less than

```
eps(single(3.14))
ans =
```

```
2.3842e-07
```

Avoiding Common Problems with Floating-Point Arithmetic

Almost all operations in MATLAB are performed in double-precision arithmetic conforming to the IEEE standard 754. Because computers only represent numbers to a finite precision (double precision calls for 52 mantissa bits), computations sometimes yield mathematically nonintuitive results. It is important to note that these results are not bugs in MATLAB.

Use the following examples to help you identify these cases:

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`.

```
e = 1 - 3*(4/3 - 1)
```

```
e =
```

```
2.2204e-16
```

Similarly, `0.1` is not exactly representable as a binary number. Thus, you get the following nonintuitive behavior:

```
a = 0.0;
for i = 1:10
    a = a + 0.1;
end
a == 1
```

```
ans =
    0
```

Note that the order of operations can matter in the computation:

```
b = 1e-16 + 1 - 1e-16;
c = 1e-16 - 1e-16 + 1;
b == c
```

```
ans =
    0
```

There are gaps between floating-point numbers. As the numbers get larger, so do the gaps, as evidenced by:

```
(2^53 + 1) - 2^53
```

```
ans =
    0
```

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

```
sin(pi)
```

```
ans =
```

```
1.224646799147353e-16
```

Example 2 — Catastrophic Cancellation

When subtractions are performed with nearly equal operands, sometimes cancellation can occur unexpectedly. The following is an example of a cancellation caused by swamping (loss of precision that makes the addition insignificant).

```
sqrt(1e-16 + 1) - 1
```

```
ans =  
    0
```

Some functions in MATLAB, such as `expm1` and `log1p`, may be used to compensate for the effects of catastrophic cancellation.

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.
```

These are only a few of the examples showing how IEEE floating-point arithmetic affects computations in MATLAB. Note that all computations performed in IEEE 754 arithmetic are affected, this includes applications written in C or FORTRAN, as well as MATLAB.

Floating-Point Functions

See **Floating-Point Functions** for a list of functions most commonly used with floating-point numbers in MATLAB.

References

The following references provide more information about floating-point arithmetic.

References

[1] Moler, Cleve, "Floating Points," *MATLAB News and Notes*, Fall, 1996. A PDF version is available on the MathWorks Web site at http://www.mathworks.com/company/newsletters/news_notes/pdf/Fall96Cleve.pdf

[2] Moler, Cleve, *Numerical Computing with MATLAB*, S.I.A.M. A PDF version is available on the MathWorks Web site at <http://www.mathworks.com/moler/>.