# MATLAB® @ Work

## MATLAB Numerical Gotchas

***Richard Johnson***

"It's a feature!"

Writing correct MATLAB can be tricky. Sometimes it does not "work" the way other languages do. As with any technical language, the results of some computations may not be intuitive. Some of us have programming habits from older releases that are not as good as current techniques.



### Matrix operations

MATLAB was developed by mathematicians interested in matrices and linear algebra, so the basic operators are defined for matrices. In working with data, we want array operations more often than matrix operations. You could litter your code with . characters to be safe, but most of us don't do that.

There only three operators that are at issue * / ^. The only cases that can produce ambiguous results are square matrices. So most of us just take care when these combinations occur.

### Vector shape

In MATLAB there are vertical and horizontal vectors. Their orientation makes a significant difference in operations. In some other languages, there are only vectors.

Vector orientation does not always make a difference in functions. Many basic arithmetic functions in MATLAB operate along columns by default. Most of these will, however, produce the same scalar result for horizontal vector and a vertical vector.

Some functions like plot will work with mixed orientations. Others like polyfit will not.

## Complex numbers

You might expect that the log of a negative number would be nan or an error. MATLAB produces a complex number. When the real line is not enough, MATLAB is nonchalant about hopping into the complex plane. This may lead to some peculiar errors when it happens to you unexpectedly. I have seen this most often in the output variable from an objective function used in optimization.

MATLAB can also have an issue when finite precision interacts with complex numbers. In particular you may get a number with a tiny imaginary part when you expected a real. A common case is inverse Fourier transforming.

Be consistent with i and j. Although many MATLAB code examples use i or j as a loop counter, programmers who work extensively with complex numbers may want to reserve i or j or both for the square root of minus one.

The letters i and j have long histories in use both as imaginary numbers and as indices or loop counters. There is an inherent conflict in usage with no solution that will delight everyone. Those who favor saving i or j for the imaginary number can use different loop counters such as iSample, or I.

Those who favor using i and j for loop counters can establish a different variable for imaginary such as I, J, jay or *use the expression 1i, which The MathWorks recommends* for speed and robustness.

On a related note, there is no direct way to have MATLAB display j in a complex number.

Part of the challenge is that in MATLAB unlike some other languages, the loop counter is a variable in the workspace after the loop completes.

```
for i = 1:5
    a(i) = i^2;
end
i
```

```
i =
     5
```

## Loop variable

The loop counter has a bit of a split personality in that the number of passes through the loop is determined at the outset and independent of any changes to the variable that is the loop counter.

```
for i = 1:5
    a(i) = i^2;
    i = i+2;
end
a
i
```

```
a =
     1     4     9    16    25
```

```
i =
     7
```

Although trying to modify the loop variable does not actually cause any harm, it can confuse anyone reading the code.

## Floating zero

Use caution with floating point comparisons. Finite binary representation of real numbers can cause trouble, as seen in this example. These values work as expected.

```
shortSide = 3;
longSide = 5;
otherSide = 4;
longSide^2 == (shortSide^2 + otherSide^2)
```

```
ans =
     1
```

These do not.

```
scaleFactor = 0.01;
(scaleFactor*longSide)^2 == ((scaleFactor*shortSide)^2 + ...
(scaleFactor*otherSide)^2)
```

```
ans =
     0
```

A better method is to test for a small difference.

```
small = eps*shortSide;
thisLongSide = scaleFactor*longSide;
thisShortSide = scaleFactor*shortSide;
thisOtherSide = scaleFactor*otherSide;
thisLongSide^2 - (thisShortSide^2 + thisOtherSide^2) < small
```

```
ans =
     1
```

## NaN and Inf

Expect NaN values in data. NaN is often used for missing data. If a NaN is encountered in data, try to work around it. You may want to use the nan* functions in the Statistics Toolbox or write your own. Often it is useful to use any(isnan(x)) to screen for the presence of NaN entries. This is especially true since nan == nan is false, so any(a == nan) does not detect nan elements.

Attend to NaN results. If NaN is the result of a computation, MATLAB by default does not issue an error or warning. You may want to issue your own. Write message IDs so that these errors or warnings can be easily recognized. The debugger has a stop on nan or inf option that can sometimes be useful.

Consider using isfinite. Some computations produce Inf rather than NaN. It may be best to screen for both results using isfinite rather than isnan.

## Pre-allocation

MATLAB programmers have long known that it is important to pre-allocate arrays that will be filled in loops for better performance. For years we wrote something like:

```
nRows = 10;
nCols = 5;
z = zeros(nRows, nCols);
```

Now that a functional form of nan is available, it is usually better to write:

```
z = nan(nRows, nCols);
```

It is often easier to spot a nan that shouldn't be there than a 0.

## One index with rectangular array

Many of the matrix generating functions like zeros, rand, and their friends want to produce 2D arrays. So they treat

```
ones(3)
```

```
ans =
     1     1     1
     1     1     1
     1     1     1
```

as

```
ones(3,3)
```

```
ans =
     1     1     1
     1     1     1
     1     1     1
```

instead of

```
ones(3,1)
```

```
ans =
     1
     1
     1
```

Fortunately this usually becomes obvious quickly.

## Direction argument

Many MATLAB functions that accept 2D arrays compute along columns by default. For example

```
z = [1 2; 3 4];
sum(z)
```

```
ans =
     4     6
```

In the old days to sum along rows we would write:

```
sum(z')'
```

```
ans =
     3
     7
```

You may see this in older code. Now we would write:

```
sum(z,2)
```

```
ans =
     3
     7
```

After getting used to the new style, it is easy to expect the second argument to be the direction, but this is not always the case. For example:

```
min(z,2)
```

```
ans =
     1     2
     2     2
```

is the min of z and 2.

```
min(z,[],2)
```

```
ans =
     1
     3
```

works in the 2 direction. The behavior of a function in this respect depends on its history, that is how the input arguments used to be arranged.

## Extreme value indices

The indices returned by max and min are the first indices in the array that correspond to the extreme values. There may be other elements that also have these extreme values. For example

```
a = [1 3 2 3];
[largest, index] = max(a)
```

```
largest =
     3
index =
     2
```

You can get all the indices by

```
allIndices = find(a==largest)
```

```
allIndices =
     2     4
```

Of course the usual non-integer limitations of == apply.

This behavior also applies to some set functions like intersect.

## Index base

MATLAB always uses one-based indexing except when it doesn't, for example with the input function dlmread.