Some Common MATLAB Programming Pitfalls and How to Avoid Them

By Loren Shure

Just like natural languages, programming languages have idioms. Mastering these nuances of syntax and meaning is essential to achieving your programming goal—whether that goal is to create a robust program or simply to get the right answer. Here are some tips and best practices to help you become fluent in MATLAB® and achieve those goals.

GETTING THE EXPECTED ANSWER

Computer programs do exactly what you code them to do. This means that precise notation is crucial. Here are some particular cases.

Matrix and Element-Wise Multiplication

Suppose that you have two large square matrices, `A` and `B`, and you want to multiply corresponding elements in each one. If you use `A*B`, you will get a result of the expected size (the same as `A`), but not the right numbers. This is because, instead of multiplying corresponding elements in `A` and `B`, you have performed a matrix multiplication. To multiply the corresponding elements, you must use `A.*B`.

```
>> A = [1 2; 3 4]; B = [1 0; -1 1];
>> C = A.*B
C =
     1     0
    -3     4
>> D = A*B
D =
    -1     2
    -1     4
```

Transpose and Conjugate Transpose

In MATLAB, you can work with double-precision arrays that are either real-valued or complex, usually without applying conditional treatment. However, you still must write the correct code. For real-valued matrices, `A'` and `A.'` produce identical results.

For matrix operations with complex-valued arrays, the matrix transpose operator (`'`) is often appropriate. For complex matrices, this operator is the complex conjugate transpose. To avoid the conjugation of a complex matrix, use the non-conjugate transpose operator (`.'`).

```
>> C= A+i*B
  C =
     1.0000 + 1.0000i   2.0000
     3.0000 - 1.0000i   4.0000 + 1.0000i
  >> E = C'
  E =
     1.0000 - 1.0000i   3.0000 + 1.0000i
     2.0000             4.0000 - 1.0000i
  >> F = C.'
  F =
     1.0000 + 1.0000i   3.0000 - 1.0000i
     2.0000             4.0000 + 1.0000i
```

Imaginary Unit `i` or `j`

MATLAB defines the constants `i` and `j` as `sqrt(-1)`, the imaginary unit. If you use `i` or `j` as variables in your code and the code following this usage depends on either of these constants, you could get unexpected results or an error.

To avoid this problem, either use different variable names or clear the variables once you have used them.

```
>> for i=1:3
   M{i} = magic(i+2)
end
M =
    [3x3 double]
M =
    [3x3 double]    [4x4 double]
M =
    [3x3 double]    [4x4 double]    [5x5 double]
>> clear i
```

Command/Function Duality

Correct punctuation can be important in MATLAB, particularly the use of parentheses. In MATLAB, the following two statements are equivalent.

```
foo baz 5
foo('baz','5')
```

By omitting parentheses when calling a function, you call the function as a command. MATLAB then treats all the arguments following the function name as string inputs. To assign the output of a function call to a variable, use the functional form of the call, not the command version.

Floating-Point Comparison

The default data type in MATLAB is floating-point double-precision. Because the double-precision data type uses a finite number of bits—64—to represent numbers (53 bits for the mantissa, or approximately 16 decimal digits), many numbers, such as 1/7, cannot be represented exactly. To compare computational results, it is inadvisable to check for strict equality between floating-point quantities. Instead, use `eps(target)` to obtain a result that matches the target within an acceptable tolerance.

In the following example, I've arbitrarily chosen 10 times the smallest possible difference:

```
>> A = 1/7
  A =
     0.142857142857143
  >> B = 0.142857142857143
  B =
     0.142857142857143
  >> dAB = A-B
  dAB =
     -1.387778780781446e-016
  >> AapproxB = abs(A-B) < (10 * eps(B))
  AapproxB =
       1
```

CREATING ROBUST PROGRAMS

It's helpful to anticipate problems that might arise when an end-user runs your code and program to avoid errors or incorrect calculations. Here are some tools for creating robust programs.

Use `MException` Instead of `lasterror`

To ensure that your program is robust, you must capture and then deal with potential errors. If you use the older `lasterror` mechanism to do this, however, you risk overwriting errors that have not yet been processed. To avoid this problem, use an `MException` object, introduced in R2007b.

Old pattern

```
try
  doSomething;
catch
  rethrow(lasterror)
end
```

New pattern

```
try
  doSomething;
catch myException
  rethrow(myException) % or throw or throwAsCaller
end
```

Use `try/catch Exception` Instead of the Two-Argument Form of `eval`

The function `eval` lets you execute just about any valid MATLAB statement. Because the argument to `eval` is a string, however, MATLAB cannot anticipate what will be executed and may be unable to take advantage of possible optimizations. The same is true for the two-argument option for `eval`, which only evaluates the second input if an error occurs while the first one is executing. To make the code transparent to MATLAB, use `try/catch`. The following script puts a `value` in the structure `S`, if `value` exists, or else sets that structure field to empty:

Old pattern

```
eval(['S.field = value'],'S.field = []')
```

New pattern

```
try
  S.field = value;
catch myException
  S.field = [];
end
```

Use Dynamic Fieldnames Instead of eval

To create an arbitrary field of a `struct`, use dynamic field names instead of `eval` to construct the field name. Using dynamic field names lets you address fields of a `struct` without knowing the exact names explicitly when you write the program. Additionally, dynamic field names result in more efficient MATLAB code and enable MATLAB to more thoroughly analyze your program. The following two programs create a structure named `S` with field names `magic1, ..., magic5.`

Old pattern

```
n = 5;
for ind = 1:n
  eval(['S.magic' int2str(ind) '= magic(ind+2)']);
end
```

New pattern

```
n = 5;
for ind = 1:n
  S.(['magic' int2str(ind)]) = magic(ind+2);
end
```

Use Function Handles Instead of Strings to Specify Functions

Some functions in MATLAB require you to specify another function as one of the input arguments. For instance, the function `fzero` lets you find a zero or root of a specified function. In the following code samples, we find the value near a set of points, `pts`, where the function `sin` is `0`.

If you specify the function of interest as a string, MATLAB must search the path to find the function. If you are performing a calculation in a loop, perhaps to find roots near multiple locations, MATLAB must search for the target function each time it passes through the loop in case the path has changed and a different version of the target is now relevant.

A function handle locks down the target version, enabling MATLAB to call the target function without searching for it each time it passes through the loop.

Old pattern

```
pts = 0:1000;
x = zeros(size(pts));
fun = 'sin';
for npts = 1:length(pts)
   x(npts) = fzero(fun,pts(npts));
end
```

New pattern

```
pts = 0:1000;
x = zeros(size(pts));
fun = @sin;
for npts = 1:length(pts)
   x(npts) = fzero(fun,pts(npts));
end
```

Keep the Datatype for a Variable Constant Throughout a Program

MATLAB tries to conserve memory by optimizing calculations. When you run an M-file, MATLAB first examines the file and then tries to execute it in an optimal way. Once the analysis determines that a variable `D` is a particular data type—or, for example, `double`—and that it remains that data type throughout the program, MATLAB can perform its optimizations.

Old pattern

```
D = 17.34;
doSomething;
D = single(D);
```

New pattern

```
D = 17.34;
doSomething;
S = single(D);
```

Use an Output Argument with `load`

MATLAB functions can behave in unexpected ways if MATLAB cannot "see" that some of the names used in the program are variables. To avoid this possible confusion, use an explicit output variable with `load`. MATLAB will then better analyze the program and more consistently recognize which programming elements are variables. A variable can spring into existence if you use `load` to include a dataset without specifying the variables. For example, suppose that you have a variable named `var` in `mydat.mat` and want to display that value from your program.

```
function mydisp
load mydat
disp(var)
```

When you run `mydisp`, you get this error:

```
>> mydisp
??? Input argument "x" is undefined.

Error in ==> var at 55
if isinteger(x)

Error in ==> mydisp at 3
disp(var)
```

This error occurs because `var` is the name of a function on the MATLAB path, and the MATLAB analysis resolved this before running the program.

To avoid the error, use one of these two alternatives:

```
function mydisp1
load mydat3 var
disp(var)
```

```
function mydisp2
S = load('mydat.mat;);
disp(S.var)
```

In both cases, MATLAB understands what the variables in the program are (`var` and `S`) and avoids the conflict with the function found on the path. ■

Quick Tips

- When debugging, use `clear` variables (or, starting with R2008a, `clearvars`) to clear the workspace. Using `clear` itself clears breakpoints in addition to variables.
- To extract the contents from a cell in a cell array, use `{}`. To treat the cell array like any other array—for example, for reshaping—use `()`.
- Keep handles to graphics objects that you plan to update later instead of depending on global state. For example, create an axes with `hAx = axes;` and then use `hAx` instead of `gca`. Using the specific reference is more robust in cases when something other than your figure window or axis has the focus in MATLAB.

Resources

BLOG: Loren on the Art of MATLAB
http://blogs.mathworks.com/nn8/loren

MATLAB HELP
www.mathworks.com/nn8/matlabdoc

91611v00 10/08