MATLAB Programming Style Guidelines

Richard Johnson

Version 1.5 October 2002

"Language is like a cracked kettle on which we beat tunes to dance to, while all the time we long to move the stars to pity." Gustave Flaubert, in *Madame Bovary*

Table of Contents

Introduction

Advice on writing MATLAB® code usually addresses efficiency concerns, with recommendations such as "Don't use loops." This document is different. Its concerns are correctness, clarity and generality. The goal of these guidelines is to help produce code that is more likely to be correct, understandable, sharable and maintainable. As Brian Kernighan writes, "Well-written programs are better than badly-written ones -- they have fewer errors and are easier to debug and to modify -- so it is important to think about style from the beginning."

This document lists MATLAB coding recommendations consistent with best practices in the software development community. These guidelines are generally the same as those for C, C++ and Java, with modifications for Matlab features and history. The recommendations are based on guidelines for other languages collected from a number of sources and on personal experience. These guidelines are written with **MATLAB** in mind, and they should also be useful for related languages such as Octave, Scilab and O-Matrix.

Guidelines are not commandments. Their goal is simply to help programmers write well. Many organizations will have reasons to deviate from them.

> "You got to know the rules before you can break 'em. Otherwise it's no fun." Sonny Crockett in Miami Vice

MATLAB is a registered trademark of The MathWorks, Inc. In this document the acronym TMW refers to The MathWorks, Inc.

This writeup is dedicated to those who care enough to improve.

Naming Conventions

Patrick Raume, "A rose by any other name confuses the issue."

Establishing a naming convention for a group of developers can become ridiculously contentious. This section describes a commonly used convention. It is especially helpful for an individual programmer to follow a naming convention.

Variables

The names of variables should document their meaning or use.

Variable names should be in mixed case starting with lower case.

This is common practice in the C++ development community. TMW sometimes starts variable names with upper case, but that usage is commonly reserved for types or structures in other languages.

```
linearity, credibleThreat, qualityOfLife
```

An alternative technique is to use underscore to separate parts of a compound variable name. This technique, although readable, is not commonly used for variable names in other languages. Another consideration for using underscore in variable names in legends is that the Tex interpreter in **MATLAB** will read underscore as a switch to subscript.

Variables with a large scope should have meaningful names. Variables with a small scope can have short names.

In practice most variables should have meaningful names. The use of short names should be reserved for conditions where they clarify the structure of the statements. Scratch variables used for temporary storage or indices can be kept short. A programmer reading such variables should be able to assume that its value is not used outside a few lines of code. Common scratch variables for integers are `i, j, k, m, n` and for doubles `x, y` and `z.`

The prefix *n* should be used for variables representing the number of objects.

This notation is taken from mathematics where it is an established convention for indicating the number of objects.

```
nFiles, nSegments
```

A MATLAB-specific addition is the use of m for number of rows (based on matrix notation), as in

```
mRows
```

A convention on pluralization should be followed consistently.

A suggested practice is to make all variable names either singular or plural. Having two variables with names differing only by a final letter s should be avoided. An acceptable alternative for the plural is to use the suffix Array.

```
point,   pointArray
```

Variables representing a single entity number can be suffixed by *No* or prefixed by *i*.

The No notation is taken from mathematics where it is an established convention for indicating an entity number.

```
tableNo, employeeNo
```

The i prefix effectively makes the variables named iterators.

```
iTable, iEmployee
```

Iterator variables should be named or prefixed with *i*, *j*, *k* etc.

The notation is taken from mathematics where it is an established convention for indicating iterators.

```
for iFile = 1:nFiles
   :
end
```

Note that applications using complex numbers should reserve i, j or both for use as the imaginary number.

For nested loops the iterator variables should be in alphabetical order.

For nested loops the iterator variables should be helpful names.

```
for iFile = 1:nFiles
   for jPosition = 1:nPositions
      :
   end
   :
end
```

Negated boolean variable names should be avoided.

A problem arises when such a name is used in conjunction with the logical negation operator as this results in a double negative. It is not immediately apparent what `~isNotFound` means.

Use `isFound`

Avoid `isNotFound`

Acronyms, even if normally uppercase, should be mixed or lower case.

Using all uppercase for the base name will give conflicts with the naming conventions given above. A variable of this type would have to be named `dVD, hTML` etc. which obviously is not very readable. When the name is connected to another, the readability is seriously reduced; the word following the abbreviation does not stand out as it should.

Use `html, isUsaSpecific, checkTiffFormat()`

Avoid `hTML, isUSASpecific, checkTIFFFormat()`

Avoid using a keyword or special value name for a variable name.

MATLAB can produce cryptic error messages or strange results if any of its reserved words or builtin special values is redefined. Reserved words are listed by the command iskeyword. Special values are listed in the documentation.

Constants

Named constants (including globals) should be all uppercase using underscore to separate words.

This is common practice in the C++ development community. Although TMW may appear to use lower case names for constants, for example pi, such builtin constants are actually functions.

`MAX_ITERATIONS, COLOR_RED`

Constants can be prefixed by a common type name.

This gives additional information on which constants belong together and what concept the constants represent.

`COLOR_RED, COLOR_GREEN, COLOR_BLUE`

Structures

Structure names should begin with a capital letter.

This usage is consistent with C++ practice, and it helps to distinguish between structures and ordinary variables.

The name of the structure is implicit, and need not be included in a fieldname.

Repetition is superfluous in use, as shown in the example.

Use `Segment.length`

Avoid `Segment.segmentLength`

Functions

The names of functions should document their use.

Names of functions should be written in lower case.

It is clearest to have the function and its m-file names the same. Using lower case avoids potential filename problems in mixed operating system environments.

`getname(.), computetotalwidth(.)`

There are two other function name conventions commonly used. Some people prefer to use underscores in function names to enhance readability. Others use the naming convention proposed here for variables.

Functions should have meaningful names.

There is an unfortunate MATLAB tradition of using short and often somewhat cryptic function names—probably due to the DOS 8 character limit. This concern is no longer relevant and the tradition should usually be avoided to improve readability.

Use `computetotalwidth()`

Avoid `compwid()`

An exception is the use of abbreviations or acronyms widely used in mathematics.

`max(.), gcd(.)`

Functions with such short names should always have the complete words in the first header comment line for clarity and to support `lookfor` searches.

Functions with a single output can be named for the output.

This is common practice in TMW code.

`mean(.), standarderror(.)`

Functions with no output argument or which only return a handle should be named after what they do.

This practice increases readability, making it clear what the function should (and possibly should not) do. This makes it easier to keep the code clean of unintended side effects.

`plot(.)`

The prefixes *get/set* should generally be reserved for accessing an object or property.

General practice of TMW and common practice in C++ and Java development. A plausible exception is the use of set for logical set operations.

`getobj(.); setappdata(.)`

The prefix *compute* can be used in methods where something is computed.

Consistent use of the term enhances readability. Give the reader the immediate clue that this is a potentially complex or time consuming operation.

`computweightedaverage(); computespread()`

The prefix *find* can be used in methods where something is looked up.

Give the reader the immediate clue that this is a simple look up method with a minimum of computations involved. Consistent use of the term enhances readability and it is a good substitute for get.

`findoldestrecord(.); findheaviestelement(.);`

The prefix *initialize* can be used where an object or a concept is established.

The American *initialize* should be preferred over the British *initialise*. Abbreviation *init* should be avoided.

`initializeproblemstate(.);`

The prefix *is* should be used for boolean functions.

Common practice in TMW code as well as C++ and Java.

`isoverpriced(.); iscomplete(.)`

There are a few alternatives to the `is` prefix that fit better in some situations. These include the `has, can` and `should` prefixes:

`hasLicense(.); canEvaluate(.); shouldSort(.);`

Complement names should be used for complement operations.

Reduce complexity by symmetry.

```
get/set, add/remove, create/destroy, start/stop, insert/delete,
increment/decrement, old/new, begin/end, first/last, up/down,
min/max, next/previous, old/new, open/close, show/hide,
suspend/resume, etc.
```

Avoid unintentional shadowing.

In general function names should be unique. Shadowing (having two or more functions with the same name) increases the possibility of unexpected behavior or error. Names can be checked for shadowing using `which -all` or `exist`.

General

Names of dimensioned variables and constants should usually have a units suffix.

Using a single set of units is an attractive idea that is only rarely implemented completely. Adding units suffixes helps to avoid the almost inevitable mixes.

```
incidentAngleRadians
```

Abbreviations in names should be avoided.

Using whole words reduces ambiguity and helps to make the code self-documenting.

Use `computearrivaltime(.)`

Avoid `comparr(.)`

Domain specific phrases that are more naturally known through their abbreviations or acronyms should be kept abbreviated. Even these cases might benefit from a defining comment near their first appearance.

```
html, cpu, cm
```

Consider making names pronounceable.

Names that are at least somewhat pronounceable are easier to read and remember.

All names should be written in English.

The MATLAB distribution is written in English, and English is the preferred language for international development.

Files and Organization

Structuring code, both among and within files is essential to making it understandable. Thoughtful partitioning and ordering increase the value of the code.

M Files

Modularize.

The best way to write a big program is to assemble it from well designed small pieces (usually functions). This approach enhances readability, understanding and testing by reducing the amount of text which must be read to see what the code is doing. Code longer than two editor screens is a candidate for partitioning. Small well designed functions are more likely to be usable in other applications.

Make interaction clear.

A function interacts with other code through input and output arguments and global variables. The use of arguments is almost always clearer than the use of globals. Structures can be used to avoid long lists of input or output arguments.

Partitioning

All subfunctions and many functions should do one thing very well.
Every function should hide something.

Use existing functions.

Developing a function that is correct, readable and reasonably flexible can be a significant task. It may be quicker or surer to find an existing function that provides some or all of the required functionality.

Any block of code appearing in more than one m-file should be considered for packaging as a function.

It is much easier to manage changes if code appears in only one file. "Change is inevitable…except from vending machines."

Subfunctions

A function used by only one other function should be packaged as its subfunction in the same file. This makes the code easier to understand and maintain.

Test scripts

Write a test script for every function. This practice will improve the quality of the initial version and the reliability of changed versions. Consider that any function too difficult to test is probably too difficult to write. Boris Beizer: "More than the act of testing, the act of designing tests is one of the best bug preventers known."

Input and Output

Make input and output modules.

Output requirements are subject to change without notice. Input format and content are subject to change and often messy. Localizing the code that deals with them improves maintainability. Avoid mixing input or output code with computation, except for preprocessing, in a single function. Mixed purpose functions are unlikely to be reusable.

Format output for easy use.

If the output will most likely be read by a human, make it self descriptive and easy to read.
If the output is more likely to be read by software than a person, make it easy to parse.
If both are important, make the output easy to parse and write a formatter function to produce a human readable version.

Statements

Variables and constants

Variables should not be reused unless required by memory limitation.

Enhance readability by ensuring all concepts are represented uniquely. Reduce chance of error from misunderstood definition.

Related variables of the same type can be declared in a common statement.
Unrelated variables should not be declared in the same statement.

It enhances readability to group variables.

```
persistent  x, y, z
global  REVENUE_JANUARY, REVENUE_FEBRUARY
```

Consider documenting important variables in comments near the start of the file.

It is standard practice in other languages to document variables where they are declared. Since MATLAB does not use variable declarations, this information can be provided in comments.

```
% pointArray    Points are in rows with coordinates in columns.
```

Consider documenting constant assignments with end of line comments.

This gives additional information on rationale, usage or constraints.

```
THRESHOLD = 10;  % Maximum noise level found by experiment.
```

Globals

Use of global variables should be minimized.

Clarity and maintainability benefit from argument passing rather than use of global variables. Some use of global variables can be replaced with the cleaner `persistent` or with `getappdata`.

Use of global constants should be minimized.

Use an m-file or mat file. This practice makes it clear where the constants are defined and discourages unintentional redefinition. If the file access is undesirable, consider using a structure of global constants.

Loops

Loop variables should be initialized immediately before the loop.

This improves loop speed and helps prevent bogus values if the loop does not execute for all possible indices.

```
result = zeros(nEntries,1);
for index = 1:nEntries
   result(index) = foo(index);
end
```

The use of break and continue in loops should be minimized.

These constructs can be compared to `goto` and they should only be used if they prove to have higher readability than their structured counterpart.

The end lines in nested loops can have comments

Adding comments at the end lines of long nested loops can help clarify which statements are in which loops and what tasks have been performed at these points.

Conditionals

Complex conditional expressions should be avoided. Introduce temporary logical variables instead.

By assigning logical variables to expressions, the program gets automatic documentation. The construction will be easier to read and to debug.

```
if (value>=lowerLimit)&(value<=upperLimit)&~ismember(value,…
valueArray)
  :
end
```

should be replaced by:

```
isValid = (value >= lowerLimit) & (value <= upperLimit);
isNew   = ~ismember(value, valueArray);
```

```
if (isValid & isNew)
  :
end
```

The usual case should be put in the if-part and the exception in the else-part of an *if else* statement.

This practice improves readability by preventing exceptions from obscuring the normal path of execution.

```
fid = fopen(fileName);
if (fid~=-1)
  :
else
  :
end
```

The conditional expression *if 0* should be avoided, except for temporary block commenting.

Make sure that the exceptions don't obscure the normal path of execution. Using the block comment feature of the editor is preferred.

A *switch* statement should include the otherwise condition.

Leaving the `otherwise` out is a common error, which can lead to unexpected results.

```
switch (condition)
case ABC
   statements;
case DEF
   statements;
otherwise
   statements;
end
```

The *switch* variable should usually be a string.

Character strings work well in this context and they are usually more meaningful than enumerated cases.

General

Avoid cryptic code.

There is a tendency among some programmers, perhaps inspired by Shakespeare's line: "Brevity is the soul of wit", to write MATLAB code that is terse and even obscure. Writing concise code can be a way to explore the features of the language. However, in almost every circumstance, clarity should be the goal. As Steve Lord of TMW has written, "A month from now, if I look at this code, will I understand what it's doing?"

Try to avoid the situation described by the Captain in *Cool Hand Luke*, "What we've got here is failure to communicate."

The importance of this issue is underlined by many authors. Martin Fowler: "Any fool can write code that a computer can understand. Good programmers write code that humans can understand." Kreitzberg and Shneiderman: "Programming can be fun, so can cryptography; however they should not be combined."

Use parentheses.

MATLAB has documented rules for operator precedence, but who wants to remember the details? If there might be any doubt, use parentheses to clarify expressions. They are particularly helpful for extended logical expressions.

The use of numbers in expressions should be minimized. Numbers that are subject to change usually should be named constants instead.

If a number does not have an obvious meaning by itself, readability is enhanced by introducing a named constant instead. It can be much easier to change the definition of a constant than to find and change all of the relevant occurrences of a literal number in a file.

Floating point constants should always be written with a digit before the decimal point.

This adheres to mathematical conventions for syntax. Also, 0.5 is more readable than .5; it is not likely to be read as the integer 5.

Use `THRESHOLD = 0.5;`

Avoid `THRESHOLD = .5;`

Floating point comparisons should be made with caution.

Binary representation can cause trouble, as seen in this example.

```
shortSide = 3;
longSide = 5;
otherSide = 4;
longSide^2 == (shortSide^2 + otherSide^2)
ans =
     1
scaleFactor = 0.01;
(scaleFactor*longSide)^2 == ((scaleFactor*shortSide)^2 + …
(scaleFactor*otherSide)^2)
ans =
     0
```

Layout, Comments and Documentation

Layout

The purpose of layout is to help the reader understand the code. Indentation is particularly helpful for revealing structure.

Content should be kept within the first 80 columns.

80 columns is a common dimension for editors, terminal emulators, printers and debuggers, and files that are shared between several people should keep within these constraints. Readability improves if unintentional line breaks are avoided when passing a file between programmers.

Lines should be split at graceful points.

Split lines occur when a statement exceeds the suggested 80 column limit.

In general:

- Break after a comma or space.
- Break after an operator.
- Align the new line with the beginning of the expression on the previous line.

```
totalSum = a + b + c + …
           d + e;
function (param1, param2,…
          param3)
setText (['Long line split' …
         'into two parts.']);
```

Basic indentation should be 3 or 4 spaces.

Good indentation is probably the single best way to reveal program structure.
Indentation of 1 is too small to emphasize the logical layout of the code. Indentation of 2 is sometimes suggested to reduce the number of line breaks required to stay within 80 columns for nested statements, but MATLAB is usually not deeply nested. Indentation larger than 4 can make nested code difficult to read since it increases the chance that the lines must be split. Indentation of 4 is the current default in the **MATLAB** editor; 3 was the default in some previous versions.

Indentation should be consistent with the MATLAB Editor.

The MATLAB editor provides indentation that clarifies code structure and is consistent with recommended practices for C++ and Java.

In general a line of code should contain only one executable statement.

This practice improves readability and allows JIT acceleration.

Short single statement if, for or while statements can be written on one line.

This practice is more compact, but it has the disadvantage that there is no indentation format cue.

```
if(condition), statement; end
while(condition), statement; end
for iTest = 1:nTest, statement; end
```

White Space

White space enhances readability by making the individual components of statements stand out.

Surround =, &, and| by spaces.

Using space around the assignment character provides a strong visual cue separating the left and right hand sides of a statement.
Using space around the binary logical operators can clarify complicated experessions.

```
simpleSum = firstTerm+secondTerm;
```

Conventional operators can be surrounded by spaces.

This practice is controversial. Some believe that it enhances readability.

```
simpleAverage = (firstTerm + secondTerm) / two;
1 : nIterations
```

Commas can be followed by a space.

These spaces can enhance readability. Some programmers leave them out to avoid split lines.

```
foo(alpha, beta, gamma)
foo(alpha,beta,gamma)
```

Semicolons or commas for multiple commands in one line should be followed by a space character.

Spacing enhances readability.

```
if (pi>1), disp('Yes'), end
```

Keywords should be followed by a space.

This practice helps to distinguish keywords from functions.

Logical groups of statements within a block should be separated by one blank line.

Enhance readability by introducing white space between logical units of a block.

Blocks should be separated by more than one blank line.

One approach is to use three blank lines. By making the space larger than space within a block, the blocks will stand out within the file. Another approach is to use the comment symbol followed by a repeated character such as * or -.

Use alignment wherever it enhances readability.

Code alignment can make split expressions easier to read and understand. This layout can also help to reveal errors.

```
weightedPopulation = (doctorWeight * nDoctors) + …
                     (lawyerWeight * nLawyers) + …
                     (chiefWeight *  nChiefs);
```

Comments

The purpose of comments is to add information to the code. Typical uses for comments are to explain usage, provide reference information, to justify decisions, to describe limitations, to mention needed improvements. Experience indicates that it is better to write comments at the same time as the code rather than to intend to add comments later.

Comments cannot justify poorly written code.

Comments cannot make up for code lacking appropriate name choices and an explicit logical structure. Such code should be rewritten. Steve McConnell: “Improve the code and then document it to make it even clearer.”

Comments should agree with the code, but do more than just restate the code.

A bad or useless comment just gets in the way of the reader. N. Schryer: “If the code and the comments disagree, then both are probably wrong.” It is usually more important for the comment to address why or how rather than what.

Comments should be easy to read.

There should be a space between the `%` and the comment text. Comments should start with an upper case letter and end with a period.

Comments should usually have the same indentation as the statements referred to.

This is to avoid having the comments break the layout of the program. End of line comments tend to be cryptic and should be avoided except for constant definitions.

Function header comments should support the use of *help* and *lookfor*.

`help` prints the first contiguous block of comment lines from the file. Make it helpful.
`lookfor` searches the first comment line of all m-files on the path. Try to include likely search words in this line.

Function header comments should discuss any special requirements for the input arguments.

The user will need to know if the input needs to be expressed in particular units or is a particular type of array.

```
% ejectionFraction must be between 0 and 1, not a percentage.
% elapsedTimeSeconds must be one dimensional.
```

Function header comments should describe any side effects.

Side effects are actions of a function other than assignment of the output variables. A common example is plot generation. Descriptions of these side effects should be included in the header comments so that they appear in the help printout.

In general the last function header comment should restate the function line.

This allows the user to glance at the help printout and see the input and output argument usage.

Writing the function name using uppercase in the function header is controversial.

This is a MathWorks practice, which is intended to make the function name prominent. Many other authors do not follow this practice. Its disadvantage is that in code the function name should usually be in lower case.

Avoid clutter in the help printout of the function header.

It is common to include copyright lines and change history in comments near the beginning of a function file. There should be a blank line between the header comments and these comments so that they are not displayed in response to help.

All comments should be written in English.

In an international environment, English is the preferred language.

Documentation

Formal documentation

To be useful documentation should include a readable description of what the code is supposed to do (Requirements), how it works (Design), which functions it depends on and how it is used by other code (Interfaces), and how it is tested. For extra credit, the documentation can include a discussion of alternative solutions and suggestions for extensions or maintenance.

Dick Brandon: “Documentation is like sex; when it's good, it's very, very good, and when it's bad, it's better than nothing.”

Consider writing the documentation first.

Some programmers believe that the best approach is “Code first and answer questions later.” Through experience most of us learn that developing a design and then implementing it leads to a much more satisfactory result.

Development projects are almost never completed on schedule. If documentation and testing are left for last, they will get cut short. Writing the documentation first assures that it gets done and will probably reduce development time.

Changes.

The professional way to manage and document code changes is to use a source control tool. For very simple projects, adding change history comments to the function files is certainly better than nothing.

```
% 24 November 1971, D.B. Cooper, exit conditions modified.
```

References

The Practice of Programming, Brian Kernighan and Rob Pike

The Pragmatic Programmer, Andrew Hunt, David Thomas and Ward Cunningham

Java Programming Style Guidelines, Geotechnical Software Services

Code Complete, Steve McConnel - Microsoft Press

C++ Coding Standard, Todd Hoff
