MATLAB *Style* Guidelines 2.0

Richard Johnson

Version 2, March 2014

ISBN
Library of Congress Control Number

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.

Some ways of coding are better than others. It's as simple as that. Coding conventions add value by helping to make mistakes obvious. 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."

When people look at your code, will they see what you are doing? The spirit of this book can be pithily expressed as "Avoid write-only code."

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.

Issues of style are becoming increasingly important as the MATLAB language changes and its use becomes more widespread. In the early versions, all variables were double precision matrices; now many data types are available. Usage has grown from small scale prototype code to large and complex production code developed by groups. Integration with Java is standard and Java classes can appear in MATLAB code. All of these changes have made clear code writing more important and more challenging.

Guidelines are not commandments. Their goal is simply to help programmers write well. Many organizations will have reasons to deviate from some of these guidelines, but most organizations will benefit from adopting some style guidelines.

For broader and deeper coverage of MATLAB style and best development practices, check out the book:

The Elements of MATLAB Style

available at

http://datatool.com/resources.html

or Amazon.

If you have corrections or comments, please contact richj@datatool.com

Naming Conventions

The purpose of a software naming convention is to help the reader and the programmer. Establishing a naming convention for a group of developers is very important, but the process can become ridiculously contentious. There is no naming convention that will please everyone.

Following a convention is more important than what the details of the convention are. This section describes a commonly used convention that will be familiar to many programmers of MATLAB and other languages.

Variables

The names of variables should document their meaning or use. MATLAB can cope with

```
z = x * y
```

but the reader will do better with

```
wage = hourlyRate * nHours
```

Write variable names in mixed case starting with lower case.

This is common practice in other languages. Names that start with upper case are commonly reserved for types or structures in other languages.

```
linearity, credibleThreat, qualityOfLife
```

Very short variable names can be in upper case if they are upper case in conventional usage and unlikely to become parts of compound variable names. Examples are typically domain specific, such as E for Young's modulus, which would be misleading as e.

Some programmers prefer 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 graph titles, labels and legends is that the Tex interpreter in MATLAB will read underscore as a switch to subscript, so you will need to apply the parameter/value pair 'interpreter', 'none' for each text string.

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 or are consistent with intended generality. For example in a general purpose function it may be appropriate to use variable names such as x, y, z, t.

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 names for scratch variables used as integers are `k,m,n` and for doubles s, `t`, `x,y,` and `z`.

Programmers who work with complex numbers may choose to reserve `i` or `j` or both for the square root of minus one. However, The MathWorks recommends using 1i or 1j for the imaginary number. These execute more quickly and cannot be overwritten.

Use the prefix *n* 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 option is the use of m for number of rows (based on matrix notation), as in

```
mRows
```

Follow a consistent convention on pluralization.

Having two variables with names differing only by a final letter s should be avoided. Some programmers make all variable names either singular or plural, but others find this can be awkward.

An acceptable usage for the plural is to use a suffix like Array.

```
point,   pointArray, PointList
```

An acceptable usage for the singular is to use a prefix like this.

```
thisPoint
```

Most programmers do not use `the` as a prefix for a single example or element.

Use the suffix *No* or *Num* or the prefix *i* in a variable name representing a single entity number.

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 variable a named iterator.

```
iTable, iEmployee
```

Prefix iterator variable names 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
```

Some programmers use the single letter variable names `i, j` or both for convenient loop iterators. Programmers who use explicit complex numbers tend to hate this practice.

For nested loops the iterator variables should usually be in alphabetical order. Some mathematically oriented programmers use a variable name starting with i for rows and j for columns.

Especially for nested loops, using iterator variable names is be helpful.

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

Avoid negated Boolean variable names.

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 is meant by names like

```
~isNotFound
```

Use
`isFound` and `~isFound`

Avoid
```
isNotFound
```

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

Using all uppercase would be inconsistent with the standard naming conventions. 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.

Use common domain-specific names.

If the software is targeted for a knowledge domain or a user group, use names consistent with standard practice.

Use
`roi,` or `regionOfInterest`
Avoid
`imageRegionForAnalysis`

Avoid Variable Names That Shadow Functions.

There are several names of functions in the MATLAB product that seem to be tempting to use as variable names. Such usage in scripts will shadow the functions and can lead to errors. Using a variable and a function with the same name inside a function will probably cause an error.
Some standard function names that have appeared in code examples as variables are

```
alpha, angle, axes, axis, balance, beta, contrast,
gamma, image, info, input, length, line, mode,
power, rank, run, start, text, type
```

Using a well-known function name as a variable name also reduces readability. If you want to use a standard function name such as length in a variable name, then you can add a qualifier, such as a unit suffix, or a noun or adjective prefix:

```
  lengthCm, armLength, thisLength
```

Avoid Hungarian notation.

There are at least two versions of Hungarian notation that have been used by some software developers. A Hungarian variable name typically involves 1 or 2 prefixes, a name root, and a qualifier suffix. These names can be pretty ugly, particularly when they are strings of contractions. A bigger problem occurs if a prefix, as is often suggested, encodes data type. Then if the type needs to be changed, all incidences of the variable name need to be changed.

Use
```
thetaDegrees
```
Avoid
```
uint8thetaDegrees
```

Constants

The MATLAB language does not have true constants (except as constant properties in objects). Use standard practices to name and define constants so that they can be recognized and not unintentionally redefined.

Constant names with local scope (within an m-file) should be all uppercase using underscore to separate words.

This is common practice in other languages.

```
MAX_ITERATIONS, COLOR_RED
```

Use meaningful names for constants.

Use
```
MAX_ITERATIONS
```
Avoid
```
TEN, MAXIT
```

Constants that are output by a function with the same name should have names that are all lowercase or mixed case.

This practice is used by The MathWorks. For example the constant pi is actually a function.

```
offset, standardValue
```

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 other languages, 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.

Use
```
Segment.length
```
Avoid
```
Segment.segmentLength
```

Be Careful with Fieldnames.

When you set the value of a structure field, MATLAB replaces the existing value if the field already exists or creates a new field if it does not. This can lead to unexpected results if the fieldnames are not consistent, for example, when a structure has field

```
Acme.source = 'CNN';
```

that you intend to update, but you type

```
Acme.sourceName = 'Bloomberg';
```

The structure will now have two fields.

Functions

The names of functions should document their use.

Write names of functions in lower or mixed case.

Initially all MATLAB function names were in lower case.

```
linspace, meshgrid
```

Almost all functions supplied by The MathWorks still follow this convention. This practice can be awkward with longer compound names as in the short-lived

```
isequalwithequalnans
```

In other languages, it is common to use mixed case, starting with lower case, for function names. Many MATLAB programmers follow this convention.

```
predictSeaLevel, publishHelpPages
```

Some programmers prefer to use underscores in function names, but this practice is not common.

Use meaningful function names.

There is an unfortunate MATLAB tradition of using short and often somewhat cryptic function names—probably due to the old 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 have a clear description in the header comment lines.

Name functions that have a single output based on the output.

This is common practice in MathWorks 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
```

Reserve the prefixes *get/set* for functions that access an object or property.

This is the general practice of The MathWorks and common practice in other languages. A plausible exception is the use of set for logical set operations.

```
getobj, setAppData
```

Reserve the prefix *compute* for functions where something is computed.

Consistent use of the term enhances readability. Give the reader an immediate clue what the function is doing.

```
computeWeightedAverage,  computeSpread
```

Avoid possibly confusing alternatives such as `find` or `make`.

Consider reserving the prefix *find* for functions 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 the overused prefix `get`.

```
findOldestRecord,  findTallestMan
```

Consider using the prefix *initialize* where an object or a variable is established.

The American *initialize* should be preferred over the British *initialise*. Avoid the abbreviation `init`.

```
initializeProblemState
```

Use the prefix *is* for Boolean functions.

This is common practice in MathWorks code as well as other languages.

```
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
```

Use complement names 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,
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`.

Overload functions will of course have the same name. Do not create an overload situation when a polymorphic function would be adequate.

General

Consider a unit suffix for dimensioned variables and constants.

Using a single set of units for a project is an attractive idea that is only rarely implemented completely. Adding unit suffixes helps to avoid the almost inevitable unintended mixed unit expressions.

```
incidentAngleRadians
```

Minimize abbreviations in names.

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.

Write names in English.

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

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.

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.
```

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

Globals

Minimize use of global constants.

Use an m-file or mat file to define global constants. This practice makes it clear where the constants are defined and discourages unintentional redefinition. If the m-file access overhead produces an execution speed problem, consider using a function handle.

Minimize use of global variables.

Clarity and maintainability of functions benefit from explicit argument passing rather than use of global variables. Some use of global variables can be replaced with the cleaner `persistent` or with `getappdata`. An alternative strategy is to replace the global variable with a function.

Loops

Initialize loop result variables immediately before the loop.

Initializing these variables improves loop speed and helps prevent bogus values if the loop does not execute for all possible indices. This initialization is sometimes called pre-allocation. Placing the initialization just before the loop makes it easier to see that the variables are initialized. This practice also makes it easier to copy all the relevant code for use elsewhere.

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

Use a named variable for the argument in both the initialization statement and the `for` line.

Minimize the use of break in loops.

This keyword is often unnecessary and should only be used if it proves to have higher readability than a structured alternative.

Minimize use of continue in loops.

This keyword is often unnecessary and should only be used if it proves to have higher readability than a structured alternative.

The *end* lines in nested loops can have identifying 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

Avoid complex conditional expressions. 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
```

Put the usual case in the if-part and the unusual in the else-part of an *if else* statement.

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

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

Avoid the conditional expression *if 0*.

Make sure that this usage does not obscure the normal path of execution. To temporarily bypass code execution, use the block comment feature of the editor instead of this expression.

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
```

Use *if* when the condition is most clearly written as an expression. Use *switch* when the condition is most clearly written as a variable.

There is possible overlap in the usages of `if` and `switch`. Following this guideline helps provide consistency.

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 MathWorks has written, "A month from now, if I look at this code, will I understand what it's doing?"

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.

Minimize the use of numbers in expressions.

Numbers that are subject to possible 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.

Write fractional values 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;
```

Use caution with floating point comparisons.

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

```
shortSide = 3;
longSide = 5;
otherSide = 4;
longSide^2 == (shortSide^2 + otherSide^2)
ans =
     1
```

But

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

A better method is to test that the difference between the values is small enough.

Use the natural, straightforward form for logical expressions.

Logical expressions including negations can be difficult to understand. Strive to use positive expressions.

Use

```
iSample>=maxSamples;
```

Avoid

```
~(iSample<maxSamples);
```

Prepare for errors.

In general errors should be caught in low level routines and fixed or passed on the higher level routines for resolution. A useful tool for protection against error conditions is the `try catch` construction with MException.

Another line of defense is to use properly ordered expressions in `if` statements so that evaluation short circuiting can avoid evaluation of expressions that will trigger an error.

Include validity checking in functions used to acquire input.

Invalid input usually leads to an error stopping execution. Validity checking allows more graceful error handling. Useful tools include `validateattributes` and `inputParser`.

Avoid use of *eval* when possible.

Statements that involve `eval` tend to be harder to write correctly, more difficult to read, and slower to execute than alternatives. Use of `eval` does not support thorough checking by M-Lint. Statements that use `eval` can usually be improved by changing from commands to functions, or by using dynamic field references for structures with `setfield` and `getfield`.

Write code as functions when possible

Functions modularize computation by using internal variables that are not part of the base workspace. They make input and output variables more obvious and tend to be cleaner, more flexible, and better designed than scripts. The main role of scripts is in development because they provide direct visibility of variable dimensions, types, and values.

Write code for automation

Minimize use of `keyboard` and `input` to support automated execution and test.

Layout, Comments and Documentation

Layout

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

Keep content within the first 80 columns.

80 columns is a common dimension for editors, terminal emulators, printers and debuggers. 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.

Split long lines at graceful places.

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

In general:

Break after a comma or space.

Break after an operator.

The Editor provides indentation after the continuation operator (…). Optionally include additional spacing to 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.’]);
```

Indent 3 or 4 spaces.

Good indentation is probably the single best way to reveal program structure.

Indentation of 1 space is too small to emphasize the logical layout of the code. Indentation of 2 spaces 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.

Indent consistently with the MATLAB Editor.

The MATLAB editor provides indentation that clarifies code structure and is consistent with recommended practices for C++ and Java. If you use a different editor, try to be consistent.

Write one executable statement per line of code.

This practice improves readability and can speed execution.

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 expressions.

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

Conventional operators can be surrounded by spaces.

This practice is controversial. Some believe that it enhances readability. Others find that it makes expressions unnecessarily long.

```
simpleAverage = (firstTerm + secondTerm) / two;
```

```
for index = 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)
```

Follow semicolons or commas for multiple commands in one line by a space character.

Spacing enhances readability.

```
if (pi>1), disp(‘Yes’), end
```

Follow keywords by a space.

This practice helps to distinguish keywords from function names.

Separate logical groups of statements within a block by one blank line.

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

Separate blocks 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. A better approach is to use editor sections defined by %%.

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.

```
value = (10 * nDimes) + …
        (5 * nNickels) + …
        (1 *  nPennies);
```

Comments

The purpose of comments is to add information to the code. Typical uses for comments are to explain usage, to express the purpose of the code, to 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.

Make the comments easy to read.

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

Write comments in English.

In an international environment, English is preferred.

Header comments

The header comments are the first contiguous block of comments in an m-file. Write them to provide the user with the necessary information to use the file.

There are two styles of header comments for functions in common use. The traditional style has the comments below the function line and uses only single % signs. It was originally designed for use in the Command Window.

A more modern style has the comments above the function line. It often uses MATLAB markup and is designed to produce an HTML file for the Help and Function Browsers.

Present the function syntax in header comments.

The user will need to know the input and output arguments, their sequences and variations.

Discuss the input and output arguments in the header comments.

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

```
% completion must be between 0 and 1.
% elapsedTime must be one dimensional.
```

Describe any side effects in the header comments.

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.

Write the function name in comments using its actual case.

Old code files often used all uppercase for function names in header comments despite the actual function names being all lower case. This practice was probably intended to make the function name prominent when displayed in the colorless command window.

Most programmers now view help information in the editor window or Help or the Function Browser. The all uppercase style does not aid the reader in these contexts. Also mixed case function names are becoming more common and the use of all uppercase in comments can be confusing.

Avoid clutter in display 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 the help documentation.

Inline comments

Comments in the code as opposed to the header are intended for a programmer, not a user.

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."

Make the comments 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* the code does.

Indent code comments the same as the statements referred to.

The code is easier to read when comments do not break the layout of the program.

Minimize use of end of line comments.

The descriptiveness of end of line comments is constrained by the typical 80 column line length. In general they should only be used as an adjunct to variable declaration.

Commenting for documentation.

Documentation is aimed at two groups: users who want to run the code and programmers who read the code. In general the header comments are intended for users, and the inline comments are intended for programmers.

Comments for publishing

MATLAB supports special commenting for publishing to HTML, XML, latex, doc, ppt, and pdf formats. Publishing to HTML can be very useful for function documentation. Publishing scripts to doc or pdf can produce basic reports.

MATLAB can display HTML reference pages for user written functions in the Help Browser. The publish feature with simple MATLAB markup can produce these pages from the function m-files.

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. Keeping related information together on the same editor screen lets you see certain types of problems and fix them right away. 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 global variables. Structures can be used to avoid long lists of input or output arguments.

Interface standards bring a more familiar and consistent experience to using the function. This makes correct use more likely.

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 writing as a function.

It is much easier to manage changes if code appears in only one file. Try to use cut and paste rather than copy and paste.

Use structures for function arguments.

Usability of a function decreases as the number of arguments grows, especially when some arguments are optional. Consider using structures whenever arguments lists exceed three.

Structures can allow a change to the number of values passed to or from the function that is compatible with existing external code.

Structures can remove the need for arguments to be in fixed order. Structures can be more graceful for optional values than having a long and ordered list of variables.

Provide some generality in functions

Functions should usually be flexible enough to accept input scalars, vectors, and arrays of at least 2 dimensions. Functions with input arguments that commonly have more than one representation should work with all of them. For example image processing functions should at least work with uint8 and double variables.

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.

MATLAB allows accessing a subfunction from outside its m-file. This is generally a bad idea.

Test scripts

Write a test script for every function. This practice will improve the quality of the initial version of the function 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 also 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.

Use feof for reading files.

Depending on line or data counting can easily lead to end of file errors or incomplete input.

Toolboxes

Organize m-files that have some generality in toolboxes. Check the function names for shadowing. Add the toolbox locations to the MATLAB path.

Typically it is useful to have both project and general purpose toolboxes.

Style quotes

Martin Fowler: “Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”

“In matters of style, swim with the current; in matters of principle, stand like a rock.” Thomas Jefferson

“You got to know the rules before you can break ‘em. Otherwise it’s no fun.” Sonny Crockett in Miami Vice

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

Plato, “Nothing has its name by nature, but only by usage and custom.”

Unknown, “All general statements are false.”

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

Kreitzberg and Shneiderman: “Programming can be fun, so can cryptography; however they should not be combined.”

Jay Rodenberry, “Space…the final frontier.”

Napoleon Hill, “First comes thought; then organization of that thought into ideas and plans; then transformation of those plans into reality.”

“Change is inevitable…except from vending machines.”

References

The Elements of MATLAB Style, Richard Johnson

Clean Code, Robert Martin

Code Complete, Steve McConnell - Microsoft Press

The Elements of Java Style, Allan Vermeulen et al.

MATLAB: A Practical Introduction, Stormy Attaway

The Practice of Programming, Brian Kernighan and Rob Pike

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

Programming Style, Wikipedia