MATLAB Programming Style Guidelines

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.

```
linearity, credibleThreat, qualityOfLife
```

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.

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 will result 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, check_tiff_format()`

Avoid `hTML, isUSAspecific, check_TIFF_Format()`

Constants

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

```
MAX_ITERATIONS, COLOR_RED
```

Structures

Structure names should begin with a capital letter.

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 and with underscore.

> 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.
>
> `get_name(.), compute_total_width(.)`

Functions should have meaningful names.

> Use `compute_total_width()`
>
> Avoid `comp_wid()`
>
> 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.

> `mean(.), standard_error(.)`

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(.)`

When appropriate, use a function prefix.

> Consistent use of the terms enhances readability. Gives the reader the immediate clue what is to be expected from the function.
>
> `compute_weighted_average(); compute_spread()`

```
find_oldest_record(.); find_heaviest_element(.);

initialize_problem_state(.);

is_overpriced(.); is_complete(.)

has_license(.); can_evaluate(.); should_sort(.);
```

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

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

Use `compute_arrival_time(.)`

Avoid `comp_arr(.)`

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.

Statements

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 `get_app_data`.

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.

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
```

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
```

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. 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.

Layout and Comments

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 4 spaces.

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

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

This practice improves readability and allows JIT acceleration.

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)
```

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 -.

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 should be easy to read and have the same indentation as the statements referred to.

There should be a space between the `%` and the comment text.

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.
```

Consider restating the function line as the last function header comment if function header is long.

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

Function header example

```
% SOME_FUNCTION Some function does this on one line
%
% [output1 output2] = some_function(input1, input2)
%
% input1              explanation of input argument 1
% input2              explanation of input argument 2
% output1             explanation of output argument 1
% output2             explanation of output argument 2
%
% Examples using some_function

% Copyright (C) 2012 Daniel Forsberg
% daniel.forsberg@liu.se
```

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.