A Brief MATLAB Style Guide

Based in part on Richard Johnson's *MATLAB Style Guidelines 2.0.*

Naming Conventions

Ordinary Variables

Large scope variables should be named informatively, and follow camelCase, e.g. **`qualityOfLifeIndex`**. Limited scope variables can have short names; it is recommended to use early/mid alphabetics (e.g. **`a,b,c,i,j,k,m,n`**) for integers, late alphabetics (e.g. **`s,t,x,y,z`**) for doubles.

Use the prefix **`n`** and s-pluralization for variables representing a count of objects, e.g. **`nRegions`**, **`nBasisFunctions`**.

For a general data structure with multiple elements of the same type, avoid pluralizing with an s-ending; instead, append a vague structure type to an illustrative name, e.g. **`regionBoundaryPointArray`**, **`residualList`**.

For singular cases, or scratch subselections from a larger structure, prefix with **`this`**, e.g. **`thisPoint`**, **`thisColumn`**.

For variables representing a single entity number, use the suffix **`Num`** in the variable name, e.g. **`regionNum`**, **`basisFunctionNum`**.

For iterator variables, prefix with **`i,j,k`**, etc., e.g.

```
for iRegion = 1 :  nRegions
    ...
end
```

In nested loops, it is preferred that iterator variable prefixes nest alphabetically; however, when nested iterator variables are used to index into multidimensional arrays, this convention may be reversed in the name of storage access efficiency.

Avoid negated Boolean variable names, instead negate by operator; e.g. avoid **`isNotFull`**, instead use **`isFull`** and **`~isFull`**.

Acronyms and initialisms in variable names should be capitalized as though they are words, e.g. **`isUsaSpecific`**, **`mopittTimePeriod`**.

MATLAB keywords cannot be used as variable names. Use the **`iskeyword`** function to determine the keywords in a particular environment.

Avoid variable names that shadow extant functions in the MATLAB product. Use the **`exist`** function to check if a name exists in the workspace (and if so, to determine its type). Some common examples to be avoided include: **`alpha, angle, axes, axis, balance, beta, contrast, gamma, image, info, input, length, line, mode, power, rank, run, start, text, type`**.

Structures

Structure names should follow upper CamelCase, e.g. **`ModelResults`**.

Field names should follow ordinary variable style.

Structure names should not be included in field names, e.g., avoid **`ModelResults.modelResultsCoefficients`**; instead use **`ModelResults.coefficients`**.

Constants

Constants should be named informatively. Constant names with local scope (contained to a single m-file) should be all uppercase using underscore to separate words, e.g. **`MAX_ITERATIONS`**.

Constants that are output by a function with the same name should have names that are all lowercase, or mixed case, e.g. **`pi, standardValue`**.

Constants of a common type should be prefixed with a vague type name, e.g. **`COLOR_RED`**, **`COLOR_BLUE`**.

Functions

Function names should be all lowercase, with words separated by underscores.

Functions with a single output should be named based on that output (e.g. **`standard_error`**); functions with no output should be named based on what they do (e.g. **`plot_histogram_with_fitted_normal`**).

Avoid unintentional shadowing of existing functons. Use the **`exist`** function to check if a name exists in the workspace.

Avoid function overloading; polymorphic functions are the preferred alternative.

Use and reserve the following prefixes for the situations indicated:

- **`get`**/**`set`** for actual object property get and set operations.
- **`compute`** for functions where something is computed.
- **`find`** for lookup/search functions without intensive computations.
- **`initialize`**/**`finalize`** for object/variable establishment/clean up.
- **`is`** for Boolean functions (or **`has`**, **`can`**, or **`should`** as is appropriate for the case).

Statements

Conditionals

Avoid complex conditional expressions; instead, introduce and evaluate temporary logical variables beforehand.

For conditionals with a usual and an unusual case, put the usual case in the **`if`** side, and the unusual in the **`else`** side, e.g.:

```
xValList = rand(2,1);
if xValList(1) ~= xValList(2)
    thisWidth = xValList(2) - xValList(1);
else
    disp('Error:  Zero-width interval.');
end
```

A **`switch`** statement should include the **`otherwise`** condition, e.g.:

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

Globals

Global constants should be avoided when possible; when used, they should be defined in a dedicated m-file.

Variables

Important variables should be documented in comments near the start of the file.

Variables should not be reused; all concepts should be represented uniquely.

Loops

Pre-allocate full-size loop result variables immediately before the loop to avoid repeated variable resizing; e.g., avoid:

```
x = 0;
for k = 2:1000000
    x(k) = x(k-1) + 5;
end
```

in favor of:

```
x = zeros(1, 1000000);
for k = 2:1000000
    x(k) = x(k-1) + 5;
end
```

Numbers

Minimize the use of numbers in expressions; code with a quantity used more than once should use a named value instead.

Write fractional values with a leading 0, e.g. **THRESHOLD = 0.5**;

Layout and Documentation

Format and Whitespace

Keep content to the first 80 columns. Split long lines gracefully; break after commas, spaces, or operators. No more than one executable statement per line.

Align split expressions for readability, not just by a single tab indentation, e.g. avoid:

```
thisData = scan(fileId, fSpec, ...
    'Delimiter', ',', 'HeadLines', 1, ...
    'ReturnOnError', false);
```

in favor of:

```
thisData = scan(fileId, fSpec, ...
                'Delimiter', ',', ...
                'HeadLines', 1, ...
                'ReturnOnError', false);
```

Surround comparators/operators by spaces; e.g. avoid **thisSum=firstTerm+secondTerm**, in favor of **thisSum = firstTerm + secondTerm**. Follow commas with single spaces.

Separate logical groups of statements within a code block using a single blank line. Separate distinct blocks using editor-default section breaks; enter two percent signs (%%) at the start of the line where you want to begin the new code section, and include a section title (text on the same line, following %%). Section titles should provide a short description of the purpose of the section they demark.

Honor automatic indentation of nested loops present in the Matlab editor.

Comments

Comment code as it is written. When commenting, place a space between the % and the text.

Use header comments for functions that are consistent with the MATLAB **help** function; a comment block immediately below the function declaration that specifes the function name (all caps) and basic description in the first line, and a more detailed explanation of the function in following lines. If appropriate, create links with a "See also" line. If a function has any side effects (actions other than assignment of output variables), these should be noted in the header. E.g. a function called **add_me** might have the following header:

```
function c = add_me(a,b)
% ADD_ME Add two values together.
% C = add_me(A) adds A to itself.
%
% C = add_me(A,B) adds A to B.
%
% See also SUM, PLUS.
```

Comments for programmers should appear inline. Inline comments should be above the lines or blocks to which they refer, and indented at the same level. Avoid recapitulating the executable code in the comment; summarize the purpose of the next code block. Avoid end-of-line comments.

Notes of a general nature for code in progress can be incorporated in a comment block at the end of the function/script.

Files and Organization

M Files

Modularize code; code longer than two editor screens should be examined for potential partitioning.

Make interaction clear; use function input/output arguments whenever possible (as opposed to global variables) to manage code interaction.

Structures are preferable to long lists of input/output arguments for internal functions; consider a structure when the argument list has more than three items.

Any block of code that appears in more than one m-file should be examined as a candidate for conversion to a function.

A function used only by one other function should be packaged as a subfunction in the same file.

Every function written should have an associated test script to check/demonstrate correct functionality and basic error handling.

Input and Output

Input and output format/management should be kept distinct from computation, in separate functions when possible.

Output should be easy to parse for software, with a formatter function to make it easy to read for a human.