ENGR 133 Programming Standards

Programming standards are guidelines that one uses to write well-documented code that is understandable by others and to develop good programming habits. All MATLAB code submissions in this course must follow the standards in this document.

Why Follow Programming Standards?

- **Make code easier to understand**. Your code must be readable by team members, graders, instructors, and eventually co-workers and supervisors. You will read code written by others. Well-written code makes its purpose and logic clear and is easily used by others.
- **Help minimize errors**. Readable, easy-to-follow code helps minimize errors and allows for easier debugging when errors do occur.
- **Develop good habits**. Learning these standards now will help you be an efficient programmer and help you resist poor programming habits in the future.

Note: Course materials will attempt to abide by these standards as closely as possible to serve as an example. However, some course materials may exclude headers and comments to save space; this is not an option for your assignments.

Use the Appropriate Code Template

If provided, you must use an ENGR 133 template for MATLAB files submitted with an assignment. There are two template types, one for script files and one for function files. Use the correct template for the type of MATLAB code you are developing. Some problem sets have problem-specific templates; use those templates when provided. Otherwise, you can download a generic template from Blackboard.

Each template has a header block that starts on the first line of the code for scripts or on the second line for functions. You must complete the header block for each m-file that you submit for a grade.

Complete the Header

The header needs to contain relevant problem, program, and author information to help others understand what the program does, how to use it, and who wrote it. All m-files submitted need a complete header. All headers require a program description and author information. Function m-files also require help lines to explain the function call and its inputs and outputs.

The header information is what appears in the MATLAB Command Window when a user calls MATLAB's help functionality to learn about the script or function. The header should provide the user with all information necessary to use the file.

Program Description – all scripts and functions

The program description briefly describes the purpose of the program. Anyone looking at your code, including your grader, should be able to use your program description to understand the purpose of the code.

Author Information – all scripts and functions

All m-files need author information. Complete the appropriate lines of the header with the following information:

- Assignment: State the assignment number and the problem number (e.g., PS02, Problem 2).
- Author: Add your name and your Purdue email address with your official login (no aliases).
- Team ID: Add your section and team number. For early assignments (prior to being assigned to a team), list only your section number.

- Paired Partner: If the code is the result of a paired assignment, list your paired partner's name and official Purdue email address.
- Contributor: If you received help from another student on the code or if you helped another student on their code, then list that student's name and official Purdue email as a contributor. Add additional lines if you had more than one contributor. Read the course Academic Integrity document for more information about contributors.
- Place an X between the brackets to indicate the type of help you received from your contributor. You can select as many as apply. Leave this as-is if you did not have a contributor.

Function Call – functions only

The function call line is a help line that demonstrates how to call the function.

Input Argument and Output Arguments – functions only

Functions can have input arguments, output arguments, both, or neither. Define each input argument and output argument with a concise description that includes units as appropriate. If the function does not have input or output arguments, then indicate 'none' in the appropriate location in the header.

This is particularly important for inputs because there is no place to describe them in the body of the code.

Assign Variables

- Use descriptive variable names that make the variable's purpose clear without having to look at any more code.

  Consider these examples:

Code 1

```
%% INITIALIZATION
x = 100
y = 88
```

vs

Code 2

```
%% INITIALIZATION
numStudents = 100 % [unitless]
avgGrade = 88     % [%]
```

  The variables x and y are vague and non-descriptive while numStudents and avgGrade clearly describe what they represent.

- Assign only one variable per line.
- Assign all calculations in your code to variables.

Comment Your Code

Comments provide information to help users understand the code. Use comments to inform others (particularly the grader) what exactly you are attempting to implement. MATLAB uses % to define a comment. MATLAB ignores all text and characters that follow a % when it executes that line.

- Use comments to describe the purpose of all variables and constants. Include units where applicable.
- Do not use %{ for multi-line comments.
- If you have several lines of related code (i.e., a "code block"), then include an explanatory comment to summarize the functionality of the code block.

  Example:

```
% Plot force vs time and format for technical presentation
plot(test_time,applied_force,'*')
title('Application of Force during Iron Beam Stress Test')
xlabel('Time (s)')
ylabel('Force (N)')
grid on
```

Minimize Hardcoding

To hardcode means to use numbers in calculations instead of variables. Variables make code adaptable and powerful. If you can assign values to variables, then do so. The only hardcoded values in your code should be constants and array indexing values.

For example, you need to write code to determine the area of a triangle, $A = \frac{1}{2}bh$, and the area of a parallelogram, $A = bh$, where the base is 10 units and the height is 8 units. Consider these two pieces of code:

Code A

```
% Calculate area of a triangle and
% parallelogram when given base and
% height
base = 10;  % [m]
height = 8; % [m]
area_tri = 0.5 * base * height; % [m^2]
area_para = base * height;      % [m^2]
```

vs

Code B

```
% Calculate area of a triangle and
% parallelogram when given base and
% height
area_tri = 0.5 * 10 * 8; % [m^2]
area_para = 10 * 8;      % [m^2]
```

Code A follows good programming standards regarding hardcoding. It assigns values to the variables `base` and `height` and then uses those variables to calculate area. Code B calculates the area directly from hardcoded values. Note that 0.5 is hardcoded in both equations because 0.5 is a constant, not a variable.

Hardcoding makes it difficult to adapt or update the code. In Code A, you can update the variables with new values and all calculations using those variables will update accordingly. If you want to update Code B with new values, you must find all the old values and replace each instance with the updated values. That process is inefficient and prone to error.

Another benefit of variables is that they allow conversion from a script to a function. The use of variables in Code A makes it easily translated into a function that can accept user inputs for base and height. Code B does not have that flexibility.

To minimize hardcoding in your script or function:

- Assign all values to variables, unless the values are array indices or equation constants.
- Do not hardcode values for printing or for use in intermediate steps within the code.

Organize Your Script/Function

Organized code is easier to read and understand. The standard template has the following sections:

Initialization: Assign constants or variables, import data, and receive input from a user.

Calculations: Perform calculations and manipulate inputs.

Formatted Text & Figure Displays: Generate results to display, including figures and print statements.

Command Window Outputs: Paste Command Window outputs; use only when required by an assignment.

Analysis: Answer questions assigned in a given problem. These are only present in problem-specific templates.

Use these sections whenever possible. If the problem has a problem-specific template, then you must use the sections provided in that template.

Formatting

Code formatting shows the code structure and makes the code more readable to other programmers. Good formatting uses whitespace and indentation to organize the code.

Whitespace – all types of code

Whitespace helps with readability. Consider these two examples.

Example 1:

```
%% Example 1: poor use of whitespace
% Define ball bearing parameters
bb_diameter=0.25;% ball diameter, cm
steel_density=8.05;% density of steel, g/cm^3
% Calculate the mass of the ball bearing
bb_radius=bb_diameter/2;% ball bearing radius, cm
bb_vol=4*pi*bb_radius^3/3;% ball bearing volume, cm^3
bb_mass=steel_density*bb_vol; % ball bearing mass, g
% Display the mass to the Command Window
fprintf('The ball bearing mass is %.2f grams.\n',bb_mass)
```

Example 2:

```
%% Example 2: good use of whitespace

% Define ball bearing parameters
bb_diameter = 0.25;     % ball diameter, cm
steel_density = 8.05;   % density of steel, g/cm^3

% Calculate the mass of the ball bearing
bb_radius = bb_diameter/2;          % ball bearing radius, cm
bb_vol = 4 * pi * bb_radius^3 / 3;  % ball bearing volume, cm^3
bb_mass = steel_density * bb_vol;   % ball bearing mass, g

% Display the mass to the Command Window
fprintf('The ball bearing mass is %.2f grams.\n', bb_mass)
```

Example 2 uses whitespace to make the code easier to read and understand. To ensure your code uses whitespace appropriately,

- Place an additional line between sections or code blocks to help "group" the code.
- Place a space between operators and operands to improve readability. For example, `y = y - g` is easier to read and quickly understand than `y=y-g`.

Indentation – specific types of code structures

Indentation establishes the structure of decisions and loops. Indent selection structures, repetition structures, and nested structures. The following examples demonstrate proper indentation for various structures.

```
%% Example 3: selection structure with indentation
if r == 1
    new_temp = (temp_matrix(r+1) + temp_matrix(r))/2;
else
    new_temp = (temp_matrix(r-1) + temp_matrix(r))/2;
end

%% Example 4: repetition structure with indentation
new_temp = temp_matrix(n,m);
temp_increase = 1;
while new_temp < boil_point
    new_temp = new_temp + temp_increase;
end
```

```
%% Example 5: nested structure with indentation
if (n < 2) || (m < 2)
    fprintf('Please enter a matrix with dimensions of at least 2x2')
else
    for x = 1:n
        for y = 1:m
            new_temp(x,y) = temp_update(temp_matrix,x,y);
        end
    end
end
```

(These standards were originally adapted in 2012 from CS 15900 Documentation, Programming, and Course Standards.)