# The Purpose of this Document

As applications get larger and more complex, organizations are adopting more formal coding practices including code reviews, automated builds, and continuous integration. Organizations which develop applications involving teams of people writing MATLAB code want to introduce regularity and consistency in their code bases to enhance the quality of their MATLAB code.

The purpose of this document is to describe a set of MATLAB coding guidelines primarily targeted at teams of MATLAB developers contributing to a large application or library. Adoption of the MATLAB Coding Guidelines is optional. Individuals who write code for their own use *may choose* to adopt these guidelines, but no one is compelled to do so.

Several sources of information were used to develop these guidelines. Those sources included

* Coding guidelines written by members of the MATLAB community
* MathWorks internal coding guidelines
* Internal and external MATLAB codebases

A guideline was favored if there was a broad consensus among these sources.

We expect these guidelines to evolve over time. Changes to the guidelines will be driven by feedback from the MATLAB community and will be reflected in subsequent versions of this document.

# Motivations for the Guidelines

The purpose of the guidelines is to allow organizations to introduce regularity and consistency in large MATLAB code bases. Beyond that, there are several important motivations for using the guidelines. Each of the guidelines is motivated by one or more of the following objectives.

* **Readability:** The ease with which code can be read and understood by others, including proper naming, formatting, and structure.
* **Understandability:** The clarity of code in terms of logic, flow, and purpose, making it easy to grasp its function without extensive effort.
* **Maintainability:** The ease with which code can be modified, extended, or debugged over time without introducing errors or unintended behavior.
* **Reusability:** The ability to use code components across different projects or contexts without modification, thereby reducing redundancy and improving developer efficiency.
* **Portability:** The ability of code to run on different platforms or MATLAB versions with minimal or no modification.
* **Testability:** The ease with which code can be tested to verify correctness, including unit testing and automated test execution.
* **Performance:** The degree to which code executes optimally in terms of speed and resource usage, minimizing computation time and memory use.
* **Correctness:** The degree to which code performs its intended function without producing incorrect results.

# Understanding the Coding Guidelines

This document contains two types of guidelines – Rules and Best Practices. Rules must be followed in order to comply with the guidelines. Rule violations are detected by the MATLAB Code Analyzer and identified in the MATLAB Editor and in the Code Analyzer report. Some rule violations are not currently detected but may be detected in a future version of MATLAB. Examples of Rules in the guidelines include:

* Limit variable name length to <= 32 characters
* Limit nesting of loop and conditional statements to 5 levels.

*Best Practices* are guidelines that contain recommendations for improving the quality of your MATLAB code. Following them is optional. Most Best Practices cannot be reliably detected by the Code Analyzer. Examples of Best Practices in the guidelines include:

* Avoid the use of the eval function. The eval function can lead to unexpected code execution especially when using the function with untrusted user input.
* Use the fileparts, fullfile, and filesepfunctions to create or parse filenames in a platform independent way.

The guidelines are organized into categories – Naming, Statements & Expressions, Formatting (use of white space), Code Comments, Function Authoring, Class Authoring, and Error Handling.

## How Guidelines are Documented

The guidelines have been written to be concise, clear, and unambiguous with the goal of making them easy to describe and apply. Every guideline has a table of information like the example below.

**Type:** Rule

**Description:** Limit variable name length to <= 32 characters

**Motivation:**

* Readability: Variable names should be descriptive but excessively long names can reduce readability because they contribute to long lines of code.

**Allowed:**

totalReactivePowerLoss  
actualRipplePassbandFirstBand  
intervalBetweenLaserTransitions

**Not Allowed:**

significancePearsonGravitationalCorrelation  
percentROIAreaContainingPositivePixels

**Detection:** Code Analyzer check naming.variable.maxLength (R2025a)

**History:** Introduced in Version 1.0

The table for each guideline has the following fields.

* **Type:** Whether the guideline is a Rule or Best Practice
* **Description:** A short description of the guideline. Rules are typically limited to a single sentence. Best Practices may be more detailed.
* **Motivation:** The reason(s) this guideline is included.
* **Allowed or Recommended:** Positive examples that obey the Rule or Best Practice.
* **Not Allowed or Not Recommended:** Negative examples that violate the Rule or Best Practice.
* **Detection:** For Rules, how detection is done and the version of MATLAB when detection first became available. For Best Practices, if optional detection is available and how it is done.
* **History:** The version of this document when the guideline was introduced.

## Rules

Rule violations are (or will be) detectable by the MATLAB Code Analyzer. The Code Analyzer is a tool in MATLAB that examines code to identify problems and make recommendations for improvement. It can identify issues related to syntax errors, compatibility, performance, deprecated functionality and much, much more. The Code Analyzer provides [over two thousand checks](https://www.mathworks.com/help/matlab/matlab_env/index-of-code-analyzer-checks.html) for various potential code issues. Those checks can be enabled, disabled, or customized by [creating a local codeAnalyzerConfiguration.json file](https://www.mathworks.com/help/matlab/matlab_env/configure-code-analyzer.html) similar to the example shown below.

The MATLAB Code Analyzer can detect violations for a subset of the Rules listed in this document. Note that some versions of MATLAB may not be able to detect violations for all of the rules. When the Code Analyzer detects a Rule violation, it identifies the issue in both the MATLAB Editor and the Code Analyzer Report. Beginning in R2025a, violations in the Editor are indicated on the right-hand side of the Editor panel as shown in the following screenshot.

This guidelines document is accompanied by a codeAnalyzerConfiguration.json file which implements the checks for the set of Rule violations that can be detected.

The Code Analyzer check for any Rule can be disabled. Consider the example in Figure 2 above. There is a Rule that specifies that function names must be lowercase or lowerCamelCase. You can disable this Rule if you want to turn off checking for function name casing. Most Rules can also be configured. In the case above, you could change the options for function name casing to use a different convention (e.g., UpperCamelCase). The Detection field in the Rule information table provides information about which Code Analyzer check is used to detect violations of the Rule. You can then disable or modify the check in your Code Analyzer Configuration file.

## Best Practices

Best Practices are simply recommendations for writing better MATLAB code. The information provided for a Best Practice is similar to that provided for a Rule. Below is an example Best Practice from the Guidelines.

There are some Best Practices that can (optionally) be detected as Rules by enabling a check in the Code Analyzer. Most of those checks are disabled by default. Information on optional detection, when available, is shown in the **Detection** field of the information for a Best Practice.

**Type:** Best Practice

**Description:** Use cell arrays only to store data of varying classes and/or sizes. Do not use cell arrays to store character vectors as text data. Use a string array instead.

**Motivation:**

* Readability: Using string arrays instead of cell arrays of character vectors improves the readability of the code.
* Performance: String operations are more performant than operations on cell arrays of character vectors.

**Allowed:**

data = {datetime "abc" 123};  
arrays = {rand(1,10) zeros(2,4) eye(5)};  
missions = ["Mercury" "Gemini" "Apollo"];

**Not Allowed:**

missions = {'Mercury' 'Gemini' 'Apollo'};

**Detection:** Optionally by enabling Code Analyzer check DAFCVC (R2024a)

**History:** Introduced in Version 1.0

## Definitions

Several important terms are used in the description of the guidelines. Those terms are defined here.

**Programming interface elements** refers to functions, classes, properties, methods, events, and enumeration members. Table variables and struct fields should be treated as elements of a programming interface if the table or struct is an input or output of a function or method. Otherwise, if a table or struct is used only inside of a single function, method, or script, the table variables and struct fields can be treated like ordinary variables.

**lowercase** is a casing convention for identifiers (names) where the identifier starts with a lowercase letter (a-z) and all subsequent characters are either lowercase letters or numbers. Underscores and other special characters are not allowed. Examples include:

* temperature
* sortrows
* trial27

**lowerCamelCase** is a casing convention for identifiers (names) where the identifier starts with a lowercase letter (a-z) and uses an uppercase letter (A-Z) at the start of each subsequent word. Numbers are allowed after the first letter but underscores and other special characters are not. Examples include:

* totalPowerLoss
* inverseTransformDecompression
* utf8Character

**UpperCamelCase** is a casing convention for identifiers (names) where the identifier uses an uppercase letter (A-Z) at the start of each word. Numbers are allowed after the first letter but underscores and other special characters are not. Examples include:

* KineticEnergy
* Visible
* Unicode16Text

**Leadinguppercase** is a casing convention for identifiers (names) where the identifier starts with a single uppercase letter (A-Z) and is followed by zero or more lowercase letters (a-z) or numbers. Underscores and other special characters are not allowed. Examples include:

* A
* Binverse
* C1

# Naming Guidelines

## General

##### Language

**Type:** Best Practice

**Description:** Use a common language, like English, for MATLAB identifiers when writing code that will be read or used by someone whose native language is different than your own.

**Motivation:**

* Readability: Globally, English is the most common language for programming.

**Recommended:**

initialValue = 4 % variable name  
transmission = DriveTrain % class name

**Not Recommended:**

anfangswert = 4 % Variablenname  
transmission = Transmisia % numele clasei

**Detection:** Not detectable

**History:** Introduced in Version 1.0

##### Names as documentation

**Type:** Best Practice

**Description:** Prefer precise and descriptive names for elements of a programming interface including functions, classes, and methods. Do not use short names for functions or methods unless the meaning is obvious.

**Motivation:**

* Understandability: Descriptive names make it easier to determine the purpose of an element in a programming interface (e.g., a function).

**Recommended:**

initializeTemperature  
findCycles  
rowWiseLast

**Not Recommended:**

calcVal  
nextTemp

**Detection:** Not detectable

**History:** Introduced in Version 1.0

# Statements and Expressions Guidelines

## Loops and Conditionals

##### Nesting of Control Statements

**Type:** Rule

**Description:** Limit nesting of loop and conditional statements to 5 levels.

**Motivation:**

* Readability: Deep nesting adds to the visual complexity of the code.
* Maintainability: Deeply nested code can lead to subtle errors that are difficult to identify and fix.

**Allowed:**

% Find the indices of all primes in matrix A  
% Requires 3 levels of nesting and works for matrices of any dimension  
index = false(size(A));  
for m = 1:numel(A)  
 if (A(m) == 1)  
 continue  
 end  
 if (A(m) == 2) || (A(m) == 3)  
 index(m) = true;  
 continue  
 end  
 if (mod(A(m), 2) == 0) || (mod(A(m), 3) == 0)  
 continue  
 end  
 index(m) = true;  
 maxFactors = floor((sqrt(A(m))+1)/6);  
 for p = 1:maxFactors  
 if (mod(A(m), 6\*p-1) == 0 || mod(A(m), 6\*p+1) == 0)  
 index(m) = 0;  
 end  
 end  
end

**Not Allowed:**

% Find the indices of all primes in matrix A  
% Requires 6 levels of nesting  
index = false(size(A));  
for m = 1:size(A,1)  
 for n = 1:size(A,2)  
 if A(m, n) > 1  
 if (A(m, n) == 2) || (A(m, n) == 3)  
 index(m, n) = true;  
 elseif (mod(A(m, n), 2) ~= 0) && (mod(A(m, n), 3) ~= 0)  
 index(m, n) = true;  
 maxFactors = floor((sqrt(A(m, n))+1)/6);  
 for p = 1:maxFactors  
 if (mod(A(m, n), 6\*p-1) == 0 || ...  
 mod(A(m, n), 6\*p+1) == 0)  
 index(m, n) = false;  
 end  
 end  
 end  
 end  
 end  
end

**Detection:** Code Analyzer check MNCSN (R2023a) **History:** Introduced in Version 1.0