MATLAB Coding Guidelines

Table of Contents

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 `filesep`functions 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 for various potential code issues. Those checks can be enabled, disabled, or customized by creating a local codeAnalyzerConfiguration.json file 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 the screenshot 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

Use of abbreviations

Type: Best Practice

Description: Avoid the use of abbreviated words in the names for elements in a programming interface whenever possible. Use whole words instead. Only use abbreviations that are unambiguous, commonly used within an organization or domain, or easily determined from context.

Motivation:

- Readability: Abbreviations can be ambiguous and prone to misinterpretation. Whole words in names make code easier to read and understand.

Recommended:

```
nextIndex
printError
calculatePressure
```

Not Recommended:

```
nxIdx
prntErr
calcPres
```

Detection: Not detectable

History: Introduced in Version 1.0

Use of acronyms

Type: Best Practice

Description: If an acronym is used in an identifier name, all the letters in that acronym should have the same case. If the identifier's casing Rule calls for the first letter of a word to be lowercase, then all the letters in the acronym should be lowercase. Similarly, if the identifier's casing Rule calls for the first letter of a word to be UPPERCASE, then all the letters in the acronym should be UPPERCASE.

Motivation:

- Readability: Mixed case acronyms are difficult to read and understand.

Recommended:

```
htmlwrite    % for Lowercase
createURL    % for LowerCamelCase
DNAMatch     % for UpperCamelCase
```

Not Recommended:

```
HTMLwrite    % for Lowercase
createUrl    % for LowerCamelCase
DnaMatch     % for UpperCamelCase
```

Detection: Not detectable

History: Introduced in Version 1.0

Avoid shadowing

Type: Best Practice

Description: Avoid naming variables, functions, and classes using the name of an existing function or class on the MATLAB path. Name collisions can lead to “shadowing” which may lead to unexpected or inconsistent behavior.

Motivation:

- Maintainability: Shadowing other functions on the path can lead to unexpected results making code hard to maintain.

Not Recommended:

```
rand
sin
sqrt
```

Detection: Not detectable

History: Introduced in Version 1.0

Variables

Variable name length

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

Variable name style

Type: Best Practice

Description: Prefer descriptive names for variables. Short variables names are permissible when the variable's meaning can be easily determined from the context in which it is used. Such cases include:

- Mathematical expressions
- Short blocks of contiguous code
- Temporary variables or iterators in a loop
- Values widely known to users in a particular domain (Mathematics: phi = golden ratio, Physics: h = Planck's constant)

Do not mix singular and plural forms for variables (e.g., point and points). Instead, consider using a suffix for pluralization. Avoid negated variable names like "isNot" or "notFound".

Motivation:

- Understandability: Well-chosen variable names are unambiguous and avoid confusion over what data the variable contains.

Recommended:

```
apparentMagnitude = 1.2
initialTemperature = 100
x = A\b
e = m*c^2            % c = speed of light
color, colorGroup    % pluralization
```

Detection: Not detectable

History: Introduced in Version 1.0

Variable name casing

Type: Rule

Description: Use lowerCamelCase for descriptive variable names consisting of multiple words. Leadinguppercase can be used for short variable names such as common mathematical symbols.

Motivation:

- Readability: Using a common casing standard can make it easier to distinguish variables from other types of identifiers (e.g., classes).

Allowed:

```
temperature
gibbsFreeEnergy
x = A\b             % A is a matrix
Binverse
```

Not Allowed:

```
KineticEnergy
BTransform
Greenwich_Mean_Time
```

Detection: Code Analyzer check `naming.variable.regularExpression` (R2025a)

History: Introduced in Version 1.0

Functions

Name length for functions and other programming interface elements

Type: Rule

Description: Limit the name length of functions, classes, methods, properties, and other elements of a programming interface to <= 32 characters.

Motivation:

- Understandability: Limiting identifier length will make it easier for others to review and understand your code.

Allowed:

```
reactivePowerLoss
inverseTransformDecompression
optimizeBresenhamConversion
```

Not Allowed:

```
validateBlockPathForModelBlockNormalModeVisibility
plotExactRectangularMembraneConstantLineLoad
```

Detection: Code Analyzer checks (R2025a)

- `naming.class.maxLength`
- `naming.function.maxLength`

- `naming.localFunction.maxLength`
- `naming.method.maxLength`
- `naming.nestedFunction.maxLength`
- `naming.property.maxLength`
- `naming.event.maxLength`
- `naming.enumeration.maxLength`

History: Introduced in Version 1.0

Function name style

Type: Best Practice

Description: Name functions and methods using a verb or verb phrase to convey the action performed. Alternatively, name functions and methods using a noun or noun phrase if the noun describes the thing being created. Use the numeral “2” in the name of a conversion function. Use the prefix “is” or “has” for a function whose primary output is a logical value. Use complementary names for functions with complementary operations (e.g., start/stop, create/destroy, etc.).

Motivation:

- Readability: Well-chosen function names are unambiguous and avoid confusion over what the function does.

Recommended:

```
calculatePower            % Verb phrase: action performed
sankeyPlot                % Noun phrase: thing created
joule2Calorie             % Conversion function
isConfigured, hasValue    % Boolean output
readData, writeData       % Symmetric functions
```

Detection: Not detectable

History: Introduced in Version 1.0

Function name casing

Type: Rule

Description: Use lowerCamelCase or lowercase for function names. For function names that combine multiple words, prefer lowerCamelCase.

Motivation:

- Readability: Using a common casing standard can make it easier to distinguish functions from other types of identifiers (e.g., class methods).

Allowed:

```
initializePressure
inverseTransform
optimizeLayout
solarRadiation
```

Not Allowed:

```
QueryDB
PRINTALL
detect_features
```

Detection: Code Analyzer checks (R2025a)

- `naming.function.casing`
- `naming.localFunction.casing`
- `naming.nestedFunction.casing`

History: Introduced in Version 1.0

Name-Value argument casing

Type: Best Practice

Description: Use UpperCamelCase for the names in Name-Value arguments.

Motivation:

- Readability: Using a common casing standard can make it easier to identify name-value arguments in a function declaration or in a function call.

Recommended:

```
plot(x, y, LineWidth=2)
surf(peaks, FaceColor="interp")
```

Detection: Not detectable

History: Introduced in Version 1.0

Classes

Class name style

Type: Best Practice

Description: If a class represents a thing, use a noun or noun phrase in the name (e.g., PrintServer). If a class implements a set of behaviors or capabilities that other classes can obtain via inheritance, such as a mixin class, use an adjective (e.g., Copyable). Do not put "class" in a class name. Do not use special attributes of the class (e.g., Abstract) in the name.

Motivation:

- Understandability: Well-chosen class names are unambiguous and gives the reader an idea what the class represents.

Recommended:

```
PrintQueue
imageAdapter
pickable
```

Detection: Not detectable

History: Introduced in Version 1.0

Class name casing

Type: Rule

Description: Use UpperCamelCase for the names of classes defined in a namespace. If the class is defined in the MATLAB global name space, use the "Function name casing" Rule above.

Motivation:

- Readability: Using a common casing standard can make it easier to distinguish classes from other identifier types. Using function name casing in the global name space allows users to call a class constructor like an ordinary function.

Allowed:

```
transmitter.OptionsBase    % in a namespace
shapes.Polynomial          % in a namespace
ecgSignal                  % in the global namespace
```

Detection: Not currently detected

History: Introduced in Version 1.0

Method name style

Type: Best Practice

Description: Method names should be either a verb phrase or a noun phrase following the same Best Practice as function names.

Motivation:

- Understandability: Well-chosen method names are unambiguous and avoid confusion over what the method does.

Recommended:

```
modulateSignal
setRollOff
receiveCode
```

Detection: Not detectable

History: Introduced in Version 1.0

Method name casing

Type: Rule

Description: Use lowerCamelCase or lowercase for method names. For method names that combine multiple words, prefer lowerCamelCase.

Motivation:

- Readability: Using a common casing standard can make it easier to identify methods and functions.

Allowed:

```
gpsCoordinates
startRecording
registerDevice
```

Detection: Code Analyzer check `naming.method.casing` (R2025a)

History: Introduced in Version 1.0

Property name style

Type: Best Practice

Description: Use a noun or noun phrase for most property names. Use a verb phrase if a property is a logical value that indicates whether the object does something, or can do something, or has something (e.g., `HasOutputPort`).

Motivation:

- Understandability: Well-chosen property names are unambiguous and tell the user of the class what information the property contains.

Recommended:

```
TextBuffer
CodeTable
HasEncoder
```

Detection: Not detectable

History: Introduced in Version 1.0

Property name casing

Type: Rule

Description: Use UpperCamelCase for property names.

Motivation:

- Readability: Using a common casing standard can make it easier to identify class properties and distinguish them from other identifier types.

Allowed:

```
StartTime
RelativeTolerance
Visible
```

Detection: Code Analyzer check `naming.property.casing` (R2025a)

History: Introduced in Version 1.0

Event name casing

Type: Rule

Description: Use UpperCamelCase for event names.

Motivation:

- Readability: Using a common casing standard can make it easier to identify events of a class.

Allowed:

```
RowSelected
DeviceAdded
```

Detection: Code Analyzer check `naming.event.casing` (R2025a)

History: Introduced in Version 1.0

Namespaces

Namespace name casing

Type: Best Practice

Description: Use short, lowercase names for namespaces.

Motivation:

- Readability: Long namespace names can be hard to read especially with inner namespaces.

Recommended:

```
multivariate
clustering
astrometry.catalogue
```

Detection: Not detectable

History: Introduced in Version 1.0

Namespace content names

Type: Best Practice

Description: Do not use the namespace name in the name of a function, class, enumeration, or inner namespace.

Motivation:

- Readability: Adding the namespace name to its contents doesn't provide any additional information and just makes namespace contents harder to read.

Recommended:

```
learning.findClusters
```

Not Recommended:

```
learning.learningFindClusters
```

Detection: Not detectable

History: Introduced in Version 1.0

Statements and Expressions Guidelines

General

Statements per line

Type: Rule

Description: Do not put multiple statements on a single line.

Motivation:

- Understandability: Multiple statements on a single line makes code more difficult to understand and review.

- Maintainability: It is harder to debug code with multiple statements on the same line.

Not Allowed:

```
fs = 1000; t = 0:1/fs:1; f = 9;
for k = 1:N; dStp(k) = (1/2)*Stp(k-2) + (-2/3)*Stp(k-1); end
```

Detection: Not currently detected

History: Introduced in Version 1.0

Use of literal values

Type: Best Practice

Description: Avoid using literal values in expressions especially when those values appear in multiple places. Similarly, avoid using literal values in a function call. In both cases, use a variable instead.

Motivation:

- Maintainability: Using the same literal value in multiple parts of the code makes the code more difficult to maintain, especially when the literal value needs to be changed.
- Correctness: Failure to change a literal value in multiple locations can lead to unexpected errors.

Recommended:

```
gasConstant = 8.314;
molarVolume = gasConstant*temperature/pressure;

employeeID = "ABF4578";
record = queryEmployees(employeeID);
```

Not Recommended:

```
molarVolume = 8.314*temperature/pressure;

record = queryEmployees("ABF4578");
```

Detection: Not detectable

History: Introduced in Version 1.0

Writing numeric literals

Type: Best Practice

Description: Write floating point literals with a digit (e.g., "0") before the decimal point.

Motivation:

- Readability: Using a zero before the decimal point makes it easier to distinguish "0.1" from "1".

Recommended:

```
x = 0.1;
x = 1.0e-1;
```

Not Recommended:

```
x = .1;
```

Detection: Not currently detected

History: Introduced in Version 1.0

Variables

Global variables

Type: Best Practice

Description: Avoid the use of global variables. Instead, pass variables as arguments to a function.

Motivation:

- Maintainability: Global variables may lead to potential errors that are hard to diagnose.
- Testability: If a global variable is changed in multiple functions, calling those functions in a different order can lead to different results.

Recommended:

```
gravity = 32;
distance = falling((0:0.1:5)', gravity);

function dist = falling(t, gravity)
h = 0.5*gravity*t.^2;
end
```

Not Recommended:

```
global gravity
gravity = 32;
distance = falling((0:0.1:5)');

function dist = falling(t)
global gravity
h = 0.5*gravity*t.^2;
end
```

Detection: Not detected as a guideline. Is detected as a Code Analyzer warning by check GVMIS (R2021b)

History: Introduced in Version 1.0

Persistent variables

Type: Best Practice

Description: Minimize the use of persistent variables. Caching data as a persistent variable between function calls *can be used* to avoid reloading or recomputing a large amount of data on each function call.

Motivation:

- Understandability: Extensive use of persistent variables can make program logic more difficult to understand.

Recommended:

```
function timeZone = getTimeZone(latitude, longitude)
% GETTIMEZONE Determine the time zone for a given
% latitude and longitude
%
% This function loads a mat file containing shapes
% for 439 worldwide time zones. The shapes are used
% to find the time zone that corresponds to a
% specified latitude and longitude. The shapes are
% persistent to avoid loading them on every call.
arguments
    latitude  (1, 1) double {mustBeInRange(latitude, -90, 90)}
    longitude (1, 1) double {mustBeInRange(longitude, -180, 180)}
end

persistent timeZones
if isempty(timeZones)
    load("timeZones.mat", "timeZones")
end

% Rest of calculations here

end
```

Detection: Optionally by enabling Code Analyzer check DAFPV (R2023a)

History: Introduced in Version 1.0

MATLAB Types

Defining structs

Type: Best Practice

Description: Define all fields in a struct in a single block of code. Do not add or remove fields from an existing struct outside of the function in which it was created.

Motivation:

- Maintainability: Structs whose fields change across multiple functions or methods are confusing, error-prone, and hard to maintain.

Recommended:

```
starData = struct(CatalogueNumber=[], IAUName="", ...
    Magnitude=[], RightAscension=[], Declination=[]);

Experiment.Frequency = 1000;
Experiment.Range = [200 400];
Experiment.Harmonic = true;
```

Detection: Not detectable

History: Introduced in Version 1.0

Use of cell arrays

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.

Recommended:

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

Not Recommended:

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

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

History: Introduced in Version 1.0

Expressions

Use of command form

Type: Best Practice

Description: Do not use command syntax in functions or methods. Use of command syntax should be limited to the command line or in scripts.

Motivation:

- Readability: Mixing command form and functional form makes code harder to read and understand.

Recommended:

```
save("catalogue.mat", "starData")   % functional form
hold("on")                          % functional form
```

Not Recommended:

```
load catalogue.mat                  % command form
clear all                           % command form
```

Detection: Optionally by enabling Code Analyzer check DAFCF (R2023a)

History: Introduced in Version 1.0

Parentheses in mathematical and logical expressions

Type: Best Practice

Description: Use parentheses in mathematical and logical expressions to improve readability.

Motivation:

- Understandability: Judicious use of parentheses can make mathematical and logical expressions easier to read and understand.

Recommended:

```
w = (c*d)/(e^f);
y = -(2^2);
m = ((A > 2) && (B < 10)) || (C == 2);
```

Not Recommended:

```
w = c*d/e^f;
y = -2^2;                       % Is this 4 or -4?
m = A > 2 && B < 10 || C == 2;
```

Detection: Not detectable

History: Introduced in Version 1.0

Floating point comparisons

Type: Best Practice

Description: Do not use == or ~= to compare two floating point values.

Motivation:

- Correctness: Use of == or ~= to compare floating point values can lead to logical errors.

Recommended:

```
areEqual = abs(a-b) < 1.0e-4;
areEqual = isapprox(a, b);      % introduced in R2024b
```

Not Recommended:

```
areEqual = (a == b);
```

Detection: Not detectable

History: Introduced in Version 1.0

Creating and parsing filenames

Type: Best Practice

Description: Use the `fileparts`, `fullfile`, and `filesep` functions to create or parse filenames in a platform independent way.

Motivation:

- Portability: These functions allow you to manage file and folder names consistently across any MATLAB supported platform (OS).

Recommended:

```
fileName = fullfile("myfolder", "mysubfolder", "myfile.m");
```

Not Recommended:

```
fileName = "myfolder" + "\" + "mysubfolder" + "\" + "myfile.m";
```

Detection: Not detectable

History: Introduced in Version 1.0

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

Growing arrays inside a loop

Type: Best Practice

Description: Avoid incrementally changing the size of an array inside a loop. Whenever possible, pre-allocate the array immediately before the loop.

Motivation:

- Understandability: Pre-allocating an array makes it explicit how much memory will be needed making the code's behavior easier to predict.
- Performance: Preallocation of arrays provides better performance.

Recommended:

```
x = zeros(1,1000);                  % double array
for k = 2:1000
   x(k) = x(k-1) + 5;
end

objs = createArray(1,5,"myClass"); % array of objects
for k = 1:5
    objs(k).seed = rand;
end
```

Detection: Not detectable as a guideline. Is detected as a Code Analyzer warning by check `AGROW` (R2006b)

History: Introduced in Version 1.0

Iterator modification

Type: Rule

Description: Do not modify a loop iterator inside a for loop.

Motivation:

- Maintainability: The logic of the loop is more predictable, less prone to error, and easier to modify.

Not Allowed:

```
data = [3 -1 4 -2 5 -3 6];
n = length(data);

% Try to remove negative values from the array
for ii = 1:n
    if data(ii) < 0
        data(ii) = [];  % Remove the negative number
        ii = ii - 1;    % Compensate for removed element
    end
end
```

Detection: Code Analyzer check `FXSET` (R2007b)

History: Introduced in Version 1.0

Use of break, continue, & return

Type: Best Practice

Description: Minimize the use of `break`, `continue`, and `return` inside a `for` or `while` loop. Use `break` and `continue` only when it makes the loop more concise or more readable.

Motivation:

- Understandability: The unnecessary use of `break`, `continue`, and `return` can introduce flow changes that make the intent of the loop more difficult to understand.

Recommended:

```
data = [4 -1 6 -3 2 8 -5];
total = 0;
for ii = 1:length(data)          % Sum positive values
    if data(ii) > 0
        total = total + data(ii);
    end
end
```

Not Recommended:

```
data = [4 -1 6 -3 2 8 -5];
total = 0;
for ii = 1:length(data)          % Sum positive values
    if data(ii) < 0
        continue
    end
    total = total + data(ii);
end
```

Detection: Optionally by enabling Code Analyzer checks `DAFCO` (continue), `DAFBR` (break), and `DAFRT` (return)

History: Introduced in Version 1.0

Else as exceptional case

Type: Best Practice

Description: When using `if-else`, put the usual case in the `if` part and the exceptional case in the `else` part.

Motivation:

- Understandability: Makes code logic easier to follow by preventing special cases from obscuring the normal execution path.

Recommended:

```
if size(A, 1) == size(b, 1)
    x = A\b;
else
    error("Size mismatch between A and b");
end
```

Detection: Not detectable

History: Introduced in Version 1.0

Otherwise in switch statements

Type: Best Practice

Description: A `switch` statement should always have an `otherwise` block. If the `otherwise` block is empty, include a comment explaining why no other cases can occur.

Motivation:

Maintainability: An `otherwise` clause will allow you to capture and handle any unexpected cases.

Recommended:

```
switch state
    case "On"
        startDevice()
    case "Off"
        stopDevice()
    otherwise
        error("Unknown state " + state)
end
```

Detection: Not currently detected

History: Introduced in Version 1.0

Making Calls to Functions

Functions with no arguments

Type: Best Practice

Description: Use empty parentheses when calling functions or class methods with no arguments. This will make it clear that a function is being used rather than a variable. Reasonable exceptions include certain common functions like `pi`, `true`, and `false` and certain graphics related functions like `gcf` and `gca`.

Motivation:

- Readability: Using parentheses with no-argument functions makes it easier to distinguish functions from ordinary variables.

Recommended:

```
randomParameters = rng();
currentTime = datetime();
x = 2*pi;
```

Not Recommended:

```
currentFolder = pwd;
```

Detection: Not detectable

History: Introduced in Version 1.0

Ignore unused outputs

Type: Best Practice

Description: Use the tilde character (~) to ignore unused, leading outputs from a function.

Motivation:

- Understandability: Use of the tilde character is a clear sign to the reader that certain outputs will not be used subsequently.

Recommended:

```
[~, ~, V] = svd(A);
```

Detection: Not detected as a guideline. Is detected as a Code Analyzer warning by check `ASGLU` (R2010b)

History: Introduced in Version 1.0

Use of name-value arguments

Type: Best Practice

Description: Use `Name=Value` syntax (R2021a) when passing Name-Value arguments to a function.

Motivation:

- Readability: `Name=Value` syntax makes it easier to associate names with values in a long list of optional values.

Recommended:

```
plot(x, y, Color="g", LineWidth=3, Marker="*")
```

Not Recommended:

```
plot(x, y, "Color", "g", "LineWidth", 3, "Marker", "*")
```

Detection: Not detectable

History: Introduced in Version 1.0

Functions to Avoid

eval function

Type: Best Practice

Description: 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.

Motivation:

- Maintainability: Use of `eval` can lead to unintended behavior.
- Reusability: A call to `eval`may be safe in a given context but can create security or other concerns in a different context.

Recommended:

```
numArrays = 10;
A = cell(numArrays,1);
for ii = 1:numArrays
    A{ii} = magic(ii);
end
```

Not Recommended:

```
numArrays = 10;
for ii = 1:numArrays
    eval("A" + int2str(ii) + " = magic(ii)");
end
```

Detection: Optionally using custom Code Analyzer check for existing functions

History: Introduced in Version 1.0

Workspace functions

Type: Best Practice

Description: Avoid the use of functions which manipulate a workspace outside the current context. The `evalin` and `assignin`functions should not be used as a replacement for function outputs. Variables in the base workspace should not be used as if they are global variables.

Motivation:

- Maintainability: Modifying variables in another context may lead to subtle and unexpected errors.

Recommended:

```
% Update configuration safely
config = updateConfig(config, "simulationSpeed", 2.5);

function config = updateConfig(config, param, value)
if isfield(config, param)
    config.(param) = value;  % Modify known parameters
else
    error("Invalid config parameter: %s", param);
end
end
```

Not Recommended:

```
% Update configuration unsafely
updateConfig(config, "simulationSpeed", 2.5);

function updateConfig(config, param, value)
command = config + "." + param + " = "  + num2str(value);
evalin("base", command);
end
```

Detection: Optionally using custom Code Analyzer check for existing functions

History: Introduced in Version 1.0

Path functions

Type: Best Practice

Description: Minimize the use of `cd`, `addpath`, and `rmpath`to modify the current folder or the MATLAB search path within a function or method. If you must use these functions, reset the current folder and path before exiting the function.

Motivation:

- Reusability: Functions that manipulate the current folder and path may not work properly in other contexts.
- Maintainability: Current folder and path changes may lead to subtle changes in behavior.

Recommended:

```
function output = myFunction(input)
oldPath = path;
c = onCleanup(@()path(oldPath));

newFolder = "C:\MATLAB\mydir";
addpath(genpath(newFolder));

% Perform some calculations
end
```

Detection: Optionally using custom Code Analyzer check for existing functions

History: Introduced in Version 1.0

Formatting Guidelines

Use of Spaces

Spaces vs. tabs

Type: Rule

Description: Use spaces rather than tabs to add white space.

Motivation:

- Readability: The tab character may be interpreted differently in different editors or on different platforms.

Allowed:

```
for ii = 1:m
□□□□for jj = 1:n
```

```
□□□□□□□□A(ii, jj) = ii + jj;
□□□□end
end
```

Not Allowed:

```
for ii = 1:m
<tab>for jj = 1:n
<tab><tab>A(ii, jj) = ii + jj;
<tab>end
end
```

Detection: Not currently detected. Applied by default Editor preference setting.

History: Introduced in Version 1.0

Indentation

Type: Rule

Description: Use 4 spaces per indent level.

Motivation:

- Readability: A consistent indent level makes code easier to read.

Allowed:

```
for ii = 1:m
□□□□for jj = 1:n
□□□□□□□□A(ii, jj) = ii + jj;
□□□□end
end
```

Detection: Not currently detected. Applied by default Editor preference setting.

History: Introduced in Version 1.0

Spaces inside grouping operators

Type: Rule

Description: Do not add spaces immediately after an opening parenthesis, square bracket, or curly brace. Do not add spaces immediately before a closing parenthesis, square bracket, or curly brace.

Motivation:

- Readability: In most cases, extra spaces do not enhance readability. They just make lines longer.

Allowed:

```
a = sin(exp(1));
A = [1 0; 0 1];
B = {12 "def"};
```

Not Allowed:

```
a = sin(exp( 1) );
A = eig([ 2 3; 4 5 ]);
B = A( (A > 2) & (A < 5) );
```

Detection: Not currently detected

History: Introduced in Version 1.0

Spaces after commas, & semicolons

Type: Rule

Description: Put spaces after commas or semicolons except at the end of a line.

Motivation:

- Readability: Spaces after commas and semicolons make code lines easier to read.

Allowed:

```
A = [6 4 23 -3; 9 -10 4 11; 2 8 -5 1];
S = std(A, 2, "omitmissing");
```

Not Allowed:

```
B = [1 2 3;4 5 6];
T = rand(5,4,3,"single");
```

Detection: Not currently detected

History: Introduced in Version 1.0

Spaces at the end of lines

Type: Rule

Description: Do not put extra whitespace at the end of lines.

Motivation:

- Maintainability: Extra whitespace can create diff and merge conflicts.

Not Allowed:

```
first = 1;□□
```

Detection: Not currently detected

History: Introduced in Version 1.0

Spaces around assignment operator

Type: Rule

Description: Use one space on either side of the assignment (=) operator in an assignment statement. Do not use spaces around = when using `Name=Value` syntax to specify optional arguments to a function.

Motivation:

- Readability: Spaces around the assignment operator make statements easier to read, especially when the left operand is a variable with a long name and/or the right operand is a complex expression. No space around = when using `Name=Value` syntax makes the grouping of named argument pairs easier to identify.

Allowed:

```
initialValue = 3.2;
plot(x, y, LineWidth=3);
```

Not Allowed:

```
apparentMagnitude=1.2;
plot(x, y, LineWidth = 3);
```

Detection: Not currently detected

History: Introduced in Version 1.0

Spaces around relational operators

Type: Rule

Description: Use one space on either side of the relational operators (<, <=, ==, ~=, >, >=).

Motivation:

- Readability: Spaces around a relational operator makes statements easier to read, especially when the operands are long, complex expressions.

Allowed:

```
if (x <= 3) || (x >= 5)
```

Not Allowed:

```
A(A>2)
```

Detection: Not currently detected

History: Introduced in Version 1.0

Spaces around logical operators

Type: Rule

Description: Use one space on either side of the logical (&, &&, |, ||) operators.

Motivation:

- Readability: Spaces around a logical operator makes statements easier to read, especially when the operands are long, complex expressions.

Allowed:

```
A(A & ~mod(A, 2))
```

Not Allowed:

```
C = A|B
```

Detection: Not currently detected

History: Introduced in Version 1.0

Spaces around colon operator

Type: Rule

Description: Do not use spaces around the colon operator or in the operands on either side of the colon operator.

Motivation:

- Readability: Indexing expressions are easier to read without any spaces.

Allowed:

```
evenNumbers = 2:2:10;
B = A(2:end-1);
for ii = first+1:last-1
```

Not Allowed:

```
evenNumbers = 2 : 2 : 10;
B = A(2 : end - 1);
for ii = first + 1:last - 1
```

Detection: Not currently detected

History: Introduced in Version 1.0

Spaces around multiply, divide, & exponent operators

Type: Rule

Description: Do not put spaces around the multiply, divide, or exponent operators (`*` `.*` `/` `./` `\` `.\` `^` `.^`).

Motivation:

- Readability: These operators are written without spaces around them in mathematical expressions.

Allowed:

```
sin(c)/exp(d)
A.^2
```

Not Allowed:

```
(a+b) * (c / d)
3 ^ 2
```

Detection: Not currently detected

History: Introduced in Version 1.0

Spaces around plus and minus operators

Type: Rule

Description: When an expression appears on the right-hand side of an assignment statement, use spaces around the plus and minus operators that operate on the main terms of that expression. Put no spaces around plus or minus in other places, such as within grouped terms, as argument to functions, or as indexing operands.

Motivation:

- Readability: Judicious use of spacing around the plus and minus operators make mathematical expressions more readable.

Allowed:

```
x = 1 + sin(pi) - cos(pi);
z = (a+b) + exp(c+d);
r = xhex + mod(k-1, 2)*D + D*2*j - (radius+3)/2;
```

Not Allowed:

```
v = exp(a + b);
w = y > x + 1;
du(np) = -upap(n, a(np), w)+meru(a(np), c0)+plterm;
```

Detection: Not currently detected

History: Introduced in Version 1.0

Spaces after unary operators

Type: Rule

Description: Do not use spaces after the unary operators +, -, or ~.

Motivation:

- Readability: Extra spaces after a unary operator makes code more difficult to read.

Allowed:

```
x = -1;
```

Not Allowed:

```
A = [- 1 1];
y = ~ x;
```

Detection: Not currently detected

History: Introduced in Version 1.0

Use of Blank Lines

Around related blocks of code

Type: Best Practice

Description: Use a single blank line to separate sections of code that perform distinct tasks or are logically related.

Motivation:

- Readability: Breaking up logical sections of code with blank lines can make logic and program flow easier to understand.

Recommended:

```
idx = solarElevation <= 0;
solarElevation(idx) = [];
solarAzimuth(idx) = [];

airMass = calculateAirMass(solarElevation);
solarRadiation = 1.353.*0.7.^(airMass.^0.678);

t1 = cosd(solarElevation).*sind(panelTilt).* ...
     cosd(180-solarAzimuth);
t2 = sind(solarElevation).*cosd(panelTilt);
panelRadiation = solarRadiation.*max(0, t1+t2);
```

Detection: Not detectable

History: Introduced in Version 1.0

Around local functions

Type: Rule

Description: Use one blank line to separate local function declarations.

Motivation:

- Readability: Blank lines clearly mark where functions begin and end.

Allowed:

```
function Tc = centigrade2Fahrenheit(Tf)
Tc = 9*Tf/5 + 32;
end

function Tk = centigrade2Kelvin(Tc)
```

```
Tk = Tc + 273.15;
end
```

Not Allowed:

```
function Tc = centigrade2Fahrenheit(Tf)
Tc = 9*Tf/5 + 32;
end
function Tk = centigrade2Kelvin(Tc)
Tk = Tc + 273.15;
end
```

Detection: Not currently detected

History: Introduced in Version 1.0

Around methods

Type: Rule

Description: Use one blank line to separate `method` declarations in a `classdef` file.

Motivation:

- Readability: Blank lines clearly mark where one method ends and the next one begins.

Allowed:

```
methods
    function signal = Signal(data, freq)
        signal.Frequency = freq;
        signal.Data = data;
    end

    function signal = removeTrend(signal, order)
        for ii = 1:numel(signal)
            signal(ii).Data = detrend(signal(ii).Data, order);
        end
    end
end
```

Not Allowed:

```
methods
    function signal = Signal(data, freq)
        signal.Frequency = freq;
        signal.Data = data;
    end
    function signal = removeTrend(signal, order)
```

```
        for ii = 1:numel(signal)
            signal(ii).Data = detrend(signal(ii).Data, order);
        end
    end
end
```

Detection: Not currently detected

History: Introduced in Version 1.0

Around method blocks

Type: Rule

Description: Use one blank line to separate `method` blocks in a `classdef` file.

Motivation:

- Readability: Blank lines clearly mark where `method` blocks with specific attributes begin and end.

Allowed:

```
methods
    function signal = Signal(data, freq)
        signal.Frequency = freq;
        signal.Data = data;
    end
end

methods (Access = Protected)
    function signal = removeTrend(signal, order)
        for ii = 1:numel(signal)
            signal(ii).Data = detrend(signal(ii).Data, order);
        end
    end
end
```

Not Allowed:

```
methods
    function signal = Signal(data, freq)
        signal.Frequency = freq;
        signal.Data = data;
    end
end
methods (Access = Protected)
    function signal = removeTrend(signal, order)
        for ii = 1:numel(signal)
```

```
            signal(ii).Data = detrend(signal(ii).Data, order);
        end
    end
end
```

Detection: Not currently detected

History: Introduced in Version 1.0

Around property blocks

Type: Rule

Description: Use one blank line to separate `property` blocks in a `classdef` file.

Motivation:

- Readability: Blank lines clearly mark where `property` blocks with specific attributes begin and end.

Allowed:

```
properties
    Frequency
    Data
end

properties (Dependent)
    Time
end
```

Not Allowed:

```
properties (Access = public)
    MinimumRadius = 1
    MaximumIterations = 1
end
properties (Access = private)
    SigmaXY
end
properties (Dependent)
    RegionOfInterest
end
```

Detection: Not currently detected

History: Introduced in Version 1.0

At the beginning or end of a file

Type: Rule

Description: Do not put extra blank lines at the top or bottom of a script, function, or classdef file.

Motivation:

- Maintainability: Extra blank lines can create diff and merge conflicts.

Detection: Not currently detected

History: Introduced in Version 1.0

Lines in Code Files

Line length

Type: Rule

Description: Code and comment lines should be <= 120 characters.

Motivation:

- Readability: Shorter code lines are easier to read and minimize horizontal scrolling.

Allowed:

```
solarAzimuth = acos((sin(solarDeclination)*cos(latitude) - ...
   cos(solarDeclination)*sin(latitude)*cos(angle))/cos(elevation));

term1 = sin(solarDeclination)*cos(latitude);
term2 = cos(solarDeclination)*sin(latitude)*cos(angle)
solarAzimuth = acos((term1 - term2)/cos(elevation));

% For each slice we will calculate a sequence of numbers by repeated
% application of the operator. We will stop looking when the
% last element of the current slice is below the lower limit.
```

Detection: Code Analyzer check `LLMNC` (R2023a)

History: Introduced in Version 1.0

Line breaks

Type: Best Practice

Description: Split long lines to maximize readability. When breaking a long line, consider breaking the line after a comma, after a space, or at a binary operator.

Motivation:

- Readability: Using a consistent strategy for splitting long lines will make them easier to read and make the logic easier to understand.

Recommended:

```
planetTemperature = estimateTemperature(starTemperature, ...
    starSolarRadii, planetEarthRadii, planetOrbitalAxis);

index = (im(:,:,1) > h & (im(:,:,2)+im(:,:,3)) < l) | ...
    (im(:,:,2) > h & (im(:,:,3)+im(:,:,1)) < l) | ...
    (im(:,:,3) > h & (im(:,:,1)+im(:,:,2)) < l);
```

Detection: Not detectable

History: Introduced in Version 1.0

Code Comments Guidelines

General

Language

Type: Best Practice

Description: Use a common language, like English, for comments in code that will be read or used by someone whose native language is different than your own.

Motivation:

- Readability: Use of English allows MATLAB users outside your home country to read and understand code comments.

Recommended:

```
% Get the exponentiated values for nDigit numbers
% Initialize the first number of the first slice
```

Not Recommended:

```
% nDigit の数値のべき乗された値を取得します
% Inizializza il primo numero della prima fetta
```

Detection: Not detectable

History: Introduced in Version 1.0

Comment symbol

Type: Rule

Description: Use at least one space after the comment symbol “%”. Use “%%” to define a new section.

Motivation:

- Readability: The extra space after the comment symbol increases readability.

Allowed:

```
% Find increasing, decreasing edge line indices
```

Detection: Not currently detected

History: Introduced in Version 1.0

Placement and Indentation

H1 and help content and placement

Type: Best Practice

Description: Place the function H1 line immediately after the function declaration and before the arguments block. The H1 line should provide a brief description of what the function does. Help text that follows the H1 line should provide the information the user needs to use the function including the syntax, a description of inputs and outputs, and any side effects.

Motivation:

- Maintainability: Well written function help makes functions easier to use and modify as needed.

Recommended:

```
function b = rowWiseLast(A)
% rowWiseLast finds the last non-zero element in each row
% Syntax:
%   rowWiseLast(A)
% Inputs:
%   A  Input matrix
% Outputs
%   b  Vector containing the last non-zero value in each row
%      of A. Note that b(i) = 0 if A(i,:) is all zeros.
arguments
    A (:, :) double
end

m = size(A, 2);
[~, loc] = max(fliplr(logical(A)), [], 2);
idx = m + 1 - loc;
b = A(sub2ind(size(A), 1:size(A,1), idx'))';
end
```

Detection: Not detectable

History: Introduced in Version 1.0

Internal comment placement

Type: Best Practice

Description: Comments should be placed just before the code they are meant to explain. Short comments can be placed at the end of a line.

Motivation:

- Maintainability: Putting comments close to code makes the code easier to understand and modify as needed.

Recommended:

```
function test = isprime(n)
% isprime(n) returns true if n is prime and false otherwise.
test = true;

% Test special cases for small values of n
if (n == 2 || n == 3)
    return
end

% Handle non-positive numbers and multiples of 2 or 3
if (n <= 1 || mod(n, 2) == 0 || mod(n, 3) == 0)
    test = false;
    return
end

% Limit potential factors to numbers less than sqrt(n).
limit = floor(sqrt(n));
```

```
% Iterate through numbers of the form 6k ± 1
for ii = 5:6:limit
    if (mod(n, ii) == 0 || mod(n, ii+2) == 0)
        test = false;
        return
    end
end
end
```

Detection: Not detectable

History: Introduced in Version 1.0

Comment indentation

Type: Rule

Description: Indent H1 and help lines at the beginning of a function using the same indent level as the function declaration. Otherwise, indent comment lines at the same level as the lines of code that immediately follow.

Motivation:

- Readability: Consistent indentation makes it easier to associate comments with related code.

Allowed:

```
function factors = primeFactors(n)
% primeFactors(n) returns all prime factors
% Inputs
%   n:  Number to factorize
% Outputs
%   factors: List of prime factors
arguments
    n (1, 1) double {mustBeInteger, mustBeGreaterThan(n, 1)}
end

% Return the number if it is prime or 1
if isprime(n) || n == 1
    factors = n;
    return
end

% Find all the prime numbers up to number/2
allPrimes = [];
for ii = 2:floor(n/2)
    if isprime(ii)
```

```
        % Add the new prime number to the list
        allPrimes = [allPrimes ii];
    end
end
nPrimes = numel(allPrimes);

% Loop through all the primes
factors = [];
quotient = n;
for ii = 1:nPrimes
    currentPrime = allPrimes(ii);
    % Divide by the current prime until remainder is not zero
    while mod(quotient, currentPrime) == 0
        % Add the current prime to the list of factors
        factors = [factors currentPrime];
        quotient = quotient/currentPrime;
    end
end
end
```

Detection: Not currently detected

History: Introduced in Version 1.0

Function Authoring Guidelines

General

File name

Type: Rule

Description: A function file name should be the same as the name of the top-level function.

Motivation:

- Understandability: It is confusing if the function name listed in the file does not match the name used to call that function.

Detection: Code Analyzer check `FNDEF` (R14)

History: Introduced in Version 1.0

Terminate functions with the end keyword

Type: Rule

Description: End all functions with the end keyword

Motivation:

- Readability: Explicitly marking the end of a function makes the code easier to read, especially in files with multiple functions or nested functions.

Allowed:

```
function Tc = centigrade2Fahrenheit(Tf)
Tc = 9*Tf/5 + 32;
end

function Tk = centigrade2Kelvin(Tc)
Tk = Tc + 273.15;
end
```

Not Allowed:

```
function Tc = centigrade2Fahrenheit(Tf)
Tc = 9*Tf/5 + 32;
function Tk = centigrade2Kelvin(Tc)
Tk = Tc + 273.15;
```

Detection: Not currently detected

History: Introduced in Version 1.0

Reset global state

Type: Best Practice

Description: Use caution when changing MATLAB global or system state. Restore the state when a function or method exits. If the modified state is not reset to the original values, subsequent code may behave incorrectly.

Motivation:

- Reusability: Functions should be self-contained and not depend on or leave behind external state changes.

- Testability: Functions that reset their state are easier to test in isolation when they start and end with a clean slate.

Recommended:

```
newFolder = fullfile("C:\", "MATLAB", "mydir");
oldPath = path();
c = onCleanup(@()path(oldPath));
addpath(genpath(newFolder));

% Perform some calculations
```

Detection: Not detectable

History: Introduced in Version 1.0

Local functions

Type: Best Practice

Description: A function used by only one other function or script should be written as a local function in the same file. Keep local functions simple. If a function needs to be independently tested, put it in its own file.

Motivation:

- Maintainability: A local function can keep related functionality in a single file, making it easier to read and maintain.

Recommended:

```
function factor = equationOfState(gas, temperature, pressure)
% Look up critical properties for gas
[criticalTemperature, criticalPressure] = lookupCritical(gas);

% Calculate compressibility factor
factor = vanDerWaals(temperature, pressure, ...
    criticalTemperature, criticalPressure);
end

function Z = vanDerWaals(T, P, Tcritical, Pcritical)
R = 8.3145;                                     % gas constant
a = 27*(R*Tcritical)^2/(64*Pcritical);
b = R*Tcritical/(8*Pcritical);

coefficients = [1 -(b + R*T/P) a/P -a*b/P];
zeros = roots(coefficients);
V = max(zeros(logical(imag(zeros) == 0)))       % molar volume
Z = P*V/(R*T);                                  % compressibility
end
```

Detection: Not detectable

History: Introduced in Version 1.0

Nested functions

Type: Best Practice

Description: Limit the use of nested functions. Nested functions can almost always be replaced by a local function.

Motivation:

- Maintainability: Nested functions have access to variables in their parent function, which can lead to unintended side effects.
- Readability: Nested functions are defined inside another function making the main function harder to read

Not Recommended:

```
function factor = equationOfState(gas, temperature, pressure)
% Look up critical properties for gas
[Tcritical, Pcritical] = lookupCritical(gas);

% Calculate compressibility factor
factor = vanDerWaals(temperature, pressure);

    function Z = vanDerWaals(T, P)
        R = 8.3145;                                % gas constant
        a = 27*(R*Tcritical)^2/(64*Pcritical);
        b = R*Tcritical/(8*Pcritical);

        coefficients = [1 -(b + R*T/P) a/P -a*b/P];
        zeros = roots(coefficients);
        V = max(zeros(logical(imag(zeros) == 0)))  % molar volume
        Z = P*V/(R*T);                             % compressibility
    end

end
```

Detection: Optionally by enabling Code Analyzer check DAFNF

History: Introduced in Version 1.0

Anonymous functions

Type: Best Practice

Description: Keep anonymous functions simple and readable. When possible, keep the definition and use of the anonymous function together in the code.

Motivation:

- Maintainability: Code is easier to maintain if anonymous functions are simple and defined where they are used. If a function becomes too long or is used multiple times, it can be converted into a local function.

Recommended:

```
equation = @(x) x^2 + log(x);
root = fzero(equation, 1);
```

Detection: Not detectable

History: Introduced in Version 1.0

Refactoring

Type: Best Practice

Description: Do not repeat blocks of code in a function. Refactor those statements into a new function or local function.

Motivation:

- Maintainability: Code changes can be made in a single place.
- Reusability: A new function can be used in other parts of an application.

Recommended:

```
function dS = entropyChange(process, gas, Vi, Vf, Ti, Tf)
R = 8.314;                      % Gas constant in J/(mol·K)
[Cp, Cv] = heatCapacity(gas);   % Heat capacities for gas

% Compute entropy for process type
switch process
    case "isothermal"
        dS = computeEntropy(R, Vi, Vf);
    case "constant-volume"
        dS = computeEntropy(Cv, Ti, Tf);
    case "constant-pressure"
        dS = computeEntropy(Cp, Ti, Tf);
    otherwise
        error("Invalid process type.");
end
end
```

```
% Common code is factored out into a separate function
function dS_val = computeEntropy(coeff, Yi, Yf)
if (Yi <= 0) || (Yf <= 0)
    error("Inputs must be positive.");
end
dS_val = coeff*log(Yf/Yi);
end
```

Not Recommended:

```
function dS = entropyChange(process, gas, Vi, Vf, Ti, Tf)
R = 8.314;                          % Gas constant in J/(mol·K)
[Cp, Cv] = heatCapacity(gas);       % Heat capacities for gas

% Common code is repeated 3 times
switch process
    case "isothermal"
        if (Vi <= 0) || (Vf <= 0)
            error("Volumes must be positive.");
        end
        dS = R*log(Vf/Vi);
    case "constant-volume"
        if (Ti <= 0) || (Tf <= 0)
            error("Temperatures must be positive.");
        end
        dS = Cv*log(Tf/Ti);
    case "constant-pressure"
        if (Ti <= 0) || (Tf <= 0)
            error("Temperatures must be positive.");
        end
        dS = Cp*log(Tf/Ti);
    otherwise
        error("Invalid process type.");
end
end
```

Detection: Not detectable

History: Introduced in Version 1.0

Inputs

Number of function inputs

Type: Rule

Description: Limit the number of input arguments in a function declaration to 6. Use name-value arguments for optional information. Multiple name-value arguments can be represented as a single argument in the function declaration.

Motivation:

- Readability: Functions with fewer arguments are easier to read, understand, and use.

Allowed:

```
function p = calculatePotential(x, y, z, dx, dy, dz)

function A = getSamples(interval, data, opt)
arguments
    interval (1, 1) double {mustBePositive}
    data     (1, :) double
    opt.Port (1, 1) double
    opt.Rate (1, 1) double
    opt.Type (1, 1) string {mustBeMember(opts.Type, ["A","D"])}
end
```

Not Allowed:

```
function dP = dipPotential(xy, Q, D, R, r0, a, b, Dx, Dy, Nxy)

function readBonay(T, diam, rh, phi, S, phi, visc, zr, kn)
```

Detection: Code Analyzer check `FCNIL` (R2023a)

History: Introduced in Version 1.0

Argument validation

Type: Best Practice

Description: Validate input arguments for functions that are intended to be part of an external, user-facing programming interface. Use an arguments block to do validation.

Motivation:

- Maintainability: Argument validation ensures that future developers understand the function's requirements.

Recommended:

```
function [elevation, azimuth] = position(latitude, longitude, date)
arguments
    latitude  (1, 1) double   {mustBeInRange(latitude, -90, 90)}
    longitude (1, 1) double   {mustBeInRange(longitude, -180, 180)}
    date      (1, 1) datetime = datetime("today")
end
```

Detection: Not detectable

History: Introduced in Version 1.0

Name-Value arguments

Type: Best Practice

Description: Avoid the use of `varargin` to handle name-value arguments. Instead use an `arguments` block with optional arguments and name/value pairs.

Motivation:

- Readability: An `arguments` block explicitly defines the expected input types, sizes, and constraints, making the function's behavior clearer.
- Maintainability: Extending and modifying the function is simpler with structured input handling.

Recommended:

```
function sankey = sankeyPlot(data, options)
arguments
    data                 (:, :) double
    options.ColorOrder   (:, 3) double = colororder
    options.Transparency (1, 1) double = 0.5
    options.TextSize     (1, 1) double = 10
end
```

Detection: Not detected as a guideline. Use of `varargin` may be detected by enabling Code Analyzer check `DAFVI` (R2023b)

History: Introduced in Version 1.0

Element-wise functions

Type: Best Practice

Description: Write element-wise functions so that they work with any array shape. Outputs which correspond to an input of a particular shape should have the same shape.

Motivation:

- Maintainability: Ensures consistent behavior across element-wise functions.

Recommended:

```
function squaredArray = squared(array)
squaredArray = zeros(size(array));
for ii = 1:numel(array)
    squaredArray(ii) = array(ii)^2;
end
squaredArray = reshape(squaredArray, size(array));
end
```

Detection: Not detectable

History: Introduced in Version 1.0

Outputs

Number of function outputs

Type: Rule

Description: Limit the number of output arguments in a function declaration to 4.

Motivation:

- Readability: Too many outputs can make it difficult to understand what the function is supposed to do.
- Reusability: A function with few, well-defined outputs is more flexible and reusable.

Allowed:

```
function [tone, left, right] = psdTone(Pxx, F, rbw, freq)
```

Not Allowed:

```
function [msg, nfft, Fs, w, p, flag, rtf] = psdchk(P, x, y)
```

Detection: Code Analyzer check `FCNOL` (R2023a)

History: Introduced in Version 1.0

Behavior changes with varargout

Type: Best Practice

Description: Do not change the meaning of an output when the number of outputs change.

Motivation:

- Maintainability: With additional outputs, the logic for nargout becomes more complicated making it difficult to modify the function's behavior.
- Testability: Testing becomes more complicated as test cases will have to be written for multiple scenarios.

Not Recommended:

```
function varargout = computeStatistics(data)
nargoutchk(1,2);

if (nargout == 1)
    varargout{1} = mean(data);
else
    varargout{1} = std(data);     % First output is different
    varargout{2} = mean(data);
end
end
```

Detection: Not detectable

History: Introduced in Version 1.0

Comma separated outputs

Type: Rule

Description: Use commas to separate outputs in a function declaration.

Motivation:

- Readability: Using commas to separate outputs in a function call clearly indicates each distinct output.

Allowed:

```
function [tone, left, right] = psdTone(Pxx, F, rbw, freq)
```

Detection: Code Analyzer check `NCOMMA` (R2012b)

History: Introduced in Version 1.0

Class Authoring Guidelines

General

File name

Type: Rule

Description: A `classdef` file name should be the same as the name of the class.

Motivation:

- Maintainability: Matching the class name and file name makes it easier to locate the class definition when debugging or modifying code.

Detection: Code Analyzer check `MCFIL` (R2008a)

History: Introduced in Version 1.0

Handle vs value classes

Type: Best Practice

Description: Prefer value classes to handle classes. Use handle classes to represent an object whose state can change without changing its identity. Consider using handle for classes that

- represent physical or unique objects like hardware devices or files
- represent visible objects like graphics components
- define objects in a relational data structure like a list or a tree

Motivation:

- Understandability: Value classes are easier to understand because different parts of the program cannot change the same data.
- Maintainability: The state of handle classes can be changed in multiple parts of the code making it harder to maintain the code.

Recommended:

```
classdef EarthquakeData
classdef quaternion

classdef BluetoothReceiver < handle     % hardware device
classdef SankeyPlot < handle            % graphics object
```

Detection: Not detectable

History: Introduced in Version 1.0

Combine property blocks

Type: Best Practice

Description: Avoid multiple `property` blocks with the same attributes unless they are used to logically group related class properties.

Motivation:

- Maintainability: Redundant `property` blocks make code harder to maintain. Changing an attribute means editing multiple `property` blocks.

Recommended:

```
classdef OpticFlow
    properties (Access = public)
        MinimumRadius = 1
        MaximumIterations = 1
    end

    properties (Access = private)
        SigmaXY
    end

    properties (Dependent)
        RegionOfInterest
    end
end
```

Detection: Not detectable

History: Introduced in Version 1.0

Combine method blocks

Type: Best Practice

Description: Avoid multiple `method` blocks with the same attributes unless they are used to logically group related class methods.

Motivation:

- Maintainability: Redundant `method` blocks make code harder to maintain. Changing an attribute means editing multiple `method` blocks.

Recommended:

```
classdef OpticFlow
    methods (Access = public)
        function obj = OpticFlow(varargin)
            % Code for OpticFlow
        end

        function obj = advanceFlow(obj)
            % Code for advanceFlow
        end
    end

    methods (Static)
        function [r, w] = logCoordinates(region)
            % Code for logCoordinates
        end

        function array = getRegion(region)
            % Code for getRegion
        end
    end
end
```

Detection: Not detectable

History: Introduced in Version 1.0

Sealed classes

Type: Best Practice

Description: Use the `Sealed` class attribute if you do not intend people to use your class as a superclass. Only leave classes unsealed when the class is designed to be extended by others.

Motivation:

- Maintainability: Sealed classes can be modified over time without risk of becoming incompatible with subclasses.

Recommended:

```
classdef (Sealed) DontSubclass
```

Detection: Not detectable

History: Introduced in Version 1.0

Properties

Property Access Control

Type: Best Practice

Description: Make property access as restrictive as necessary to support the needs of the user of the class. This can make it easier to evolve the design of the class over time. For example, only allow set access when a property need to be set by a user of the class.

Motivation:

- Understandability: By limiting how a property can be accessed, you make the behavior and intent of the class more explicit.
- Maintainability: Changing the internal structure of the class is less likely to affect users of the class because they interact with the class through a public interface.

Recommended:

```
classdef OpticFlow
    properties (Access = public)
        maximumIterations = 1
    end

    properties (SetAccess = private)
        MinimumRadius = 1
    end

    properties (Access = private)
        SigmaXY
    end
end
```

Detection: Not detectable

History: Introduced in Version 1.0

Validation

Type: Best Practice

Description: Avoid using set methods purely for validation. Use property validation syntax instead. If you have a situation where the property needs to be validated and transformed it may be more efficient to use a set method.

Motivation:

- Maintainability: Property validation ensures that future developers and users understand the requirements for the class properties.

Recommended:

```
classdef Rectangle
    properties
        Origin (1,2) double {mustBeReal}
        Width  (1,1) double {mustBeReal, mustBeNonnegative}
        Height (1,1) double {mustBeReal, mustBeNonnegative}
    end
end
```

Not Recommended:

```
classdef Rectangle
    properties
        Origin
        Width
        Height
    end

    methods
        function obj = set.Origin(obj, point)
            validateattributes(point, {'double'}, ...
                {'size', [1 2], 'real'});
            obj.Origin = point;
        end

        function obj = set.Width(obj, value)
            validateattributes(value, {'double'}, ...
                {'size' [1 1], 'real', 'nonnegative'});
            obj.Width = value;
        end

        function obj = set.Height(obj, value)
            validateattributes(value, {'double'}, ...
                {'size' [1 1], 'real', 'nonnegative'});
            obj.Height = value;
        end
    end
end
```

Detection: Not detectable

History: Introduced in Version 1.0

Use of dependent properties

Type: Best Practice

Description: Use dependent properties only when one or more of the following is true:

- Its value is computed solely from the value of other properties
- Compatibility mandates the property be available to users of the class even if the property is no longer used in the implementation
- A property change needs to cause a side effect on other properties

Otherwise use a non-dependent property.

Motivation:

- Understandability: Non-dependent properties are easier to understand.
- Testability: Dependent properties can complicate unit testing because they can automatically change when related properties are modified.

Recommended:

```
classdef Rectangle
    properties
        Origin (1,2) double {mustBeReal}
        Width  (1,1) double {mustBeReal, mustBeNonnegative}
        Height (1,1) double {mustBeReal, mustBeNonnegative}
    end

    properties (Dependent)
        Area
    end

    methods
        % Area calculated from Width and Height
        function area = get.Area(obj)
            area = obj.Width*obj.Height;
        end
    end
end
```

Detection: Not detectable

History: Introduced in Version 1.0

Methods

Argument validation

Type: Best Practice

Description: Validate input arguments for those methods that are intended to be part of an external, user-facing programming interface (public methods). Use an `arguments` block, introduced in R2019b, to do validation.

Motivation:

- Maintainability: Method argument validation ensures that future developers and users understand the requirements for the public methods of the class.

Recommended:

```
classdef Rectangle
    properties
        Origin (1, 2) double = [0 0]
        Width  (1, 1) double {mustBeNonnegative} = 1
        Height (1, 1) double {mustBeNonnegative} = 1
    end

    methods (Access = public)
        function R = enlarge(R, x, y)
            arguments (Input)
                R (1, 1) Rectangle
                x (1, 1) {mustBeNonnegative}
                y (1, 1) {mustBeNonnegative}
            end
            R.Width = R.Width + x;
            R.Height = R.Height + y;
        end
    end
end
```

Detection: Not detectable

History: Introduced in Version 1.0

Class constructor

Type: Best Practice

Description: Avoid writing class constructors that return more than one argument. A class constructor must return a valid object or an array of objects of the class.

Motivation:

- Understandability: The constructor is designed only to instantiate the object. Returning multiple outputs increases the complexity of the constructor.

Recommended:

```
classdef SquareMatrix
    % Custom validator mustBeSquare not shown
    properties
        Data (:,:) double {mustBeMatrix, mustBeSquare} = eye(2)
    end

    methods
        % Constructor returns an object of class SquareMatrix
        function obj = SquareMatrix(varargin)
            if nargin == 1
                obj.Data = varargin{1};
            end
        end

        function c = conditionNumber(obj)
            c = cond(obj.Data);
        end
    end
end
```

Not Recommended:

```
classdef SquareMatrix
    % Custom validator mustBeSquare not shown
    properties
        Data (:,:) double {mustBeMatrix, mustBeSquare} = eye(2)
    end

    methods
        % Constructor returns object and another value
        function [obj, condNum] = SquareMatrix(varargin)
            if nargin == 1
                obj.Data = varargin{1};
            end
            condNum = cond(obj.Data);
        end
    end
end
```

Detection: Not detectable

History: Introduced in Version 1.0

Private methods

Type: Best Practice

Description: Make methods `private` or `protected` unless they are intended to be called by users of the class.

Motivation:

- Maintainability: Making methods private reduces the risk of those methods being used incorrectly by external code and private methods can be modified without impacting users of the class.

Recommended:

```
classdef SquareMatrix
    % Custom validator mustBeSquare not shown
    properties
        Data (:,:) double {mustBeMatrix, mustBeSquare} = eye(2)
    end

    methods
        function obj = SquareMatrix(varargin)
            if nargin == 1
                obj.Data = varargin{1};
            end
        end

        function c = conditionNumber(obj)
            c = cond(obj.Data);
        end
    end

    methods (Access = private)     % Internal calculation
        function d = determinant(obj)
            d = det(obj.Data);
        end
    end
end
```

Detection: Not detectable

History: Introduced in Version 1.0

Get methods

Type: Best Practice

Description: Avoid the use of get methods for non-dependent properties.

Motivation:

- Performance: Get methods can add unnecessary overhead.

Recommended:

```
classdef Rectangle
    properties
        Origin (1,2) double {mustBeReal}
        Width  (1,1) double {mustBeReal, mustBeNonnegative}
        Height (1,1) double {mustBeReal, mustBeNonnegative}
    end

    properties (Dependent)
        Area
    end

    methods
        % get method for dependent property Width
        function area = get.Area(obj)
            area = obj.Width*obj.Height;
        end
    end
end
```

Detection: Not detectable

History: Introduced in Version 1.0

Overloaded indexing

Type: Best Practice

Description: Use modular indexing when creating a class with custom indexing. Avoid overloading `subsref` and `subsasgn` whenever possible.

Motivation:

- Maintainability: Modular indexing allows different indexing operations (e.g., paren, brace, and dot) to be customized individually.

- Performance: Modular indexing is more performant than `subsref` and `subsasgn`.

Recommended:

See documentation on Modular Indexing for detailed examples

- matlab.mixin.indexing.RedefinesParen
- matlab.mixin.indexing.RedefinesBrace
- matlab.mixin.indexing.RedefinesDot

Detection: Not detectable

History: Introduced in Version 1.0

Error Handling Guidelines

General

Code Analyzer warnings

Type: Best Practice

Description: Fix all Code Analyzer warnings before submitting code to source control or when making code available for use by others.

Motivation:

- Readability: Fixing Code Analyzer warnings ensures that code is free of potential issues like unused variables, unreachable code, or poor formatting resulting in cleaner and more readable code.

Detection: Indicated in Editor or Code Analyzer Report

History: Introduced in Version 1.0

Informative error messages

Type: Best Practice

Description: Write error messages that provide specific information to help the user understand the issue and what to do about it. Error messages should take one of three forms:

- *Problem and solution form:* The first sentence of the message states the problem. The next sentence explains ways to fix it.

- *Solution form:* The error message is a statement of what the user could do or what must be true to fix the problem.
- *Problem form:* The error message is a statement of the problem. Used when it is not possible to state a specific solution to the problem.

Motivation:

- Understandability: Clear, specific error messages help users understand what to do when an error occurs and help future developers know what conditions in the code trigger specific messages.

Recommended:

Problem and Solution Form:

- `Too many figure objects. Print 1 figure at a time.`
- `Sparse matrices not supported. Use eigs instead.`

Solution Form:

- `Matrices must be the same size.`
- `"RelTol" value must be a nonnegative scalar.`

Problem Form:

- `Matrix contains NaN or Inf.`
- `Invalid option combination.`

Detection: Not detectable

History: Introduced in Version 1.0

Reset state on error

Type: Best Practice

Description: Reset global state or settings to their original values when handling errors. Consider using a `try-catch` block or an `onCleanup` object to reset the state.

Motivation:

- Reusability: Errors should not leave behind external state changes.

Recommended:

```
function output = myFunction(input)
oldPath = path;
c = onCleanup(@()path(oldPath));

newFolder = "C:\MATLAB\mydir";
addpath(genpath(newFolder));

% Perform some calculations which might error
end
```

Detection: Not detectable

History: Introduced in Version 1.0

Try-Catch

Use try-catch with the exception object

Type: Best Practice

Description: Use `try-catch` blocks for error handling or to process exceptional conditions. Do not use `try-catch` for normal flow control. Include a matching `catch` block for every `try` block. If a `catch` block is empty, include a comment explaining why no further processing is required. Use the `MException` object when a `catch` block tries to recover from a specific error. Do not assume which error has occurred.

Motivation:

- Understandability: Using `try-catch` allows readers to quickly find the place in the code where specific errors or events are handled.

Recommended:

```
function manageGlobalState()
% Store original path and current folder
originalPath = path();
originalDir = pwd();

try
    % Modify global state
    addpath("tempFolder");
    cd("C:\Temp");

    % Perform some operations that might error

catch exception
    % Restore path and current folder
    cd(originalDir);
    path(originalPath);

    % Rethrow the error to inform the caller
    rethrow(exception);
end
end
```

Detection: Optionally by enabling Code Analyzer check `CTCH` (R2010b)

History: Introduced in Version 1.0

Avoid throwAsCaller

Type: Best Practice

Description: Avoid the use of `throwAsCaller`.

Motivation:

- Understandability: Using `throwAsCaller` requires that the error originates exactly one level below the function that calls it. It can give misleading error traces if the error is deeper in the call stack.

Recommended:

```
function numRepos = queryGitHubRepos()
    try
        numRepos = getMatlabRepoCount();
    catch exception
        throw(exception);      % Preserves the full error stack
    end
end

function numRepos = getMatlabRepoCount()
    prefix = "https://api.github.com/search/";
    url = prefix + "repositories?q=language:matlab";
    try
        response = webread(url);
        numRepos = response.total_count;
    catch exception
        error("getMatlabRepoCount:RequestFailed", ...
            "GitHub API request failed: %s", exception.message);
    end
end

>> queryGitHubRepos()
Error using queryGitHubRepos (line 7)
```

```
GitHub API request failed: Could not access server.
https://api.github.com/search/repositories?q=language:matlab.
```

Not Recommended:

```
function numRepos = queryGitHubRepos()
    try
        numRepos = getMatlabRepoCount();
    catch exception
        throwAsCaller(exception);   % Hides true source of the error
    end
end

function numRepos = getMatlabRepoCount()
    prefix = "https://api.github.com/search/";
    url = prefix + "repositories?q=language:matlab";
    try
        response = webread(url);
        numRepos = response.total_count;
    catch exception
        error("getMatlabRepoCount:RequestFailed", ...
            "GitHub API request failed: %s", exception.message);
    end
end

>> queryGitHubRepos()
GitHub API request failed: Could not access server.
https://api.github.com/search/repositories?q=language:matlab.
```

Detection: Optionally using custom Code Analyzer check for existing functions.

History: Introduced in Version 1.0