GENERAL PRINCIPLES
1. Avoid Causing Confusion
2. Avoid Throw-Away Code
3. Help the Reader
4. Maintain the Style of the Original
5. Document Style Deviations

FORMATTING

Layout
6. Keep Content Within the First 80 Columns
7. Split Long Code Lines at Graceful Points
8. Indent Statement Groups 3-4 Spaces
9. Indent Consistently with the MATLAB Editor
10. Do Not Use Hard Tabs
12. Define Each Variable on a Separate Line
13. Use Argument Alignment if it Enhances Readability
14. Avoid Heavily Nested Code

White Space
15. Include White Space
16. Surround =, &, |, &&, and || by Spaces
17. Use White Space Around Operators When in Enhances Readability
18. Follow Commas with a Space When it Enhances Readability
19. Insert Spaces for Multiple Commands in One Line
20. Do Not Put Spaces Just Inside Parentheses
21. Do Not Follow Function Names With a Space
22. Do Not Space Out Semicolon at the End of Lines

Code Blocks
23. Break Code of Any Appreciable Length into Block
24. Separate Logical Groups of Statements Within a Block by One Blank Line
25. Separate Major Code Blocks by More Than One Blank Line
26. Separate Subfunctions by More Than One Blank Line
27. Use Editor Cells

NAMING

General
28. Use Meaningful Names
29. Use Familiar Names
30. Use Consistent Names
31. Avoid Excessively Long Names
32. Avoid Cryptic Abbreviations
33. Tweak Familiar Acronyms as Words
34. Avoid Names that Differ Only by Capitalization
35. Avoid Names that Differ Only by One Letter
36. Avoid Names with Hard-to-Read Character Sequences
37. Make Names Pronounceable When You Can
38. Write Names in English

Variables and Parameters
39. Avoid Ambiguous or Vague Names
40. Name According to Meaning, Not Type
41. Use Lowercase for Simple Variable Names
42. Use lowerCamelCase for Compound Variable Names
43. Use Meaningful Names for Variables with a Large Scope
44. Limit Use of Very Short Names to Variables with a Small Scope
45. Be Consistent With i and j
46. Use the Prefix n for Variables Representing the Number of Entities
Follow a Consistent Convention on Pluralization
Use the Prefix `this` for the Current Variable
Use the Suffix `No` or Prefix `i` for Variables Representing a Single Entity Number
Prefix Iterator Variables with `i, j, k, etc`
Embed `is, has, etc.` in Boolean Variable Names
Avoid Negated Boolean Variable Names
Use the Expected Logical Names and Values
Avoid Using a Keyword or Special Value Name for a Variable Name
Avoid Hungarian Notation
Avoid Variable Names that Shadow Functions
Avoid Reusing a Variable for Different Contents
Consider a Unit Suffix for Names of Dimensioned Quantities

Constants
Use All Uppercase for Constant Names with Local Scope
Use Function Names for Constants Defined by Functions
Use Meaningful Names for Constants
Define Related Constants Based on the Relation
Consider Using a Category Prefix

Structures and Cell Arrays
Use UpperCamelCase for Structure Names
Do Not Include the Name of the Structure in a Fieldname
Use Fieldnames that Follow the Naming Convention for Variables
Name Cell Arrays Following the Style for Variables

Functions
Give Functions Meaningful Names
Name Functions for What They Do
Follow a Case Convention for Function Names
Reserve the Prefixes `get/set` for Accessing an Object Property
Use Expected Verbs in Expected Ways
Use the Prefix `is` for Boolean Functions
Use Complement Prefixes in Compound Names for Complement Operations
Be Selective in the Use of Numbers at the Ends of Names
Use Numbers Inside Function Names Only for Common Conventions
Avoid Unintentional Shadowing

Classes
Use Nouns When Naming Classes
Use UpperCamelCase for MATLAB Class and Object Names
Use UpperCamelCase for Exception Names
Name Properties Like Structure Fields
Name Methods Line Functions
Name Accessor Methods after their Properties
Use a Single Lowercase Word as the Root Name of a Package

Data Files and Directories
Use Directory and Filenames that are Easy to Work with
Use Sortable Numbering in Data Filenames
Use ISO Data Format

DOCUMENTATION
General
Provide Well-Written Code
Document Each Module Before or During Its Implementation
Document the Interface for Those Who Will Use It
Document the Design and Implementation for Those Who Will Maintain It
Consider Documenting Some Changes Header Comments
Generate HTML Reference Pages
Integrate the Reference Pages with the Help Browser
Consider Providing PDF Documentation

Comments
Make Comments Useful
Be Sure that Comments Agree with the Code
98. Revise Comments to be Correct When the Code is Changed
99. Put Restrictions in the Code, Not the Comments
100. Clean up Commented Out Code before Release
101. Make Comments Easy to Read
102. Write Comments for the Publish Feature
103. Minimize Use of End-Line Comments
104. Consider End-of-Loop Comments
105. Consider Documenting Important Variables Near the Start of the File
106. Consider Documenting Constant Assignments Where They are Defined
107. Use Voice and Person Appropriately
108. Use Present Tense to Describe Code
109. Use Complete Sentences in Descriptive Blocks
110. You Can Use Incomplete Sentences in One-Liners
111. Use Short Words
112. Eliminate Cute Comments
113. Minimize the Use of ASCII Art
114. Write All Comments in English

Header Comments
115. Format the Header Comments for Easy Publishing as Documentation
116. Put the Header Comments in the Right Place
117. Use Title Markup for the Function Name
118. Document the Function Interface in the Syntax Section
119. Use the Actual Function Name Case in Comments
120. Describe the Function Arguments in the Description Section
121. Describe Any Function Side Effects
122. Describe Any Custom Exception that May be Generated
123. Include Examples in the Header Comments
124. Include See also in the Header Comments
125. Avoid Clutter in the Reference Page
126. Format Header Comments of classdef Files for the Help Browser
127. Clarify Subclass Methods

Block Comments
128. Indent Block Comments to Match Code
129. Place a Blank Line or Cell Break before a Block Comment

Minimize y within constraints on x
130. Do Not Use Comment Blocks for Block Comments
131. Use Comment Block Syntax to Temporarily Bypass a Block of Code

Interspersed or Inline Comments
132. Indent Comments with the Code Block
133. Do Not Follow a Single-Line Comment with a Blank Line
134. Resolve TODO/FIXME Comments

PROGRAMMING
General
135. Avoid Cryptic Code
136. Avoid Off-By-One Mistakes
137. Attend to NaN Results
138. Consider Using isfinite
139. Use Programming Patterns or Idioms

Variables and Constants
140. Do Not Reuse Variable Names Unless Required by Memory Limitation
141. Beware of Mistyping Variable Names
142. Minimize the Use of Literal Numbers in Statements
143. Write Floating Point Values with a Digit Before the Decimal Point
144. Avoid Showing Excessive Decimal Places
145. Avoid Mixing Units within a Program
146. Use Caution with Floating Point Comparison
147. Limit Boolean Variable Values to True or False
148. Do Not Assume Array Size
149. Use Appropriate Numerical Class Conversions
150. Minimize the Use of Global Variables
151. Minimize the Use of Global Constants
152. Define Local Constants Only Once
153. Do Not Declare Unrelated Variables in a Single Line

Character Strings
154. Consider Using `strcmpi`
155. Use `lower` or `upper` When You Cannot Use `strcmpi`
156. Use `isempty`
157. Use `fullfile`
158. Compute with Numbers for Speed
159. Consider Using Character Arrays for Speed
160. Consider Using `unique`

Structures
161. Use Structures for Associated Data
162. Use Structures for Metadata
163. Organize a Structure Based on How It Will be Accessed
164. Put Structure Field in a Helpful Order
165. Be Careful with Fieldnames

Cell Arrays
166. Consider Using Cell Arrays for String
167. Look Out for Cells within Cells
168. Consider Using Cell Arrays for Ragged Arrays

Expressions
169. Write Short Expressions
170. Put Numbers Before the Multiplication Operator
171. Make the Denominator Clear
172. Use Parentheses
173. Use a Clear Form for Relational Expressions
174. Use `&&` or `||` by Default for Scalar Operands

Statements
175. Write Short Statements
176. Avoid Use of `eval` When Possible

Loops
177. Initialize Loop Variables Immediately Before the Loop
178. Initialize Using `nan` or `false` Rather Than `zeros`
179. Do Not Change the Loop Index Variable Inside a Loop
180. Minimize the Use of `break` in Loops
181. Minimize the Use of `continue` in Loops
182. Avoid Unnecessary Computation within Loops
183. Be Careful of Infinite `while` Loops
184. Be Careful of Count Errors in `while` Loops

Conditionals
185. Avoid Complicated Conditional Expressions
186. In General, Include an `else` Statement with `if`
187. Put the Normal Branch in the `if` Part
188. Avoid Unnecessary `if` Constructs
189. Use a Practical Order for `elseif` Conditions
190. Avoid Unnecessary `elseif` Blocks
191. Avoid Unnecessary Levels of Nesting in Control Structures
192. Try to Simplify Nested `if` Constructs
193. Avoid the Conditional Expression `if 0`
194. Include otherwise with `switch` Statements
195. Consider Using a Cell Array to Simplify a `switch` Construct
196. Use `if` When the Condition is Most Clearly Written as an Expression
197. Use `switch` When the Condition is Most Clearly Written as a Value
198. When Either `if` or `switch` Can Work, Use the More Readable One

Logical Functions
199. Use `logical` to Cast Variables
200. Use `true` or `false` Functions and Values
In General Use `isequal` Rather Than `==` Vectorization

Be Thoughtful with Vectorization

Use `repmat` Functions

Modularize

Write Code as Functions When Possible

Write Simple Functions

Design for Loose Coupling

Use Subfunctions Appropriately

Hide Implementation Details

Write for High Cohesion

Use Existing Functions

Eliminate Overlapping Functions

Provide Some Generality in Functions

Write a Function at One Level of Abstraction

Write Convenience Functions

Make Interaction Clear

Name All Input Arguments

Write Boolean Functions to Return `true` or `false`

Make Logical Output and Function Name Consistent

Minimize Input Flag Arguments

Write Arguments in Useful Order

Use Lazy Evaluation

Make Input and Output Arrays Consistent

Use a Structure to Replace a Long List of Function Arguments

Consider an Options Structure

Write Parameter-Value Pairs for Optional, Unordered Input Arguments

In General, Use Caller Variable Names Consistent with the Function Argument Names

Define Imports Where They are Easy to Find

Use Anonymous Functions Rather Than Inline Functions

Use Function Handles

Avoid Including Hidden Side Effects

Refactor

Input and Output

Write Input Functions

Provide Some Input Validity Checking

Expect NaN Values in Data

Use `feof` for Reading Files

Make Output Modules

Format Output for Easy Use

Provide for Automation

Classes and Objects

Keep Classes Simple

Avoid Classes with Ambiguous Overlap

Construct Valid Objects

Follow Constructor Conventions

Define Small, Simple Methods

Write Methods That You Can Unit Test

Do Not Write a Method That Can Produce an Invalid Property

Avoid Incomplete Public Methods

Try to Make Properties Private

Do Not Expose Properties with Behavior

Avoid Writing Methods with Many Input Arguments

Validate Method Argument Values

Check for a Property’s Existence Before Using It

Refactor Repeated Code Into Methods

Overload Standard Functions When Possible
Avoid Unconventional Usage of Overload Names
Do not Overload && or ||
Do not Get Carried Away with Inheritance
Place Method Functions in Folders Consistently
Use Java Syntax for Java Methods

Exceptions, Errors, and Warnings
Use Appropriate Error Handling
Prepare for Errors
Make Error Messages Informative
Use Message IDs with Errors and Warnings
Use Exceptions
Support Error Returns
Use Appropriate Assertions

Output Style
Learn to Use sprintf
Learn to Use Tex

Tests
Write at Least One Test for Every Function or Method
Write Small Tests
Write Uncoupled Tests
Write Tests with Boolean Outputs
Test for Expected Exceptions
Write Tests You Can Automate
Use Related Names in the Function and the Test Code
Identify Test Files by Name
Develop Test Patterns
Consider Tests for External Software

Data Files
Make Use of mat Files
Follow Database Conventions
Follow the MATLAB Convention for Data Array Orientation

FILES AND ORGANIZATION
Toolboxes
Organize General-Purpose m-Files in Toolboxes
Put Test Files in a Separate Directory
Consider Writing Demo Files
Use a Consistent Toolbox Folder Organization
Provide for Integration with MATLAB
Provide a Reference Page for Every Public Function
Integrate the Reference Pages with the Help Browser
Do not Make Your Toolbox a Subfolder of the MATLAB Folder

Project Files
Organize Your Project-Specific Files by Project Directory
Organize Your Data Directory for Ease of Access

DEVELOPMENT
Design
Expect Change
Include Appropriate Flexibility
Use the Right Algorithm
Program by Contract
Write for Automation
Make Associated Data Easy to Use
Recompute When Data Changes
Leave Code Optimization for Last or Never
Consider Coding Standards

Development Practices
Develop in Small Steps
Develop in the Editor
Use Version Control
Run Tests Often
307. Run all Tests Before Release

**MATLAB IDE Tools**

308. Try the MATLAB Editor
309. Use Smart Indent
310. Use Find and Replace
311. Pay Attention to M-Lint
312. Use Consistent Preference Settings
313. Use the Debugger Effectively
314. Use The TODO/FIXME Report
315. Use the Profiler
316. Use the Dependency Report
317. Publish During Development
318. Find a Desktop Layout That Works for You