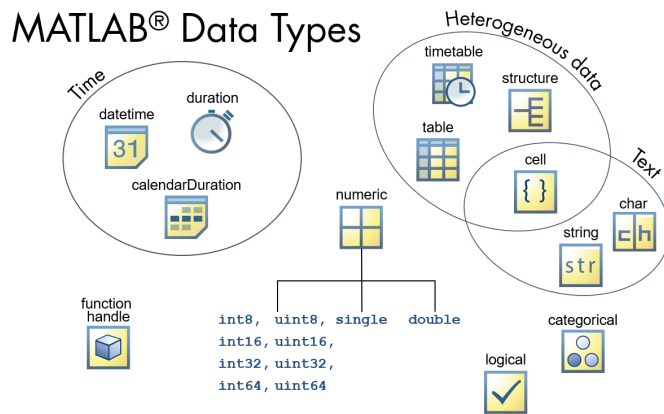


1. Introduction

2. Structuring Data

Summary: Overview of MATLAB Data Types

Each array type is intended to store data with a characteristic organization. Some arrays hold data all of the same data type, and others allow storage of different data types.



Homogeneous Arrays

Homogeneous arrays are scalars, vectors, matrices, and higher dimensional arrays whose elements all have the same data type. Elements of an array can be numbers, logical values, dates and times, strings, or one of many other other data types. For instance, a logical matrix would be a rectangular array with elements that are all logical values.

Array Type	Intended Contents
single double	Floating-point numbers with ~8 (single) or ~16 (double) decimal places of precision Unless otherwise specified MATLAB stores all numeric variables as type double .
int* uint*	Integer values of various ranges
char string	Text
datetime duration calendarDuration	Dates, times, durations between dates and times
logical	Boolean (true/false) values
categorical	Values from a finite set of alphanumeric (usually text) labels
function_handle	The definition of a function

Heterogeneous Arrays

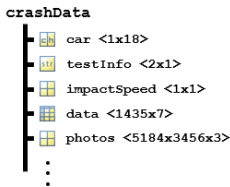
Sometimes data are not all the same data type or size. There are specific containers for these heterogeneous data.

Tables/Timetables

Make	Model	Year	Type	Weight	Length	Wheels

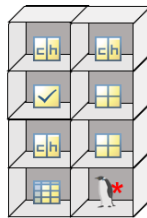
Tabular data with named columns that may have different types

Structures



Heterogeneous data with named fields

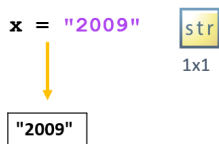
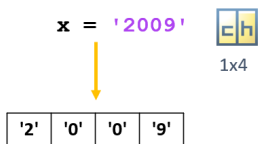
Cell Arrays



Heterogeneous data, indexed numerically

Summary - Structuring Data

Character Arrays vs String Arrays



The primary difference between a character array and a string array is that an individual element of a character array is a character and an element of a string array is a piece of text, which consists of several characters.

Organizing Data in Tables

When several variables in your workspace hold data about the same observations, you can use the table function to collect this information into a table.

You can index into a table using array indexing with parentheses. Columns can be specified either numerically or by name.

```

t1 = table(var1,var2, 'VariableNames',{'Thought1','OnSecondThought'})

t1 =
    2x2 table
    Thought1    OnSecondThought
    -----
    "Ford"      32.7
    "Toyota"     1

t2 = t1(2,"Thought1");

t2 =
    table
    Thought1
    -----
    "Toyota"
  
```

Organizing Data in Cell Arrays

Create a cell array using curly braces and separating elements with commas and semicolons.

You can index into a cell array using standard MATLAB indexing with parentheses, (). This returns a portion of the original cell array.

```

car = {'Ford','Expedition';32.7,true}
car =

    2x2 cell array

    {'Ford' }    {'Expedition'}
    {[32.7000]}    {[ 1]}

car(1,1)
ans =
    1x1 cell array
    {'Ford'}
  
```

You can access the contents of cells in a cell array by indexing with curly braces, {}, rather than parentheses, ().

```
car{1,1}
ans =
    'Ford'
```

Organizing Data in Structures

This syntax creates a structure named `car` with a field named `make` and assigns a value to that field.

The same dot notation can be used to create new fields.

To add an element to a structure array, use parentheses and assign a value to a field of the struct.

You can use the field name and array indexing to extract the data.

```
car.make = "Ford"
car = struct with fields:
    make: "Ford"

car.model = "Expedition"
car = struct with fields:
    make: "Ford"
    model: "Expedition"

car(2).model = "Prius"
car = 1-by-2 struct
```

make	model
"Ford"	"Expedition"
□	"Prius"

```
car(2).model
ans =
    "Prius"
```

3. Manipulating Heterogeneous Data

How Do You Collect Output from Comma-separated Lists?

Concatenating Horizontally

Regardless of where the comma-separated list comes from, square brackets concatenate the data into a row vector.

Cell Arrays

The 2-by-2 cell array stored in the `x` variable contains four scalars.

Using square brackets creates a numeric row vector. The elements of `x` are concatenated columnwise.

```
x
x =
    [10]    [20]
    [30]    [40]

[x{:}]
ans =
    10     30     20     40
```

Structure Arrays

The structure array stored in the `data` variable has four elements. `data.coords` returns the `coords` field of each element separately.

Concatenating with square brackets concatenates columnwise, which creates a single 1-by-12 vector from the four 1-by-3 vectors.

```
data.coords
ans =
    0.2081    0.1663    0.5292
ans =
    0.0265    0.6137    0.8135
ans =
    0.6473    0.7456    0.4188
ans =
    0.9763    0.9831    0.0538

y = [data.coords]
y =
Columns 1 through 4
    0.2081    0.1663    0.5292    0.0265
Columns 5 through 8
    0.6137    0.8135    0.6473    0.7456
Columns 9 through 12
    0.4188    0.9763    0.9831    0.0538
```

Concatenating Vertically

You can use the `vertcat` function on comma-separated lists to concatenate vertically.

Cell Arrays

The 2-by-2 cell array stored in the `x` variable contains four scalars.

Extract multiple outputs from `x` and concatenate vertically.

```
x
x =
    [10]    [20]
    [30]    [40]

vertcat(x{:})
ans =
    10
    30
    20
    40
```

Structure Arrays

The structure array stored in the `data` variable has four elements.

Stack the data vertically

```
data.coords
ans =
    0.2081    0.1663    0.5292
ans =
    0.0265    0.6137    0.8135
ans =
    0.6473    0.7456    0.4188
ans =
    0.9763    0.9831    0.0538

x = vertcat(data.coords)
x =
    0.2081    0.1663    0.5292
    0.0265    0.6137    0.8135
    0.6473    0.7456    0.4188
    0.9763    0.9831    0.0538
```

Concatenating into a Cell Array

If your comma-separated list contains data that cannot be combined into a homogeneous array, then you can combine it into a cell array using curly braces.

`y` is a 3-element structure with fields `name` and `price`

The `name` fields contains the name of fruits.

Those names can be put into cells and concatenated into a cell array using curly braces.

```
y
y =
1x3 struct array with fields:
    name
    price

y.name
ans = 'Banana'
ans = 'Apple'
ans = 'Orange'

{y.name}
ans =
    {'Banana'}    {'Apple'}    {'Orange'}
```

How Do You Create and Use a Function Handle?



Create a Function Handle to a Defined Function

Use the `@` symbol to create a function handle referencing an existing function. You can reference a MATLAB function, a local function, or a function defined in a function file.

`f = @myFun`   `function y = myFun(a,b,c)`
`y = a*(b-sin(c))`
`end`

Create a Function Handle to a Specific Command

These are called *anonymous functions*. Again, you use the `@` symbol. Specify inputs with parentheses.

`g = @(a,b,c) a*(b-sin(c))`  

Here, the function handle `g` has three inputs: `a`, `b`, and `c`.

Using Function Handles

Once defined, the function handles `f` and `g` can be used the same way. They can be called with inputs or passed as an input to another function.

Construct a function handle with an `@` sign.

Call the function handle in the same way that you would call the function directly.

Some functions, such as `integral`, require an input function. In this case, pass a function handle as an input.

```
h = @sin;

h(0)
ans =
    0

h(pi/2)
ans =
    1

integral(h,0,pi)
ans =
    2.0000

integral(@log,1,2)
ans =
    0.3863
```

How Do You Apply Functions to Arrays?

Apply a Function to Each Element of an Array

You can apply a function to each element of an array with `arrayfun`. The `arrayfun` function works on any array.

```
>> B = arrayfun(func,A)
```

Outputs		Inputs	
B	An array whose elements are the output of func operating on each element of A.	func	A function handle
		A	An array

If `func` returns scalars of the same data type, then `B` is an array of that data type. If the outputs of `func` are non-scalar or non-uniform, set the `"UniformOutput"` property to `false`. In this case, `B` is a cell array.

```
B = arrayfun(func,A,"UniformOutput",false)
```

Applying a Function to the Contents of a Cell Array

You can apply a function to the contents of each element of a cell array with `cellfun`.

```
>> B = cellfun(func,C)
```

Outputs		Inputs	
B	An array whose elements are the output of func operating on each element of C.	func	A function handle
		C	A cell array

Use the `"UniformOutput"` property to specify that the output of `func` is either not a scalar or non-uniform.

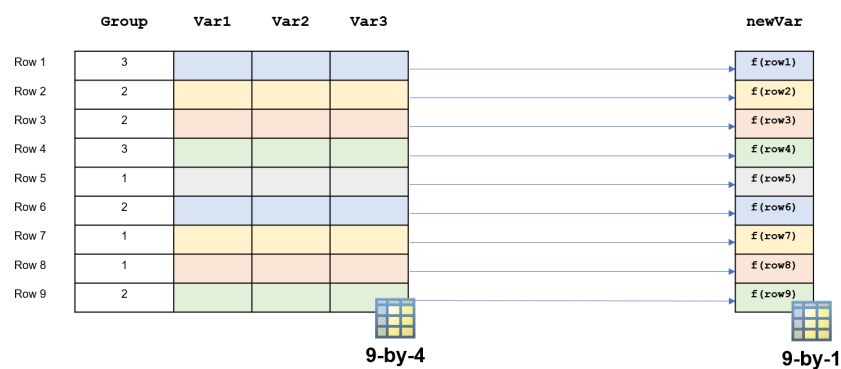
```
B = cellfun(func,C,"UniformOutput",false)
```

Applying Functions to Rows or Variables of a Table

The `varfun` and `rowfun` functions allow you to apply a function to columns or rows of a table, respectively. They have very similar syntax.

Applying a Function to the Rows of a Table

rowfun allows you to perform calculations on rows.



You can specify input, output, and grouping variables.

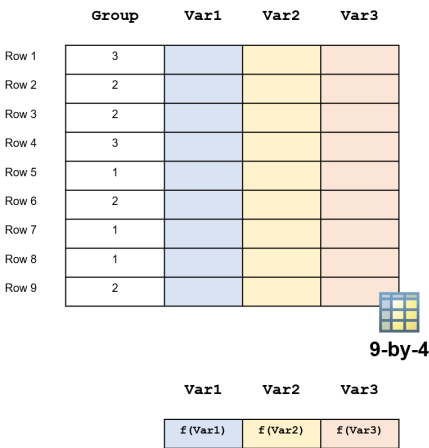
```
>> B = rowfun(func, tbl)
```

Outputs		Inputs	
B	A table or timetable where the i^{th} row is equal to $\text{func}(A\{i, : \})$.	func	A function to apply to rows of the table.
		tbl	A table or timetable.

- The following name-value pairs can be used with both rowfun .
- "InputVariables" : Specify which variables to pass to func as a string array. These elements are passes as separate inputs to the function.
 - "GroupingVariables" : Specify one or more variables that define groups of rows as a string array.
 - "OutputVariableNames" : Specify the names for the outputs of func .

Applying a Function to the Variables of a Table

varfun allows you to perform calculations on specific variables.



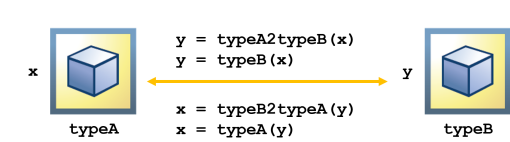
```
>> B = varfun(func, tbl)
```

Outputs		Inputs	
B	A table or timetable containing the calculations on the specified columns.	func	A function to apply to columns of the table.
		tbl	A table or timetable.

- The following name-value pairs can be used with both varfun .
- "InputVariables" : Specify which variables to pass to func as a string array. These elements are passes as separate inputs to the function.
 - "GroupingVariables" : Specify one or more variables that define groups of rows as a string array.

Converting Data Types

MATLAB provides conversion functions of the form *typeA2typeB* to convert between data types.



Structuring Heterogeneous Data

Extracting Multiple Elements from Cell and Structure Arrays

Function Handles

Applying Scalar Functions to Arrays

Converting Data Types

The table below summarizes most of the functions available in MATLAB to convert between data types.

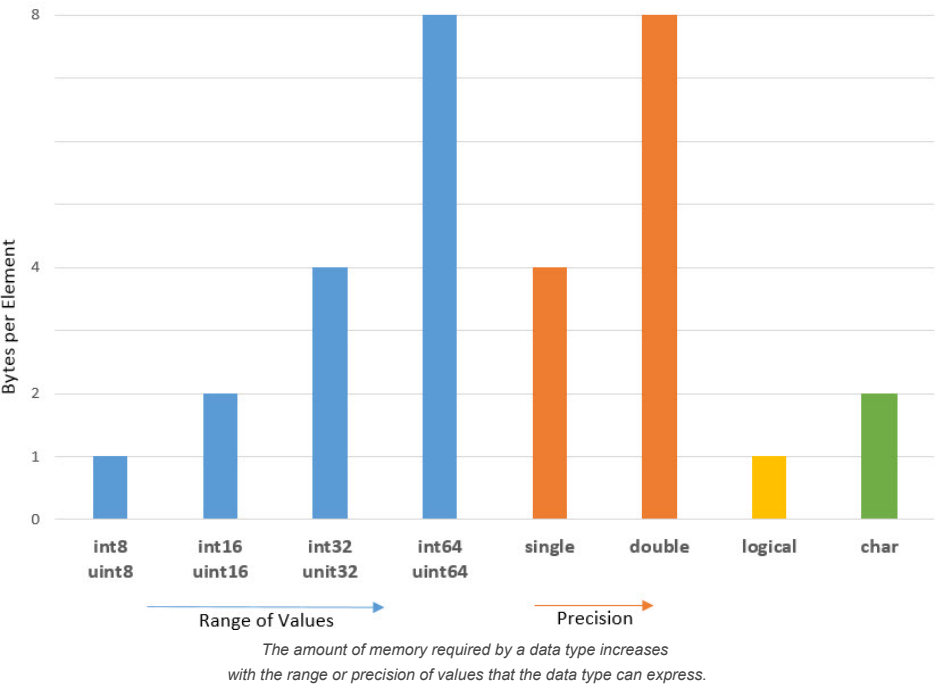
To → ↓ From										
		num2str	num2cell mat2cell	datetime	Various *	array2table	logical			categorical
	str2double str2num		cellstr	datetime					str2func	categorical
	cell2mat	char		datetime		cell2table		cell2struct		categorical
	Various *	char	cellstr							
	Various *	char	cellstr							
	table2array		table2cell					table2struct		
	double		num2cell mat2cell							categorical
			struct2cell			struct2table				
		func2str								
	double	char	cellstr num2cell							

4. Optimizing Your Code

Data Types and Memory

Data Types with Fixed Memory Requirement

Some homogeneous data types require a fixed amount of memory per element.

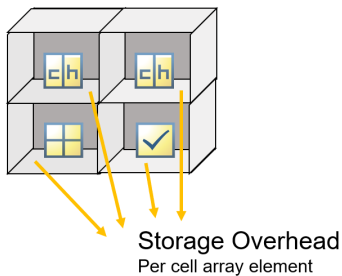


Container Data Types

Container variables (tables, cell arrays, and structure arrays) require overhead in addition to the data they store. This additional memory is used to store information about the contents of the variable. The amount of the overhead depends on the size of the variable.

```
car = {'Ford', 'Expedition'; 32.7, true}
```

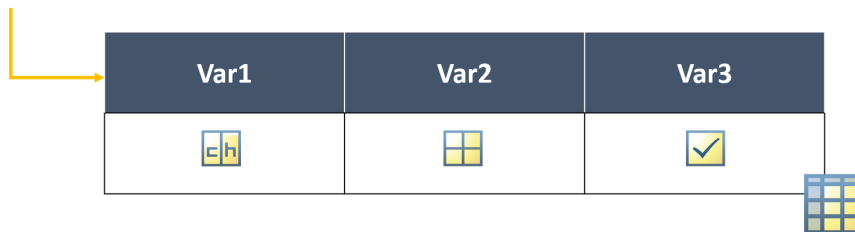
2-by-2



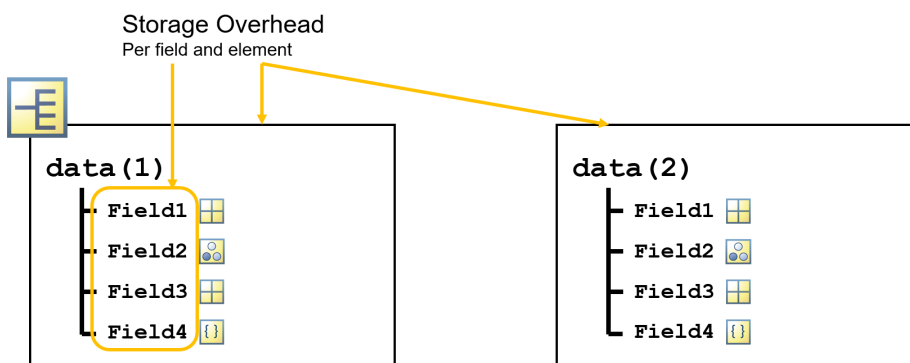
Each cell of a cell array requires memory overhead.

Product or brand names of cars are trademarks or registered trademarks of their respective holders.

Storage Overhead



A table requires minimal overhead for each variable and for storing table properties.



A structure requires 64 bytes of overhead for each field name and 112 bytes for each container. If a structure array has more than one element, each element will require additional overhead. So the total memory used is

$$112 * (\text{number of elements}) * (\text{number of fields}) + 64 * (\text{number of fields}) + \text{data}$$

In the example below, you can see how to calculate the total memory of a cell array of character vectors.

<code>headquarters = 'Natick, MA';</code>	<code>headquarters = {'Natick, MA'};</code>	<code>headquarters = {'Natick' 'MA'};</code>
Data Type - char	Data Type - cell	Data Type - cell
Memory - 20 Bytes (10 characters, 2 bytes each)	Memory - 132 Bytes (112 bytes for cell array element, 20 bytes for characters)	Memory - 240 Bytes (112 bytes for each element of the cell array, 12 + 4 bytes for the character vectors)

Preallocation

Preallocating Numeric Arrays

You can use the functions `zeros` and `ones` to preallocate numeric arrays.

By default, `zeros` and `ones` create an array having the data type **double**.

You can create an array having other data types by specifying an additional input.

Another way to preallocate an array is to assign a value to the last element of the array.

```
x = zeros(1,50000);
```

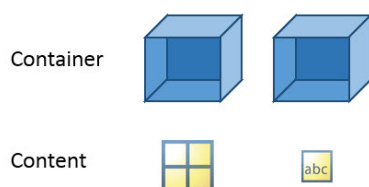
```
x = zeros(1,50000,'single');
y = zeros(1,50000,'int16');
```

```
x(8) = 3;
x =
    0    0    0    0    0    0    0    3
```

Preallocating Cells and Structures

You can also preallocate non-numeric data types, such as cell arrays and structure arrays.

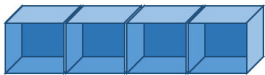
Think of these arrays as consisting of the containers (cells and structures) and the contents (any data type and size).



Preallocating a cell array or a structure array assigns space for the containers themselves, not the contents. This means that preallocation of the container variables is most beneficial when the container array itself is large, regardless of the size of the contents of the individual containers.

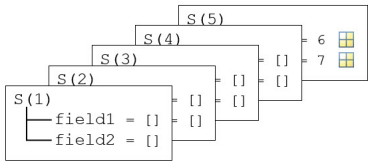
You can preallocate a cell array using the function `cell`.

```
C = cell(1,4);
```



To preallocate a structure array, start by defining the last element of the array. MATLAB will automatically replicate the field names to all of the preceding elements in the array.

```
S(5) = struct('field1',6,'field2',7)
```



Summary - Optimizing Your Code

Measuring Performance and Finding Bottlenecks

To measure the execution time of MATLAB code, you can use the `tic` and `toc` functions.

Start timer

Run code

Stop timer, report results

```
tic

x = rand(1000);

toc

Elapsed time is 1.206429 seconds.
```

To improve the performance of your code, you should first understand what parts of it take the most time. The MATLAB Profiler analyzes where your code spends the most time.

▼ Lines that take the most time

Line Number	Code	Calls	Total Time (s)	% Time	Time Plot
13	F(kk) = sqrt(Fx(kk)^2 + Fy(kk)^2 + Fz(kk)^2);	73477	0.009	17.2%	<div></div>
5	s = trimCell(sensors(k,:));	18	0.007	14.6%	<div></div>
14	end	73477	0.006	11.2%	<div></div>
17	Fmean = mean(F);	18	0.004	7.3%	<div></div>
16	[Fmax(k), maxIdx(k)] = max(F);	18	0.003	5.4%	<div></div>
All other lines			0.023	44.4%	<div></div>
Totals			0.051	100%	

Using Vectorized Operations

Vectorized code is more concise and faster than the non-vectorized solutions.

Using a for loop

nonVectorized.mlx

```
1. % Generate data
2. r = rand(1,5000);
3.
4. % Compute the difference between
5. % the adjacent elements
6. d = zeros(1,4999);
7. for i=1:4999
8. d(i) = r(i+1)-r(i);
9. end
```

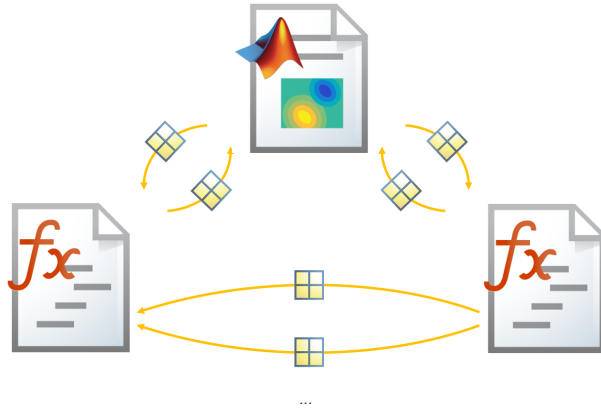
Using vectorization

vectorized.mlx

```
1. % Generate data
2. r = rand(1,5000);
3.
4. % Compute the difference between
5. % the adjacent elements
6. d = diff(r);
```

Improving Memory Usage in Functions

Scripts and functions that call each other often need to share data. Passing inputs to functions is one way functions can share data.



5. Creating Flexible Functions

Allowing Different Input Types

In some cases, you can leave inputs in their original type. For example, many mathematical operations will work on any numeric type. MATLAB also performs some conversions as needed. For example, numeric variables can be used in logical contexts, in which case they are automatically converted to logical (with zero becoming false and all non-zero becoming true).

In other cases, you can use conversion functions to force any allowed input type into the desired type.

Representations of Data Types

Multiple data types that can hold the same piece of information.

Data	Potential representations
Text	Char, string, cell array of char vectors
Numbers	Any numeric type (double , single , int*, uint*)
Dates	Datetime, numeric array (representing components, such as day, month, year), text
Discrete categories	Categorical, text, logical (for only two categories), numeric

Converting Between Data Types

Data	Conversion functions
Text	char , cellstr, string convertCharsToStrings, convertStringsToChars
Numbers	double , single , int*, uint*
Dates	datetime
Discrete categories	categorical

Note that these functions all leave the inputs unchanged if they are already in the desired type. That is, if t is a datetime variable, $t = \text{datetime}(t)$; will have no effect.

Converting Text Inputs

The convertCharsToStrings and convertStringsToChars functions convert all text inputs to one uniform type with a single function call.

```
[a,b,c,d] = convertCharsToStrings({'hello','world'},42,['this';'is';'text'],'so is this')
a =
    1x2 string array
    "hello"    "world"
b =
    42
c =
    3x1 string array
    "this"
    "is"
    "text"
d =
    "so is this"
```

*Only the text data types were converted to strings. The variable b is type **double**.*

Querying Inputs

MATLAB includes [numerous functions](#) that test a particular attribute of a variable and return a logical result. These function names typically start with the word `is`.

Querying Data Type

Function	Returns
isnumeric	True if input is numeric (single, double, int*, uint*)
isfloat	True if input is of floating point type (single or double)
isinteger	True if input is of integer type (int* or uint*)
ischar	True if input is of char type
isstring	True if input is of string type
iscellstr	True if input is a cell array of char vectors
isdatetime	True if input is a datetime
isduration	True if input is a duration
iscalendarduration	True if input is a calendar duration
islogical	True if input is of logical type
iscategorical	True if input is of categorical type
istable	True if input is a table
istimetable	True if input is a timetable
iscell	True if input is a cell array
isstruct	True if input is a structure
isa	True if input is of a specified data type
class	Name of the input data type

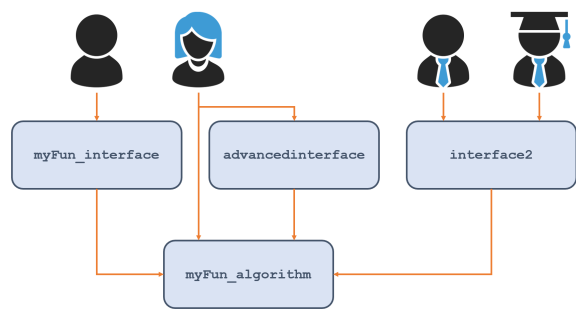
Querying Size and Shape

Function	Returns
size	Array size
length	Length of largest array dimension
numel	Total number of array elements
ndims	Number of array dimensions
isempty	True if input is empty
isscalar	True if input is a scalar
isvector	True if input is a vector
ismatrix	True if input is a matrix
isrow	True if input is a row vector
iscolumn	True if input is a column vector

Summary - Creating Flexible Functions

Creating Multiple Interfaces

Use multiple wrapper functions that use the same underlying algorithm. Each wrapper function can present a different interface to users for different applications.



Most users can use `interface2`. Advanced users can use `advancedInterface` (or call the algorithm directly).

Setting Default Inputs

To check if all the inputs were provided in the function call, you can use the function `nargin`. When called within the function body, `nargin` returns the number of inputs that were passed when the function was called.

<code>nargin</code>	Number of input arguments
---------------------	---------------------------

To allow the users to specify certain inputs (but not others), use an empty array `[]`.

- In the function body - check if the input is empty using the function `isempty`. If it is empty, assign a default value.
- In the function call - use an empty array `[]` instead of the input value.

Function Call	Editor
<code>loanint = interest(500,[],3.1,"compound")</code>	<pre>function int = interest(p,n,r,type) if isempty(n) n = 12; end</pre>

Variable Length Input Arguments

You can map multiple input arguments (e.g. a set of name-value pairs) to a single input variable `varargin` in the function definition.

Function Call	Editor
<code>analyzeAndPlot(time,position,"LineWidth",3,"Color","r")</code>	<code>function analyzeAndPlot(x,y,varargin)</code>

Matching Text Inputs

Use `validatestring` to ensure that any user input is exactly as you expect (including type and capitalization), without overly constraining your user. If the input text does not unambiguously match any of the valid strings, `validatestring` generates an error.

```

str = validatestring("distribution",["DistanceMetric","DistributionName","NumSamples"])
str =
    "DistributionName"
str = validatestring("dist",["DistanceMetric","DistributionName","NumSamples"])

Expected input to match one of these values:
"DistanceMetric", "DistributionName", "NumSamples"
The input, dist, matched more than one valid value.

str = validatestring("Algorithm",["DistanceMetric","DistributionName","NumSamples"])

Expected input to match one of these values:
"DistanceMetric", "DistributionName", "NumSamples"
The input, "Algorithm", did not match any of the valid values.

```

Variable Number of Outputs

Similar to how you handle a variable number of inputs, you can use `nargout` to determine how many outputs were requested when a function was called. You can use `varargout` to represent any number of output arguments.

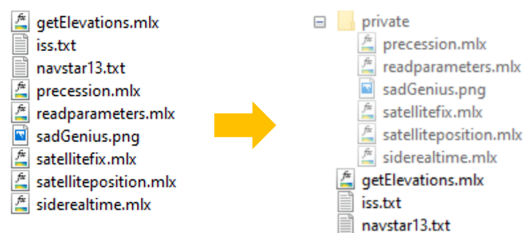
Function Call	Editor
<code>[x,y,z] = myfun(1,2,3)</code>	<code>function [a,varargout] = myfun(p,q,r)</code>

6. Creating Robust Applications

Summary - Creating Robust Applications

Private Functions

To prevent your application's internal functions from being accessible outside the application, you can place them in a folder named `private`.



A `private` folder is just a regular folder with the name `private`

Local Functions

To create local functions, place them below the primary function in the same code file.

```

findShapes.mlx
1. function findShapes %Primary function
2. ...
3. end
4.
5. function detectCircle %Local function
6. ...
7. end
8.
9. function detectSquare %Local function
10. ...
11. end

```

Generating Custom Warning and Errors

You can use the `warning` and `error` functions to generate custom warnings and errors.

<u>warning</u> Generate a custom warning message	
<u>error</u> Generate a custom error message	
Error	Warning
<pre>error("MyProject:invalidValue",... "The value must be numeric.");</pre> <p>The value must be numeric.</p>	<pre>warning("MyProject:ValueOutOfRange",... "The value must be between 1 and 10.");</pre> <p>The value was expected to be between 1 and 10.</p>

Validating Function Inputs

To check multiple attributes of a given input, use the function `validateattributes`.

<u>validateattributes</u> Check multiple attributes of a given variable.	
Command Window	Editor
<pre>int = calculateInterest(500,2,-0.03)</pre> <p>Expected input to be a scalar with value >= 0.</p>	<pre>function interest = calculateInterest(p,n,r) validateattributes("r","numeric",... {'scalar','>=',0}) ...</pre>

Catching and Handling Errors

The `try/catch` construct attempts to run the code within the `try` block. If an error condition occurs, then the execution is interrupted and switched immediately into the `catch` block.

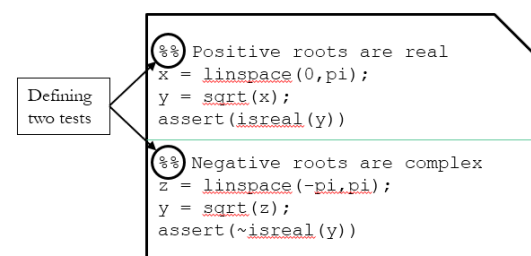
```
try
    % Attempt code
catch mexc
    % Backup case - something went wrong
    % An MException object called mexc now exists
end
```

7. Verifying Application Behavior

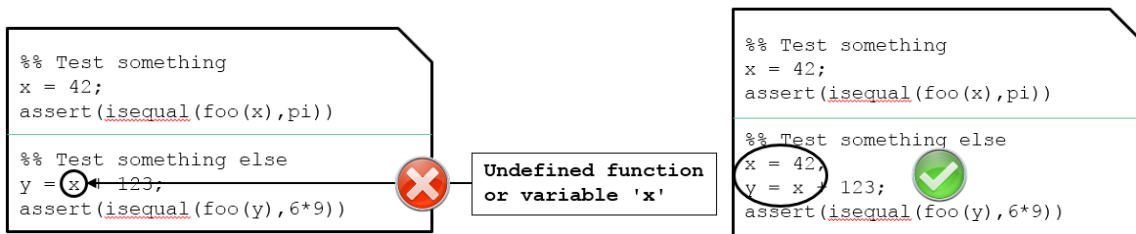
Writing a Test Script

A test script is a MATLAB script that checks that the outputs of MATLAB files are as you expect. To create a test script, use the `assert` function to create tests and separate the tests into sections.

Code sections allow you to compile multiple tests into a single file. The section header describes the test.



Each section of a test script maintains its own workspace. Thus, in each section, you must define all needed variables.



Creating a Test Function

A test function follows a specific template:

The name of the main function:

- starts or ends with the case-insensitive word "test"
- has the same name as the function file.

The main function has

- no inputs
- one output, here testA. The variable testA is a test array created by functiontests with the input localfunctions.

The local functions:

- contain the tests
- have names that start or end with the case-insensitive word "test"
- have a single input, here testCase

Many simple function-based unit tests will not use the input to the local functions, but it should always be included. Tests that verify behavior (e.g. warnings or errors) or allow for shared variables between tests use this input explicitly.

```
function testA = mainFunctionNameTest
testA = functiontests(localfunctions);
end

function testName1(testCase)
% code for test 1
end

function testName2(testCase)
% code for test 2
end
```

Each test goes in its own local function.

You run the test function the same way you run a test script.

```
result = runtests("mainFunctionNameTest");
```

Verification Functions


There are many qualification functions available in the function-based testing framework.

Verification functions available for function-based tests

verifyClass	verifyNotEmpty
verifyEmpty	verifyNotEqual
verifyEqual	verifyNotSameHandle
verifyError	verifyNumElements
verifyFail	verifyReturnsTrue
verifyFalse	verifySameHandle
verifyGreaterThan	verifySize
verifyGreaterThanOrEqual	verifySubstring
verifyInstanceOf	verifyThat
verifyLength	verifyTrue
verifyLessThan	verifyWarning
verifyLessThanOrEqual	verifyWarningFree
verifyMatches	

The local test functions have a single input, named `testCase` in the example below. Your test code should use it as the first input to the verification function.

```
function testPositiveRootsAreReal(testCase)
x = linspace(0,pi);
y = sqrt(x);
verifyTrue(testCase,isreal(y))
end
```



Pre- and Post-Test Tasks

You can add pre- and post-test tasks to your testing function by adding functions with the special names `setupOnce` and `teardownOnce` which have the same signature as the other test functions.

In this example, the test code and the application code are in separate locations. You will use the application folder in the `teardown` function. So, rather than redefining it, you can save it in the `testCase` variable to the `TestData` property. This property is a structure variable to which you can add fields as you desire. Any fields that are added to this variable are accessible by all functions in the test function file.

Now, remove the directory from the path in the `teardownOnce` function. See that the variable you previously defined, `appDir`, is accessible through `testCase.TestData`. Note that the structure `TestData` is named automatically, whereas you can name the function input, `testCase` in this example, whatever you want.

```
function setupOnce(testCase)
```

```
% save the name of the application folder you will add  
% to the TestData structure  
testCase.TestData.appDir = ...  
    fullfile("C:", "class", "work", "ApplicationDirectory");
```

```
% add the application folder to the path for testing  
addpath(testCase.TestData.appDir)
```

```
end
```

```
function teardownOnce(testCase)
```

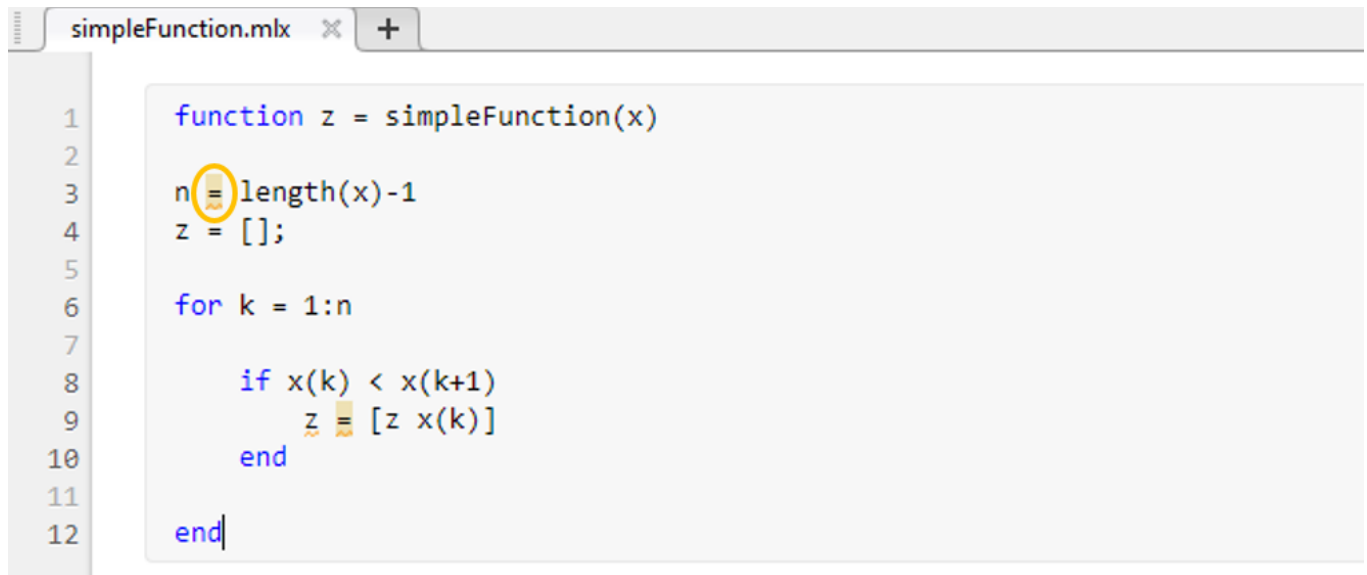
```
% removes the added path  
rmpath(testCase.TestData.appDir)
```

```
end
```

8. Debugging Your Code

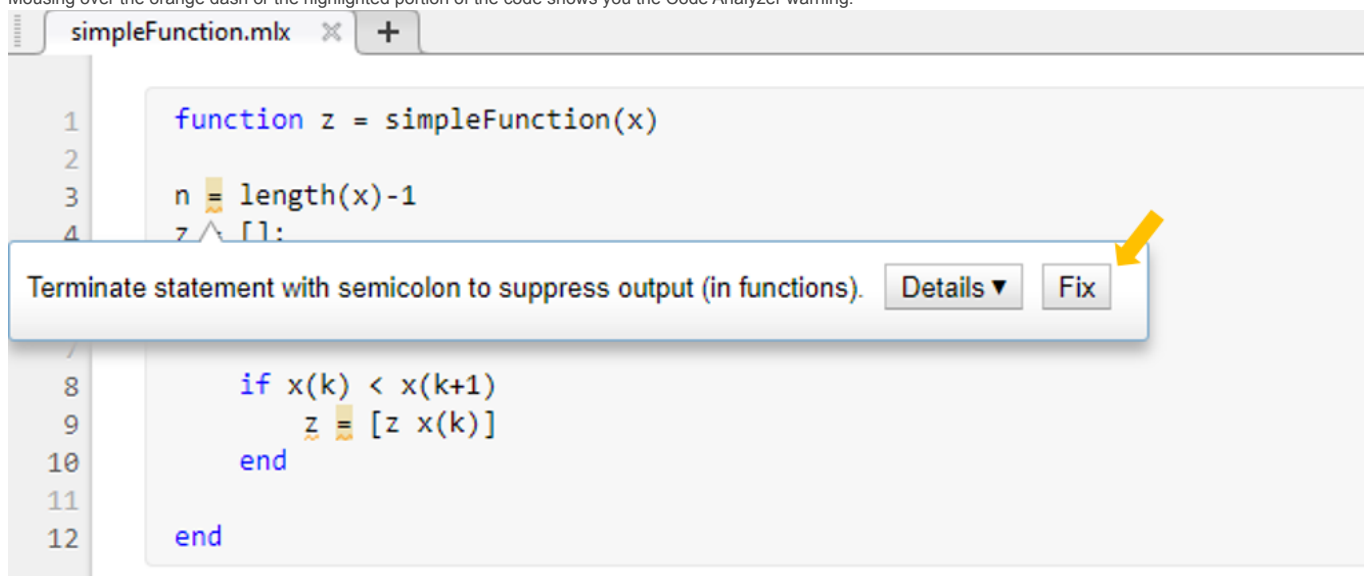
Suppressing and Fixing Code Analyzer Warnings

For some warnings, the Code Analyzer can help you automatically fix the code.



```
1 function z = simpleFunction(x)
2
3 n = length(x)-1
4 z = [];
5
6 for k = 1:n
7
8     if x(k) < x(k+1)
9         z = [z x(k)]
10    end
11
12 end
```

Mousing over the orange dash or the highlighted portion of the code shows you the Code Analyzer warning.

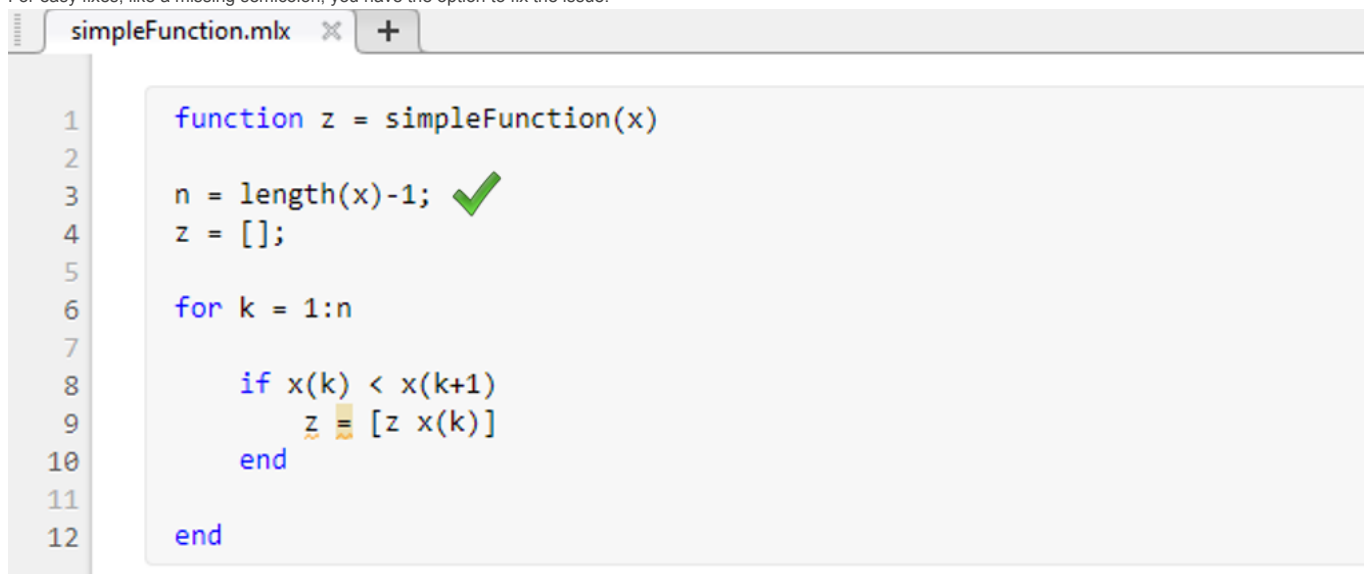


```
1 function z = simpleFunction(x)
2
3 n = length(x)-1
4 z = [];
```

Terminate statement with semicolon to suppress output (in functions). Details ▾ Fix

```
7
8     if x(k) < x(k+1)
9         z = [z x(k)]
10    end
11
12 end
```

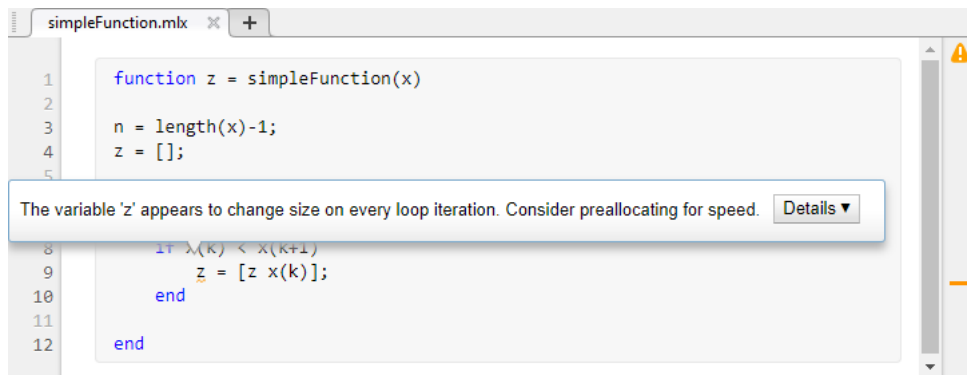
For easy fixes, like a missing semicolon, you have the option to fix the issue.



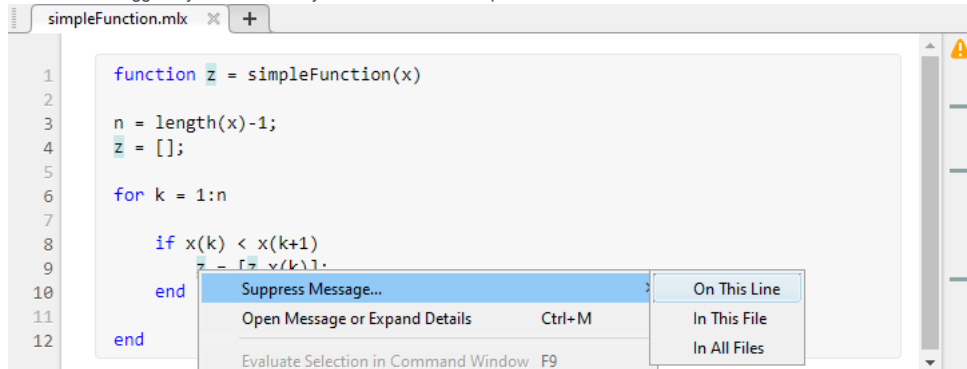
```
1 function z = simpleFunction(x)
2
3 n = length(x)-1; ✓
4 z = [];
5
6 for k = 1:n
7
8     if x(k) < x(k+1)
9         z = [z x(k)]
10    end
11
12 end
```

Semicolon is added, and warning is removed

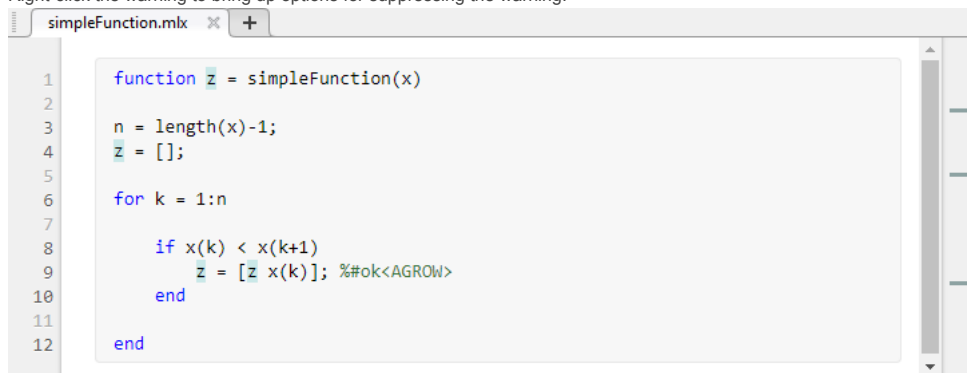
When the Code Analyzer does not provide an automatic fix, you can either fix the warning yourself, or suppress it.



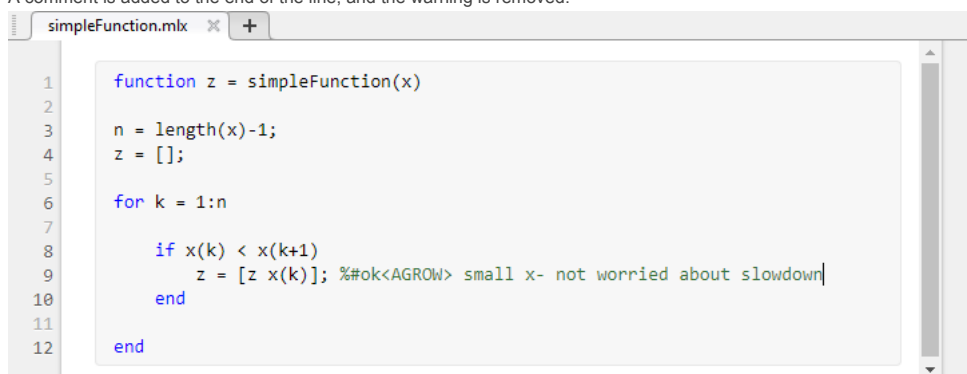
Other issues flagged by the Code Analyzer do not have a simple fix.



Right-click the warning to bring up options for suppressing the warning.



A comment is added to the end of the line, and the warning is removed.

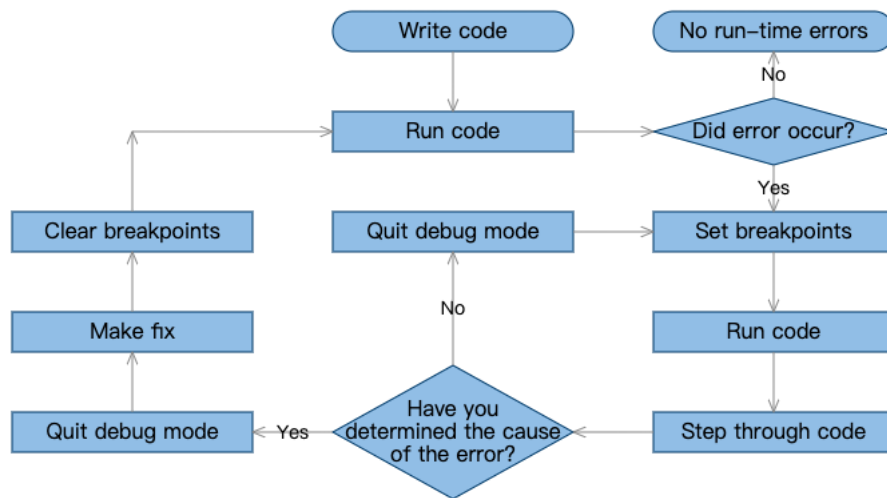


It is a good idea to add further comments to explain why you've suppressed the message.

Suppressing warnings can be a helpful way to demonstrate that the issue has been considered. This means developers in the future won't spend time worrying about a nonexistent problem with the code.

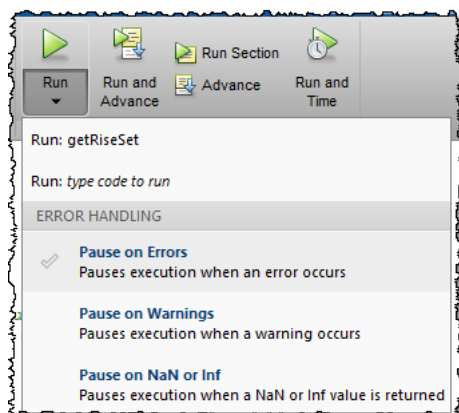
Debugging Run-Time Errors

When debugging MATLAB code, a common workflow is as follows.



Note that, before making a fix to your code, you should always stop your debugging session first so that you can save your changes.

If you want to enter debug mode immediately when an uncaught error is generated, you can enable the **Pause on Errors** option.



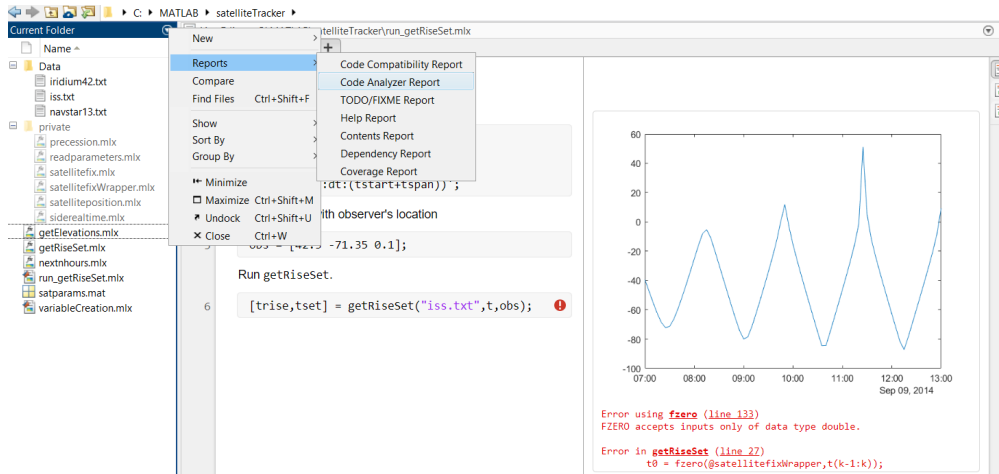
Automatically break on an error

9. Organizing Your Projects

Summary - Organizing Your Projects

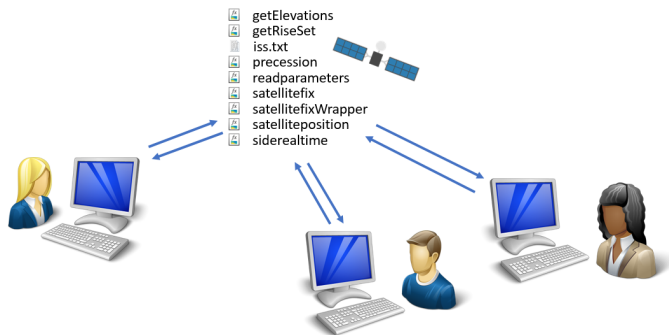
Running Folder Reports

To run a folder report on the current folder, access the Reports menu from the Current Folder Actions menu.



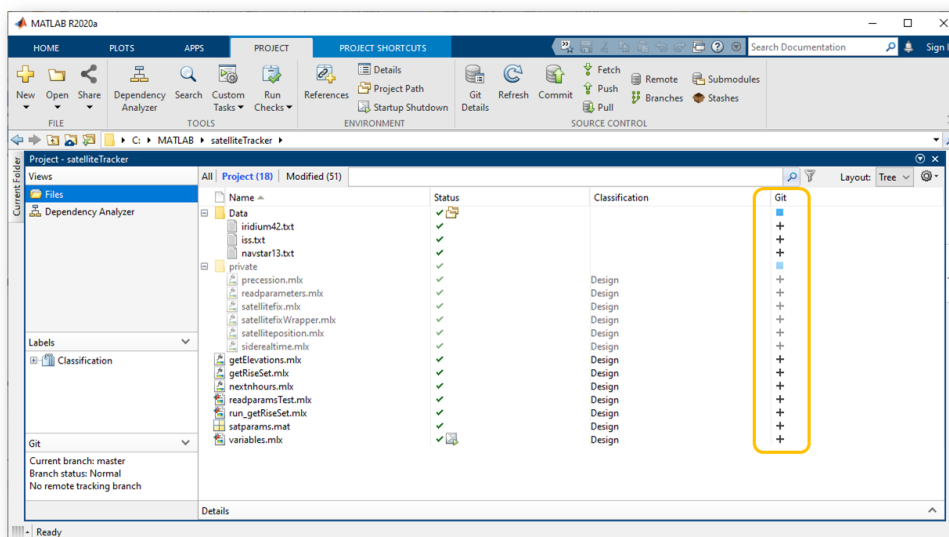
MATLAB Projects

MATLAB Projects help manage code spread across multiple files. They are also using for sharing code with others.



Source Control

Source control enables you to keep track of and manage previous versions of your project.



This course uses Git with MATLAB Projects.

10. Conclusion