CHAPTER 5

■ ■ ■

Testing and Debugging

The MATLAB unit test framework now allows you to incorporate testing into your MATLAB software, just as you would your C++ or Java packages. Since entire textbooks have been written on testing methodologies, this chapter is limited to covering the mechanics of using the test framework itself. We also present a couple of recipes that are useful for debugging.

We should, however, say a few words about the goal of software testing. Testing should determine if your software functions as designed. The first step is to have a concrete design against which you are coding. The functionality needs to be carefully described as a set of requirements. The requirements need to specify which inputs the software expects and what outputs it will generate. Testing needs to verify that for all valid inputs, it generates the expected outputs. A second consideration is that the software should handle expected errors and warn the user. For example, a simple function adds two MATLAB variables:

```
c = a + b;
```

You need to verify that it works for any numeric a and b. You generally do not need to warn the user if a or b is not numeric; that would just fill your code up with unneeded tests. A case where you might want a check is a function containing

```
b = acos(a);
```

If it is supposed to return a real number (perhaps as part of another function), you might want to limit a to have a magnitude less than 1. If you have the code,

```
if( abs(a) > 1 )
  a = sign(a);
end
b = acos(a);
```

In this case, your test code needs to pass in values of a that are greater than one. This is also a case where you might want to add a custom warning to the user if the magnitude limiting code is exercised, as shown next. If you have custom warnings and errors in your code, you also need to test them.

```
if( abs(a) > 1 )
  warning('MyToolbox:MyFunction:OutOfBounds','Input_a_is_out_of bounds');
  a = sign(a);
end
b = acos(a);
```

For engineering software, your test code should include known outputs generated by known inputs. In the preceding code, you might include inputs of 1.1, 1, 0.5, 0, −0.5 − 1, and −1.1. This would span the range of expected inputs. You might also be very thorough and input linspace (−1.1, 1.1) and test against another source of values for the inverse cosine. As shown in the later chapters, we usually include a demo function that tests the function with an interesting set of inputs. Your test code can use the code from the demo function as part of the testing.

All test procedures should employ the MATLAB code coverage tools. The Coverage Report, used in conjunction with the MATLAB Profiler, keeps track of which lines of code are exercised during execution. For a given function or script, it is essential that all code be exercised in its test. Studies have shown that testing done without coverage tools typically exercises only 55 % of the code. In reality, it is impossible to actually test every path in anything but the simplest software, and this must be factored into the software development and quality assurance processes. MATLAB does not currently support running the coverage tools on a suite of tests, or during your regression testing, so you should exercise the coverage tools on a per-test basis as you design them.

Once you start using your software, any bug you find should be used to add an additional test case to your tests.

5-1. Creating a Unit Test

Problem

Your functions require unit tests.

Solution

Use MATLAB's built-in test capabilities (now available using Java classes) to write and execute unit test functions. Test functions and scripts are identified by using the word *test* as a prefix or suffix to the file name, and are run via the runtests function.

How It Works

The matlab.unittest package is an xUnit-style, unit-testing framework for MATLAB. You can write scripts with test cases separated using cell titles, or functions with test cases in subfunctions, and execute them using the framework. We will show an example of each. There is extensive documentation of the framework and examples in the MATLAB documentation; these lists will get you started:

These are the relevant MATLAB packages implementing the framework:

- matlab.unittest

- matlab.unittest.constraints

- matlab.unittest.fixtures

- matlab.unittest.qualifications

The qualifications package provides all the methods for checking function results, including numerical values, errors, and warnings. The fixtures package allows you to provide setup and teardown code for individual tests or groups of tests.

Here are the relevant classes you will use when coding tests:

- matlab.unittest.TestCase
- matlab.unittest.TestResult
- matlab.unittest.TestSuite
- matlab.unittest.qualifications.Verifiable

TestCase is the superclass for writing test classes.
Here are the relevant functions:

- assert
- runtest
- functiontests
- localfunctions

The simplest way to implement tests for a function is to write a script. Each test case is identified with a cell title, using %%. Use the assert function to check function output. The script can then be run via runtest, which runs each test, even if a prior test fails, and collates the output into a useful report.

Let's write tests for an example function, CompleteTriangle, that computes the remaining data for a triangle given two sides and the interior angle:

```
function [A,B,c] = CompleteTriangle (a,b,C)

c = sqrt(a^2 + b^2 - 2*a*b*cosd(C));
sinA = sind(C)/c*a;
sinB = sind(C)/c*b;
cosA = (c^2+b^2 - a^2)/2/b/c;
cosB = (c^2+a^2-b^2)/2/a/c;
A = atan2(sinA,cosA)*180/pi;
B = atan2(sinB,cosB)*180/pi; % insert typo: change a B to A

end
```

This is similar to the right triangle function used as an example in the MATLAB documentation, but you need the four quadrant inverse tangent, as you are allowing obtuse triangles. Since there are very similar lines of code for the two angles, A and B, there is a note that having a typo in one of these lines would be likely, especially if you use copy/paste while writing the function; we'll demonstrate the effect of such a typo via our tests.

Now let's look at a script that defines a few test cases for this function, TriangleTest. Each check uses assert with a logical statement.

```
%% Test1: sum of angles
% Test that the angles add up to 180 degrees.
C = 30;
[A,B] = CompleteTriangle(1,2,C);
theSum = A+B+C;
assert(theSum == 180,'PSS:Book:triangle','sum_of_angles:_%f',theSum)
```

```
%% Test 2: isosceles right triangles
% Test that if sides a and b are equal, angles A and B are equal.
C = 90;
[A,B] = CompleteTriangle(2,2,C);
assert(A ==B,'PSS:Book:triangle','Isoceles_Triangle')

%% Test 3: 3-4-5 right triangle
% Test that if side a is 3 and side b is 4, side c (hypotenuse) is 5.
C=90;
[˜,˜,c] = CompleteTriangle(3,4,C);
assert(c == 5,'PSS:Book:triangle','3-4-5_Triangle')

%% Test 4: equilateral triangle
% Test that if sides a and b are equal, all angles are 60.
[A, B, c] = CompleteTriangle(1,1,60);
assert(A == 60,'PSS:Book:triangle','Equilateral_Triangle_%d',1)
assert(B == 60, 'PSS:Book:triangle','Equilateral_Triangle_%d',2)
assert(c == 1,'PSS:Book:triangle','Equilateral_Triangle_%d',3)
```

Note how the additional inputs available to assert are used to add a message ID string and an error message. The error message can take formatted strings with any of the specifiers supported by sprintf, such as %d and %f.

You can simply execute this script, in which case it exits on the first assert that fails. Even better, you can run it with runtests, which automatically distinguishes between the test cases and runs them independently should one fail.

```
>> runtests('TriangleTest');

Running TriangleTest
...
================================================================================
Error occurred in TriangleTest/Test4_EquilateralTriangle and it did not run to
    completion.

    --------------
    Error Details:
    --------------
    Equilateral Triangle 1
================================================================================
.
Done TriangleTest
_____


Failure Summary:
    Name                                Failed   Incomplete   Reason(s)
    ======================================================================
    TriangleTest/Test4_EquilateralTriangle   X         X        Errored.
```

The equilateral triangle test failed. You can tell it was the first `assert` in that case due to the index that was printed, `Equilateral Triangle 1`. If you run the code for that test at the command line, you see that the output does, in fact, look correct:

```
>> [A,B,c] = CompleteTriangle (1,1,60)
A =
            60
B =
            60
c =
     1
```

If you actually subtract the expected value, 60, from A and B, you see why the test has failed.

```
>> A - 60
ans =
    7.1054e - 15
>> B - 60
ans =
    7.1054e - 15
```

You are within the tolerances of the trigonometric functions in MATLAB, but the assert statement did not take that into account. You can add a tolerance, like so:

```
assert(abs (A - 60) < 1e - 10,'PSS:Book:triangle', 'Equilateral_Triangle_%d',1)
assert(abs (B - 60) < 1e - 10,'PSS:Book:triangle','Equilateral_Triangle_%d',2)
```

And now the tests all pass:

```
>> runtests('TriangleTest')
Running TriangleTest
....
Done TriangleTest
_____


ans  =
  1×4 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
Totals:
   4 Passed, 0 Failed, 0 Incomplete.
   0.012243 seconds testing time.
```

Note that the terminating semicolon is left off; so in addition to the brief report, you see that `runtests` returns an array of `TestResult` objects and prints additional totals information, including the test duration.

Now let's consider the case of a typo in the function that you have not yet debugged. You change a B to an A on the last line of the function, so that it reads:

```
B = atan2(sinB, cosA)*180/pi; % insert typo: change a B to A
```

Run the tests again, using the tolerance check. Use the table class with the TestResult output to get a nicely formatted version of the test results.

```
>> tr = runtests('TriangleTest');
>> table(tr)
ans =
```

Name	Passed	Failed	Incomplete	Duration
'TriangleTest/Test1_SumOfAngles'	False	true	true	0.0040209
'TriangleTest/Test2_IsoscelesRightTriangles'	true	false	false	0.002971
'TriangleTest/Test3_3_4_5RightTriangle'	true	false	false	0.0027831
'TriangleTest/Test4_EquilateralTriangle'	true	false	false	0.0031556

Despite this being a major error in the code, only one test has failed: the sum of angles test. The isosceles and equilateral triangle tests still passed because A and B are equal in both cases. You could introduce errors into each line of your code to see if your tests catch them!

Now let's consider the other possibility for the unit tests: a test function, as opposed to the script. In this case, each test case has to be in its own subfunction, and the main function has to return an array of test function objects. This provides you the opportunity to write setup and teardown functions for the tests. It also makes use of the TestCase class and the qualifications package. Here is what the tests look like in this format:

```
function tests = TriangleFunctionTest
% Create an array of local functions
tests = functiontests(localfunctions);
end

%% Test Functions
function testAngleSum(testCase)%#ok<*DEFNU>
C      = 30;
[A,B]  = CompleteTriangle(1, 2, C);
theSum = A+B+C;
testCase.verifyEqual (theSum,180)
end

function testIsosceles(testCase)
C      = 90;
[A,B] = CompleteTriangle(2,2,C);
testCase.verifyEqual(A,B)
end

function test345(testCase)
C       = 90;
[~,~,c] = CompleteTriangle(3,4,C);
testCase.verifyEqual(c,5)
end
```

```
function testEquilateral(testCase)
[A,B,c] = CompleteTriangle(1,1,60);
assert(abs(A-60)<testCase.TestData.tol)
testCase.verifyEqual(B,60,'absTol',1e-10)
testCase.verifyEqual(c,1)
end

%% Optional file fixtures
function setupOnce(testCase)  % do not change function name
% set a tolerance that can be used by all tests
testCase.TestData.tol=1e-10;
end

function teardownOnce(testCase)  % do not change function name
% change back to original path, for example
end

%% Optional fresh fixtures
function setup(testCase)  % do not change function name
% open a figure, for example
end

function teardown(testCase)  % do not change function name
% close figure, for example
end
```

If you just run this function, you get an array of the four test methods.

```
>> TriangleFunctionTest
ans =
  1×4 Test array with properties:

    Name
    Parameterization
    SharedTestFixtures
```

Two methods for setting a tolerance for the tests in testEquilateral have been shown; in one case, you hardcode a tolerance in using the absTol parameters, and in the other, you use a setup function to pass a tolerance in via TestData. There are two types of setup and teardown functions to choose from: *file* fixtures, which run just once for the entire set of tests in the file, and *fresh* fixtures, which run for each test case. The file fixtures are identified with the *Once* suffix. In the case of this tolerance, the setupOnce function is appropriate.

To run the tests, use runtests as for the script. Happily, the tests all pass!

```
>> runtests('TriangleFunctionTest')
Running TriangleFunctionTest
....
Done TriangleFunctionTest


...
Totals:
   4 Passed, 0 Failed, 0 Incomplete.
   0.043001 seconds testing time.
```

You can run either set of tests in the Profiler (i.e., Run and Time) to verify coverage of the function being tested. It is a bit easier to navigate to the results for `CompleteTriangle` using the script version of the tests; the results from the test function lists many functions from the test framework. The result in the Profiler, showing 100 % coverage of the function, is shown in Figure 5-1.
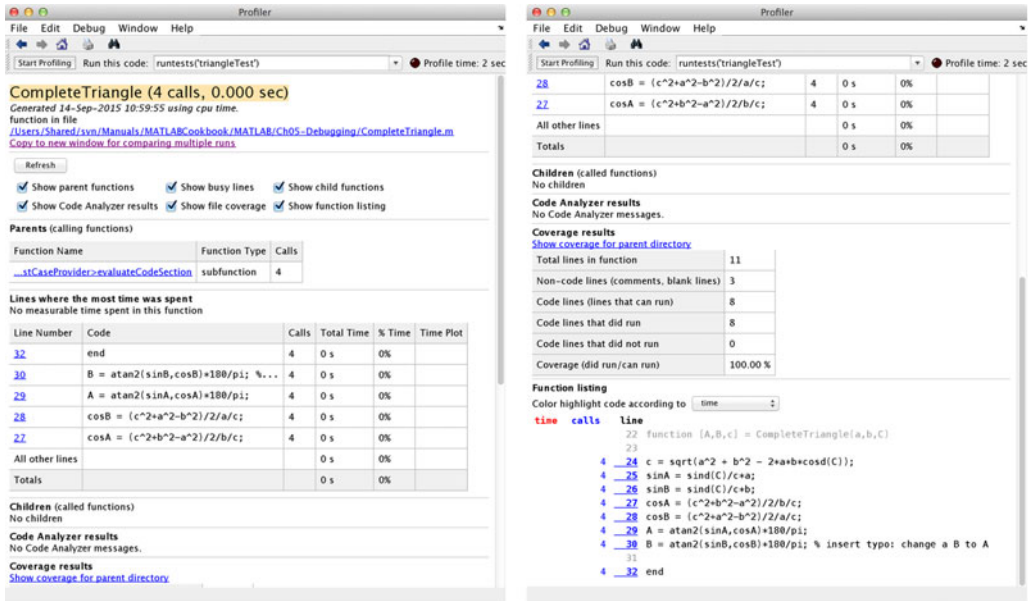

Figure 5-1. *Triangle tests in the Profiler*

After you have run the Profiler, you can run a Coverage `Report`. To run the report, you have to use the Current Folder pane of the Editor, and select `Reports/Coverage Report` from the context menu. An example is shown in Figure 5-2. The example function runs too quickly to take any measurable time, but generally, this report gives you insight into the time taken by your function, as well as the coverage you achieved.
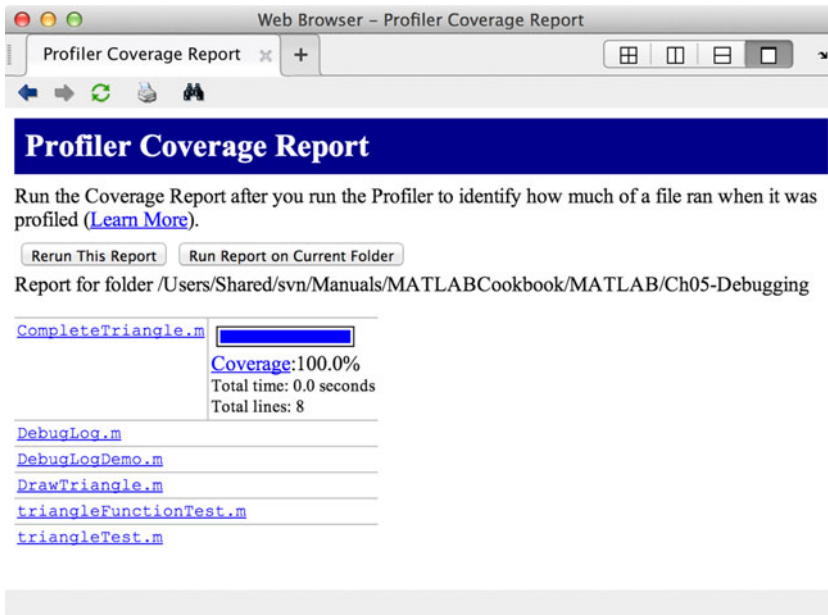
Figure 5-2. *Coverage Report for CompleteTriangle*

5-2. Running a Test Suite

Problem

Your toolbox has dozens or hundreds of functions, each with unit tests. You need an efficient way to run them all, or even better, run subsets.

Solution

MATLAB's test framework includes the construction of test suites.

How It Works

After you have generated tests for the functions in your toolbox, you can group them into suites in several ways. The help for the TestSuite class lists the options:

```
TestSuite methods:
    fromName    - Create a suite from the name of the test element
    fromFile    - Create a suite from a TestCase class filename
    fromFolder  - Create a suite from all tests in a folder
    fromPackage - Create a suite from all tests in a package
    fromClass   - Create a suite from a TestCase class
    fromMethod  - Create a suite from a single test method
```

You can also concatenate test suites made using these methods and pass the array to the test runner. In this way, you can easily generate subsets of your tests to run.

In the previous recipe, you created two test files for `CompleteTriangle`: a test script and a test function. You can create a test suite for the folder containing this code, and it will automatically find both sets of test cases. Assume that the current folder contains the two test files.

```
>> import matlab.unittest.TestSuite
>> testSuite = TestSuite.fromFolder(pwd);
>> result = run(testSuite)

Running TriangleFunctionTest
....
Done TriangleFunctionTest
_____


Running TriangleTest
.......
Done TriangleTest
_____
result =
  1×8 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
Totals:
   8 Passed, 0 Failed, 0 Incomplete.
   0.04218 seconds testing time.
```

As you can see, test suites are really quite simple. Some advanced features include the ability to apply selectors to a suite to obtain a subset of tests. To see the full documentation of `TestSuite` at the command line, type either

```
>> help matlab.unittest.TestSuite
```

or

```
>> import matlab.unittest.TestSuite
>> help TestSuite
```

The function for performing selections is `select If`. Here is an example that selects the two tests of an equilateral triangle from the suite:

```
>> subSuite = testSuite.selectIf('Name','*Equilateral*');
>> subSuite
subSuite =
  1×2 Test array with properties:

    Name
    Parameterization
    SharedTestFixtures
```

```
>> subSuite.Name
ans =
TriangleFunctionTest/testEquilateral
ans =
TriangleTest/Test4_EquilateralTriangle
```

You can run the tests in the resulting suite, or concatenate with other suites, as before.

5-3. Setting Verbosity Levels in Tests
Problem

The printouts from your tests are getting out of control, but you don't want to just delete or comment out all the information you have needed as you are developing the tests. If a test fails in the future, you may need those messages.

Solution

The test framework includes a logging feature that has four levels of verbosity. To utilize it, you create a test runner using the logging plugin and add log calls in your test cases.

How It Works

The four verbosity levels supported are Terse, Concise, Detailed, and Verbose, which are enumerated in Table 5-1.

Table 5-1.

1	Terse	Minimal amount of information
2	Concise	Typical amount of information
3	Detailed	Supplemental amount of information
4	Verbose	Surplus of information

The default test runner uses the lowest verbosity setting, Terse. The log function you use in your test cases is a method of TestCase, so to access the help, you need to use the fully qualified name:

```
>> help matlab.unittest.TestCase/log
```

The following is the log method syntax from the help:

> log(TESTCASE, LEVEL, DIAG) logs the diagnostic at the specified LEVEL. LEVEL
> can be either a numeric value (1, 2, 3, or 4) or a value from the matlab.unittest.
> Verbosity enumeration. When level is unspecified, the log method uses level
> Concise (2).

Logging requires a `TestCase` object. The diagnostic data for `DIAG` can be a string or an instance of `matlab.unittest.diagnostics.Diagnostic`. Let's write an example test for eig that demonstrates verbosity.

```matlab
%% VerboseEigTest Demonstrate verbosity levels in tests
% Run a test of the eig function using log messages. Demonstrates
% all four levels of verbosity. To run the tests, at the command line use
% a TestRunner configured with the LoggingPlugIn:
%
%   import matlab.unittest.TestRunner;
%   import matlab.unittest.plugins.LoggingPlugin;
%   runner = TestRunner.withNoPlugins;
%   runner.addPlugin(LoggingPlugin.withVerbosity(4));
%   results = runner.run(VerboseEigTest);
%% Form
%   tests = VerboseEigTest
%% Inputs
% None.
%% Outputs
%   tests (:) Array of test functions

function tests = VerboseEigTest
% Create an array of local functions
tests = functiontests (localfunctions);
end


%% Test Functions
function eigTest (testCase)
log(testCase,'Generating_test_data'); % default is level 2
m = rand(2000);
A = m'*m;
log(testCase,'About_to_call_eig.');
[V,D,W] = eig(A);
log(testCase, 4, 'Eig_finished.');
assert(norm(W'*A-D*W')<1e-6)
log(testCase,3, 'Test_of_eig_completed.');
end


% If you want to use the Verbose enumeration in your code instead of numbers,
% import the class matlab.unittest.Verbosity
function eigWithEnumTest(testCase)
import matlab.unittest.Verbosity
m = rand(1000);
A = m'*m;
log(testCase, Verbosity.Detailed, 'About_to_call_eig_(with_enum).');
[V,D,W] = eig(A);
assert(norm(W'*A-D*W')<1e-6)
log(testCase, Verbosity.Terse, 'Test_of_eig_(with_enum)_completed.');
end
```

If you just run this test with runtests, you get the Terse level of output. Note that the system time is displayed along with your log message.

```
>> runtests('VerboseEigTest');
Running VerboseEigTest
   [Terse] Diagnostic logged (2015-09-14 T12:15:29): About to call eig.
.   [Terse] Diagnostic logged (2015-09-14 T12:15:40): Test of eig (with enum) completed
.
Done VerboseEigTest
```
_____

To get a higher level of verbosity requires a test runner with the logging plugin. This requires a few imports at the command line (or in your script). You need to generate a "plain" runner, with no plugins, and then add the logging plugin with the desired level of verbosity. The verbosity level of the message is displayed in the output.

```
>> import matlab.unittest.TestRunner;
>> import matlab.unittest.plugins.LoggingPlugin;
>> runner = TestRunner.withNoPlugins;
>> runner.addPlugin (LoggingPlugin.withVerbosity(4));
>> results = runner.run(VerboseEigTest);
  [Concise] Diagnostic logged (2015-09-14 T12:19:57): Generating test data
    [Terse] Diagnostic logged (2015-09-14 T12:19:57): About to call eig.
  [Verbose] Diagnostic logged (2015-09-14 T12:20:01): Eig finished.
 [Detailed] Diagnostic logged (2015-09-14 T12:20:07): Test of eig completed.
 [Detailed] Diagnostic logged (2015-09-14 T12:20:07): About to call eig (with enum).
    [Terse] Diagnostic logged (2015-09-14 T12:20:08): Test of eig (with enum) completed.
```

5-4. Create a Logging Function to Display Data

Problem

It is easy and convenient to print variable values by removing the semicolons from statements, but code left in this state can produce unwanted printouts that are very difficult to track down. Even using disp and fprintf can make unwanted printouts hard to find, because you probably use these functions elsewhere.

Solution

Create a custom logging function to display a variable with a helpful identifying message. You can extend this to a logging mechanism with verbosity settings similar to that described in the previous recipe, as used in the MATLAB testing framework, and in most C++ and Java testing frameworks.

How It Works

An example logging function is implemented in DebugLog. DebugLog prints out a message, which can be anything, and before that displays, the path to where DebugLog is called. The backtrace is obtained using dbstack.

```
%% DEBUGLOG Logging function for debugging
% Use this function instead of adding disp() statements or leaving out semicolons.
%% Form
%   DebugLog( msg, fullPath )
%% Description
% Prints out the data in in msg using disp() and shows the path to the message.
% The full path option will print a complete backtrace.
%% Inputs
%   msg         (.)      Any message
%   fullPath    (1,1)    If entered, print the full backtrace
%% Outputs
%   None

function DebugLog( msg, fullPath )

% Demo
if( nargin < 1 )
  DebugLog(rand(2,2));
  return;
end

% Get the function that calls this one
f = dbstack;

% The second path is only if called directly from the command line
if(length(f) > 1)
  f1 = 2;
else
  f1 = 1;
end

if( nargin > 1 && fullPath )
  f2 = length(f);
else
  f2 = f1;
end

for k = f1:f2
  disp(['- >␣' f(k).name]);
end
disp(msg);
```

DebugLog is demonstrated in DebugLogDemo. The function has a subfunction to demonstrate the backtrace.

```
%% Demonstrate DebugLog
% Log a variable to the command window using DebugLog.

function DebugLogDemo
```

```
y = linspace(0,10);
i = FindInY(y);

function I = FindInY(y)

i = find(y < 0.5);
DebugLog( i, true );
```

This the output of the demo:

```
>> DebugLogDemo
-> FindInY
-> DebugLogDemo
     1     2     3     4     5
```

One extension of this function is to add the name of the variable being logged, if msg is a variable, using the function inputname. This additional lines of code look like this:

```
str = inputname(1);
if ~isempty(str)
  disp(['Variable: ' str]);
end
```

The demo output now looks like this:

```
>> DebugLogDemo
-> FindInY
-> DebugLogDemo
Variable: i
     1     2     3     4     5
```

Consistently using your own logging functions for displaying messages to the user and printing debug data will make your code easier to maintain.

5-5. Generating and Tracing MATLAB Errors and Warnings
Problem

You would like to display errors and warnings to the user in an organized fashion.

Solution

Always use the additional inputs to warning and error to specify a message ID. This allows your message to be traced back to the function in the code that generated it, as well as controlling the display of certain warnings.

How It Works

The warning function has several helpful parameters for customizing and controlling warnings displays. When you are generating a warning, use the full syntax with a message identifier:

```
warning('MSGID','MESSAGE', A, B, ...)
```

The MSGID is a mnemonic in the form <component>[:<component>]:<mnemonic>, such as PSS:FunctionName:IllegalInput. The ID is not normally displayed when you give a warning, unless you have turned verbose display on, via warning on verbose and warning off verbose. This is easy to demonstrate at the command line:

```
>> warning('PSS:Example:DemoWarning', 'This_is_an_example_warning')
Warning: This is an example warning
>> warning verbose on
>> warning('PSS:Example:DemoWarning','This_is_an_example_warning')
Warning: This is an example warning
(Type "warning off PSS:Example:DemoWarning" to suppress this warning.)
```

As displayed, you can turn a given warning off using its message ID by using the command form shown or the functional form, warning('off','msgid').

The lastwarn function can also return the message ID if passed an additional output, as in:

```
>> [lastmsg, lastid] = lastwarn
lastmsg =
This is an example warning
lastid =
PSS:Example:DemoWarning
```

The error and lasterr functions work the same way. An added benefit of using message identifiers is that you can select them when debugging, as an option when stopping for errors or warnings. The debugger is integrated into the editor window, and the debugger options are grouped under the Breakpoints toolbar button. The button and the More Options pop-up window are shown in Figure 5-3. In this case, we entered an example PSS message identifier.
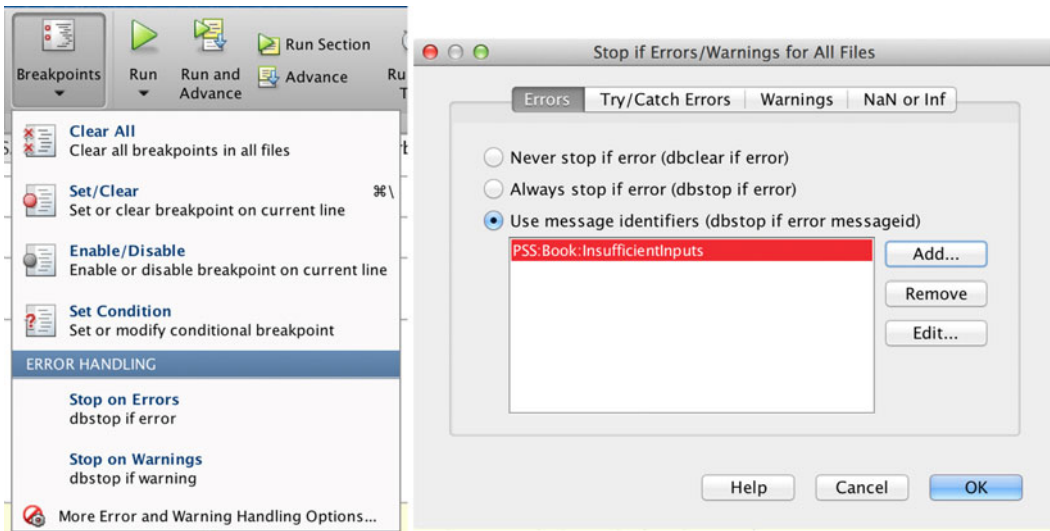

Figure 5-3. *Option to stop on an error in debugger*

Remember, you should always mention any warnings and errors that may be generated by a function in its header!

5-6. Testing Custom Errors and Warnings

Problem

You have code that generates warnings or errors for problematic inputs and you need to test it.

Solution

You have two possibilities for testing the generation of errors in your code: try/catch blocks with assert and the verifyError method available to a TestCase. With warnings, you can either use lastwarn or verifyWarning.

How It Works

A comprehensive set of tests for your code should include all paths, or as close to all paths, as possible; and it must exercise all the warnings and errors that can be generated by your code. You can do this manually by using try/catch blocks to catch errors and comparing the error (MException object) to the expected error. For warnings, you can check lastwarn to see that a warning was issued, like so:

```
>> lastwarn('');
>> warning('PSS:Book:id','Warning!')
Warning: Warning!
>> [anywarn, anyid] = lastwarn;
>> assert(strcmp (anyid, 'PSS:Book:wrongid'))
Assertion failed.
```

Here is an example of a try/catch block with assert to detect a specific error:

```
%% Test that we get the expected error, and pass
errFun = @() error('PSS:Book:id','Error!');
try
  feval(errFun);
catch ME
  assert(strcmp(ME.identifier,'PSS:Book:id'));
end
```

This test verifies that the error thrown is the one expected; however, it does not detect if no error is thrown at all. For this, you need to add a boolean variable to the try block.

```
%% This time we don't get any error at all
wrongFun = @() disp('some_error-free_code.');
tf = false;
try
  feval(wrongFun);
  tf = true;
catch ME
  assert(strcmp(ME.identifier,'PSS:Book:id'));
end
if (tf)
  assert(false,'CatchErrorTest:_No_error_thrown');
end
```

133

When you run this code segment, you get the following output:

```
Some error-free code.
CatchErrorTest: No error thrown
```

If you run the test as part of a test script with runtests, the test fails.

A far better way to test for warnings and errors is to use the unit test framework's qualifiers to check that the desired warning or error is generated. Here is an example of verifying a warning, with a test that will pass and a test that will fail; note that you need to pass a function handle to the verifyWarning function.

```
function tests = WarningsTest
% Create an array of local functions
tests = functiontests (localfunctions);
end

%% Test Functions
function passTest (testCase)
warnFun = @()warning('PSS:Book:id','Warning!');
testCase.verifyWarning(warnFun,'PSS:Book:id');
end

function failTest (testCase)
warnFun = @() warning ('Wrong:id','Warning!');
testCase.verifyWarning (warnFun,'PSS:id','Wrong_id');
end
```

When you run this test function with runtests, you can see that failTest did, in fact, fail.

```
>> runtests('WarningsTest')
Running WarningsTest
.Warning: Warning!

================================================================
Verification failed in WarningsTest/failTest.

    ----------------
    Test Diagnostic:
    ----------------
    Wrong id

    --------------------
    Framework Diagnostic:
    --------------------
    verifyWarning failed.
    --> The function handle did not issue the expected warning.

        Actual Warnings:
                Wrong:id
        Expected Warning:
                PSS:id
```

```
    Evaluated Function:
            @()warning('Wrong:id','Warning!')
    ------------------
    Stack Information:
    ------------------
    In /Users/Shared/svn/Manuals/MATLABCookbook/MATLAB/Ch05-Debugging/WarningsTest.m
        (failTest) at 12
=========================================================
.
Done WarningsTest

_____


Failure Summary:
     Name                    Failed   Incomplete   Reason(s)
     ===================================================================
     WarningsTest/failTest    X                    Failed by verification.

Totals:
    1 Passed, 1 Failed, 0 Incomplete.
    0.047691 seconds testing time.
```

verifyError works the same way. In practice, you need to make a function handle that includes the inputs that cause the error or warning to be generated.

For advanced programmers, there is a further mechanism for constructing tests using verifyThat with the Constraint class. You can supply your own Diagnostic objects as well. For more information, see the reference pages for these classes along with the Verifiable class.

5-7. Testing Generation of Figures

Problem

Your function generates a figure instead of an output variable. How do you test it?

Solution

Although you may need a human to verify that the figure looks correct, you can at least verify that the correct set of figures is generated by your function using findobj.

How It Works

Routinely assigning names to your figures makes it easy to test that they have been generated, even if you don't have access to the handles. You can also assign tags to figures, such as having a single tag for your entire toolbox, which allows you to locate sets of figures.

```
>> figure ('Name', 'Figure_1', 'Tag', 'PSS');
>> figure ('Name', 'Figure_2', 'Tag', 'PSS')
>> h = findobj('Tag', 'PSS')
h =
  2×1 Figure array:
```

```
  Figure    (PSS)
  Figure    (PSS)
>> h = findobj('Name','Figure_1')
h =
  Figure (PSS) with properties:

      Number: 1
        Name: 'Figure_1'
       Color: [0.94 0.94 0.94]
    Position: [440 378 560 420]
       Units: 'pixels'
```

In your test, you can then check that you have the correct number of figures generated using length(h) or that each specific named figure exists using strcmp. If you are storing any data in your figures using UserData, you can test that as well.

If you are not using tags or need to check for figures that do not have names or tags, you can find all figures currently open using the type input to findobj:

```
>> findobj('type','figure')
ans =
  2×1 Figure array:

  Figure    (PSS)
  Figure    (PSS)
```

Note that figures are only returned by findobj if they are visible to the command line via their HandleVisibility property. This property can have the values 'on', 'off', and 'callback'. GUIs generated by the design tool GUIDE are generally hidden to prevent users from accidentally altering the GUI using plot or similar commands; these figures use the value 'callback'. Regular figures have the value 'on' and can be located. A figure with HandleVisibility set to 'off' can only be accessed using its handle.

Summary

This chapter demonstrated how to use MATLAB's unit test framework and provided recipes to help you debug your functions. Table 5-2 lists the code files.

Table 5-2. *Chapter Code Listing*

File	Description
CatchErrorTest	Script showing how to catch errors in a try block.
CompleteTriangle	Example function calculating angles in a triangle.
DebugLog	Custom data logging function.
DebugLogDemo	Demo of DebugLog showing a backtrace.
TriangleFunctionTest	A function with test cases for CompleteTriangle.
TriangleTest	A script with test cases for CompleteTriangle.
VerboseEigTest	A test function showing all levels of verbosity.
WarningsTest	A test function using verifyWarning