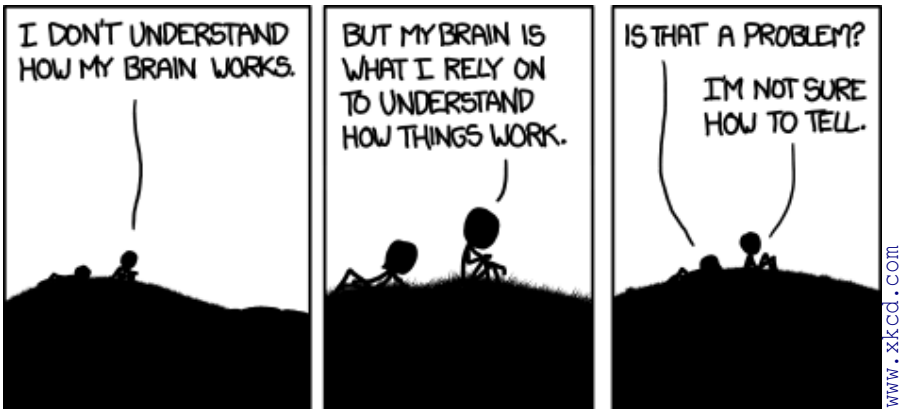
Chapter 2

Debugging



When writing a program from scratch we almost always make mistakes. Accordingly, a quite substantial amount of time is invested into finding and fixing errors. This process is called *debugging*. Don't be frustrated that a self-written program does not work as intended and produces errors. It is quite exceptional if a program appears to be working on the first try and, in fact, should leave you suspicious.

In this chapter we will talk about typical mistakes, how to read and understand error messages, how to actually debug your program code and some hints that help to minimize errors.

2.1 Types of errors and error messages

There are a number of different classes of programming errors and it is good to know the common ones. Some of your programming errors will will lead to violations of the syntax or to invalid operations that will cause MATLAB© to *throw* an error. Throwing an error ends the execution of a program and there will be an error messages shown in the command window. With such messages MATLAB© tries to explain what went wrong and to provide a hint on the possible cause.

Bugs that lead to the termination of the execution may be annoying but are generally easier to find and to fix than logical errors that stay hidden and the results of, e.g. an analysis, are seemingly correct.

Try — catch

There are ways to *catch* errors during *runtime* (i.e. when the program is executed) and handle them in the program.

Try catch clause

```
try
   y = function_that_throws_an_error(x);
catch
   y = 0;
end
```

This way of solving errors may seem rather convenient but is risky. Having a function throwing an error and catching it in the *catch* clause will keep your command line clean but may obscure logical errors! Take care when using the *try-catch clause*.

Syntax errors

The most common and easiest to fix type of error. A syntax error violates the rules (spelling and grammar) of the programming language. For example every opening parenthesis must be matched by a closing one or every `for` loop has to be closed by an `end`. Usually, the respective error messages are clear and the editor will point out and highlight most *syntax error*s.

Listing 2.2: Unbalanced parenthesis error.

```
>> mean(random_numbers
                      |
Error: Expression or statement is incorrect--possibly unbalanced (, {, or
    [.

Did you mean:
>>   mean(random_numbers)
```

Indexing error

Second on the list of common errors are the indexing errors. Usually MATLAB© gives rather precise infromation about the cause, once you know what they mean. Consider the following code.

Listing 2.3: Indexing errors.

```
>> my_array = (1:100);
>> % first try: index 0
>> my_array(0)
Subscript indices must either be real positive integers or logicals.

>> % second try: negative index
>> my_array(-1)
Subscript indices must either be real positive integers or logicals.
```

```
9
10 >> % third try: a floating point number
11 >> my_array(5.7)
12 Subscript indices must either be real positive integers or logicals.
13
14 >> % fourth try: a character
15 >> my_array('z')
16 Index exceeds matrix dimensions.
17
18 >> % fifth try: another character
19 >> my_array('A')
20 ans =
21      65 % wtf ?!?
```

The first two indexing attempts in listing 2.3 are rather clear. We are trying to access elements with indices that are invalid. Remember, indices in MATLAB© start with 1. Negative numbers and zero are not permitted. In the third attemp we index using a floating point number. This fails because indices have to be 'integer' values. Using a character as an index (fourth attempt) leads to a different error message that says that the index exceeds the matrix dimensions. This indicates that we are trying to read data behind the length of our variable `my_array` which has 100 elements. One could have expected that the character is an invalid index, but apparently it is valid but simply too large. The fith attempt finally succeeds. But why? MATLAB© implicitely converts the *char* to a number and uses this number to address the element in `my_array`. The *char* has the ASCII code 65 and thus the 65th element of `my_array` is returned.

Assignment error

Related to the Indexing error, an assignment error occurs when we want to write data into a variable, that does not fit into it. Listing 2.4 shows the simple case for 1-d data but, of course, it extents to n-dimensional data. The data that is to be filled into a matrix hat to fit in all dimensions. The command in line 7 works due to the fact, that matlab automatically extends the matrix, if you assign values to a range outside its bounds.

Listing 2.4: Assignment errors.

```
1  >> a = zeros(1, 100);
2  >> b = 0:10;
3
4  >> a(1:10) = b;
5       In an assignment  A(:) = B, the number of elements in A and B must be
            the same.
6
7  >> a(100:110) = b;
8  >> size(a)
9  ans =
10      110     1
```

Dimension mismatch error

Similarly, some arithmetic operations are only valid if the variables fulfill some size constraints. Consider the following commands (listing 2.5). The first one (line 3) fails because we are trying to do al elementwise add on two vectors that have different lengths, respectively sizes. The matrix multiplication in line 6 also fails since for this operations to succeed the inner matrix dimensions must agree (for more information on the matrixmultiplication see box **??** in chapter **??**). The elementwise multiplication issued in line 10 fails for the same reason as the addition we tried earlier. Sometimes, however, things apparently work but the result may be surprising. The last operation in listing 2.5 does not throw an error but the result is something else than the expected elementwise multiplication.

Listing 2.5: Some arithmetic operations make size constraints, violating them leads to dimension mismatch errors.

```
1  >> a = randn(100, 1);
2  >> b = randn(10, 1);
3  >> a + b
4  Matrix dimensions must agree.
5
6  >> a * b      % The matrix multiplication!
7  Error using  *
8  Inner matrix dimensions must agree.
9
10 >> a .* b
11 Matrix dimensions must agree.
12
13 >> c = a .* b';  % works but the result may not be what you expected!
14 >> size(c)
15 ans =
16      100    10
```

2.2 Logical error

Sometimes a program runs smoothly and terminates without any complaint. This, however, does not necessarily mean that the program is correct. We may have made a *logical error*. Logical errors are hard to find, MATLAB© has no chance to detect such errors since they do not violate the syntax or cause the throwing of an error. Thus, we are on our own to find and fix the bug. There are a few strategies that should we can employ to solve the task.

1. Be sceptical: especially when a program executes without any complaint on the first try.
2. Clean code: Structure your code that you can easily read it. Comment, but only where necessary. Correctly indent your code. Use descriptive variable and function names.
3. Keep it simple.
4. Test your code by writing *unit tests* that test every aspect of your program (2.3).

5. Use scripts and functions and call them from the command line. MATLAB© can then provide you with more information. It will then point to the line where the error happens.

6. If you still find yourself in trouble: Apply debugging strategies to find and fix bugs (2.4).

2.3 Avoiding errors

It would be great if we could just sit down, write a program, run it, and be done with the task. Most likely this will not happen. Rather, we will make mistakes and have to bebug the code. There are a few guidelines that help to reduce the number of errors.

Keep it small and simple

> *Debugging time increases as a square of the program's size.* Chris Wenham

> *Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?* Brian Kernighan

Break down your programming problems into small parts (functions) that do exactly one thing and are thus easily testable. This has already been discussed in the context of writing scripts and functions. In parts this is just a matter of feeling overwhelmed by 1000 lines of code. Further, with each task that you incorporate into the same script the probability of naming conflicts (same or similar names for variables) increases. Remembering the meaning of a certain variable that was defined in the beginning of the script is simply hard.

Many tasks within an analysis can be squashed into a single line of code. This saves some space in the file, reduces the effort of coming up with variable names and simply looks so much more competent than a collection of very simple lines. Consider the following listing (listing 2.6). Both parts of the listing solve the same problem but the second one breaks the task down to a sequence of easy-to-understand commands. Finding logical and also syntactic errors is much easier in the second case. The first version is perfectly fine but it requires a deep understanding of the applied functions and also the task at hand.

Listing 2.6: Converting a series of spike times into the firing rate as a function of time. Many tasks can be solved with a single line of code. But is this readable?

```
1 % the one-liner
2 rate = conv(full(sparse(1, round(spike_times/dt), 1, 1, length(time))),
     kernel, 'same');
3
4 % easier to read
5 rate = zeros(size(time));
6 spike_indices = round(spike_times/dt);
7 rate(spike_indices) = 1;
8 rate = conv(rate, kernel, 'same');
```

The preferred way depends on several considerations. (i) How deep is your personal understanding of the programming language? (ii) What about the programming skills of your target

audience or other people that may depend on your code? (iii) Is one solution faster or uses less resources than the other? (iv) How much do you have to invest into the development of the most elegant solution relative to its importance in the project? The decision is yours.

Unit tests

The idea of unit tests to write small programs that test *all* functions of a program by testing the program's results against expectations. The pure lore of test-driven development requires that the tests are written **before** the actual program is written. In parts the tests put the *functional specification*, the agreement between customer and programmer, into code. This helps to guarantee that the delivered program works as specified. In the scientific context, we tend to be a little bit more relaxed and write unit tests, where we think them helpful and often test only the obvious things. To write *complete* test suits that lead to full *test coverage* is a lot of work and is often considered a waste of time. The first claim is true, the second, however, may be doubted. Consider that you change a tiny bit of a standing program to adjust it to the current needs, how will you be able to tell that it is still valid for the previous purpose? Of course you could try it out and be satisfied, if it terminates without an error, but, remember, there may be logical errors hiding behind the facade of a working program.

Writing unit tests costs time, but provides the means to guarantee validity.

Unit testing in MATLAB©

Matlab offers a unit testing framework in which small scripts are written that test the features of the program. We will follow the example given in the MATLAB© help and assume that there is a function `rightTriangle` (listing 2.7).

Listing 2.7: Slightly more readable version of the example given in the MATLAB© help system. Note: The variable name for the angles have been capitalized in order to not override the matlab defined functions `alpha, beta,` and `gamma`.

```
1 function angles = rightTriangle(length_a, length_b)
2     ALPHA = atand(length_a / length_b);
3     BETA = atand(length_a / length_b);
4     hypotenuse = length_a / sind(ALPHA);
5     GAMMA = asind(hypotenuse * sind(ALPHA) / length_a);
6
7     angles = [ALPHA BETA GAMMA];
8 end
```

This function expects two input arguments that are the length of the sides a and b and assumes a right angle between them. From this information it calculates and returns the angles α, β, and γ.

Let's test this function: To do so, create a script in the current folder that follows the following rules.

1. The name of the script file must start or end with the word 'test', which is case-insensitive.
2. Each unit test should be placed in a separate section/cell of the script.
3. After the `%%` that defines the cell, a name for the particular unit test may be given.

Further there are a few things that are different in tests compared to normal scripts.

1. The code that appears before the first section is the in the so called *shared variables section* and the variables are available to all tests within this script.
2. In the *shared variables section*, one can define preconditions necessary for your tests. If these preconditions are not met, the remaining tests will not be run and the test will be considered failed and incomplete.
3. When a script is run as a test, all variables that need to be accessible in all test have to be defined in the *shared variables section*.
4. Variables defined in other workspaces are not accessible to the tests.

The test script for the `rightTrianlge` function (listing 2.7) may look like in listing 2.8.

Listing 2.8: Unit test for the `rightTriangle` function stored in an m-file testRightTriangle.m

```
tolerance = 1e-10;

% preconditions
angles = rightTriangle(7, 9);
assert(angles(3) == 90, 'Fundamental problem: rightTriangle is not
    producing a right triangle')

%% Test 1: sum of angles
angles = rightTriangle(7, 7);
assert((sum(angles) - 180) <= tolerance)

angles = rightTriangle(7, 7);
assert((sum(angles) - 180) <= tolerance)

angles = rightTriangle(2, 2 * sqrt(3));
assert((sum(angles) - 180) <= tolerance)

angles = rightTriangle(1, 150);
assert((sum(angles) - 180) <= tolerance)

%% Test: isosceles triangles
angles = rightTriangle(4, 4);
assert(abs(angles(1) - 45) <= tolerance)
assert(angles(1) == angles(2))

%% Test: 30-60-90 triangle
angles = rightTriangle(2, 2 * sqrt(3));
assert(abs(angles(1) - 30) <= tolerance)
assert(abs(angles(2) - 60) <= tolerance)
assert(abs(angles(3) - 90) <= tolerance)

%% Test: Small angle approx
angles = rightTriangle(1, 1500);
smallAngle = (pi / 180) * angles(1); % radians
approx = sin(smallAngle);
assert(abs(approx - smallAngle) <= tolerance, 'Problem with small angle
    approximation')
```

In a test script we can execute any code. The actual test whether or not the results match our predictions is done using the `assert()` assert function. This function basically expects a boolean value and if this is not true, it raises an error that, in the context of the test does not lead to a termination of the program. In the tests above, the argument to assert is always a boolean expression which evaluates to `true` or `false`. Before the first unit test ("Test 1: sum of angles", that starts in line 5, listing 2.8) a precondition is defined. The test assumes that the γ angle must always be 90° since we aim for a right triangle. If this is not true, the further tests, will not be executed. We further define a `tolerance` variable that is used when comparing double values (Why might the test on equality of double values be tricky?).

Listing 2.9: Run the test!

```
1 result = runtests('testRightTriangle')
```

During the run, MATLAB© will put out error messages onto the command line and a summary of the test results is then stored within the `result` variable. These can be displayed using the function `table(result)`

Listing 2.10: The test results.

```
1  table(result)
2  ans =
3    4x6 table
4
5              Name                       Passed  Failed  Incomplete  Duration   Details
6  ___________________________________    ______  ______  __________  ________  ___________
7
8  'testR.../Test_SumOfAngles'             true   false   false       0.011566   [1x1 struct]
9  'testR.../Test_IsoscelesTriangles'      true   false   false       0.004893   [1x1 struct]
10 'testR.../Test_30_60_90Triangle'        true   false   false       0.005057   [1x1 struct]
11 'testR.../Test_SmallAngleApprox'        true   false   false       0.0049     [1x1 struct]
```

So far so good, all tests pass and our function appears to do what it is supposed to do. But tests are only as good as the programmer who designed them. The attentive reader may have noticed that the tests only check a few conditions. But what if we passed something else than a numeric value as the length of the sides a and b? Or a negative number, or zero?

2.4 Debugging strategies

If you still find yourself in trouble you can apply a few strategies that help to solve the problem.

1. Lean back and take a breath.
2. Read the error messages and identify the line or command where the error happens. Unfortunately, the position that breaks is not always the line or command that really introduced the bug. In some instances the actual error hides a few lines above.
3. No idea what the error message is trying to say? Google it!
4. Read the program line by line and understand what each line is doing.
5. Use `disp` to print out relevant information on the command line and compare the output with your expectations. Do this step by step and start at the beginning.

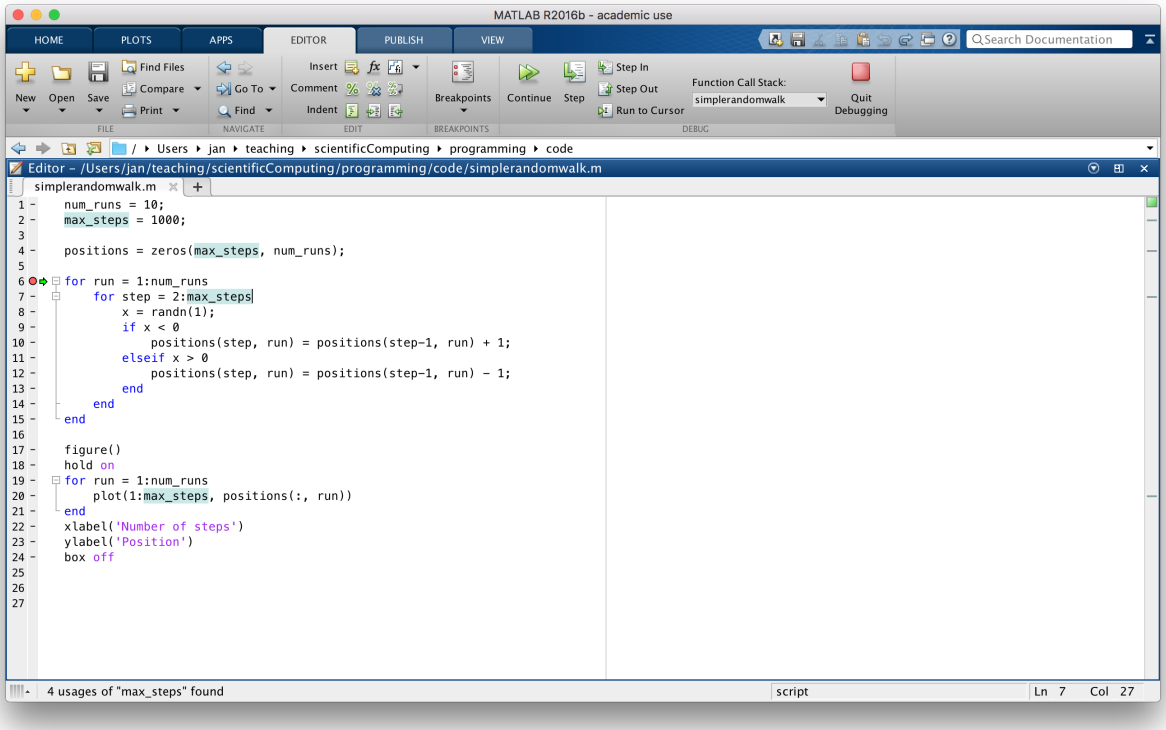


Figure 2.1: Screenshot of the MATLAB© m-file editor. Once a file is saved and passes the syntax check (the indicator in the top-right corner of the editor window turns green or orange), a breakpoint can be set. Breakpoints can be set either using the dropdown menu on top or by clicking the line number on the left margin. An active breakpoint is indicated by a red dot. The line at which the program execution was stopped is indicated by the green arrow.

6. Use the MATLAB© debugger to stop execution of the code at a specific line and proceed step by step. Be sceptical and test all steps for correctness.

7. Call for help and explain the program to someone else. When you do this, start at the beginning and walk through the program line by line. Often it is not necessary that the other person is a programmer or exactly understands what is going on. Often, it is the own reflection on the problem and the chosen approach that helps finding the bug. (This strategy is also known as *Rubber duck debugging*.

Debugger

The MATLAB© editor (figure 2.1) supports interactive debugging. Once you save an m-file in the editor and it passes the syntax check, i.e. the little box in the upper right corner of the editor window is green or orange, you can set one or several *break point*s. When the program is executed by calling it from the command line it will be stopped at the line with the breakpoint. In the editor this is indicated by a green arrow. The command line will change to indicate that we are now stopped in debug mode (listing 2.11).

Listing 2.11: Command line when the program execution was stopped in the debugger.

```
>> simplerandomwalk
6   for run = 1:num_runs
K>>
```

When stopped in the debugger we can view and change the state of the program at this point, we can also issue commands to try the next steps etc. Beware however, the state of a variable can be altered or even deleted which might affect the execution of the remaining code.

The toolbar of the editor offers now a new set of tools for debugging:

1. **Continue** — simply move on until the program terminates or the execution reaches the next breakpoint.
2. **Step** — Execute the next command and stop.
3. **Step in** — If the next command is a function call, step into it and stop at the first command.
4. **Step out** — If the next command is a function call, proceed until the called function returns, then stop.
5. **Run to cursor** — Execute all statements up to the current cursor position.
6. **Quit debugging** — Immediately stop the debugging session and stop the further code execution.

The debugger offers some more (advanced) features but the functionality offered by the basic tools is often enough to debug a program.