How To Test Your Code

A CS 1371 Homework Guide

Introduction

After you have completed each drill problem, you should make it a habit to test your code. There are good ways of testing your code and there are bad ways of testing your code. This guide is written to teach you the good ways and to help you avoid the bad ways.

What To Avoid

The first thing many students try is to copy and paste each test case from the hw##.m file into the Command Window to execute the function and to compare the printout in the Command Window with the solution in the he##.m file by inspection. Not only is this method slow and tedious, it is also prone to many errors, including the following:

1. Your function, as you have coded it, may simply be printing the values of the outputs to the Command Window instead of actually setting output variables. Please make sure your function is producing output variables that appear in the Workspace.

2. Your outputs may look correct when inspected by eye, but you cannot be sure that they are exactly identical to the correct outputs. This is especially true of output strings, where it is easy to miss a space or punctuation mark or even several English words entirely! For example, `'A C'` vs. `'A C '` or `'I like turtles.'` vs. `'I like turtles'` or `'This item is worth $9.'` vs. `'The item is worth $9.'`.

3. You may also fall victim to misinterpreting the typed-out solution on the hw##.m file. The typed-out solution (which usually looks like `out2 => [4, 5; 6, 7; 8, 9]` or like `str1 => 'Congratulations! You''re the big winner!'`) is meant to be interpreted as a block of code to be entered into the Command Window. For example, in the first case, a correctly-coded function would produce an output variable called `out2` that is a 3x2 array of type double. If you were to compare this variable to the result of entering `out2_soln = [4, 5; 6, 7; 8, 9];` into the Command Window, the two variables `out2` and `out2_soln` should be identical. In the second case, a correctly-coded function would output a variable called `str1` that is identical to the result of entering `str1_soln = 'Congratulations! You''re the big winner!';` into the Command Window. Note that the two single quotation marks are necessary to create a single quotation mark character inside of a string in MATLAB.

4. Your function will ultimately be tested by the Homework Autograder, which compares the outputs of your function with the outputs of the solution file (the file that looks like functionName_soln.p), given inputs from several different test cases. This means that, ideally, you should be testing your code against the solution file instead of the typed-out solutions on the hw##.m file, which are not guaranteed to be correct.

What To Do Instead

Write a *test script*!

Test Scripts

A test script is a *script* that you run in order to execute your functions with *test cases*, each of which is just an input or series of inputs with an associated expected output. A script is just a block of code that you can run without any inputs or outputs. The ABCs pretest file that comes with every homework assignment is a perfect example of a test script. You can't open it, because it's encrypted, but if you could look inside of it, you would see nothing more than a block of code that calls your ABCs function using one or two test cases and compares the outputs of your function to the expected outputs. It then prints out a series of statements to the Command Window which tell you which outputs were correct and which were incorrect. Our goal is to design a similar script to test the code that you write for the drill problems.

Writing A Test Script

Writing a test script is easy! The first thing you should start with is the built-in `clear` function. This function will delete all of the variables in your current Workspace so that you can test your code in a clean environment without the risk of mixing up variable names. Remember that you should never use `clear` in a *function*, only in a *script*. The next thing to do is to call your functions with a couple of test cases. For simplicity, we will only call one function in this simple example. Suppose your function is supposed to find the area and perimeter of a rectangle given two side lengths. The function should work as follows:

```
[area, perim] = rectangleMath(width, height)
```

Notice that the function is called rectangleMath and that it takes two inputs and produces two outputs. This means that we will have to *execute* or *call* this function in our test script using two inputs from a test case. Which test case should we use? Theoretically, you can use any test case you want. However, for your benefit, there are suggested test cases in the hw##.m file. They might look the following:

```
width1 = 2;
height1 = 5;
```

and

```
width2 = 0;
height2 = 1;
```

Each pair of inputs is a test case. Notice that the names of the inputs of the first test case end with the number 1 and the names of the inputs of the second test case end with the number 2. This makes sense and is good coding practice. After copying the code above into our test script, we are ready to use them to call our function. Remember that we need to assign two outputs for each function call. We will assign them as follows:

```
[area1, perim1] = rectangleMath(width1, height1);
[area2, perim2] = rectangleMath(width2, height2);
```

Again, notice the use of the numbers 1 and 2 at the end of the names of our outputs. Keep the naming of your variables consistent to avoid unnecessary confusion. At this point, you may think that we are done. We can run the script as it is and inspect the outputs, comparing them to what we know should be the correct outputs. Not so fast! Remember what we said about using the solution files. It may seem trivial in this simple example, but it is very important when testing more complicated code. In your homework folder, you should have been provided with encrypted solution files. In this example, the solution file will be a function called `rectangleMath_soln`. Let's call the solution function on *the same test cases* as follows:

```
[area1_soln, perim1_soln] = rectangleMath_soln(width1, height1);
[area2_soln, perim2_soln] = rectangleMath_soln(width2, height2);
```

Notice that we are storing our outputs in new variables, with _soln appended at the end of each. This is to distinguish them from the output variables of our own implementation of the function. If we had not done this, we would have *overwritten* the previous values of `area1`, `perim1`, `area2`, and `perim2`. *Now* we can compare the value of `area1` with the value of `area1_soln` and compare the value of `perim1` with the value of `perim1_soln` and so on for the second test case, right? We're done, right? Wrong! Remember that you should refrain from comparing your outputs to the solution outputs by inspection. Instead, we will use a more precise, foolproof method of comparison, the built-in `isequal` function. The `isequal` function takes two inputs and tells you if they are equal or not by outputting a logical `true` or `false`. Let's call `isequal` for each pair of output variables to see if our function behaves the same as the solution function. We will write the following lines of code:

```
  check_area1 = isequal(area1, area1_soln);
check_perim1 = isequal(perim1, perim1_soln);

  check_area2 = isequal(area2, area2_soln);
check_perim2 = isequal(perim2, perim2_soln);
```

Note that the naming of the logical output variables is consistent with what they represent. For example, if after running out test script, `check_area1` has the value `true`, then we know that the area computed by our function is exactly the same as the area computed by the solution file, at least for the first test case. If all of these check variables are `true`, then we know that our function is probably correct. This does not guarantee, however, that your code will receive full credit when you submit it. That will depend on which test cases the Autograder runs. That is why it is important to test your code with many different test cases and why we provide you with several suggested test cases. If any of the check variables are `false`, then there is something wrong with your function and you should go back to your code and try to fix the problem.

Congratulations! You just wrote a test script! Of course, on your actual homework, you will have more than one function to implement and test. The process of adding more functions to your test script is the same. Define some input variables from the suggested test cases. Call your function on those inputs. Call the solution function on those same inputs. Generate check variables by using `isequal` to compare the resulting outputs. Inspect the check variables in the Workspace to ensure that all of them are `true`. That's it! Just make sure that you do not reuse any variable names. This will cause unexpected and potentially frustrating results. Here is the complete test script:

```
% Test Script
clear

% Test Case 1
width1 = 2;
height1 = 5;

% Test Case 2
width2 = 0;
height2 = 1;

% My Function Outputs
[area1, perim1] = rectangleMath(width1, height1);
[area2, perim2] = rectangleMath(width2, height2);

% The Solution Function Outputs
[area1_soln, perim1_soln] = rectangleMath_soln(width1, height1);
[area2_soln, perim2_soln] = rectangleMath_soln(width2, height2);

% Check Test Case 1
check_area1 = isequal(area1, area1_soln);
check_perim1 = isequal(perim1, perim1_soln);

% Check Test Case 2
check_area2 = isequal(area2, area2_soln);
check_perim2 = isequal(perim2, perim2_soln);
```

Testing Outputs That Are Not Variables

The example test script above will work for all types of output variables, including doubles, logicals, chars (strings), cells, and structures. However, some weeks, the homework will require you to write functions that output things other than variables, such as text files, Excel files, plots, or image files. How can you test these? Let's break it down.

Testing Text Files

The first thing to check is to see that your text file is named correctly, exactly as specified in the problem statement. You may choose to do this by inspection. Now comes the hard part. How can you compare the contents of your output text file with the contents of the solution text file? It turns out that MATLAB has a built-in tool that will take care of this for you. Suppose your output file is called 'textFile1.txt' and the solution text file provided in the homework is called 'textFile1_soln.txt'. From the Command Window, type and run the following command:

```
visdiff('textFile1.txt','textFile1_soln.txt');
```

At this point, a new window should pop up. This is the MATLAB File Comparison Tool. It will not only tell you if the selected files match, but it will also tell you exactly what and where all of the differences are. Use this tool to your advantage. Please note that sometimes it will say, 'No differences to display. The files are not identical, but the only differences are in end-of-line characters.' Do not be alarmed if you see this; you will still receive full credit.

Testing Excel Files

There is no file comparison tool for Excel files. Instead, you can simply read the Excel files you would like to compare into MATLAB using `xlsread` and compare the raw cell array outputs using `isequal`. For example, to compare 'excelFile1.xlsx' and 'excelFile1_soln.xlsx', you could type the following in your test script:

```
    [~,~,raw1] = xlsread('excelFile1.xlsx');
[~,~,raw1_soln] = xlsread('excelFile1_soln.xlsx');
  check_excelFile1 = isequal(raw11,raw1_soln);
```

Testing Plots

Unfortunately, there is no easy way to test plots. By inspection, make sure that your plot has the same title and axis labels as the solution plot. Also make sure that you plotted all the correct points with the correct colors and styles. If your plot should contain more than one curve, it does not matter what order you plot the curves in. Therefore, the overlap of the lines or points on your plot will not affect the "correctness" of your plot.

Testing Image Files

Comparing two image files is very similar to comparing two Excel files. Instead of comparing the images outside of MATLAB, read the images into MATLAB using `imread` and compare the resulting image arrays as follows:

```
     im1 = imread('imageFile1.png');
im1_soln = imread('imageFile1_soln.png');
check_imageFile1 = isequal(im1,im1_soln);
```

Conclusion

Test your code the smart way! Don't do more work than you have to! Happy coding!