6

Debug and Optimize

Now that you are beginning to be fairly competent in MATLAB, we should talk about what to do when you're making your own code and things aren't going as planned. Errors can be frustrating to all of us, especially those without much previous experience in programming.

6.1 General Practices

In general, save often but also save backups of previous versions of your code (e.g., using revision numbers as done in this book or using last-edited dates). Having previous versions of your scripts and functions saved can go a long way in helping you solve an error, as you can "roll back" parts of your script and potentially easily correct your error and avoid more involved debugging issues.

6.2 Breaking Out of Unresponsive Code

For beginners, one type of error is even more problematic than an actual error in MATLAB being unresponsive because it is still "busy" when it really should have finished running the script or function. The usual cause of this is a bug in a `for` or `while` loop that prevents the loop from ever finishing. One way to prevent this from happening in the first place is to use `disp` to get status updates as your script runs, but this isn't a particularly helpful suggestion when your MATLAB process is already unresponsive.

To have MATLAB abort the current command without killing its process directly (e.g., "End Task," "Force Quit," or "kill"), press `CTRL+C`. You may have to press this several times if MATLAB was stuck in a nested loop. MATLAB will report an error on the line where it was when it aborted, but this line is usually not related to the actual issue. You can also have MATLAB automatically abort the current script or function if a certain condition is met by using the `break` function in conjunction with an `if` statement. Note that `break` can only abort the innermost loop (see `help` for more details).

6.3 Locating the Error

If you're working on code in the MATLAB command window, usually you can locate your errors fairly easily, even if you don't yet know what is wrong with the code. When working with scripts and functions, this isn't as easy. Though MATLAB will provide you with a line number for where your code is producing an error, sometimes that's not the best place to actually fix the code.

In the case of scripts, the best approach is to use `disp` to get status updates and see which portions of your code are being run and which are not. Another option is to use **`pause`**, which causes MATLAB to pause the script and wait for you to press a key before it will continue. **`echo`** is also useful to see what portions of the code are being run. When you get an error, you may also want to use `who` or `whos` to see which variables are in your workspace and what their values are.

With functions, debugging is a bit more difficult, as functions have their own internal workspace that you can't access if the code returns an error or if you use `CTRL+C` or `break` to abort the code. In this situation, the best function you can use is **`keyboard`**. This function is great in that it pauses the function where it stands and lets you take over with the *keyboard*. As a result, you can now check the contents of variables that are internal to the function and work to ascertain what is wrong with your code. After you are done, type **`return`** to have MATLAB continue running the function from where it paused. You can also alternatively type **`dbquit`** to exit this debug mode and completely abort the current function or code.

6.4 Common Errors: Typing Related

Now that we are better able to localize our problematic lines of code, we need to learn what errors that MATLAB reports really mean and what is the likely cause. Here, we will go through the most common MATLAB errors that you are likely to encounter, along with a simple example that can create them and where to look to remedy them.

Expression or statement is incorrect—possibly unbalanced (, {, or [

This error simply means that the number of brackets you open (of any type, (, [, or {) does not match the number that you close. MATLAB usually tries to point out where in the line the error is, but it may not be correct. This is a particularly common error and arises when you combine many functions within one line of code.

```
>> disp(sprintf('Today is %s.',date))
Today is 06-Jan-2013.
>> disp(sprintf(('Today is %s.',date))
??? disp(sprintf(('Today is %s.',date))
                 |
Error: Expression or statement is incorrect--possibly unbalanced (, {,
or [.
```

The simplest solution for this is to take apart the line of code and work back from the inside out and see how much of it works, and confirm that the values returned actually make sense. Below is an example of how this can look.

Too many input arguments

This error message means that you are providing more input variables into a function than it is designed to handle. However, it is more likely that you either made a mistake with the brackets, where you have all the brackets you need (i.e., not an imbalance), but that one or more of them is in the wrong position. It is also possible that you may have accidentally skipped over a step in the logic and missed using one of your functions. Here are examples of either likely mistake.

```
>> disp(sprintf('Today is %s.',date))
Today is 06-Jan-2013.
>> disp(sprintf('Today is %s.'),date)
??? Error using ==> disp
Too many input arguments.
>> disp('Today is %s.',date)
??? Error using ==> disp
Too many input arguments.
```

Undefined function or variable . . .

This error is fairly straightforward: You made a typo, such that the function or variable you are referring to doesn't exist. Since the errors tell you what doesn't exist, it should be easy to search your code for the typo.

```
>> disp(sprintf('Today is %s.',date))
Today is 06-Jan-2013.
>> disp(sprintf('Today is %s.',dat))
??? Undefined function or variable 'dat'.
```

Undefined function or method . . . for input arguments of type . . .

This is almost the same as the previous error, but it occurs when your function has inputs specified.

```
>> disp(sprintf('Today is %s.',date))
Today is 06-Jan-2013.
>> dis(sprintf('Today is %s.',date))
??? Undefined function or method 'dis' for input arguments
of type 'char'.
>> disp(sprint('Today is %s.',date))
??? Undefined function or method 'sprint' for input arguments
of type 'char'.
```

Inner matrix dimensions must agree

This error is usually caused by accidentally telling MATLAB to do matrix multiplication (*) rather than element-wise multiplication (.*). (Also see p. 12.)

```
>> ones(2,4) * ones(2,4)
??? Error using ==> mtimes
Inner matrix dimensions must agree.
>> ones(2,4) .* ones(2,4)
ans =
     1     1     1     1
     1     1     1     1
```

6.5 Common Errors: Value Related

The previous common errors were all based on some sort of typing error where MATLAB was confused by the bracket placement or was not sure what variable you were referring to. The next set of common errors comprises those related to the contents of your variables and how you are attempting to interact with them.

Index exceeds matrix dimensions

This error occurs when you try to refer to an index that does not exist in the matrix. Specifically, the index you provided was larger than that of the matrix.

In the case of this error, you are likely to have accidentally swapped your indexes for the row and column dimensions. Here, we'll make a quick example matrix, so we can try and reproduce and isolate these errors.

```
>> M = rand(8,4)
M =
      0.1656    0.2290    0.1067    0.2599
      0.6020    0.9133    0.9619    0.8001
      0.2630    0.1524    0.0046    0.4314
      0.6541    0.8258    0.7749    0.9106
      0.6892    0.5383    0.8173    0.1818
      0.7482    0.9961    0.8687    0.2638
      0.4505    0.0782    0.0844    0.1455
      0.0838    0.4427    0.3998    0.1361
>> M(3,6)
??? Index exceeds matrix dimensions.
>> M(6,3)
ans =
      0.8687
```

Attempted to access . . . index out of bounds because size . . .

This error is very similar to the previous one, but here we are referring to a whole row or column of the matrix rather than a single index.

A likely cause of this error is using `length` when your dimension of interest is *not* the longest. Here you should use `size`. Note that `length` is effectively the same as `max(size)`.

```
for i = 1:length(M)
    meanM(i) = mean(M(:,i));
end
meanM
```

Using length produces the following error:

```
??? Attempted to access M(:,5); index out of bounds because
size(M)=[8,4].
```

Using size, we can see that we should be using the second dimension.

```
>> size(M)
ans =
     8     4
```

The corrected code would look as follows.

```
for i = 1:size(M,2)
    meanM(i) = mean(M(:,i));
end
meanM
```

As noted earlier, using `size(M,1)` in this particular case would be effectively the same as using `length`, as `length(M)==max(size(M))`.

Subscript indices must either be real positive integers or logicals

This error occurs when you attempt to access an index of a variable that is simply not possible, such as a zero or a non-integer (i.e., a number that has decimals).

```
>> M(1)
ans =
    0.1656
>> M(0)
??? Subscript indices must either be real positive integers
or logicals.
>> M(3.4)
??? Subscript indices must either be real positive integers
or logicals.
```

In all likelihood, you did not type this yourself and instead fed one variable in as the index for the other. You probably forgot to include the find function or maybe round.

Subscripted assignment dimension mismatch

This error occurs when you copy values from one matrix to another, but the variables aren't of the same length. The same error is produced regardless of which is longer.

```
>> N = nan(4,2);
>> N(1,:)
ans =
   NaN NaN
>> M(1,:)
ans =
    0.1656    0.2290    0.1067    0.2599
>> N(1,:) = M(1,:)
??? Subscripted assignment dimension mismatch.
>> M(1,:) = N(1,:)
??? Subscripted assignment dimension mismatch.
>> length(N(1,:))
ans =
    2
>> length(M(1,:))
ans =
    4
```

This one is a bit harder to fix, as it really depends on what you meant to do. Nonetheless, one possible solution is to only copy the number of values that would fit. Depending on which variable is the longer one, you may need to change the left or the right side of the equal sign. In either case, the key point is to accommodate the shorter of the two variables.

```
>> M(1,1:length(N(1,:))) = N(1,:)
M =
          NaN        NaN    0.1067      0.2599
    0.6020     0.9133    0.9619      0.8001
    0.2630     0.1524    0.0046      0.4314
    0.6541     0.8258    0.7749      0.9106
    0.6892     0.5383    0.8173      0.1818
    0.7482     0.9961    0.8687      0.2638
    0.4505     0.0782    0.0844      0.1455
    0.0838     0.4427    0.3998      0.1361
```

```
>> N(1,:) = M(1,1:length(N(1,:)))
N =
     0.1656    0.2290
        NaN       NaN
        NaN       NaN
        NaN       NaN
```

In an assignment A(I) = B, the number of elements in B and I must be the same

This error is quite similar to the last, but this one occurs when you try and store multiple values from one variable into a single index of another variable. This can usually be solved by adjusting your code to store the multiple values of one variable into the same number of values in the second variable, but you should be particularly careful here to make sure that this is what you intended.

```
>> A = ones(1,4)
A =
      1      1      1      1
>> A(1) = M(1,:)
???  In an assignment   A(I) = B, the number of elements in B
and I must be the same.
>> A(1,:) = M(1,:)
A =
     0.1656     0.2290     0.1067     0.2599
```

CAT arguments dimensions are not consistent

This error is produced when you try and concatenate two variables that are of different lengths. This error occurs regardless of whether you are using `[ ]`, `cat`, `horzcat`, or `vertcat`.

```
>> M(1,:)
ans =
     0.1656     0.2290     0.1067     0.2599
>> N(1,:)
ans =
   NaN    NaN
>> [ M(1,:) N(1,:) ]
ans =
     0.1656     0.2290     0.1067     0.2599     NaN     NaN
>> [ M(1,:); N(1,:) ]
??? Error using ==> vertcat
CAT arguments dimensions are not consistent.
```

For this error, it is particularly difficult to suggest how you fix it, as the uses of concatenation can vary greatly. Whatever you do choose, make sure you manually confirm that the resulting output is what you were expecting.

That's all for the common errors. It is quite possible that you will come across other errors as well, but hopefully you will now find the MATLAB error messages a bit less cryptic and you now have a better idea how to resolve them. Good luck!

6.6 Timing Your Code With Tic–Toc

Moving on from debugging, let's work on optimizing your code. In other words, for the sections of Chapter 6 from this point forward, we are assuming that your code works, and you are interested in making it more efficient.

Your first functions toward optimizing your code are `tic` and `toc`. When you use `tic`, MATLAB starts a timer (i.e., a stopwatch). When you use `toc`, MATLAB checks the time elapsed since the last `tic` and outputs it in the command window.

```
>> tic
>> toc
Elapsed time is 2.185080 seconds.
```

You can also use `toc` multiple times to get the time since the most recent `tic`. Using `tic` again will reset to start from zero again.

```
>> tic
>> toc
Elapsed time is 2.185080 seconds.
>> toc
Elapsed time is 7.660244 seconds.
>> tic
>> toc
Elapsed time is 2.335965 seconds.
>> toc
Elapsed time is 20.789621 seconds.
>> toc
Elapsed time is 26.334178 seconds.
```

You can also store the time from toc in a variable.

```
>> tic
>> now = toc
now =
    9.1857
```

You might be wondering: "What's so special about tic and toc? Why would I care about the time when I'm writing my code?" The answer is to try and optimize your code so that it runs faster. Let's try this out. If you go in the demo folder, you will find a script called timer1.m. The contents of this script are copied below. Run this script and see how long it takes to run.

```
tic

numbers = 1:1000;
nSum = 0;
for i = numbers
    nSum = nSum + numbers(i);
end

elapsed1 = toc
```

This code is intended to add up all of the numbers from 1 to 1,000. It is intentionally written a bit badly, but hopefully you can still tell that's what it does. Let's run it and see what elapsed1 is.

```
>> timer1
elapsed1 =
    0.0034
```

Now, let's compare this to a much more efficient version of this, which has been saved as timer2.

```
tic

numbers = 1:1000;
nSum = sum(numbers);

elapsed2 = toc
```

```
>> timer2
elapsed2 =
   3.0986e-05
```

Clearly `elapsed2` is a smaller value. Now, you might be thinking that these aren't very big numbers; does 0.0034 seconds really matter? Well, these are intended to be extremely simple cases, much simpler than your own analyses. It can be very helpful to measure the time it takes for your code to run, especially if you end up doing mathematical simulations, where the same block of code is run tens of thousands of times. Even still, we can also readily see that the small change between `timer1.m` and `timer2.m` made the code run just over 100 times faster.

```
>> elapsed1/elapsed2
ans =
  108.2763
```

6.7 Semicolons Are Your Friend

Toward the aim of optimizing your code, one of the smallest changes you can make that will make a world of difference is to add semicolons to the ends of lines that output text to the MATLAB command window. This is especially important if the line of code is contained within a `for` loop. Let's remove a semicolon from `timer1.m`, save the file as `timer3.m`, and see how much difference it makes.

```
>> timer3
nSum =
     1
nSum =
     3
nSum =
     6
nSum =
    10
    ...
elapsed3 =
    0.0213
```

That was a lot slower, but how much?

```
>> elapsed3/elapsed1
ans =
    6.3477
>> elapsed3/elapsed2
ans =
  687.3026
```

Removing that one character/key press made our script take more than 6 times as long and nearly 700 times as long as our "optimized" version. Clearly, printing to the MATLAB command window can slow scripts down markedly. (Remember that you can use `CTRL+C` to abort code that takes too long to run!) Suppressing the output can easily speed up your code. This suggestion works well as long as you don't particularly need to be informed of the output of that line of code. If you *do* want to know its contents, such as a counter in a loop that iterates through participant data or some other counter, you can use `if` in conjunction with `mod` to get periodic updates on the counter.

`mod` is a particularly interesting function, but the breadth of its usefulness will probably surprise you. Back in grade school, we learned how to do long division and find the remainder. In analyses, this operation can be quite useful and is more formally known as "modulo." In many programming languages, 'modulo' is abbreviated to 'mod' and uses `%` as the operator symbol (e.g., 23 % 4 = 3). However, as you know, `%` is used for other purposes in MATLAB.

One good example of the uses of `mod` is to determine if a number is odd or even, by dividing it by 2 and seeing what the remainder is.

```
>> mod(23,4)
ans =
     3
>> mod(23,2)
ans =
     1
>> mod(22,2)
ans =
     0
```

Similarly, we can also easily extract the "ones" digit of a number by looking at the remainder after dividing a number by 10.

```
>> mod(12,10)
ans =
     2
>> mod(113,10)
ans =
     3
>> mod(19208,10)
ans =
     8
```

Additionally, mod is quite useful in conjunction with floor.

```
>> value = 23;
>> divisor = 4;
>> mod_value = mod(value,divisor)
mod_value =
     3
>> floor_value = floor(value/divisor)
floor_value =
     5
>> divisor*floor_value+mod_value
ans =
    23
```

Returning to our current situation with optimizing code, we can use mod in an if statement to only print the contents of a variable periodically, such as every 100th cycle of our for loop, as shown below.

```
if mod(i,100) == 0
    i
end
```

We have copied timer1.m, added in this code, and saved it as timer4.m.

```
tic

numbers = 1:1000;
nSum = 0;
for i = numbers
    nSum = nSum + numbers(i);
    if mod(i,100) == 0
        i
    end
end

elapsed4 = toc
```

Let's run it and see how it compares.

```
>> timer4
i =
    100
i =
    200
i =
    300
i =
    400
i =
    500
i =
    600
i =
    700
i =
    800
i =
    900
i =
         1000
elapsed4 =
     0.0132
```

```
>> elapsed4/elapsed1
ans =
     3.9286
>> elapsed4/elapsed3
ans =
     0.6189
```

This version is definitely slower than `timer1.m` but not as bad as `timer3.m`. Keep in mind that there is also a cost to adding this additional code for the `if` and `mod`, as MATLAB has to do more calculations on every cycle of the `for` loop. To make this more apparent, let's set the `mod` function to divide by 1, effectively making the code in the `if` statement always run. This file is saved as `timer5.m`.

```
>> timer5
i =
      1
i =
      2
i =
      3
i =
      4
i =
      5
      ...
elapsed5 =
    0.0483
```

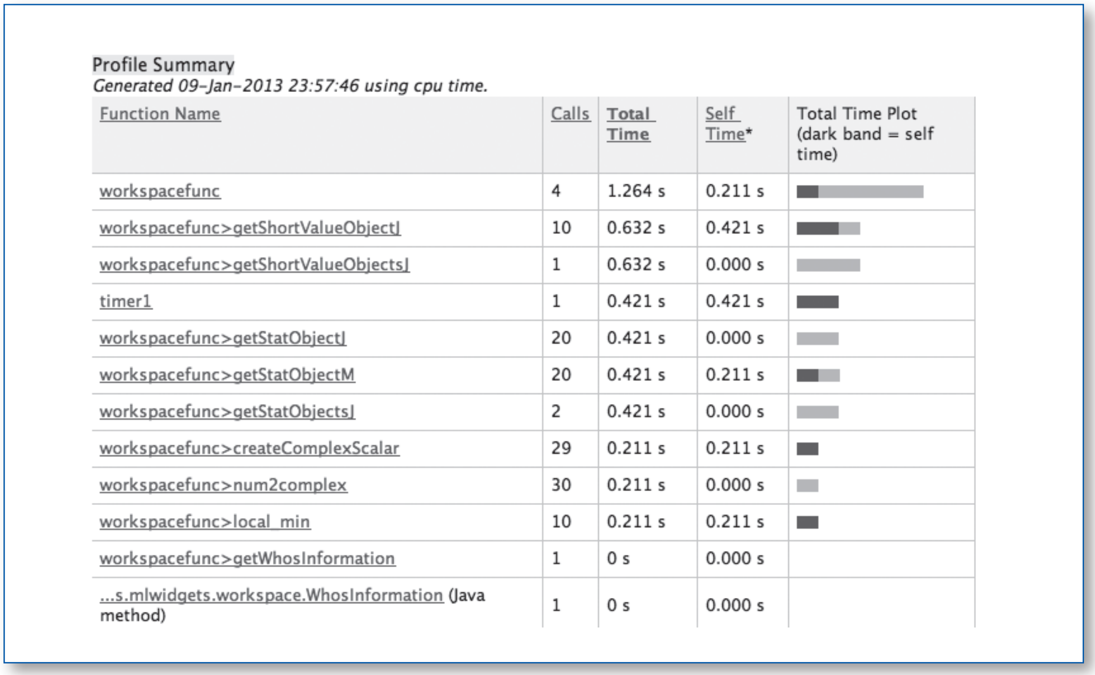
```
>> elapsed5/elapsed1
ans =
   14.3882
>> elapsed5/elapsed4
ans =
    3.6624
>> elapsed5/elapsed3
ans =
    2.2667
```

Clearly, this made the script run much slower. That about covers it for `tic` and `toc`, but hopefully, that gave you some idea that even though there are many ways to code an analysis in MATLAB, they are not all equally optimal. That being said, it may sometimes also be better in the long run to make your code more flexible so that it can be reused in a different analysis, despite making the code a bit slower to run.

6.8 Profiling Your Code

If `tic` and `toc` aren't enough for you, MATLAB does have more powerful tools to help you optimize your code. Just like the FBI profilers you see on TV profile unsavory people to better understand what makes them tick, you can use the `profile` function to profile your MATLAB code to better understand what makes your code work. This is the function that you need to clean up complex, resource-hungry MATLAB code. You can find the bottlenecks by profiling your code. A screenshot of the `profile` viewer is shown in Figure 6.1.

Figure 6.1. Screenshot of the `profile` viewer.



Profile Summary
Generated 09-Jan-2013 23:57:46 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
workspacefunc	4	1.264 s	0.211 s	
workspacefunc>getShortValueObjectJ	10	0.632 s	0.421 s	
workspacefunc>getShortValueObjectsJ	1	0.632 s	0.000 s	
timer1	1	0.421 s	0.421 s	
workspacefunc>getStatObjectJ	20	0.421 s	0.000 s	
workspacefunc>getStatObjectM	20	0.421 s	0.211 s	
workspacefunc>getStatObjectsJ	2	0.421 s	0.000 s	
workspacefunc>createComplexScalar	29	0.211 s	0.211 s	
workspacefunc>num2complex	30	0.211 s	0.000 s	
workspacefunc>local_min	10	0.211 s	0.211 s	
workspacefunc>getWhosInformation	1	0 s	0.000 s	
...s.mlwidgets.workspace.WhosInformation (Java method)	1	0 s	0.000 s	

When you are ready to start profiling, just type `profile on`. When you're all done and ready to see the report, type `profile viewer`.

```
>> profile on
>> % run your code/script here
>> profile viewer
```

6.9 A Fresh Pair of Eyes

As a final note, when you're not sure how to solve an error or could use some direction in optimizing your code, don't be afraid to ask a friend to take a look. Hopefully, you know someone else working through this book or who already knows MATLAB. A fresh pair of eyes looking through your code can make all the difference. You sometimes need to ask someone else to proofread a paper you write; your code is no different.

EXERCISES

This time will be a bit different; here we will fix errors and optimize code.

1. Load the `worddb` data set.

   Find the error produced by each line of code, and try and correct the mistake.

2. Code:

```
scatter(worddata{10},worddata{8}(1:460))
```

3. Code:

```
imagTab = mean(worddata{20}(find(strcmp,worddata{2},'taboo')))
```

4. Code:

```
types=unique(worddata(2,1:460))
```

5. Try to optimize this script:

```
% find numbers divisible by 3 within certain range
numbers = 277:300;
div3 = [];

for n = numbers
    if (n/3) == round(n/3)
        div3 = [ div3 n ];
    end
end

div3
```

   Output:

```
div3 =
     279    282    285    288    291    294    297    300
```

See page 209 for the solutions. Next, we will add basic statistics to our MATLAB skills.

FUNCTION REVIEW

General: `mod`

Debug: `break pause keyboard return dbquit`

Timing: `tic toc profile`