Chapter 04 - Debugging and Improving Performance.

Author: Ken Deeley, ken.deeley@mathworks.co.uk

This chapter provides techniques for maintenance, troubleshooting and debugging of MATLAB applications. We will use a variety of approaches to diagnose problems, identify common errors and evaluate code performance. We will also look at strategies for writing code with performance in mind, including vectorisation techniques and managing memory effectively in MATLAB. The idea is that we have previously developed and tested algorithms, which may now be fine-tuned to improve performance. This can be done safely in the presence of existing unit tests. A common situation for many people is to inherit code from others which may require debugging and maintenance. In this chapter there are two examples. The first example will assume that we have inherited a "black box" algorithm which we are expected to use. However, the code does not currently work, so it is necessary to debug it first. The second example will focus on vectorisation and memory management strategies to improve code performance. In this chapter we will use integrated MATLAB development tools to diagnose errors and identify potential for performance improvement. We will also write vectorised MATLAB code.

Outline:

- Tools for diagnosing errors
- Directory reports
- Breakpoints
- Tools for measuring performance
- Timing functions
- The MATLAB Profiler
- Improving performance
- Vectorisation strategies
- Vectorising operations on cells and structures
- Memory preallocation
- Efficient memory management

Reference files for this chapter:

- ../findBestPredictors.mlx
- ../S04_Vectorisation.mlx
- ../Reference/F04_*.m
- ../Reference/S04_Compact.m
- ../Reference/S04_CellData.mat
- ../Reference/S04_StructData.mat

Course Example: Best BMI Predictors.

Let's assume that we have inherited code from another person. Possibly this person is another researcher in the same department, or a former work colleague, or a research supervisor. In any case, we have been left with some code that we would like to use (possibly adapt or modify for our specific needs). Unfortunately, the code does not work as it stands, so we will need to go through a debugging process to repair it.

We have a function, findBestPredictors, which is intended to identify the two best predictor variables from the medical data (excluding BMI and actual and reported height and weight) for the purpose of classifying an individual's BMI category. When working correctly, findBestPredictors should return a table of results (one observation for each pair of predictor variables) and render a visualisation of the classified data for the best pair of predictor variables.

```
% View the non-functional version of findBestPredictors.
edit findBestPredictors
```

Look at the documentation for try,catch to see examples of using try/catch blocks to catch exceptions.

```
% Attempt to invoke the function.
try
    predictorRankings = findBestPredictors();
catch MExc
    disp(MExc.message)
end
```

This produces an error message, so the code does not work as intended. We need to start debugging. This chapter is intended to provide a typical workflow a programmer may follow when debugging code.

Directory Reports.

Directory reports may be used as a first step to obtain an overview of particular aspects of files in a given directory (folder). Directory reports are available in the MATLAB section of the Apps tab. Note that these reports are specific to a given directory.

As a first step, we can produce a Code Analyzer Report listing all Code Analyzer warnings for each file in the current folder. This will help us to fix syntactical and other errors present in the current code.

Run a Code Analyzer Report to generate a summary of all Code Analyzer warnings and errors.

Analyzing Code in the Editor.

The MATLAB Editor has various integrated tools intended for code debugging. The status box in the upper right-hand corner provides an indication of the current status of the code. Green means that there are no detectable errors in the code; orange means that there is potential for unexpected results or poor performance, and red means that there are errors which currently will prevent the code from running.

A summary of the code issues can be obtained from the Code Analyzer Report as described above. This analysis can also be done programmatically using the CHECKCODE function:

```
checkcode('findBestPredictors')
```

Entering Debug Mode.

Now that we have fixed all of the existing Code Analyzer warnings, we might hope that the code now runs as expected.

```
try
    predictorRankings = F04_findBestPredictors_V1(); %#ok<*NASGU>
catch MExc
    disp(MExc.identifier)
end
```

It doesn't, which is not that surprising. The Code Analyzer only performs a static check of the code; it is unable to detect run-time errors. Many run-time errors are most easily debugged by viewing the workspace of the function during execution. In MATLAB, this can be achieved by entering debug mode. If we want to enter debug mode automatically in the presence of an error, use the following command:

```
dbstop if error

% This can be cleared using:
dbclear if error
```

Breakpoints may also be inserted manually in the MATLAB Editor by clicking on the line number beside each line of code. Conditional breakpoints may be set by modifying the breakpoint using the right-click menu. This enables the programmer to enter debug mode only when a certain condition is satisfied.

We will diagnose the run-time problems and look at how can we repair them and prevent similar errors from occurring in the future. (The EXIST and ISFIELD functions may be useful here.)

See the file F04_findBestPredictors_V2 for a version of the code which fixes the first two run-time errors involving LOAD and accessing structure fields.

Resolving Dependencies.

Many applications, especially larger applications, consist of multiple functions and code files. These functions may call each other, as well as other MATLAB or toolbox functions. Resolving all dependencies can be challenging. If we try to call our function at this stage, we are faced with a dependency problem:

```
try
    predictorRankings = F04_findBestPredictors_V2();
catch MExc
    disp(MExc.identifier)
end
```

```

% Note that the WHICH command can help us determine which function or
% variable is being referenced in a given command:
which F04_findBestPredictors_V2
which getpairs in F04_findBestPredictors_V2
which getPairs in F04_findBestPredictors_V2
```

Running a Dependency Report on the directory may also help to diagnose and resolve the "undefined function" error we obtained above.

Try running a Dependency Report on the directory to understand the dependency problem. Fix the problem.

Preallocation of Memory.

We now have a program which runs but the remaining warnings indicate that it may not be efficient.

MATLAB allows variables to be resized dynamically, for example, during the individual iterations of a loop. This is convenient, but in some cases can lead to poor performance. Preallocating memory in advance is the recommended approach. For numeric arrays, preallocation can be done using the zeros/ones/NaN function. Note that it is important to preallocate for the correct data type. All numeric data types are supported by zeros and ones; NaN may initialise double or single values. Cells and structures may be preallocated using the cell, struct and repmat functions.

Debugging Completion.

After fixing the dependency problem, we now have a function that works.

```
predictorRankings = findBestPredictors();
```

Diagnosing Performance.

This is good news - we can get started using this function for our own needs. However, as well as debugging code to remove errors, it is also good practice to investigate performance aspects of the code. Just because code works and gives the correct results, it does not mean that it cannot be improved by refactoring.

Options for evaluating code performance include the following techniques.

- The tic/toc stopwatch timing functions provide the total system time elapsed between commands. Use this technique to time small blocks of code (not complete functions).
- The timeit function provides an accurate estimate of the time required to run a function. This works by repeatedly calling the function and estimating an average run time based on known behaviour of MATLAB function invocation.
- Using the MATLAB Profiler (see the next section).

The MATLAB Profiler.

The MATLAB Profiler is intended to provide a more in-depth breakdown of the time spent executing code. It produces a fully hyperlinked HTML report which details the number of calls, the total time and the self time for each function. Using the profiler results can help to identify bottlenecks in code performance.

There is a video describing its use here: Profiler video

Open the profiler and profile the working version of findBestPredictors. Either use the code below or run the Profiler from the Apps tab.

```
profile on
predictorRankings = findBestPredictors();
profile off
```

```
profile viewer
```

Identify potential code bottlenecks using the profile results. Reorganise the update of the uitable data in the local visualisation function, by removing the get/set commands and performing the data update only once, at the end of the loop.

See the file F04_findBestPredictors_Final for reference.

At this point, we have now completed a sample workflow for debugging and performance-tuning a MATLAB application. In the remainder of the chapter we will discuss additional techniques for improving performance of code, focussing on vectorisation and memory management.

Vectorisation.

MATLAB is an array-based language, and as such all vector and matrix operations are optimised for performance. Replacing sequences of scalar operations performed in loops with smaller numbers of vector and matrix operations is referred to as vectorisation. This process can lead to more readable and efficient code.

Again, let's assume that we have been handed some code written by someone else. We are now interested in improving its performance and readability, given that we have been through the debugging and profiling stages to remove errors and more obvious bottlenecks.

```
edit S04_Vectorisation
```

We will vectorise the code in the first section of calculations by using standard mathematical operations. Next, we will use standard table functionality (VARFUN) to vectorise common mathematical and statistical operations on tabular data.

See the file S04_Compact for reference here.

Vectorising Operations on Cells and Structures.

There are situations when working with cell and structure arrays when it becomes necessary to apply a function to the contents of each cell or structure field. In this situation, it is possible to vectorise the operations using CELLFUN or STRUCTFUN.

These functions allow a function to be applied to an entire cell or structure.

For example, the following commands will apply the `length` function to every element of the cell array C:

```
Clengths = cellfun(@length, C);
```

Note that the function to apply to the array is specified using a function handle.

By default, `cellfun,` assumes that the output of the function (applied to a single cell element) is a scalar. If so, `cellfun`, returns a vector of these individual values. If not, use the `UniformOutput` option to return the results as a cell array:

```
sz = cellfun(@size, C, 'UniformOutput', false)
```

We will vectorise the code in the next section of the S04_Vectorisation medical data calculation script using CELLFUN. The final section can be vectorised in a similar way using STRUCTFUN.

See the file S04_Compact for reference here.

Memory Anatomy.

On a Windows system, the MEMORY command can be used to obtain available memory information. Calling the MEMORY function with two outputs produces two structures which contain user and system information, respectively.

```
[user, sys] = memory;
disp(user)
disp(sys)
```

Copy-on-Write Behaviour.

When assigning one variable to another, MATLAB does not create a copy of that variable until it is necessary. Instead, it creates a reference. A copy is made only when code modifies one or more values in the reference variable. This behaviour is known as copy-on-write behaviour and is designed to avoid making copies of large arrays unless it is necessary. For example:

```
clear
A = rand(6e3);
m = memory;
disp('Memory available (GB):')
disp(m.MemAvailableAllArrays/1073741824)
B = A;
% This is only a reference at the moment, so available memory should
% remain constant.
m = memory;
disp('Memory available (GB):')
disp(m.MemAvailableAllArrays/1073741824)
B(1, 1) = 0;
% Now that we have made a change, copy-on-write should kick-in.
m = memory;
disp('Memory available (GB):')
disp(m.MemAvailableAllArrays/1073741824)
```

In-Place Optimisation.

An in-place optimisation conserves memory by using the same variable for an output argument as for an input argument. This is valid only when the input and output have the same size and type, and only within functions. When working on large data and memory is a concern, in-place optimisations could be used to attempt to conserve memory use. However, in some situations, it is not possible to have a true in-place optimisation, because some temporary storage is necessary to perform the operation.

See the reference file "inPlaceExample" and call the function `inPlaceExample()` from the command window.