Chapter 01 - The MATLAB Language and Desktop Environment.

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

This chapter provides the introductory MATLAB material necessary to work effectively in the MATLAB environment. It serves as a mini "crash-course" covering the fundamental programming techniques and syntax which are required for the remainder of the bootcamp. It's not recommended to enter the bootcamp with no previous MATLAB knowledge, although a very intelligent person should be able to follow the gist of this chapter without getting too much into the low-level programming details. As pre-work for the bootcamp, we strongly recommend the interactive online tutorial available free of charge for academic users at: https://www.mathworks.co.uk/academia/student_center/tutorials/

This contains more than three hours of introductory MATLAB materials and provides suitable prerequisite training prior to attendance at a MATLAB-based bootcamp.

Note that this chapter can be customised depending on the background and existing knowledge level of the audience. To be consistent with Software Carpentry guidelines, programming fundamentals should be covered. Therefore, a minimal core set of materials for this chapter should cover vectors and matrices, indexing, programming constructs such as loops, logical variables and conditional logical statements. Depending on majority interests, an overview of data types would be useful. Standard numeric arrays should be covered. Cell arrays, structures and tables are probably going to be useful for a lot of delegates as well, but how much emphasis is given to each data type can be left up to the instructor.

Outline:

- The MATLAB Desktop
- Importing data from one file
- Importing data from multiple files
- Indexing into vectors and matrices to retrieve data
- Concatenating vectors and matrices to construct data
- Removing missing values from data
- Basic plot options
- Annotating plots
- Cell and structure arrays
- Saving data to MAT-files
- Running and publishing scripts
- Code sections

Reference files for this chapter:

- ../MedicalData.txt
- ../HeightWaistData.txt
- ../ArmsLegs/*.txt
- ../Reference/S01_Import.m
- ../Reference/S01_HealthData.mat

Contents

Course example: Biomedical data.

We will work with an anonymised biomedical data set from the 2007-2008 US National Health and Nutrition Examination Survey (NHANES). This data is freely available at: http://www.cdc.gov/nchs/nhanes/nhanes_questionnaires.htm

The idea behind choosing this dataset is that delegates from all cultural and language backgrounds should be able to relate to this data. Measurements such as arm circumference, height, weight, age are applicable to everyone.

The MATLAB Desktop.

It's worth introducing the four main components of the MATLAB Desktop (the screen that appears when MATLAB is started). The four main components are:

- The Current Folder Browser (CFB)
- The Command Window
- The Command History
- The Workspace Browser

Recommendations: Show people how to change folders, how to identify which folder they are working in, and how to add/remove directories to/from their MATLAB path. Understanding how to manipulate the path to ensure that code files are visible is a key point. Show some basic commands at the command window and emphasise that they are recorded in the command history. Show how to recall commands from the command history. Explain that the Workspace Browser stores your current MATLAB data (variables).

The MATLAB Editor.

The MATLAB Editor is used to write, edit, run, debug and publish MATLAB code. We will collect our code into a script in this initial chapter of the course. Open the Editor, and create a new script. You might find it easier to work with the Editor window docked into the main MATLAB Desktop, so that it's easy to transfer code from the Command Window and History into the Editor. Modify your Editor preferences via the main MATLAB Preferences dialog:

```
preferences
```

Code Sections.

Best practice is to write code in sections when developing scripts. This is strongly recommended, for ease of debugging and publishing. A common mistake here is not to type the space after the two percentage signs (it's worth explicitly saying this, because at least one person generally does it).

Help and Documentation.

How do we know how to get started, and which functions we need to use? Emphasise two main entry points into the documentation:

- If the function name is known, use the F1 key to access immediate pop-up help.
- If the function name is unknown, search in the box in the top-right.

Function hints can be shown with the shortcut Ctrl+F1.

Import data from a single file containing heterogeneous data.

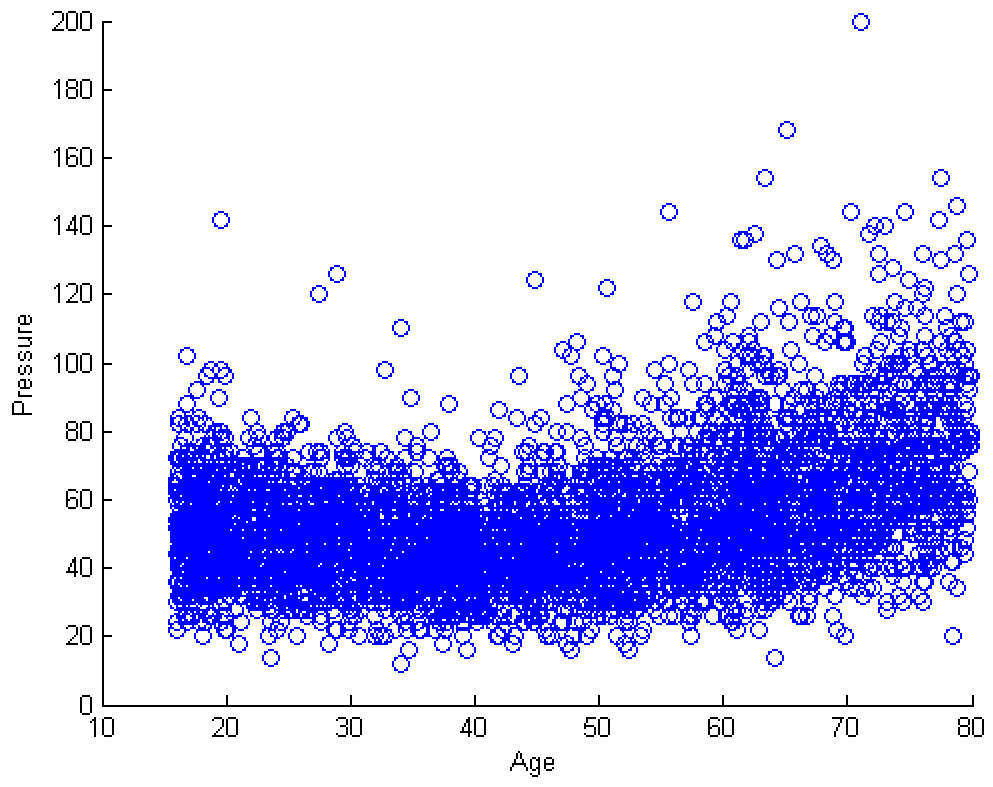
We will use a table to hold the results. Tables are for storing heterogeneous tabular data.

```
% Inspect the data file:
edit('MedicalData.txt')

% Use READTABLE to import the data. If unfamiliar with this function, this
% is a good opportunity to demonstrate the use of F1/searching in the
% documentation. It's also worth mentioning the table of import/export
% functions at this point. Search for "Supported File Formats" to bring up
% this table in the documentation, and point out the use of the READTABLE
% function.
medData = readtable('MedicalData.txt', 'Delimiter', '\t');

% Why are people so old? They're not really, it's just that Age is recorded
% in months. So we might want to convert the data.

% Table variables are accessed using the dot syntax (similar to structures,
% if people are familiar with those).
medData.Age = medData.Age/12; % Note: age is recorded in months.

% Best practice would be to have comments explaining any "magic numbers" in
% the code. For example, where did the 12 come from?
% As an alternative, we could use "self-commenting" variables, e.g.
months2years = 1/12;

% Compute pulse pressure and store this as an additional table variable.
medData.BPDiff = medData.BPSyst1-medData.BPDias1;

% What about naming conventions? Generally we would use camel case.
% Table/structure variable names can be capitalised. Try to get people to
% agree on some convention, and then stick with it.
```

Plotting vectors and annotating graphs.

How can we visualise the information we have just computed? It's worth showing all available plots from the Plots tab at this point.

```
figure % New figure.

% Basic discrete plot and annotations.
scatter(medData.Age, medData.BPDiff)
xlabel('Age')
ylabel('Pressure')
```



Low-Level File I/O.

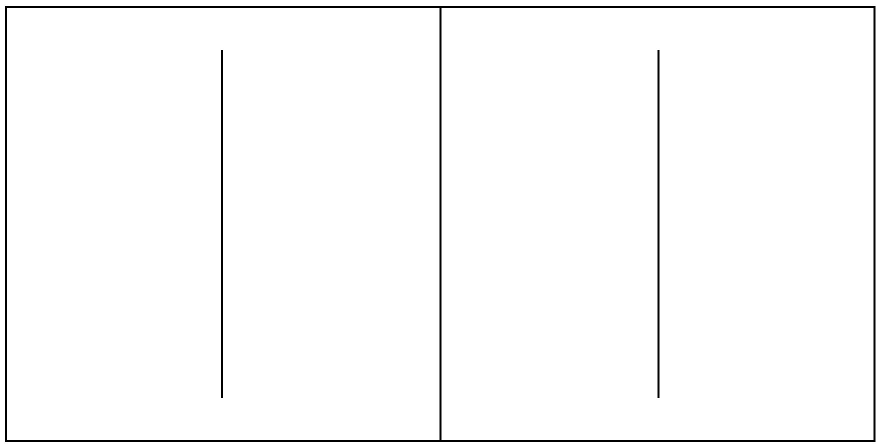
Suppose we have data from a file containing numeric data. How can we read this in efficiently? Three-step process, using textscan for text files.

```
% The open/close statements are essential. Ask the audience: what happens
% if the file remains open? Why is it good practice to ensure files are
% closed after use?
fileID = fopen('HeightWaistData.txt');

% The next step is critical. This is where you can control the precise
% format MATLAB will use to import the data. See the documentation for
% textscan for more details.
dataFormat = '%f %f'; % Two columns of numeric (double, floating point) data.
heightWaistData = textscan(fileID, dataFormat, ...
    'Headerlines', 1, 'Delimiter', '\t');

fclose(fileID);

% You can ask here: do we really need to use DOUBLE to store the data? How
% many significant figures are there? What else could we use to save
% memory? (SINGLE, specified via %f32 in TEXTSCAN).
```

Cell Arrays.

The data from textscan is imported as a cell array in the MATLAB workspace. The data is imported into a 1x2 cell array (each column of data is stored in a separate cell). A useful visualisation function for inspecting cell arrays is CELLPLOT. This can help to alleviate misunderstanding of how cell arrays work.

```
figure; cellplot(heightWaistData)
```



Converting Data Types.

All the elements of a MATLAB array must be the same type. This rule is particularly important when attempting to concatenate data of different types (e.g. strings and numbers). In the documentation, you can show the reference page at: MATLAB -> Language Fundamentals -> Data Types -> Data Type Conversion which has a comprehensive list of conversion functions.

```
% Convert the data to a useful format.
HW = cell2mat(heightWaistData);

% Alternatively, you can show HW = [heightWaistData{1}, heightWaistData{2}]
% or even HW = [heightWaistData{:}]. However, this will require explanation
% of cell indexing (curly brackets give contents) versus standard array
% indexing (round brackets give a sub-cell array).

% Try computing the mean of each variable.
disp(mean(HW))

% Why does this not give us "proper" values? NaNs - missing data. We would
% like to remove them to obtain proper results. However, we need to know
% how to index into arrays before attempting this.
```

```
      167.44          NaN
```

Accessing Data in Arrays.

Data in MATLAB can be accessed using several indexing methods. Generally, row/column, or subscript, indexing is the natural method to introduce first. Useful tips here include the use of the "end" keyword, as well as the use of the colon as an index as a shortcut for 1:end.

Emphasise that row/column indices do not have to be scalar values.

```
% Extract height and waist data separately.
Height = HW(:, 1);
Waist = HW(:, 2);
```

Dealing with Missing Data.

This requires use of logical indexing (passing logical arrays as indices to other arrays). The key function here is ISNAN (it's worth showing the reference page IS* in the documentation which has a list of all useful functions for detecting certain conditions or states). Also, you can show the reference page >> doc ops which has a list of all MATLAB's operators, including logical operators which are required here.

```
% Remove any observations containing NaNs.
badObs = isnan(Height) | isnan(Waist);
cleanHWData = HW(~badObs, :);
disp(mean(cleanHWData))
```

```
        167.47       97.612
```

Importing Data from Multiple Files.

We have seen how to import homogeneous and heterogeneous data from single files. Often, researchers have multiple data files with similar formats, so it is useful to know how to batch process a list of files. The documentation contains a list of functions for performing file and directory management, such as changing directory, moving files, and creating directories: MATLAB -> Data and File Management -> File Operations -> Files and Folders.

```
% The first step is to form a list of files to import data from.
% Warning: be careful here to avoid platform-dependent issues. Use of
% FILESEP is generally recommended to avoid hard-coding slashes or
% backslashes. This should recover a list of .txt files in the "ArmsLegs"
% directory.
fileInfo = dir(['ArmsLegs', filesep, '*.txt']);
fileList = {fileInfo.name};
```

Programming Constructs.

Here, we need to operate on each data file. We know how many data files there are, so a for-loop is appropriate. Use ISKEYWORD to display a list of MATLAB's programming keywords. You may want to mention other common constructs such as if-elseif-else, try-catch, switch-case and while-loops.

Structure Arrays.

We would like to store the name of each file (a string) as well as the numerical data it contains (a matrix with three columns) together in the same variable. Tables are not suitable for this, because in a table all variables must have the same number of observations. We can use a structure array instead. This allows access to the data using the same convenient dot syntax we used for tables, but also allows storage of mixed-size data. (However, structures have far less available functionality associated with them than tables. For example, SORTROWS comes with tables but not with structures.)

```
% You can show the loop being performed sequentially to begin with, and
% then initiate a discussion about the Code Analyzer warning that comes up.
% This should lead to a discussion about preallocation, which can be shown
% explicitly using the STRUCT function, for example. However, in this case
% it's much easier just to do the loop backwards and therefore have the
% preallocation performed implicitly.
for k = numel(fileList):-1:1
    ArmsLegs(k).FileName = fileList{k};
    ArmsLegs(k).Data = dlmread(fileList{k}, '\t', 1, 0);
end
```

Concatenation.

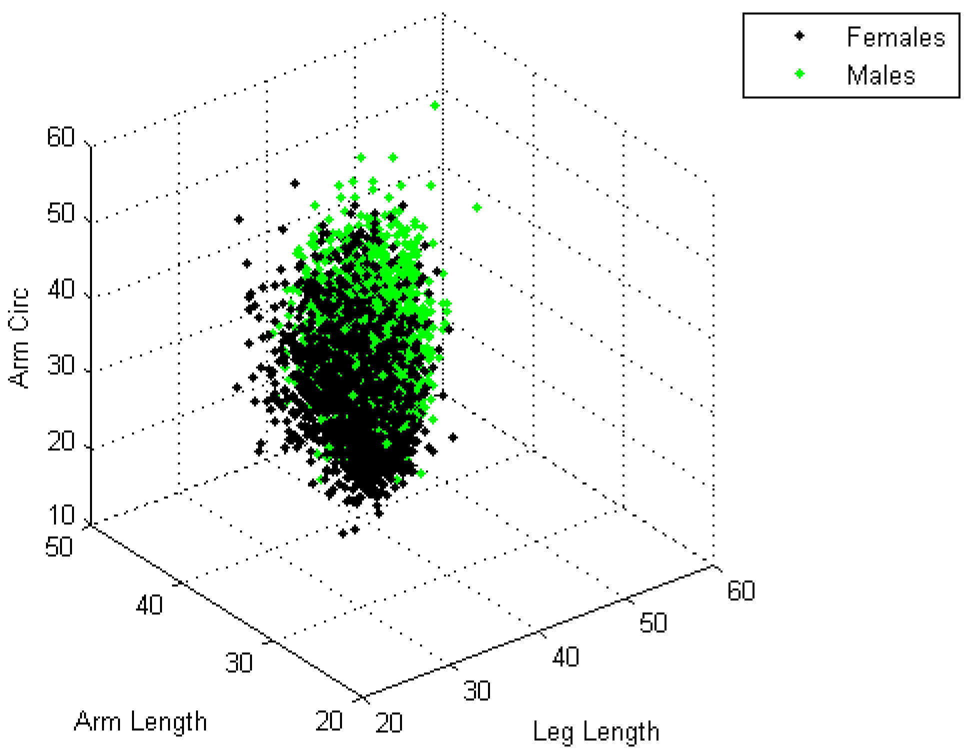
After reading all the data, we would like to merge it into one variable so that we have all of the body measurements together. It's worth spending a bit of time talking about concatenation with square brackets [], and then mention that for comma-separated lists, you can use VERTCAT, HORZCAT and CAT.

```
ALData = vertcat(ArmsLegs.Data);
```

Logical Indexing.

We have seen logical indexing above. We used it to detect and remove missing values. It is an important and widely applicable MATLAB language feature. For example, we might want to visualise the split/separation between two different groups in our population, e.g. males and females.

```
females = strcmp(medData.Sex, 'F');
figure
scatter3(ALData(females, 1), ALData(females, 2), ALData(females, 3), 'k.')
hold on % Ask what happens if we don't use HOLD here.
scatter3(ALData(~females, 1), ALData(~females, 2), ALData(~females, 3), 'g.')
hold off
xlabel('Leg Length')
ylabel('Arm Length')
zlabel('Arm Circ')
legend('Females', 'Males')
```



Saving and Loading MAT-Files.

We have spent a lot of time and effort into importing our data effectively. Do we really want to repeat this effort if we need the data again? Probably not. You can save all the data to disk using a MAT-file (designed to hold MATLAB data and variables). Loading from MAT-files is also faster in general terms than running file import code. The idea is to import the data only once, and then save it to a MAT-file for future use or distribution to others.

```
% Save data to a MAT-file.
save('S01_HealthData.mat')
```

Publishing Code.

At this point, we have a nice script summarising what we've done. To share results with others and for the purposes of presentation, we can export the MATLAB code to a supported external format (HTML, XML, TEX, PDF, DOC, PPT). Publishing code is a key reason to use sections when writing MATLAB scripts, as sections will automatically be interpreted as boundaries in the published documents. There are many text mark-up options available from the "Publish" menu, including insertion of LaTeX code, bullet lists, enumerated lists, images and hyperlinks.

Published with MATLAB® R2013b

Chapter 02 - Algorithm Design in MATLAB.

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

This chapter provides the necessary programming techniques for algorithm design and development using MATLAB. The idea is that we are now happy writing scripts to organise and collect sequences of MATLAB commands, but these become increasingly difficult to manage as the complexity of our algorithms grows. Moreover, a common requirement is to run scripts repeatedly using different values for certain parameters within the script. Functions are a more effective programming construct for managing these issues. In this chapter we will learn how to develop and structure an algorithm for performing simple preprocessing, model fitting and visualisation. The important concept here is how to modularise code so that it becomes reusable, maintainable, flexible and robust.

Outline:

- Formulating an algorithm for 1D model fitting
- Linear regression models
- Visualising the results
- Generalising the algorithm to 2D model fitting
- Anonymous function handles
- Surface plots
- Code modularisation: from scripts to functions
- Local functions
- Nested functions
- Code robustness and flexibility
- Parsing user-supplied input arguments
- Defining flexible interfaces
- Errors and error identifiers

Reference files for this chapter:

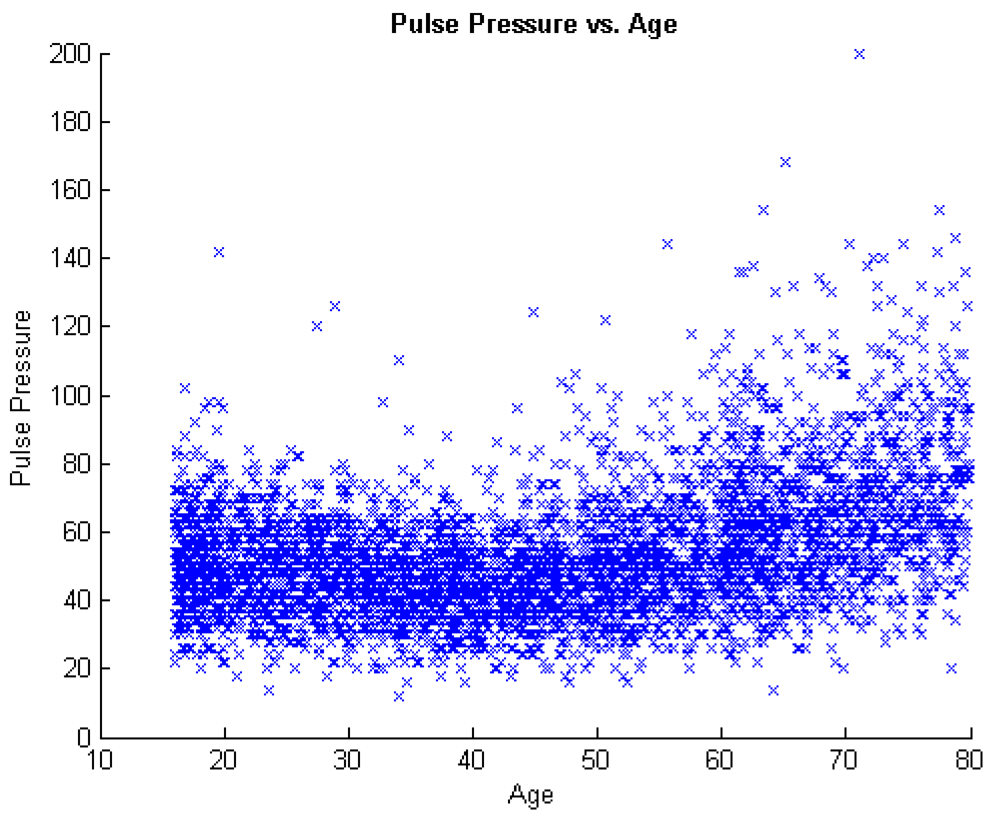
- ../Reference/S02_Algorithm.m
- ../Reference/F02_fitQuadModel.m
- ../Reference/S02_MedData.mat

Contents

Linear Regression Models.

The main example in this chapter is developing an algorithm to perform data preprocessing, fitting and visualisation. We will use a linear regression model with variables chosen from the biomedical data discussed in the first chapter of the course. The examples are fairly simple, but if attendees are unfamiliar with regression, the concepts can be explained in terms of inputs and outputs. For example, we have an input variable (e.g. Age) and we would like to model the dependency of an output variable (e.g. PulsePressure) on the input. It's easier to gather the commands in these sections into a single script, which can then be run section by section and then eventually converted into a properly structured function by the end of the chapter.

```
% Visualise the dependency of pulse pressure on age.
load('S02_MedData.mat')
x = MedData.Age;
y = MedData.BPDiff;
figure
scatter(x, y, 'bx')
xlabel('Age')
ylabel('Pulse Pressure')
title('\bfPulse Pressure vs. Age')

% This dependency looks as if it could be modelled using a quadratic curve
% (i.e. using a model of the form PP ~ C0 + C1*Age + C2*Age.^2, where the
% coefficients C0, C1 and C2 are unknown and to be determined).
```



Matrix Equations.

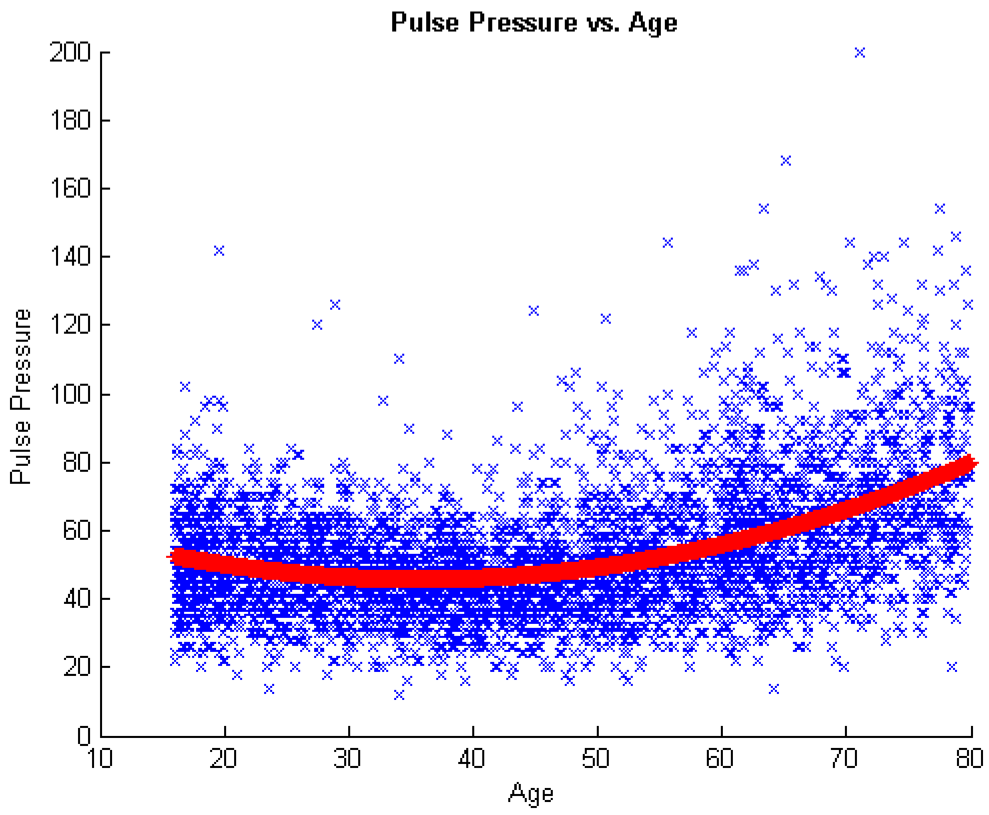
Formulating a linear regression model leads to a system of linear equations A*x = b for the unknown vector of coefficients x. The design matrix A comprises the model terms, and the vector b is the data to be modelled (in this example, the vector of pulse pressure observations). Note carefully that solving such linear systems is a matrix operation, and so any missing values contained in the data must be dealt with in some way before attempting to solve the system. (High-level functionality in Statistics Toolbox can remove missing observations automatically before fitting, but here we use a core MATLAB solution.)

```
% Set up the system of equations for fitting a model to the pulse pressure
% data.
missingIdx = isnan(x) | isnan(y);
xClean = x(~missingIdx);
yClean = y(~missingIdx);
designMat = [ones(size(xClean)), xClean, xClean.^2];
```

Slash and Backslash.

The least-squares solution of the system A*x = b in MATLAB is given by x = A\b. If the system is formulated as x*A = b, then its solution is x = b/A.

```
% Solve the linear system to find the best-fit coefficients for modelling
% pulse pressure as a quadratic function of age.
modelCoeffs = designMat\yClean;

% Compute the fitted model.
pulseModel = designMat*modelCoeffs;

% Visualise the results.
hold on
plot(xClean, pulseModel, 'r*')
hold off
```



Matrix and Array Operators.

Note that in the above computations we have used the operators * and \. A complete list of matrix, array and logical operators is available by entering the command >> doc ops.

```
doc ops
% It's useful to show this list at this point to point people in the right
% direction for discovering matrix and array operators.
```

Generalising the Model.

We now have an algorithm for fitting a one-dimensional quadratic model to our data. We would like to generalise this to fit a two-dimensional quadratic surface to a single response variable given two input variables. For example, we might be interested in predicting a person's weight given knowledge of their height and waist measurements. We will follow similar steps to preprocess and clean the data before fitting.

```
% Extract input variables.
x1 = MedData.Height;
x2 = MedData.Waist;

% Extract output variable.
y = MedData.Weight;

% Clean the data.
missingIdx = isnan(x1) | isnan(x2) | isnan(y);
x1Clean = x1(~missingIdx);
x2Clean = x2(~missingIdx);
yClean = y(~missingIdx);

% Formulate the system of linear equations.
designMat = [ones(size(x1Clean)), x1Clean, x2Clean, x1Clean.^2, ...
             x2Clean.^2, x1Clean.*x2Clean];

% Solve the system.
modelCoeffs = designMat\yClean;
```

Function Handles.

Now that we have the second set of model coefficients for the 2D quadratic model, we would like to visualise the fitted results. To do this, we need to substitute the model coefficients into the model equation. One way to achieve this is to use a function handle, which is a variable containing a reference to a function. Using a function handle reduces the problem of calling a function to that of accessing a variable. The function handle contains all information necessary to evaluate the function at given input arguments. For further information on function handles, see MATLAB -> Language Fundamentals -> Data Types -> Function Handles.

```
modelFun = @(c, x1, x2) c(1) + c(2)*x1 + c(3)*x2 + c(4)*x1.^2 + ...
                        c(5)*x2.^2 + c(6)*x1.*x2;
% It's helpful to draw the analogy with the mathematical notation:
% f(c, x1, x2) = c(1) + c(2)*x1 + c(3)*x2 + c(4)*x1.^2 + ...
%                       c(5)*x2.^2 + c(6)*x1.*x2
% The only difference between the mathematical notation and the function
% handle definition is the "@" symbol, signifying to MATLAB that a function
% handle is being created.
% This function handle can now be evaluated at any point or any series of
% points, just like a normal mathematical function. For example:
disp(modelFun(modelCoeffs, 0, 0)) % Evaluates the model function at (0,0).
```

```
        11.661
```

Creating Equally-Spaced Vectors.

At this point, we are ready to visualise our quadratic surface. In MATLAB, surface plots can be created using a fairly straightforward step-by-step procedure. The first step of this process is to create equally-spaced vectors of points of "x" and "y" data (the data forming the horizontal plane). To achieve this, we could use either the colon operator (:) or the LINSPACE function.

```
x1Vec = linspace(min(x1Clean), max(x1Clean));
x2Vec = linspace(min(x2Clean), max(x2Clean));
% Note that 100 points in the vector is the default. The number of points
% is optional and can be specified as the third input argument to LINSPACE.
```

Making Grids.

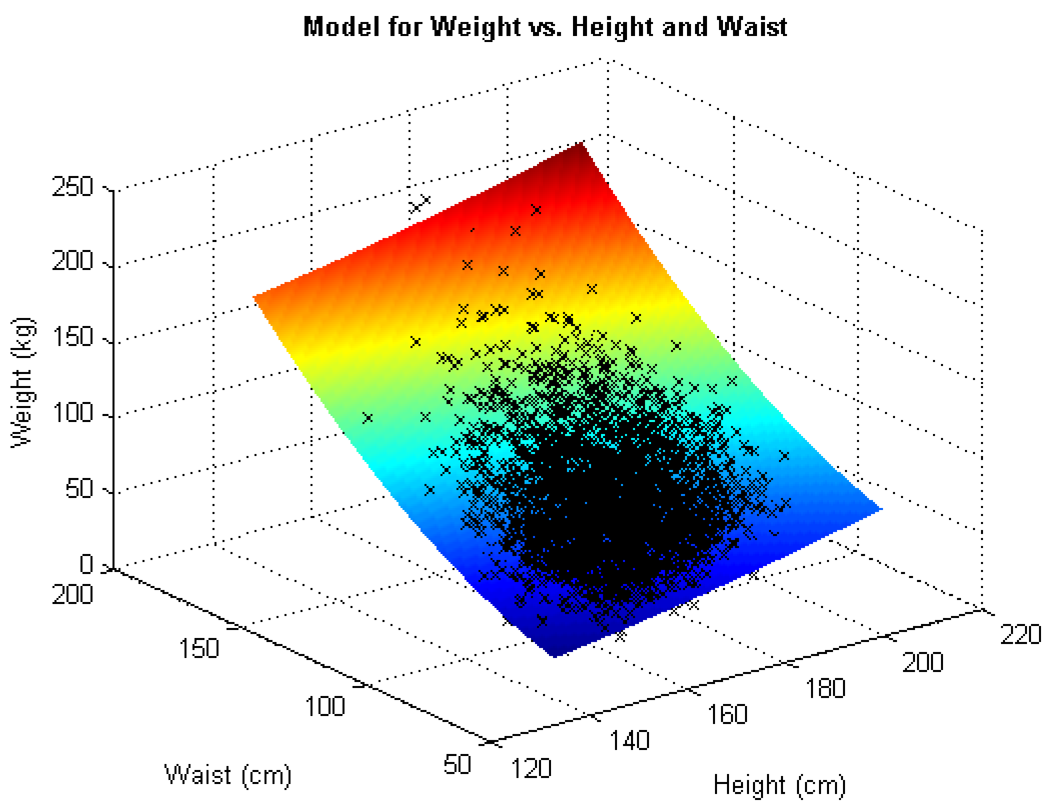
The second step in the procedure is to create matrices containing the coordinates of all grid points in the lattice. This can be achieved using MESHGRID.

```
[X1Grid, X2Grid] = meshgrid(x1Vec, x2Vec);
% X1Grid contains all the "x" coordinates of all the lattice points.
% X2Grid contains all the "y" coordinates of all the lattice points.

% Now that we have all lattice points, our surface function can be
% evaluated over the lattice using the function handle defined above.
YGrid = modelFun(modelCoeffs, X1Grid, X2Grid);
```

Surface Plots.

Finally, we can visualise the surface. The basic function here is SURF, into which we provide the three matrices of surface data (the two coordinate matrices and the values of the surface function over the lattice).

```
figure; surf(X1Grid, X2Grid, YGrid)
% We can customise this display by specifying name-value pairs.
surf(X1Grid, X2Grid, YGrid, 'FaceColor', 'interp', 'EdgeAlpha', 0)
% This removes gridlines and uses interpolated shading for the surface,
% instead of colouring each patch a uniform colour.

% View the original data.
hold on
plot3(x1Clean, x2Clean, yClean, 'kx')
xlabel('Height (cm)')
ylabel('Waist (cm)')
zlabel('Weight (kg)')
title('\bfModel for Weight vs. Height and Waist')

% Other 3D visualisation functions can be found in the Plot Gallery.
```



Creating a Function.

At this point, we now have a script which we can run to perform our data clean-up, model fitting and visualisation. However, it is not straightforward to apply the algorithms in our script to another set of data. The next step in modularising our code is to create a function to automate the tasks or subtasks represented in our current script. Functions are created in text files with the .m extension, just like script files. However, functions must begin with a function declaration of the form: function [out1, out2, ...] = function_name(in1, in2, ...) The keyword "function" must be the first non-comment code in the file. Syntax for calling user-defined functions is identical to that used for calling existing MATLAB functions.

```
% Recommended activity: create a new function "fitQuadModel" with the
% following declaration line:
% function [modelCoeffs, fh] = fitQuadModel(X, y, showplot)
%
% See the file F02_fitQuadModel_001.
```

Workspaces.

Functions operate within their own function workspace, which is separate from the base workspace accessed at the prompt or from within scripts. If a function calls another function, each maintains its own separate workspace. Once a function has completed, its workspace is destroyed. Instructors may wish to demonstrate the concept of separate workspaces by setting breakpoints and entering debug mode.

Local Functions.

There may be times when we wish to outsource certain tasks in a function to another function. For example, if we think about our algorithm at a high-level, we are performing three distinct tasks: * we are cleaning up the data by removing NaNs; * we are performing the model fitting; * we are visualising the results of the fitted model.

We are performing these three tasks on two distinct sets of data. One possibility for structuring our main function is to have three local functions defined inside, each of which represents one of the three tasks above. This enables us to reuse the same function for multiple different sets of data.

```
% Recommended activity: write separate local functions removeNaNs, fitModel
% and visualiseResults inside the main fitQuadModel function file.
% See the file F02_fitQuadModel_002.

% Next, start to move the code from the script into the appropriate places
% in the function. We will need to deal with the cases of 1D data and 2D
% separately, although some code is appropriate for both cases.
%
% See the file F02_fitQuadModel_003.
%
% Nested functions are another construct which could be introduced at this
% stage of the course. These become more important when building
% interactive user applications, so if this was the main interest of
% audience members, then it makes logical sense to introduce them here.
% Otherwise, they can be briefly mentioned or even just skipped, since we
% don't need them for this particular example.
```

Defining Flexible Interfaces.

As we have written it, the fitQuadModel function requires three inputs, the X data, the y data, and a logical flag indicating whether or not the data should be plotted. MATLAB will return an error message unless exactly three inputs are provided whenever fitQuadModel is called. We can use the NARGIN function to determine how many input arguments were provided when the function was called. With this information, we can set default input values if necessary.

```
% Recommended activity: allow some flexibility for the end user by making
% the showplot input argument optional (set its default value to false, for
% example).
%
% See the file F02_fitQuadModel_004.
```

Checking Input Arguments.

Error conditions often arise because run-time values violate certain implicit assumptions made in code. When designing algorithms, especially those that will be used by others, it is important to catch any user-introduced errors. These may not become apparent until later on in the code. In a weakly-typed language such as MATLAB, checking types and attributes of user-supplied input data is particularly important. Recommended approaches include using the family of is* functions (see doc is*) and the more sophisticated function VALIDATEATTRIBUTES.

```
% Recommended activity: ask the audience to make a list of assumptions that
% we are making about the X, y and showplot input arguments. Use
% VALIDATEATTRIBUTES and if-statements to respond to errors introduced by
% users and provide meaningful feedback. At the end of this activity we
% should have a correct, robust functional interface. Also check that we
% have between 2 and 3 input arguments using NARGINCHK.
%
% See the file F02_fitQuadModel_005.
```

Errors and Error Identifiers.

When unexpected conditions arise in a MATLAB program, the code may issue an error or a warning. Custom errors can be implemented using the ERROR function. Best practice here is to use an error identifier as part of the custom error, to enable users to diagnose and debug problems more readily. The error identifier is a string which should contain a colon (:), and is the first input argument to the error function. For example, an error identifier might be 'fitQuadModel:EmptyMatrix'.

```
% Recommended activity: implement custom errors with custom error
% identifiers so that the input arguments satisfy the following criteria:
%
% * X has at least three rows.
% * The number of elements of y coincides with the number of rows of X.
% * X has either one or two columns.
% * All values in both X and y should be finite.
%
% Implement the following robustness conditions as well.
%
% * If removing NaNs from the data results in empty arrays, issue a custom
% error.
% * If the design matrix resulting from the data has deficient rank, issue
% a custom error.
%
% See the file F02_fitQuadModel_006.
%
% At the end of this chapter, we now have a completely robust, flexible
% function which can be used to compute and visualise quadratic models for
% any given 1D or 2D dataset with one response variable. This can be
% contrasted with the script we started with after originally developing
% the algorithm.
```

Published with MATLAB® R2013b

Chapter 03 - Test and Verification of MATLAB Code.

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

This chapter focusses on formal test and verification procedures for MATLAB algorithms, using the unit testing framework introduced in version R2013a of MATLAB (note that R2013a introduced an object-oriented test framework, and R2013b introduced a function-based test framework). In this chapter we are working exclusively with the function-based unit testing framework. The idea is that we have now written an algorithm in MATLAB, and before it is deployed in a real situation, we should take some measures to ensure that the code is robust and meets its required specifications. In the previous chapter we claimed that the resulting function was robust, flexible and responds correctly to error conditions, but we have not tested this in any way. In this chapter we will learn how to use the function-based unit testing framework to write function-based unit tests and formally verify correct behaviour of our algorithm.

Outline:

- The MATLAB Unit Testing Framework
- Function-based unit testing
- Local functions
- The test environment
- Organising test paths and test data
- Setup and teardown functions
- Effective test design
- Writing test functions
- Testing robustness of function interfaces
- Testing numerical algorithms
- Test design considerations
- Running tests and evaluating results

Reference files for this chapter:

- ../Reference/test_F03_fitQuadModel.m
- ../Test_Data/fitQuadModel_TestData.mat

Contents

Unit Testing Framework: Overview.

Unit testing refers to the process of verifying that source code is fit for purpose, meets design specifications and exhibits correct behaviour. The term “unit” refers to the individual entities of source code under test. For example, in MATLAB, a unit could be a function which we have designed to perform certain tasks (e.g. removing NaNs from data or determining regression coefficients). A unit test may be regarded as a means of formalising the specifications that code is required to meet. Writing and using unit tests provides several benefits, including:

- early identification of potential problems and bugs in code;
- facilitation of subsequent code changes (i.e. ensuring that code modification in the future does not alter code correctness);
- greater motivation to modularise code into smaller, individually testable units;
- clearer documentation of code requirements and specifications.

In this chapter we will focus on the function-based approach to unit testing in MATLAB. This approach is documented under "MATLAB" --> "Advanced Software Development" --> "Unit Testing Framework" --> "Write Unit Tests" --> "Write Function-Based Unit Tests".

Function-Based Unit Testing.

Function-based unit testing is available in MATLAB from release R2013b onwards. Function-based unit testing allows MATLAB programmers with experience of writing functions to quickly develop unit tests. The first step in function-based unit testing is to create the main test function file. This is a single MATLAB file containing a main function and all the individual local test functions. The main function must be named using the word "test" either at the beginning or at the end of the name. This is not case-sensitive. For example,

- test_myAlgorithm
- myAlgorithmTest
- TestMyAlgorithm
- testMyAlgorithm

are all valid main test function names. The main function has no input arguments and returns one output (a test array gathering each of the individual unit tests into one array). The main test function must collect all of the local test functions into a test array. This test array is generated via a call to the functiontests function, which assembles the test array using a cell array of function handles to the local functions in the file. In turn, a cell array of function handles to the local functions in the main file is generated using a call to localfunctions. In order to be recognised as valid individual unit tests, local functions must include the case-insensitive word “test” at the beginning or end of the function name, in exactly the same way that the main test file must be named in order to be recognised as a valid test file. The main test file is sometimes referred to as a "test harness", because it collects together all of the local tests.

Recommended activity: create a new test file for fitQuadModel and name it test_fitQuadModel. Define the test array output using functiontests and localfunctions.

See the file test_F03_fitQuadModel_001.

Local Test Functions.

The second step in creating function-based unit tests is to write the local test functions. These are included as local functions in the main test-generating function. To indicate that a local function should be used as a local test function, include the case-insensitive word "test" either at the beginning or at the end of its name. Local test functions have a fixed interface. Each local test function must accept a single input argument (a function test case object) and return no outputs. The MATLAB Unit Testing Framework automatically generates the function test case object. A typical local test function therefore has the following basic skeleton:

function test_localFunc(testCase) … end

The function test case object can be used to pass data between the local test functions.

Recommended activity: write local test function stubs for the following unit tests:

- test_nargin
- test_invalidInputs
- test_validInputs
- test_basicFit
- test_allNaNs
- test_deficientRank

Call the test harness with one output to ensure that an array of tests is returned.

See the file test_F03_fitQuadModel_002.

Setup and Teardown Functions.

Before running tests, it is often required to change directory or load some data from a MAT-file. We may also require a certain system state to be established before running a test. For example, we may require a figure to be open before testing a graphical function, or we might need to set a specific starting point in a random number stream in order to test a Monte Carlo simulation. Setup and teardown code, also referred to as test fixture functions, set up the pre-test state of the system and return it to the original state after running the test. This is important because the results of tests should be completely reproducible, and we also do not want to interfere with existing settings or the existing system state. Anything that is modified in the setup code before running the tests should be "undone" in the teardown code. There are two types of these functions: file fixture functions that run once per test file, and fresh fixture functions that run before and after each local test function. As with local test functions, the only input to test fixture functions is a function test case object, which is automatically passed to each function by the unit test framework. The function test case object is a means to pass information between setup functions, local test functions, and teardown functions. By default, every test case object has a TestData property which is a 1x1 struct. This allows easy addition of fields and data to the TestData property, which updates the function test case object. Typical uses for this test data include paths and graphics handles. File fixture functions are used to share setup and teardown functions across all the tests in a file. The names for the file fixture functions must be setupOnce and teardownOnce, respectively. These functions execute a single time for each file. On the other hand, fresh fixture functions are used to set up and tear down states for each local test function. The names for these fresh fixture functions must be setup and teardown, respectively.

Recommended activity: add file fixtures to the main test file. The setup file fixture should add the Test_Data folder to the path, load data from a MAT-file and save the current rng status as follows:

- addpath('Test_Data')
- testCase.TestData = load('fitQuadModel_TestData.mat');
- testCase.TestData.currentRNG = rng;

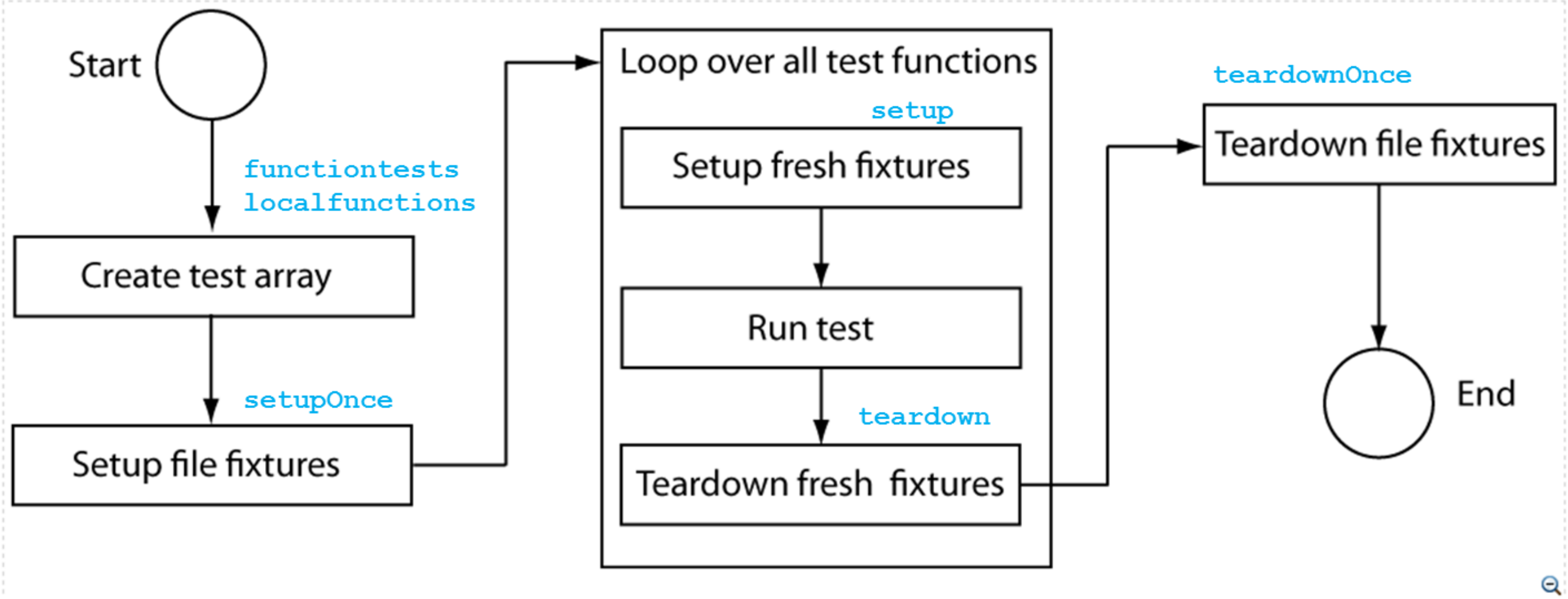
The teardown file fixture should undo these operations and restore the original state of the system, as follows:

- rmpath('Test_Data')
- s = testCase.TestData.currentRNG;
- rng(s)

See the file test_F03_fitQuadModel_003.

Task Execution.

At this point, we have the basic layout of the main test function in place. However, we have not written any test code. The diagram below illustrates the sequence of tasks performed by the unit testing framework when we come to run our tests. Notice that file fixture functions are only invoked once per test file, whereas fresh fixture functions are invoked once per local test function. Our next task will be to implement the test code for each individual local test function. When this is complete, we will then be able to run the tests, which initiates the sequence of tasks detailed in the diagram below.



Writing Test Functions - Workflow.

Our next step is to write the individual local test functions within the main test function file. We will use the following workflow when designing and writing tests:

- Select the appropriate qualifications for testing values and responding to failures.
- Select the type of the test.
- Write the local test function code using the appropriate functions determined from steps 1 and 2.

We will discuss steps 1 and 2 in detail during the next two topics in this chapter.

Select Qualifications.

Qualifications are functions for testing values and responding to failures. Within the MATLAB unit testing framework, there are four types of qualifications, and it is important to choose the most appropriate qualification when designing individual unit tests.

- Verifications: verifications produce and record code failures without throwing an exception. This means that any remaining local test function code will still run. In the event of verification failure, the unit testing framework will mark the test as Failed. In general, use verifications when you want to test other conditions in the presence of some code failures.

- Assumptions: assumptions ensure that a test runs only when certain prerequisite conditions are satisfied. Importantly, the event of assumption failure does not produce a test failure. When an assumption failure occurs, the unit testing framework marks the test as Incomplete. In general, use assumptions when you want to verify that certain prerequisite conditions are in place before running the remainder of the test, but you do not want the failure of the prerequisite conditions to count as a test failure.

- Assertions: assertions ensure that the preconditions of the current test are met. In the event of assertion failure, the test is marked as Failed by the unit testing framework. Any remaining code in the local test function will not be executed; the unit testing framework will proceed to the next local test function. In general, use assertions when you want to verify that certain prerequisite conditions are in place before running the remainder of the test, and you want the failure of the prerequisite conditions to count as a test failure.

- Fatal assertions: fatal assertions terminate the test running process. This means that if a fatal assertion fails, the test process will stop and an exception will be thrown. Any remaining local test function code will not run, and any remaining local test functions will not run either. Fatal assertions would generally be used when a failure at the assertion point renders the remainder of the current tests invalid. For example, it might not make sense to run subsequent tests if it is known that a previous test has failed.

Select Test Type.

After selecting the qualification, the second step in the workflow when writing unit test code is to choose the type of the test. There is a large number of preexisting functions for testing specific conditions. For full details of each type, see the documentation available at "MATLAB" --> "Advanced Software Development" --> "Unit Testing Framework" --> "Write Unit Tests." For each type of test, the function name is prefixed with either "verify", "assume", "assert" or "fatalAssert" as appropriate.

Testing Robustness of Function Interfaces.

We are now in a position to write implementation code for the individual local test functions (the final step of the workflow for writing test functions). As a first step, we will implement tests for robustness of the function interface. This means that we are testing the code to ensure it correctly handles the following situations:

- the function is invoked with an incorrect number of input or output arguments;
- the function is invoked with correct numbers of input and output arguments, but with incorrect input arguments (i.e. inputs with the incorrect data type, dimension or other attributes).

In this situation, many test conditions are similar and the failure of any given condition does not imply that we wish to terminate the test process. In each test condition, we require to check that the function under test throws an exception when user-supplied input data is incorrect. We will therefore select "verification" as our qualification, and "error" as our test type, which leads us to the verifyError function.

The verifyError function is used with the following syntax:

verifyError(testCase, expression, identifier, diagnostic)

Here,

- testCase is the function test case object passed as the first input to the local test function;
- expression is a function handle to the code under test – usually this will be a function handle with no input arguments which in turn invokes the function under test with the required input arguments;
- identifier is a string representing the identifier of the expected exception thrown by the function under test;
- diagnostic is an optional input argument and represents the diagnostic information to display to the user on the event of failure.

The identifier of the last exception issued by MATLAB may be retrieved as follows.

- Return the last exception using the syntax: lastExc = MException.last;
- Extract the identifier of this exception by accessing the identifier property: lastID = lastExc.identifier;

Recommended activity: write code for the local test functions test_nargin and test_invalidInputs implementing the following test conditions.

- fitQuadModel should throw an exception when called with 0, 1 or 4 or more input arguments;
- fitQuadModel should throw an exception unless all of the following conditions are satisfied:

1. the first input argument, X, is a real, 2D, nonempty double matrix with no infinite values;
2. the second input argument, y, is a real, nonempty column vector of type double with no infinite values;
3. the third input argument, showplot, is a logical scalar value;
4. X has at least three rows and at most two columns;
5. the number of rows of X and y coincide.

See the file test_F03_fitQuadModel_004.

Testing Numerical Algorithms.

In addition to verifying that our function exhibits correct behaviour when invoked with incorrect input arguments, we will also test that it returns correct numerical output values. In general terms, testing this type of condition is more subtle than testing the robustness of the function interface. One reason for this is because the programmer is required to think about what constitutes reasonable testing for the particular numerical algorithm under consideration, as well as a number of edge cases which may not be common in practice, but could still cause problems if they arise during the operation of the algorithm.

For our particular algorithm, we will test for the following conditions.

- When the function is called with two vectors, the results should be a 3x1 double vector.
- When the function is called with a vector a matrix with two columns, the results should be a 6x1 double vector.

- When the function is called with particular input data for which we know the exact quadratic curve passing through the points, it should return the known coefficients within a certain error tolerance.
- The results from our fitting algorithm should coincide with the results from other available fitting functions in MATLAB (to within a certain error tolerance) such as linsolve and polyfit.
- When the input data is such that all observations are removed due to the presence of at least one NaN in each row, our fitting function should throw an exception.
- In the case when the input matrix X produces a rank-deficient design matrix, the fitting algorithm should throw an exception.

Recommended activity: write code for following local test functions implementing the test conditions abpve.

- test_validInputs
- test_basicFit
- test_allNaNs
- test_deficientRank

See the file test_F03_fitQuadModel_005.

Running Tests and Evaluating Results.

Finally, we need to run the test suite and evaluate the results. When all of the local test functions have been written and the main function test file is complete, the results of running the tests are obtained using the runtests function. The runtests function takes the name of the main function test file as its input argument:

```
results = runtests('test_fitQuadModel');
```

The output is an array of test results, one for each local test function contained in the main function test file. Each element of the test results array has the following properties:

- Name – name of the local test function (a string).
- Passed – logical scalar value indicating whether the test passed.
- Failed – logical scalar value indicating whether the test failed.
- Incomplete – logical scalar value indicating whether the test is incomplete.
- Duration – scalar double value storing the time taken to run the individual local test.

When the test results array is displayed, summary information comprising the total number of test results in each state (Passed, Failed, Incomplete) and the total time taken to run the test suite is shown. Note that you can make use of logical indexing on the test results array to find the test results meeting a certain condition (for example, all tests that have failed or incomplete status, or all tests which took longer than a specified time to run).

Code coverage may be analysed by running the tests in the MATLAB Profiler, and then launching the Code Coverage report from the Current Folder Browser menu.

Recommended activity: run all tests in the test suite and evaluate the results.

```
>> results = runtests('test_fitQuadModel');
```

Use the MATLAB Profiler and Code Coverage report to analyse the test coverage.

```
>> profile on >> results = runtests('test_fitQuadModel'); >> profile off
```

Now run the Code Coverage report from the Current Folder Browser menu.

Published with MATLAB® R2013b

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

Contents

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

% 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.
```

```
Error: File: C:\Users\kdeeley\Desktop\MathWorks\Training\2014
Q1\02_CC01_Manchester_University_January_14th_15th\02_Files\findBestPredictors.m Line: 128 Column: 60
Unbalanced or unexpected parenthesis or bracket.
```

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 from a drop-down list in the Current Folder Browser menu. 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.

Recommended activity: 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')

% Recommended activity: understand, diagnose and fix all warnings
% identified from the Code Analyzer Report (or obtained via the CHECKCODE
% function). This will require knowledge of preallocation (see the
% following section for more information). The functions CELL and NCHOOSEK
% may come in useful here.
%
% See the file F04_findBestPredictors_V1.
```

```
L 85 (C 33-40): The function 'getPairs' might be unused.
L 94 (C 9-11): FOR might not be aligned with its matching END (line 98).
L 95 (C 13-20): The variable 'varPairs' appears to change size on every loop iteration. Consider preallocating for speed.
L 96 (C 13-20): The variable 'varPairs' appears to change size on every loop iteration. Consider preallocating for speed.
L 97 (C 15): Terminate statement with semicolon to suppress output (in functions).
L 128 (C 60): Invalid syntax at ';'. Possibly, a ), }, or ] is missing.
```

Preallocation of Memory.

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.

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 small dash 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.
%
% Recommended activity: diagnose the run-time problems with LOAD and
% accessing structure fields. 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.
```

```
MATLAB:load:couldNotReadFile
```

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.
%
% Recommended activity: run a Dependency Report on the directory to
% understand the dependency problem. Fix the problem.
%
% See the file F04_findBestPredictors_V3 for reference.
```

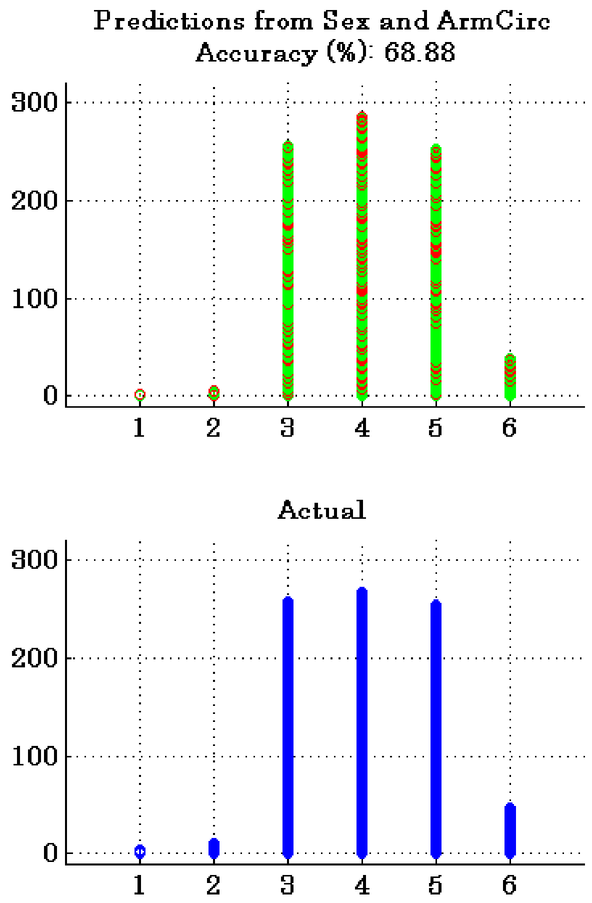
```
MATLAB:UndefinedFunction
c:\Users\kdeeley\Desktop\MathWorks\Training\2014
Q1\02_CC01_Manchester_University_January_14th_15th\02_Files\InstructorMaterials\F04_findBestPredictors_V2.m
'getpairs' not found.
c:\Users\kdeeley\Desktop\MathWorks\Training\2014
Q1\02_CC01_Manchester_University_January_14th_15th\02_Files\InstructorMaterials\F04_findBestPredictors_V2.m (getPairs)  %
Subfunction of F04_findBestPredictors_V2
```

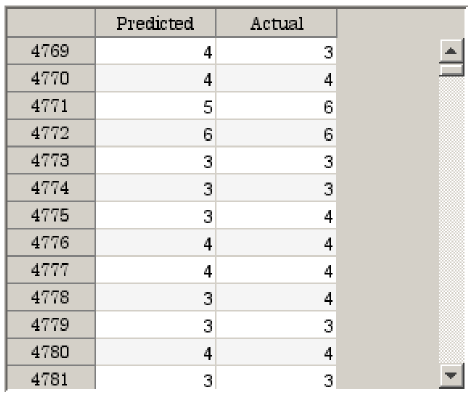
Debugging Completion.

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

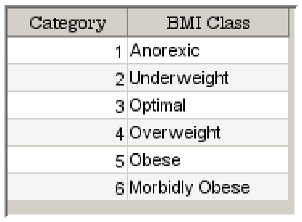
```
predictorRankings = F04_findBestPredictors_V3();
```

```
Accuracy for predictors ArmCirc and Age (%): 59.62
Accuracy for predictors ArmLength and Age (%): 32.07
Accuracy for predictors ArmLength and ArmCirc (%): 65.20
Accuracy for predictors BPDias1 and Age (%): 33.37
Accuracy for predictors BPDias1 and ArmCirc (%): 58.79
Accuracy for predictors BPDias1 and ArmLength (%): 30.17
Accuracy for predictors BPDiff and Age (%): 32.54
Accuracy for predictors BPDiff and ArmCirc (%): 57.96
Accuracy for predictors BPDiff and ArmLength (%): 31.35
Accuracy for predictors BPDiff and BPDias1 (%): 31.47
Accuracy for predictors BPSyst1 and Age (%): 33.73
Accuracy for predictors BPSyst1 and ArmCirc (%): 55.82
Accuracy for predictors BPSyst1 and ArmLength (%): 34.92
Accuracy for predictors BPSyst1 and BPDias1 (%): 30.88
Accuracy for predictors BPSyst1 and BPDiff (%): 31.00
Accuracy for predictors Ethnicity and Age (%): 30.17
Accuracy for predictors Ethnicity and ArmCirc (%): 62.59
Accuracy for predictors Ethnicity and ArmLength (%): 37.65
Accuracy for predictors Ethnicity and BPDias1 (%): 36.22
Accuracy for predictors Ethnicity and BPDiff (%): 32.66
Accuracy for predictors Ethnicity and BPSyst1 (%): 35.99
Accuracy for predictors LegLength and Age (%): 32.66
Accuracy for predictors LegLength and ArmCirc (%): 64.01
Accuracy for predictors LegLength and ArmLength (%): 33.97
Accuracy for predictors LegLength and BPDias1 (%): 28.74
Accuracy for predictors LegLength and BPDiff (%): 27.43
Accuracy for predictors LegLength and BPSyst1 (%): 28.03
Accuracy for predictors LegLength and Ethnicity (%): 31.12
Accuracy for predictors LikeToWeigh and Age (%): 40.02
Accuracy for predictors LikeToWeigh and ArmCirc (%): 67.81
Accuracy for predictors LikeToWeigh and ArmLength (%): 45.49
Accuracy for predictors LikeToWeigh and BPDias1 (%): 49.52
Accuracy for predictors LikeToWeigh and BPDiff (%): 46.56
Accuracy for predictors LikeToWeigh and BPSyst1 (%): 46.56
Accuracy for predictors LikeToWeigh and Ethnicity (%): 49.17
Accuracy for predictors LikeToWeigh and LegLength (%): 42.28
Accuracy for predictors Overweight and Age (%): 42.40
Accuracy for predictors Overweight and ArmCirc (%): 68.76
Accuracy for predictors Overweight and ArmLength (%): 46.91
Accuracy for predictors Overweight and BPDias1 (%): 49.52
Accuracy for predictors Overweight and BPDiff (%): 47.15
Accuracy for predictors Overweight and BPSyst1 (%): 48.69
Accuracy for predictors Overweight and Ethnicity (%): 50.24
Accuracy for predictors Overweight and LegLength (%): 44.89
Accuracy for predictors Overweight and LikeToWeigh (%): 51.54
Accuracy for predictors Pulse and Age (%): 30.88
Accuracy for predictors Pulse and ArmCirc (%): 56.89
Accuracy for predictors Pulse and ArmLength (%): 30.52
Accuracy for predictors Pulse and BPDias1 (%): 29.57
Accuracy for predictors Pulse and BPDiff (%): 32.54
Accuracy for predictors Pulse and BPSyst1 (%): 32.66
Accuracy for predictors Pulse and Ethnicity (%): 36.22
Accuracy for predictors Pulse and LegLength (%): 31.12
Accuracy for predictors Pulse and LikeToWeigh (%): 47.03
Accuracy for predictors Pulse and Overweight (%): 48.22
Accuracy for predictors Sex and Age (%): 33.37
Accuracy for predictors Sex and ArmCirc (%): 68.88
Accuracy for predictors Sex and ArmLength (%): 34.80
Accuracy for predictors Sex and BPDias1 (%): 34.68
Accuracy for predictors Sex and BPDiff (%): 33.14
Accuracy for predictors Sex and BPSyst1 (%): 38.48
Accuracy for predictors Sex and Ethnicity (%): 36.94
Accuracy for predictors Sex and LegLength (%): 34.32
Accuracy for predictors Sex and LikeToWeigh (%): 49.17
Accuracy for predictors Sex and Overweight (%): 51.19
Accuracy for predictors Sex and Pulse (%): 34.56
Accuracy for predictors Subscapular and Age (%): 43.11
Accuracy for predictors Subscapular and ArmCirc (%): 63.78
Accuracy for predictors Subscapular and ArmLength (%): 42.40
Accuracy for predictors Subscapular and BPDias1 (%): 40.97
Accuracy for predictors Subscapular and BPDiff (%): 38.72
Accuracy for predictors Subscapular and BPSyst1 (%): 38.36
Accuracy for predictors Subscapular and Ethnicity (%): 42.40
Accuracy for predictors Subscapular and LegLength (%): 42.40
Accuracy for predictors Subscapular and LikeToWeigh (%): 43.59
Accuracy for predictors Subscapular and Overweight (%): 46.79
Accuracy for predictors Subscapular and Pulse (%): 39.67
Accuracy for predictors Subscapular and Sex (%): 46.08
Accuracy for predictors Triceps and Age (%): 38.00
Accuracy for predictors Triceps and ArmCirc (%): 65.44
Accuracy for predictors Triceps and ArmLength (%): 42.16
Accuracy for predictors Triceps and BPDias1 (%): 37.29
Accuracy for predictors Triceps and BPDiff (%): 39.79
Accuracy for predictors Triceps and BPSyst1 (%): 40.62
Accuracy for predictors Triceps and Ethnicity (%): 39.07
Accuracy for predictors Triceps and LegLength (%): 38.24
Accuracy for predictors Triceps and LikeToWeigh (%): 50.00
Accuracy for predictors Triceps and Overweight (%): 50.59
Accuracy for predictors Triceps and Pulse (%): 37.89
Accuracy for predictors Triceps and Sex (%): 48.46
Accuracy for predictors Triceps and Subscapular (%): 39.79
Accuracy for predictors Waist and Age (%): 64.01
Accuracy for predictors Waist and ArmCirc (%): 67.46
Accuracy for predictors Waist and ArmLength (%): 64.85
Accuracy for predictors Waist and BPDias1 (%): 62.00
Accuracy for predictors Waist and BPDiff (%): 61.05
Accuracy for predictors Waist and BPSyst1 (%): 59.38
Accuracy for predictors Waist and Ethnicity (%): 63.54
Accuracy for predictors Waist and LegLength (%): 64.13
Accuracy for predictors Waist and LikeToWeigh (%): 66.86
Accuracy for predictors Waist and Overweight (%): 65.20
Accuracy for predictors Waist and Pulse (%): 63.30
Accuracy for predictors Waist and Sex (%): 65.20
Accuracy for predictors Waist and Subscapular (%): 66.75
Accuracy for predictors Waist and Triceps (%): 66.51
```





	Predicted	Actual
4769	4	3
4770	4	4
4771	5	6
4772	6	6
4773	3	3
4774	3	3
4775	3	4
4776	4	4
4777	4	4
4778	3	4
4779	3	3
4780	4	4
4781	3	3



Category	BMI Class
1	Anorexic
2	Underweight
3	Optimal
4	Overweight
5	Obese
6	Morbidly Obese

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).

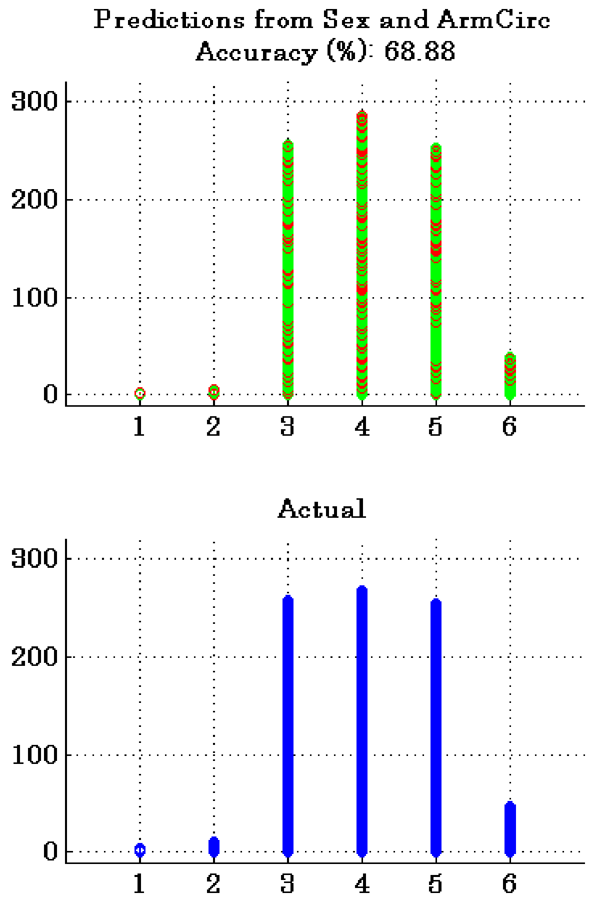
The MATLAB Profiler.

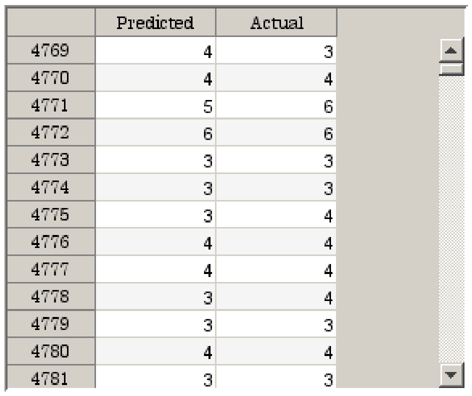
The MATLAB Profiler is intended to provide a more in-depth breakdown of the time spent executing code. It produces a fully hyperlinked HTML report which details the number of calls, the total time and the self time for each function. Using the profiler results can help to identify bottlenecks in code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

Recommended activity: open the profiler and profile the working version of findBestPredictors.

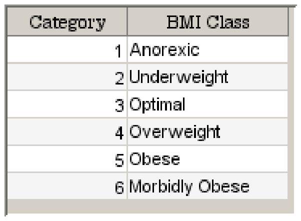
```
profile on
predictorRankings = F04_findBestPredictors_V3();
profile off
profile viewer

% 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.
```

```
Accuracy for predictors ArmCirc and Age (%): 59.62
Accuracy for predictors ArmLength and Age (%): 32.07
Accuracy for predictors ArmLength and ArmCirc (%): 65.20
Accuracy for predictors BPDias1 and Age (%): 33.37
Accuracy for predictors BPDias1 and ArmCirc (%): 58.79
Accuracy for predictors BPDias1 and ArmLength (%): 30.17
Accuracy for predictors BPDiff and Age (%): 32.54
Accuracy for predictors BPDiff and ArmCirc (%): 57.96
Accuracy for predictors BPDiff and ArmLength (%): 31.35
Accuracy for predictors BPDiff and BPDias1 (%): 31.47
Accuracy for predictors BPSyst1 and Age (%): 33.73
Accuracy for predictors BPSyst1 and ArmCirc (%): 55.82
Accuracy for predictors BPSyst1 and ArmLength (%): 34.92
Accuracy for predictors BPSyst1 and BPDias1 (%): 30.88
Accuracy for predictors BPSyst1 and BPDiff (%): 31.00
Accuracy for predictors Ethnicity and Age (%): 30.17
Accuracy for predictors Ethnicity and ArmCirc (%): 62.59
Accuracy for predictors Ethnicity and ArmLength (%): 37.65
Accuracy for predictors Ethnicity and BPDias1 (%): 36.22
Accuracy for predictors Ethnicity and BPDiff (%): 32.66
Accuracy for predictors Ethnicity and BPSyst1 (%): 35.99
Accuracy for predictors LegLength and Age (%): 32.66
Accuracy for predictors LegLength and ArmCirc (%): 64.01
Accuracy for predictors LegLength and ArmLength (%): 33.97
Accuracy for predictors LegLength and BPDias1 (%): 28.74
Accuracy for predictors LegLength and BPDiff (%): 27.43
Accuracy for predictors LegLength and BPSyst1 (%): 28.03
Accuracy for predictors LegLength and Ethnicity (%): 31.12
Accuracy for predictors LikeToWeigh and Age (%): 40.02
Accuracy for predictors LikeToWeigh and ArmCirc (%): 67.81
Accuracy for predictors LikeToWeigh and ArmLength (%): 45.49
Accuracy for predictors LikeToWeigh and BPDias1 (%): 49.52
Accuracy for predictors LikeToWeigh and BPDiff (%): 46.56
Accuracy for predictors LikeToWeigh and BPSyst1 (%): 46.56
Accuracy for predictors LikeToWeigh and Ethnicity (%): 49.17
Accuracy for predictors LikeToWeigh and LegLength (%): 42.28
Accuracy for predictors Overweight and Age (%): 42.40
Accuracy for predictors Overweight and ArmCirc (%): 68.76
Accuracy for predictors Overweight and ArmLength (%): 46.91
Accuracy for predictors Overweight and BPDias1 (%): 49.52
Accuracy for predictors Overweight and BPDiff (%): 47.15
Accuracy for predictors Overweight and BPSyst1 (%): 48.69
Accuracy for predictors Overweight and Ethnicity (%): 50.24
Accuracy for predictors Overweight and LegLength (%): 44.89
Accuracy for predictors Overweight and LikeToWeigh (%): 51.54
Accuracy for predictors Pulse and Age (%): 30.88
Accuracy for predictors Pulse and ArmCirc (%): 56.89
Accuracy for predictors Pulse and ArmLength (%): 30.52
Accuracy for predictors Pulse and BPDias1 (%): 29.57
Accuracy for predictors Pulse and BPDiff (%): 32.54
Accuracy for predictors Pulse and BPSyst1 (%): 32.66
Accuracy for predictors Pulse and Ethnicity (%): 36.22
Accuracy for predictors Pulse and LegLength (%): 31.12
Accuracy for predictors Pulse and LikeToWeigh (%): 47.03
Accuracy for predictors Pulse and Overweight (%): 48.22
Accuracy for predictors Sex and Age (%): 33.37
Accuracy for predictors Sex and ArmCirc (%): 68.88
Accuracy for predictors Sex and ArmLength (%): 34.80
Accuracy for predictors Sex and BPDias1 (%): 34.68
Accuracy for predictors Sex and BPDiff (%): 33.14
Accuracy for predictors Sex and BPSyst1 (%): 38.48
Accuracy for predictors Sex and Ethnicity (%): 36.94
Accuracy for predictors Sex and LegLength (%): 34.32
Accuracy for predictors Sex and LikeToWeigh (%): 49.17
Accuracy for predictors Sex and Overweight (%): 51.19
Accuracy for predictors Sex and Pulse (%): 34.56
Accuracy for predictors Subscapular and Age (%): 43.11
Accuracy for predictors Subscapular and ArmCirc (%): 63.78
Accuracy for predictors Subscapular and ArmLength (%): 42.40
Accuracy for predictors Subscapular and BPDias1 (%): 40.97
Accuracy for predictors Subscapular and BPDiff (%): 38.72
Accuracy for predictors Subscapular and BPSyst1 (%): 38.36
Accuracy for predictors Subscapular and Ethnicity (%): 42.40
Accuracy for predictors Subscapular and LegLength (%): 42.40
Accuracy for predictors Subscapular and LikeToWeigh (%): 43.59
Accuracy for predictors Subscapular and Overweight (%): 46.79
Accuracy for predictors Subscapular and Pulse (%): 39.67
Accuracy for predictors Subscapular and Sex (%): 46.08
Accuracy for predictors Triceps and Age (%): 38.00
Accuracy for predictors Triceps and ArmCirc (%): 65.44
Accuracy for predictors Triceps and ArmLength (%): 42.16
Accuracy for predictors Triceps and BPDias1 (%): 37.29
Accuracy for predictors Triceps and BPDiff (%): 39.79
Accuracy for predictors Triceps and BPSyst1 (%): 40.62
Accuracy for predictors Triceps and Ethnicity (%): 39.07
Accuracy for predictors Triceps and LegLength (%): 38.24
Accuracy for predictors Triceps and LikeToWeigh (%): 50.00
Accuracy for predictors Triceps and Overweight (%): 50.59
Accuracy for predictors Triceps and Pulse (%): 37.89
Accuracy for predictors Triceps and Sex (%): 48.46
Accuracy for predictors Triceps and Subscapular (%): 39.79
Accuracy for predictors Waist and Age (%): 64.01
Accuracy for predictors Waist and ArmCirc (%): 67.46
Accuracy for predictors Waist and ArmLength (%): 64.85
Accuracy for predictors Waist and BPDias1 (%): 62.00
Accuracy for predictors Waist and BPDiff (%): 61.05
Accuracy for predictors Waist and BPSyst1 (%): 59.38
Accuracy for predictors Waist and Ethnicity (%): 63.54
Accuracy for predictors Waist and LegLength (%): 64.13
Accuracy for predictors Waist and LikeToWeigh (%): 66.86
Accuracy for predictors Waist and Overweight (%): 65.20
Accuracy for predictors Waist and Pulse (%): 63.30
Accuracy for predictors Waist and Sex (%): 65.20
Accuracy for predictors Waist and Subscapular (%): 66.75
Accuracy for predictors Waist and Triceps (%): 66.51
```





	Predicted	Actual
4769	4	3
4770	4	4
4771	5	6
4772	6	6
4773	3	3
4774	3	3
4775	3	4
4776	4	4
4777	4	4
4778	3	4
4779	3	3
4780	4	4
4781	3	3



Category	BMI Class
1	Anorexic
2	Underweight
3	Optimal
4	Overweight
5	Obese
6	Morbidly Obese

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

% Recommended activity: vectorise the code in the first section of
% calculations by using standard mathematical operations. Next, 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.

Recommended activity: vectorise the code in the remaining sections of the S04_Vectorisation medical data calculation script using CELLFUN and 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)
```

```
    MaxPossibleArrayBytes: 9.5969e+09
    MemAvailableAllArrays: 9.5969e+09
            MemUsedMATLAB: 1.6609e+09
      VirtualAddressSpace: [1x1 struct]
             SystemMemory: [1x1 struct]
           PhysicalMemory: [1x1 struct]
```

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)
```

```
Memory available (GB):
    8.6786
Memory available (GB):
    8.6786
Memory available (GB):
    8.4098
```

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.

Recommended activity: open and explain the inPlaceExample function. See the reference file "inPlaceExample".

```
inPlaceExample()
```

```
Memory available after creating large array (GB):
    8.2415
Memory available inside the not in-place function (GB):
    8.0823
Memory available after calling the not in-place function (GB):
    8.2508
Memory available inside the in-place function (GB):
    8.2508
Memory available after calling the in-place function (GB):
    8.2508
```

Published with MATLAB® R2013b

Appendix A - Exercises.

Authors: Ken Deeley, ken.deeley@mathworks.co.uk Juan Martinez, juan.martinez@mathworks.co.uk

This appendix contains several hands-on exercises designed to provide practice with the concepts covered during each chapter of the course. Full worked solutions are available for each exercise. These materials are stored in the "Exercises" folder of the main course directory.

Contents

The MATLAB Language and Desktop Environment.

Exercise - Gas Prices Solution - Ex01_GasPrices.m

- Create a new script, and use the readtable function to import the data from gasprices.csv into a table in your Workspace. Hint: You will need to specify the Delimiter and Headerlines options – see the documentation for the readtable function.

- Extract the data for Japan from the table into a separate numeric variable, using the dot syntax (.). Compute the mean and standard deviation of the Japanese prices.

- Extract the data for Europe into a numeric array, and compute the mean European price for each year. Hint: check the documentation for the mean function.

- Compute the European return series from the prices using the following formula:

```
   R(t) = log( P(t+1)/P(t) )

Here, P(t) represents a single price series. There are four price
series in the European data.
```

- Compute the correlation coefficients of the four European return series. Hint: See the documentation for the corrcoef function. Display the resulting correlation matrix using the imagesc function.

The MATLAB Language and Desktop Environment.

Exercise - Australian Marriages Solution - Ex02_AusMarriages

- Create a new script, and use the readtable function to import the data from AusMarriages.dat into a table in your Workspace. Hint: You will need to specify the Delimiter option – see the documentation for the readtable function. This data file is tab-delimited ('\t').

- Insert an additional variable dn in the table containing the numerical representations of the dates. Hint: MATLAB uses serial date numbers to represent date and time information. Check the documentation for the datenum function (note that the date format for this data is dd/mm/yyyy).

- Plot the number of marriages as a function of time. Format the x-axis tick labels using the datetick function.
- Create a 1x25 row vector v of equal values 1/25. Hint: use ones.
- Use the conv function (with the 'same' option) and the vector from step 4 to smooth the marriage data.
- Overlay the smoothed data on the original plot, using a different colour. Hint: Use hold on.

Algorithm Design in MATLAB.

Exercise - Pacific Sea Surface Temperatures Solution - Ex03_SeaSurfaceTemperatures, SST_grid_sol

- Create a new script, and load the data from Ex03_SST.mat into the Workspace.
- Visualise the first set of temperature data (corresponding to the first column of the variable sst) using: a 2D scatter plot, specifying the colours using the temperature data; a 3D scatter plot, using black points as markers.
- Create equally-spaced vectors of points ranging from min(lon) to max(lon) and min(lat) to max(lat), respectively. Next, create grids lonGrid and latGrid of lattice points using meshgrid.
- Create a variable tempGrid by interpolating the first set of temperature data over the lattice of points created in the previous step. Hint: use griddata.
- Visualise the resulting interpolated values using the surf function.
- Write a function SST_grid taking the longitude, latitude and a set of temperature measurements as its three input arguments. The function should return lonGrid, latGrid and tempGrid as output arguments. Therefore, the first line of your function will be:

[lonGrid, latGrid, tempGrid] = SST_grid(lon, lat, seaTemp)

- In your script, invoke your function for each different set of sea surface temperature measurements (i.e. the different columns of the sst variable), and visualise the results in a 4x6 subplot array. Hint: use a for-loop and the subplot function.

Algorithm Design in MATLAB.

Exercise - House Prices to Median Earnings Solution - Ex04_PricesToWages, plotPrices_sol, houseUI_complete

- Create a new script, and load the data from Ex04_House.mat into the Workspace.
- Open and view the data table in the Variable Editor. Extract the region names from the table as a cell array of strings, by accessing the Properties metadata of the table. Hint: At the command line, enter >> pricesToWages.Properties
- Use the strcmp function and the cell array from the previous step to identify the row of the table containing the data for Manchester. Extract the data from this row, and plot this data as a function of year (note that the years range from 1997 to 2012).
- Write a function plotPrices with the following specifications:

```
plotPrices accepts a row number as its only input argument;
plotPrices plots the price information from the corresponding row of
the table as a function of year.

Note that since functions have their own private workspace, you will
need to load the required data inside the function.
```

- Bonus: open the houseUI function file and modify the popup menu callback so that the plot is updated when the user selects a region of interest.

Test and Verification of MATLAB Code.

Exercise - String Subset Replacement Solution - test_repblank_sol

- Open the script app_repblank and run the code. The objective of this exercise is to write unit tests for the repblank function. Open the repblank function and read the help to understand the function's specifications and requirements.
- Open the main test function test_repblank. This main test function currently has four local test function stubs.
- Write code in the test_OneBlank local test function to verify that repblank exhibits correct behaviour when the input string contains one blank character (i.e. one space). Hint: use verifyEqual.
- Write code in the test_ManyBlank local test function to verify that repblank exhibits correct behaviour when the input string contains multiple consecutive blank values.
- Write code in the test_FirstBlank local test function to verify that repblank exhibits correct behaviour when the input string contains leading spaces.
- Write code in the test_LastBlank local test function to verify that repblank exhibits correct behaviour when the input string contains trailing spaces.
- Run all tests using the runtests command and ensure that all tests pass.
- Bonus: write another local test function to verify that repblank throws an error when called with a string consisting entirely of blanks. Hint: use verifyError with an appropriate error identifier. Ensure that all tests still pass as before.

Test and Verification of MATLAB Code.

Exercise - Levenshtein String Distances Solution - test_strdist_sol

- Open the script app_strdist and run the code. The objective of this exercise is to write unit tests for the strdist function. Open the strdist function and read the help to understand the function's specifications and requirements.
- Open the main test function test_strdist. This main test function currently has a setupOnce method, a teardownOnce method and three local test function stubs.
- Write code in the setupOnce method which adds the test data folder STRDIST_Test_Data to the path, and then initialises the TestData structure contained in the testCase object by loading the MAT-file words.mat.
- Write code in the teardownOnce method which removes the test data folder STRDIST_Test_Data from the path.
- Write code in the test_OneWord local test function to verify that strdist exhibits correct behaviour when called with only the first word from the words.mat MAT-file. (In this case, the expected answer is 0.)
- Write code in the test_ThreeWords local test function to verify that strdist exhibits correct behaviour when called with the first three words from the words.mat MAT-file. (In this case, the expected answer is the row vector [7, 5, 7].)
- Write code in the test_DimensionsOutput local test function to verify that strdist returns a solution of the correct dimensions when called with the first four words from the words.mat MAT-file. (In this case, the expected answer is the row vector [1, 6].)

Test and Verification of MATLAB Code.

Exercise - Counting Word Frequencies Solution - test_getwords_sol

- Open the script app_getwords and run the code. The objective of this exercise is to write unit tests for the getwords function. Open the getwords function and read the help to understand the function's specifications and requirements.

- Open the main test function test_getwords. This main test function currently has a setupOnce method, a teardownOnce method and three local test function stubs.
- Write code in the setupOnce method which adds the test data folder Literature to the path.
- Write code in the teardownOnce method which removes the test data folder Literature from the path.
- Write code in the test_IncorrectFilename local test function to verify that getwords exhibits correct behaviour when called with a non-existent file. Hint: use verifyError. In this case, the expected error ID is getwords:noFile.
- Write code in the test_AllWords local test function to verify that the first output of getwords has the correct size when the function is called on the file 'sherlock_holmes.txt'. In this case, the expected size of W is [207, 1]. Hint: use verifySize.
- Write code in the test_MaxFreq local test function to verify that getwords correctly returns the frequency of the most common word in 'sherlock_holmes.txt'. The most common word in 'sherlock_holmes.txt' occurs with frequency 10. Hint: use verifyEqual.

Debugging and Improving Performance.

Exercise - Debugging and Improving Existing Code Solution - analyzerEx_sol, analyzerEx_sol_vectorised

- In this exercise, you will debug an existing code file. Open the function analyzerEx in the Editor and inspect the Code Analyzer warning messages.
- Fix each of the Code Analyzer warnings, and call the function to ensure that the results are as expected.
- Bonus: Improve the code by vectorising the double for-loop. Hint: look up the repmat function, or use matrix multiplication or the cumsum function.

Published with MATLAB® R2013b