Working with a dataset

Peter Lamb

Week 7

In the final week of the module we will bring together most of the concepts covered so far to complete the data processing and analysis for a pilot study.

Study info

A well documented study should have good file and folder organisation, as we've already covered, but should also be documented with a README file. For the pilot study covered here, refer to the README file in the documents folder to clarify details of the dataset, such as the file naming convention, how the data were collected, orientation of coordinate systems (important in biomechanics), and the variable names and corresponding columns. We'll be looking at computing X-Factor variables from raw data to reuse our processing steps from last week, but we will have to build in a few more steps to handle working with multiple files and bring all the data together to get an initial look at the results.

The following sections will lay out processing and analysis scripts to achieve our goals of reproducible research introduced last week.

Overview

If we were start by writing all the comments describing the steps required in our script, the main sections would be:

- Clear the workspace and close figures

- Set up files, directories and filenames
- Initialise storage variables to be built up
- Loop through files and perform various tasks
  - Read in data
  - Calculate variables
  - Extract metadata
  - Add to the data to the storage variables
- Visualise data
- Analyse data
- Save or export data

These steps are all incorporated in the scripts 'step01_process_xf_data.m' and 'step02_analyse_xf_data.m' in the 'code' folder.

Data processing

Clear workspace and close figures

It's usually a good idea to clear your workspace as the first step in your script so that the variables in your workspace are only those from the data being read in or calculated afterwords. Remember functions only have access to the information provided as input, but scripts have access to the entire Workspace – so best to clear the Workspace as a first step.

```
clear;
```

You will also probably want to close any existing figures, to ensure plotting is done on the correct axes (figure windows) and so that you don't slowly build up loads of figure windows, which can really start to slow down MATLAB.

```
close all;
```

Set up files, directories and filenames

Our raw data are stored in 'data/raw'. We will use the **dir** function identify the files in the 'raw' folder.

```
txtFiles = dir(fullfile('data', 'raw', '*.txt'));
```

We've passed the **dir** function one input, the relative path from the **Current Folder** to the data files. Note the use of the wildcard character `'*'`, which tells MATLAB any file in the 'raw' folder ending with '.txt'. The files all end in .txt, but this is a good way to exclude hidden files (which vary between operating systems). Try running the command without the third input to **fullfile** – I get two additional files that I don't want to open for analysis.

The **txtFiles** variable is a struct with one row for each file found in the 'raw' folder and several fields describing the files. We're interested in the **name** field so that we can use each file's filename to pass to the **csvread** function. To review the syntax for structs, if we want to get the filename of the fifth row in the struct we type:

```
>> txtFiles(5).name

ans =

    'P001_T03_C1.txt'
```

Initialise variables

We know that we'll be reading in 24 trials.

```
>> length(txtFiles)

ans =

    24
```

From each trial we will extract the X-Factor value at the start of the downswing (call the variable **XF**) and the difference between the maximum X-Factor and X-Factor at the start of the downswing (known as *X-Factor Stretch*, we'll call the variable **XFS**). When dealing with larger datasets it can be much more efficient for an empty variable to be created first and then filled rather than increased

as you go along[1]. The **nan** function is useful when creating blank variables as the 'NaN' (Not A Number) value is often used to represent missing values. The following command will create a matrix of NaN values that is the correct size.

```
nFiles = length(txtFiles); % number of files

XF = nan(nFiles,1); % Initialised X-Factor variable
```

Using the Debugger

As we go through the script, try using the **debugger** to follow along. The debugger lets you put **breakpoints** at certain lines and gives you control to step along one line at a time, continue to the end of the section, until the next breakpoint or the end of the script. To insert a break point, simply click on the short horizontal bar next to the line number – a red filled circle should appear. Now if you run the script it will stop at the red circle, leaving you in *debug mode*. In debug mode you will notice there is a K on your command prompt

```
K>>
```

You can now see the values of all the steps in your script up until this point (the variables in your Workspace). You'll also see a green arrow in the **Editor** showing how far you've progressed in the script. Click **Step** to go to the next line or **Continue** to proceed until the next breakpoint if there is one, or to the end of the script if there is not. Being comfortable with the debugger is very useful both for fixing errors and just following along to understand what the script is doing.

Loop through files

Since we know what's in the 'raw' folder, through the use of the **dir** command, we can set up a `for` loop to cycle through each file.

[1] This is related to how a computer stores the data in its on-board memory. Imagine that someone asks you to store a box of their things. You rent a room just big enough and move it there. Later they ask you to store more stuff and so you have to rent a bigger room and move the old and new stuff into the new room. They keep giving you more things and you have to keep shifting it all to a new larger location each time. Now imagine that they told you at the beginning that they want to eventually store enough stuff to fill a small warehouse. You rent the warehouse and begin by moving the initial small amount into one corner. As the new items arrive you just put them next to the old stuff, and more next to that and so on. Much more efficient!

```
%% Loop through each file in the data folder
for i = 1 : nFiles

    % Determine filename for "i-th" file in the data
    % folder
    fileName = fullfile('data', 'raw', txtFiles(i).name);

    % Read in current file using csvread
    gD = csvread(fileName,9,0);

    % X-Factor calculations go here...
end
```

From this point in the loop we can calculate the X-Factor variables using (roughly) the same process as last week.

```
%% Calculate dependent variables
% Calculate the X-Factor throughout the swing
xfVector = gD(:,17) - gD(:,14);

% Determine the frame number of the start of the downswing
[minVal,minIndex] = min(gD(:,14));

% Extract the X-Factor value at start of downswing
xf = xfVector(minIndex);

% Calculate the maximum X-Factor (actually minimum
% because angles are negative in the backswing).
[xfMin,xfMinInd] = min(xfVector);

% Calculate the difference between max X-Factor and X-
% Factor at start of downswing
xfs = xf - xfMin;
```

Extract metadata

Metadata are data describing our data; for this dataset the metadata can be extracted from the filenames. Note the file naming conventions from the README

file, we could extract the Participant ID, the experimental condition and the trial number. In the processing script we will just extract the condition.

We haven't done much with strings yet, so we'll have to introduce a few new concepts in this section. Because our file naming convention is consistent we'll break the file name into parts separated by the underscore character. The **strsplit** function takes a string as the first input and a character as a second input, here we pass it the filename and the underscore character. The function returns a 1×3 cell array of strings:

```
>> nameParts = strsplit(fileName, '_')

nameParts =

  1x3 cell array

    {'data/raw/P001'}    {'T01'}    {'C1.txt'}
```

As you can see, the information about the condition of the trial is in the third column of the cell array, 'C1'. Within the third cell we're interested in extracting the 1, as this trial is from condition 1 (easy swing). We can get the 1 by extracting the second character in the string `'C1.txt'` and in case we get double digit conditions, just to be safe, we'll extract until we reach the fullstop, which is at position end-4.

```
% Extract the third cell
condCell = nameParts{1,3};

% Skip the 'C' character at the start and '.txt' at the
% end
cond = condCell(2:end-4);
```

Final step, the data type of the 1 we extracted is a string character rather than numeric, so we'll convert it to numeric with the **str2double** function. Note that the final three lines extracting the string information could be put on one line, I've just separated them here so each operation is clear.

```
% Convert the character to numeric
cond = str2double(cond);
```

Add to the dataset

Recall that we initialised a few variables before our `for` loop using the **nan** function. If we calculated the X-Factor variables inside the loop without adding the values to a *storage* variable, the values could just be overwritten each time through the loop. So before the loop restarts we need to store the variable values in our storage variables.

```
% Assign current value to row i of XF
XF(i) = xf;

% Assign current value to row i of XFS
XFS(i) = xfs;

% Assign the current condition to row i of the COND
% variable
COND(i) = cond;
```

We store the values in row *i* of the storage variables so the values are not overwritten on the next loop iteration.

That's the end of our loop and the script terminates by saving the **XF**, **XFS** and **COND** variables to a .mat file.

```
%% Save data to file
save(fullfile('data','mat','01_processed_xf_data.mat'),...
    'XF','XFS','COND')
```

Data analysis

I like to separate the processing and analysis into different scripts, mainly because the processing steps can take quite a while when using large datasets – often several minutes, which is long time to wait each time if you're just tweaking a plot command. Open the second script 'step02_analyse_xf_data.m' to follow the remaining analysis steps.

To begin we clear the workspace and load the dataset created in the processing script.

```
% Clear workspace and close open figures
clear;
```

```
close all;

%% Set up directories, files and variables
load(fullfile('data','mat','01_processed_xf_data.mat'))
```

Data visualisation

We've already seen boxplots from the ANOVA analysis in the Statistics section of the module. Here we use the **boxplot** function making use of a few extra options. Three Name-Value pairs are given as input, a full list of the options can be seen in the documentation. We turn the 'Notch' option on, as non-overlapping notches give a good general idea of whether there is a significant difference between the conditions. The 'Labels' option is give a two element cell of strings for the labels to use for each group: 'Easy' and 'Hard'. We've also specified the width of the boxes, just to make it look nice.

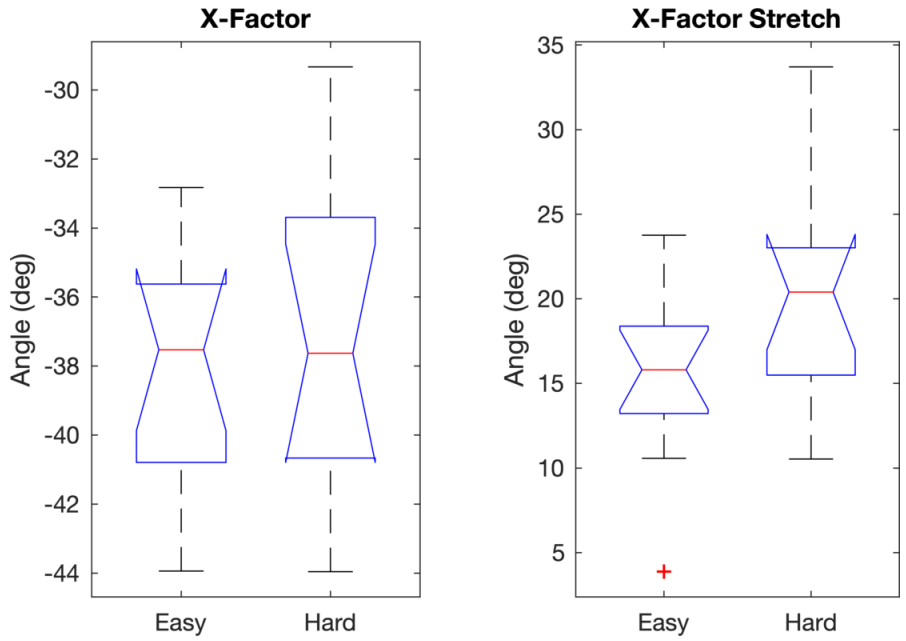

Figure 1: Boxplot showing the distribution X-Factor and X-Factor Stretch values for each swing effort condition.

Since we're working with a pilot dataset we can't make any conclusive statements, but from the boxplots it appears that the X-Factor Stretch is more related to swing effort than X-Factor at the start of the downswing.

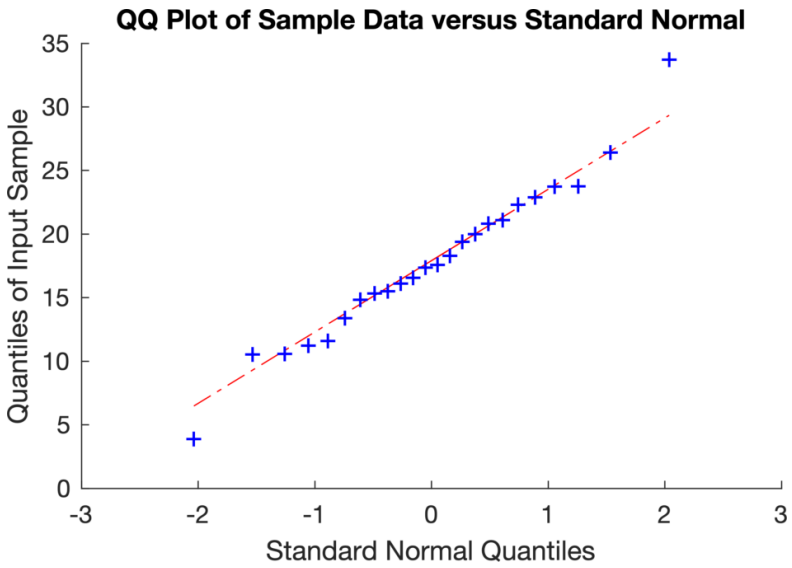

Figure 2: Quantiles from measured data versus matching quantiles sampled from normal distribution.

Preliminary analysis

There are many ways to test some of the assumptions underlying statistical tests. In this section we'll check that the data are approximately normally distributed with a *q-q plot*. A q-q plot plots the observed data from their respective quantiles in the dataset versus matching quantiles from a normal distribution.

Figure 2 shows there are a couple outliers at the ends of the distribution, but it looks fine for a small dataset.

We also need to make a correction to our alpha threshold (0.05 is common, as in $p < .05$) because we are making two comparisons. Remember the p-value gives you probably of getting group differences as, or more, extreme than the differences you got if the groups were sampled from the same distribution. So with an alpha of 0.05, there's a 5% chance your group difference is random chance. When you increase the number of variables being statistically compared the alpha value loses its meaning. A fairly common method for adjusting alpha for multiple comparisons is called the Bonferroni correction, which is fairly conservative and quite easy to implement. Most statistical software packages have an option for the type of adjustment, but the Bonferroni is simply the desired alpha divided by the number of comparisons, which is easy enough to do on our own.

```
% Since we have two comparisons, adjust alpha to 0.05/2
```

```
alpha = 0.05/2;
```

Now that we're happy with the roughly normal distribution of our data and our alpha has been adjusted, we run a paired-sample *t*-test to compare the means of the swing effort groups.

```
% Conduct two-sample t-test on XFS
[h_xfs,p_xfs,ci_xfs,stats_xfs] = ttest(XFS(COND==1),...
    XFS(COND==2), 'Alpha', alpha);

>> p_xfs

p_xfs =

   1.5054e-04
```

Confirming what we saw in the boxplot, X-Factor Stretch looks to be related to swing effort, $t(11) = -5.64$, $p < .001$, 95% CI $[-6.24, -2.31]$. This finding, which is from a pilot study looks promising, but we should be very cautious about over-generalising from the results of four participants. Our preliminary results should be used to focus our analysis and sampling for a full study.

Sample size estimation

You may have already covered the importance of recruiting a sample of participants that adequately powers your statistical analysis. There are many ways to perform sample size calculations; all involve the specification of the test and either the effect size you want to be able to detect, or the expected mean and standard deviation of the samples. You also need to specify the desired power, i.e. power of 0.9 means you have a 90% of rejecting the null hypothesis when it actually should be rejected. Generally, if you're trying to detect a small change you need more participants than if the expected change were larger. Increased variability also increases the size of the required sample.

We'll use the built-in function **sampsizepwr** and follow the documentation to estimate our required sample.

```
% Mean of sample 1
mu1 = mean(XFS(COND==1));

% Standard deviation of sample 1
```

```
sigma1 = std(XFS(COND==1));

% Mean of sample 2
mu2 = mean(XFS(COND==2));

% Sample size calculation
sampleSize = sampsizepwr('t', [mu1 sigma1], mu2 , 0.9);
```

Looks like we should aim for 19 participants in the full study if we:

- want to use a paired-sample *t*-test,
- expect about a 4° difference in X-Factor Stretch comparing easy to hard effort swings,
- want power of 0.9.

Summary

In these two scripts, we've put together code that runs independent of the user and operating system, assuming our project folder and file structure. The user will be required to add the code folder to the MATLAB search path on first use in the session, but after that no user interaction is required to run the analysis: it is completely reproducible and automatic – the goal of this module!