Lecture 1: MATLAB - advanced use cases

Data handling and analysis

Juha Kuortti and Heikki Apiola

February 10, 2018

Aalto University
juha.kuortti@aalto.fi

Importing and exporting data: basics

Creating and accessing files `save,load`

Variables are erased from memory after quitting Matlab (`>>quit` or `>> exit`).

- The command `>>save` saves all workspace variables into the file `matlab.mat` in the current directory.
- `>>save A B` saves just `A` and `B`.
- `>>save myfile A B C*` saves `A,B` and all variables starting with `C` into `myfile.mat`. **Note:** If you happen to have a variable called `myfile` in your workspace, then this variable together with the above variables will be stored in `matlab.mat (:-))`.

Files: Loading from a .mat-file

- `>>load` reads the file `matlab.mat` (in current directory or on matlab path) and loads all variables in the workspace, i.e. restores the state of the workspace after the corresponding `save`-command.
- `>>load myfile` does the same with `myfile.mat`.
- `>>load myfile A B` loads just variables `A,B`.

Note: These `.mat`-files are in Matlab's internal format. The next slide treats `ASCII`-file handling.

Important: `save` and `load` can't be used to save your session. Usually, much more important than saving variables, is saving the commands that created those variables, i.e. saving your session. For that you need scripts and `.m` - files.

ASCII Files, `load textmat.dat`

- To create user-readable files, append the flag `-ascii` to the end of a save command.
- **Note:** In this case, MATLAB does not append any extension to the file name, so you may want to add an extension such as `.txt` or `.dat` .

Example: Create a text-file `textmat.dat` outside Matlab.

textmat.dat:

```
1 2 3 4
5 6 7 8
```

```
>> load -ascii textmat.dat
>> % You can omit -ascii here
>> textmat
textmat =
     1     2     3     4
     5     6     7     8
```

Numerica data in a text file

If you have straight up numerical data in an array, with no missing values, you can use simply use `load`.

However, if there is something missing, e.g. remove a number from `textmat.dat`, or it contains something else than just numerical data (headers, for example) `load` will give you an error.

Importing data: advanced

*read-functions

There are various functions for importing data (`lookfor read`). For our basic examples we'll use `csvread` and `dlmread`.

Both are designed to read delimited textfiles, `csvread` if the delimiter is comma, and `dlmread` for a general delimiter.

The issue with both is that they can only read *numerical data*. If your file contains something else — if your file contains something else, you'll need to use the options to exclude the offending entries.

Exercise

Download the files `gasprices.csv` and `formants.csv`.

- Try reading in the numerical data of `gasPrices.csv` using function `csvread`. The calling convention is as follows `csvread('fileToRead',startRow,StartCol)` — look at the file to decide good place to start reading.
- Then try reading `formants.csv`. The delimeter is now semicolon, so `csvread` won't work. Use help page to work out how to make `dlmread` work.

The GUI-way

Whenever you have a file containing data you wish to import, try dragging and dropping the file onto the command window. If the file is recognised by MATLAB as importable, an import wizard will appear giving you a plethora of options to import the data.

One final read function

MATLAB has a data type that can contain various types of entries. Table datatype is a fairly recent addition, and is specifically created for the purposes of data-analysis.

You can use function `readtable` to read a csv-file to a table.

Demonstration of table datatype

- Read in file `electricity.csv` using `readtable`.
- Separate the variables from descriptors
- Separate the dates from data.
- Extract the numerical data.
- Plot the data

The Case of Missing Data

How to handle missing data

- Leave the data as is and ignore any NaN elements when performing calculations. Maintains the integrity of the data but can be difficult to implement for involved calculations.
- Remove all NaN elements from the data. Simple but, to keep observations aligned, must remove entire rows of the matrix where any data is missing, resulting in a loss of valid data.
- Replace all NaN elements in the data. Keeps the data aligned and makes further computation straightforward, but modifies the data to include values that were not actually measured or observed.

Case 1: Ignoring NaNs

When performing any operation containing a NaN, the result will always be either NaN, or false, depending on context.

Especially:

```
A = [1,2,3,4,3,1,3,4,5,nan,13,5];
avg = mean(A) % will be nan
```

To avoid this, either use functions called `nan*` (e.g. `nanmean`), that automatically omit nans, or look for appropriate flags in the documentation:

```
avg = mean(A,'omitnan')
```

Case 2: Hunt'em down

We can also identify all corrupted entries, and just delete them from our data entirely. This can lead to huge data loss, but uncertainty involve in guessing data will be less.

You can locate the corrupted entries using either `isnan` or `ismissing` functions. `isnan` deals specifically with NaN values, while `ismissing` is more general, allowing you to specify the erroneous values.

Case 2 continued

```
data = readtable('electricity.csv');
idx = ismissing(data); % logical indices of ...
    missing values
idxR = any(idx,2); % look for all the rows that ...
    are missing data
data(:,idxR) = []; % delete all the rows that ...
    have data missing
```

Case 3: Try and Guess

Sometimes data is too spotty to remove all corrupted entries — sometimes we know that data is by nature continous. In such cases we can make educated guesses as to what the data would be, and obtain more data points. This is called *interpolation*.

From previous course, we remember the `interp*`-functions, that did the interpolation. For data science, there is a useful helper-function called `fillmissing`, that is much more forgiving about the sampling points (e.g. you can omit the eval points, or you can use dates etc.)

Case 3 continued

`fillmissing` has a lot of options, as one might imagine. Help page is your friend, as usual.

```
data = readtable('electricity.csv');
usage = data{:,2:end}; % extract the numerical data
dates = data.Date; % extract the date
intUsage1 = ...
    fillmissing(usage,'linear','samplepoints',dates);
intUsage2 = fillmissing(usage,'nearest'); % ...
    assumes even sampling
```

Exercise

- Read in the file `hurricanes2.csv` using `readtable`. You'll need to do some sleuthing in the documentation to find out how to exclude the comment lines.
- Remove all the datapoints that have Country listed as `N/A`.
- Do a scatter plot of windspeed plotted against air pressure.

Additionally, if time allows:

- Read in the dataset `hurricanes3.csv`.
- The country identifier is `N/A` if observation has happened over sea. Replace all `N/A` entries with identifier "Sea" and all others with "Land".
- Scatterplot the windspeeds and pressures of sea observations in blue and land observations in red.

Smoothing data

Why smoother is usually better

Oftentimes the data is too noisy to discover possible underlying trends. Smoothing is a technique similar to interpolation in technique, but rather than trying to create new points of data, we are trying to exclude the possible noise components in the data.

Needless to say - since we are actually modifying the underlying data and observations, smoothing should not be done without justification.

Underfitting a polynomial I

One of the usual ways (depending on the data, of course) to do smoothing is to *underfit* a polynomial to it. An N points of data allows for a $N-1$ degree polynomial to be fitted; however it allows for all the lower degrees as well. It will mean that fit won't pass through all the datapoints, but sometimes a better model can be produced.

Underfitting a polynomial II

Let's try to fit a polynomial to US census data.

```
% create time vectors -- dense and sparse
t = 1900:10:1990;
tt = 1900:1:2010;
pop = [76 92 106 122 132 150 179 203 226 248];
% for show, try the maximum degree polynomial
P = polyfit(t,pop,length(pop)-1);
plot(tt,polyval(P,tt));
% It fails -- as is expected
P = polyfit(t,pop,3);
plot(tt,polyval(P,tt));
```

If your data is highly sinusoidal

If your data contains clear frequency components that you would like to keep, and the noise is zero-averaged (“white”), then it may be possible to identify the noise components in frequency domain. When you’re using the signal strength to identify the noise, it is called *noise gating*.

Note that mathematically this is a crude operation, with possibility of causing odd artefacts. Look to the Signal Processing Toolbox for more sophisticated methods in frequency domain.

Noise gating — example

```
% Create and visualize the data
t = linspace(0,0.5*pi,256);
carrier = sin(2*pi*t) + sin(6*2*pi*t);
plot(t,carrier)
%% add some noise
noisySignal = carrier + 2*rand(size(carrier))-1;
plot(t,carrier,t,noisySignal)
%% Go to frequency spectrum
X = fft(carrier);
plot(abs(X))
Xnoisy = fft(noisySignal);
plot(abs(Xnoisy))
%% Identify the noise component, and gate it out
Xnoisy(abs(Xnoisy)<30) = 0;
%% Go back to time domain
xnew = real(ifft(Xnoisy));
plot(t,carrier,t,xnew)
```

Exercise

Let's do a third way of smoothing — a moving average.

- First, read in the data in `electricity.csv`.
- The table has some spots missing. Fill them in using linear interpolation (go back a few slides for hints).
- Extract from the dataset the variable `total` and plot it.
- The data is taken on the first of every month, so therefore it is affected by seasonal changes — i.e. cooling during the summer, heating during winter (or something else). In order to get clear view of how energy demand behaves, we will smooth the data using *moving average*.
- Read the documentation of `movmean` and smooth the data and plot the result. You'll also need to decide a proper window length, but remember that data is monthly and changes mostly seasonal.

Developing a model

What's a model?

A topic so wide entire books have been written about it, and as such much too wide for us to discuss.

For our purposes we treat the model as a function f that is dependent on the variables $\mathbf{x}$ and parameters $\boldsymbol{\lambda}$; we judge the quality of the model by comparing to the data $\boldsymbol{y}$.

We take a look at few examples of finding the parameters below.

Curve fitting - Example

The very basic example — fitting a trend line on a dataset.

```
    % create some data
x = 5:21;
y = 3.5*x+4 + 4*randn(size(x))+5;
plot(x,y,'ro')
%% the hardcore mathy way
V = [x',ones(size(x'))];
coeff = V\y';
xfit = linspace(x(1),x(end),256);
yfit = coeff(1)*xfit + coeff(2);
plot(x,y,'ro',xfit,yfit)
%% The more consistent way
p = polyfit(x,y,1);
plot(x,y,'ro',xfit,polyval(p,xfit))
```

Nonlinear fitting

```
clear; close all;
x = 20:65;
y = [0  0  0  1  2  3  15  65  71  80  55  48  46  26  25  25  16  9 ...
    18 ...
 8 8  6  4  6  5  5  2  6  4  2  0  0  1  1  1  0  1  1  0  0  0  0  0  1 ...
     0  0];
f = @(x,beta)(beta(1)*(x-beta(3)).^2 .* ...
    exp(-beta(2)*(x-beta(3)).^2));
fobj= @(lam,x,y)(norm (f(x,lam)-y).^2);
beta0 = [2 0.01 15];
[beta fval eflag] = fminsearch(fobj,beta0,[],x,y);
bar(x,y,'c');
hold on;
plot(x,f(x,beta),'r');
xlabel('Age of Ph.D'); ylabel('Number of Ph.Ds');
hold off
```