Importing and Exporting Data

Before you start analyzing data, you have to import them into Matlab. Data come in all kinds of formats: text (ASCII) files, spreadsheets, comma separated value files, Excel spreadsheets, XML files, or proprietary text and binary formats generated by specific recording software. Matlab offers many functions to simplify reading from such external data files. You will learn about the most important ones below.

Importing data is obviously critical, as any mistake you make at this stage will contaminate all analyses that follow. The difficulty of this stage depends to a large extent on the output format of the recording software. Some packages generate rigorously specified output data, others are more "flexible" and the format changes depending on recordings settings, or other parameters. The latter kind are very difficult to deal with. Usually, however, this is outside your control and you simply have to work with the files you get. It's a good practice to test your assumptions about the file format repeatedly throughout your import routines and generate warnings to the user (i.e. yourself) if those assumptions turn out to be false.The goal of these sanity checks is not only to protect yourself against unforeseen circumstances, but also to changes in the data format.

At the end of an analysis you probably want to save your results for later use in Matlab. There are a number of options for doing so; this is discussed in the second part of this handout. Finally, you may want to export some of your results for further analysis in specialized analysis packages, or to share with colleagues who do not use Matlab. This is discussed in the final section of this handout.

1.1. Importing Data

Most data anlysis projects start by reading the data files generated by your recording software. If your software generates an output file in a standard format then you're in luck. Most likely Matlab will either have a built-in function to import those data, someone may have written a routine and posted it on the Mathworks File Exchange, or the vendor of your software may provide import routines. The table below lists a few common generic data files and the corresponding Matlab functions used to import them. To see the full list, search for 'Supported File Formats' in the Matlab documentation.

Format	Function
HDF 5	h5read
DICOM	dicomread (Image Processing Toolbox)
Analyze	analyze75read
XML	xmlread
BMP, GIF, JPEG, TIFF, PNG	imread
WAVE, FLAC, MP3, MPEG-4	audioread
AVI, QuickTime Mpeg-4 WMV	VideoReader
Microsoft Excel	xlsread

For some formats, however, no import routines exist, or they are far from optimal and contain bugs. It can be quite a challenge to read data in a fast and reusable way.

Files come in two main variants. The first are text or ASCII files, the second are called binary files. The next two sections discuss how to import data from these kinds of files.

1.1.1. Reading Text Files

A text file contains text and numbers in a format that any text editor (e.g. the Matlab editor, but also other programs such as Notepad or Word) can read and display. The main advantage of these text files is that they are readable, so it is easy to inspect the data visually and compare the results of an import routine that you write with the values shown in the editor. The disadvantage of text files is that they can take up a lot of space on a hard disk, and they are slower to read than binary files (see below).

Many programs generate ASCII or text files as output. If the data in the file is space-separated *and* the number of elements on each row is the same (i.e. it is a matrix) you can use the `load` command to read the file.

```
load -ascii filename
```

A slightly more flexible function is `importdata` this will try to guess the format of your file on the basis of the file extension (i.e ascii text for .txt files, or jpeg compressed images for .jpg) and then read the data. For text files, importdata can only handle files that contain simple rectangular tables (optionally with header rows), such as those that could be exported from a spreadsheet program.

But, if the file contains rows of different length, or if it contains not just numbers but also some text comments, then you will have to write some code to import the data.

First you open the file:

```
[fileHandle,msg] = fopen('filename')
```

The `fopen` function returns a file handle, which is a bit like a bookmark; it points to the location in the file that Matlab will read next. When you first open a file, the handle points to the start of the file. If Matlab could not open the file, the `fileHandle` will equal -1. You should always check this value to make sure the file was actually opened. If it is not, the `msg` output argument will tell you what went wrong. Once you're done with the file, don't forget to close it with `fclose(fileHandle)`. Leavin a file open can cause problems later on when you try to read from or write to the file in Matlab or from another application. If you have trouble opening, moving, or deleting a file after opening it from Matlab, then you may have forgotten to call fclose on the file handle. Sometimes this happens because your read(or write) function crashed sometime after you opened the file handle, but before you closed it again. To avoid this situation, it is a good practice to add `try-catch` statements to file reading/writing functions. If you forgot to include such error handling, and no longer have access to the file handle but still need to close a file, you can use `fclose all` to close all open file handles.

Depending on the structure of the file you now have many options to read the data. If the structure of the file is very rigorous (every line has the same kind of data; like a table or a spreadsheet), then the `textscan` function is your best bet.

You provide the textscan function with information on the (fixed) format of the file. For instance, consider a file in which each file contains a date (%D), specifying the date of an experiment, a single character (%c) specifying the name of a condition, and 5 numbers (%f) that represent the number of spikes recorded in 5 trials. The format of such a line would be specified as

```
formatString = '%D%c%f%f%f%f%f'
```

With that format string, `textscan` can read an entire file with just one line

```
[data] = textscan(fileHandle,formatString)
```

All of the dates, one per line, will be in data{1}, all of the condition names in data{2}, etc. The textscan function has a number of optional arguments that allow you to deal with comments in the file (e.g. you can instruct textscan to ignore every line starting with '//' by specifying `'CommentStyle','//'`). Another useful feature is that textscan can deal with missing values. For instance, if the example file had only 4 spike counts for one of the lines, then `textscan` would replace the missing value with a NaN.

The main problem is that any mismatch between the format specification and the actual file content will often generate very strange outputs. Essentially what happens is that `textscan` believes it has found a match with the format string, but due to an error in the format string, or an irregularity in the file, one match is wrong. All subsequent matches, however, are also wrong because the file contents that `textsscan` reads is out of sync with the format string. In any case, it is very difficult to debug incorrect results, and you may find yourself randomly replacing elements of the format string. That is unlikely to result in a robust file reading solution. The same issues apply to fscanf, a close relative of textscanf. The main difference is that the format strings used for fscanf are vectorized. This means that a single `'%f'` can represent a single floating point number or a whole list of numbers. That can be useful if you know that a file contains numbers, but not how many. In general, however, the scan routines are appropriate for files that look like spreadsheets or tables, but for more complex files a sequential/manual file reading routine is more appropriate.

The `fgets` function is used when the format of a file changes line by line. A typical example is a file in which each line starts with a tag or a parameter name and the content of the rest of that line depends on the tag. For instance, a file that can contain comments (lines starting with '%'), identifiers of an electrode (lines starting with the Electrode tag), information on the impedance of the previously identified electrode ('Z') and the times at which spikes occurred on the electrode (lines starting with the SpikeTime tag)

```
% This file was recorded on 10-Oct-2014
Electrode: 1
Z: 1.4
SpikeTime: 1 10 23 100 123
Electrode: 2
Z: 1.2
SpikeTime: 2 100 201 204
```

The code below reads this file and stores the information in a struct array with one element per electrode.

```
clear data;
fileHandle = fopen('filename'); % Open the file
while true
   line = fgets(fileHandle);
   if line(1)=='%';continue;end % Skip comment lines
   if line==-1;break;end % End of File.

   tagValue = strsplit(line,':');
   tag = tagValue{1};
   value = tagValue{2};
   switch upper(tag)
      case 'ELECTRODE'
         % Found the electrode tag. All information after this will refer
         % to this electrode.
         currentElectrode = str2double(value);
      case 'Z'
         % Found impedance measure, store it in a struct array.
         data(currentElectrode).Z = str2double(value);
      case 'SPIKETIME'
         % Found spike time tag. Put the times into the struct array.
         data(currentElectrode).spikeTime = str2num(value);
      otherwise
         % If some tag does not know the file specification, warn the
         % user.
         error(['Found an unknown tag ' tag]);
   end

end
fclose(fileHandle); % Close the file
```

This code snippet first opens the file, then reads the lines one by one and stores this in the variable `line`. When the `fgets` function has found the end of the file, it returns -1 and stores this in `line`. At that point the `while` loop terminates and the file is closed. Try it on a simple text file! Don't forget to issue the `fclose` command otherwise you may have trouble later on, in another program, to open this file. Moreover, open files take up resources and are more prone to corruption than closed files.

The output variable of `fgets` always contains *strings* representing the current line in the file. If a line contains more than one number, *all* numbers will be part of this string.In the example file, all the numbers following a specific tag referred to the same physical quantity so I stored them in the same variable. In general, however, the numbers on a line could represent a hodge podge of information. Separating out the different parts from a string is no different from scanning a whole file of numbers and, in fact, you can use functions based on format strings for this purpose (`sscanf` and `textscan`), or you can use any of the string functions (e.g. `strsplit, strfind, strcmp`). For instance, if the SpikeTime tag lines contained both the spike times and then a character indicating the quality of spike isolation

```
SpikeTimes: 1 3 6 23 34 56 67 69 great
```

then, you could add the following code to the example above to extract this:

```
[spkTimes,~,~,next] = sscanf(value,'%f '); % Read all floating point
values from value
quality  = sscanf(value(next:end),'%s')    % Read a string from the
remainder
```

1.1.2. Reading Binary Files

Binary files are just a sequence of bits (0's and 1's); you have to know how to group those bits to make sense of such a file. For instance, a file with 256 bits in it, could represent four 64 bit double precision values. To read those numbers you read 64 bits, combine them as a single number, then read the next 64 bits, and so on. That same file, however, could also represent *eight* 32 bit integer numbers; in this case you'd read 32 bits, combine them as a single number, and move to the next 32 bits, etc.

For example, here is a binary number that consists of 64 bits:
1110010110010111011001101001110000111000101101100110010 1

At the most basic level all information in a computer, whether in memory or in a file, is stored in this binary format (sequences of 1's and 0's). To determine what this sequence *means* we have to know how to group the bits. For instance, this sequence could represent one double precision number (64 bits) or two 32 bit integers, or two 32 bit single precision floating point numbers. Each of those interpretations uses the same 64 bits in a different way and will result in entirely unrelated numbers.

In fact, even a text file looks like a sequence of 0's and 1's to the computer; in that case the bits are grouped in groups of 8 (a byte) and each such byte corresponds to a specific character. For instance, the byte 01000001corresponds to the letter A. In other words, a text file is a binary file too! The only difference is that the rule of how to combine bits into meaningful groups is fixed in a very specific way. In a binary data file there are no rules, or rather, the rules are whatever the program that generates the file decides. For instance, a binary file specification could state that the first 8 bits are always to be interpreted as a character, the second group of 64 bits is a floating point number, and the next 32 bits are an integer number.

Given this flexibility it should be clear that reading a binary file is almost impossible without a very explicit specification (e.g. how the bits should be combined to create numbers). The main advantage of a binary file is that it is very fast to read and can store much more information than a text file of the same size.

To read binary data files you use `fread` and, based on the file specification, you tell fread exactly how to interpret the bits in the file. Consider a file, for instance, in which the first number is a 32 bit integer that determines the number of spikes that were recorded. Immediately after this number you'll find the times at which those spikes occurred; these spike times were stored using double precision (=64 bits) floating point numbers.

```
fid = fopen(filename,'r');
nrSpikes   = fread(fileHandle,1,'int32');
spikeTimes = fread(fileHandle,nrSpikes,'float64');
fclose(fid);
```

You have to be very careful when reading binary files: be sure to follow the definition of the format of the file to the letter. Reading files with the wrong size specifier (i.e. reading an int as a float) will lead to complete nonsense numbers in your data. (Numbers such as 1e+109 in a vector that was read from a binary file are an almost sure sign that your file reading routine is incorrect).

One way to help debug your code is to read numbers one at a time instead of all at once. In the code above we could have read the spike times one at a time by calling `fread(fid,1,'float64')` five times. If one of the spike times suddenly had an impossible or unlikely value, this would give you a clue where the read routine failed. However, once debugging is complete, you should read all values with the same format at the same time; that is much faster than reading values one by one.

Apart from the wrong size specifier, errors do occur due to differences in binary formats. Matlab's default is to read data in the format that the machine on which the program runs uses. I.e. if you run Matlab under MS Windows on a PC, it will use a PC format. This spells trouble when you import data generated under UNIX, where the byte ordering of binary numbers is (sometimes) reversed. This is called big-endian versus little-endian formats. Note that it does not matter where the file is *stored*, it matters which operating system (or even program) was used to *generate* this file. If nonsense numbers show up in your data, this is another thing to keep in mind.

One further twist on binary file reading is that you can choose how to represent numbers in Matlab. For instance, you can read a 32 bit integer ('int32') from a file, but choose to represent it as a 64 bit floating point number (called a double) in Matlab. For instance:
`values= fread(fid,10, 'int32=>double')`
reads in 10 32 bit integers from the `fid` file handle, but those values will be stored in Matlab as 10 doubles (in the `values` variable). See `help fread` for further details. This transformation deos not cause a loss of information because every integer that can be represented with 32 bits can also be represented with the 64 bits of a double. A reason for representing integers as doubles is that many built-in Matlab functions will only work numers with double precision. The only price you pay is that the memory used by Matlab will fill up more rapidly. So, if you have truly large datasets that use 16 bit values, it may be worth keeping them in their native format even after importing into Matlab.

1.2. Storing Matlab Results

1.2.1. Data

A Matlab matrix can be saved to a file by typing
`save filename variableName1 variableName2`
This stores the variables in a Matlab format which can easily be read back into Matlab, but is difficult to decipher by other programs. If you want to save your whole Matlab session (i.e. all the variables currently in scope) just type
`save filename`
Next time, when you want to continue working on these data, just type
`load filename`
to continue where you left off.

Note that Matlab's file format has changed over the years, so to exchange data with someone who uses an older version of Matlab you can use a `-v` argument to the `save` function to tell it to use an older format. For very large matrices (above 2 GB), however, you need the `v7.3` format. Even in recent Matlab versions this is not the default format because it creates very large files (even when the matrices are not that large). If you find that your analysis is slow mainly because reading data from .mat files takes a long time, you should have a look at the fast save routines on the Mathworks file exchange.

1.2.2. Figures

Figures can be saved from their menu bar. Selecting File|Save will prompt you for a filename. This can be a useful way to store the results of a long analysis, together with the fine-tuning that you may have done to get the figure to look just right. Saving it as a Matlab figure will allow you to continue editing at some later time; this would not have been possible (at least not in Matlab) if you printed the figure to file. When you try to save the figure again Matlab will prompt you for another filename, to save it to the same name, just select the filename and confirm the "overwrite" prompt. To open this figure, just type its name on the command prompt: you can treat it as just another m-script.

Apart from being a future proof way of storing the results of an analysis, saved figures can also be used as base figures on which similar data are plotted. You could, for instance, prepare a figure with labels, titles, colors etc, call this `fancyPlot` and instead of running the command `figure` before plotting a new dataset, type `fancyPlot` to open a new copy of your fancy figure. Issuing the plot command afterward will simply add the new data to the pre-fab figure.

1.3. Exporting Data

To use data that you have analyzed in Matlab in other software packages, you will have to export them. The easiest way to do this is to use the `save` command with the `-ascii` option. This will save a matrix as a text file that most other packages will be able to read. As you learned in the importing data section, deviations from the nice rectangular layout of a table can be hard to deal with; saving data as a straight table makes it a lot easier for the person who wants to process your results in some other environment.

As shown in the table, Matlab can also generate a number of specialized files representing images, sounds, and movies. If your data do not fit any of these formats, and you cannot represent the data as a straight table, I'd recommend using an xml based format. Most other analysis packages have built-in routines for reading XML data, and XML by its very nature enforces a certain degree of rigidity and adherence to structured specifications, which is a good thing for output files.

Another way to share data is to use a database. The Matlab Database Toolbox offers a relatively easy way to communicate with local or remote databases that are based on the SQL database language. Even though this is increasingly important in this age of Big Data, it is beyond the scope of this introductory book.

Format	Function
HDF 5	h5write
DICOM	dicomwrite (Image Processing Toolbox)
XML	xmlwrite
BMP, GIF, JPEG, TIFF, PNG	imwrite
WAVE, FLAC, MP3, MPEG-4	audiowrite
AVI, QuickTime Mpeg-4 WMV	VideoWriter
Microsoft Excel	xlswrite

Of course you can also export data using binary files. The sequence of events is very similar to reading a file: first you open a file handle, then you call `fwrite` and specify a format.

```
fid = fopen('file.img','Wb'); % Open for *b*inary *W*riting
fwrite(fid,myIntegers,'uint16');
fwrite(fid,myDoubles,'double');
fclose(fid);
```

Note that I chose to write `myIntegers` as 16 bit unsigned integers. If the `myIntegers` variable contained floating point numbers then this would cause a loss of information (all values would be rounded down to the nearest integer, and values above 65535 ($=2^{16}$-1, the largest 16 bit integer) would be stored as 65535), and values below 0 would become 0). So, before you decide on a format to use in the file, you'd better make sure that the variable that you are writing can be fully captured by that format. To be extra careful, you'd want to check this in your code using function such as `intmin` and `intmax`.

Finally, when creating an output format, think about the code that the user of that file will have to write to import it. As you learned in the import section, the more rigorous the specification, and the fewer possibilities, the better!

1.4. Excercises

1. Write error-handling routines that issue a warning if a file could not be opened for reading or writing. As this is a routine that is useful in many applications, write it as a separate function, called safeOpen, which takes a filename and a mode-string as its argument and which returns the fileHandle. Make the error-messages user friendly: determine the cause for the error (does the file exists at all?, is it open already?, is it in binary/text format?)

2. Create a import function that can read data files that represent the result of an experiment in which subjects performed multiple trials of a reaction time task. Neither the number of subjects nor the number of trials is specified. The only information you have is that each line contains either a comment (lines starting with '%') or it contains the reaction times for one subject. Note that comment lines can happen anywhere in the file. You are allowed to assume that there are at most 25 subjects, and at most 100 trials for a given subject.

   The output argument of the function should be a matrix (not a cell array!) in which each column contains the reaction time data for a single subject. The number of columns should equal the number of subjects, and the number of rows the maximum number of trials that any subject ran. Missing values/trials should be NaN in this matrix. A short example file:

```
% This file contains the data for all subjects in the reaction time experiment.
% Each line contains the data for one subject. The number of trials varies per subject.
 0.1067   0.4314   0.8530   0.4173   0.7803   0.2348   0.5470   0.9294
% Comment lines can happen anywhere in the file
   0.9619   0.9106   0.6221   0.0497   0.3897   0.3532   0.2963   0.7757   0.3786   0.3012   0.9234
   0.0046   0.1818   0.3510   0.9027   0.2417   0.8212   0.7447   0.4868
```

3. It is fairly common for data files to consist of two parts. The first part is a header, with information on what can be found in the file. The second part is the real data. The header and data can be inside the same file (with the header just the first part) or header and data can be stored in separate files.

   In this assignment the goal is to write Matlab code that reads in data from a pair of header (.hdr) and image (.img) files. The header contains text describing the content of the image file. The image file contains binary data (In this case the image of one or more slices of brain obtained using MRI).

   The .hdr file contains text that describes the number of rows, columns, and slices in the corresponding .img file as well as some text deta (comments) in a tag:value format, as follows:

```
nrRows: 10
nrColumns:20
nrSlices: 2
Comments: A nice picture
```

   Each of the entries is on its own line and has the format parameter name:value. You can open the .hdr file in a text editor to see what it looks like. Write a function that opens this kind of file, extracts the values for each of the parameters, and stores them in a struct with the fields named after the parameters (e.g. data.nrRows = 10, data.nrColumns=20, etc.);