Chapter 1
Mission Possible

System-, process- and control engineers (and maybe other engineers) are frequently faced with simulation studies and tasks to analyze system and process data. They collect the data in terms of log files from equipment or data acquisition systems and analyze the data in order to understand the underlying systems or processes, monitor them and establish a data based concept or theory that enables them to improve the control of the system/process.

Engineers in research and development areas are usually forced to develop their own software tools for the analysis and interpretation of the investigated phenomena, and MATLAB® can help them as a powerful tool to support these tasks. This chapter presents some basic building blocks for a tiny MATLAB® based log data analysis tool, which has its major value in demonstrational reasons.

The tiny analysis tool will be developed later on into a powerful dialog driven analysis tool based on a MATLAB® object class, called *Carabao*, which supports rapid prototyping of a graphical user interface (a so called 'shell') for importing log data, analyzing the data and rapid creation of reports.

Getting Started

Let's assume we got some log data consisting of two data streams *x* and *y*, each containing a sequence of 1000 numbers, and we are assigned with the following mission:

Mission #1: Analyze this data by plotting it and calculating statistical numbers like mean values, standard deviations and the correlation coefficient.

Let's get hands-on! I hope the reader has some basic knowledge about MATLAB®, otherwise he might take some introducing lectures ([1] would provide some good basis training). And I hope the reader has access to MATLAB®, so I invite him to open MATLAB® and to play a bit around with me. The first job we do is to generate some sample data which we can treat as our log data to be analyzed. Let's start to generate a log data matrix with 1000 rows and 2 columns. Creating a 1000 x 2 matrix of normal distributed random numbers would be a reasonable start,

```
>> log = randn(1000,2);
```

but this data set would be too boring for us since all we would get is unbiased and uncorrelated data. So a better choice is something like

```
>> rng('default'); % reset random generator
>> log = ones(1000,1)*randn(1,2) + randn(1000,2)*randn(2,2)
log =
   -1.7813    1.8336
    1.7186    1.2289
    0.6787    2.2148
   -0.6088    1.4329
    0.4465    1.1098
    0.7128    2.1951
       :         :
```

Note that before using the *randn* function we reset the random generator to its default seed value. This gives the reader the opportunity to reproduce the same 'random' log data as is shown here. Some reader might challenge him or herself already with the statistical meaning of this sample log data and might conclude that the variable `log` will hold two columns of data sequences which are now both biased and correlated to each other. Let's assign the columns of our sample log data to variables *x* and *y*.

```
>> x = log(:,1);  y = log(:,2);
```

Now we can perform our first graphical analysis. Let us study a scatter plot.

```
>> scatter(x,y,'k')  % black scatter plot
```

MATLAB® will open a new figure which displays the following graphics.

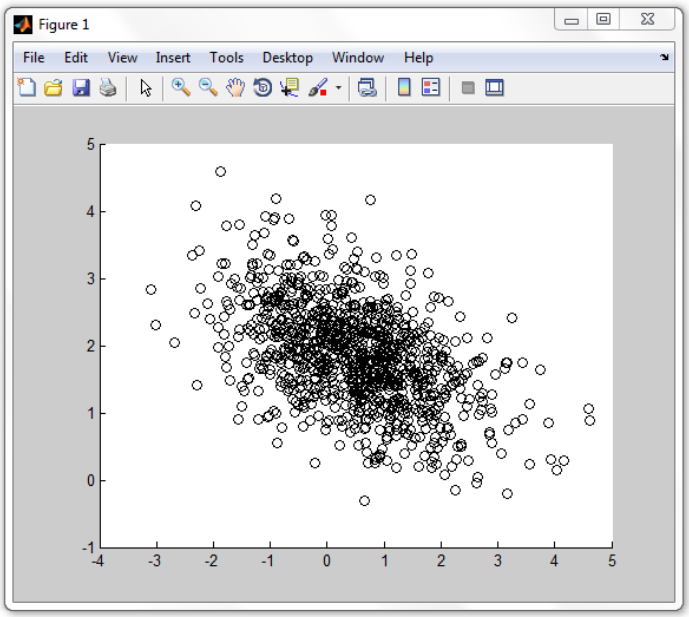

Fig.1.1: scatter plot of our sample log data

We see a crowd of black circles and we can recognize easily that the center of this heap is not the coordinate orgin (0,0) so the data streams x and y are biased and the mean values of x and y could be somewhere at x0 ≈ 0.5, y0 ≈ 2. Let us do the calculation of the actual mean values.

```
>> m = mean([x y]) % mean value
m =
    0.4857    1.8666
```

Our estimate was not too bad, so let us try a guess of the standard deviation. A simple approximation formula says that the standard deviation is one sixth of the data range which gives us std(x) ≈ (5-(-3))/6 = 1.33 and std(y) ≈ (4.5-(-0.5))/6 = 0.83. Let's compare these values with the actual numbers.

```
>> s = std([x y]) % standard deviation (sigma)
s =
    1.1480    0.7454
```

Again I would say that we made a good estimate. What about the correlation of the data? We can recognize that for greater x values the y values have a trend to get lower and vice versa. Thus we would expect a negative correlation coefficient. If we ask MATLAB® for the result we get the value -0.4803 for its value.

```
>> c = corrcoef(x,y)
c =
    1.0000   -0.4803
   -0.4803    1.0000
```

We can label our figure and display the correlation coefficient in the graph's title. See the resulting graphics in Fig. 1.2.

```
>> xlabel('x data');
>> ylabel('y data');
>> title(sprintf('correlation coefficient %g',c(1,2)));
```

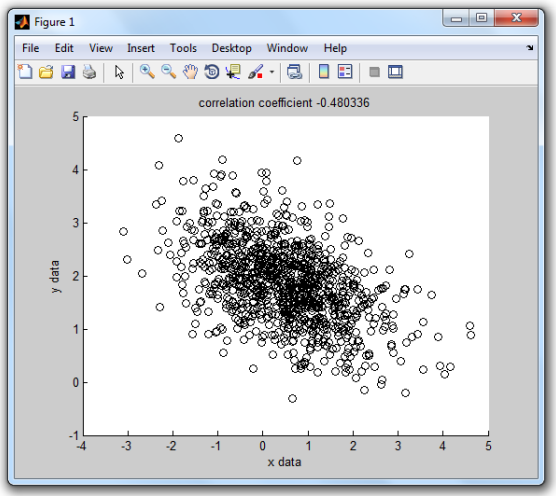

Fig 1.2 – labeled scatter plot

We can open another figure and plot the data sequence of our x-stream in red color and label the graph title with the mean value and standard deviation of the x-sequence.

```
>> figure % open new figure
>> plot(x,'r');
>> xlabel('data index');
>> ylabel('x data');
>> title(sprintf('x-stream: mean %g, sigma %g',m(1),s(1)));
```

In the same way we can plot the y-stream in blue color.

```
>> figure % open new figure
>> plot(y,'b');
>> xlabel('data index');
>> ylabel('y data');
>> title(sprintf('y-stream: mean %g, sigma %g',m(2),s(2)));
```

Fig. 1.3 shows what we should get.

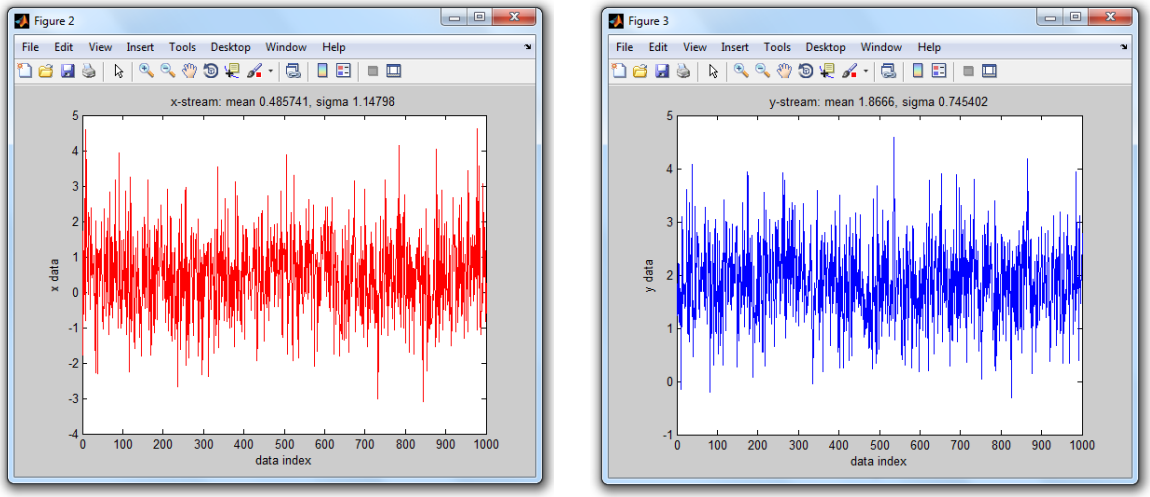

Fig. 1.3 – stream plots of x- and y-streams

Caramba, we had a good start! We generated some (non-boring) sample log data, assigned the two data streams to working variables *x* and *y*, did a basic graphical analysis by generating a scatter plot and stream plots of the *x* and *y* data stream. We calculated statistical numbers like mean values, standard deviations and the correlation coefficient and displayed the results in the titles of our graphs. We can be satisfied about mastering our first mission – make a break – time for an espresso!

Fig. 1.4 – Time for an espresso!

Ready For Our Next Mission

I hope you enjoyed the espresso and could have some relax. And I hope you are ready for our next mission. This time we have to assume that the log data is stored in several log files, for convenience let the files be *data1.log*, data2.log, … , *data<n>.log.* The log data contains both parameters, like a title, and the data.

> *Our mission #2 is to provide an easy-to-use MATLAB® tool that allows any user (who gets some short instruction) to do an analysis of a given data log file (which can be e.g. data2.log) in order to produce the three graphics plots with proper labeling, which we introduced in the previous section.*

How would we solve our second mission? One of the solutions is to think of an *analysis* function which, when called, presents a file selection dialog box that allows the user to select the desired log data file, reading then the parameters and data from the selected log file into variables *x* and *y* and opening three figures that will be used for a scatter plot and twice a stream plot (one for *x,* the other for *y)* provided with proper labeling. Such a function, let's call it *analysis*, could look as follows.

```
function analysis                    % log data analysis
   path = filedialog;                % open file dialog, select log file
   if ~isempty(path)                 % if dialog not canceled
      [x,y,par] = read(path);        % read data (x,y) and parameters
      figure                         % open new figure
      scatterplot(x,y,par);          % draw black scatter plot
      figure                         % open new figure
      streamplot(x,'x','r',par);     % plot x-stream in red color
      figure                         % open new figure
      streamplot(y,'y','b',par);     % plot y-stream in blue color
   end
end
```

I may assume that the reader is familiar with *MATLAB® m-file functions*, the most common way to create user defined functions (otherwise the reader is strongly advised to read some MATLAB® introduction where this topic is explained, e.g. [1]). This means we have to create some text file *analysis.m* in a so called *path folder* (a folder which is on the MATLAB® path). But wait a moment!

Organizing Our Files

We should suspend our mission at this point and think a little bit of how we want to organize our files in the file system. We talked about *log files* which are going to be analyzed, and we got just yet in touch with *m-file functions* which we plan to create. Later on we will hear about *class definitions* which are also stored in m-files and which have to be located in so called *class folders* (or *class directories* – see [2])

In addition we should notice that our approach is learning by doing – we just started by playing around, writing some code pieces with just the minimum required functionality, with a further aim to refine these code pieces step by step until they reach their final power level. This means we will deal with different versions of code pieces, and we should think about some smart file organization that allows us to keep the overview.

When I am starting some new project and I'm forseeing that there will be a growing number of files related to it (Word and Excel docs, power points, MATLAB®-files, images, etc.) I usually create a project folder for the project which contains somewhere in its depths all the project related files according to some proper organization structure created in my mind. This has the big advantage that I can zip just this project folder and get a quick backup file of everything related to this project.

If I develop a MATLAB® tool I always work with *version folders*. Beyond the fact that this allows easy backups of the actual developed version there is the additional advantage that a switch between versions is easily possible by adoption of the MATLAB® path. As a further benefit this structuring allows straight forward documentation where code of a provided source code package can be found, since a simple comment like `% log data analysis (v1a/analysis.m)` tells the reader everything where the source code for e.g. function *analysis* can be found.

```
function analysis   % log data analysis (v1a/analysis.m)
```

Let us follow this idea. Let us call the actual project *PLAY* which we assign with some project folder *play* that may be located somewhere in the file system (the specific location does not matter as long as we are able to remember it). We will create a log folder *play/log* to hold the several log data files, and create the first version folder

play/v1a which we have to include in the MATLAB® path (MATLAB® GUI>HOME>ENVIRONMENT>Set Path). This leads us to the following (initial) folder structure.

play
play/log
play/v1a

Then let us select the folder play/log as our current folder, e.g. by selecting it in the MATLAB® GUI using a sequence of mouse clicks.

Creating Log Data Files

We are ready now to proceed, but before we start writing MATLAB® code we need to have some log files containing the data with which we want to play. Let us write a simple MATLAB® function *create* which allows us to create a log file:

```
function create(path)  % create random data log file (v1a/create.m)
%
%  CREATE   Create random data & log to a log file: create(path)
%
   [~,name] = fileparts(path);
   log = ones(1000,1)*randn(1,2) + randn(1000,2)*randn(2,2);
   x = log(:,1);  y = log(:,2);

   fid = fopen(path,'w');                    % open log file for write
   if (fid < 0)
      error('cannot open log file');
   end

   fprintf(fid,'$title=%s\n',upper(name));
   fprintf(fid,'%10f %10f\n',log');          % write x/y data
   fclose(fid);                              % close log file
end
```

The created log file shall begin with a parameter line which defines a title for our log data. The parameter definition should follow the generic syntax

'$' <parameter> '=' <value>

which we will use later on to provide additional parameter settings like title, date, time, serial number, etc. In the current example, however, we will restrict ourselves to a single parameter definition line providing the *title*.

There should not be big difficulty to understand how our *create* function works. The function is called by passing the file path information. At the beginning the *fileparts* built-in function is used to extract the file name from the file *path*, which is later on used to compose the title string. After that the log data is created and assigned to the variable *log* (in the same way as we did in the *"Getting Started"* section), from which the two columns are picked and assigned to the stream variables *x* and *y*. After opening a text file for writing and cross checking whether the file open operation was successful the parameter definition is written first using `fprintf(fid,'$title=%s\n', upper(name));` followed by a `fprintf(fid,'%10f %10f\n',log')` statement for the data columns, terminated by `fclose(fid)` to close the file which we had opened before.

Let us use this *create* function to create 5 log data files in our log folder. As our *create* function makes use of the random generator we should not forget to reset the random generator before any *create* call in order to get the same data as shown here!

```
>> rng('default');   % reset random generator
>> create('data1.log');
>> create('data2.log');
>> create('data3.log');
>> create('data4.log');
>> create('data5.log');
```

This is how the file *data1.log* should look like.

```
>> type data1.log
$title=DATA1
 -1.781269   1.833640
  1.718563   1.228938
  0.678696   2.214829
 -0.608834   1.432873
     :           :
```

Now we have enough log data files for playing around. Let's see next how we can get the data back from file into MATLAB®.

Reading Data

We need a *read* function which receives a file *path* on input, opens the specified file in reading mode, reads the parameters into the structure variable *par*, and scans the two data columns into the variables *x* and *y*. At this time we may insist on the assertion

that there is exactly one parameter line at the beginning of the file which will simplify the code for data import. Here we go!

```
function [x,y,par] = read(path)        % read log data (v1a/read.m)
   fid = fopen(path,'r');
   if (fid < 0)
      error('cannot open log file!');
   end
   par.title = fscanf(fid,'$title=%[^\n]');
   log = fscanf(fid,'%f',[2 inf])';    % transpose after fscanf!
   x = log(:,1); y = log(:,2);
end
```

Don't forget that every MATLAB® function we are creating has to be saved now in folder *play/v1a.* Let us test our *read* function.

```
>> [x,y,par]=read('data1.log');
```

Let us check whether the data has been imported correctly.

```
>> par
par =
    title: 'DATA1'

>> [x(1:5),y(1:5)]
ans =
   -1.7813    1.8336
    1.7186    1.2289
    0.6787    2.2148
   -0.6088    1.4329
    0.4465    1.1098
       :         :
```

Everything seems to work well, we can observe that the parameter *title* is very well assigned to the component *par.title* of the *par* structure, and that the variables x and y hold the same data streams that we have seen in our first mission.

Proceeding with our mission is now straight forward. Reviewing the concept of our *analysis* function we have to write the function definitions for *scatterplot* and *streamplot*.

```
[x,y,par] = read(path);       % read data (x,y) and parameters
figure                        % open new figure
scatterplot(x,y,par);         % draw black scatter plot
figure                        % open new figure
streamplot(x,'x','r',par);    % plot x-stream in red color
figure                        % open new figure
streamplot(y,'y','b',par);    % plot y-stream in blue color
```

We just have to put the commands we used in mission #1 into m-file functions. For function scatterplot we would have to draw the scatter plot, calculate the cross correlation coefficient and provide xlabel, ylabel and titel. The resulting m-file function would look as follows.

```
function scatterplot(x,y,par) % black scatter plot (v1a/scatterplot.m)
%
% SCATTERPLOT   Draw a black scatter plot: scatterplot(x,y,par)
%
   scatter(x,y,'k');          % black scatter plot
   c = corrcoef(x,y);         % correlation coefficients

   xlabel('x data');
   ylabel('y data');
   title(sprintf('%s: correlation coefficient %g',par.title,c(1,2)));
end
```

Testing *scatterplot* by invoking

```
>> scatterplot(x,y,par);
```

will open a new figure with a similar graphics as of Fig 1.2, except the graphics title be extended with the title parameter of our data file (fig. 1.5). Cool! Let's proceed with the *streamplot* function definition. Note that this function will be used to plot both the x-stream and y-stream. In order to perform proper labeling we need to provide symbolic data stream information (input argument *sym*, which has the actual value `'x'` or `'y'`).

```
function streamplot(x,sym,col,par) % stream plot (v1a/streamplot.m)
%
% STREAMPLOT   Plot data stream: streamplot(x,'x','r')
%
   plot(x,col);                   % stream plot
   m = mean(x);                   % mean value
   s = std(x);                    % standard deviation (sigma)

   xlabel('data index');
   ylabel([sym,' data']);
   format = '%s: %s-stream: mean %g, sigma %g';
   text = sprintf(format,par.title,sym,m,s);
   title(text);
end
```

Testing *streamplot* by invoking

```
>> figure; streamplot(x,'x','r',par);
>> figure; streamplot(y,'y','b',par);
```

opens two more figures with graphics similar as of fig. 1.3, except extended title (see fig. 1.6).

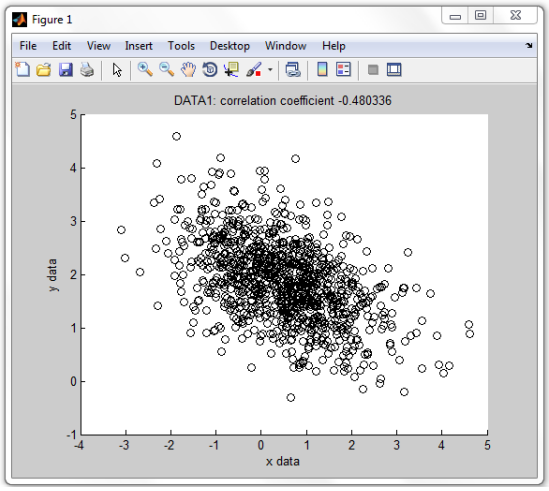

Fig. 1.5 – scatter plot with data title

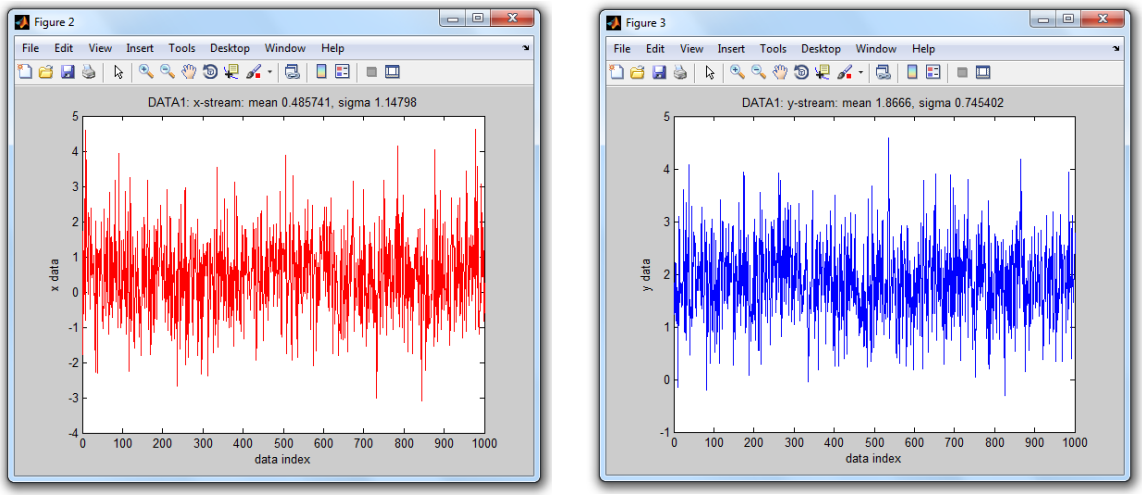

Fig. 1.6 – stream plots with data title

Awesome! We are already close to complete our mission. What is missing? The first line of function analysis asks for a *filedialog* function that allows us to select a file.

```
path = filedialog;              % open file dialog, select log file
```

The MATLAB® built-in function *uigetfile* can be used as the underlying working horse. Let's look at the following function definition for *filedialog*.

```
function path = filedialog   % select a log file (v1a/filedialog.m)
%
% FILEDIALOG   Dialog to select data log file: path = filedialog
%
   [file, dir] = uigetfile('*.log', 'Open .log file');
   if isequal(file,0)
      path = '';
   else
      path = [dir,file];
   end
end
```

The first line asks *uigetfile* to open a file selection dialog with filter `'*.log'` and caption `'Open .log file'`. On successful file selection the output argument *file* (file name) is not equal to zero and we return the output argument path as a concatenation of directory and filename. Otherwise, if the user terminated with CANCEL, we return an empty character string to indicate that the user aborted the dialog. Let's test it and select *data2.log*.

```
>> path = filedialog
path =
.../play/log/data2.log
```

Perfect! Now all parts for our *analysis* function are ready and we can create our *analysis* m-file function. Note that after invoking *filedialog* the subsequent tasks are only performed if the *path* variable has a non-empty value, which means that the user did not cancel the file selection dialog.

```
function analysis                % log data analysis (v1a/analysis.m)
   path = filedialog;            % open file dialog, select log file
   if ~isempty(path)             % if dialog not terminated with cancel
      [x,y,par] = read(path);    % read data (x,y) and parameters
      figure                     % open new figure
      scatterplot(x,y,par);      % draw black scatter plot
      figure                     % open new figure
      streamplot(x,'x','r',par); % plot x-stream in red color
      figure                     % open new figure
      streamplot(y,'y','b',par); % plot y-stream in blue color
   end
end
```

Function analysis provides every requirement of mission #2. Invoking

```
>> analysis
```

opens the file selection dialog of fig. 1.7 with a filter on file extension *.log and asks the user to select a .log file.

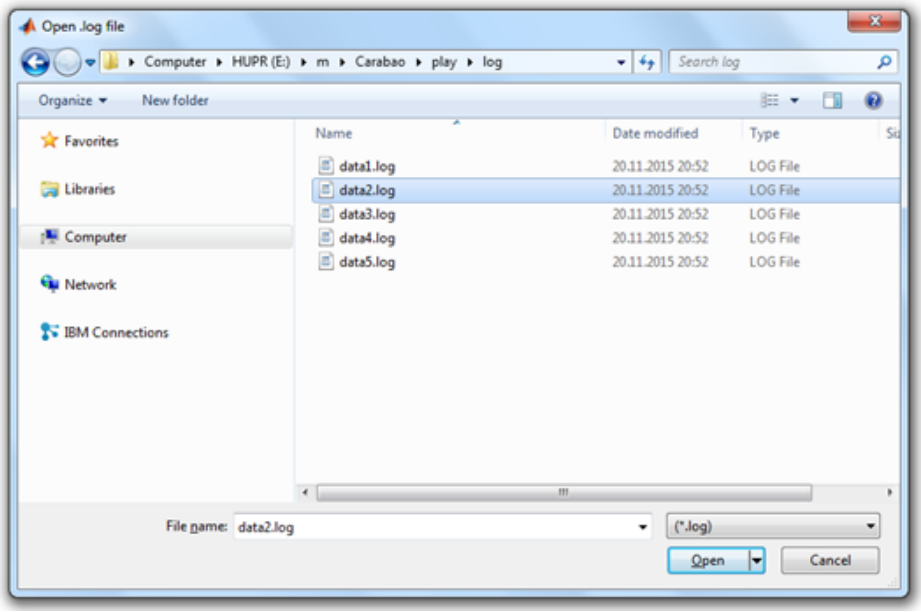

Fig. 1.7 – log file selection dialog

Once a data log file is selected the rest is running automatically. Three figures will pop up (see fig. 1.5 and fig. 1.6) with the scatter plot and the stream plots for x-stream and y-stream, including the statistical values of mean values, standard deviations and cross correlation coefficient in the title of the plots. Everything required for our mission #2 is available, and the usage of function *analysis* is easy and straight forward. Awesome, we completed mission #2 – it's again time for relax – and for another espresso!

Fig. 1.8 – Awesome, it's time for another espresso!

What More?

We made some nice building blocks which lead us finally to a fairly pretty data analysis tool. Frankly speaking our data log file had a very simple structure, and our analysis functions were also of very simple design. But this was by intention. I hope the reader has at least some idea how for more complex tasks our building blocks could be extended, even if the log file contains more parameters than only a title, and more than two data streams. It would be straight forward to increase the number of analysis functions which could be improved with more sophistication.

Anyway here are some aspects which would ask for additional non straight forward functionality:

- Let us realize that our log data and parameters were initially encapsulated in the log file. After reading parameters and data into variables (*title,x,y*) the 'ingredients' of the log file lost their encapsulation. As long as we work with data and parameters of only one single log file this should not be a big issue, but when we compare different log data with each other or generate overviews of several log data we would appreciate concepts based on encapsulated data. Such concepts are supported by *object oriented programming*.
- The current version of our analysis function pops-up three figures. Imagine that instead of 3 analysis graphs we have a need of 20 different graphs, and our modified analysis function would pop-up 20 figures in a batch. This is not what we expect as an ergonomic user interface for data analysis. Instead we would prefer a dialog driven user interface which allows us to pop-up interactively those graphics which are in the focus of our current interest. There might be also the possibility to provide parameter settings (like filter parameters, fitting order numbers, ...) which are used to tune the quality of the data analysis. Thus a good data analysis tool would be *menu driven* and *dialog based*.

Object Oriented Programming

The programming style we used in mission #2 is called procedural programming, where data is typically represented by individual variables or fields of a structure. Operations are typically represented as functions that take the variables as arguments. Programs usually call a sequence of functions, each one of which is passed data, and then returns modified data. The reader is advised to study again our *analysis* function which makes use of the individual variables *x* and *y* as well as the structure field *par.title* to be passed between the functions *read*, *scatterplot* and *streamplot*, which are called in a program sequence by function (or program?) *analysis*.

In an object oriented programming style you would study a family of applications (say data analysis tools) and identify patterns to determine what components and functionality are used repeatedly and in common. After that you would define base classes (super classes) that provide the common properties (data elements) and methods (function elements). For a particular application you would work with objects that are defined by a derived class (sub class) which inherits all properties and methods of the base class. The derived class further adds individual (or overloaded) methods and properties which are used to solve the tasks of a specific application. In next chapters we will get in touch with the object oriented approach.

References

[1] *Stormy Attaway*: MATLAB® – A Practical Introduction to Programming and Problem Solving (3rd edition); Butterworth-Heinemann, Elsevier Inc. 2013, ISBN: 978-0-12-405876-7

[2] MATLAB® – Object-Oriented Programming – R2015b; Mathworks, online on the internet