Getting in Touch with the Carabao

*What is a Carabao – and what has a Carabao to do with rapid prototyping? A carabao is a swamp-type domestic water buffalo, and for a Philippine farmer a carabao is a source of draft animal power which is endlessly helpful for mastering the challenges of his daily life. A Carabao is a MATLAB*®*-type 'domestic' class object, and for a system-, process- or control-engineer a Carabao is a source of conceptual power which is endlessly helpful for mastering the rapid prototyping challenges of his daily life.*

[](http://www.google.co.jp/imgres?imgurl=https://c1.staticflickr.com/3/2653/3991299542_5eb705e9f5_b.jpg&imgrefurl=https://www.flickr.com/photos/curufinwe-xiane/3991299542&h=683&w=1024&tbnid=IcvOhi2zfOm_FM:&docid=rAnOCX3ekEyOkM&ei=Bo9IVtu0Joq_0ASwy6zQCQ&tbm=isch&ved=0CHwQMyhWMFZqFQoTCNvPm8PKkskCFYoflAodsCULmg)System-, process- and control engineers (and maybe other engineers) are frequently faced with 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 system or process, monitor them and establish a data based concept or theory that enables them to improve the control of the system and process.

[](http://www.google.at/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0ahUKEwi61IeYsfnJAhWGKQ8KHc0MBHcQjRwIBw&url=http%3A%2F%2Fwww.lakas.com.ph%2F2012%2F06%2F11-things-to-do-in-carabao-island-romblon%2F&bvm=bv.110151844,d.ZWU&psig=AFQjCNGGJ4jwovesJH8Ye2OtYw6yvyY5IA&ust=1451214402512585)

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 paper is about 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. The data is contained in properties of objects, and those objects can be stored to a database and retrieved from there. Like files in a file browser these objects can be copied and pasted into so called packages which allow to generate overviews over an ensemble of objects and to calculate statistical KPIs over the ensemble.

# 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

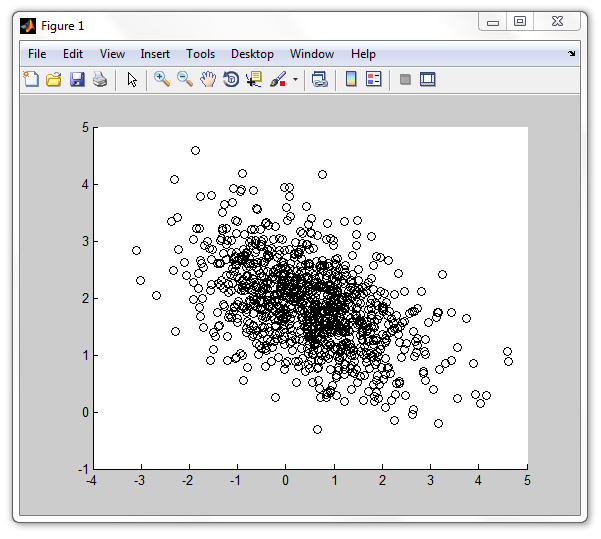
MATLAB® will open a new figure which displays the following graphics.

Fig.1: 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.2.

>> xlabel('x data');

>> ylabel('y data');

>> title(sprintf('correlation coefficient %g',c(1,2)));

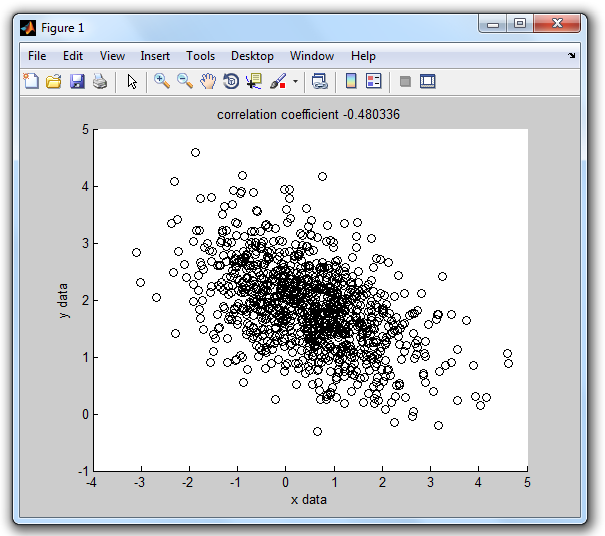


Fig.2 – labeled scatter plot

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.3 shows what we should get.

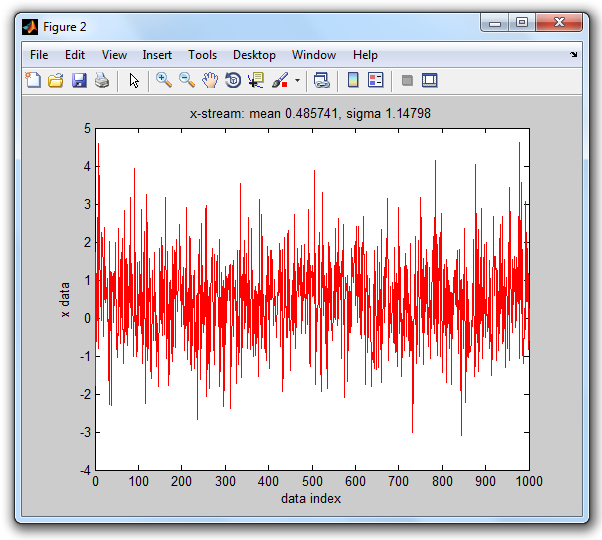
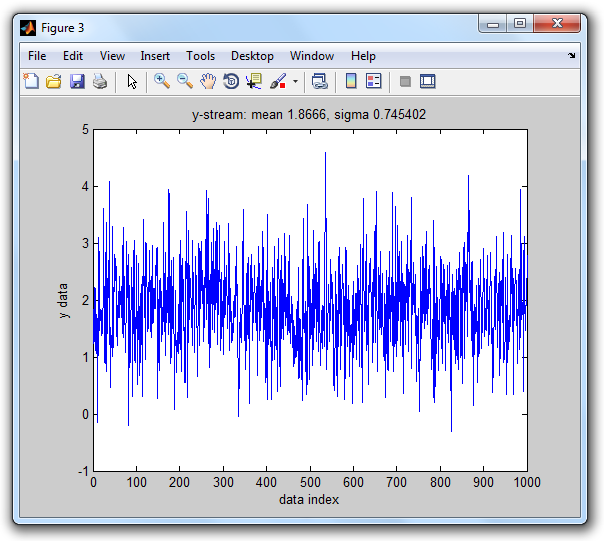


Fig.3 – stream plots of x- and y-streams

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.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 an *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.2, except the graphics title be extended with the title parameter of our data file (fig. 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(1),s(1));

title(text);

end

Testing *streamplot* by invoking

>> figure; streamplot(x,'x','r',par);

>> figure; streamplot(y,'y','b',par);

will open two more figures with graphics similar as of Fig.3, except extended title (see fig. 6).

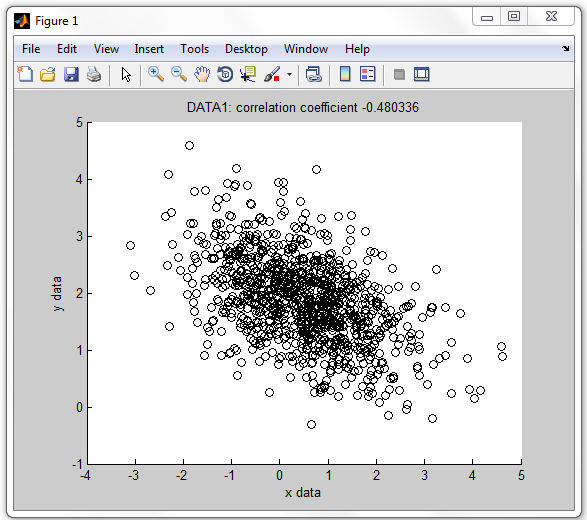


Fig. 5 – scatter plot with data title

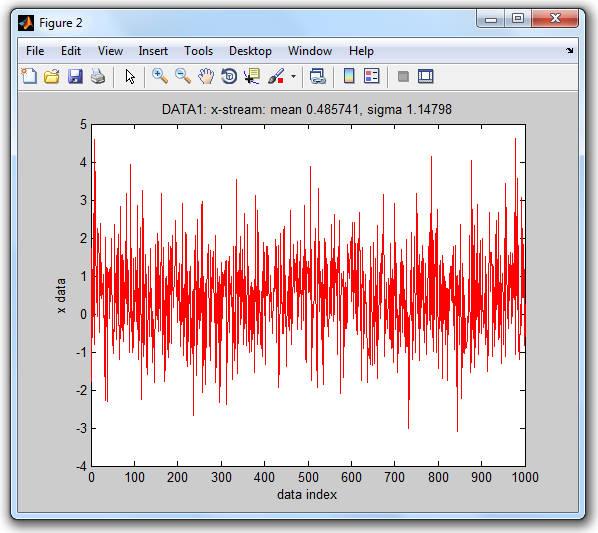
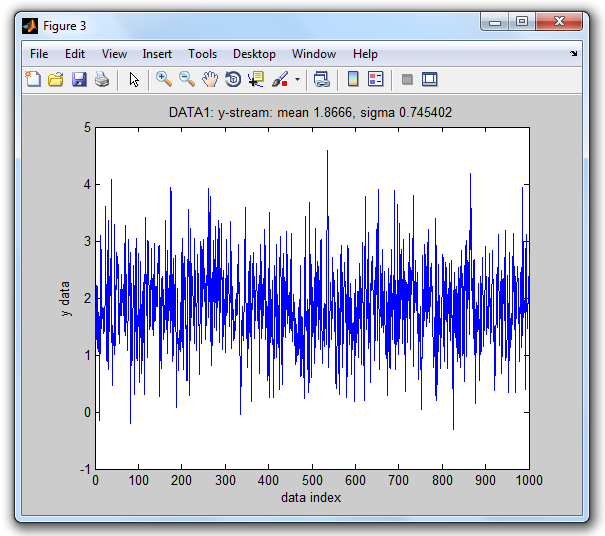


Fig. 6 – stream plots with data title

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. 7 with a filter on file extension \*.log and asks the user to select a .log file.

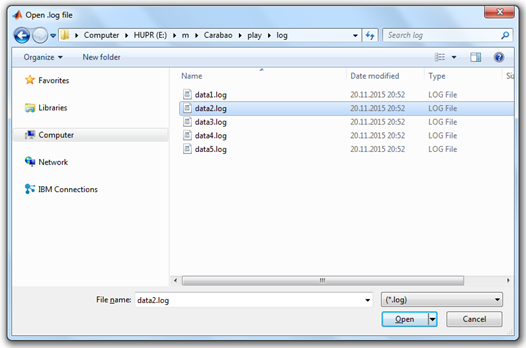


Fig. 7 – log file selection dialog

Once a data log file is selected the rest is running automatically. Three figures will be popped up (see fig. 5 and fig. 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. 8 – Awesome, it's time for another espresso!

# What More?

We made some nice building blocks which leaded 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 I will list 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 were easily accessible but we got rid of the 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 define by a derived class (sub class) which inherits all properties and methods of the base class. The derived class futher adds individual (or overloaded) methods and properties.

# Getting in Touch with Carabao

This is the right time to get in touch with Carabao. Carabao is an object class which can be utilized for rapid prototyping of dialog driven data analysis tools (and also other type of tools) based on object oriented programming. The Carabao class serves as a generic base class (super class – see [2]) supporting common functionality like parameter and data management, menu and refresh control, as well as mass storage functionality. Carabao includes also a *rapid prototyping* tool which can be used to derive a specific object class with customized functionality. We will use this *rapid prototyping* functionality later to prototype a data analysis tool that can deal with the kind of log data we used in previous sections.

To get a better feeling what is meant by *common functionality* let us first study the functionality of the standard Carabao shell. To get a standard Carabao shell we have to construct a Carabao object and then have to apply the method *shell* to the constructed object. We will heavily use the variable name *o* (somitemes *oo*) for object variables.

>> o = carabao; % construct a carabao object

>> shell(o); % same as: shell(carabao);

It is required to have a Carabao class version folder on the MATLAB® path. The examples presented here are based on version *Carabao V1c*, which means that the folder *…/Carabao/v1c* has to be included in the MATLAB® path. Assuming this kind of MATLAB® path setup the statement >> o=carabao constructs a *Carabao* object and assigns it to the object variable *o* while the second statement >> shell(o) invokes the carabao method *shell* (note that the object *o* is passed as the first method argument [2]) in order to launch a graphical shell. A new figure with a menu bar looking like Fig. 9 will appear on the screen.

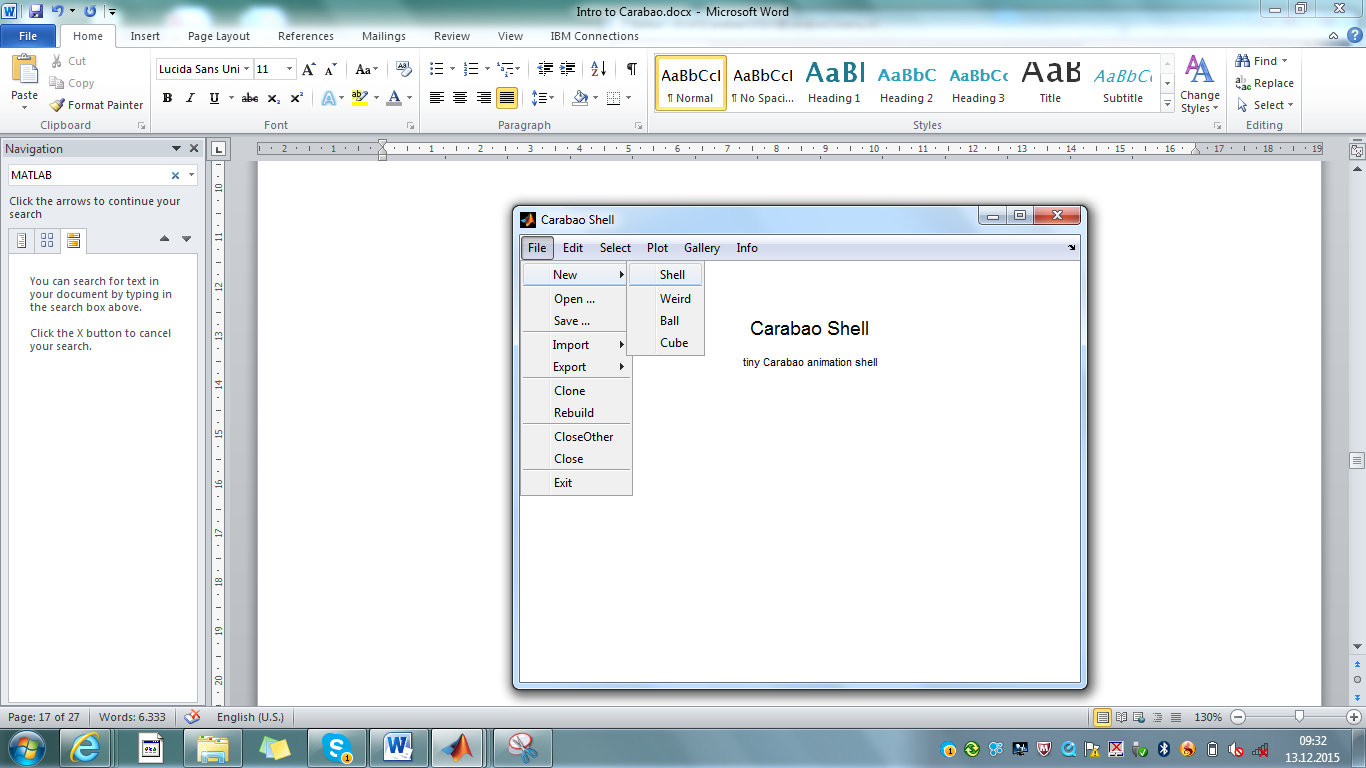


Fig. 9 - carabao shell

# A Weird Object

What can we do with this shell? Not much, as long as there are no objects assigned with the shell! Luckily there is a menu functionality to create new sample objects which are immediately 'pasted into the shell'. Let us click on *File>New>Weird*. A new title 'Weird Object – (11-14-7-11) ' appears on the figure screen (see fig. 10).

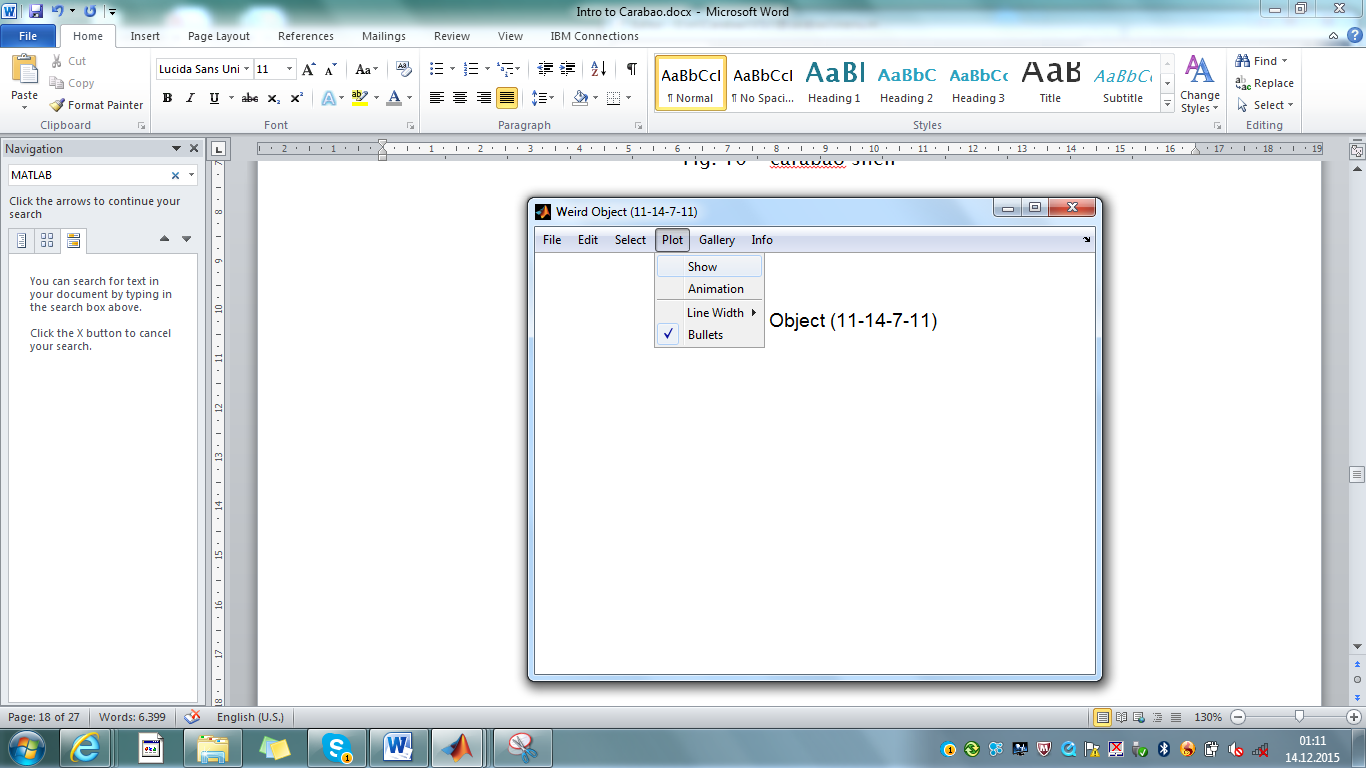


Fig. 10 - carabao shell, containing a 'weird object'

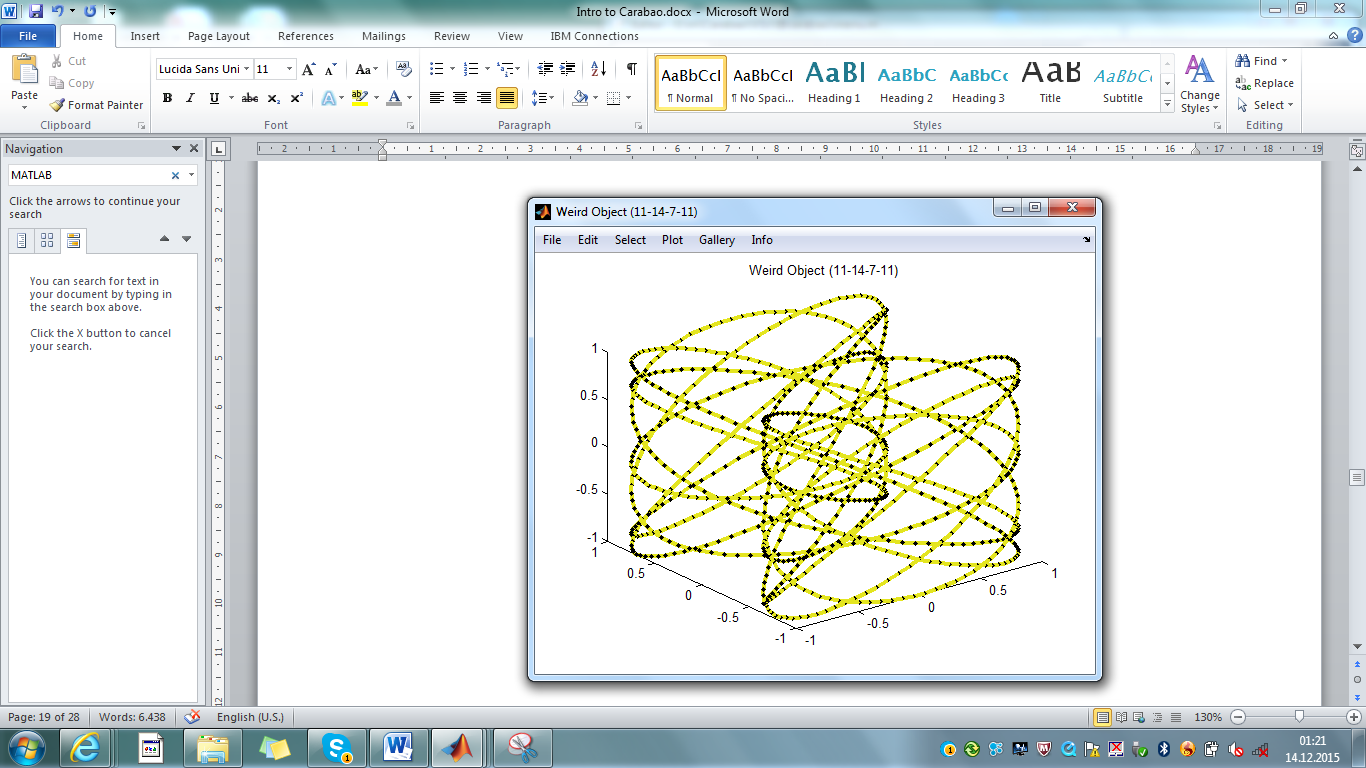


Fig. 11 – clicking on *Plot>Show*) shows the 'weird object'

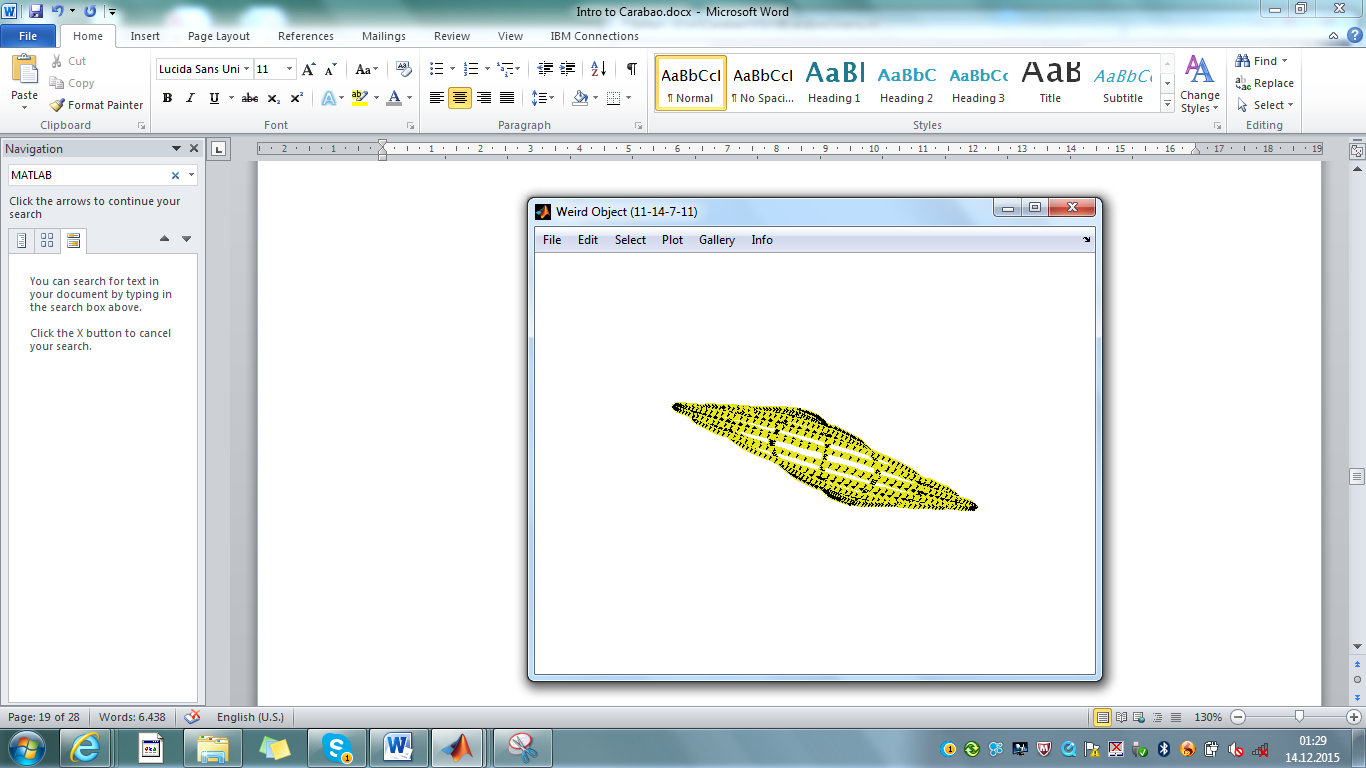
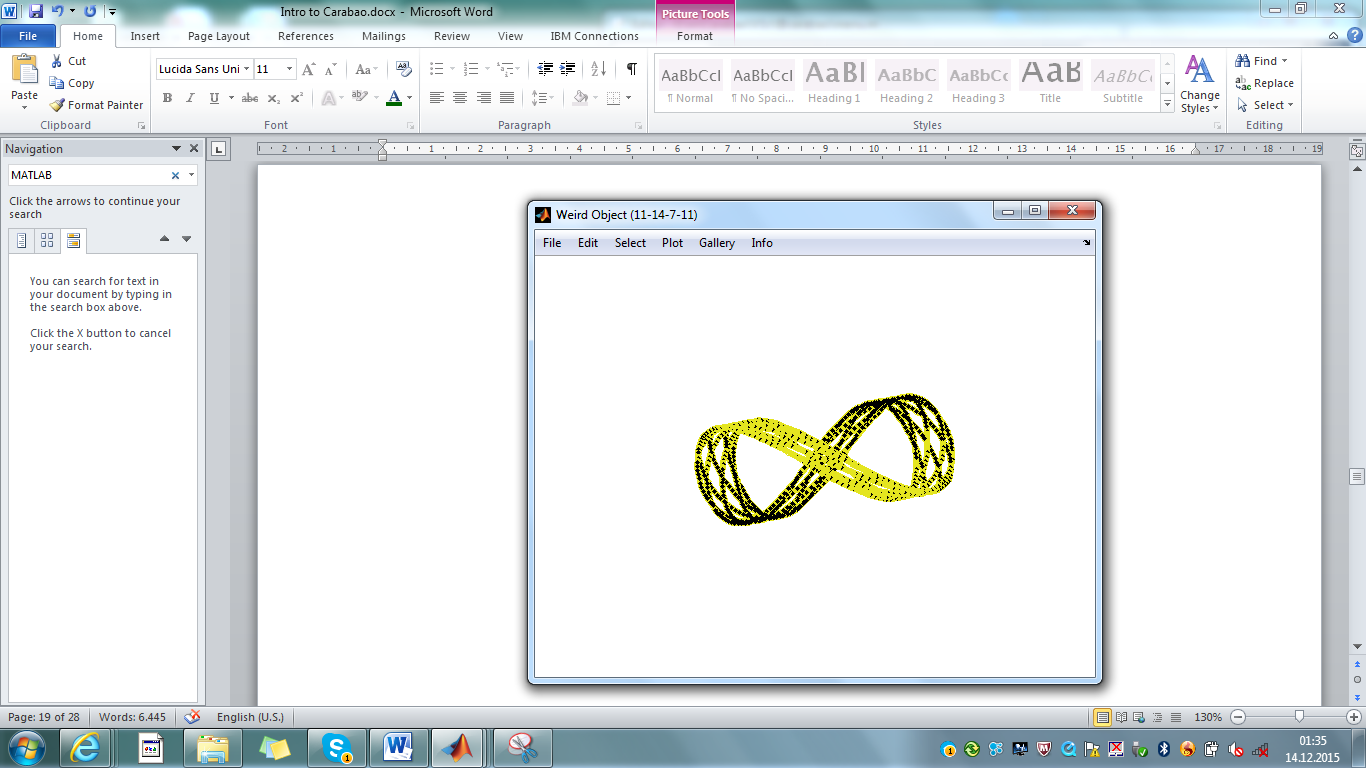


Fig. 12 – *Plot>Animation starts* some 'weird animation'

If we activate the menu item *Plot>Show* to show a plot of this weird object, we get some understanding why this object is called a 'weird object' (fig. 11). The full weirdness of the object comes into scene if we start animation (click on *Plot>Animation -*  a click into the figure's plotting area will stop animation). Initially it seems that we are dealing with some 3-dimensional 'spaghetti object' which is spinning around some axis. After some time, however, we get the feeling that the spaghetti takes place on the surface of a 3D prism with a cross section shape of a ∞-character (fig. 12 left), but later on the 3D shape seems to get lost as the whole spaghetti seems to be flattened, squeezed into a 2D quadratic area which is rotating in 3D space (fig. 12 right).

What is the secret behind this weird impressions? The whole magic is based on a 3D-projection of a 4D curve where the projection coordinate system is rotated in 4D space. The interested reader may inspect method carabao/shell>Create>Weird for the details (e.g. enter >> edit carabao/shell). A 4-vector of random integer numbers between 1 and 20 is assigned to variable *n* (the 4 order numbers *ni* which also show up in the object's title), and then the curve coordinates are built-up of *cos(ωI t)* and *sin(ωI t)* functions with circular frequencies *ωi* = 2π*ni* .

function oo = Create(o) % Create New Object

: :

function oo = Weird(o) % Create Weird Object

: :

t = (0:999)'/999; % time vector, 1000 points

n = 1+round(18\*rand(1,4)); % order numbers

w = cos(2\*pi\*n(1)\*t); % w vector

x = sin(2\*pi\*n(2)\*t); % x vector

y = cos(2\*pi\*n(3)\*t); % y-vector

z = cos(2\*pi\*n(4)\*t); % z-vector

: :

end

end

The resulting curve could be called a '4D Lissajous graph'. As human beings have normally no intuitive experience with 3D-projections of 4D objects this kind of animation will cause an effect of 'weirdness'.

# Pulling an Object from the Shell

Can we see some details of 'the object' assigned with the Carabao shell? Certainly we can – using the *pull* method. But be aware that for every application of a method[[1]](#footnote-1) we need some object to pass to the method by argument. If we have no object we can simply use a Carabao constructor (method *carabao*) to return a Carabao object.

>> o = pull(carabao) % same as: o = carabao; pull(o)

CARABAO object

MASTER Properties:

tag: carabao

type: shell

par:

title: 'Carabao Shell'

comment: {'tiny Carabao animation shell'}

data: [1x1 carabao]

WORK Property:

opt: [1x1 struct]

arg: {1x0 cell}

figure: 1

What can we see now? Obviously the statement >> o = pull(carabao) 'pulls the object out of the shell' and assigns it to variable *o*. Since we did not provide a semicolon terminator the MATLAB® command interpreter will display some object internals. The first line CARABAO object gives us confidence that the object is really of class *carabao*. The statement

>> class(o)

ans =

carabao

would be an alternative to retrieve the class name. Furthermore we can see that the object has four master properties (*tag*, *type*, *par* and *data*) and a working property (*work*). Master properties are something like 'permanent' properties which make-up the object personality. The *work* property can be considered as a temporary property which might change in value during the working process of a method, e.g. after application of the *pull* method.

# The 5 Carabao Properties

In total we have 5 properties, 4 *master properties* and the *work property*. The reader may now be noted that we always have exactly these 5 properties (fig.13), and it is a strict rule of Carabao philosophy that even for derived *carabao* classes no additional properties must be provided. This might sound strange to some readers as it seems like an essential restriction of flexibility. On the other hand this philosophy is the basis for a simplified common functionality, especially for mass storage management, and it supports easy casting from one class to another.



Fig.13 – the 5 properties of a *carabao* object

Thus the reader should be aware that there are no essential restrictions as the *par* (object parameters), *data* and *work* properties hold usually structures, so they can hold any number of values of arbitrary complexity by its structure fields. It is easy to assign an additional parameter to the object or to change a parameter. With the following statements

>> o.par.date=datestr(now,'dd-mmm-yyyy');

>> o.par.title = [o.par.title,' (',o.par.date,')'];

>> o

CARABAO object

MASTER Properties:

tag: carabao

type: shell

par:

title: 'Carabao Shell (14-Dec-2015)'

comment: {'tiny Carabao animation shell'}

date: '14-Dec-2015'

data: [1x1 carabao]

WORK Property:

opt: [1x1 struct]

arg: {1x0 cell}

figure: 1

an additional parameter *date* will be added to the object, and the *title* parameter will be changed, containing now also a date information. The parameters title and comment have some special meaning to the shell, e.g. they are displayed in the home screen.

# Package Objects

The reader might already have missed to see data information of our weird object like *t, w, x, y, z*. The reason is that our object *o* does not represent the 'weird object' that we have created by clicking *File>New>Weird*. Our object *o* is the so called *package object* of the shell which can contain an arbitrary number (including zero number) of other Carabao objects. There is a method *package* which returns a boolean result indicating whether we deal with a *package object* or not, and the convention for a package object is that the data property is of class *cell array*.

>> ispack=package(o) % equivalent to: ispack=iscell(o.data)

ispack =

1

>> class(o.data)

ans =

cell

As for package objects the data property is a list of objects (cell array row) we catch the idea now how to access our 'weird object' as a child object of the package[[2]](#footnote-2).

>> oo=o.data{1}

CARABAO object

MASTER Properties:

tag: carabao

type: weird

par:

title: 'Weird Object (11-14-7-11)'

color: [0.8929 0.8963 0.1256]

data:

t: [1000x1 double]

w: [1000x1 double]

x: [1000x1 double]

y: [1000x1 double]

z: [1000x1 double]

WORK Property:

opt: [1x1 struct]

arg: {}

var: []

The title parameter tells us that we are actually dealing with our 'weird object', we see no *comment* parameter, but a *color* parameter, and we can find now finally the missed data details *t, w, x, y* and *z* in the structure fields of the data property.

# The Tag Property

Let us start with the *tag* property. For non-expert programming level the *tag* property is not of special importance and we will not cover the tag property in later sections in more detail. For completeness, however, and for the interested reader, we will present an overview of all essential aspects of the tag property[[3]](#footnote-3) in this section.

The *tag* property holds the class name at the time of object construction. Later on we will learn more about the important concept of *object packing* (into a 'bag' structure), which is leads to a data representation free of any objects used for mass storage representations[[4]](#footnote-4). The tag information, which holds the class name, enables the Carabao method *construct* to reconstruct the object in a general way from the bag structure.

>> bag=pack(o)

bag =

tag: 'carabao'

type: 'shell'

par: [1x1 struct]

data: {[1x1 struct]}

work: [1x1 struct]

>> o = construct(carabao,bag);

The *tag* property plays also an important role for casted objects which can be re-casted into the original class using the *balance* method. E.g. there is a superclass *caracow* which can be used for demonstration of a cast.

>> cow=caracow(o)

CARACOW object

MASTER Properties:

tag: carabao

type: shell

par:

title: 'Carabao Shell (14-Dec-2015)'

comment: {'tiny Carabao animation shell'}

date: '14-Dec-2015'

data: [1x1 carabao]

WORK Property:

opt: [1x1 struct]

arg: {1x0 cell}

figure: 1

Applying the *caracow* constructor to a carabao object will cast the object to a *caracow* class. Using casting will only change the object class, but will not change the object properties. After a cast the *tag* property ('carabao') will differ from the class name ('caracow'), and we will call this object status *unbalanced*. Based on the unchanged *tag* property a common method *balance* can bring any casted object back into balanced state, which will be used internally by some Carabao methods.

>> o=balance(o) % alternatively try: o = construct(carabao,pack(o))

CARABAO object

MASTER Properties:

tag: carabao

type: shell

par:

title: 'Carabao Shell (14-Dec-2015)'

comment: {'tiny Carabao animation shell'}

date: '14-Dec-2015'

data: [1x1 carabao]

WORK Property:

opt: [1x1 struct]

arg: {1x0 cell}

figure: 1

# The Type Property

Let's spend some words on the *type* property. The object *type* is important for two reasons:

1. Object data interpretation depends on the object type. Therefore class methods have always to check the type for proper data interpretation.
2. The type determines the default method to launch a shell. We will learn more about shell launching in the next section

Splitting of object information into parameters and data should also following the rules:

1. If the interpretation of the information depends on the object type the information should be stored in the *data* property.
2. If the interpretation of the information does not depend on the object type it should be stored in the *par* (parameter) property.

Let us launch another Carabao shell

>> shell(carabao); % launch another carabao shell

and create a 'ball object' (click on *File>New>Ball*). Selecting *Plot>Show* displays a violett ball (fig. 14). Let us study the object internals, and we know already how to do this.

>> o=pull(carabao); oo = o.data{1} % pull package & access child

CARABAO object

MASTER Properties:

tag: carabao

type: ball

par:

title: 'Ball (14-Dec-2015 05:56:02)'

color: [0.7081 0.2909 0.5108]

data:

radius: 1.0069

WORK Property:

opt: [1x1 struct]

arg: {}

var: []

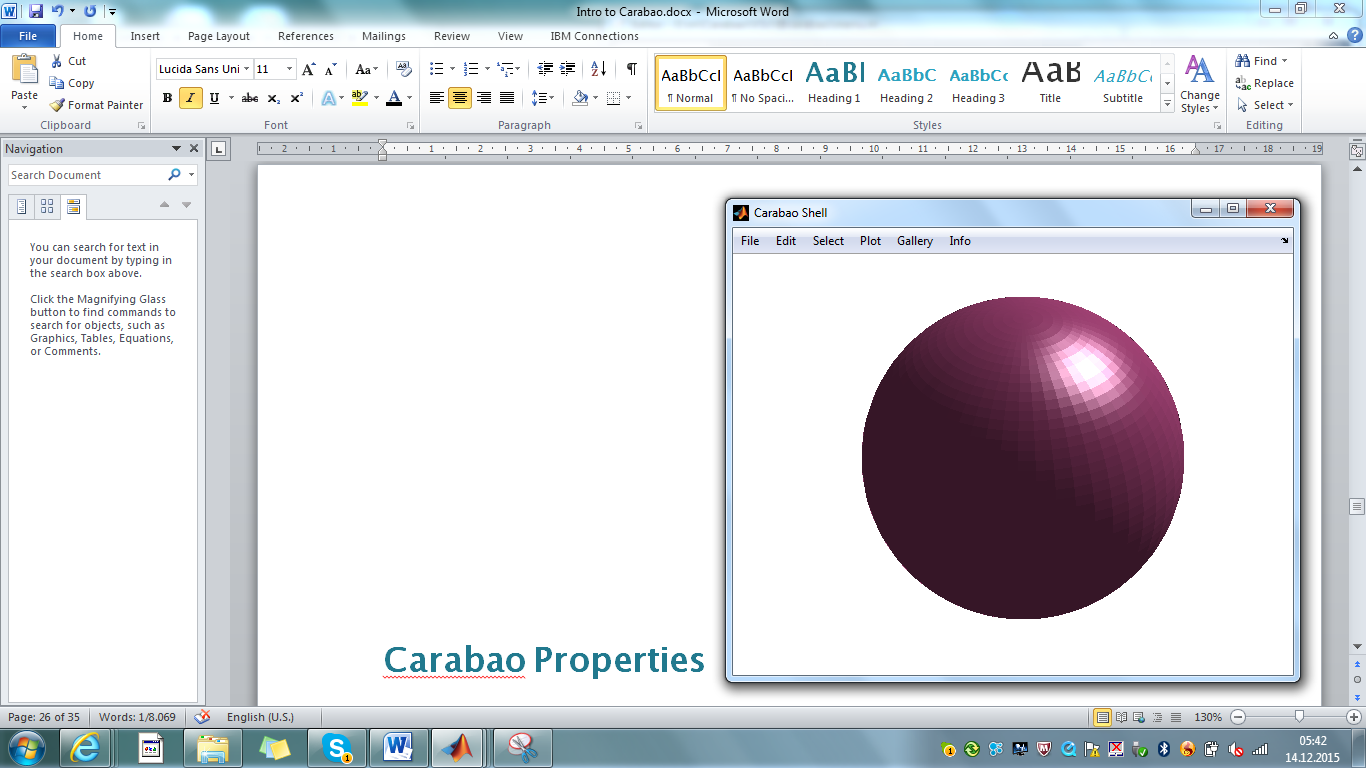


Fig.14 - A ball object, created and displayed with *File>New>Ball*, *Plot>Show*

While with the previous weird object the type had the value 'weird' the object under study has now type value 'ball', and the supported parameters are again *title* and *color*, which we can assume to have the same interpretation as with the weird object. The difference can be found in the *data* property where in contrast to data fields *t, w, x, y, z* for the weird object we have now the only data field *radius* for the ball object. It is obvious that the two data settings must be interpreted in a different way, and actually the Carabao shell's local Draw function must do some dispatching on the object type (we do not explain the details here, but the interested reader might look-up the code by invoking >> edit carabao/shell and proceeding to the local function definition of Draw).

function Draw(o,t,idx) % Draw an Object

color = get(o,{'color',[0 0 0]}); % get color (default [0 0 0])

**switch o.type**

case 'ball'

[X,Y,Z] = sphere(50);

Surf(o,X,Y,Z,color);

case 'cube'

[X,Y,Z] = cylinder([0 1 1 0],4);

Z(2,:) = Z(1,:); Z(3,:) = Z(4,:); Z = (Z-0.5)\*sqrt(2);

Surf(o,X,Y,Z,color);

case 'weird'

Weird(o,t,idx); % plot weird object

otherwise

menu(o,'Home');

return

end

end

Let's spend some words on the *type* property. The object *type* is important for two reasons:

1. Object data interpretation depends on the object type. Therefore class methods have always to check the type for proper data interpretation.
2. The type determines the default method to launch a shell. We will learn more about shell launching in the next section

Splitting of object information into parameters and data should also following the rules:

1. If the interpretation of the information depends on the object type the information should be stored in the *data* property.
2. If the interpretation of the information does not depend on the object type it should be stored in the *par* (parameter) property.

Let us launch another Carabao shell

>> shell(carabao); % launch another carabao shell

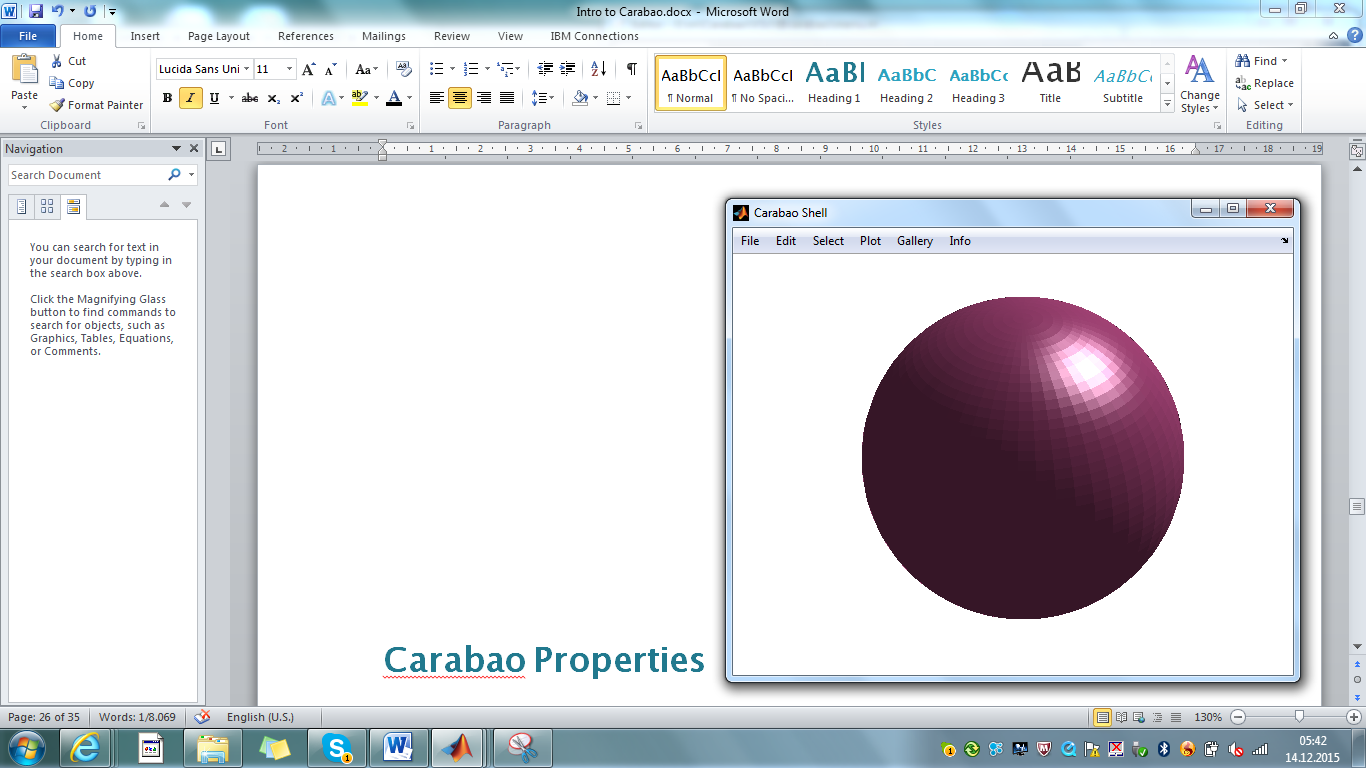
and create a 'ball object' (click on *File>New>Ball*). Selecting *Plot>Show* displays a violett ball (fig. 14).

Fig.14 - A ball object, created and displayed with *File>New>Ball*, *Plot>Show*

Let us study the object internals, and we know already how to do this.

>> o=pull(carabao); oo = o.data{1} % pull package & access child

CARABAO object

MASTER Properties:

tag: carabao

type: ball

par:

title: 'Ball (14-Dec-2015 05:56:02)'

color: [0.7081 0.2909 0.5108]

data:

radius: 1.0069

WORK Property:

opt: [1x1 struct]

arg: {}

var: []

While with the previous weird object the type had the value 'weird' the object under study has now type value 'ball', and the supported parameters are again *title* and *color*, which we can assume to have the same interpretation as with the weird object. The difference can be found in the *data* property where in contrast to data fields *t, w, x, y, z* for the weird object we have now the only data field *radius* for the ball object. It is obvious that the two data settings must be interpreted in a different way, and actually the Carabao shell's local Draw function must do some dispatching on the object type (we do not explain the details here, but the interested reader might look-up the code by invoking >> edit carabao/shell and proceeding to the local function definition of Draw).

function Draw(o,t,idx) % Draw an Object

color = get(o,{'color',[0 0 0]}); % get color (default [0 0 0])

**switch o.type**

case 'ball'

[X,Y,Z] = sphere(50);

Surf(o,X,Y,Z,color);

case 'cube'

[X,Y,Z] = cylinder([0 1 1 0],4);

Z(2,:) = Z(1,:); Z(3,:) = Z(4,:); Z = (Z-0.5)\*sqrt(2);

Surf(o,X,Y,Z,color);

case 'weird'

Weird(o,t,idx); % plot weird object

otherwise

menu(o,'Home');

return

end

end

# Pushing and Launching a Carabao Object

see

>> oo.par.color=[1 0 0]; % change color parameter of child object

>> o.data{1}=oo; % store modified child in package

>> push(o); % push package object back into shell

Let us launch another shell for our modified *carabao* object. Last time we used the method *shell* for shell launch.

# Carabao Properties

The most important things that a MATLAB® class defines are class properties and class methods. We can study the Carabao class properties.

>> properties(o)

Properties for class carabao:

tag

type

par

data

work

A Carabao object has exactly five properties which are *tag*, *type*, *par* (parameter), *data* and *work*. Whenever we derive a sub-class from Carabao this sub-class will also support these five properties. In general it is possible to provide additional properties by the sub class definition, but the concept of Carabao does not allow additional properties, and the reader shall be trusted that there is no necessity.

Let us use our Carabao object to encapsulate the data and parameters that we read from a logfile. Let us read the log data from file *data3.log* and store the x and y data to the data property, and the parameter structure par to the par property of our object.

>> [x,y,par]=read(filedialog); % select data3.log

o.data.x=x;

o.data.y=y;

o.par=par;

Typing >> o in the command line without a terminating semicolon gives us some insight about the actual property settings.

>> o

CARABAO object

MASTER Properties:

tag: carabao

type: shell

par:

title: 'DATA3'

data:

x: [1000x1 double]

y: [1000x1 double]

WORK Property:

[]

The five properties are devided into the master properties tag, type, par and data, and into the work property. The tag property holds in most of the cases the class name of the object (exceptions are not discussed in the current context).

Let us

The

>> o=carabao

CARABAO object

MASTER Properties:

tag: carabao

type: shell

par: []

data: {}

WORK Properties:

[]

Omitting the semicolon shows us some details of the Carabao's property structure.

# The Shell

Let us think about a simple shell (a graphical user interface) providing a roll-down menu *Plot* that allows us to select menu items *Plot*/*X-Stream* (to plot a sequence of red x-values) or *Plot/Y-Stream* (to plot a sequence of blue y-values), both plots showing a title which tells us the analyzed mean and standard deviation of the specific data stream. In addition we want to have a menu item *Plot/Scatter* which generates an x/y scatter plot of circles for each pair *[x(i), y(i)]* provided with a title telling us the cross correlation coefficient. There would be much more fancy functions that our phantasy could demand from our shell, but let it be enough for now with plot and analysis functions.

Let us focus even more on the *File*  roll-down menu. We want to have a menu item *File/SaveAs* which enables us to store the object to mass storage (to the file system), and finally we need a menu item File/Open to load a stored object from mass storage and to open subsequently a *shell* for the loaded object, to allows us further analysis of the data assigned with the object.

# Playing With Stuff

Let's start! Our 'process data' is currently held by two variables x and y. MATLAB®'s object oriented language structures allow us to pack related data into a single objects. An object is created by invoking a class constructor. Since we do not want to stress our phantasy at this point for a fancy class name we just decide to use the name *stuff* for our first class to play around.

There are two styles in MATLAB® to define an object class: an old style and a new style. In the new style, which we want to choose here, we might create a class folder *@stuff*, and we have to make sure that the MATLAB® path 'sees' the class folder. But on before you start creating a directory. Since we will make some evolution steps in our code development we put the code into version directories. So it would make sense to start from a root directory, say *play*, in which we have several sub directories *play/v1a*, *play/v1b*, *play/v1c* … and so on. Now you can start to create your root directory with a first version directory *v1a*. Subsequently we make our class directory *v1a/@stuff*, in which we create the class definition file [*v1a/@stuff/stuff.m*](mailto:v1a/@stuff/stuff.m) with the following contents.

classdef stuff % v1a/@stuff/stuff.m

properties % object properties

x % x data

y % y data

end

methods % public methods

function o = stuff % stuff constructor

% not much to do

end

end

end

Always keep in mind that we must not forget to set the MATLAB® path to this directory. The class definition starts with a *classdef* clause which comprises in our case a *properties* section and a *methods* section. The *properties* section defines two public properties *x* and *y*, and the *methods* section defines a function *stuff* which will be identified by *MATLAB*® as the Stuff class constructor (since the method name *stuff*' matches the class definition file name *stuff.m*. As we got now a tiny bit familiar with our class definition let us going to watch now the class constructor in operation. In order to construct a *Stuff* object we need to call the class constructor method *stuff*  and assign its return value to a variable (in our case with the simple and short name *o*).

>> o=stuff

o =

stuff with properties:

x: []

y: []

Congratulations! You just created your first stuff object. A *Suff* object created this way is very boring since it has empty properties. Let's change this situation and assign the data hold by the variables x and *y* to our properties. Typing finally the variable name o without semicolon will cause MATLAB® to display some basic info about our object.

>> o.x=x; o.y=y;

>> o

o =

stuff with properties:

x: [1x100 double]

y: [1x100 double]

# Creating a Shell Method

So good so far. We have now a constructor for our STUFF class, we know how to construct those objects and we have seen how we can assign data values to the properties. It is time now to create a 'shell' method. What do we expect from this method? First of all we expect that by invoking

>> shell(o);

a new figure will open with a menu bar that comprises two roll down menus 'File' and 'Plot' with the sub menu items 'Open ...' and 'Save as ...' for the 'File' menu, and three sub menu items 'X-Stream', 'Y-Stream' and 'Scatter' for the 'Plot' menu. Furthermore we expect that the call shell(o); manages to store the object 'o' somewhere in the shell (the user data of the shell's figure seems to be a proper candidate for such a storage location). Last but not least we expect the shell method to contain an implementation of all callback functions which could be potentially invoked. Each callback function would then recall the object 'stored in the shell', analyze the data and plot the required graphics.

This is quite some functionality that the shell method has to provide, but each of the readers who has programmed already some roll-down menu for a figure (using MATLAB's uimenu function) will not be seriously impressed by the degree of challenge for these tasks.

Let's start the implementation of a STUFF shell step by step. To add another method to the STUFF class there are two possibilities. Either write an additional function in the 'methods' section of the classdef clause or to write a separate m-file function 'shell.m' stored in the class directory 'v1a/@stuff'. We will go for the second method.

function shell(o) % setup STUFF shell (v1a/@stuff/shell.m)

fig = figure; % create new figure

set(fig,'numbertitle','off'); % no number title

set(fig,'menubar','off'); % don't need menu bar

set(fig,'name','Stuff Shell'); % show a title in figure's window bar

set(fig,'userdata',o); % push object 'into figure'

LB = 'label'; CB = 'callback';

h = uimenu(fig,LB,'File'); % add File menu header

uimenu(h,LB,'Open ...',CB,{@OpenCb});

uimenu(h,LB,'Save as ...',CB,{@SaveCb});

h = uimenu(fig,LB,'Plot'); % add Plot menu header

uimenu(h,LB,'X-Stream',CB,{@StreamCb,'x','r'});

uimenu(h,LB,'Y-Stream',CB,{@StreamCb,'y','b'});

uimenu(h,LB,'Scatter',CB,{@ScatterCb};

StreamCb([],[]); % refresh figure

function OpenCb(ho,ev) % Open callback

end % to be done

function SaveCb(ho,ev) % Save callback

end % to be done

function StreamCb(ho,ev,sym,col) % Stream plot callback

end % to be done

function ScatterCb(ho,ev) % Scatter plot callback

end % to be done

end

First note that the whole code of file shell.m is not complete. There are four callback functions OpenCb, SaveCb, StreamCb and ScatterCb which have well defined calling interfaces but no code in the body. This is intentional, as we want to implement and test the shell method step by step. Note that the main function shell is globally accessible as a STUFF method, but the four mentioned callback functions are local functions and the scope is only inside the file shell.m. For local functions we will consistently use names with leading upper case letters.

Let's test our shell method and launch the shell by invoking

shell(o);

If the code is bug free then a new empty figure will be created with a menu bar supporting both a File and Plot menu. The several menu items of the menu bar can be selected and clicked, but since the callback bodies are not yet implemented nothing essential will happen when a menu item is selected per mouse button click.The code of the shell's body is not too difficult to understand. It has three parts. The first part creates a figure and catches the figure handle in the local variable 'fig'. After that some configuration tasks are performed like disabling number title and tool bar, and setting a title in figure's title bar.

The basic configuration of the new created figure ends with an important task: 'pushing the object into the figure'. There are several methods how this can be done, but the simplest way is to store the object in the user data of the figure. It should be mentioned that an alternative method to 'push the object into the figure' is the utilization of a shelf, which allows more flexibility for future adoptions. Anyway we will not go into the details of the shelf concept at this point and will continue with the user data storage location.

The second part sets up the menu utilizing MATLABs uimenu function. The first onput argument of a uimenu call is a graphics handle to the parent object of the uimenu item. On top level the parent graphics object is the figure so we call uimenu for the File and Plot menu with the figure handle. uimenu returns itself a figure handle to the new created object which we store in the variable h to serve as the next parent object handle for nested sub menu item creation. uimenu uses property tags like 'label' or 'callback' to initialize the label or callback property of the created uimenu item. It is common technique to use short hands for these property tags (like LB and CB) to allow compacter calling syntax.

Some comments are spent for the reader who is not so familar with MATLAB's modern callback conventions. The callback is provided by a list (1xn cell array) having a function handle as first liat element followed with the actual arguments of the callback function. E.g. the statement

uimenu(h,LB,'X-Stream',CB,{@StreamCb,'x','r'});

will invoke, when activated, the callback function

function StreamCb(ho,ev,sym,col)

end

while assigning sym='x'; col='r'; Note that the first two actual arguments (ho: handle to calling graphics object, ev: event structure) will be provided by MATLAB® as a standard..

## Detailing the Callback Functions

So far by invoking the shell method (applied to some object) starts nicely a new figure and sets up a roll down menu as expected. But when we click on a nested menu item like 'Scatter' then nothing happens., since the code in the callback bodies is still missing. Let us fix this gap now, starting with the graphics callbacks.

function ScatterCb(ho,ev) % Scatter plot callback

fig = gcbf; % figure hdl of calling object

o = get(fig,'userdata'); % pull object from figure

hax = cla(fig); % clear axes & get axes handle

h = scatter(hax,o.x,o.y); % actual scatter plot

set(h,'color','k'); % black color

c = corr(o.x,o.y) % correlation coefficient

title(sprintf('Correlation: %g',c));

end

# Appendix

function shell(o) % shell for STUFF (v1a/shell.m)

LB = 'label'; CB = 'callback'; % short hands

h = Begin(o); % begin menu setup

hh = uimenu(h,LB,'Plot'); % add roll down header menu item

uimenu(hh,LB,'Scatter',CB,{@Scatter});

uimenu(hh,LB,'X-Stream',CB,{@Stream,'x','r'});

uimenu(hh,LB,'Y-Stream',CB,{@Stream,'y','b'});

End(o); % end menu setup (will refresh)

end

function fig = Begin(o) % Begin Menu

fig = figure; % open a new figure

set(fig,'menubar','none'); % no standard menubar

set(fig,'numbertitle','off'); % no number titles in figure

set(fig,'userdata',o); % push structure into fig's userdata

end

function End(o) % End Menu

Scatter; % refresh figure

end

%==========================================================================

% Menu Callback Functions

%==========================================================================

function o = Scatter(varargin) % Scatter Plot Callback

o = get(gcf,'userdata'); % access our STUFF object

cla; % clear axes

scatter(o.x,o.y,'k'); % black scatter plot

shg; % show graphics

end

function o = Stream(varargin) % Plot stream

o = get(gcf,'userdata'); % access our STUFF object

cla; % clear axes

switch varargin{3} % dispatch on mode

case 'x'

h = plot(o.x);

case 'y'

h = plot(o.y);

end

set(h,'color',varargin{4}); % set line color

shg; % show graphics

end

drink a beer – maybe a bottle of Corona - for sure with a piece of lime.

[](http://www.google.at/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0CAcQjRxqFQoTCM3f39ymmMkCFUFBlAodIYsDuA&url=http://sixfiftyml.com/why-corona-is-garnished-with-lime/&bvm=bv.107763241,d.dGo&psig=AFQjCNFZa6KH4zIJcuPiFQAQETUR8hXEOQ&ust=1447878629518763)

Fig.4 – Caramba, we had a good start!

# 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

1. Static methods are exceptions which do not require passing an object by argument – see [2] [↑](#footnote-ref-1)
2. If the context allows we will always use the variable name *o* for the package object and *oo* for the child object. [↑](#footnote-ref-2)
3. the non-interested reader might skip this section without loss of essential basics [↑](#footnote-ref-3)
4. look at the data member: the child object is also replaced by a structure representing the child's properties [↑](#footnote-ref-4)