Chapter 2
Meeting with 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 [3]. 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.

In the previous chapter we followed the challenge to build an easy-to-use MATLAB® based data analysis tool. We constructed some building blocks for file selection, log data and parameter import, graphical data display and basic statistical analysis. Everything was finally put into a single analysis tool represented as a MATLAB® m-function *analysis* which calls these building blocks in a linear sequence. Our analysis tool does this single one job perfectly, for what it has been made, however, there is no possibility for further user interaction. A lack of our tool is that it does not support encapsulation of the data, and there is no possibility for dialog driven user interaction.

Getting in Touch with Carabao

This is the right time to get in touch with *Carabao*. *Carabao* is a MATLAB® object class which supports rapid prototyping of a graphical user interface (a so called 'shell') that allows importing of 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 containers which allow to generate overviews over an ensemble of objects and to calculate statistical KPIs over the ensemble.

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 in the next chapter to prototype a data analysis tool that can deal with the kind of log data we used in the previous section.

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 to apply the method *launch* to the constructed object.

```
>> o = carabao;    % construct a carabao object
>> launch(o);      % same as: launch(carabao);
```

It is required to have a *Carabao* class version folder on the MATLAB® path. The examples presented here are based on version *Carabao V1d*, which means that the folder *.../Carabao/v1d* 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[1] *o* while the second statement `>> launch(o);` invokes the *Carabao* method *launch* [2] in order to launch a graphical shell. A new figure with a menu bar looking like Fig. 2.1 will appear on the screen.

[1] we will heavily use the variable name *o* for 'the' object variable

[2] note that the object *o* has to be passed as the first method argument [2]

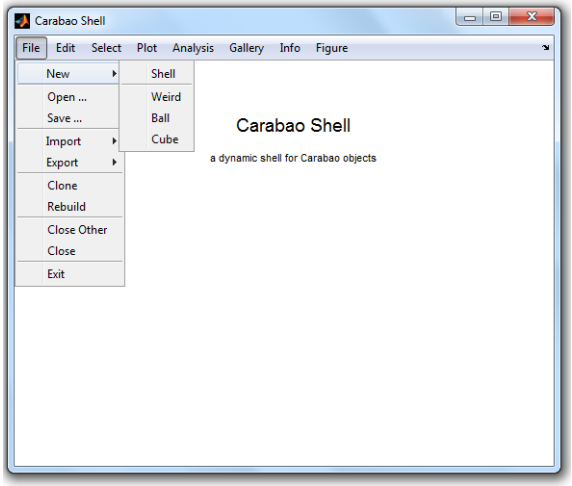

Fig. 2.1 – *Carabao* shell

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 are menu functions 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. 2.2).

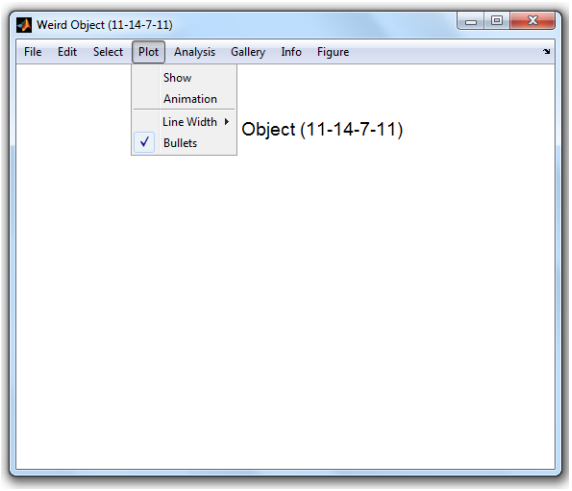

Fig. 2.2 – *Carabao* shell, holding a 'weird object'

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. 2.3). The full weirdness of the object, however, 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 impression that the spaghetti takes place on the surface of a 3D prism with a cross section shape of a ∞-character (fig. 2.4 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. 2.4 right).

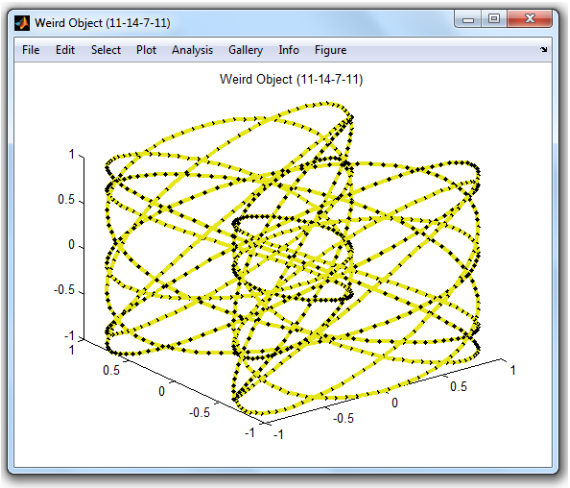

Fig. 2.3 – clicking on *Plot>Show* shows the weird object

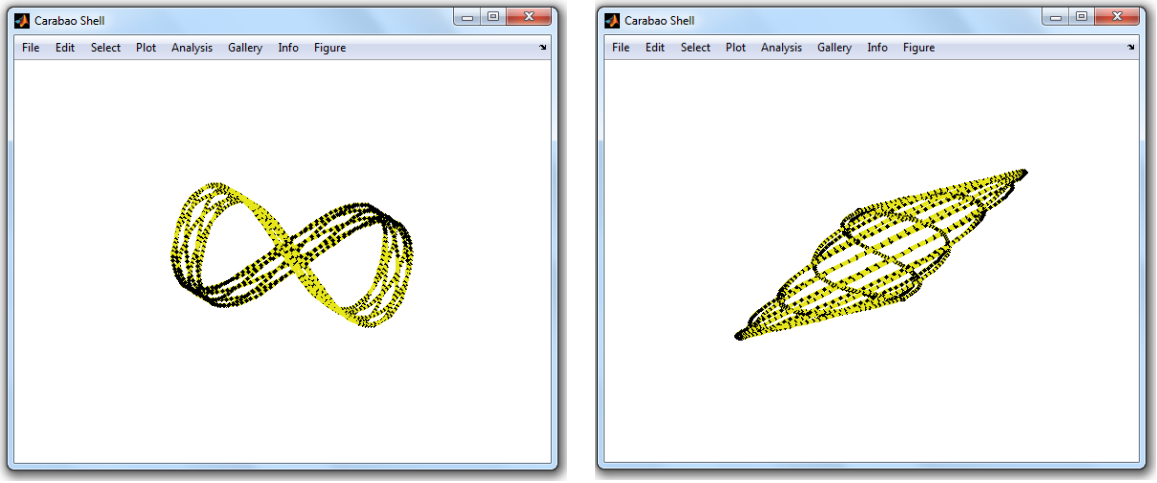

Fig. 2.4 – *Plot>Animation starts* some 'weird animation'

What is the secret behind these 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 function *shell@carabao>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 *f* (the 4 frequencies f_i which also show up in

the object's title), and the curve coordinates are built-up of *cos(ω_i t)* and *sin(ω_i t)* functions with circular frequencies $\omega_i = 2\pi f_i$.

```
function oo = Create(o)                 % Create New Object
        :          :
   function oo = Weird(o)               % Create Weird Object
        :          :
      t = (0:999)'/999;                 % time vector, 1000 points
      f = 1+round(19*rand(1,4));        % frequencies (1..20)
      w = cos(2*pi*f(1)*t);             % w vector
      x = sin(2*pi*f(2)*t);             % x vector
      y = cos(2*pi*f(3)*t);             % y-vector
      z = cos(2*pi*f(4)*t);             % z-vector
        :          :
   end
end
```

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

Creating Other Objects

If we study the *File>New* sub-menu we see other menu items for object creation. Thus we expect the possibility to create additional objects. Let's try it and create a ball object by clicking *File>New>Ball.* Selecting *Plot>Show* displays a ball (fig. 2.5 left). By clicking *File>New>Cube* a cube will show up in the shell's screen (fig. 2.5 right).

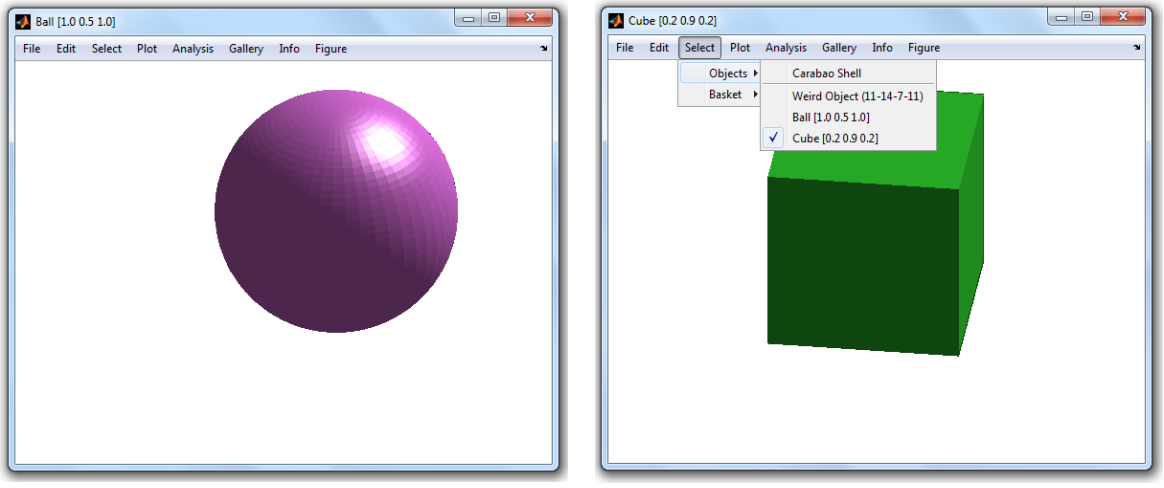

Fig. 2.5 – *ball (File>New>Ball) (File>New>Cube)*

Using the *Select>Objects* menu (fig. 2.5 right) we can switch now between our 3 objects. In the Select>Objects sub-menu there is also a menu item 'Carabao Shell'. Selecting this item will show us all objects of the shell in an overlay mode.

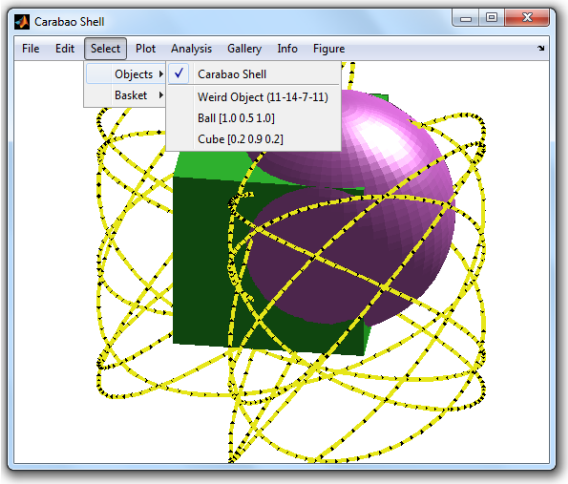

Fig. 2.6 – *Overlay of all shell objects (Select>Objects>Carabao Shell)*

Exporting and Importing Objects

The *Carabao* shell provides also a possibility to export objects into a text file and to import an object from a text file. Let us select the weird object (*Select>Objects>Weird Object (11-14-7-11*) and activate *File>Export>Text File (.txt)*. A file selection dialog appears which allows us to export the current selected (weird) object to file *weird object (11-14-7-11).txt.*

We might inspect this file with the MATLAB® editor and might find the following text lines – a bit similar to the log file format we used in the previous chapter [3].

```
$title=Weird Object (11-14-7-11)
$type=weird
$color=[0.9 0.9 0.1]
  0.000000   1.000000   0.000000   1.000000   1.000000
  0.001001   0.997608   0.087939   0.999031   0.997608
  0.002002   0.990442   0.175196   0.996126   0.990442
  0.003003   0.978538   0.261097   0.991290   0.978538
  0.004004   0.961952   0.344974   0.984533   0.961952
      :          :          :          :          :
```

Selecting *File>Import>Text File (.txt)* should allow us to import the object from the just created text file. Let's do it – you see it works, and investigating the *Select>Objects* menu we see an additional entry of our just imported object, certainly a duplicate of our weird object that we created initially (fig. 2.7).

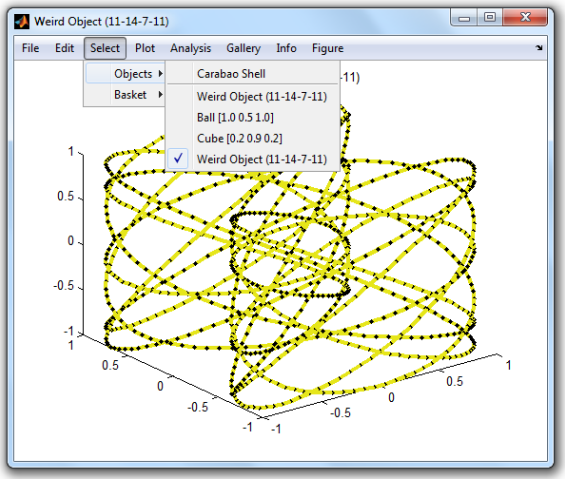

Fig. 2.7 – *After importing an object from text file*

Copy & Paste

But the *Carabao* shell provides much more powerful functionality. Let us open another shell. This would work with command `>> launch(carabao)`, but this time we choose an alternative way by clicking on *File/New/Shell*. Let us change the shell title into 'Another Shell' by selecting *Info/Shell Info>Edit …* (fig. 2.8).

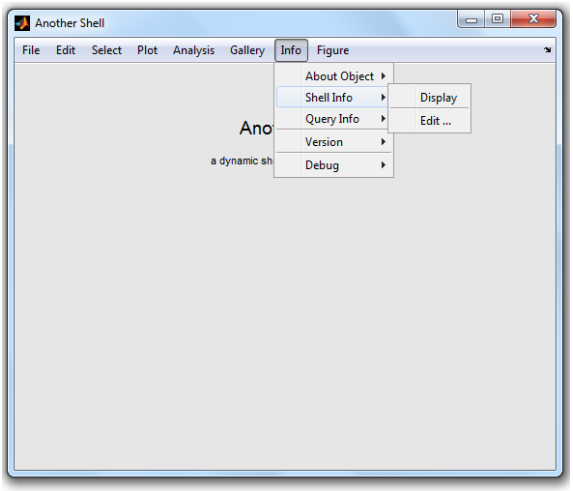

Fig. 2.8 – opening *a second empty shell*

Is it possible to copy objects from our first 'Carbao Shell' to our second 'Another Shell'? Using menu items *Edit/Copy* and *Edit/Paste* the reader should not have difficulties to perform this task. Let us copy the ball object and cube object to 'Another Shell' Finally activating *Select/Objects/Another Shell* and *Plot/Show* gives us confidence that the operation was successful.

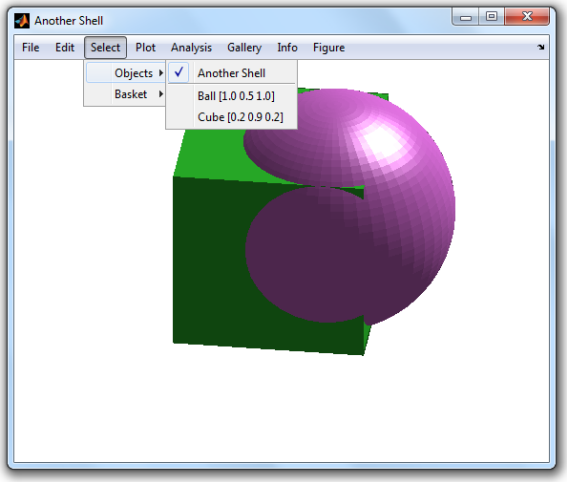

Fig. 2.9 – s*econd shell, after copy & paste*

Cutting & Clearing Objects

There is also a menu function *Edit>Cut*, and the reader may verify what he/she already expects: the cut operation will copy the selected object into the 'clip board' with a subsequent removal of the object from the shell. The cutted object might be pasted back into the same shell which allows a re-ordering of the object list, or be pasted into another shell (try it!). Repeating this operation will finally clear the shell from all objects. The second method to clear the shell from all objects is to select *Edit>Clear Objects*. Finally if the shell is selected (*Select>Objects>Carabao Shell*) an *Edit/Copy* operation will copy the whole collection of objects, which in end-effect also clears the shell[3]. The collection of copied objects is ready for pasting into the same or another shell (try it!).

[3] what really happens is that all objects in the basket are copied or cutted; selecting the shell object will collect all children objects of the shell objects into the basket

Saving and Loading the Shell

We come now to another set of powerful functions. Consider that we import some objects from text files[4] and we select an object, decide for a plot mode and choose some plot parameters, like line width or disabled bullets. Then we can save the shell including all associated objects to a .mat file, close MATLAB®, open MATLAB® again and load the shell from the .mat file. The nice thing is that after load all menu settings are the same as they have been at the time of saving, which implies that the refreshed screen of the shell will show exactly the same graphics as we had at saving time.

Let's study this, learning by doing. Let's close all open shells (*File/Exit*), open an empty shell from command line (`>> launch(carabao);`), create a weird object (*File>New>Weird*), plot the object (*Plot/Show*) and change some plotting attributes, i.e. disable bullet plotting (*Plot/Bullets*) and increasing line width (*Plot/Line Width/5*). Finally we will see the picture of fig. 2.10.

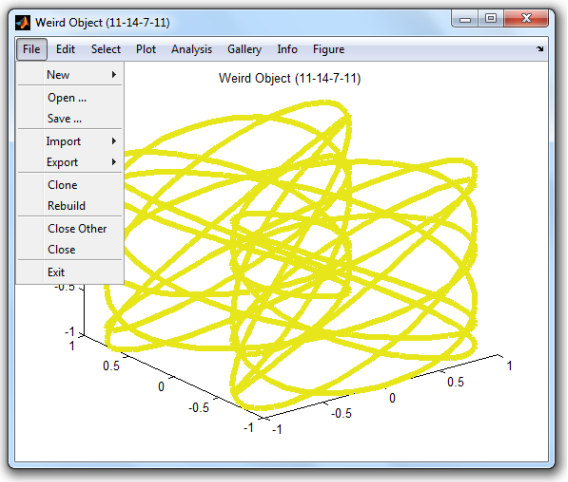

Fig. 2.10 – weird object with line width 5, no bullets

Can we save the shell, load it again and have the same menu settings and the same plot on the shell's screen? We were told that this should work. Choose *File/Save …* to save the shell to a file named e.g. *myshell.mat* and after that close the shell by selecting *File/Exit*. All figures are closed now. How can we load the shell from the .mat file?

[4] in the next section we will be able to import an object from a log data file

Easy! Open an empty shell (`>> launch(carabao);`) and select File/Open … to open *myshell.mat*. We will not only get a shell with the weird object. All the menu settings are the same as we had at saving time (line width 5, no bullets), also the graphics displayed in the shell's screen is exactly refreshed as it was at saving time, like shown in fig. 2.9. Do you have already an idea how you would implement such kind of generic mechanism?

We used an empty *Carabao* shell to launch a shell which has been saved to a .mat file. Is there another method that allows us to open a 'saved shell' from a .mat file from the MATLAB® command line interpreter? Certainly there is!

```
>> open(carabao);  % present a dialog to 'open a .mat file'
```

Cloning and Rebuilding a Shell

Once we saved a shell to a .mat file it is easy to create clones of the shell. We just have to load the shell from the .mat file two times, three times, …, as many times as you like. There is an easier way to clone a shell, using menu item *File/Clone*. And if we just changed something in the menu structure, there is an easy and efficient way to re-build the shell by selecting *File/Rebuild*. Play a bit around with these powerful functions! They all lead to the same plot as seen in fig. 2.9.

The Gallery

Finally we get now in touch with the most powerful feature of the *Carabao* shell – the gallery functionality, which includes report generation. Imagine that we are doing data analysis. Assume that we have imported plenty of log data objects and we are going to analyze the data interactively using the available set of tools that our shell provides. Some data will be very boring because we are not able to recognize interesting patterns. Other data might show interesting characteristics, and we might take screen shots, providing the screen shots with comments to have the raw materials for a concluding report.

Why so complicated? We can directly make a report using the gallery functionality of the *Carabao* shell. What … ? Yes you heard it right away, the screens that we build-up interactively can be treated like pictures that we put into a picture gallery provided with some documentation. For a completed gallery there is an automated report generator which puts the pictures of the gallery including the text documentation into a Word document!

All this is easier to understand by studying an example. Get an empty shell (e.g. clicking on *File>New>Shell* and selecting *File>Close Other*). After that change the title and the comment field of the shell by selecting *Info/Shell Info/Edit …* and entering '`Weird Object Study`' for the title and '`This is a study of 10 weird objects`'. After confirming the dialog with OK the text in the shell screen as well as the title in the figure bar is immediately refreshed. Now create 10 weird objects by repeated clicks on *File>New>Weird*. If you are done select the whole shell (*Select>Objects>Weird Object Study*) and show an overlay of all weird objects (*Plot>Show*). The fancy plot of fig. 2.11 appears on the shell's screen.

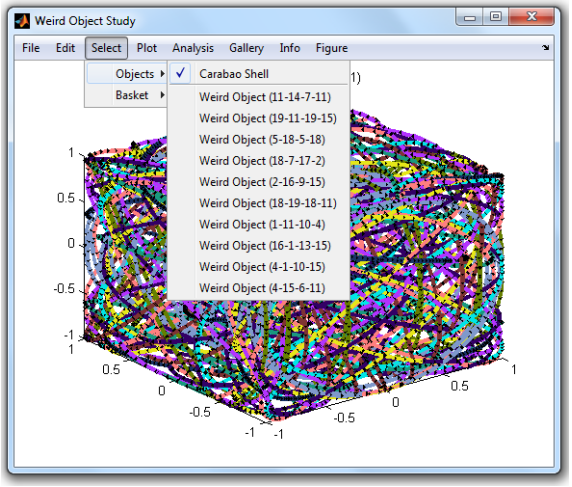

Fig. 2.11 – weird object study (10 weird objects)

How do we use the gallery. First think about how you would organize a report. We might start with some abstract which tells a potential reader with a few words what he/she might find in the report. Edit again the shell information (*Info>Shell Info>Edit …*) by adding '`Task is to select 3 of 10 weird objects with nice animation effects`' to the comment field. If you followed all the instructions carefully you should see a shell with the abstract of fig. 2.12.

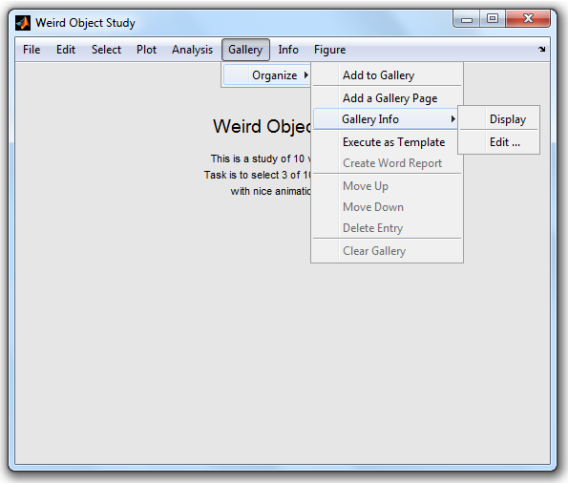

Fig. 2.12 – weird object study / abstract

We can add this screen to the gallery. Select *Gallery>Organize Gallery>Add to Gallery,* enter '`Abstract`' in the title field, clear the comment field and leave it empty and exit the dialog with OK. A new entry *Abstract* will appear in the *Gallery* menu.

Which weird object shows the nicest animation effect. I like the red one. You can make your own choice, but if you prefer to follow me click on *Select>Objects>Weird Object (2-16-9-15)*, and study the animation (*Plot> Animation*). If you are fine with the choice then stop the animation (by clicking into the shell's screen) and get back to a static plot (Plot/Show[5]). Now add the current screen to the gallery (*Gallery>Organize >Add to Gallery),* enter the text '`Red Weird Object`' into the *title* field, and '`a heap of red spaghetti, filled into cubical space`' into the *comment* field and confirm with *OK*.

Repeat the procedure by selecting the *Weird Object (1-11-10-4),* this time selecting a line width of 1 *(Plot>Line Width>1)* adding the screen to the gallery with *title* '`Black Weird Object`' and comment '`seems to be a very simple object, line width is 1`'.

Finally proceed in a similar way by selecting *Weird Object (5-18-5-18),* this time selecting a line width of 5 (*Plot>Line Width>5*) and disabling bullets (*Plot>Bullets*). After checking the animation select the static plot *(Plot>Show) and add the screen* to

[5] for a proper report generation always change from animation mode to a static plot mode, before you add a screen to the gallery

the gallery with title '`Blue Weird Object`' and comment '`Looks like a coil, but transforms into a piece of a fence`'.

Let's finally close the report by adding some kind of conclusion! Select *Gallery>Organize>Add a Gallery Page*, enter '`Conclusion`' into the title field and write some concluding remarks into the comment field: '`concluding remarks: very hard to say which objects show the best animation effects`'.

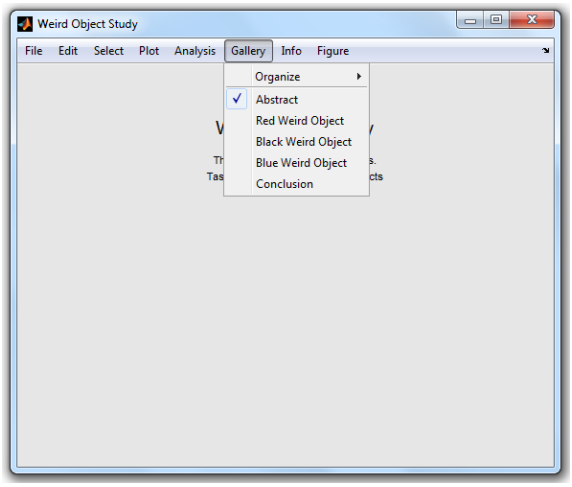

Fig. 2.13 – dressed-up gallery with 5 screens

Enough for now. We dressed-up the gallery with five screens. Click on Gallery>Abstract. The abstract screen will come into sceene (fig. 2.13). Save the shell (*File>Save …* and choose *weirdstudy.mat* for the file name) and close all figures (*File>Exit*). Whenever we load the shell again from .mat file (`>> open(carabao);`) we get the full gallery, and the gallery is like a report, telling us the most important findings and conclusions of our study. Click through all the main menu items of the Gallery menu: the shell screens that we stored in the gallery will show-up again with all the shell settings restored that are necessary to refresh a graphics on the shell's screen.

Report Generation

The 'gallery report' is an incredible feature as we can recall each stored screen and immediately go ahead to change any shell setting, e.g. we can recall the 'Blue Weird Object' (*Gallery/Blue Weird Object*) an re-enable the (disabled) bullets (*Plot/Bullets*).

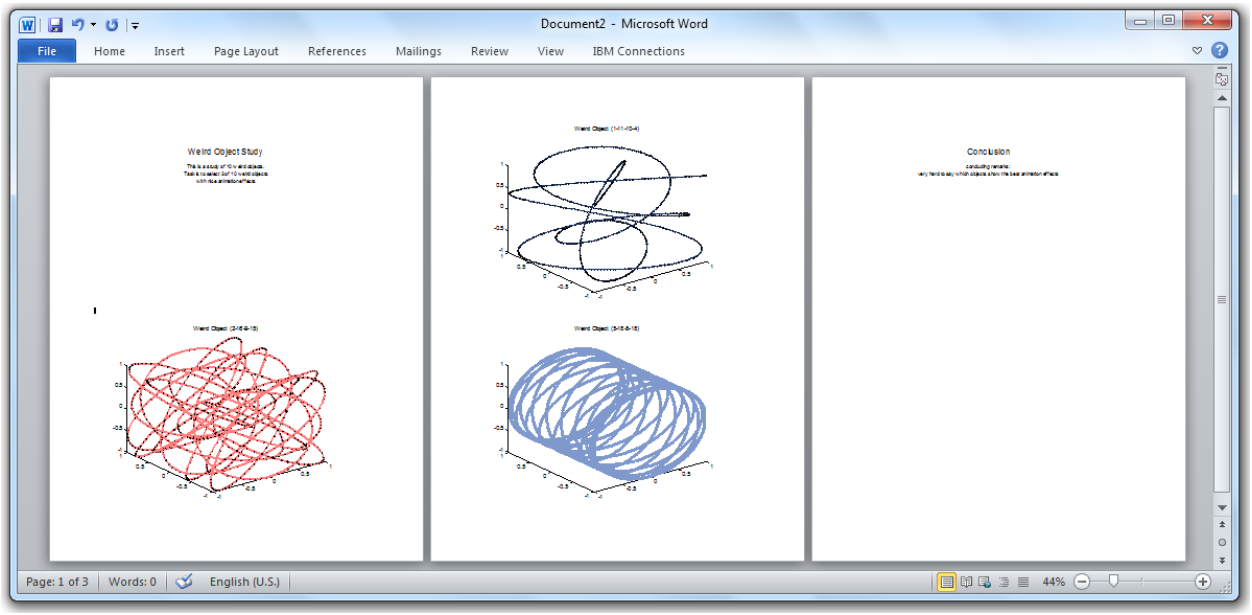

Fig. 2.14 – automated Word document report

In addition to this 'living report' feature we can also generate a traditional (static) report based on an automated Word document. Clicking on *Gallery>Organize>Create Word Report* will generate the Word document of fig. 2.14.

The Basket

There is a further powerful concept supported by the *Carabao* shell – the *basket*. We can imagine the basket as an abstract concept (fig. 2.15) which allows us to collect objects according to some collection criteria which can be set interactively in the menu or programmatically by object options.

Fig. 2.15 – Abstract Concept of a Basket

Consider that we do some statistical analysis regarding an ensemble of objects of specific class and type, which by proper menu settings are collected in the *basket*. Our statistical analysis function retrieves then a list of all objects in the basket and applies the analysis to the collection of those basket objects, excluding all other objects. Let us demonstrate this concept with a practical example. Close all figures, launch a *Carabao* shell, create three weird objects and three ball objects and then *Plot>Show*. The last created object is titled 'Ball [0.9 1.0 0.9]', which becomes the current selected object.

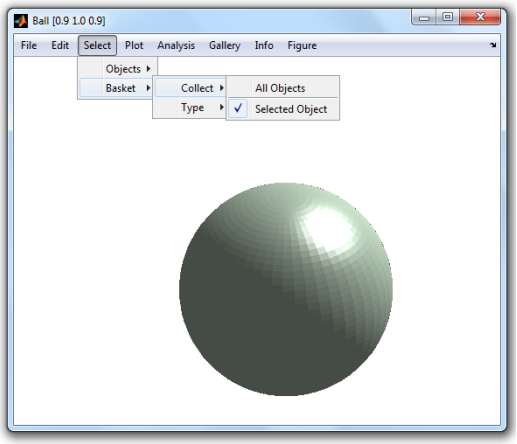

Fig. 2.16 The *Select>Basket>Collect* menu

Now have a look on the *Select>Basket>Collect* menu. The current selection of this menu is *Selected Object* (fig. 2.16). that's why the *Show* function displays only the current selected object. The Show function is implemented to show all objects in the basket, thus we may conclude that the basket contains only this single object.

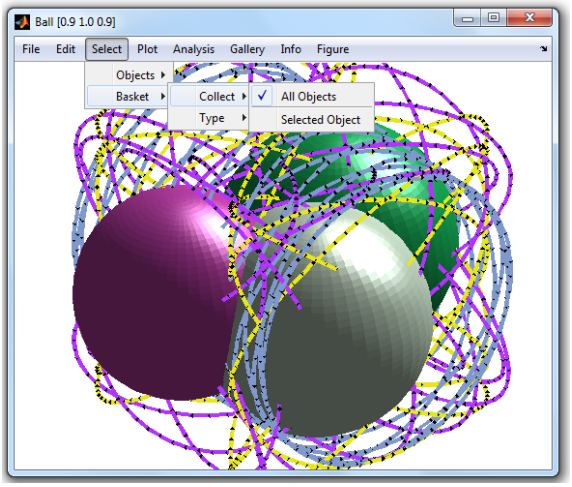

Fig: 2.17 – Showing all objects (*All Objects* selected)

Now select menu item *Select>Basket>Collect>All Objects,* which will cause to collect all objects assigned with the shell into the basket, independent of the current object selection[6]. All objects assigned with the shell are now shown on the screen. (fig. 2.18).

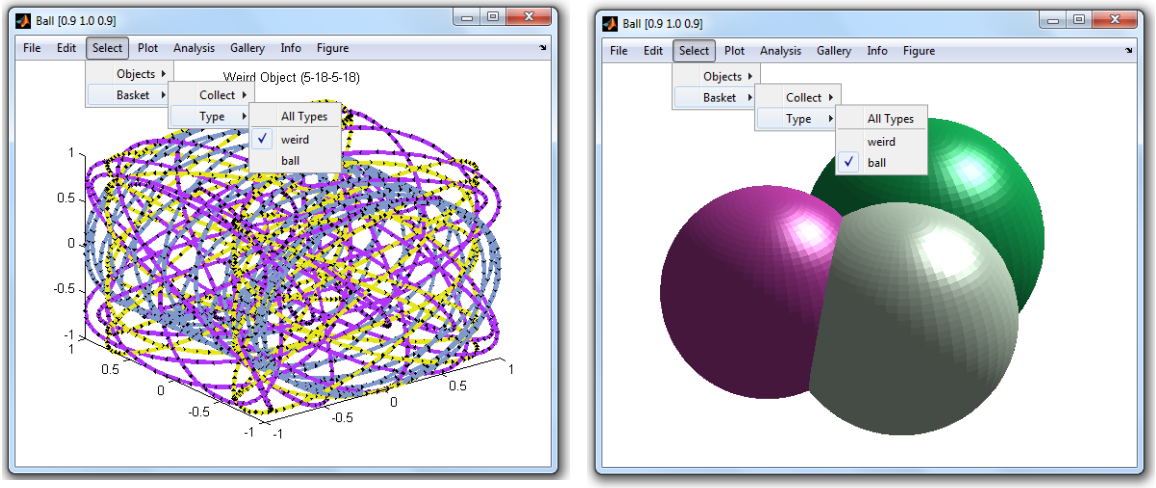

Fig: 2.18 – Collecting either weird or ball objects into the basket

Based on the list of basket objects the *Carabao* shell has three analysis functions (menu *Analysis*), each collecting a different set of objects into the basket (all *weird* typed, all *ball* typed, all *cube* typed) and plotting from those ensembles some statistical overview[7] (fig. 2.19).

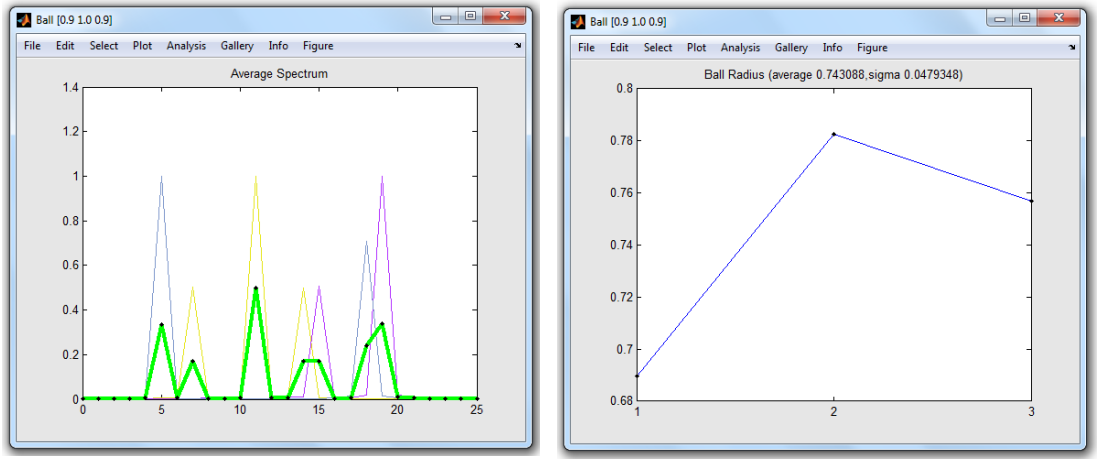

Fig: 2.19 – *Analysis>Weird Frequency* and *Analysis>Ball Radius*

[6] we have earlier seen that the selection of the shell object also leads to a display of all objects of the container. Selecting the shell object also means collecting all objects into the basket

[7] the three types of overview graphics are mainly for demonstrational and less practical use

The basket concept splits a task into two parts. The first part allows the user to collect objects interactively into the basket by proper choice of shell settings. Alternatively the collection can also be programmatically, as it is the case with the implementations of the analysis functions of fig. 2.19. The second part is just getting a list of objects from the basket and performing a proper operation on this list of objects.

No Espresso at this Time

Plenty of cool features – but now our head is full. Our mind needs a break, needs some escape! No espresso at this time! Maybe a short walk in fresh air is the right activity which gives our mind a proper environment for relaxation.

Fig 2.20 – no espresso, but a short walk in fresh air

A Short Review

So far we got in touch with *Carabao*, or more precisely, in touch with the *Carabao* shell. What are the coolest features of the *Carabao* shell? I want to mention at least seven of them.

- **Interactive graphical shell**: the *Carabao* shell can hold one or more objects, and there are menu items which allow to invoke operations that act on those objects in order to show information about the objects or to plot some graphics. Besides of object parameter and data there are independent *shell settings* which act as parameters for the selected operations.
- **Copy & paste**: the objects of a shell can be transferred between different shells with copy & paste functions.
- **Gallery and report generation**: The *Carabao* shell supports a *gallery* which can store actual graphic screens, including some text documentation. The gallery itself can act like a living report which allows to *recall* plots that can be immediately modified and tuned by tweaking shell settings, either via the menu or from program level.
- **Basket concept**: There are shell menu settings which allow to setup specific selection criteria that determine which objects are collected into the basket. Plot or analysis functions operate then on this list of basket objects.
- **Mass storage management**: There are *export* and *import* functions which allow to export/import an object to/from a text file. Besides of object export/import the whole shell can be *saved* to a .mat file and loaded from there. The .mat file not only holds the object information, it also stores the shell settings, as well as the gallery, which are restored by the *open* operation.
- **Refresh & rebuild mechanism**: There is a sophisticated *refresh* & rebuild mechanism. This refresh mechanism allows to reconstruct a graphics plot after opening a .mat file, after shell cloning and *rebuilding* the shell, after selecting a *gallery* menu item and for report generation. The rebuild mechanism rebuilds the menu structure in order to match current settings
- **Building blocks**: the *Carabao* shell is made-up of building blocks and any of those building blocks can be picked for a specific application.

So far we learnt a lot about *Carabao* shell functionality, but except from a few command lines that allowed us to launch a *Carabao* shell or to open a .mat file we did not really get in touch with *Carabao* properties and *Carabao* methods.

Definitely there would be a lot to tell about *Carabao* properties and methods, but by the time it is better to stay focused on a short overview of the properties and to get in touch with a small subset of the *Carabao* methods, in the sense "less is more".

Next Focus – Next Mission

A good approach to stay focused on "less is more" is to challenge ourselves with a further mission. Even we do not want to dive into a sea of details it would be interesting to understand how we can get an object out of the shell, manipulate it and push it back into the shell. This is exactly what we need to understand for our next mission, which gives us the scope for the rest of this chapter.

> **Mission #3**: launch a shell and create a *weird object*, a *ball object* and a *cube object*. Change the color of these objects in order to get a red weird, a green ball and a blue cube object.

It needs to be mentioned that there is no menu item of the *Carabao* shell that allows us to change the color of an object. Thus we have to solve this mission programmatically. The first part seems to be easy. With

```
>> launch(carabao);
```

we get a new *Carabao* shell. After launch we just have to repeat what we have done earlier: Create a weird object (*File>New>Weird*), a ball object (*File>New>Ball*) and a cube object (*File>New>Cube*). That is easy! But how can we change the color of those objects into red, green and blue? We have to get access to all of those objects, assign them to object variables and study their properties.

Pulling the Shell Object

There is always an object assigned to the shell, which is called the *shell object*. Can we see some details of the shell object? Certainly we can – using the *pull* method. Be aware that for every application of a method[8] we need some object which is passed to the method by argument[9]. If we have no object we can simply use a *Carabao* constructor (method *carabao*) to return a *Carabao* object.

```
>> o=pull(carabao)
CARABAO object
 MASTER Properties:
  tag: carabao
  type: shell
  par:
      title: 'Carabao Shell'
    comment: {'tiny Carabao animation shell'}

  data:     [1x1 carabao]    [1x1 carabao]    [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 shell object from the shell'[10] and assigns it to variable *o*. Since we did not provide a semicolon terminator the MATLAB® command interpreter displays some object internals. The first line `CARABAO object` gives us confidence that the object is really of class `'carabao'`. We can use built-in function *class* to cross check the object's class name.

```
>> class(o)
ans =
carabao
```

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 during the working process of a method, e.g. during the pulling process of the *pull* method, where temporarily options and a figure handle are assigned to the *work* property.

---

[8] Static methods are exceptions which do not require passing an object by argument – see [2]

[9] usually the object is the first input argument of the method, but MATLAB® allows exceptions

[10] actually we pull a copy of the shell object – the shell object is still assigned with the shell

The 5 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
```

In total we have 5 properties, 4 *master properties* (*tag, type, par, data*) and the *work property*. I have now an important message to you: For any *Carabao* object and for any derived *Carabao* object we always have exactly these 5 properties (fig. 2.21).

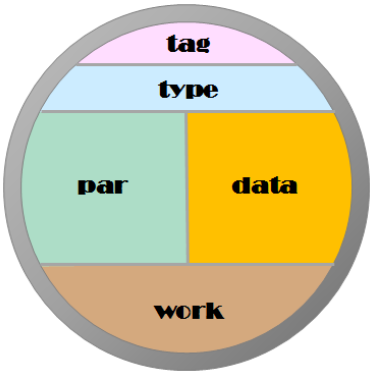

Fig. 2.21 – the 5 properties of a *Carabao* object

It is a strict rule of *Carabao* philosophy that for any derived *carabao* class 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.

Thus you should be relaxed and should realize that in practice 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[11].

[11] if even this does not convince you I suggest you use specific objects as part of the properties

Here is a short description of the 5 *Carabao* properties:

- *tag*: holds the initial class name after object construction, and is used to reconstruct the original object after packing or casting[12]
- *type*: determines data interpretation, and determines the default method for launching a shell
- *par*: object parameters, i.e. all permanent object information which has not type dependent interpretation
- *data*: object data, i.e. all permanent object information which has type dependent interpretation
- *work*: temporary information carried by the object (figure handle, options, callback arguments, working variables, GUI element handles, error messages, …)

It is easy to access object properties, as all object properties are public, and using the following syntax

```
>> title = o.par.title;            % get title parameter
>> o.par.title = 'My new title';   % set title parameter
```

MATLAB® is able to execute these access methods with very high performance[13].

Container 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 shell object *o* does not represent the 'weird object' that we have created by clicking *File>New>Weird*. Our shell object *o* is the so called *container object* of the shell which we provided when we launched the shell. Basically a *container object* can contain an arbitrary number (including zero number) of other *Carabao* objects as children, and whenever we created a new (weird/ball/cube) object the new object has been added to the children list of the *shell object*. There is a method *container* which returns a boolean result indicating whether we deal with a container object or not.

[12] an introduction to packing and casting will follow soon

[13] about 4 µs for *get* and 15 µs for *set* on my computer

```
>> ok=container(o)  % equivalent to: ok=iscell(o.data)
ok =
     1
>> class(o.data)
ans =
cell
```

The criteria for a container object is that the data property must be of class *cell array*, and this cell array represents exactly the container's list of children. Since for container objects the data property is a list of objects (cell array row) we should be aware now that our weird/ball/cube objects can be accessed by indexing a child object of the shell object's data list[14].

```
>> oo=o.data{1}
CARABAO object
 MASTER Properties:
  tag: carabao
  type: weird
  par:
    title: 'Weird Object (11-14-7-11)'
    color: [0.9000 0.9000 0.1000]
  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', and the *type* property value '`weird`' confirms this fact. And look – here is also the *color* parameter. Finally we can find the missed data details *t, w, x, y* and *z* in the structure fields of the data property. Let's have a look on the second child object.

```
>> oo=o.data{2}
CARABAO object
 MASTER Properties:
  tag: carabao
  type: ball
  par:
    color: [1 0.5000 1]
    title: 'Ball [1.0 0.5 1.0]'
```

[14] If the context allows we will always use the variable name *o* for the shell object and *oo* ("object's object") for a shell object's child object.

```
 data:
   radius: 1.0744

WORK Property:
   opt: [1x1 struct]
   arg: {}
   var: []
```

The type property has the value 'ball' which matches our expectation. And we also find a color parameter. A quick look on the third child object shows us a similar result, with *type* property '`cube`' and again with a *color* parameter.

```
>> oo=o.data{3}
CARABAO object
 MASTER Properties:
  tag: carabao
  type: cube
  par:
    color: [0.2000 0.9000 0.2000]
    title: 'Cube [0.2 0.9 0.2]'

  data:
    radius: 1.2635

 WORK Property:
    opt: [1x1 struct]
    arg: {}
    var: []
    var: []
```

Changing the Color Parameters

We should know enough to complete our mission. All objects of the container have a *color* parameter. The commands we have to invoke should look as follows:

```
>> oo=o.data{1};          % get the weird object
>> oo.par.color=[1 0 0]; % change to red (RGB color code)
>> o.data{1}=oo;          % store weird object back into shell object

>> oo=o.data{2};          % get the ball object
>> oo.par.color=[0 1 0]; % change to green (RGB color code)
>> o.data{2}=oo;          % store ball object back into shell object

>> oo=o.data{3};          % get the cube object
>> oo.par.color=[0 0 1]; % change to blue (RGB color code)
>> o.data{3}=oo;          % store cube object back to shell object
```

All objects have been changed and stored back into the shell object! But how do we get the modified shell object back into the shell?

Pushing a Shell Object

The reader might already have guessed it. In analogy to the *pull* method (to get an object from the shell) there is also a *push* method which pushes the object back to the shell. If the object came already from the shell the following syntax is a proper one.

```
>> push(o)   % push object back to the shell
```

The *push* method replaces the current shell object. We can verify if our change of colors was effective.

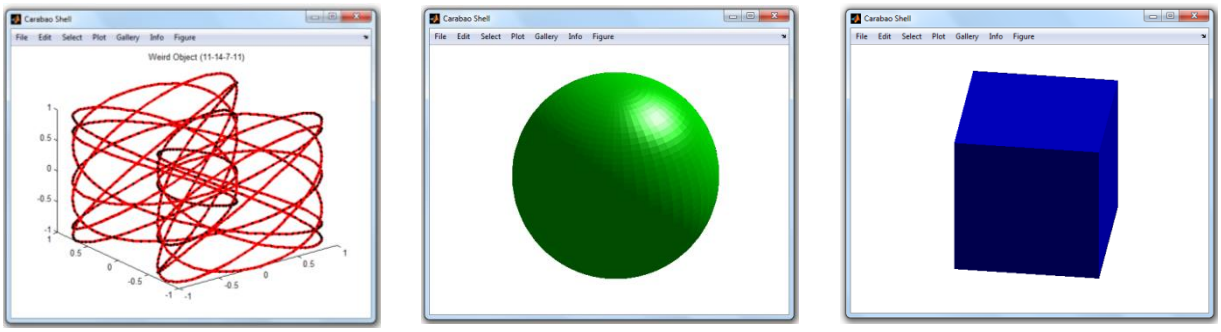

Fig. 2.22 – red weird object, green ball and blue cube

Select the weird object (*Select>Weird Object (11-14-7-11)*) and show the plot (*Plot>Show*). The weird object which was formerly yellow has now a red color. In the same way we can bring the green ball and the blue cube into view (fig. 2.22). Well done – our mission is complete!

Lessons Learnt

What did we learn so far about *Carabao* objects and *Carabao* methods?

- Using the *Carabao* constructor method we can construct a new *Carabao* object.

```
>> o = carabao;          % construct a carabao object
```

- Applying the *launch* method we can launch a graphical shell. The object that is passed to the shell is called the *shell object* and acts as a container object

```
>> launch(o);            % launch a default shell
```

- The shell object can be retrieved by pulling it. It can be stored back to the shell by pushing it.

```
>> o = pull(carabao);    % pull shell object from shell
>> push(o);              % push shell object back to shell
```

- Each *Carabao* object has exactly five properties: tag, type, par, data and work. Properties and property fields can be accessed in analogy to structure and sub-structure field access.

```
>> title = o.par.title;            % get title parameter
>> o.par.title = 'My new title';   % set title parameter
```

- A container object is characterized by a data property of class *cell array*, and method *container* tells us whether we deal with a container object or not.

```
>> iscont=containter(o);  % equivalent to: iscont=iscell(o.data)
```

- Getting a container's child object out of the container, we can change the properties of a child object.

```
>> oo=o.data{1};              % get 1st child object
>> oo.par.color=[1 0 0];      % change color parameter
>> o.data{1}=oo;              % store object back to shell object
```

- To open a shell from a .mat file we can invoke the *open* method from the command line which will pop-up a file selection dialog to open a .mat file.

```
>> open(carabao);            % open a shell from a .mat file
```

Mission #3 completed – lessons learnt! Now we have some idea about *Carabao*. How can we utilize the *Carabao* functionality for our log data analysis tool? This is the objective of the next chapter, and before we start with the next chapter I would say "time for espresso"!

Fig. 2.23 – time for espresso

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

[3] Wikipedia "Carabao"