Chapter 3
Time For Espresso

"The popularity of espresso developed in various ways; a detailed discussion of the spread of espresso is given in (Morris 2007 [3]), which is a source of various statements below." … "The words express, expres *and* espresso *each have several meanings in English, French and Italian. The first meaning is to do with the idea of "expressing" or squeezing the flavour from the coffee using the pressure of the steam. The second meaning is to do with speed, as in a train.[1]" [4] Making an Espresso with my coffee maker takes me less than 30 seconds. Making an Espresso object with the Carabao rapid prototyping tool takes me also less than 30 seconds.*

We learnt now a lot about the *Carabao* shell's functionality and something about the *Carabao* properties and methods. We also understand that with an object oriented programming approach common functionalities are identified and implemented in a base class [2]. For specific applications a new class is being derived from the base class which inherits all its properties and methods – meaning, all the functionality of the base class.

Is it possible to extend our log file analysis tool from chapter 1 to a menu based dialog driven analysis tool which supports all the fancy features of the *Carabao* shell, like mass storage (save and load), data importing and exporting, shell cloning, package

[1] the German Wikipedia version of espresso claims that the association of 'espresso' with 'express' in the sense of 'fast' is a misunderstanding of many people as 'express' has to be understood in the meaning of expressing emotions

management, copy & paste, gallery functionality, report generation, … ? Certainly – that's the reason why *Carbao* has been made!

According to the object oriented approach we would have to build a new class derived from the *Carabao* base class, which inherits all the *Carabao* functionality. We would call such kind of object class a strong class with powerful functionality. If the creation process of such a strong class would be fast and easy, like making a strong espresso on modern coffee makers, say in less than 30 seconds – well, this would be fantastic. Such kind of object class could carry the name *Espresso* – strong, fast and easy.

Making Espresso

Let's make an *Espresso* class, deriving it from *Carabao*. The steps are:

- launch the *Carabao* rapid prototyping shell, providing the new class name `'espresso'`
- select a path directory and confirm to create the new class

That's it. I can do it in less than 30 seconds[2]. But don't rush! Take your time now and try to catch everything important. Later on you can try yourself to do the class creation in less than 30 seconds. I'm sure you can do it. Let's go!

```
>> rapid(carabao,'espresso')
```

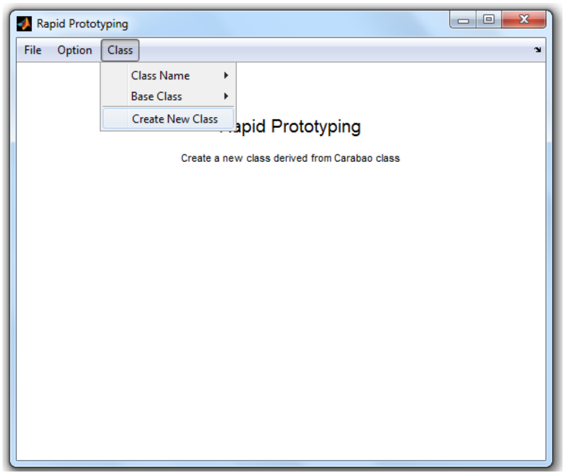
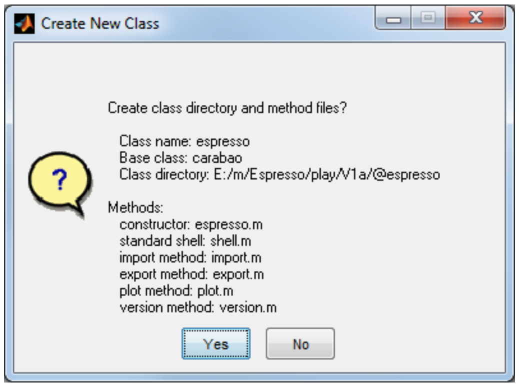


Fig. 3.1 – Carabao rapid prototyping shell

[2] I even can do it in less than 20 seconds

The *Carabao* rapid prototyping shell opens. All options are already setup properly. As we can investigate (menu item *Options*) the new class will have the methods *shell* (standard shell), *import* (general import method), *export* (general export method), *plot* (general plot method) and *version* (return version number, release notes). The class name `'espresso'` (see *Class>Class Name*) has been taken from the second command line argument, and the base class `'carabao'` (see *Class>Base Class*) has been initialized on base of the first argument's class name.



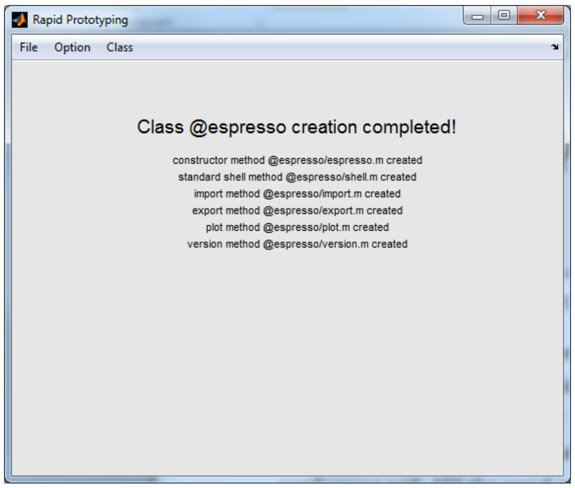
Create class directory and method files?

Class name: espresso
Base class: carabao
Class directory: E:/m/Espresso/play/V1a/@espresso

Methods:
constructor: espresso.m
standard shell: shell.m
import method: import.m
export method: export.m
plot method: plot.m
version method: version.m

Fig. 3.1 – "Create New Class" confirmation dialog

We are ready to create our new *Espresso* class, so let's start creation process with *Class/Create New Class*. A dialog opens to "Select a parent folder for new class folder @espresso". Select the folder *play/v1a* and exit the dialog with the *Select Folder* button. The confirmation dialog of fig. 3.1 pops up. After cross checking and confirming with *YES* the rapid prototyping machinery starts to work and ends up finally with the screen text of fig. 3.2.



Class @espresso creation completed!

constructor method @espresso/espresso.m created
standard shell method @espresso/shell.m created
import method @espresso/import.m created
export method @espresso/export.m created
plot method @espresso/plot.m created
version method @espresso/version.m created

Fig. 3.2 – Class espresso creation completed!

As class creation is complete, we want to play now a little bit with an Espresso object. Try the following commands:

```
>> o=espresso   % construct an Espresso object
ESPRESSO object
 MASTER Properties:
  tag: espresso
  type: shell
  par: []
  data: {}
 WORK Property:
  []
>> launch(o);   % launch an Espresso shell
```

A new shell with title 'Espresso Shell' opens (see fig. 3.3) – perfect!

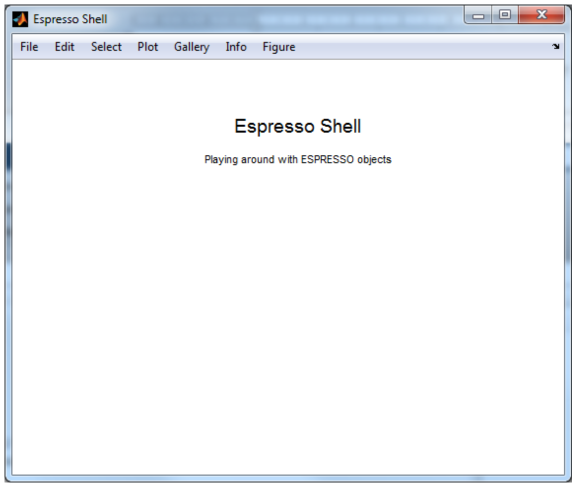


Fig. 3.3 – *Espresso* shell

Espresso Properties and Methods

We learnt already about the *Carabao* properties – there are exactly five, and *Carabao* philosophy advises us strictly that we are not allowed to add further properties for derived classes like Espresso.

```
>> properties(espresso)
Properties for class espresso:
    tag
    type
    par
    data
    work
```

What are the methods?

```
>> methods(espresso)
Methods for class espresso:

add         clean      export    menu      perform  server   util
animate     clip       figure    message   plot     set      var
arg         cls        fselect   mhead     prop     setting  version
background  construct  gallery   mini      provide  shelf    what
balance     control    gcf       mitem     pull     shell    word
basket      current    get       mseek     push     shg      work
call        data       import    open      rapid    shit
cast        directory  inherit   opt       rebuild  spin
charm       display    launch    pack      refresh  text
check       dynamic    load      container remove   type
choice      espresso   manage    paste     save     upath

Static methods:

color       either     iif       master    ticn     trim
cuo         gfo        is        rd        tocn
```

Holy carabao shit! That's a lot - more than 80 methods, and we got all that stuff in less than 30 seconds!

Manual Import of Log Data

Before we add code to *Espresso* methods let us first understand how to import one of our log files. In chapter 1 we created m-file functions *filedialog* and *read*, which can be used to select a log file and read the log file data.

```
>> path=filedialog;             % e.g. select data1.log
>> [x,y,par]=read(path);        % read data & parameters of data1.log
```

After reading the data we have to store data and parameters into the properties of an object. Based on the lessons learnt in last chapter we have already an idea how we can do that.

```
>> oo = espresso;               % create an Espresso object
>> oo.data.x = x; oo.data.y = y; % set object data
>> oo.par = par;                % set object parameters
```

Let us display some object details. What we see makes quite sense.

```
>> oo                              % display some object details
ESPRESSO object
 MASTER Properties:
  tag: espresso
  type: shell
  par:
    title: 'DATA1'

  data:
    x: [1000x1 double]
    y: [1000x1 double]

 WORK Property:
  []
```

Pasting an Object

How do we get the object back to the shell. Also this task can be performed based on our lessons learnt.

```
>> o=pull(espresso);     % pull shell object from active shell
>> o.data{1}=oo;         % store imported object in shell object
>> push(o);              % push shell object back to shell
```

Now we expect to see the object in the object list (click *Select>Objects*). Oops – the new object that we added to the shell is not visible in the menu. Maybe some refresh of the shell menu is additionally necessary? Bingo – this is exactly the reason why we did not see the imported object in the *Objects* menu list.

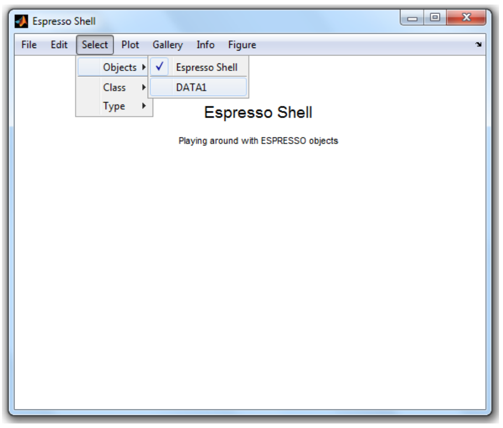


Fig. 3.3 – Espresso shell with one child object

We can click the *File>Rebuild* [3] menu item to rebuild the menu structure. Finally the new object shows-up in the object list as *Select>Objects>DATA1* (fig. 3.3). There is an easier method to bring a list of new child objects into a shell and this method is called *paste*. Let's clear the shell (*Edit>Clear Objects*, confirm with *YES*) and paste the object *oo* enclosed in a list.

```
>> paste(carabao,{oo});  % paste list with 1 child object into shell
```

The *paste* method does a perfect job, as it finally rebuilds also the *Select>Objects* menu properly (see fig. 3.4).

Log Data Import

When we investigate the *File>Import* menu of the Espresso shell we discover a *File>Import>Log Data (.log)* menu item. By selecting this menu item a file selection dialog opens which allows us to select a log data file, but when we close the file selection with OK a weird object will be created and added to the shell. Obviously this menu item is some dummy menu item which waits for our modification in order to get a proper behavior (importing log data from a .log file, creating an object and pasting it into the shell).

The code for object import can be found in method espresso/import. Let's have a look on this code with `>> edit espresso/import`. The method consists of the main function *import* and a local import driver function *LogData*.

```
function list = import(o,driver,ext) % Import ESPRESSO Object From File
   caption = sprintf('Import object from %s file',ext);
   [files, dir] = fselect(o,'i',ext,caption);

   list = {};                         % init: empty object list
   gamma = eval(['@',driver]);        % import driver function handle
   for (i=1:length(files))
      oo = gamma(o,[dir,files{i}]);   % call import driver function
      list{end+1} = oo;               % add imported object to list
   end
end
```

[3] note that we distinguish between the terminology "rebuild" and "refresh"; we talk of "refresh" if the shell's screen has to be refreshed, and of "rebuild" if the menu has to be re-built

```
function oo = LogData(o,path)            % Import Driver For .log File
   fid = fopen(path,'r');                % open log file for write
   if (fid < 0)
      error('cannot open export file');
   end
   % add code for import                 % put your own code here
   fclose(fid);
   oo = shell(o,'Create','weird');       % put your own code here
end
```

Method *import* needs to be called with three input arguments (e.g. `>> list = import(o,'LogData','.log')`). It starts with a caption generation for the following file selection dialog, which is opened in multi selection mode[4]. The result of the file selection is returned by two output arguments, a file list (*file*s) and the according folder path (directory).

As a next step a function handle for the local import driver function is created and assigned to variable *gamma*[5]. In the following the file list is processed in a for-loop, and for every file in the file list the local import driver function (*gamma* - holds the value *@LogData*) will be called in order to perform the actual import. The imported objects are finally collected in a *list* which is returned as output arg. It is then the task of the the *import* method's calling function to *paste* the *list* of objects into the shell.

The code which we have to focus is obviously the local import driver function *LogData*. With the code generated by the *Carabao* rapid prototyping tool we get only some dummy code for *LogData* , which creates the weird (dummy) object (see last line).

[4] In the multi selection mode the dialog allows to select multiple files

[5] we use the variable name *gamma* frequently for function handles

Pimping the Import Method

We have to pimp the *import* method in order to read our data log files from chapter 1. Before doing that it is time to make a version migration. Copy the whole folder play/v1a to folder play/v1b. This leads us to the following folder structure.

play
play/log
play/v1a
play/v1a/@espresso
play/v1b
play/v1b/@espresso

Remove *play/v1a* from the MATLAB® path and add *play/v1b* to the MATLAB® path. After that I suggest to exit MATLAB® and to re-open MATLAB® again, not forgetting to make sure that the path settings are correct.

Well done, we can start now our actual work and pimp the dummy local driver function *LogData*. What do we have to do? Obviously our previous file selection dialog (*filedialog*) is obsolete, as main function of *import* manages already the file selection and passes each file path to the import driver function *LogData.* Thus we provide code for data read, object creation and log data & parameter setup in the object's properties. Here are the proper code lines.

```
function oo = LogData(o,path)              % Import Driver For .log File
   [x,y,par] = read(path);                 % read log data
   oo = espresso;                          % construct espresso object
   oo.data.x = x; oo.data.y = y;           % set data
   oo.par = par;                           % set parameters
end
```

And that's it! After saving the edited file (*v1b/@espresso/import.m*) we can test our pimped *import* method in order to import our log files. To stay synced: exit the current shell (*File>Exit* - close all figures) and launch a new *Espresso* shell (>> `launch(espresso);`). Then enter:

```
>> list=import(espresso,'.log')    % import log data (DATA1 ... DATA3)
list =
      [1x1 espresso]     [1x1 espresso]     [1x1 espresso]
```

We can enjoy the improved *import* functionality by selecting e.g. the first three log data files *data1.log* … *data3.log* at once. To get the 3 imported objects into the shell we just have to paste the list.

```
   >> paste(carabao,list);    % paste list of objects into current shell
```

A short look into *Select>Objects* makes us sure that the three objects have been successfully imported and pasted into the shell. All right, this was the manual test. But everything should also be ready for the integrated shell functionality. Just clear the objects of the shell (*Edit>Clear Objects*), click on *File>Import>Log Data (.log)* and select (this time) all five log data files *data1.log* … *data5.log* at once. A quick cross check (*Select>Objects*) confirms that the five objects are imported into the shell (fig. 3.5), named *DATA1*, …, *DATA5*.

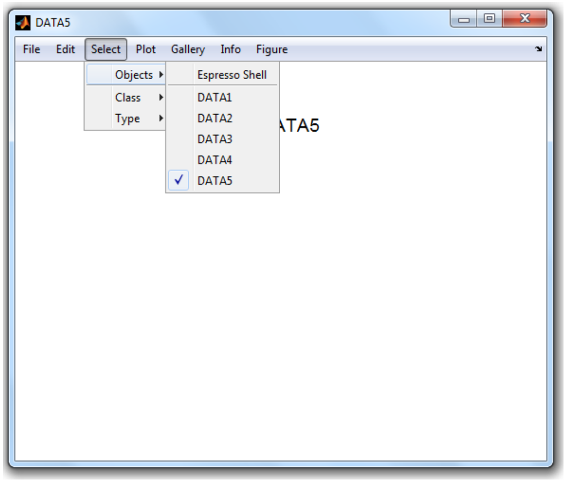


Fig. 3.5 – Espresso shell with 5 imported objects from log data

The Current Object

We have to master now a small challenge. If we are going to plot the data we have to access the current selected object. E.g. after importing our five data log files the object with title 'DATA5' is the current selected one (see fig. 3.5). We learnt how to get the shell object out of the shell and how to access a child object from the shell object, but how can we know that the 5-th child object is the current selected object? Easy – using Carabao method *current*.

```
   >> o=pull(espresso);        % pull shell object from shell
   >> [oo,idx]=current(o);     % get current object
```

```
>> oo.par.title              % display object title
ans =
DATA5
>> idx                       % display current index
idx =
     5
```

Method *current* allows us also to select another object.

```
>> current(o,3);             % set 3rd child of shell object as current
```

Check that the *Select>Objects* menu shows now the 3rd object as the selected one. We even can select the shell object.

```
>> oo = current(o,0);        % select shell object as current (and get)
```

Pimping the Plot Method

We have now all important building blocks for drawing a scatter plot of any imported object. We need to dig out the m-file function *scatterplot* from chapter 1*.* To draw a scatter plot for object *DATA3* we have to invoke:

```
>> o=pull(espresso);                                 % pull shell object
>> oo=current(o,3);                                  % select 3rd object
>> cla;                                              % clear axes
>> scatterplot(oo.data.x,oo.data.y,oo.par);          % plot
```

And to draw a stream plot of x-data and y-data the commands are

```
>> cla;                                              % clear axes
>> streamplot(oo.data.x,'x','r',oo.par);             % plot red x-stream
```

and

```
>> cla;                                              % clear axes
>> streamplot(oo.data.y,'y','b',oo.par);             % plot blue y-stream
```

We need to integrate these building blocks in our shell. The shell's *Plot* menu has three dummy entries *Plot>My Plot #1*, *Plot>My Plot #2* and *Plot>My Plot #3* which can be used for our scatter plot and the two x/y stream plots. Let us first implement the underlaying functionality, in the next section we change the shell outfit (menu item labels). In analogy to the *Espresso import* method we have to pimp the *Espresso plot* method. Let's go and edit the plot method (`>> edit espresso/plot`). Here's what we get.

```
function oo = plot(o,varargin)          % Espresso Plot Method
%
% PLOT   Espresso plot method
%
%            plot(o,'Plot1')            % user defined plot function #1
%            plot(o,'Plot2')            % user defined plot function #2
%            plot(o,'Plot3')            % user defined plot function #3
%            plot(o,'Show')             % show object
%            plot(o,'Animation')        % animation of object
%
   [gamma,oo] = manage(o,varargin,@Error,@Plot1,@Plot2,@Plot3,...
                                 @Show,@Animation);
   oo = gamma(oo);
end

%==========================================================================
% Local Plot Functions
%==========================================================================

function oo = Error(o)                  % report an error
   error('no plot mode provided!');
end

function o = Plot1(o)                   % User Defined Plot Function #1
   message(o,'Modify espresso.plot>Plot1 function!');
end

function o = Plot2(o)                   % User Defined Plot Function #2
   message(o,'Modify espresso.plot>Plot2 function!');
end

function o = Plot3(o)                   % User Defined Plot Function #2
   message(o,'Modify espresso.plot>Plot2 function!');
end

function o = Show(o)                    % Show Object
   oo = cast(o,'carabao');              % cast to a carabao object
   plot(oo,'Show');                     % delegate to plot@carabao
end

function oo = Animation(o)              % Animation Callback
   oo = cast(o,'carabao');              % cast to a carabao object
   plot(oo,'Animation');                % delegate to plot@carabao
end
```

What do we see? First of all most of the methods or local functions[6] carry the shell object *o* as their first input argument, and they return exactly one output argument *oo* ('output object', which in most cases has not much meaning, but sometimes it has).

[6] for names of local functions (functions inside an m-file function or m-file method) we always use a capital letter for the 1st character

The main method *plot* has only two code lines which are obviously used for dispatching.

```
function oo = plot(o,varargin)          % Espresso Plot Method
   [gamma,oo] = manage(o,varargin,@Error,@Plot1,@Plot2,@Plot3,...
                                  @Show,@Animation);
   oo = gamma(oo);
end
```

The work horse of the dispatch mechanism is method *manage* which operates on the variable input arguments list *varargin* and on a couple of function handles for local functions. When we invoke e.g. `>> plot(o,'Plot1')` then method *manage* is capable to return a proper function handle *@Plot1* as 1st output argument *gamma* which by subsequent application to the second output argument *oo* (pimped object) dispatches program control finally to local function *Plot1*.

This kind of dispatching mechanism based on method *manage* is one of the smart core mechanisms of the *Carabao* class. The whole *Carabao* refresh functionality is based on this approach. Definitely there would be a lot more to say about this mechanism, but at this point it would lead us too far away from our mission focus and thus we will stop at this point with this topic and come back later.

Let us continue with the *plot* method. We can use the local function *Plot1* for our scatter plot. Let's start with simple code.

```
function o = Plot1(o)                   % Scatter Plot
   oo = current(o);                     % get current object
   cla;
   scatterplot(oo.data.x,oo.data.y,oo.par);
end
```

Selecting an object (e.g. *Select>Objects>DATA1*) and clicking on *Plot>My Plot #1* shows us the scatter plot for object *DATA1*. What if we select *Select>Objects>DATA2* ? The shell displays the title 'DATA2' in the screen, and with another *Plot>My Plot #1* we get the scatter plot of *DATA2*.

But why doesn't the shell refresh the graphics for *DATA2* immediately? Simple answer: "because we did not tell the shell that local function Plot1 shall be used from now on for refresh. This can be easily changed by updating the refresh callback function.

```
function o = Plot1(o)                        % Scatter Plot
   refresh(o,{'plot','Plot1'});             % update refresh callback
   oo = current(o);                         % get current object
   cla;
   scatterplot(oo.data.x,oo.data.y,oo.par);
end
```

By adding the statement `refresh(o,{'plot','Plot1'});`[7] at the beginning of *Plot1* we tell the shell that from now on for any screen refresh the following command should be invoked.

```
plot(pull(carabao),'Plot1'); % command invoked by the shell for refresh
```

With this modification of local function *Plot1* we can test the shell. With every selection of a different object in the *Select>Objects* menu the scatter plot of the selected object is now immediately updated (refreshed). Looks good!

But the code of Plot1 is still not on professional level. When we select *Select>Objects>Espresso* Shell the shell crashes with an error. Why? Because the statement

```
oo = current(o);                     % get current object
```

returns the shell object which has no data fields *x* and *y*. So we have to prevent calling *scatterplot* for shell objects. Note also that the *cls* method clears the whole figure screen, and is thus a better choice than built-in function *cla,* which would leave empty axes on the screen for the shell object selection.

```
function o = Plot1(o)                        % Scatter Plot
   refresh(o,{'plot','Plot1'});             % update refresh callback
   oo = current(o);                         % get current object
   cls(o);                                  % clear screen
   if ~container(oo)
      scatterplot(oo.data.x,oo.data.y,oo.par);
   end
end
```

Providing also functions for x/y stream plot the pimped code of *Plot2* and *Plot3* could look like:

[7] Note that the 1st list element of the 2nd argument is a character string and not a function handle

```
function o = Plot2(o)                     % Stream Plot For X-Data
   refresh(o,{'plot','Plot2'});           % update refresh callback
   oo = current(o);                       % get current object
   cls(o);                                % clear screen
   if ~container(oo)
      streamplot(oo.data.x,'x','r',oo.par);
   end
end

function o = Plot3(o)                     % Stream Plot For Y-Data
   refresh(o,{'plot','Plot3'});           % update refresh callback
   oo = current(o);                       % get current object
   cls(o);                                % clear screen
   if ~container(oo)
      streamplot(oo.data.y,'y','b',oo.par);
   end
end
```

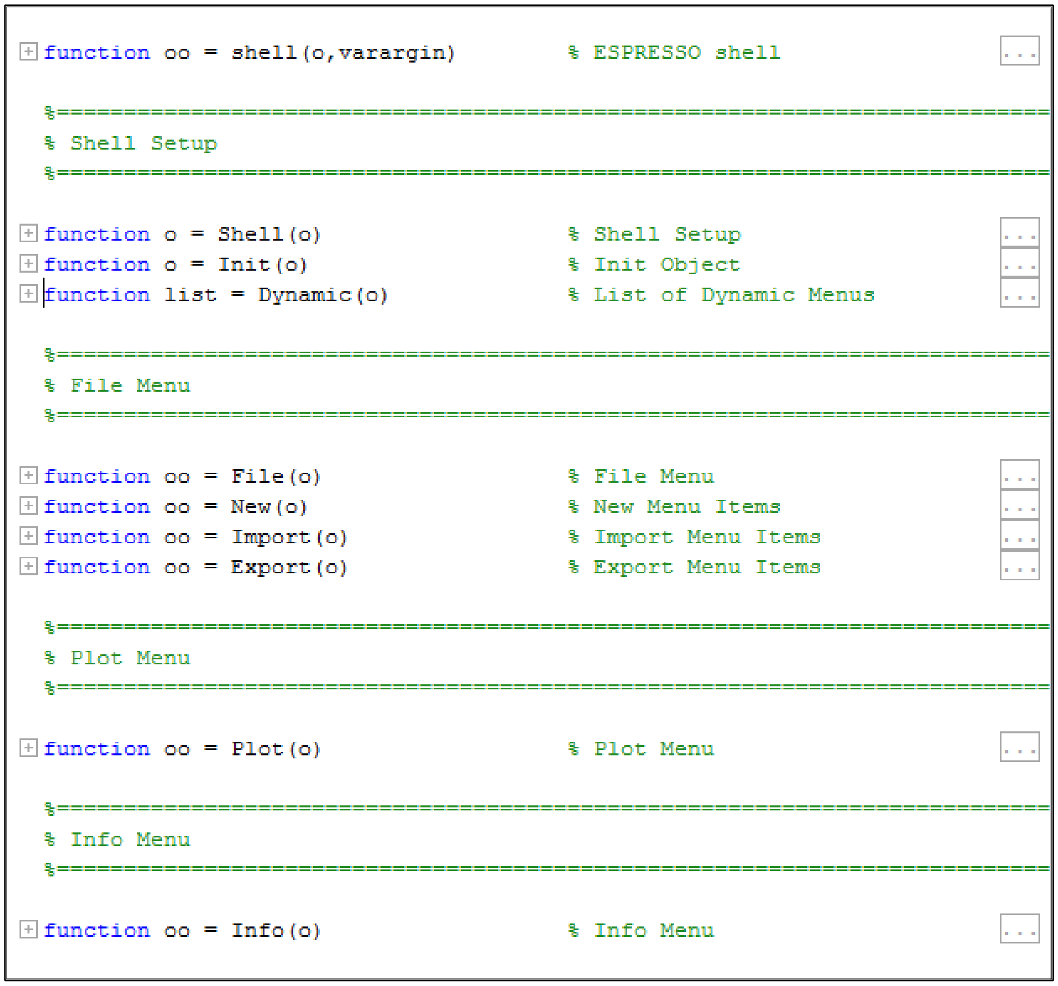
We can test now the whole plot functionality. We can select different log data objects and the plotted graphics changes immediately when a different object is selected. Awesome – we are close to complete our mission.

Changing the Shell Outfit

The only thing that is missing is to change the shell outfit. Remember that we focused on the right functionality while we still were OK with the ugly menu item names *My Plot* #1, ..., *My Plot #3*. This is what we want to fix now, and the location for the necessary changes is somewhere inside the shell method. The shell method is responsible for the menu construction and the callback handling of the menu items, when they are activated. Let us edit and investigate the *shell* method of the Espresso class.

```
>> edit espresso/shell     % edit Espresso shell
```

The according m-file (*v1b/@espresso/shell.m*) will open. To get an overview of this method the reader is advised to collapse all functions. This can be done by selecting tab *VIEW* in the MARLAB® editor and then in section *CODE FOLDING the* button *Collapse All*. The collapsed code of the Espresso *shell* is present as in fig. 3.6. It starts with function *shell*, which is the main function of method *shell*, and like the main function of method plot it is only responsible for dispatching. Furthermore there is a group of three local functions which are responsible for shell setup, and there are two further sections, one for the *File* menu, the other one for the *Info* menu. That's it. Everything else is made of building blocks from Carabao.



```
function oo = shell(o,varargin)        % ESPRESSO shell

   %=====================================================================
   % Shell Setup
   %=====================================================================

function o = Shell(o)                  % Shell Setup
function o = Init(o)                   % Init Object
function list = Dynamic(o)             % List of Dynamic Menus

   %=====================================================================
   % File Menu
   %=====================================================================

function oo = File(o)                  % File Menu
function oo = New(o)                   % New Menu Items
function oo = Import(o)                % Import Menu Items
function oo = Export(o)                % Export Menu Items

   %=====================================================================
   % Plot Menu
   %=====================================================================

function oo = Plot(o)                  % Plot Menu

   %=====================================================================
   % Info Menu
   %=====================================================================

function oo = Info(o)                  % Info Menu
```

Fig. 3.6 – collapsed Espresso shell

We will study the *shell* method in more detail later, at this point we want to fix the menu texts in the plot menu. A good guess for the right code location is to expand local function Plot. Here we go!

```
function oo = Plot(o)                   % Plot Menu
   default(o,'plot.bullets',true);      % provide bullets default
   default(o,'plot.linewidth',3);       % provide linewidth default

   oo = mhead(o,'Plot');                % add roll down header menu item
   dynamic(oo);                         % make this a dynamic menu
   ooo = mitem(oo,'My Plot #1',{@PlotCb,'Plot1'});
   ooo = mitem(oo,'My Plot #2',{@PlotCb,'Plot2'});
   ooo = mitem(oo,'My Plot #3',{@PlotCb,'Plot3'});
   ooo = mitem(oo,'-');                 % separator
```

```
      ooo = mitem(oo,'Show',{@PlotCb,'Show'});
      ooo = mitem(oo,'Animation',{@PlotCb,'Animation'});
      ooo = mitem(oo,'-');
      ooo = mitem(oo,'Bullets','','plot.bullets');
      check(ooo,'');
      ooo = mitem(oo,'Line Width','','plot.linewidth');
      choice(ooo,[1:5],'');
      return

      function oo = PlotCb(o)
         oo = plot(o);            % forward to espresso.plot method
      end
   end
```

Some readers might have expected plenty of calls to built-in function *uimenu*, but those readers will now be disappointed. Instead of built-in function *uimenu* there are lots of calls to 'wrapper methods' *mhead* and *mitem*[8]. Anyway, without explanation of details it is not difficult to see which code lines have to be changed. The three lines

```
   ooo = mitem(oo,'My Plot #1',{@PlotCb,'Plot1'});
   ooo = mitem(oo,'My Plot #2',{@PlotCb,'Plot2'});
   ooo = mitem(oo,'My Plot #3',{@PlotCb,'Plot3'});
```

need to be fixed with proper menu text, and the rest of the code down to the *return* statement can be deleted. Here is how it should look.

```
   function oo = Plot(o)                  % Plot Menu
      oo = mhead(o,'Plot');               % add roll down header menu item
      dynamic(oo);                        % make this a dynamic menu
      ooo = mitem(oo,'Scatter', {@PlotCb,'Plot1'});
      ooo = mitem(oo,'X-Stream',{@PlotCb,'Plot2'});
      ooo = mitem(oo,'Y-Stream',{@PlotCb,'Plot3'});
      return

      function oo = PlotCb(o)
         oo = plot(o);            % forward to espresso.plot method
      end
   end
```

Save file shell.m and rebuild the shell (*File>Rebuild*). With this Espresso shell we got a menu driven log file analysis tool which allows to import log files and analyse them on a menu driven basis. We even can open a new empty Espresso shell (*File>New>Shell*), copy & paste objects, have a full *Gallery* functionality which allows us even to generate a report. Isn't that awesome?

[8] the menu structure could also be built up with uimenu function calls, but the code would be spoiled with a lot of details which have no significance

Review of the Building Blocks

There were a couple of new things we got in touch with, a lot of 'single trees' which made up a 'wood'. Are we able to see the 'wood'? Let's review all important building blocks in terms of another mission.

> **Mission #3**: Use the available building blocks to build an object based log data analysis tool. The tool shall have a graphical menu driven shell which is able to construct random test data objects, export the data to file, import data from file, make a scatter plot and stream plot, has copy&paste functionality, a gallery with report generation and allows to save the whole objects and shell settings into a .mat file and load it from there.

Even if this sounds like a lot of things we are already aware that we can solve this mission with only a couple of steps.

Step 1: Version Migration

First of all make a version migration, making a new folder *play/v1c*, restart MATLAB® and change the MATLAB® path to include this folder.

Step 2: Rapid Protyping of Espresso Class

Launch the Carabao rapid prototyping shell in order to generate an *Espresso* class version v1c [9].

```
>> rapid(carabao,'espresso');
```

Start *Espresso* class creation with *Class>Create New Class* and choosing *play/v1c* as the parent folder for the class directory to create (*play/v1c/@espresso*) .

Step 3: Private Functions

Make a private folder *play/v1c/@espresso/private* and copy the m-file functions *read.m, scatterplot.m* and *streamplot.m* from folder *play/v1b* into the private folder. These four functions have only internal character for *Espresso* methods and putting them into the private folder of the *Espresso* class avoids that they are visible on global level.

[9] this time you can try to do it in 30 seconds

Step 4: Pimping the Import Method

Pimp the *import* method. Edit *espresso/import.m*

```
>> edit espresso/import;   % edit v1c/@espresso/import.m
```

and pimp the local driver function for .log extension.

```
function oo = LogData(o,path)           % Import From .log File
   [x,y,par] = read(path);              % read log data
   oo = espresso;                       % construct espresso object
   oo.data.x = x; oo.data.y = y;        % set data
   oo.par = par;                        % set parameters
end
```

After saving the m-file an *Espresso* shell can be launched (`>> launch(espresso);`) and the log data import can be tested with *File>Import>Log Data (.log)*.

Step 5: Pimping the Plot Method

Pimp the *plot* method. Edit *espresso/plot.m*

```
>> edit espresso/plot   % edit v1c/@espresso/plot.m
```

and adopt the following three local functions.

```
function o = Plot1(o)                   % Scatter Plot
   refresh(o,{'plot','Plot1'});         % update refresh callback
   oo = current(o);                     % get current object
   cls(o);                              % clear screen
   if ~container(oo)
      scatterplot(oo.data.x,oo.data.y,oo.par);
   end
end

function o = Plot2(o)                   % Stream Plot For X-Data
   refresh(o,{'plot','Plot2'});         % update refresh callback
   oo = current(o);                     % get current object
   cls(o);                              % clear screen
   if ~container(oo)
      streamplot(oo.data.x,'x','r',oo.par);
   end
end

function o = Plot3(o)                   % Stream Plot For Y-Data
   refresh(o,{'plot','Plot3'});         % update refresh callback
   oo = current(o);                     % get current object
   cls(o);                              % clear screen
   if ~container(oo)
      streamplot(oo.data.y,'y','b',oo.par);
   end
end
```

After saving the m-file we can immediately test the plot functionality by selecting a log data object (*Select>Objects*) and testing scatter plot (*Plot>My Plot #1*), x-stream (*Plot>My Plot #2)* and y-stream plotting (*Plot>My Plot #3*).

Step 6: Changing the Shell Outfit

Edit *espresso/shell.m*

```
>> edit espresso/shell   % edit v1c/@espresso/shell.m
```

and adopt the following local function.

```
function oo = Plot(o)                     % Plot Menu
   oo = mhead(o,'Plot');                  % add roll down header menu item
   dynamic(oo);                           % make this a dynamic menu
   ooo = mitem(oo,'Scatter', {@PlotCb,'Plot1'});
   ooo = mitem(oo,'X-Stream',{@PlotCb,'Plot2'});
   ooo = mitem(oo,'Y-Stream',{@PlotCb,'Plot3'});
   return

   function oo = PlotCb(o)
      oo = plot(o);                 % forward to espresso.plot method
   end
end
```

Clicking on *File>Rebuild* rebuilds the shell and gives us the menu labeling for our Espresso shell in the desired outlook.

We are at the same level with our version v1c Espresso shell, as we have been with version v1b. With version v1b we did it 'tree' by 'tree'. Now with version v1c the reader should be able to see the 'tree'. But we still are not done with our mission. There are two tasks left.

First we should be able to export data into a .log file. As we are aware that our Espresso class has an export method we can expect that this task might be similar to pimping the import method.

Step 7: Pimping the Export Method

Pimp the *export* method. Edit *espresso/export.m*

```
>> edit espresso/export;   % edit v1c/@espresso/export.m
```

In analogy to the import driver we find sample code for an export driver function.

```
function ExportLog(o,path)                 % Export Driver For .log File
   fid = fopen(path,'w');                  % open log file for write
   if (fid < 0)
      error('cannot open export file');
   end

   fprintf(fid,'$title=%s\n',get(o,{'title',''}));
   % add more code for export              % put your own code here
   fclose(fid);                            % close log file
end
```

In the line with comment `'% put your own code here'` we have to add some code similar to what we used in our *create* m-file function from chapter 1. Change as follows:

```
function ExportLog(o,path)                 % Export Driver For .log File
   fid = fopen(path,'w');                  % open log file for write
   if (fid < 0)
      error('cannot open export file');
   end

   fprintf(fid,'$title=%s\n',get(o,{'title',''}));
   log = [o.data.x,o.data.y];              % compose data matrix
   fprintf(fid,'%10f %10f\n',log');        % write x/y data
   fclose(fid);                            % close export file
end
```

After saving the m-file an the log data export can be tested immediately with *File>Export>Log Data (.log)* and verified with *File>Import>Log.Data (.log)* by importing it again.

Also this step was not difficult. Finally we should provide a menu item that allows us to create a new object with test data, similar like we did in chapter 1 when we had to create some sample log files. Having an easy method to create test objects is always of benefit during a rapid prototyping process. There is a menu item File>New>Weird which we can pimp. Studying the compressed Espresso shell of fig. 3.6 we

Step 8 – Fine Tuning of the Shell Outfit

There is a *File>Import>Text File (.txt)* menu entry and another one *File>Export>Text File (.txt)*, which we do not want to see in our menu.

```
>> edit espresso/shell;    % edit v1c/@espresso/shell.m
```

and visit local functions *Import* and *Export* which are responsible for the menu construction and callback handling. Comment the lines which create the menu items for *Text File (.txt)* import and export.

```
function oo = Import(o)                % Import Menu Items
   oo = mhead(o,'Import');             % locate Import menu header
   %ooo = mitem(oo,'Text File (.txt)',{@ImportCb,'.txt',@carabao});
   ooo = mitem(oo,'Log Data (.log)',{@ImportCb,'.log',@espresso});
   return
      :             :              :
end

function oo = Export(o)                % Export Menu Items
   oo = mhead(o,'Export');             % locate Export menu header
   %ooo = mitem(oo,'Text File (.txt)',{@ExportCb,'.txt','carabao'});
   ooo = mitem(oo,'Log Data (.log)',{@ExportCb,'.log','espresso'});
   return
      :             :              :
end
```

Step 9: Pimping the New Menu

Pimp the local function *shell/New*. In the edited *espresso/shell.m* visit local function *New* which is responsible for the menu construction and callback handling.

```
function oo = New(o)                   % New Menu Items
   types = {'weird'};                  % supported object types
   oo = shell(carabao(o),'New',types); % add New menu items
end
```

Obviously the menu build-up is delegated to the Carabao *shell* method which provides a menu entry *File>New>Shell* and another entry *File>New>Weird* , defined by the line `types = {'weird'};` Modify the code as follows:

```
function oo = New(o)                   % New Menu Items
   types = {};                         % no supported object types
   oo = shell(carabao(o),'New',types); % add New menu items
   ooo = mitem(oo,'Espresso Object',{@NewCb});
   return

   function oo = NewCb(o)
      log = ones(1000,1)*randn(1,2) + randn(1000,2)*randn(2,2);
      oo = espresso;
      oo.data.x = log(:,1);  oo.data.y = log(:,2);
      oo.par.title = ['Espresso (',datestr(now),')'];
      paste(o,{oo});
   end
end
```

The first line sets the *types* list empty which means that we do not need additional Carabao *File>New* menu items except for the *File>New>Shell* creation. After consulting the *Carabao shell* building block `oo = shell(carabao(o),'New',types);` we create our own menu sub-item *File>New>Espresso Object* which we setup with the local callback function handle *@NewCb*.

The actual job of sample object creation is done in *NewCb*. We create a log data table *log* like we did in chapter 1, construct an *Espresso* object which is assigned to variable *oo*, set the data and parameter properties and finally *paste* the object into the shell. The whole code can be tested after saving by rebuilding the menu (*File>Rebuild*) and clicking one or more times on *File>New>Espresso Object*.

The rest of the mission tasks like copy & paste functionality, gallery and report generation, saving/loading the shell to a .mat file is automatically supported by the *Carabao* mechanisms of the Espresso shell and needs no further customization. Our mission #3 is complete.

Fig. 3.7: Espresso class leverages the powerful Carabao mechanisms

Ensemble Analysis

So far we got a nice interactive log data analysis tool, and we are pleased now that our *Espresso* shell supports most of the powerful features of the Carabao shell. But anyway, our analysis has still one limitation: The analysis can only be done focusing on one by one object. There is no possibility to see an overlay over all objects, or over a subset. The *Carabao* class, however, is designed to solve especially this kind of task. So let's get it done!

> Mission #4: augment the Espresso shell to allow overlayed scatter plot and stream plots. Also add a further plot mode where the standard deviations of the x/y stream and the correlation coefficients are plotted over the object index.

Remember the *basket* functionality of the Carabao shell. We will demonstrate how the Carabao *basket* can be utilized to master our mission.

The Basket

The *Carabao basket* is an abstaction of a configurable ensemble of objects which are on base of certain 'collection' criteria which can be chosen via the menu. The basic Carabao collection procedure is as follows:

- **Collection**: collect either from all objects of the shell or the selected one. (this can be the shell object or a shell's child object). If the shell object is selected then all child objects of the shell are collected into the basket.
- **Class Filtering**: the class of the selected object (no matter whether the shell object or a child object is selected) determines the class filter. The basket can only contain objects with a class matching the selected object.
- **Type Filtering**: If all types are selected then the type filter does not additionally reduce the current collected basket objects, otherwise only those objects are allowed to stay in the basket which are matching the selected type.

The collection process is implemented by the *Carabao* method *basket*. There is a possibility to extend this functionality for derived *Carabao* classes. E.g. in such an extension there could be a marking mechanism for objects, and the *basket* method could be extended to switch between four choices:

- all objects
- the selected object
- all marked objects
- all unmarked objects

It can also be imagined to introduce groups of objects which can be enabled/disabled interactively, and the collection mechanism for the basket could be extended in a way that it filters those groups which enabled, excluding those objects from the basket which belong to a disabled group. Such kind of mechanism makes a lot of sense in data analysis.

For us it does not matter at this point how the collection procedure is specifically implemented when we focus on the implementation of an analysis function. The only thing we have to imagine is that there is some abstract basket which holds a list of objects, and that we have a *basket* method which returns this list of objects to us.

Rewriting the Plot Method

Let us make a further version migration and turn to version *play/v1c*. Duplicate folder *play/v1b* with contents to *play/v1c*. Next let us realize that in version play/v1b the functions *read*, *scatterplot* and *streamplot* were at global scope level. Since they are only auxillary functions for *Espresso* methods *import* and *plot* we can convert them to private functions by creating a folder *play/v1c/@espresso/private* and moving them to this location. The other functions *analysis*, *create* and *filedialog* can be deleted, as we have no more use for them. Finally exit and restart MATLAB® and make sure that instead of folder *play/v1b* the new folder *play/v1c* is on the MATLAB® path.

Now we can start improving our code. Edit the *Espresso* plot method (`>> edit espresso/plot`). There are three driver plot functions *Plot1*, *Plot2* and *Plot3*. These names are good for quick & dirty code, but we can make the code looking nicer by giving them the new names *Scatter, Xstream and Ystream.* To keep the dispatch mechanism of the *plot* method working we need to change its main function. Since we will add later an additional local plot function *Ensemble* we pass also function handle *@Ensemble* as an input argument to the dispatcher method *manage*.

```
function oo = plot(o,varargin) % Espresso Plot (v1c/@espresso/plot.m)
   [gamma,oo] = manage(o,varargin,@Scatter,@Xstream,@Ystream,@Ensemble);
   oo = gamma(oo);
end
```

Now replace the definition of *Plot1* by the code for *Scatter*.

```
function o = Scatter(o)                    % Scatter Plot
   refresh(o,{'plot','Scatter'});          % update refresh callback
   cls(o);                                 % clear screen
   list = basket(o);                       % get list of objects in basket
   for (i=1:length(list))
      oo = list{i};
      scatterplot(oo.data.x,oo.data.y,oo.par);
      hold on;
   end
   if length(list) > 1
      title(get(o,{'title','Espresso Shell'}));
   end
   what(o,'Scatter Plot','scatter plot y(x) of log data');
end
```

As before the code starts with an update of the refresh function. Note the modified callback list in `refresh(o,{'plot','Scatter'});` which means that on refresh request a call `plot(o,'Scatter')` will be invoked. After clearing the screen the list of basket objects is being retrieved from the basket,

```
list = basket(o);    % get list of objects in basket
```

and for each object of the basket list the private function scatterplot is invoked. Don't forget to hold the current plot such that subsequent *plot* calls do not replace that plot. After the loop has been processed we make a final conclusion if the screen shows only the scatter plot of one single object (then the graphics title is appropriate) or we see an overlay of more scatter plots. In the second case we overwrite the graphics title with the shell object's title. Notice the special syntax of *get* to fetch the *title* from the object

```
get(o,{'title','Espresso Shell'})
```

If o.par.title is non-empty then this value will be returned, otherwise the default value `'Espresso Shell'` will be provided.

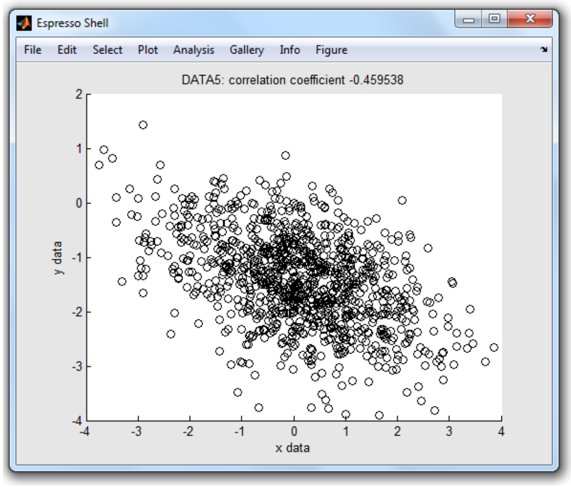
The last line finally stores some plot information which can be shown with menu item *Figure>What We See>Display*. To make the code running we need also to change the menu item callback. Edit the shell method (`>> edit espresso/shell`) and change the local Plot function as follows.

```
function oo = Plot(o)                      % Plot Menu
   setting(o,{'plot.bullets'},true);       % provide bullets default
   setting(o,{'plot.linewidth'},3);        % provide linewidth default

   oo = mhead(o,'Plot');                   % add roll down header menu item
   dynamic(oo);                            % make this a dynamic menu
   ooo = mitem(oo,'Scatter',{@PlotCb,'Scatter'});
   ooo = mitem(oo,'X-Stream',{@PlotCb,'Xstream'});
   ooo = mitem(oo,'Y-Stream',{@PlotCb,'Ystream'});
   return

   function oo = PlotCb(o)
      oo = plot(o);                        % forward to espresso.plot
   end
end
```

The changes are regarding the lines which create the menu items and setup the callback functions with proper arguments.

```
ooo = mitem(oo,'Scatter',{@PlotCb,'Scatter'});
ooo = mitem(oo,'X-Stream',{@PlotCb,'Xstream'});
ooo = mitem(oo,'Y-Stream',{@PlotCb,'Ystream'});
```

Now we can test the code: launch an Espresso shell and import our 5 log data files which leaves the last imported object *DATA5*.


Espresso Shell
File Edit Select Plot Analysis Gallery Info Figure
DATA5: correlation coefficient -0.459538
y data
x data

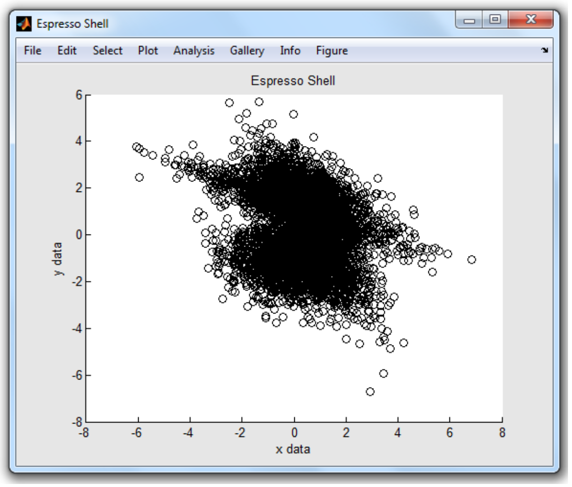


Fig. 3.8 – Scatter plot of DATA5 (left) and all objects (right)

By selecting *Plot/Show* we see the scatter plot of *DATA5* (fig. 3.8 left). Selecting the shell object (*Select>Object>Espresso Shell*) gives us an overlay of five scatter plots (fig. 3.8 right). An alternative way to select all objects is with the basket option *Select>Basket>Collect>All Objects* which overwrites any selection of a single object.

In the same way we replace *Plot2* with *Xstream*, and *Plot3* with *Ystream*.

```
function o = Xstream(o)                    % Stream Plot For X-Data
   refresh(o,{'plot','Xstream'});          % update refresh callback
   cls(o);                                 % clear screen
   list = basket(o);                       % get list of objects in basket
   for (i=1:length(list))
      oo = list{i};
      streamplot(oo.data.x,'x','r',oo.par);
      hold on
   end
   if length(list) > 1
      title(get(o,{'title','Espresso Shell'}));
   end
   what(o,'X-Stream Plot','x-stream of log data over index');
end

function o = Ystream(o)                    % Stream Plot For Y-Data
   refresh(o,{'plot','Ystream'});          % update refresh callback
   cls(o);                                 % clear screen
   list = basket(o);                       % get list of objects in basket
   for (i=1:length(list))
      oo = list{i};
      streamplot(oo.data.y,'y','b',oo.par);
      hold on
   end
   if container(current(o))
      title(get(o,{'title','Espresso Shell'}));
   end
   what(o,'Y-Stream Plot','x-stream of log data over index');
end
```

Basket Settings and Basket Options

The Scatter and X/Y-Stream functions operate on the basket which is controlled by menu item selections. Depending on the specific selections we see either a plot which is regarding a single object or an overlay of plots regarding the entire collection of objects. This is not what is always wanted, e.g. if we want to show an overview of statistic data of the whole ensemble, independent of current menu selections.

Let us investigate the basket method in more details. Setup the following menu setting: *Select>Objects>DATA5* and *Select>Basket>Collect>Selected Object.* This will bring up the graphics of Fig. 3.8 left. With this settings let us investigate the *basket*.

```
>> basket(o)
  Current Selection
    object #5: DATA5
  Basket class
    class:  espresso
  Basket options
    collect: selected
    type:    *
  Objects in basket
    object #5: DATA5
```

We see that object #5 (*DATA5*) is the current selection which determines the basket class `'espresso'`, i.e. all objects in the *basket* must have class `'espresso'`. The next interesting information is regarding the *basket* options. There are two basket options *collect* and *type* and there is another possibility to see these option settings:

```
>> opt(o,'basket')
ans =
    collect: 'selected'
       type: '*'
```

We did not introduce the concept of shell settings and object options[10] but at this point it is sufficient to assume that there are some basket shell settings which can be changed from the menu, and when an object is pulled from the shell those settings are passed to the object in terms of options. Method opt does not only allow to retrieve the options, as shown above, we can also change the value of an option.

```
>> o=opt(o,'basket.collect','*');  % change option to '*' (all objects)
```

Now investigate again the basket.

```
>> basket(o)
  Current Selection
    object #5: DATA5
  Basket class
    class:  espresso
  Basket options
    collect: *
    type:    *
  Objects in basket
    object #1: DATA1
```

[10] those will be covered in more detail in the next chapter

```
      object #2: DATA2
      object #3: DATA3
      object #4: DATA4
      object #5: DATA5
```

All five objects are now included in the basket. This is the way how we can change the basket collection criteria programmatically.

Ensemble Analysis

Let us now present the code for a simple ensemble analysis for Espresso log data objects. Extend the Espresso plot method with a further local function *Ensemble*.

```
function o = Ensemble(o)                % Ensemble Analysis
   refresh(o,{'plot','Ensemble'});      % update refresh callback
   o = opt(o,'basket.type','shell');
   o = opt(o,'basket.collect','*');     % all traces in basket

   list = basket(o);
   n = length(list);

   if (isempty(list))
      message(o,'No objects in basket!');
      return
   end

   hax = cls(o);
   for (i=1:n)
      oo = list{i};
      sx(i) = std(oo.data.x);
      sy(i) = std(oo.data.y);
      c = corrcoef(oo.data.x,oo.data.y);  cc(i) = c(1,2);
   end

   hax = subplot(211);
   plot(hax,1:n,sx,'r', 1:n,sy,'b', 1:n,sx,'r.', 1:n,sy,'b.');
   set(hax,'xtick',1:n);
   title('Standard Deviation x(red), y(blue)');

   hax = subplot(212);
   plot(hax,1:n,cc,'k',1:n,cc,'k.');
   set(hax,'xtick',1:n);
   title('Correlation Coefficient');

   what(o,'Analysis of Ensemble',...
          'standard deviations x/y','correlation coefficient');
end
```

The crucial part of the code are the lines where the object options are overwritten (`o = opt(o,'basket.type','shell');` and `o = opt(o,'basket.collect','*');`). The rest of the code is nothard to understand. The code can be tested by pulling an object from the shell and invoking the Espresso *plot* method with mode `'Ensemble'`.

```
>> o=pull(espresso);     % pull object from shell
>> plot(o,'Ensemble');   % invoke espresso/plot>Ensemble
```

As a result we get the graphics of fig. 3.9 which displays the standard deviation and the correlation coefficient over the five log data sets.

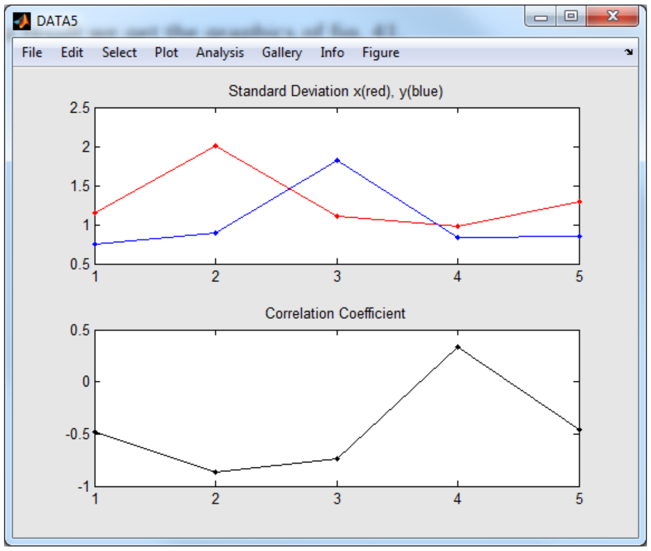


Fig. 3.9 – Ensemble Analysis

For a full integration of this function we need to modify the menu in the *Espresso* shell method. The rapid prototyping tool has already provided a dummy local function *Analysis* which needs to be modified as follows:

```
function oo = Analysis(o)            % Analysis Menu
   oo = mhead(o,'Analysis');         % add roll down header menu item
   dynamic(oo);                      % make this a dynamic menu
   ooo = mitem(oo,'Ensemble',{@PlotCb,'Ensemble'});
   return

   function oo = PlotCb(o)
      oo = plot(o);                  % forward to espresso.plot method
   end
end
```

After saving the m-file and rebuilding the shell (*File>Rebuild*) we can select the ensemble analysis from menu (*Analysis>Ensemble*).

Lessons Learnt

Let us summarize our further learnings about Carabao methods?

- There is a rapid prototyping shell which can be used for class derivation from an existing class based on *Carabao*. We can easily derive an *Espresso* class from *Carabao*, and in the same way e.g. derive a *Capuchino* class from *Espresso*.

```
>> rapid(carabao,'espresso');     % derive espresso from carabao
>> rapid(espresso,'capuchino');   % derive capuchino from espresso
```

- With the default settings the rapid prototyping shell creates the following methods.

```
espresso:  constructor
shell:     default shell method
import:    general import method
export:    general export method
plot:      general plot method
version:   class version/releasenotes
```

  The created methods (except the constructor) can be considered as template files which usually have to be customized for particular needs. Any additional method can be easily added by creating an m-file in the class directory.

- Method *current* can be used to get the current object and the index of the current object.

```
>> [oo,idx] = current(o);      % get current object & index
```

- There is a *paste* method to paste a list of objects into the current shell.

```
>> paste(carabao,{oo});        % paste object oo into shell
>> paste(carabao,{o1,o2,o3}); % paste object o1,o2,o3 into shell
```

- Method *refresh* is used to setup a refresh callback. For a callback to execute as

```
foo(pull(carabao),arg1,arg2,…)
```

  there is an according setup and invoking syntax for *refresh*.

```
>> refresh(o,{'foo',arg1,arg2,...});   % setup refresh callback
>> refresh(o);                         % invoke refresh callback
```

- Local functions *Loc1*, *Loc2*, … can be dispatched by the *manage* method which returns a function handle *gamma* and a pimped object *oo* such that a subsequent `oo = gamma(oo)` call invokes the local function. The arguments of the *varargin* list are passed to the objects arguments `arg(o,0)`.

```
function oo = foo(o,varargin)
   [gamma,oo] = manage(o,varargin,@Loc1,@Loc2,...);  % dispatch
   oo = gamma(oo);                              % invoke local function
end
```

- The overloaded *shell* method uses methods mhead and mitem for menu construction.

  ```
  oo = mhead(o,label);                          % create menu headeroo
  ooo = mitem(oo,label,callback,userdata);      % create menu item
  ooo = mitem(oo,'-');                          % create menu separator
  ```

  These methods are not described in detail at this point, but will get more attention in the next chapter.

- The overloaded *import* method is called from a shell's *File>Import>...* callback with two input arguments.

  ```
  list = import(o,'.log');     % calling import method
  ```

  After opening a multiple file selection dialog it dispatches control for each selected file to a local import driver function *import/ImportLog* which has to create and return an object *oo* from the selected file's data. The imported objects *oo* are collected in the returned output argument *list*.

  ```
  oo = LogData(o,path);       % local import driver function call
  ```

- The overloaded *export* method is called from a shell's *File>Export>...* callback with two input arguments.

  ```
  list = import(o,'.log');     % calling import method
  ```

  After opening a file selection dialog it dispatches control to a local export driver function *export/ExportLog* which has to write the object's parameters and data to file.

  ```
  oo = LogData(o,path);       % local export driver function call
  ```

- The overloaded *plot* method is called from a shell's *File>Plot>...* callback with two or more input arguments.

  ```
  oo = plot(o,'Plot1',arg1,arg2,…);  % calling plot/Plot1 function
  oo = plot(o,'Plot2',arg1,arg2,…);  % calling plot/Plot2 function
  ```

The plot method usually dispatches control to a local function defined by the second input argument, e.g. to Plot1, Plot2, etc.

```
function oo = plot(o,varargin)
   [gamma,oo] = manage(o,varargin,@Plot1,@Plot2,...);
   oo = gamma(oo);
end
```

- The overloaded *version* method returns a character string with the actual version of the class definition. The m-file is usually also used to contain class or toolbox release notes

```
>> v=version(espresso)    % get class version
v =
V1C
>> edit espresso/version % see release notes
```

- There is a *basket* method that retrieves the list of objects in the basket. The current basket collection criteria can be overwritten in terms of option settings.

```
>> o = pull(o);                            % pull object from shell
>> o = opt(o,'basket.type','shell'); % type filter for basket
>> o = opt(o,'basket.collect','*');  % all traces in basket
>> list = basket(o);                       % get list of basket objects
```

The calling syntax `>> basket(o);` displays the basket collection criteria.

References

[1] *Stormy Attaway*: "MATLAB® (2013) – 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] *Morris, Jonathan (2007):* "The Cappuccino Conquests. The Transnational History of Italian Coffee", Academia.org (*University of Hertfordshire*)

[4] *Wikipedia* "Espresso"