Riding the Carabao

If you ever have the pleasure to visit the Philipines you can take a guided tour on a carabao. There is even the option to book a "drunken carabao tour". The tour costs 1000PHP per person, most tours last at least 2-3 hours. It is completely safe besides the fact that you may fall off but you’ll be so drunk that is doesn’t matter. You’ll end up with grass stains or dirt on your leg.

Now that we have a good understanding about the value of a Carabao we will go to the next level – "Riding the Carabao" , which means fully leveraging the draft conceptual power of the Carabao class design for data analysis requirements. If you fall off, it does not matter! You try it again until you feel safe at the next level.

[](https://www.google.at/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0CAcQjRxqFQoTCOqD0fqz7cgCFUHHGgodkXYB0Q&url=https://sacadalang.wordpress.com/tag/family/&psig=AFQjCNEVSJd5MmImjrJUUN2dT8AL6d2ShA&ust=1446404803993090)If you don't live in Malaysia, on the Philippines or other related countries in Asia, you probably do not really have a clue what a carabao is, but you can associate much more with the word 'carabao' if I tell you that a carabao is a swamp-type domestic water buffalo. Philippine farmers leverage carabaos as a source of draft animal power and live from their meat and milk. They use carabaos as strong and robust 'working engines' for their basic farming tasks, like drawing sledges, carriages or a plugs. The farmers guide a group of carabaos in trampling the planting area until it is soggy enough to receive the rice seedlings. Last but not least you can ride a carabao, and you should be noted that every Philipine farmer kid is skilled to ride a carabao.

In this chapter we will develop the *Carabao object class* which is equipped with draft conceptual power for repeated basic tasks to be utilized in prototyping applications:

* simple but powerful data management
* shell association
* mass data storage
* modular building block environment

When we talk about 'riding the *Carabao object* ' we mean that we want to achieve advanced skill level to leverage the power of *Carabao objects*.

# Carabao Capabilities

Let us develop a Carabao class that provides us an object that will master the following challenges

* the object class allows us to manage any data structures, and those data structures can be extended dynamically in a very simple way
* the object instance can be associated with a graphical shell (e.g. a figure equiped with a roll down menu, or a GUI shell generated by MATLAB GUIDE), and the shell callback functions can easily access the object data members and methods
* any object instance associated with a shell is uniquely able to retrieve the graphical handles associated with the shell
* the shell is always associated with a current refresh function which is responsible for the current visual outlook of the shell
* the object class provides a mechanism for managing shell settings which can be manipulated easily by the GUI elements of the shell (menus, buttons, …)
* the object class provides basic mass storage management, which means it has save and load methods to store and retrieve the object to/from a mass storage data base
* the save/load methods are also capable to manage the storage of derived object classes with higher complexity
* during object save also the current refresh function will be saved, and after object load the shell of an object can be re-launched which includes invoking the saved refresh function
* the storage form should allow to save an object class version n, and to load the object into a class version n+m, to allow future modifications of a class definition
* the shell of an object can be easily cloned, which means both the object and the graphical environment will be cloned
* the object shell can be launched by an external application, e.g. a database browser application, or a managing tool that manipulates objects

The reader should not bother if the meaning of the listed capabilities is not well understood in detail by the time. We will move through the details later by studying several examples.

# The Carabao Constructor

Let us design a constructor for the Carabao object class. We will provide exactly five class properties which are

* *tag*: object's tag (to be used to reconstruct an object from a 'bag')
* *type*: object's type (determines associated shell type and data interpretation)
* data: object's data (usually structured, provided with integrity rules)
* par: object's parameters (usually non or weakly structured)
* work: object's work properties (work variables, object arguments, options)

The reader has still to stress his patience about the detailed meaning of the described properties. By the time the reader should just recognize that the *Carabao* class provides exactly five properties (*tag*, *type*, *par*, *data* and *work*) which will be used for different tasks. But the reader should also recognize that every derived class from the *Carabao* base class should only deal with these five basic properties. We will see that these five properties are powerful enough to provide the most complex data structures for much more sophisticated object classes, which can all be derived from the *Carabao* basis class.

MATLAB provides two possibilities of object class definitions: the old one, which for the *Carabao* class would require a class directory *@carabao* containg the class methods written as 'normal' m-file functions, and the additional requirement of an m-file function *carabao.m* which will be identified as the *Carabao* constructor method.

Alternatively there is the new way of MATLAB class definition which requires also a *@carabao* class directory, but demands the *carabao.m* file to contain a *classdef* clause which defines the class interface and the class structure.

For the *Carabao* class definition we will still go with the old MATLAB method, since MATLAB is very picky if the class definition will be modified. MATLAB would then require the user to do some cleanup activities, which we do not want to involve here.

So let's get started by creating a *v0a@carabao* directory, making sure that the MATLAB path 'sees' that directory, and create a file *carabao.m* inside with the following contents.

function o = carabao % v0a/@carabao/carabao.m

o.tag = 'carabao'; % tag holds class name

o.type = 'shell'; % object type

o.par = []; % parameter property

o.data = []; % data property

o.work = []; % work property

o = class(o,'carabao'); % create class object

end

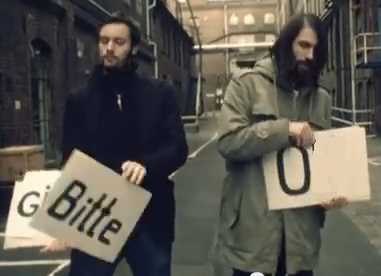
The constructor initializes a structure *o* that will be converted to a class object by the final o = class(o,'carabao')statement. Note that the *tag* property needs to carry the class name, and the *type* property is initialized with the value 'shell' (the meaning of this convention will become more clear at a later time when we introduce the concept of *'object launching'*). In our simple case the created object has empty *par,* *data* and *work* properties, and we call such an object a *generic* *Carabao* object. Let's actually construct a *generic Carabao* object.

>> o=carabao

o =

carabao object: 1-by-1

# Please Give Me Only an 'o'

The German pop & rock band 'Wir sind Helden' could land a song hit in 2005 with the refrain "Bitte gib mir nur ein oh" which means "please give me only an 'o' ". When we need a variable name for an object we will neither write object nor will use obj, we will use the simplest variable name o. Several text books intro­ducing into programming methods are recommending not to use the letter o in identifiers as this might give confusion with the zero character 0. There will be no confusion, at least with the lower case o, believe me, and we will use the variable name o from now on for an object variable so frequently, that the short name length will find its justification in the high frequency of usage.

Within a function we might have need for different object names, and we will most likely introduce names like oo, ooo, oooo, etc. Love it or not, but oo might be the acronyme for 'output object', which we frequently have to deal with, and we will write code for menu tree definitions where this notation gives a very nice impression of the sub menu nesting levels. These should have been enough comments regarding object variable name notations.

# Getting Object Properties

Let us introduce a new method *property* which can be used to get or set object properties. To do so we will create a file *@caramba/property.m* with the following contents:

function o = property(o,tag,value) % v0a/@carabao/property.m

if (nargin == 1)

o = struct('tag',o.tag,'type',o.type,'par',o.par,...

'data',o.data,'work',o.work);

elseif (nargin == 2)

o = eval(['o.',tag],'[]');

elseif (nargin == 3)

eval(['o.',tag,' = value;']);

else

error('1, 2 or 3 input args expected!')

end

end

The repeated if-elseif-else clauses let us easily conclude that the method can be called with 1, 2 or 3 input arguments. Let us study a call with one input argument.

>> bag = property(o)

bag =

tag: 'carabao'

type: 'shell'

par: []

data: []

work: []

The statement >> bag = property(o) packs the 5 object properties into a structure o (this time o does not denote an object variable) which will be returned by method property as an output argument, where it afterwards is assigned to the variable bag, according to our command line input. We will learn more about 'bag packing' later. Actually bag packing is one of the fundamental data management philosophies of the Carabao class concept and provides an efficient solution for a challenge which I call the *Mass Storage Migration Challenge* (we will hear more about that in detail later).

We can use a property call with two input arguments to get the property values of the *tag* and *type* property.

>> tag = property(o,'tag')

tag =

carabao

>> typ = property(o,'type')

typ =

shell

There should not be any surprise for the reader when studying the results. Nevertheless we need to pay additional attention to the powerful functionality of this tiny code segment. What will happen, if we request the property value of a non existing property, like junk ?

>> property(o,'junk')

ans =

[]

As the reader can recognize the property method returns just an empty value []. This is by intention, since the whole Carabao philosophy is based on the fact that a request of an unexisting property value will not lead to an error but return of an empty value. Frankly speaking there is no benefit if the request of a non existing property like junk leads to an empty value, but usually shells are written to operate on objects which can have very different parameter and data initializations. Thus a shell method should tolerantly deal with empty parameter and data properties like

>> property(o,'par.date')

ans =

[]

>> property(o,'data.x')

ans =

[]

in order to respond with tolerant recovery actions. The way how the property method manages the property *get* request is not difficult to understand. The call property(o,'data.x') would be finally processed by the eval statement

o = eval(['o.',tag],'[]');

which after calculation of the actual arguments would be in equivalence with

o = eval('o.data.x','[]');

what would cause *eval* to assign either the value o.data.x if the structure member o.data.x exists, or otherwise would cause *eval*  to return the result of the alternative expression '[]', which yields in the empty return value [].

# Setting Object Properties

We did not utilize a call to the property method with 3 input arguments, so let's give it a trial.

>> property(o,'type','trial')

ans =

carabao object: 1-by-1

>> typ = property(o,'type')

typ =

shell

We had some hope that a call of *property* with three input arguments could be used for setting (changing) an object property, but we see now that the type property has been left unchanged, still carrying the value 'generic' and not 'trial' what we hoped to see. What's going wrong here?

The proper solution can be found below.

>> o = property(o,'type','trial');

>> typ = property(o,'type')

typ =

trial

What we conclude here is of fundamental importance. In order to change a property of an object the result of the object manipulation must be re-assigned to the object variable. We can check whether this method works also for non-existing properties like 'data.x'.

>> o=property(o,'data.x',1:10);

>> property(o,'data.x')

ans =

1 2 3 4 5 6 7 8 9 10

And how it works – it works perfect!

# There is Still More to Say

We have mentioned that if an object property needs to be changed we have to reassign the new object value to our object variable, otherwise our object variable would still refer to the old value. This is 100% valid for the Carabao object, but there are also exceptions

MATLAB supports also so called *handle objects.* The reader can study the new style class definition (content of file *carahob.m* in directory *@carahob* ) which defines a handle class (the *…hob* stands for handle object), since it derives class *carahob* from class *handle* (see first line).

classdef carahob < handle % v1a/@carahob/carahob.m

properties % object properties

tag % object class tag

type % object type

par % object parameters

data % object data

work % object work variables

end

methods % public methods

function o = carahob % handle Class Constructor

o.tag = 'carahob'; % tag holds class name

o.type = 'shell'; % object type

o.par = []; % parameter property

o.data = []; % data property

o.work = []; % work property

end

end

end

Using the same code for a *carahob property* method and trying the following command lines based on the *carahob* object construction proves that the behavior is now different and according to a *handle object class*, as we do not have to reassign the output value of the call property(o,'data.x',1:10) to our object variable o.

>> o=carahob;

>> property(o,'data.x',1:10);

>> x=property(o,'data.x')

x =

1 2 3 4 5 6 7 8 9 10

Handle class objects are in some aspects easier to use and usually also faster in the property access. An important handle object class is MATLAB's handle graphics object class, which is supported from MATLAB version R2014. On the other hand handle class objects can very easily lead to undesired side effects thus they require much more caution during programming.

Anyway, the Carabao object class is not a handle class, and we always have to keep in mind that if we want that our object variable refers to an object with changed properties then we have to reassign the modified object to our object variable.

A final statement has to be made regarding the *eval* statement. The compact code of the *Carabao property* method supported by the *eval* functionality is highly elegant but it has its price. The *eval* function has to setup its internal exception handler, which costs a lot of CPU time. In most of our applications we will not really feel this impact. If, however, the *Carabao property* method is called repeatedly in a loop, we should carefully consider how to make use of our *property* method. Anyway it is not a big deal to replace the current *property* method by a speed optimized version (see MATLAB functions *fieldnames*, *isfield*, *getfield* and *setfield* to support this job).

# Displaying Object Properties

When we type the name of an object without a terminating semicolon MATLAB will display a short description of the object.

>> o=carabao

o =

carabao object: 1-by-1

This description is not of big value when we want to see more details about the actual property values, so we can use a cast to a structure, which will give us more insight.

>> struct(o)

ans =

tag: 'carabao'

type: 'shell'

par: []

data: []

work: []

Obviously we get now enough understanding about the property values of our object, but without explaining details I'd like to warn the reader that this method would fail, if we would derive new object classes from the *Carabao class*. In debug cases it is of tremendous help to have a display function that tells us details about the property values. So let us study the following code for the *display* method.

function display(o) % v1a/@carabao/display.m

fprintf('%s object\n',upper(class(o)));

tags = {'tag','type','par','data'};

fprintf(' MASTER Properties:\n');

for (i=1:length(tags))

tag = tags{i};

value = property(o,tag);

fprintf(' %s: ',tag);

if isstruct(value)

fprintf('\n'); disp(value);

elseif ~isempty(value)

disp(value);

elseif isa(value,'cell')

fprintf('{}\n');

elseif isa(value,'char')

fprintf('''''\n');

else

fprintf('[]\n');

end

end

fprintf(' WORK Properties:');

bag = property(o,'work');

if isstruct(bag)

fprintf('\n'); disp(bag);

elseif ~isempty(bag)

disp(bag);

else

fprintf('\n []\n');

end

end

It is not difficult to understand what the *display* method will present to us. Let's try it!

>> o=carabao; o = property(o,'data.radius',1.1);

>> display(o);

CARABAO object

MASTER Properties:

tag: carabao

type: shell

par: []

data:

radius: 1.1000

WORK Properties:

[]

In the above example we have set the value of a data property (*data.r*). Let us set also a parameter property, e.g. assign *par.color* with the value 'r' (a string indicating *red*).

>> o=property(o,'par.color','r')

CARABAO object

MASTER Properties:

tag: carabao

type: shell

par:

color: 'r'

data:

radius: 1.1000

WORK Properties:

[]

Oops – we forgot to terminate the command >> o=property(o,'par.color','r') with a semicolon, and MATLAB implicitly uses our *display* method to display the object. That's quite awesome! Now we have a convenient way to see the object property values and if we have more need for details we can use the property method to fetch some object internal values to the outside world.

# The Bag Concept

One of the most import concepts around the *Carabao object class* is the *concept of a bag*. We learned already about the 5 properties of a *Carabao* object (tag, type, par, data and work), and we can imagine a bag as a structure with these five entities.



*bag.tag*

*bag.type*

*bag.par*

*bag.data*

*bag.work*

On a metaphorical level we can imagine a bag structure as a kind of back bag with five pockets, one for each of the bag structure members. We can easily write a *pack* method which packs the properties of a *Carabao* object into a bag.

function bag = pack(o) % file @carabao/pack

bag = property(o); % get all properties

end

Let's see the *pack* method in operation.

>> bag=pack(o)

bag =

tag: 'carabao'

type: 'shell'

par: [1x1 struct]

data: [1x1 struct]

work: []

The *Carabao concept* also insists on an easy way to convert a *bag* back to a *Carabao object*. How would we implement this requirement? We could write a method *unpack* that constructs first a generic object and sets then property by property with the values of the bag's structure members. No doubt – this would work, but it makes more sense to let this *construction from a bag* be done by our *Carabao* constructor. For this reason we have to replace the preliminary version of the *Carabao* constructor by the following final one.

function o = carabao(arg) % v1b/@carabao/carabao.m

tag = mfilename; % our class tag

if (nargin == 0)

arg = 'shell'; % 'shell' type by default

end

if ischar(arg) % construct typed CARABAO object

o.tag = tag; o.type = arg; % object tag & type

o.par = []; o.data = []; % parameter & data property

o.work = []; % work property

elseif isstruct(arg)

o = arg; % construction from a bag

elseif isobject(arg)

o = pack(arg);

else

error('bad argument class (arg1)!');

end

o = class(o,tag); % construction of class object

if (nargout == 0)

launch(o); % launch a shell for the object

end

end

Our constructor can now be called with zero or one input arguments. The call with no input argument o = carabao is equivalent to the (one argument) call o = carabao('shell') which initializes the *type* property with the value 'shell'.

The type property will determine which kind of shell will be launched, and the type also determines the interpretation of the data and is associated with type specific data integrity rules. But let's first try the constructor with one input argument which we choose to have the value 'plot' in order to initialize the type property accordingly.

>> o=carabao('plot')

CARABAO object

MASTER Properties:

tag: carabao

type: plot

par: []

data: []

WORK Properties:

[]

# Construction From Bag

A quick check tells us that everything looks good and straight forward. Let's now move to construction from a bag. We still have the bag variable in the MATLAB work space with values from the previous object packing. Let's construct an object from the bag.

>> o=carabao(bag)

CARABAO object

MASTER Properties:

tag: carabao

type: shell

par:

color: 'r'

data:

radius: 1.1000

WORK Properties:

[]

That works well. That means we have now a method *pack* that converts an object to a bag structure, and we can use the new Carabao constructor to construct an object from the bag data. Thus the following composite operations leave the value of the *object* variable o or *structure* variable bag invariant.

>> o=carabao(pack(o)); % does not change o

>> bag=pack(carabao(bag)); % does not change bag

Finally our new constructor can also construct from an object.

>> oo=carabao(o); % copy constructor

This kind of construction is called *copy construction*. It is not difficult to understand how it works. In the second *elseif* clause of the *Carabao* constructor the object calls the *pack* method to pack the input argument into a *bag structure* (here assigned to the variable o) in order to be processed later on to the Caramba class object. But what should such a copy construction be good for – a simple assignment >> oo=o; would do the same job? That is true, but wait a minute. The copy construction would also work for any other class object that provides a proper pack method. Thus we realize now that the copy construction has universal character and there are several cases where the copy construction is very useful. Consider the case where Alice uses a class *alice* derived from *carabao* to import log files and storing the data in the Carabao data part. Then Bob could make an agreement with Alice to share the same data structures, and derive his own object class bob from class carabao, which he provides with analysis algorithms. Bob could utilize the following scenario.

>> ao = alice(sometype); % construct an alice object

>> ao = import(ao); % import data into alice object

>> bo = bob(ao); % copy construct a bob object

>> bo = analysis(bo); % apply analysis to bob object

Almost everything has been said now about the revised *Carabao* constructor, except when we invoke the constructor without output arguments, like shown in the following example, since the if-statement at the end of the Carabao constructor will invoke a launch call.

carabao; % launch a Carabao shell

carabao('tiny'); % launch a 'tiny' Carabao shell

For this kind of calling syntax we always expect that a shell is being opened which is associated with the object. The m-file name of the shell matches the type name. Since a construction of a generic object (constructor with no input arguments) would initialize the type property with the value 'shell' the launch method will apply the *shell* method of the object to launch a shell. If this is not desired, e.g. we want to use a method *tiny* to launch a tiny shell then we have to initialize the *type* property with the value 'shell', e.g. by invoking

carabao('tiny'); % launch a 'tiny' Carabao shell

That's all for now with shell launching. We will come back later to this topic.

# Mass Storage Migration Issues

We mentioned in the introduction that we expect the *Carabao* object class to support us with mass storage management functionality, e.g. methods to save the object to a data base or load it respectively from a data base. MATLAB has fast *save* and *load* functions to store objects with complex property structures to a *.mat* file, and it is our strong intention to utilize these two functions for fast object mass storage access.

But first let us review the requirements regarding mass storage capabilities.

* the object class provides basic mass storage management, which means it has save and load methods to store and retrieve the object to/from a mass storage data base
* the save/load methods are also capable to manage the storage of derived object classes with higher complexity
* during object save also the current refresh function will be saved, and after object load the shell of an object can be re-launched which includes invoking the saved refresh function
* the storage form should allow to save an object class version n, and to load the object into a class version n+m, to allow future modifications of a class definition

The last requirement is the tricky one. We need to get more awareness about the potential issues that can arise. Consider two preliminary *Carabao* methods *save* and *load* which we might used for object mass storage management.

function save(o,path) % v1a/@carabao/save.m

save(path,'o'); % MATLAB save to .mat file

end

function oo = load(o,path) % v1b/@carabao/load.m

items = load(path); % MATLAB load from .mat file

oo = items.o; % fetch from loaded object list

end

Let's try our Carabao save method. We need to change to a proper working directory where we can create files.

>> save(o,'myobject.mat');

Invoking the call save(o,'myobject.mat'); will consult MATLAB's *save* function which saves our *Carabao* object to file *myobject.mat*. Let's do the inverse operation to load the object from file *myobject.mat*. Our *load* method consults the MATLAB *load* function which returns all objects/variables stored in file *myobject.mat* to be stored in variable *items*. From there the object to be returned can be picked as *items.o*.

>> oo=load(carabao,'myobject.mat')

CARABAO object

MASTER Properties:

tag: carabao

type: shell

par:

color: 'r'

data:

radius: 1.1000

WORK Properties:

[]

Everything works well, as we can cross check, so where is the challenge?

# Deriving Classes from the Carabao Class

The challenge arises if we deal with derived class objects with changing class definitions. Assume we derive a class *carabaobull* from class *carabao*. Let us denote this relation by

carabaobull < carabao

Assume now that we store a *carabaobull* object to a file mybull.mat using the Carabao method *save*. Assume that three years later we decide to rewrite the class definition of the *carabaobull* object by introducing an intermediate carabaokid object class, according to the scheme

carabaobull < carabaokid < carabao

MATLAB will get in troubles if the actual (new) class definition does not match the class definition that was stored in the .mat file.

Let us study this scenario in some detail. Doing so we can exercise working with the methods we introduced, and we will also get insight how classes are derived from the *Carabao* object class.

Let's get started: Create a @carabaobull directory, after that create a constructor *@carabaobull/carabaobull.m*.

function o = carabaobull(arg) % file @carabaobull/carabaobull.m

if (nargin == 0)

bob = carabao; % create (carabao) base object

else

bob = carabao(arg); % create (carabao) base object

end

tag = mfilename; % target class name

o = class(struct([]),tag,bob); % construct derived object

o = property(o,'tag',tag); % set tag

if (nargout == 0)

launch(o);

end

end

We still have a bag with object properties, so let us construct a *carabaobull* from the bag and save the object to *mybull.mat*.

>> o=carabaobull(bag)

CARABAOBULL object

MASTER Properties:

tag: carabaobull

type: shell

par:

color: 'r'

data:

radius: 1.1000

WORK Properties:

[]

>> save(o,'mybull.mat');

Loading the object shows that there is no issue so far (note that the first argument can be any *Carabao* object or class object derived from *Carabao* object).

Now let us redesign the *carabaobull* object class. Introduce a further (itermediate) class *carabaokid*.

function o = carabaokid(arg) % file @carabaokid/carabaokid.m

if (nargin == 0)

bob = carabao; % create (carabao) base object

else

bob = carabao(arg); % create (carabao) base object

end

tag = mfilename; % target class name

o = class(struct([]),tag,bob); % construct derived object

o = property(o,'tag',tag); % set tag

if (nargout == 0)

launch(o);

end

end

Finally change the class definition of the *carabaobull* object by deriving the base object (bob) from *carabaokid* instead of *carabaobull*.

function o = carabaobull(arg) % file @carabaobull/carabaobull.m

if (nargin == 0)

bob = carabaokid; % create (carabaokid) base object

else

bob = carabaokid(arg); % create (carabaokid) base object

end

tag = mfilename; % target class name

o = class(struct([]),tag,bob); % construct derived object

o = property(o,'tag',tag); % set tag

if (nargout == 0)

launch(o);

end

end

To test the new constructor we can construct an object, however, MATLAB will respond with an error and suggests to clear first all classes. When we clear all classes also all variables in the current work space get lost. Luckily we have stored a carabao object previously to disk and are thus able to reconstruct the bag.

>> clear classes

>> bag=pack(load(carabao,'myobject')); % reconstruct bag

After these preparation we can test the modified constructor which obviously works well.

>> o=carabaobull(bag)

CARABAOBULL object

MASTER Properties:

tag: carabaobull

type: generic

par:

color: 'r'

data:

radius: 1.1000

WORK Properties:

[]

Let us now load the previously stored carabaobull object from file mybull.mat. We expect that MATLAB comes with an error.

>> oo=load(carabao,'mybull.mat')

Warning: Fields of object '' do not match the current

constructor definition for class 'carabaobull'. The object has

been converted to a structure.

> In carabao.load at 2

oo =

carabao: [1x1 carabao]

Now we can see the nature of the migration issue crystal clear: we saved an object with class structure [carabaobull < carabao], but when we were going to load the object the class structure had already changed to a structue [carabaobull < carabaokid < carabao] , what caused MATLAB to respond with an error.



# The Bag Concept Solves the Issue

For the solution of the migration issue we have prepared already all building blocks. Instead of saving an object directly to mass storage (the file system) we pack the object into a bag and store then only the bag to the file system. Once all parameters and data are in the bag MATLAB has no clue how to assign the bag to an object structure, and therefore no reason to complain in case of object structure changes.



When we want to load the object from mass storage we actually load only the bag. We can trust, however, that the bag contains a property *tag* which tells us which constructor we actually have to consult for the construction of an object from our bag.

Let's see how our save and load methods have to be modified to be adopted to the bag concept.

function save(o,path) % v1c/@carabao/save.m

bag = pack(o); % requires class support of PACK

save(path,'bag'); % save bag

end

function oo = load(o,path) % v1c/@carabao/load

items = load(path); % load all saved items

bag = items.bag; % get bag from items

construct = eval(['@',bag.tag]); % function handle to constructor

oo = construct(bag); % construct from bag

end

So far the compatibility issue is solved with version v1c save & load. Now we can save an object in any object class version by knowing that any save operation will first convert the object to a bag structure which is actually stored to mass storage. After that we are free to change our object class to any internal structure as long as the class name does not change.

When we consult the load method we will first load the data into a bag structure. From the bag structure we take the tag entry to tell us which class should be constructed. Then the load method uses an *eval* statement to construct a function handle for a constructor. If the value of *bag.tag* equals 'carabao' then the mystery command construct = eval(['@',bag.tag]) is equivalent to the command construct = @carabao which creates a function handle to a Carabao class constructor. That's how our load method can reconstruct the object, whatsoever the class version details are.

# Corona Services

Still we are not at the final implementation level for our mass storage methods save & load.

* we want to have file selection dialog boxes for save & load
* the selected directory to/from which we saved/loaded our object should be remembered if we open a dialog box repeatedly
* the save/load mass storage access functions should leverage the Corona services for mass storage save & load.

Corona services? This sounds like there is some powerful 'Corona mass storage server' which will be consulted for save and load of MATLAB objects. That is 100% correct. But the word 'powerful' must not be confused with a high performance hardware database server. The meaning is more regarding conceptual strength of the Corona concept, the Corona server itself is a tiny MATLAB object class based on 9 methods and exactly 200 lines of MATLAB code lines for the whole object class (comments not included).

So which tasks does the Corona server solve? Think of an object browser which can display MATLAB objects in a tree view style.

<to be continued>

# Carabao Shit

Let us do a bit graphics. It would be nice to provide some data to a Carabao object and then plot a graphics of a carabao which depends on the object parameters. We could download an image of a carabao from the internet and display it in graphical shell. But how can we easily parametrize an image to get different personalities of the displayed carabao. I saw no possibility to draw a 3D carabao with a few MATLAB commands, but there is a possibility to draw carabao shit using 4 MATLAB lines.

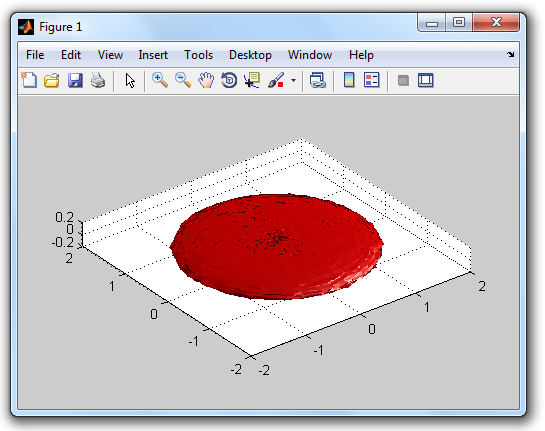
[X,Y,Z] = sphere(75);

f = @(D) 1.5\*D + 5\*randn(size(D))/250;

surf(gca,f(X),f(Y),0.1\*f(Z),'FaceColor','r','LineStyle','none');

light('color',[1 1 1]); daspect([1 1 1]); shg;

MATLAB brings up the following picture. It's easy to parametrize this kind of 3D objects and to give them different colors. Red, green or blue carabao shit can only exist in a virtual world, so let's use parametrized 'virtual carabao shit' for the visualization of our demo objects – why not!



Let us parametrize an object and write a plot function which allows us to display virtual carabao shit. Let us use roughness and color as parameters and the radius in the data part.