# Solo core functions – User's guide

by Carlos Brody

Draft of January/February 2006

## 1 Overview

The Solo core functions are a set of Matlab routines designed to assist in the building of complex user interfaces. We will principally use them in the context of writing control protocols for behavior and electrophysiology, but they could be used for other interfaces as well.

In a complex interface, one usually has many parameters that are controllable through user-interface graphics objects – for example, menus, pushbuttons, editable fields, etc. Sometimes there are so many of these that keeping track of all of them conflicts with a different, but highly desirable aspect of programming: *compartmentalization*. When writing a large program, keeping different parts of it separate from each other is an enormously useful way of breaking the problem down into manageable chunks. Typically, one bases the program on many small functions, each of which keeps its internal variables private and secret from the others. That way, each of the small functions can be cleanly and completely debugged, quite separately from the others.

So, how do we separate variables that might all have values determined by objects in the same graphics window? The Solo core functions are designed to address this problem.

In addition, (1) Solo core functions implement a functionality available in Exper: having variables internal to a program whose value is automatically tied to graphics interface objects. (2) Solo further enhances that functionality by using Matlab object-oriented programming to allow a syntax that is easier to read and write than the one available in Exper. (3) Solo allows passing variables by reference, in either read/write or read-only modes, a functionality otherwise unavailable in Matlab.

# 2 SoloParamHandles : variables by reference

In Matlab, variables are always passed by value[1], not by reference. But for graphics interfaces, we often want to pass a variable by reference. For example, a variable that is tied to a menu object might be passed to a function f() that might want to change the value of the menu object. If that happens, every other function that accesses the value of the menu object, including the function that called f(), needs to be aware of what the new menu value is.

*What does passing a variable by value mean?*

Suppose the beginning of function foo is defined as:

```
function [s] = foo(a)
```

And now suppose you call function foo from the base workspace as:

```
>> a = 10; foo(a);
```

Then, within function foo(), "a" will have the value 10. But if, within foo(), you change the value of "a," this will have no effect on variable "a" in the base workspace. This is because the variable "a" within function foo() acquired only the *value* of the variable "a" in the base workspace, but is a completely separate variable.

SoloParamHandles are Matlab variables that are always, automatically, passed by reference. SoloParamHandles were originally created to be used for graphics objects, and that was the motivation for making them variables that are passed by reference. But SoloParamHandles need not be tied to graphics objects: they can also hold arbitrary values (cells, matrices, strings, etc.) that have nothing to do with the graphics interface.

SoloParamHandles are created through calls to a function of the same name ("`SoloParamHandle`"). This function takes as its first argument the *owner* of the variable, and as a second parameter the name of the variable to be created. It can also take an optional argument that tells it the initial value of the SoloParamHandle. For example, to create a SoloParamHandle that isn't owned by any object class, we can use the special owner designation 'base', and we can do the following:

*SoloParamHandle owners and Matlab objects*

We will be using SoloParamHandles in conjunction with Matlab objects a lot. Objects are like data types that you can define; once you define the datatype (called a class), you can make instantiations of objects of that class. To define an object class named 'newguy', you make a directory in Matlab's path called '@newguy', and an m-file inside it of the same name: '@newguy/newguy.m'. This m-file should return an instantiation of objects of class 'newguy.' Inside the @newguy/ directory you can define other functions that operate on objects of this class.

Typically, each behavior protocol we write will have a corresponding object class. The nice thing about this is that different protocols then each get their own directory in which they can have functions that don't conflict with each other. For example, say Hatim wrote a protocol that had a method called @HatimProt/GiveReward.m; say LungHao also wrote a protocol, and wanted to use the same function name in his protocol's directory: @LungHaoProt/GiveReward.m. If we then type

```
>> GiveReward(obj)
```

Hatim's function will be called if obj is a HatimProt object, whereas Lung-Hao's function will be called if obj is a LungHaoProt object. In other words, objects help to keep function names separate. SoloParamHandles either (1) have an owner which is an object class; or (2) have the owner 'base', for base workspace.

[1] An exception is that variables passed to mex-files (e.g., functions written in C) are passed by reference, not by value. However, we will not be writing any mex-files, nor will we be asking users to do that.

```
>> SoloParamHandle('base', 'myvar', 'value', 10);
```

This will create, in the workspace in which we ran the above line, a variable called "myvar," which will have the value 10. To examine the value of this variable, we can type:

```
>> value(myvar)

ans =

   10
```

To *set* the value of this variable, we can type:

```
>> myvar.value = 20;
>> value(myvar)

ans =

   20
```

Now suppose that we define function foo() as:

```
1   function [] = foo(a)
2
3   a.value = value(a)*2;
```

SoloParamHandles are always passed by reference. Thus the following lines cause the value of myvar to be multiplied by 2 each time that foo(myvar) is called:

```
>> SoloParamHandle('base', 'myvar'); myvar.value = 10; foo(myvar);
>> value(myvar)

ans =

   20

>> foo(myvar); value(myvar)

ans =

   40
```

Note that in this example, we created the SoloParamHandle without a value, and only after its creation did we assign a value. Assigning the value at the same time as creation of the variable is optional.

# 3 Operator syntax

You will have noticed, in the examples above, that accessing and setting the value of a SoloParamHandle is a little clumsy – in addition to the name of the SoloParamHandle, one has to type in the letters for "`value`." If you use SoloParamHandles a lot, pretty soon you might find your code littered all over with the word "`value`"! But in many cases, Matlab's object operator syntax allows us to define SoloParamHandles so that the value is implicitly accessed. For example, if you want to get the result of adding 5 to the value of a SoloParamHandle, you can simply use the + sign, as in this example:

```
>> myvar.value = 25; myvar + 5

ans =

   30
```

Matlab knows that if a + sign is used with a SoloParamHandle, this means "get the value and add something to it." The same is true for many other operators – times, minus, greater than, etc. By dropping all the redundant "`.value`" parts from the program statements, the program becomes much more readable. Thus, for example, we could redefine our function foo() from above as

```
1    function [] = foo(a)
2
3    a.value = a*2;
```

And it would still work exactly as before.

There are two cases in which writing "`value`" is unavoidable: First, *setting* the value always requires the "`.value`" construct, as in "`myvar.value = 20;`" The second, other case is when you *access* the value of a SoloParamHandle by itself, without any operator. If there is no operator, and you want the value of the SoloParamHandle, you need to surround the SoloParamHandle with "`value()`" as in the second line of this example:

```
>> myvar.value = 5;
>> value(myvar)

ans
```

*Common Bugs!*

Writing "`myvar = 10`" when really you meant "`myvar. value = 10`" is very common! What'll happen if you use the first version is that `myvar` will no longer be a SoloParamHandle. Instead, it will be a double with the value 10.

Another common bug happens when you want to pass the value of a SoloParamHandle to a function. For example, you want to write "`foo(value(myvar))`" but you forget to write the value() part and you write "`foo(myvar)`". Watch out for these!

```
= 5;
```

Note the asymmetry between *accessing* a value, which is done with "`value()`" and *setting* a value, which is done with "`.value`". It would be nicer if one could do both of these things with the same construct, but sadly, for technical Matlab reasons, one can't.

These are the operators currently defined for use with SoloParamHandles:

```
==   >=   >   <=   <    -   /   +   *   ./   .*  .^  ^
```

In addition, the following functions operate directly on the value of a SoloParamHandle:

```
length()   double()   logical()   char()
```

If you want to add more functions or operators, talk to one of the Solo core developers—it's fairly easy to add them. (For example, if you'd like `exp()` to operate directly on the value of a SoloParamHandle, that'll be easy to add.)

# 4 Subscript syntax

## *4.1 Cell subscripts*

As mentioned above, SoloParamHandles can have a value that is any of the types that Matlab handles. This includes matrices and cells. For example, we could say:

```
>> SoloParamHandle('base', 'myvar', 'value', cell(3,1));
>> value(myvar)

ans =

   []
   []
   []
```

That is, we've created a SoloParamHandle whose value is a 3-by-1 cell, the elements of which are all empty. Suppose we wanted the first element to be the string 'foo' and the second element to be the number 7, while leaving the third element unchanged. We could do this:

```
>> v = value(myvar); v{1} = 'foo'; v{2} = 7; myvar.value = v;
>> value(myvar)

ans =

   'foo'
```

```
   [  7]
   []
```

However, once again Matlab syntax allows making things a little more readable. When a curly brace is right next to a SoloParamHandle, Matlab knows that this means "oh, the value of the SoloParamHandle must be a cell, and the user is trying to access the contents of the cell." So we can actually write the much more direct lines:

```
>> myvar{1} = 'foo'; myvar{2} = 7;
>> value(myvar)

ans =

   'foo'
   [  7]
   []
```

Similarly, if you just ask for `myvar{1}`, Matlab knows how to interpret this—you're trying to access the contents of a SoloParamHandle whose value is a cell:

```
>> myvar{1}

ans =

foo
```

You are not limited to just cells that are vectors (i.e., have one subscript). They can have as many as Matlab allows. Thus, for example, "`myvar{2,3}`" is perfectly valid syntax.

If you use the curly bracket notation when the value of a SoloParamHandle is *not* a cell, you will, of course, get an error.

## *4.2 Matrix subscripts*

Exactly the same process applies to matrix subscripts with parentheses instead of curly brackets. For example, you can do this:

```
>> myvar.value = zeros(2,2);
>> myvar(1,2) = 3;
>> myvar(2,1) = 1;
>> value(myvar)

ans =

   0  3
   1  0
```

# 5 GUI SoloParamHandles

The main motivation for writing the Solo system was to assist in user interfaces; so in this section we now we finally get to the source of it all, the user interface!

Interactive graphics objects often have "callback" functions attached to them: when the user interacts in some way with the graphics object, the callback corresponding to it gets called. This allows for great flexibility (and we will talk much more about callbacks below), but quite often all one wants from the callback is to make the internal variable, corresponding to the GUI, have a value that matches whatever the user sets in the GUI. SoloParamHandles (following the functionality of Exper) do this automatically, so users don't have to write callback functions for this.

There are a number of predefined graphics objects types in Solo, all of which share some optional parameters and settings. We'll go through one of them in detail – the Menu type – and use it to illustrate some of the common parameters. After that we'll go a little faster through the other types (Edit, Numedit, Toggle, Pushbutton, Header, and Subheader).

## *5.1 Menus*

Ok, let's start with the Menu example. We'll open a new figure, and put a pop-up menu on it. There will be three possible items on the menu, which we'll give the strings 'blank', '10', and '20'.

```
>> figure;
>> SoloParamHandle('base', 'menuvar', 'type', 'menu', 'string', …
      {'blank', '10', '20'}, 'value', '10', 'label', 'menuvar');
```

This will bring up the following picture in your figure:


We can access the value of this SoloParamHandle just as we access the value of any other SoloParamHandle:

```
>> value(menuvar)

ans =

   10
```

There's an extra useful wrinkle, which we can see if we ask what type of object the value of the menuvar is:

```
>> class(value(menuvar))
```

```
ans =

   double
```

Menu items are defined as strings. However, the Solo system checks to see whether the string for the current menu value is something that can be converted into a number. If it is, then the value that is stored is the number itself. The reason for this is that most often, when the value of a menu is the string for a number, we want the number. Thus we can do:

```
>> menuvar + 5

ans =

   15
```

Whenever a user changes the graphics object, the value of menuvar, the SoloParamHandle, will also auomatically change. Try it! Change the menu on the figure, and then type `value(menuvar)`.

Conversely, if the value of `menuvar`, the SoloParamHandle, changes, the graphics object will also automatically change— the graphics object and the variable are tied together. Thus you can try:

```
>> menuvar.value = 'blank';
```

or

```
>> menuvar.value = '10';
```

or

```
>> menuvar.value = 10;
```

Notice that the last two have exactly the same effect. When you set a menu SoloParamHandle to a numeric value, the system (1) first checks to see whether a string that matches that number exists in the menu; if so, sets the menu to that value; (2) if approach #1 failed, the system then checks to see whether the number is an integer and there are at least that number of items in the menu; if so, sets the menu to the value of that item number. Thus, using our example, "`menuvar.value = 1;`" would result in setting the value to 'blank', the first of the items in the menu.

### 5.1.1 Optional parameters common to all graphics SoloParamHandle types: 'position' and 'TooltipString'.

*Position.* Suppose you want to add a second menu to your window. You don't want this new menu to overlay the existing one, of course! An optional parameter when creating any GUI SoloParamHandle, is `'position'`, a 1-by-4 vector specifying [x y width

height], in units of pixels. The x and y refer to the position of the lower left hand corner of the GUI, relative to the lower left hand corner of the figure; width and height are again, in pixels, and indicate the total width and height of the GUI and the label next to it.

So, for example, we can create:

```
>> SoloParamHandle('base', 'secondmenu', 'type', 'menu', 'string', …
      {'another' 'menu'}, 'label', 'secondmenu', 'value', 2, …
      'position', [100 100 150 20]);
```

*TooltipString.* For most control windows and most protocols, you will soon find that you have a million menus and buttons and parameters, and remembering what each of them does is quite difficult. It is very convenient to have a little text window with explanatory text that pops up whenever the mouse rests over the GUI or the label. To do this, you can either add a tooltip string after creation, or pass it as an optional parameter during creation. For example.

```
>> set_tooltipstring(secondmenu, 'This is the second menu');
```

Place the mouse over secondmenu to see the effect of this. Alternatively, if defining the tooltip string upon creation, you could have done:

```
>> SoloParamHandle('base', 'secondmenu', 'type', 'menu', 'string', …
      {'another' 'menu'}, 'label', 'secondmenu', 'value', 2, …
      'position', [100 100 150 20], 'TooltipString', 'version 2');
```

To do multiple lines, you can do it like this:

```
>> set_tooltipstring(secondmenu, sprintf('First line;\nSecond line.'));
```

### 5.1.2 MenuParam() shortcut function

A slightly shorter way of creating a menu type SoloParamHandle is with a utility function called MenuParam.m:

```
function [sp] = MenuParam(obj, parname, parlist, parval, x, y)
```

The first argument, as usual, is either an object or the string 'base', and the second is the name of the variable to be created. The third is the list of strings for the menu items. The fourth indicates the initial value of the menu. The fifth and sixth are the position, in pixels, of the lower left corner of the menu object. The width and height take on a default value. Thus we can call

```
>> MenuParam('base', 'thirdmenu', {'a third' 'menu'}, 2, 140 100);
>> value(thirdmenu)

ans =
```

```
menu
```

After the first six obligatory parameters, MenuParam.m takes other, optional arguments, one of which is `'TooltipString',` for example:

```
>> MenuParam('fourthmenu', {'still' 'another'}, 1, 140, 200, …
      'TooltipString', 'detailed tooltips are better than short ones');
```

See MenuParam.m's documentation (under construction) for information on other optional parameters.

## 5.2 Numerical edit fields

Now try

```
>> NumeditParam('base', 'numeditvar', 200, 10, 240);
```

This will produce a numeric editable field:


The first argument to `NumeditParam.m` is, as usual, either an object or the string 'base'; the second is the name of the variable to be instantiates; the third is its initial value; and the fourth and fifth are the position, in pixels, of its lower left hand corner. `NumeditParam.m` can take, as optional arguments, '`TooltipString`' or '`position`'.

If you try to edit the entry in the `NumeditParam`, it will accept new values only if they are numeric; non-numeric text is interpreted as a typo and ignored.

## 5.3 String edit fields

```
>> EditParam('base', 'editvar', 'brown', 10, 260);
```

will produce:


`EditParam.m` is just like `NumeditParam.m`, except its entries can also be non-numeric strings, as in this example. If you do enter a numeric value, you can do usual arithmetic things with it:

```
>> editvar.value = 10; numeditvar + editvar

ans =

   210
```

## 5.4 Display params


```
>> DispParam('base', 'dispvar', 'this', 10, 10);
```

## 5.5 Headers


```
>> HeaderParam('base', 'headervar', 'A header', 10, 10);
```

## 5.6 Subheaders


```
>> SubheaderParam('base', 'headervar', 'A header', 10, 10);
```

## 5.7 Toggle buttons


```
>> ToggleParam('base', 'togglevar', initval, x, y);
```

Toggled take only the values 0 or 1.

## 5.8 Pushbuttons


```
PushbuttonParam('base', 'buttonvar', x, y);
```

Pushbuttons have no defined value. They're useful for generating callbacks (see below).

### 5.9 Sliders


```
SliderParam('base', 'slidervar', initval, minval, max, x, y);
```

### 5.10 Logarithmic sliders


```
LogsliderParam('base', 'slidervar', initval, minval, max, x, y);
```

## 6 SoloParamHandle callbacks

Perhaps you want the program to respond in some special way when a user changes one of the GUI parameters—update a graph, for example, or, based on a menu entry, change to a new type of state matrix. You can do this using *callbacks*—functions that are automatically called after a change to the GUI.

At this writing, you can use callbacks for SoloParamHandles only when their owner is an object. So let's make a really simple object to run some examples an tests on. Make a directory called `@myobj`, and within it make an m-file called `myobj.m` that contains the following lines:

```
function [obj] = myobj(a)

   obj = class(struct, 'myobj');
```

This is the simplest possible object class: objects of this class contain no data, and so far, can't do anything very interesting. Currently, in the Solo system, only one object of any one class can exist at one time. This will change in future releases, but for now, that's how it is. We'll add the following lines to every object creator function when we're going to use that object class with SoloParamHandles:

```
if nargin==1 && ischar(a) && strcmp(a, 'empty'), return; end;
flush_solo(['@' mfilename ]);
```

These will be standard lines—you don't need to understand what they do, as long as you always use them! But, for the curious, the first one says "only proceed if you have the single argument `'init'`"; and the second one says "clear any Solo system entries related to previous instantiations of this object class."

Ok, with these lines in `myobj.m`, let's create one of the `myobj` objects. In Matlab's base workspace, type:

```
>> m = myobj('init')

m =

    myobj object: 1-by-1
```

Now let's make a menu that is owned by this object class. First we'll clear the figure to get rid of all the old example GUIs we'd created.

```
>> clf;
>> MenuParam(m, 'amenu', {'this' 'that'}, 'this', 10, 10);
```

The owner of this variable is the `myobj` class:

```
>> get_owner(amenu)

ans =

@myobj
```

Ok. Now we get into the interesting stuff! Let's add an m-file called `amenu.m` inside the `@myobj/` directory. Let the contents of that file be these:

```
function [] = amenu(obj)

   fprintf(1, 'Hello world\n');
```

Now change the menu GUI, and see what happens. Every time there is activity in the GUI, the system checks whether there is a file with the same name as the SoloParamHandle (in this case, `amenu.m`) in the directory of the object that owns the SoloParamHandle (in this case, `@myobj`). If the file exists, it gets called.

This is fine and good. But often we have many many GUI objects, and if we made a new file for each of them that causes a callback, we'd end up with a gazillion files. Having the SoloParamHandle call the file with its own name if it exists is simply the default behavior. We can change the behavior by telling the system which function we want to have used as a callback. Add a file called `tester.m` to the `@myobj/` directory, and in that file put this:

```
function [] = tester(obj, arg1)

   fprintf(1, 'My argument was "%s"\n', arg1);
```

And now, from the command line, register this function as the callback for `amenu`, by typing:

```
>> set_callback(amenu, {'tester', 'Uncle Richie was a crazy dude!'});
```

Now change the menu GUI and see what happens. When you register a callback in this way, you can specify as many arguments as you like to the callback function[2]. In this example we specified one, but we could have specified zero or more. (E.g., if we had typed `set_callback(amenu, {'tester'});` we would have been specifying zero arguments for tester.)

Furthermore, you can have more than one callback per SoloParamHandle. For example, change `amenu.m` so that its first line now reads:

```
function [] = amenu(obj, arg1)
```

And now register both `tester.m` and `amenu.m` as callbacks for SoloParamHandle `amenu`:

```
>> set_callback(amenu, {'tester', 'Uncle Richie was a crazy dude!'; …
           'amenu', ''});
```

Each row in the cell passed as the second argument to `set_callback` is interpreted as specifying another function to be called; the functions are called in the order in which they are found in the cell. Note that because all rows of the cell must have the same number of columns, this means that all the callback functions must be capable of accepting the same number of arguments (which is why we added the unused `arg1` to `amenu.m`).

So far, so good. But what if you need to know the *value* of `amenu` or of other SoloParamHandles inside those callback functions? In order to explain how to do this, we need to explain where and how SoloParamHandles and their values are made visible—that is, their "scope."

# 7 SoloParamHandle scope

One key goal of writing the Solo system was to allow compartmentalization. For example, one section of your protocol program might be about how much water reward to give. Another section of your program might be about what the correct response port is in each particular trial. Sometimes you will want these two things to be related. But often they are completely independent—for example, when water reward magnitude is constant throughout a session. In the latter case, the two sections should be completely compartmentalized, separate from each other: the water reward section of the program should not be able to modify which of the ports is the correct response port. When things are compartmentalized this way, they are easier to debug: If you know, *for certain,* that nothing in the water reward section of the program could affect choice of correct response port, then you won't ever have to even glance at the water section of the program if you are debugging something to do with the response ports.

---

[2] NOTE: the very first argument to a function that belongs to an object class – also called a *method* for that object class – must always be an object of that class. Any arguments that we specify for SoloParamHandle callbacks are in addition to that first parameter.

In order to facilitate compartmentalization, SoloParamHandles are automatically limited in their scope. Normally, if a SoloParamHandle is declared in function `foo()`, other functions or the base workspace know nothing about it: they can't change its value, they can't read its value, they don't even know it exists! – unless you explicitly give them access to the SoloParamHandle.[3]

Going back to the menu example from the previous section, suppose you wanted `tester.m` to have access to the `amenu` SoloParamHandle. To do this, we must do two things: (1) We add the following special call at the very top of `tester.m`:

```
GetSoloFunctionArgs;
```

What this call does is to go to the variable registry, look up all the SoloParamHandles that have been registered for the calling function, and instantiate them inside that function. The second thing we have to do is:

(2) add `amenu` to the registry of SoloParamHandles that `tester.m` has access to:

```
>> SoloFunctionAddVars('tester', 'func_owner', '@myobj', …
        'rw_args', 'amenu');
```

Now, when `GetSoloFunctionArgs` is called from within `tester.m`, a variable named `amenu` will be created, whose value will be the same SoloParamHandle that you registered.

Let us now change the body of `tester.m` so that it looks like

```
fprintf(1, 'The value of amenu is "%s" and my argument was "%s"\n', ...
           value(amenu), arg1);
```

Now see what happens when you change the menu GUI.

## *7.1 By default, SoloParamHandles are local and persistent*

From now on, we will assume that every function you write that involves SoloParamHandles has as its very first line `GetSoloFunctionArgs`; doing that by habit will be good practice.

By default, SoloParamHandles are *local* to the function in which they were declared. This means that only the function that created them has access to them—no other function or workspace even knows they exist, unless you explicitly take the step to grant access using `SoloFunctionAddVars.m`. You don't have to declare a SoloParamHandle as known to its own function. It works that way by default.

---

[3] For this reason, two different functions can have SoloParamHandles of the same name without the two variables interfering with each other.

Now, if `GetSoloFunctionArgs` is the first line in your function, then SoloParamHandles declared in your function are also *persistent.* A "persistent" variable is one whose value is maintained in between separate calls to a function. This can be very useful when a program is driven by external events (such as rodent behavior or interaction with a user through a GUI): the various functions of the program will be called time and time again in response to those external events, and having variables that remember what happened on previous calls is essential.

When you call `GetSoloFunctionArgs` from some function, say function `foo()`, the system asks: "which SoloParamHandles were declared previously within function `foo()`? Let's find them all and instantiate them, with their current values, within function `foo()`." The system then proceeds to ask, "were any other SoloParamHandles registered as visible to `foo()`? If so, let's instantiate them, too."

Thus, for example, you might write a function within an object's directory (this is also called a *method* for that object class), and have that function look like this:

```
function [] = Section1(obj, action)

GetSoloFunctionArgs;

switch action,
  case 'init'
    MenuParam(obj, 'mymenu', {'oogle' 'boogle'}, 'oogle', …
        10, 10);
    set_callback(mymenu, {'Section1', 'menu'});

    SoloParamHandle(obj, 'my_count', 'value', 0);

    NumeditParam(obj, 'numvar', 0, 10, 40);

  case 'menu',
    my_count.value    = my_count + 1;
    fprintf(1, 'Section 1''s value is now %s\n', value(mymenu))
    fprintf(1, ['My count of changes is %d\n'], value(my_count));
end;
```

Suppose that this function is part of the directory `@myobj/`, i.e., it is a method for `myobj` objects. You could then do this:

```
>> m = myobj('init'); Section1(m, 'init');
```

This `init` call to `Section1` will put up a menu at position (10, 10) of the current figure and register a callback to it that calls `Section1` if the menu GUI is ever changed, but with argument `'menu'`. If you do change the menu GUI, the SoloParamHandles `mymenu` and `my_count` will get instantiated. Since they are persistent, `my_count` will remember its value in between calls. Incrementing it by one every time the callback gets called means that `my_count` will keep track of the number of times that the meny GUI has been changed.

In the above example, the NumeditParam called `numvar` isn't doing anything—we're going to use it later below, when we discuss autosetting values.

## 7.2 Granting SoloParamHandle access read/write and read-only

You might have noticed, in the examples used so far, that when we've used `SoloFunctionAddVars.m` to give a function access to a SoloParamHandle, we used the keyword `'rw_args'`. This gives read/write access to the SoloParamHandle: i.e., the function receiving access can modify the value of the SoloParamHandle. One way to reduce bugs is to give read/write access only when you *really* have to: if instead you give read-only access, you can let a function read the value of a SoloParamHandle and use that value for its own purposes, but it is guaranteed that it can't change the value, so you never have to worry about that happening. To give read-only access, you also use `SoloFunctionAddVars.m`, but you now use the keyword `'ro_args'`. In fact, you can use both keywords in the same call.

For example, to give function `foo.m` read-write access to SoloParamHandles `blooh` and `blah`, but read-only access to `dweeb`, you would write:

```
SoloFunctionAddVars('foo', 'rw_args', {'blooh', 'blah'}, …
    'ro_args', 'dweeb');
```

Notice that when listing the names of the SoloParamHandles, you can either use a single string if you want to refer to a single SoloParamHandle, or a cell vector of strings to refer to several. Notice also that when called from within an object's method, `SoloFunctionAddVars` doesn't need to specify `'func_owner'`—by default, it knows that the `func_owner` is the object class name.

When `GetSoloFunctionArgs` instantiates a SoloParamHandle for a function that has been given read/write access, the variable gets instantiated as a SoloParamHandle. However, when the access is read-only, the variable gets instantiated as the current *value* of the SoloParamHandle. Thus the function that gets the read-only access knows what the value is, but doesn't have the SoloParamHandle itself, and therefore cannot change the SoloParamHandle.

A function that has read-write access to a SoloParamHandle can grant read/write access to further functions. A function that has read-only access can only grant read-only access to other functions.

## 7.3 Global SoloParamHandles

It is often convenient to have some SoloParamHandles that *every* function within an object class has access to. For example, in a behavioral protocol, the number of trials that

have been completed, or the history of hits/errors might be convenient to have available everywhere. But just like with regular access, while you may want every function to be able to read these, you rarely want every function to be able to change their value. For this reason, globals are usually declared as read-only. (You can also declare read/write globals, but this is not recommended.) The call is

```
DeclareGlobals(obj, 'ro_args', {'sph1', 'sph2', …});
```

where the ellipsis above means that you can put in as many SoloParamHandles in the list as you like.

If a global is read-only for everybody, how can it ever change its value? The answer is that it is strictly read-only for every function *except* those that are *explicitly* given read-write permission to it. Thus, if for object class `@myobj` you have the methods `foo.m` and `bar.m` and `zwat.m`, and `foo.m` reads like this:

```
function [] = foo(obj)

   GetSoloFunctionArgs;

   SoloParamHandle(obj, 'mine', 'value', 10);
   DeclareGlobals(obj, 'ro_args', 'mine');
   SoloFunctionAddVars('bar', 'rw_args', 'mine');
```

Then `foo.m` will have read-write access to `mine` (because all SoloParamHandles are local, persistent, and read/write for the function that declared them), and `bar.m` will have read-write access to `mine`, because it was explicitly given that access. But `zwat.m`, and any other methods of `@myobj`, will have read-only access.

A typical use of globals in our behavior protocols might, for example, involve a function that runs immediately after the end of each trial and sets the values of some commonly useful globals – say, `hit_history, n_done_trials, last_trial_events, hit_or_miss,` etc. – that every other function that will subsequently run has read-only access to.

## *7.4 Shit, I can't find my SoloParamHandle! What can I do?*

You're debugging your program, and you want access to a SoloParamHandle from the command line. But the command line hasn't been granted access! How can you even see whether the SoloParamHandle exists? There's a cheat command that can get you any SoloParamHandle. The command

```
>> sp = get_sphandle('name', 'this');
```

will return in `sp` a cell vector containing all the SoloParamHandles whose name contains the string `'this'` somewhere within it. (For aficionados, its `regexp` matching.) You will

then have full read/write access to the SoloParamHandles thus returned. Don't abuse this power! It's for debugging only.

Want only the SoloParamHandles belonging to a particular object class? Try

```
>> sp = get_sphandle('owner', '@myclassname');
```

You can use both name and owner specifications at the same time. If after this call `sp{1}` is now a SoloParamHandle, and you want to know which method it was declared in, try:

```
>> get_fullname(sp{1})
```

# 8 Saving and loading SoloParamHandles

When called from within an object's method,

```
>> save_soloparamvalues(animalname)
```

Will save the values of all the SoloParamHandles belonging to that object class in a file. It will suggest a name for the file that combines the string `animalname`, the object class name, and the date, and will prompt the user for confirmation.

```
>> load_soloparamvalues(animalname)
```

does the obvious converse.

```
>> save_solouiparamvalues(animalname)
```

saves only the values of the GUI SoloParamHandles. This is useful if you want to save the GUI settings that you've been using.

```
>> load_solouiparamvalues(animalname)
```

once again, does the obvious converse.

# 9 Autosetting GUI values

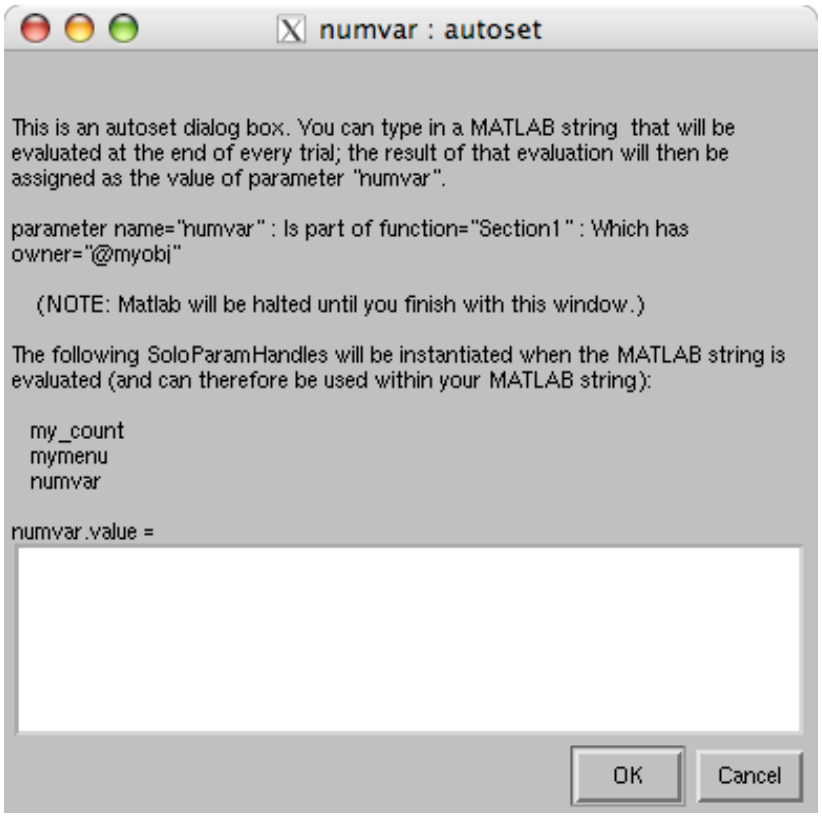
Ok, we'll end up with one of the niftiest features of GUI SoloParamHandles. This is their capacity to automatically set themselves to new values.

In a typical behavior protocol, this might be step (3) in the following sequence: (1) go through a trial (under control of the Real-Time State Machine); (2) at the end of the trial, compute some basic statistics from the trial (e.g., update the number of completed trials, set a variable to indicate whether the trial was correct or error, etc.); (3) Update automatically set SoloParamHandles, based on the data from the last trial; (4) Compute any further statistics that we want, update graphs in the graphical interface, etc; (5)

Decide what we want to do in the next trial, send the appropriate State Matrix to the Real-Time State Machine, and loop back to step (1).

Let's suppose we're using the `@myobj/` example from above. Remember that we had a NumeditParam, `numvar`, in our window, and we weren't really doing anything with it yet. Let's use it now.

Right-click on the GUI (Apple-click on Macs). You should see a window that looks like this:




Here you can type any arbitrary Matlab string. (Note: Matlab will halt until you're done typing. So no new trial will start until you click "OK" or "Cancel.") The string you type in will be evaluated, and the resulting value assigned to `numvar`, every time you call the function

```
ComputeAutoSet;
```

This function goes through all the SoloParamHandles that have non-empty autoset strings associated with them, evaluates them, and does the value assignments. Typically you will run this function once per trial.

For example, as the string you put in, type `my_count.^2` and then click through the OKs.

You will notice that the GUI for `numvar` now takes on a greenish background, the text goes italic, and it no longer allows you to change its value manually: this is to indicate that it is under autoset control.


Now make some changes to the menu, and afterwards call `ComputeAutoSet` from the command line:

```
>> ComputeAutoSet;
```

The Matlab string can be anything that Matlab understands. It will be evaluated in the same context of existing SoloParamHandle variables as that found in the function that declared `numvar`: in the above example, function Section1 has read/write access to SoloParamHandles `mymenu`, `my_count`, and `numvar`, so therefore these are the ones that are visible SoloParamHandles when you evaluate the string. The autoset dialog window lists them so you can know what's available.

What if you want a complicated computation that can't be easily expressed as a single evaluatable expression? That's fine too, as long as you set the value of your SoloParamHandle within it. For example, for numvar we could have typed:

```
if strcmp(value(mymenu), 'oogle'), adder = 3;
else adder = 1;
end;

numvar.value = numvar + adder;
```

If you declare a certain SoloParamHandle global (as usual, preferably as a read-only global), then it will be available for use in autoset strings for all the GUI SoloParamHandles. Typical variables and their names that might be used this way would be things like `n_done_trials`, `hit_history`, etc. Thus performance-based adjustments are easy to program in.

Be careful when using autoset strings! They're powerful, and precisely because of that, can be dangerous! Some protections are built in: If you type something in that Matlab can't evaluate (e.g., because of a typo in the string), the system won't crash, it'll continue working fine, but your SoloParamHandle's value won't be updated, either. A warning will be printed out and the value will stay as is. A more subtle error might be one where the string evaluates without obvious error, but it evaluates to a value you didn't want! Matlab has no way to catch that.