Coding Manual

Note: This document is still missing very important information. I'm including just so you know it will exist in the near future.

If you would like to develop plugins for rave and need more information, contact me and we can set up a phone call to discuss it.

Reminder: If you installed RAVE to MATLAB's toolbox directory, any new files you put in the RAVE directory won't be recognized by MATLAB until you run the command "rehash toolboxcache". Also, if you create new directory, don't forget to add it to your path.

Licensing your plug-ins

If you make a new plug-in for RAVE, we could certainly appreciate it if you give it a GPLv2-compatible license and distribute it freely. However, as long as you don't distribute it with rave (i.e. in the same zip), then it is probably OK to give it any license you want. Note that anything you distribute WITH RAVE must be GPL compatible. (Also note that I say "probably ok" because GPL's language isn't very clear on this issue. When in doubt, consult an expert.)

If you make a new plug-in (and give it a GPL-compatible license) that you would like to be included in the rave distribution that is downloadable from our (currently nonexistent) website, let me know and I will be happy to include it.

Things you should know

Before attempting to code plugins for rave, you should understand the difference between numerical (double) arrays and cell arrays, and how to interact with each. (i.e. when to user parentheses vs. braces) You should also understand what structures (the variable class) are.

You should be familiar with MATLAB's handle graphics system. In particular, you should understand what a "handle" is, how to use handles to set/get properties, how to determine an object's handle using findobj.

You should also understand MATLAB's callback function format. Almost every function in rave uses this format, which takes two inputs – the handle of some object, and a second input that is almost always just an empty matrix.

Lastly, you should understand how the guidata function works, which is reviewed briefly here:

Almost all data tracked by rave that is NOT particular to one graph is stored in a variable named "handles". Unfortunately this is the same name as the concept of the handle of a graphics object, but it is also the default name MATLAB uses for this variable and I didn't change it. (For consistency, you shouldn't either). handles is a structure, which you can think of as a folder that holds other variables

(many of which are themselves also structures). The handles structure is special in that it is stored and retrieved using the "guidata" function, which allows it to easily be shared by all functions in rave. (A better way to say this is that there is a single special variable that can be stored/retrieved using the guidata function, and in this case that variable is a structure which we will always name "handles".)

There is always only ONE guidata variable that is shared by EVERY object/callback of a single figure (i.e. window) This variable is retrieved using the syntax handles=guidata(src); where "src" is the handle of any object in the figure. Regardless of what the "src" handle is, you will always get the same handles structure as you'd get using any other "src" in the same figure. IMPORTANTLY, this means that handles is only shared among callbacks that originate from the main rave window. Callbacks from popup windows will have their own guidata, and must access the main handles structure using another method (described elsewhere).

Any time you make changes to the handles structure, you must also store the updated results, also done using the guidata function, but with the syntax guidata(src,handles); (note in either case, src is always the first input to guidata.) Again, it does not matter what "src" is – this function call sets the guidata for EVERY object in the figure.

What are the implications of all this? Simply that most functions in rave do not have inputs and outputs in the traditional sense. Instead, each function takes a handle (src) as an input, then uses handles=guidata(src) to retrieve ALL your data. Then the function modifies handles as necessary, and stores the results using guidata(src,handles).

Consequently, handles=guidata(src) is the first line of almost every function in rave. And guidata(src,handles) is the last line of almost every function (the exception being those functions that don't actually change handles.

Review of Structures and Cell Arrays

In MATLAB, a structure contains "fields," each of which can be thought of as a new variable. Fields are accessed by typing the name of the structure, a period, then the name of the field. For example, the statement handles.index=5 creates a field named "index" in the structure "handles" and gives this field the value 5. Since 5 is a number, handles.index is effectively a double precision array with dimension 1x1. Depending on what is to the right of the equals sign, handles.index could also have been made into a string, cell array, or other class of numeric array. If you want handles.index to itself be a structure, just append another period and field name. For example, handles.index.data=10.

Now, if handles.index is an array, you can directly reference into it using parentheses. For example, handles.index=[2 4 6 8], then handles.index(3) will return "6". Similarly, if handles.index is a cell array, you can reference into it using braces.

As with any cell array in MATLAB, a structure field that is a cell array can contain other (nested) cell arrays. You can index into nested cell arrays by simply appending more and more curly braces, and (if necessary) a final reference into a numerical array in parentheses. (You can ALWAYS only have 1 parenthetical reference and it must be the last reference).

For example, the line mycolumn=handles.alldata{1}{3}(:,5); Tells you that handles.alldata is a cell array that contains at least 1 entry, which must also be a cell array (since the next reference is also to a cell) that in turn contains at least 3 elements. The 3rd element of THAT cell array must be a numerical matrix, since the next reference is in parentheses (instead of braces) , and it must contain at least 5 columns, since the index references the 5th column. Thus, mycolumn is the 5th column of the 3rd cell of the 1st cell of handles.alldata.

If all of that made sense to you, you'll do just fine. If instead you said "huh?" you should probably read the MATLAB help files for structures and cell arrays and practice using them before you try coding for rave. Note there are also more complex references possible where the fields themselves have dimensions, things like handles.alldata(5).names{3}, but rave (I think) never uses these.

The *handles* structure

Ok, so what does handles actually contain? While it would be easy to store EVERYTHING in handles, it would quickly become gigantic and difficult to use. One of the guiding principles you should follow when using rave is **only store information in handles if there is no other place you can put it.** At first this might sound confusing, since we already established that handles is the only variable your functions can access. But there are a few workarounds to let you store data elsewhere.

First off, suppose you have a red line, and you have a function that needs to know what color the line is. You could store this in handles, but don't. It is already "stored" in the line itself. You function should ask the line what color it is (using something like get(linehandle,'color')). Additionally, EVERY graphical object has two properties that can be used to store additional data. The first is the "tag" which MUST be a single string. Usually this is used to "name" the object with a unique identifier that can be used later by findobj to retrieve that object's handle (which can in turn be used with get to retrieve other properties of the object). The tag can also sometimes be useful for storing other text data that doesn't act as a name. The second (more useful for the purpose of storing data) property is called "userdata". It can be used to store ANYTHING.

In particular, each graph you create has its own data structure, always named "udata" (short for userdata) that is stored in the userdata property of one object that composes the graph (this is described in much more detail below). The important thing to note here is that you have two main options for storing information: if the information you need to store is particular to one graph, you should store that information in the graph's udata. If the information is "high-level" enough to be shared by several graphs, store it in handles.

Side note, optimizers have their own data structure named "settings" described later.

But PLEASE think very hard if there is any way to avoid making a new field under the main handles structure. I think it is in everyone's best interest if we keep the list of handles fields to just those listed below. Anything stored directly under handles should be VERY high level and useful to most if not all graphs. If you have information that needs to be stored temporarily, for example while an optimizer is running, store it under handles.temp.

Again, PLEASE try not to add new fields to handles, as this will break your users older saved .rve files, and NEVER modify the behavior of the existing fields, as this will certainly break many other features.

The list below summarizes the most commonly used fields in handles (eventually I'll expand to include all of them). The index {d} represents the number of a data set, and the index {a} represents the number of an analysis. If a field is referenced with {d}{a} then its contents differ for each data set AND for each analysis. If a field is referenced with just {d} then it differs between each dataset, but is the same for each analysis.

Fields related to data sets and variables

handles.datasetnames – the names of all data sets.

handles.datasetnames{d} is a string containing the name of the d-th data set. The first data set that is loaded has d=1, the second data set loaded has d=2, etc.

handles.analysisname – the names of all analyses.

handles.analysisnames{a} – the name of the a'th analysis.

Handles.alldata – all of the data is contained in here.

Handles.alldata{d} – an m*n matrix containing all data of the d'th data set. This data set has n variables and m data points for each variable.

Handles.alldata{d}(:,c) – is a m*1 vector containing all data values for the c'th variable in the d'th data set.

handles.allnames – the names of the variables

Handles.allnames{d}{c} – a string containing the name of the c'th variable in the d'th data set.

handles.targetvalues – the "ideal" values of the variables. –inf indiciates desire to minimize a variable, inf indicates desire to maximize a variable

handles.targetvalues{d}(c) = the target value of the c-th variable in the d-th data set.

handles.preferences – the weight each variable has when calculating certain types of objective functions

handles.preferences{d}(c)= the weight of the c-th variable in the d-th data set.

handles.isfunction – a logical vector indicating whether variables are calculated by a user-supplied function. If "false" then the variable is "independent".

handles.isfunction{d}(c)= true if the c-th variable of d-th dataset is calculated by a user-supplied function. Otherwise false.

handles.funnames – names of user-supplied functions

handles.funnames{d}{c}= the name of the function that calculates the c-th variable of the d-th dataset. Or an empty string if c-th variable is independent.

handles.funvars – input variables on which calculated variables depend

> handles.funvars{d}{c} – vector of the variables that are inputs to the function that calculates the c-th variable. IMPORTANT: this list can only contain values from 1 to c. In other words, only feedforward function dependencies are allowed.

handles.nargout{d}{c}=

handles.datatype{d}{c}=

Fields related to functions used to generate data

Fields related to analyses

handles.discretevalue – the list of "feasible" values that discrete variables in a data set can take

handles.discretevalues{d}{a}{c}= a vector of allowable values for the c-th variable in the d-th dataset in the a-th analysis.

handles.xdials – the "current value" of each variable

handles.xdials{d}{a}(c)=

handles.minlimit{d}{a}(c)=

handles.maxlimit{d}{a}(c)=

handles.isselected{d}{a}(r)=

handles.rowcolors{d}{a}(r)=

handles.isvisible{d}{a}(r)=

Fields related to constraints

handles.constrainthandles{d}{x}=

handles.constrainttext{d}{x}=

handles.constrainttype{d}{x}=

handles.constraintvalues{d}{x}=

handles.constraintadders{d}{x}=

handles.constraintscalers{d}{x}=

handles.constraintvars{d}{x}=

handles.constraintcolors{d}{x}=

Fields related to graphs

handles.activeindex=

handles.allblockers --

handles.allgraphs(ai)=

handles.fakeax(ai)=

handles.fakerax(ai)=

handles.currentgraph=

handles.activegraph=

handles.activetype=

handles.activetab=

handles.activesubgraph=

handles.subax=

handles.subfakeax=

handles.subfakerax=

handles.statsax=

handles.grabbers

handles.graphnames=

Fields related to plugins

handles.graphinfo.<graphname>=

handles.optimizerinfo.<optimizername>=

Fields related to rave interface

handles.wspanel – The handle of the uipanel that forms the workspace. All graphs and other objects in the workspace are children of this.

handles.mainaxes – The handle of the axes that shows the background of the workspace. The gridlines are children of this.

handles.infobar – The handle of the information bar at the top of the screen. Change what it says using set(handles.infobar,'string','what you want it to say'). Clear it using set(handles.infobar,'string','').

handles.prefs – The preference structure. This is an exact copy of the raveprefs variable that you can access using "load raveprefs".

handles.colors – The color preferences structure. This is an exact copy of raveprefs.colors.

handles.explorer – The axes that form the "navigator" at the top left of the screen.

handles.expblockers – The handles of the rectangles that represent each graph in the navigator.

handles.fig – The handle of the rave window itself. I often use this as the input to guidata, because you're guaranteed it is always a valid handle.

handles.tabobjects – A vector of handles for EVERYTHING that is currently displayed in the tabs, so that the function delete(handles.tabobjects) will clear the tabs. The last line of any "tab" function (e.g. formattabline.m) should ALWAYS be handles.tabobjects=[the handles of everything you created in the tabs];

Adding New Graph Types

The easiest way to create a new graph is to find an existing similar graph, copy its code, and change as little as necessary to generate the graph you want. In particular, try to find an existing graph with the same axes layout, since that can be difficult to code.

Before you start coding, try to plan how your graph will interact with rave. What options need to be available to the user on the view and format tabs? How will the graph react to data being selected, recolored, or hidden? If the graph needs to generate new data (i.e. it is a "continuous" graph) how can you minimize the computational expense?

The ravecreategraph function

Note: Throughout this discussion I use "axes" as a singular noun meaning "the thing that gets created by MATLAB when you enter the command axes()".

At a minimum, each graph must include 3 axes. Even though you only see one axes, there are actually 3 of them stacked on top of each other. Using multiple axes stacked on top of each other allows more formatting options, such as different colored gridlines. It also givs each graph 3 userdata properties, each of which is used by rave for a particular purpose. The 3 axes are:

handles.allgraphs(ai) = this is the main axes… the one in which you actually plot the data. The userdata of this axes

Handles.fakeax(ai) = this is the next axes down. This axes doesn't really do anything, but you can use it if you need to plot something at an intermediate layer. Important: The udata for your graph MUST be stored in this axes's userdata property, i.e. set(handles.fakeax(ai),'userdata',udata) and udata=get(handles.fakeax(ai),'userdata',udata).

handles.fakerax(ai) = the bottom axes. This axes is used to display the graph's background color and gridlines.

These three axes are created by the function ravenewgraph, which is ALWAYS run when the user makes a new graph and before your graph's creation function is run. Thus these axes will already exist when your function gets run, so you don't need to create these. You can access them using the fields indicated above.

IMPORTANT: Any plotting commands (e.g. plot,line,scatter,patch,surf,text) MUST take one of the above axes as a parent (unless you are making a matrix graph, described below). ALSO – BEWARE: many "high level" plotting commands will reset the userdata property of the axes. So you will need to set it again AFTER you finish plotting. For example, handles.allgraphs(ai)'s userdata should contain the keyword for your graph, something like "scatter", but after you plot into the axes, it may get reset to an empty matrix. So check at the end of your graph creation function that it still says "scatter". Otherwise rave won't know which tabs to load when the user selects your graph (among other problems).

If you are making a matrix graph with multiple axes (I use "matrix graph" to refer to any graph that comprises multiple visible axes, regardless of whether they are actually layed out in a matrix), the 3 axes listed above will still be created. DON'T delete them, just make them invisible in your graph creation function, then create your own matrix of axes, EACH OF WHICH SHOULD ALSO STILL BE STACKED 3 high. Note: I recommend you use the position of handles.allgraphs(ai) to determine the boundary within which your graph matrix will be placed. See ravecreatescattermatrix.m for an example of how to do this.

Note: The "ai" (handles.activeindex) associated with each graph does not change when you delete a graph. If you have 3 graphs and you delete the second one, then you just have no graph with ai=2.

Adding New Optimizers

Asdfasdfasdf

Adding New Explore Tab Methods

Asfasdfasdf

Adding New Metamodels

Asdfasdf

Asdfasdf

Evaluating user-supplied functions to generate data

Updating all linked graphs in an analysis

Updating all graphs in a data set