autocli.help - Function

Package	**display**
Module	**widgets**
Tool	**autocli**

Show help

Synopsis
help()

Description
This function shows the help attached to the methods and arguments.

---

autocli.get - Function

Package	**display**
Module	**widgets**
Tool	**autocli**

Get a record of the parameters

Synopsis
get()

Description
This returns a record containing the current parameters. WYSIWYG applies. The corresponding event is setoptions.

1.4.3 autogui - Tool

Package	**display**
Module	**widgets**

Tool to build and operate a graphical form interface *include "autogui.g"*

Description

This `tool` *is an extremely useful one which builds a "form-like" graphical user interface (GUI) from a simple record whose fields describe individual parameters which the user may be allowed to edit. The GUI is displayed with the various parameters arranged vertically. Some of the parameters may be stored in rollup widgets. Each parameter is displayed with a text label, a widget (from the widgets module) to display the current value and optionally allow the user to edit it, a "spanner" menu (see guientry), and optionally an "auto-apply" check-button. Generally, an autogui* `tool` *will be used to show the state of some "parent"* `tool` *to the user, and/or allow them to edit the state. Consequently, the autogui* `tool` *is a* subsequence, *allowing the* `tool` *to emit events, and catch inbound events. These events can be used to ensure that the GUI and any parent* `tool` *remain synchronised as the user modifies widgets in the GUI. Furthermore, if the autogui is used in this way, multiple (duplicate) autogui* `tools` *can be used to control one and the same underlying parent. This is done, for example, in the viewerdisplaydata* `tool`. *An obvious use of this capability would be to display identical autogui* `tools` *on separate desktops (or even screens!) for a given* `tool`. *In addition to the event interface, the autogui* `tool` *provides a small number of functions to control its behaviour.*

Parameter description

Most transactions of the autogui `tool` *involve a "parameter record." This is a* Glish *record containing one or more top-level fields. In* full *form, each top-level field contains a* Glish *record which completely describes a particular parameter in terms of its name, parameter type, default value, current value, its possible values (for 'choice' or 'range' parameters), whether it can be 'unset', whether it can be 'auto-apply' and whether it is editable by the user. In* short *form, each top-level field simply contains the value for its parameter.*
The standard fields in a (full) single parameter record are:

- `dlformat:` *a string giving the name of the field of the parameter record in which this particular parameter is found.*
- `listname:` *a string giving the name of this parameter which should be displayed in the GUI to identify it.*
- `ptype:` *a string giving the type of the widget to be used for this parameter. If this field is not included, no widget will be created. Leaving this field blank will create a default entry box.*
- `value:` *the present value of this parameter.*
- `default:` *the default value of this parameter.*

The optional fields in a (full) single parameter record are:

- `context`: *if present, this parameter will be placed in a rollup widget having as its name the string contained in this field of the record.*
- `autoapply`: *if the autogui* `tool` *was constructed with* `autoapply=T` *[default], then a check-button will be placed next to this parameter. When this button is checked, any modification to this parameter will be immediately emitted from the autogui* `tool` *in a* `setoptions` *event (see below). This optional field, if present, indicates the initial state of this auto-apply button.*
- `dir`: *if present, and is* `'out'`, *then this parameter is considered to be output-only, so the user will be unable to edit it.*
- `allowunset`: *if present, and is* `T`, *then the constructed widget will allow this parameter to be 'unset'.*
- `dependency_group`: *if present, this specifies a tag which identifies this as a parameter which has a dependency on other parameters with the same tag. At present, all this means is that parameters with the same* `dependency_group` *tag must be mutually exclusive.*

We now list the available parameter types (`ptype`*), and their additional required record fields:*

- `boolean`: *a boolean guientry widget is provided with two choices:* `T` *or* `F`. *No additional required fields.*
- `choice`: *a choice guientry widget is provided for selection from a set list of items. Additional required record fields are:*
  - `popt`: *a vector of strings which are the valid options.*
- `directory`: *a file guientry widget is provided for selecting a directory. No additional required fields.*
- `file`: *a file guientry widget is provided for selecting a file. No additional required fields.*
- `floatrange`: *a range guientry widget is provided for selecting a single float or double from a range. Additional required record fields are:*
  - `pmin`: *minimum allowed real value*
  - `pmax`: *maximum allowed real value*
  - `presolution`: *resolution of provided scale widget*

  Additional optional record fields are:

- `provideentry:` *if present, and is* `T`*, then an entry widget will be provided alongside the scale widget to allow the user to type a specific value. In this case, the scale will be "extendable" so that values outside the initial acceptable range will cause the scale to resize itself, and the allowed minimum and maximum to expand to accomodate the new specific value.*

- `intrange:` *a range guientry widget widget is provided for selecting a single integer from a given range. Additional required record fields are:*
  - `pmin:` *minimum allowed integer value*
  - `pmax:` *maximum allowed integer value*

  Additional optional record fields are:
  - `provideentry:` *if present, and is* `T`*, then an entry widget will be provided alongside the scale widget to allow the user to type a specific value. In this case, the scale will be "extendable" so that values outside the initial acceptable range will cause the scale to resize itself, and the allowed minimum and maximum to expand to accomodate the new specific value.*
- `measure:` *a measure widget is provided for selecting an AIPS++ measure. No additional required options.*
- `quantity:` *a quantity guientry widget widget is provided for selecting an AIPS++ quantity. No additional required fields.*
- `region:` *a region guientry widget is provided for selection an AIPS++ region. No additional required fields.*
- `scalar:` *a scalar guientry widget is provided for entering a scalar (integer, double, or complex). No additional required fields.*
- `table:` *a file guientry widget is provided for selecting an AIPS++ table. No additional required fields.*
- `userchoice:` *an extendoptionmenu is used for selection from a user-extendable list of items. Additional required record fields are:*
  - `popt:` *a vector of strings which are the initial valid options.*
- `minmaxhist:` *a minmaxhist guientry widget is provided for selection of minimum and maximum in a graphical way. Additional required record fields are:*
  - `minvalue` *The lower limit of the range.*
  - `maxvalue` *The upper limit of the range.*

  Additional optional record fields are:

- default *The default range.*
- histarray *The array containing a histogram to draw.*
- imageunits *The brightness units used as labels.*

Event description

Two important events are emitted by the autogui tool*, to which well-written* Glish *code will respond! The* setoptions *event is emitted whenever an "auto-apply" parameter is modified in the GUI. Its* $value *is a short-form parameter record, where the top-level fields simply contain the values of each parameter. It is also emitted when the action button is pressed, if the autogui* tool *was constructed without a parent frame (*parent=F*), and the* actionlabel *argument was a valid string.*
The newuserchoicelist *event is emitted whenever the extendoptionmenu widget for a parameter of* ptype='userchoice' *was extended by the user. This simply allows the programmer to detect when this happens, and modify their own local copy of the parameter set accordingly. This might be useful, for example, when they will create another autogui* tool *later on, and would like it to contain the additions the user made to any extendoptionmenus. The* $value *of this event has two fields:*

- param: *the internal name of the parameter which whose option list was extended (ie.* dlformat*)*
- newvalue: *the new value added to the choice list*

Example

In the example below, we build a simple autogui tool *with three parameters.*

```
include 'autogui.g'

# we will put a "parameter set" into "parameters":
parameters := [=];

# an example of the floatrange parameter type.  Its auto-apply
# button will be unchecked, and this parameter cannot be 'unset':
parameters.power := [=];
parameters.power.dlformat    := 'power';
parameters.power.listname    := 'Scaling power';
parameters.power.ptype       := 'floatrange';
parameters.power.pmin        := -5.0;
parameters.power.pmax        := +5.0;
parameters.power.presolution := 0.1;
parameters.power.default     := 0.0;
parameters.power.value       := 1.5;
```

```
parameters.power.autoapply   := F;
parameters.power.allowunset  := F;

# an example of the userchoice parameter type.  Its auto-apply
# button will be checked, and it also cannot be 'unset':
parameters.color := [=];
parameters.color.dlformat   := 'color';
parameters.color.listname   := 'Line color';
parameters.color.ptype      := 'userchoice';
parameters.color.popt       := "black white red yellow green";
parameters.color.default    := 'black';
parameters.color.value      := 'black';
parameters.color.autoapply  := T;
parameters.color.allowunset := F;

# an example of the region parameter type.  Its auto-apply
# button will be checked, and it CAN be 'unset':
parameters.region := [=];
parameters.region.dlformat   := 'region';
parameters.region.listname   := 'Displayed region';
parameters.region.ptype      := 'region';
parameters.region.default    := unset;
parameters.region.value      := unset;
parameters.region.autoapply  := T;
parameters.region.allowunset := T;

# construct an autogui.  We will put auto-apply=T since some of
# our parameters are auto-apply, and ask for a button with label
# 'Go!'.

myautogui := autogui(parameters, 'Demo autogui', actionlabel='Go!',
                     autoapply=T);

# and respond to the events it emits:
whenever myautogui->setoptions do {
  # do something more interesting than this though!
  print "New options for ", field_names($value), "emitted ...";
}
```

If you run this code, you should see a GUI like that shown in 1.10. Modifying either the 'Displayed region' or 'Line color' widgets should result in information being printed at the command line by the `whenever` *statement at the end of the example. Likewise, pressing the 'Go!' button will emit* all *parameters.*

autogui - Constructor

Package	**display**
Module	**widgets**
Tool	**autogui**

Create an autogui `tool`

Synopsis

autogui(params, title, toplevel, map, actionlabel, autoapply, expand, widgetset)

Arguments

params	*Record to define the autogui* *Allowed: Record*
title	*title string if top-level frame constructed internally* *Allowed: String* *Default: 'autogui'*
toplevel	*top-level parent frame* *Allowed: frame or boolean* *Default: F*
map	*map the GUI to the screen immediately?* *Allowed: boolean* *Default: T*
actionlabel	*label for action button* *Allowed: string or boolean* *Default: F*
autoapply	*Show auto-apply buttons* *Allowed: boolean* *Default: T*
expand	*Expand policy* *Allowed: 'x'—'y'—'both'—'none'* *Default: 'none'*
widgetset	*widgetserver* `tool` *to use to build the GUI* *Allowed: widgetserver* `tool` *Default: dws*

Returns

subsequence

Description

This is the only constructor for the autogui `tool`*. The first argument,* `params`*, contains the various parameters which will be used to generate the GUI. If a parent frame is not given in the* `toplevel` *argument, a top-level frame will be created with title as given in the* `title` *argument. If the* `actionlabel` *argument is given as a string, then a button will be placed at the bottom of the GUI with that string as its label. Pressing this button will cause the autogui* `tool` *to emit all parameters and their values via a* `setoptions` *event. If the* `map` *argument is* `F`*, then the GUI will be constructed but not shown to the user immediately, but can be shown via a call to autogui.map. The same effect can be controlled more carefully if a presently unmapped parent frame is given to* `toplevel`*. The final two arguments just provide some future flexibility with respect to widgetserver and guientry* `tools`*.*

---

autogui.buttonbar - Function

Package	**display**
Module	**widgets**
Tool	**autogui**

Return the frame in which buttons are placed

Synopsis
buttonbar()

Returns
frame

Description

This function returns a handle to the frame in which the autogui `tool` *places any buttons, such as the action button and the dismiss button. This is provided so that programmers can add their own buttons in a simple way. A more flexible way to do this is to provide a parent frame to the autogui* `tool` *constructor, and build your own GUI in the same frame.* Note that this function is only available if a top-level frame was *not* given during construction.

Example

Assuming you have already created an autogui `tool` *in the variable* **myautogui***, perhaps as per the above example, you may do something like this to place a "Free Beer!" button near the "Go!" button:*

```
# ...
myf := myautogui.buttonbar();
myb := dws.button(myf, 'Free Beer!', type='action');
whenever myb->press do {
  # ...
}
# ...
```

---

autogui.get - Function

Package	**display**
Module	**widgets**
Tool	**autogui**

Read the GUI and return the current settings

Synopsis
get()

Returns
short-form parameter record

Description

This function reads the GUI, extracting all *parameter values, and returns the result in a* short-form *parameter record. In this form, the top-level fields simply contain the values corresponding to the parameters having the top-level field names as their* `dlformat` *parameter bit. When this function is called, a* `setoptions` *event is also emitted. This assists in keeping multiple clients aware of the state of the autogui* `tool`.

Example

Assuming you have already created an autogui `tool` *in the variable* **myautogui**, *perhaps as per the above example, you may do something like this to read the current values into a record:*

```
# ...
currvalues := myautogui.get();
# ...
```

---

autogui.map - Function

Package	**display**
Module	**widgets**
Tool	**autogui**

Map the GUI to the screen

Synopsis
map()

Returns
T

Description

This function is only useful if at construction, the `map` *argument was forced to be* `F`*. In this case, you can use this function to map the GUI to the screen at some later stage.*

---

autogui.done - Function

Package	**display**
Module	**widgets**
Tool	**autogui**

Finish using this autogui `tool`

Synopsis
done()

Returns
T

Description

This function should be used when you have finished using a particular autogui `tool`*. It unmaps the GUI from the screen, stops it emitting and responding to events, and releases any resources being used by the autogui.*

Example

Assuming you have already created an autogui `tool` *in the variable* **myautogui**, *perhaps as per the above example, you may finish using it by doing:*

```
# ...
myautogui.done();
# ...
```

---

autogui.fillgui - Function

Package	**display**
Module	**widgets**
Tool	**autogui**

Fill the autogui `tool` *with updated values*

Synopsis
fillgui(wparams, what)

Arguments

wparams	*new parameter values* *Allowed: full-form parameter record*
what	*which sub-field to extract new values from* *Allowed: 'value' or 'default'* *Default: 'value'*

Returns
T

Description

This function is mostly used internally to construct and fill the GUI. However, it can be used externally to refresh the GUI to a new set of parameters.

Example

For example, suppose you want to add a "Restore defaults" button to your autogui. When the user presses it, then the autogui `tool` *will be told to re-fill the GUI with* `what='default'`.

```
# ... myautogui setup already ...
myf := myautogui.buttonbar();
myb := dws.button(myf, 'Restore defaults');
whenever myb->press do {
  myautogui.fillgui(parameters, 'default');
}
# ...
```

Of course, we should also update the `value` *fields of the parameters to be the same as the* `default` *fields, but that is left to the reader...*

---

autogui.modifygui - Function

Package	**display**
Module	**widgets**
Tool	**autogui**

Modify the GUI parameter set

Synopsis
modifygui(wparams)

Arguments

wparams	*new parameters* *Allowed: full-form parameter record*

Returns
T

Description

This function is not completely implemented yet. *It is provided so that an existing autogui* `tool` *can be modified in terms of the names of its parameters, and the parameter types etc. Presently, only fields of* `ptype` *'intrange', 'floatrange' and 'minmaxhist' are noticed by this function. Their labels are updated according to the* `listname` *subfield, their ranges according to the* `pmin` *and* `pmax` *subfields, and in the case of 'minmaxhist', the data being displayed can be changed via the* `histarray` *subfield.*

1.4.4 autoguitest - Function