Orbital Data Collection Software

Reference Manual

Contents

Getting Started

Requirements

- Python 3.7+ (32-bit)
- Windows 10

Overview and Basic Script Format

Orbital is simple, user-friendly application for running Python scripts for real-time data collection. The intent of this application is to simplify user scripts as much as possible while providing many useful features for monitoring collected data and instrument feedback. It is essentially nothing more than a graphical interface for Python scripts with built-in support for real-time data collection.

The application requires a basic format to properly interface with user scripts. Shown below is a skeleton for creating new user scripts. It contains the functions necessary for interfacing with Orbital and can be considered as the basis for every user script. Shown on the right is the default script (ScriptSkeleton.py).

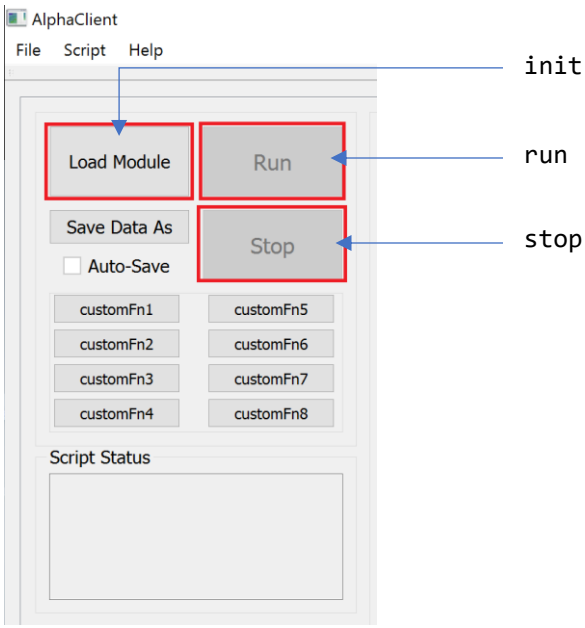
```
# Script Layout
import orbital

def init():
    # initialization

def run(params):
    # primary run function

def stop():
    # set stop flag(s)
```

```
# ScriptSkeleton.py
import orbital

stop_execution = False

plots = [[0, 1, 0, 1]];
input_params = [
    "PARAM1 [Param 1]",
    "PARAM2 [Param 2]",
    "PARAM3"]

def init():
    orbital.init(plots, input_params)

def run(params):
    global stop_execution
    stop_execution = False

def stop():
    global stop_execution
    stop_execution = True
```

Orbital has three basic controls (Load, Run, and Stop) that map to the three functions in the skeleton:

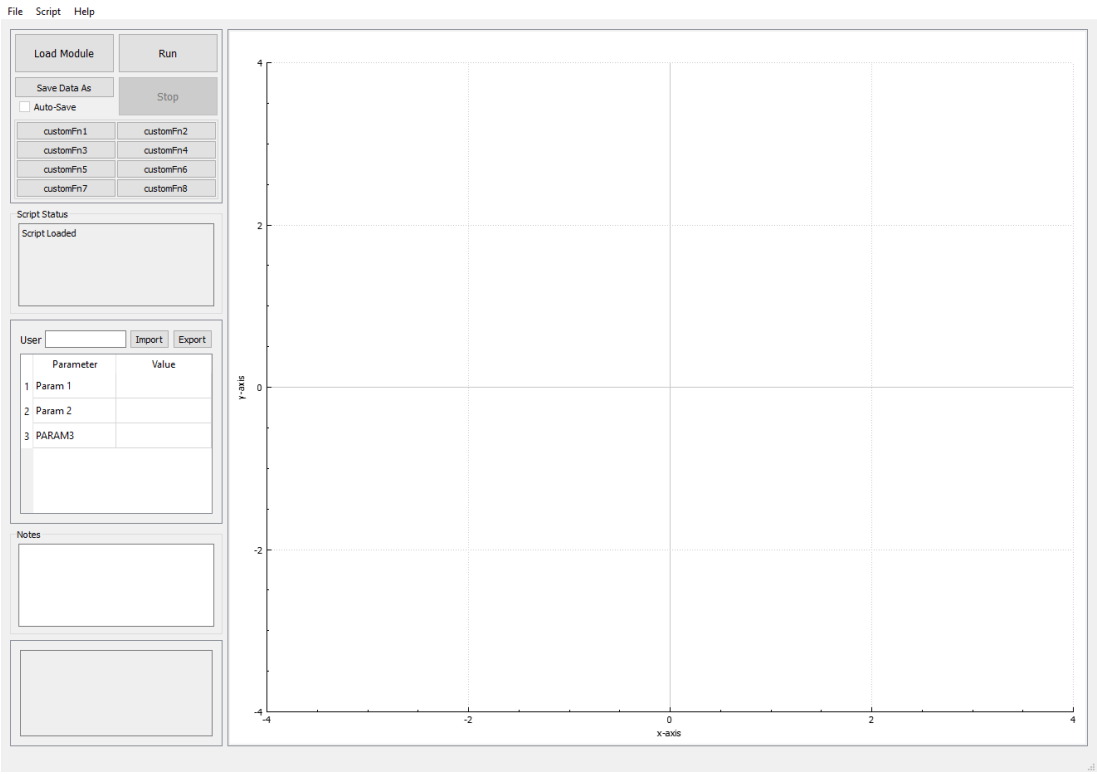
- `init` is called by the Load Module button and provides Alpha with the script setup parameters via the `orbital.init` function (more on this later).
- `run` is called by the Run button. A dictionary of the script parameters is passed to this function (labelled `params` above). Note that all keys and values in the parameter dictionary are string objects.
- `stop` is called by the Stop button and can be used to set global variables for exiting any loops in the run thread. Note that there is no way to kill the script from the Alpha interface without closing the application, so making good use of the global stop variable is highly advised.

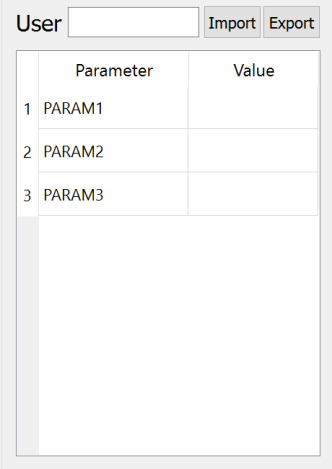
Technically, these functions can do anything the user programs them to do—they're only used as handles for the corresponding controls. However, it's advised to use them as their name suggests.



Loading a Module

The Load Module button will prompt the user with an “Open File” dialog window to navigate to and select their Python script. Once selected, the script’s `init` function will be called. For example, loading `ScriptSkeleton.py` will show this:





The script variables have now populated the parameter box in Alpha's control panel. Each entry corresponds to the parameter name in the `input_params` string list which was passed in the `Orbital.init` function along with the `plots` parameter.

The Import and Export buttons can be used to import parameter values or export the current parameter values.

The values entered here are passed to the script in a dictionary (with string key-value pairs corresponding to the parameters and values) when Run is clicked. The constructed dictionary is also shown in the error log for debugging purposes (more on this later). Note that empty values will not create a key-value item in the dictionary.

Callback Functions

These are the functions that are used to communicate back to the Alpha interface:

- `orbital.init`
- `orbital.msg`
- `orbital.sendData`
- `orbital.sendXData`
- `orbital.setDataInfo`
- `orbital.setDataFileInfo`
- `orbital.set2DPlotProperty`
- `orbital.setColormapProperty`
- `orbital.setupColormap`
- `orbital.showColormap`
- `orbital.sendCMData`
- `orbital.saveData`
- `orbital.clearData`
- `orbital.renameCustom`
- `orbital.setAutosave`
- `orbital.setPlotArrangement`

init

Sends the initialization parameters.

Function signature/overloads:

- `orbital.sendInit([n_plots, plot_arrangement], input_params)`
- `orbital.sendInit([n_plots, plot_arrangement], input_params, visa_resources)`

`n_plots`:

- Integer > 0
- Tells the application how many plots to set up

`plot_arrangement`:

- List of 4-tuples (integers): [row, row_span, col, col_span]
- Tells the application how many plots to set up and how to arrange them
- Users can copy and paste desired plot arrangements by clicking the Arrangement button in the Plot Editor

`input_params`:

- String list
- A list of the input parameters to be used when running the script
- Each string in the list will be used as a key in the run function's parameter dictionary
- An optional preferred text (surround by square brackets) can be used to display a preferred text to the user

Example:

```
input_params = [
        "PARAM1",
        "PARAM2[Parameter 2 (m/s)]",
        "PARAM3 [Parameter 3 (V)]",
        "Parameter 4 (Hz)"
]
```

will display…

	Parameter	Value
1	PARAM1	
2	Parameter 2 (m/s)	
3	Parameter 3 (V)	
4	Parameter 4 (Hz)	

and will return a dictionary with keys…

```
"PARAM1",
"PARAM2",
"PARAM3",
"Parameter 4 (Hz)"
```

`visa_resources` (requires visa module):

- String tuple
- Equivalent to the return of visa.ResourcesManager().list_resources()
- These visa resources populate the visa instrument selections in the *Script->Visa Setup* menu
- Appends the parameter dictionary using keys VISAX where X = 0,1,2,… corresponding to the visa selection in the Visa Setup menu (see Visa menu section for a more detailed explanation)

msg

Sets the script status message.

Function signature/overloads:

- `orbital.msg(msg, append=False)`

`msg`:

- String
- Message to send to the Script Status textbox

append:

- Boolean
- Set to true to append to the current script status

sendData

Primary command for plotting and/or storing data.

Function signature/overloads:

- `orbital.sendData(data_set, 0)`
- `orbital.sendData(data_set, n_data, data0, data1, …)`

`data_set`:

- Integer >= 0
- Tells the application which graph/data container to associate the data with
- data_set = 0 will store the data in the non-plotted container

`n_data`:

- Integer >= 0
- Tells the application how much data to expect
- n_data = 0 will clear the graph corresponding to the plot_id

`dataX`:

- Numerical (converts to double)
- Data to be stored/graphed
- Note that only the first two data will be plotted, but up to 32 data values can be sent

sendXData

Plots and/or stores (x,y) data in array format.

Function signature/overloads:

- `orbital.sendXData(data_set, x_data, y_data, save_data=True)`

`data_set`:

- Same as sendData

`x_data`, `y_data`:

- Numerical lists of equal size (can contain Integer and/or Float values)
- Appends these pairs of (x,y) data to the graph
- Note that if lists are of different sizes, the larger list will be truncated to match the size of the smaller list (a warning will be prompted to the user indicating that this has occurred)

`save_data`:

- Boolean
- Indicates whether this data should be appended to the internal data buffer
- Defaults to True

setDataInfo

Updates the data set information.

Function signature/overloads:

- `orbital.setDataInfo(data_set, info)`

`data_set`:

- Integer >= 0
- Data set for which the desired info should be changed

`info`:

- String
- A description of the data set

setDataFileInfo

Updates the general data information.

Function signature/overloads:

- `orbital.setDataFileInfo(info)`

`info`:

- String
- A description of the data

set2DPlotProperty

Updates the property of a 2-D plot.

Function signature/overloads:

- `orbital.set2DPlotProperty(plot_id, property, new_value)`

`plot_id`:

- Integer >= 1
- Plot to be updated

`property`:

- String
- Property to be updated
- Can be any of the following:
    - “title”
    - “x label”
    - “y label”
    - “line style”
    - “scatter style”
    - “scatter size”
    - “color”

`new_value`:

- String
- Property value
- Must match a value (given as a string) in the Graph Editor menu

setColormapProperty

Updates the property of a colormap.

Function signature/overloads:

- `orbital.setColormapProperty(plot_id, property, new_value)`

`plot_id`:

- Same as set2DPlotProperty

`property`:

- String
- Property to be updated
- Can be any of the following:
  - "title"
  - "x label"
  - "y label"
  - "z label"
  - "color min"
  - "color mid"
  - "color max"

`new_value`:

- Same as set2DPlotProperty

setupColormap

Initializes a plot as a colormap.

Function signature/overloads:

- `orbital.setupColormap(plot_id, x_min, x_max, y_min, y_max, x_size, y_size, show=True)`
- `orbital.setupColormap(plot_id, x_min, x_max, y_min, y_max, x_size, y_size, z_min, z_max, show=True)`

`plot_id`:

- Same as set2DPlotProperty

`x_min`, `x_max`, `y_min`, `y_max`, `z_min`, `z_max`:

- Float
- Minimum and maximum x and y values (defines the range of each axis)
- Note that inputting z_min and z_max will set the colormap range to manual.

`x_size`, `y_size`:

- Integer
- Number of cells for each axis (x_size = columns, y_size = rows)

`show`:

- Boolean
- Sets whether the colormap should be shown on initialization
- Defaults to True

showColormap

Shows or hides the colormap.

Function signature/overloads:

- `orbital.showColormap(plot_id, show=True)`

`plot_id`:

- Same as setupColormap

`show`:

- Same as setupColormap

sendCMData

Sets a color map cell to some value.

Function signature/overloads:

- `orbital.sendCMData(plot_id, x_index, y_index, z)`

`plot_id`:

- Same as setupColormap

`x_index`, `y_index`:

- Integer >= 0
- Cell coordinates

`z`:

- Float
- Value of the cell to be set

saveData

An instant save feature (referenced by the term “auto-save” in the application) that allows for script-side saving (see the Saving section for more details).

Function signature/overloads:

- `orbital.saveData(data_set)`
- `orbital.saveData(data_set, autosave_header)`

`data_set`:

- Same as sendData

`autosave_header`:

- String
- Annotation for the data set in the save file

clearData

Clears the internal data buffer.

Function signature/overloads:

- `orbital.clearData(data_set)`

`data_set`:

- Same as sendData

renameCustom

Renames a custom function label. This is the same as renaming it through the Edit Custom Controls menu. See the Custom Controls section for more information.

Function signature/overloads:

- `orbital.renameCustom(custom_fn, new_name)`

`custom_fn`:

- Integer from 1 to 8
- Designates the custom function label to change

`new_name`:

- String
- New custom function label

setAutosave

Sets the auto-save flag. When set, the application will prompt the user with the auto-save settings prior to running the script. See the Saving section for more information.

Function signature/overloads:

- `orbital.setAutoSave(enable)`
- `orbital.setAutoSave(enable, save_file)`

`enable`:

- Boolean
- Enables or disables the auto-save flag (the application defaults to False when loading a script)

`save_file`:

- String
- File (including file path) to use as the auto-save file (this can be altered in the prompt)

setPlotArrangement

Sets the plot arrangement.

Function signature/overloads:

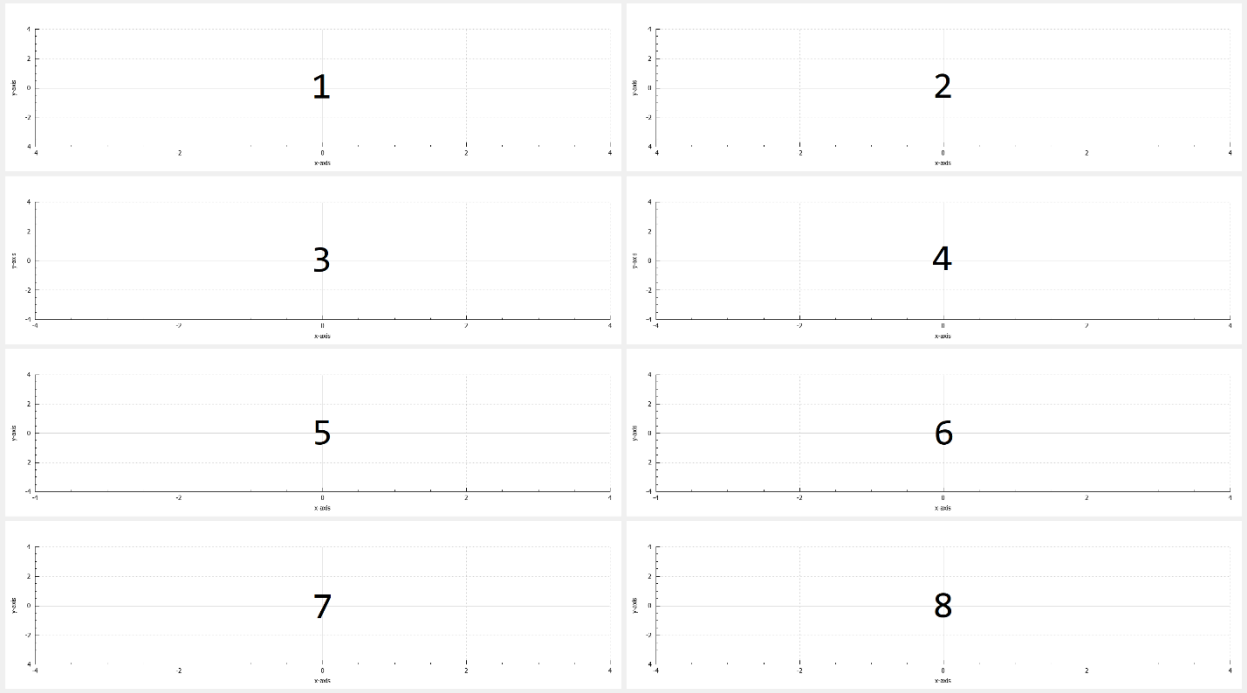
- `orbital.plotArrangement(plot_arrangement)`

`plot_arrangement`:

- Same as init

Plotting and the Plot Editor

One of the main features of Orbital is plotting data in real-time. Plots in Orbital are built using the QCustomPlot library by Emanuel Eichhammer. Orbital supports two types of plotting: 2-D plots and colormaps.



A maximum of 64 plots (8 x 8) can be created, and each plot can be either a 2-D plot or a colormap (or both, however only one may be displayed at a time). Each plot is indexed with an ID (labelled accordingly within the Plot Editor menu).

Interacting with a Plot

Axis Manipulation:

Each plot supports click-and-dragging to adjust the axes. Click and dragging a specific axis will lock the action to that specific axis. It is also possible to zoom in and out using the scroll wheel. This can also be locked to a specific axis.

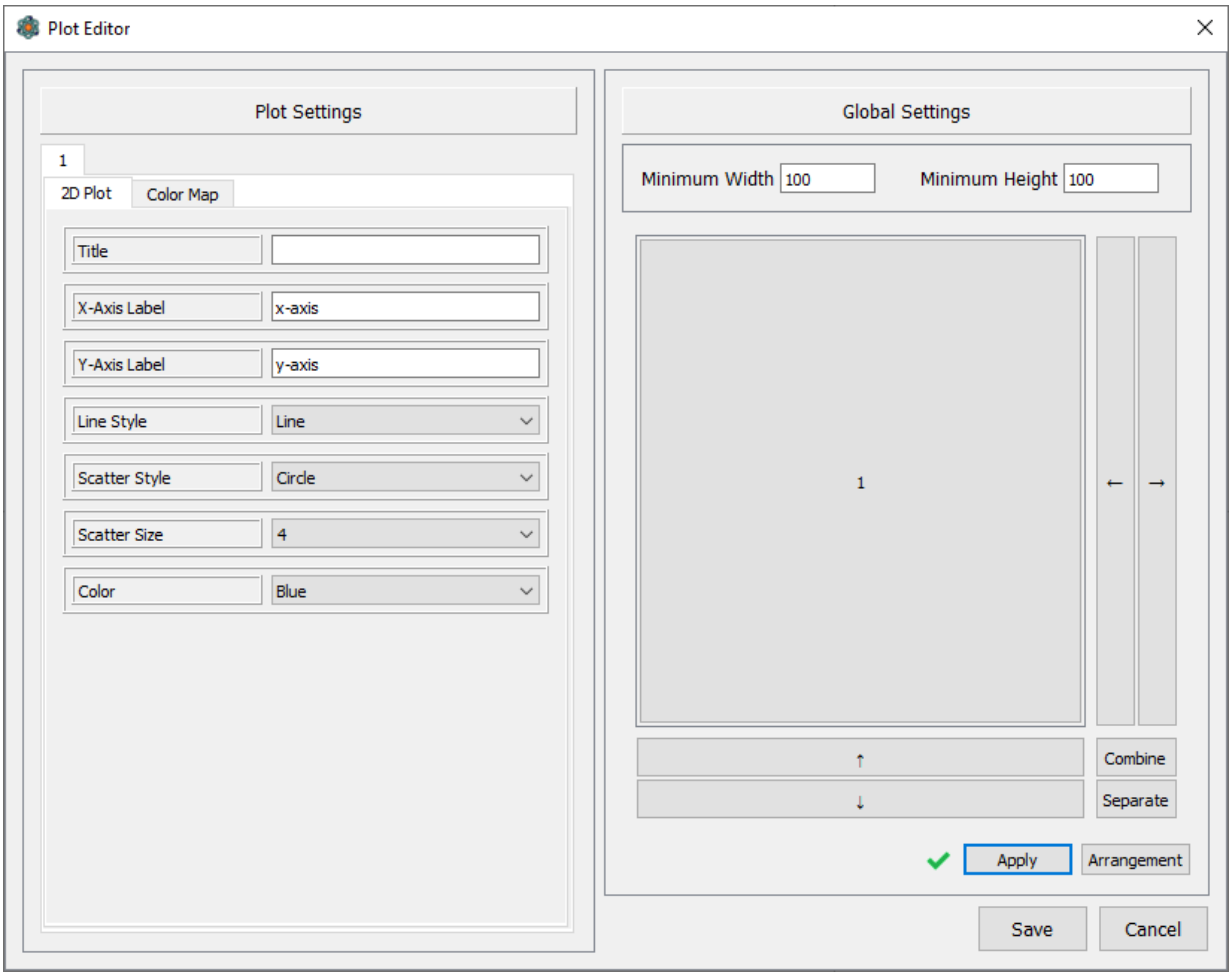
Reset Focus:

Double-clicking a plot will reset it to display all data points, effectively resetting the axes.

Auto-Rescale and Saving Image:

Right-clicking a plot gives two options: toggle the auto-rescale and save as PNG. The default behavior when displaying data is to automatically rescale the axes to show all data points. This can be toggled on and off by selecting the “Toggle Auto-Rescale” option.

The Plot Editor



The Plot Editor menu can be accessed via *File->Plot Editor*. Here, you have access to all the current plot properties. Most of the plot properties can also be set from scripts using the `set2DPlotProperty`, `setColormapProperty`, or `setupColormap` functions.

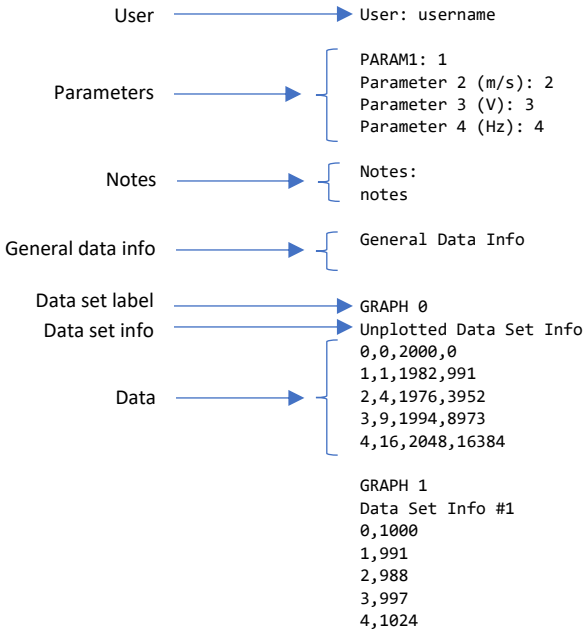
The global settings *Minimum Width* and *Minimum Height* effectively determine the size of each plot. Individual plot width and height are determined by the global minimum width and height and their arrangement—combining or separating plots in the arrangement will increase or decrease their size relative to other plots. The plots will *at least* fill the plot space (hence the "Min"). However, if the minimum size is set high enough, then the plot space will expand to accommodate the desired size.

Saving and Data Management

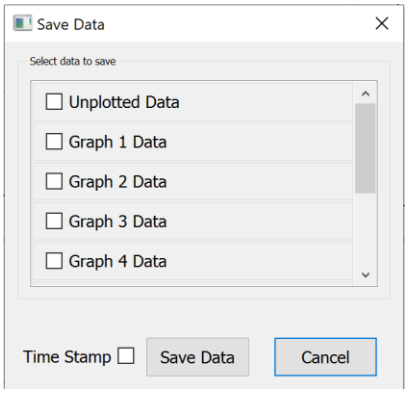
Saving in Alpha is accomplished one of two ways:

1. Manual saving (client-side saving)
2. Automatic saving (script-side saving)

Save files are saved in plaintext ASCII format. The default layout of a save file will display user information including parameters and values, general data information, and sequentially each data set including data set information followed by each data point in CSV format. A sample save file output is shown below. Each section is optional and can be enabled or disabled through either the Settings or Data menus. A "#" character in the data set label field will be replaced with the graph ID (use "\#" if you want the literal "#" character in the label).



Manual Saving



Manual saving is done via the "Save Data As" button. After data collection, a user can save the collected data by clicking the "Save Data As" button and select which data set to save (note that manual saving will be disabled while the script is running). Data sets are explained in more detail in the Data Management section.

Automatic Saving

Automatic saving (or auto-save) is done via the `saveData` callback function. In order for an auto-save to succeed, Alpha requires a save file. A save file can be set either through the Auto-Save settings in the Settings menu, or through the `setAutoSave` callback function. Note that the auto-save feature can be used without having the auto-save flag set, however, it is recommended if auto-saving is being used within a script. When an auto-save occurs, two additional headers are prepended to the data set label: an auto-save label and, if one was provided with the `saveData` callback function, an auto-save header. The auto-save label can be set from the application, but the header is always set from the callback function.

The auto-save callback function is a relatively slow process (compared to the other callback functions), and as such should be used sparingly or in between large batches of data. Auto-saving also doesn't clear the data it saves from its container, so it may be necessary to call the `clearData` callback function after an auto-save.

Data Management

A data set in the context of Alpha refers to the data corresponding to either the unplotted data container or any one of the plots. Note that only plot data can be saved—color map data cannot be saved!

While color map data cannot be saved, saving vector data (more than 2-tuple) is entirely supported. Alpha buffers data in containers of data points where each data point is a vector of up to 32 elements. A maximum of 9 containers can be managed at a time: one for unplotted data and 8 for each graph. The only caveat is that vector data must be saved one data point at a time as the `sendData` callback function must be used for this.