Oscopy
An interactive program for viewing electrical simulation results API Manual

Arnaud Gardelein

May 14, 2013

Abstract

Oscopy is an interactive oscilloscope written in python designed to simplify the electrical design workflow. It allow to read, view and post-process signals with support for automatic dependency tracking. File re-reading (updates) can be triggered by external applications like gEDA suite through D-Bus messaging system, and then Oscopy can call netlist generator and electrical simulator programs automatically. As oscopy is built on top of IPython, post-processing include as well as simple arithmetics operation as complex functions like FFT. Oscopy can be easily extended to a multi-purpose viewer, as adding new data file formats and new types of plots is really easy.

This document covers all important concepts and classes, and explain interactions between them.

1 Introduction

In the electrical system design workflow, viewing results from analog simulation or experiment is not a trivial task: there exist numerous different program with even more different file formats, the user interface has to be friendly and functional, and the program should be memory efficient due to the number of data points per file that can quickly grow.

The gEDA suite contains mainly all tools required to design electrical boards, from scheme drawings to PCB routing. There already exist several programs to view analog simulation results: gwave, GSpiceUI, dataplot.

Gwave is designed as a waveform viewer, an can read text file as well as binary file generated by Spice2, Spice3, ngspice, CAzM or gnucap. The user interface present features such as drag and drop signal into the graphs, vertical bar cursors, support for multiple files and multiples panels.

GSpiceUI is more focused on the user interaction between the user and the simulation program: it import the schematic from gschem, allow the user to build the file to be used by the simulation and plot the results, eventually using GWave.

Dataplot has support for format like gnucap, ngspice, hdf5 and touchstone. The user interface has a tabs for multiple plots, and present the data in a hierarchical manner.

Another way of viewing results is to use Octave (and generally gnuplot). This approach permit to post-process the results with operation such as FFT, diff. Support for multiple figures is present. Octave support HDF5 file format and tab-separated text-based files such as gnucap output. The user interaction is essentially based on command line interface.

The idea behind Oscopy is to combine the better of those approaches into a single program easily extendable. In this purpose, it present features like multiple plots, multiple windows, different plot types (linear, log) and allow the user to do math with data, including basic operations, trigonometry, fft, diff. It support the gnucap file format for input and output, and has an update mechanism to reread data from files. New file formats and new graph types can be added by following the guidelines presented in this document.

2 Oscopy: The API

Oscopy is designed to be an interface between the user and the results that comes as well from simulation as experiment (Figure 1). Thus the program interacts with two entities:

1. the user
2. the data

The results are considered to be a list of point representing an electrical signal property (amplitude, intensity, power...). In the following, results, data and signals appellations are used indifferently.

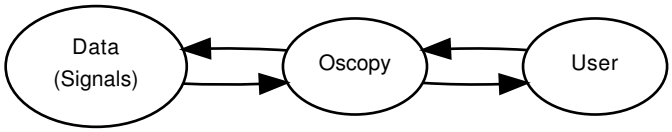

Figure 1: Oscopy is designed to be an interface between the electrical simulation (or experimental) results and the user.

2.1 User interaction

In a standard session, the user configure the way of viewing data, examine the results, do some adjustments such as add more data, do some post-processing... There are then two modes of the program:

1. Data viewing configuration
2. Result examination

Those two modes are non-exclusive and can be executed either consecutively or simultaneously.

The data viewing configuration mode gather all operations such as:

- interaction with the filesystem (read, write, update data from files)
- post-process data, mainly do math operations
- manipulate data to be displayed (e.g. add, remove data)

Results examination mode contains operations:

- zoom, pan views
- manipulate cursors

2.2 Data interaction

2.2.1 Interaction with files

Oscopy can import and export signals to files. This is done through the use of Reader-derived objects for import and Writer-derived objects for export. Reader parse the file and create as many Signal as needed, and transmit a dict of Signals to Oscopy.

For export, Oscopy transmit a dict of Signals to the Writer object, which write the data to the file in the desired format.

For more details, see Reader and Writer sections.

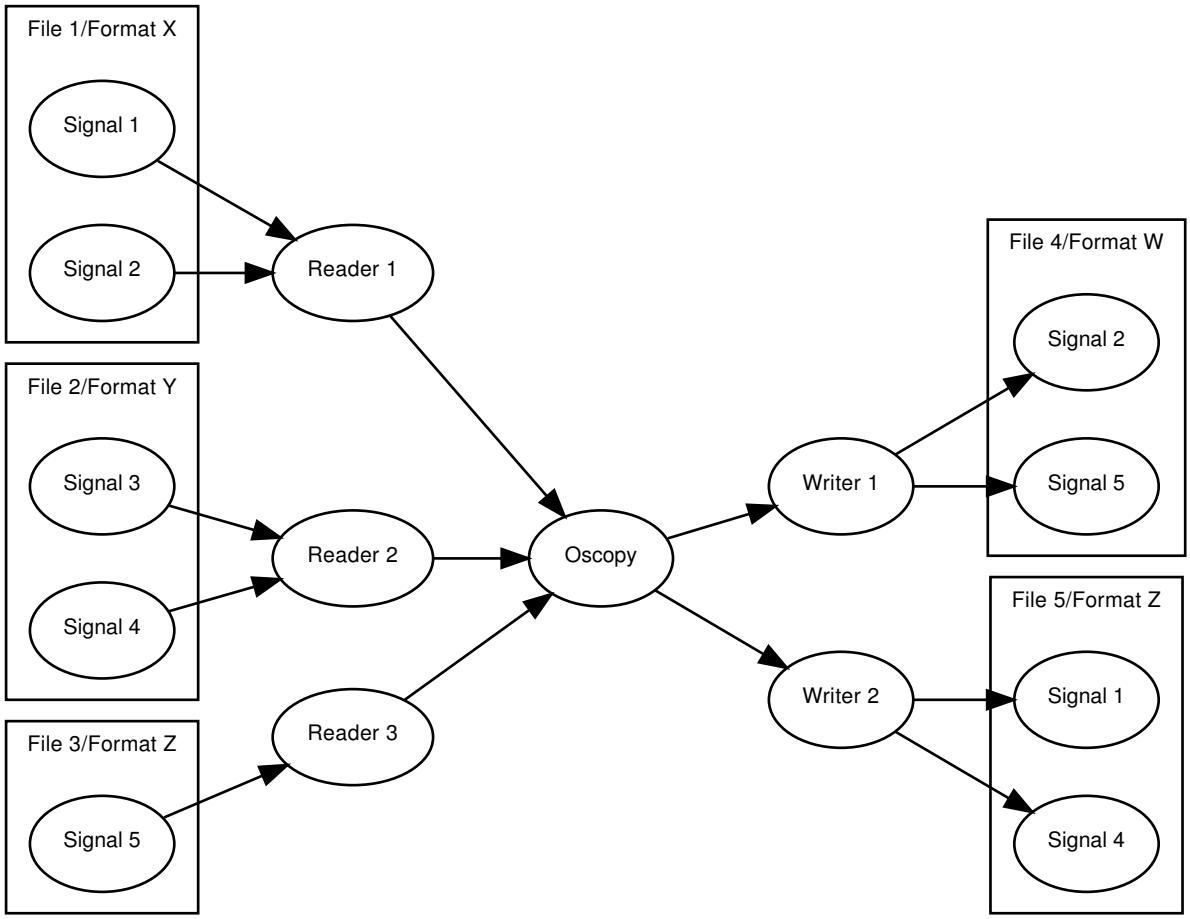

Figure 2: Oscopy get the data (or Signals) from files through Reader objects, and put data into files through Writer objects. Each Reader/Writer support a specific file format.

2.2.2 Viewer interaction

The results are presented to the user in a graphical manner, as a set of figures where the data is plotted. On each figures, there can be several graphs containing each one containing a plot. Each plot can hold as many signals as requested. The interaction between Oscopy and the

graphical representation is handled by Figure and Graph objects. They both use MATPLOTLIB to plot the data.

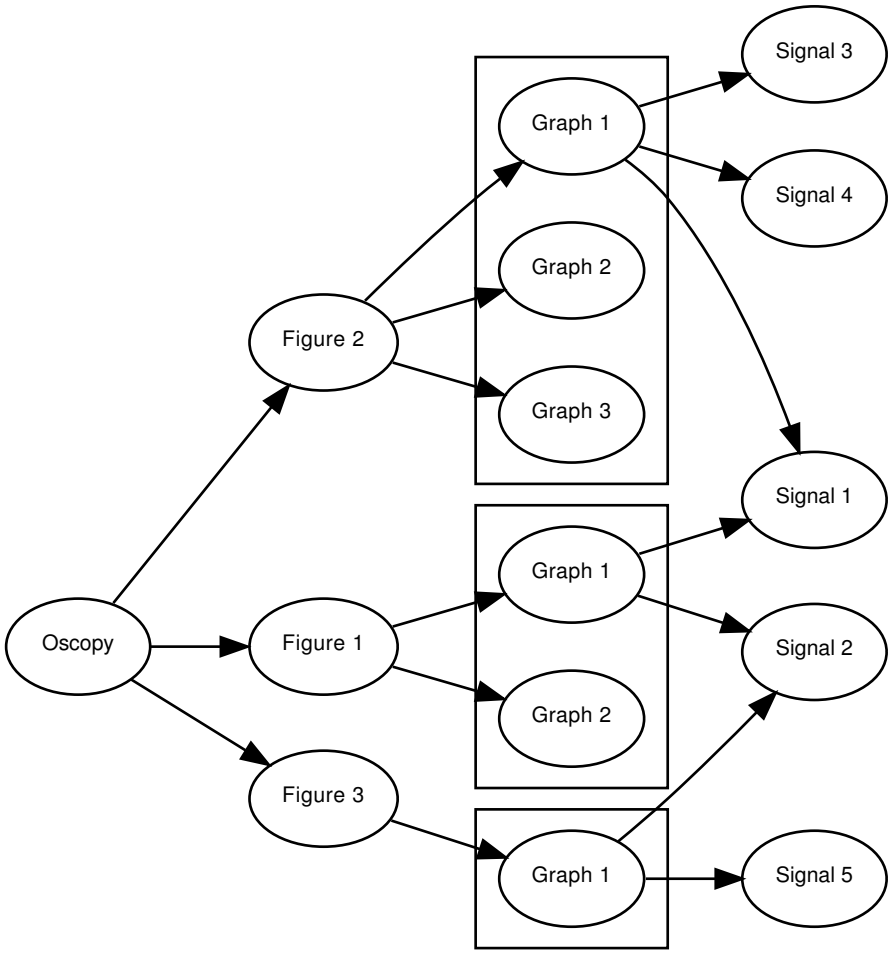

Figure 3: Interaction of Oscopy with the viewer. Oscopy communicate with Figures, which contains up to 4 Graphs. The Graphs contains the Signals.

2.3 Context

The user and data interaction are gathered into the object Context, which handle the instanciation of readers, writers and figures objects. It present an interface that can be split into a few parts:

- Data management
- Figure management

The methods of this interface are summarised in Table 1.

The property `signals` is a dict of Signals where keys are Signal names. The property `figures` is a list of figures handled.

Method	Access	Action
__init__	Public	Initialise the object
Data management		
read	Public	Load data from file
write	Public	Save data to file
update	Public	Reload data from files
freeze	Public	Disable update of some signals
unfreeze	Public	Enable update of some signals
import_signal	Public	Make an existing signal visible in this Context
names_to_signals	Private	Return a dict of signals from provided names
Figure management		
create	Public	Create a new figure
destroy	Public	Destroy a figure

Table 1: Methods of Context

Property	Read	Write	Type	Summary
figures	✓		dict	List of figures
signals	✓		list	Signals loaded

Table 2: Properties of Context.

3 Signals

A Signal is an object that basically contain the data points. For example this could be the Y axis data of a temporal measurement. Properties and methods are reported in Tab. 3 and 4 respectively.

Since generally in a simulation or experimental results many signals share the same scale, e.g. time scale or frequency scale, the Signal is associated to a reference signal. To make the difference between a "Signal" and a "reference Signal", the latter has its `ref` attribute set to `None`.

We humans give to each signal a name, e.g. V1, Iout... and associate a unit (Volts, Watts...). The Signal thus possess those two attributes.

And finally, when rereading the data from file, one want not always update each Signal, for instance to compare two simulations with different component values. When the attribute `frozen` is set, the Signal is not updated.

All those properties are implemented as GObject properties.

Attribute	Type	Read	Write	Comment
data	numpy.array	✓	✓	Data points
name	string	✓		Name of the signal
ref	Signal	✓	✓	Reference signal
unit	string	✓		Unit of the data
frozen	bool	✓	✓	Update signal or not

Table 3: Properties of Signals.

Method	Access	Action
Update mechanism		
on_begin_transaction	Public	Go to transaction and notify dependencies
on_end_transaction	Public	Exit transaction once all dependencies did
on_changed	Public	A dep of this Signal changed
on_recompute	Public	Recompute the data of this Signal
Operator overloading		
__make_method	Private	Wrap operator
__make_method_inplace	Private	Wrap operator, inplace version
__neg__	Public	Compute opposite signal

Table 4: Methods of Signals. Getters and setters are not reported here

4 Readers

The module READERS is used to load signals, mainly from files. It composed of a main class Reader, an exception class ReadError, the back-end classes and a function `DetectReader()`. Each file format is supported through a specific back-end. The back-end class is created by deriving the base class Reader and redefining only the way to 1) check if the file is supported and 2) load the signals from the file. The function `DectectReader()` should be used to automagically find the right back-end from the file name. The interaction between objects is summarised in Figure 4.

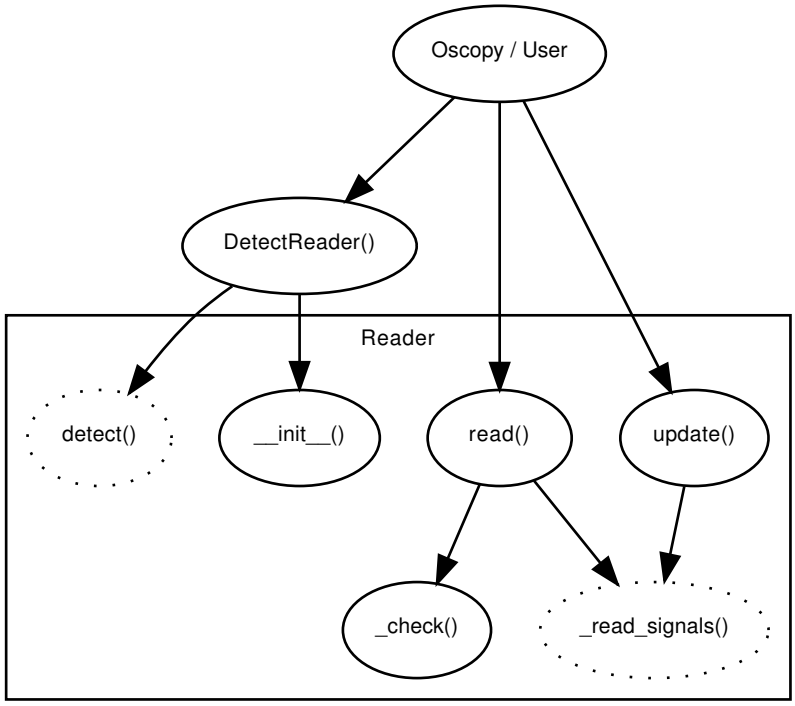

Figure 4: Call graph of Readers module. `DetectReader()` create the object for the user. The only visible functions to the user are `read()` and `update()`. The dashed method should be redefined when deriving Reader to support new file formats.

4.1 Reader

A Reader is used to load the data from a file. The Reader class provide the tools to ensure the file is readable and the update mechanism to make the new back-end definition easier. The Table 5 show the methods of this class.

The main method of this object is then `read()`, which is user-visible. However, this is `_read_signals()` that does the real work, i.e. parse the file and create the signals. The file format support is verified by `detect()`. Those two methods should be redefined when deriving Reader.

The file access is verified by `_check()`, and the update mechanism is handled through `update()`.

The property `signals` return the dict of signals handled by the reader.

The method `rename_signal()` set a new name for a Signal, which is kept across subsequent calls to `updates()`.

Method	Access	Action
__init__	Public	Initialise the object
__str__	Public	Return a string with the filename
read	Public	Read the data
update	Public	Reread the data
detect	Public	Return True if file is handled
_check	Private	Raise an exception if the file is not readable
_read_signals	Private	Read the data
rename_signal	Public	Rename a Signal
Update mechanism		
on_begin_transaction	Public	Go to transaction and notify dependencies
on_end_transaction	Public	Exit transaction once all dependencies did

Table 5: Methods of Reader

Property	Read	Write	Type	Summary
signals	✓		dict	Signals contained in the reader
info	✓		dict	Various information from and about the file

Table 6: Properties of Reader.

4.2 DetectReader

This function find the right back-end to read the file and return a valid Reader. It call the method `detect()` of each known Reader-based object until a True is returned, meaning the object can handle the file.

4.3 ReadError

This exception class is raised whenever an error is encountered during the file access, e.g. no file, bad file type... It contains only the constructor and a str method that returns the error message `value`.

4.4 Adding new Readers

New back-end can be created to support new file formats, this is done by deriving the Reader object. This object provide the necessary framework so that the derived class should only redefine two methods:

- `detect()` that returns True if the file format is supported
- `_read_signals()` that effectively parse the file and return a dict of signals

Boring tasks like file access check or update management are handled by Reader. Note that the constructor should also call the base class one.

5 Writers

This module is used to save data into a file. It is composed of a main class Writer, an exception class WriteError, the back-end classes and a function `DetectWriter()`. Each format is supported through a specific back-end. The back-end class is defined by deriving the base class Writer and redefining 1) the format name, 2) the way to check the format is supported and 3) how to save the data to file. The function `DetectWriter()` should be used to automagically find the right back-end from the format name. The interaction between the objects is summarised in Figure 5.

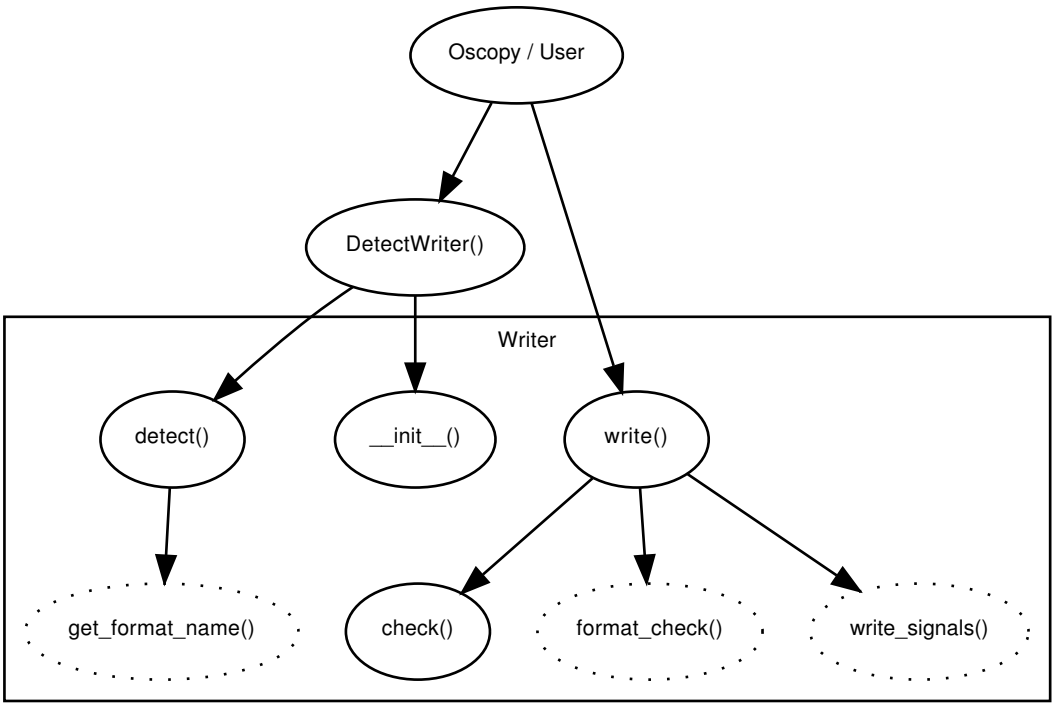

Figure 5: Call graph of Writers module. `DetectWriter()` create the object for the user. Only `write()` is visible to the user. The dashed method should be redefined when deriving Writer to support new file formats.

5.1 Writer

The purpose of the Writer is to save the data to a file. The Writer class provide the tools to make the definition of new export format easier. The Table 7 shows the methods of this class.

Only the methods `detect()` and `write()` are user visible. The real work is done by `write_signals()` that format the data and put them into the file. `detect()` is used to check whether the back-end class support the format. For now this is only a comparison between the name provided by the user and the result of the call to `_get_format_name()`.

Before calling `write_signals()`, `write()` check the file access (with `_check()`) and do a format specific verification by calling `_format_check()`. For example, for text columns-based formats like CSV, this could be checking that all signals share the same reference. This function should be redefined by the derived class.

Method	Access	Action
__init__	Public	Initialise the object
write	Public	Write the data, user visible function
detect	Public	Return True if format is handled
_check	Private	Raise an exception if the file is not writable
write_signals	Private	Write the data, to be defined by the back-end
_get_format_name	Private	Return the name of the format
_format_check	Private	Format-specific checks

Table 7: Methods of Writer

5.2 DetectWriter

This function find the right back-end to write the file and then return a valid Writer. It call the method `detect()` of each known Writer until a True is returned, meaning the object can handle the file format.

5.3 WriteError

This exception class is raised when an error occurs during the file access, for example bad file permissions. There is only a constructor and a str method which return the error message `value`.

5.4 Adding new Writers

New formats are supported by adding a back-end to the module, by deriving the Writer class. This class provide the requested framework (file access checking, detection mechanism) to make the new format adding simpler. Thus the new class should redefine the methods:

- `_get_format_name()` that return a string identifying the format
- `_format_check()` that verifies if signals are compliant with the format
- `write_signals()` that effectively write the data to the file

Note that the constructor of the new class should call the base class one.

The user can pass options to the back-end through the use of a dict, which is then available as `self.opts`. For example, the option "ow" is supported by the base class. When set to "1", the destination file can be overwritten.

6 Figures

A Figure is a container that manage Graphs, i.e. handle adding, deleting, formating and updating, as shown in Table 8. It handle the layout and the mode of the graphs, and pass all data-plotting specific commands to the Graphs, through the use of an alias to the current graph `curgraph`.

When updating, for each graph it prepare a list of signals to be updated and deleted and then pass it to the graph.

When `plot()` is called, it set up the figure, the layout, and then call the method `plot()` of each graph. Finally it set the method `key()` as a call-back for key-press related event in Matplotlib. Currently this method handle the keys '1', '2', '3' and '4' to toggle cursors, as summarised in Table 10. It then call `toggle_cursors()` from Graph.

The list of graphs contained into the figure is provided by the property `graphs`.

Method	Access	Action
__init__	Public	Initialise the object
Graph management		
add	Public	Add a graph into the figure
delete	Public	Delete a graph from the figure
Viewing management		
key	Private	Handle keystrokes when displaying plots
Data management		
update	Public	Update signal list of all graphs
Inherited methods		
draw	Public	Draw the figure on the canvas

Table 8: Methods of Figure

Property	Read	Write	Type	Summary
graphs	✓		list of graph	List of graphs
Graph management				
layout	✓	✓	str	Disposition of graphs in the figure
Data management				
signals	✓		iter	Signals contained in the figure

Table 9: Properties of Figure.

Key	Action
1	Toggle first vertical cursor
2	Toggle second vertical cursor
3	Toggle first horizontal cursor
4	Toggle second horizontal cursor

Table 10: Keystrokes handled by `key()` in Figure.

7 Graphs

The GRAPHS module is in charge to plot the data on one figure. Since there are many ways of plotting data like linear graphs or smith charts, the module is composed of one base class Graph that handle the signal dict and the basic plotting. This class is then derived, and the new ones generally redefine the way of plotting the data.

Additionally the user want a way to put references on the displayed data, thus the module contains a Cursor class to handle this.

7.1 Graph

A Graph is an object that present the data to the user. It has mainly two types of methods:

data management that e.g. insert or remove data from the Graph

cursors management that handle cursors related work e.g. toggling, displaying

The method `get_type()` that identify the type of the graph, `get_type()` do not fit in the previous categories. The Table 11 summarise the methods of the Graph object, and Table 12 the properties.

Method	Access	Action
__init__	Public	Initialise the object
find_scale_factor	Private	Find a human-readable multiplier to the data
Data management		
insert	Public	Insert signals into the graph
remove	Public	Take away signals from the graph
update_signals	Public	Synchronize plotted data with Signal data
Cursors management		
toggle_cursor	Public	Change cursor status
cursors_as_list	Public	Get a list of Cursors
_draw_cursors	Private	Display the cursors on the graph
_set_cursors	Private	Set the properties of a cursor
_print_cursors	Private	Display the numerical values of the cursors
Zoom and span management		
on_select	Public	Callback for spanning

Table 11: Methods of Graph

Property	Read	Write	Type	Summary
type	✓		str	Graph identification
unit	✓	✓	Tuple of str	The unit to be displayed in x/y labels
scale	✓	✓	str	Scale of the axis (one of lin,logx,logy,loglog)
range	✓	✓	Tuple of list[2]	Axis limits
signals	✓		iter	Signals contained in the graph
scale_factors	✓	✓	float, float	Scale factors
axis_names	✓		[str, str]	Name of each axis

Table 12: Properties of Graph.

7.1.1 Data management methods

The data is inserted into the graph with `insert()`, and removed with `remove()`. `insert()` return a dict of the signals that failed to be inserted, i.e. the reference signal name is different of the x axis name, the Y units do not match or signal with same name already present. When the graph is empty, the x axis name take the name of the reference signal. It can be parsed with the property `signals`.

7.1.2 Viewing management

A set of properties is present to customise the range, scale type and units. The range is the zoom area, the scale type is either linear or logarithmic, and the units are strings displayed along each axis.

7.1.3 Cursors management

From the external user perspective, cursors can only be toggled with `toggle_cursors()`. By default there are up to two cursors per axis to keep graph readable by a human. Internally, `toggle_cursors()` call `set_cursor()` to define cursors properties, and `plot()` call `draw_cursors()` and `print_cursors()` to display respectively the cursors shape and value onto the graph. The figure 6 summarise this paragraph.

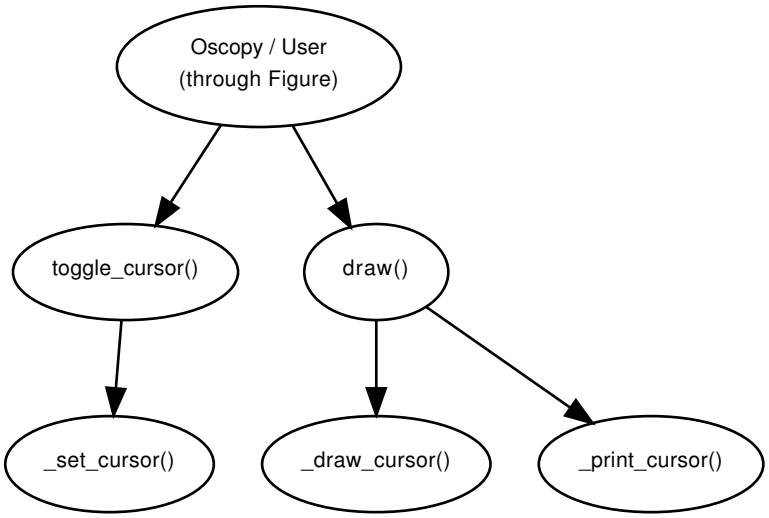

Figure 6: Interaction between the user, Graph and Cursor.

7.2 Cursors

When annotating the graphs with Matplotlib, changes are lost when the figure window is closed. Oscopy support cursors, i.e. a way to print references on a graph, through the use of annotation. The goal of the Cursor object is to make the references persistent between figure close.

The tables 13 and 14 summarise the methods and attributes of this object. Apart from the constructor and the string representation, it contains one method `draw()`, that handle the representation of the reference on the graph.

The main attribute is `value`, or the location on the graph of the reference. The cursor can be `visible`, i.e. printed on the graph or not, and can be either horizontal or vertical, as defined by `type`. The `line` object is a reference to the matplotlib.lines.Line2D where the cursor is drawn.

Method	Access	Action
__init__	Public	Initialise the object
draw	Public	Draw the cursor on the graph
__str__	Public	Print value, type and visible
get_line	Public	Get the matplotlib.lines.Line2D object

Table 13: Methods of Cursor

Attribute	Type	Read	Write	Comment
value	float	✓	✓	Position
visible	bool	✓	✓	State
line	matplotlib.Line2D			Where it is dawn
type	str	✓	✓	"horiz" or "vert"

Table 14: Properties of Cursor.

7.3 Adding new Graphs

Adding new ways of plotting data is done by deriving Graph. At least the method `get_type()` should be redefined, and the constructor of the new class should call the base class one.

7.4 Adding new Cursors

By deriving Cursor, new way of putting references on the graph can be defined. The method `draw()` then needs to be redefined.

8 Signal Update Mechanism

When simulation data file change, the user can update the files already loaded using `Context.update()`, oscopy will automatically manage the dependencies related issues.

To manage dependencies between computed signals this is performed using the GObject event system where Signals and Readers have the additional callbacks and events. Those are depicted respectively in Table 15 and Table 16.

To define a dependency of a Signal S1 with another Signal S2 (or a Reader):

1. Connect S1 callbacks `on_begin_transaction()` and `on_end_transaction()` to the events of S2 *begin-transaction* and *end-transaction* respectively
2. Connect S1 callback `on_changed()` to the event *changed* of S2
3. Connect S1 callback `on_recompute()` to the event *recompute* of S1 (not S2)

Once done, to change S2 data and make S1 be aware of it:

Make S2 emit *begin-transaction* This will notify all S2 dependencies (e.g. S1) that its data might changed

Change S2 data S2 will automatically emit *changed* to notify its dependencies that its data has changed

Make S2 emit *end-transaction* This will notify its dependencies that S2 will not change anymore. S1 will emit *recompute* and then recalculate its data with update S2 one before forwarding *end-transaction*.

A trivial example of use:

```
oscopy> s1=Signal('s1', 'V')
oscopy> s2=Signal('s2', 'V')
oscopy> s1.data=[1,2,3,4]
oscopy> s2.data=[1,2,3,4]
oscopy> s3=s1+s2
oscopy> s3
Out[16]: <Signal[0xaad8784] (s1 + s2) / s1 [None] data=[2, 4, 6, 8]>
oscopy> s1.emit('begin-transaction')
oscopy> s1.data=[11,12,13,14]
oscopy> s1.emit('end-transaction')
oscopy> s3
Out[20]: <Signal[0xaad8784] (s1 + s2) / s1 [None] data=[12, 14, 16, 18]>
```

Method	Signal	Reader	Action
on_begin_transaction	✓	✓	Go to transaction and notify dependencies
on_end_transaction	✓	✓	Exit transaction once all dependencies did
on_changed	✓		A dep of this Signal changed
on_recompute	✓		Recompute the data of this Signal

Table 15: Methods of Context

Event	When emitted	Meaning
begin_transaction	Before changing Signal data	Dependencies might change
changed	Right after changing Signal data	A dependency has been modified
recompute	Before ending transaction	Force recalculation of Signal data
end_transaction	After changing Signal data	Dependencies changes finished

Table 16: Methods of Context