Grapa

User manual

Table of content

1 General description

1.1 Description of Grapa

Grapa is a python package providing a graphical interface and the underlying code dedicated to the visualization, analysis and presentation of scientific data, with a focus on photovoltaic research.

A wide range of data formats can loaded by default. The produced graphs are saved both as graphical objects and as text files for later modifications.

The data analysis functions currently focus on photovoltaics and semiconductor material science. Advanced analysis or fitting functions are notably provided for the following characterization techniques: EQE, JV, C-V, C-f, TRPL, SIMS (list not exhaustive).

The software has extended capabilities for advanced plotting, and can be used for creating high-quality figures for scientific publications.

Last, the user can add to the software and to the graphical interface his own specific data loading functions as well new data types and analysis functions, with no modification of the existing code.

Grapa stands for "GRAphing and Photovoltaics Analysis".

Cheers!

1.2 Features overlook

- Open and merge a wide range of experimental data. Data import possible from clipboard.
- Graphical data presentation and graph customisation.
- Graph export in various file formats with user-defined dpi. Additionally create a .txt file containing the raw data for later modifications.
- The supported image formats are the one supported by the savefig function of matplotlib. .emf is also supported, provided a path to an inkscape executable is provided in the config.txt file.
- Drop-down menu to switch between data-relevant views (e.g. log-view for JV data, Tauc plot for EQE)
- Drop-down menu to switch between linear or logarithmic data visualisation.
- A data picker allows selecting and storing chosen data points.
- Can colorize a graph using color gradients.
- Some data types can be fitted. The parameters of the fitted curves can be modified afterwards.
- Data can be loaded from other files or from clipboard, then converted to the desired data type to benefit from data visualisation and processing and fitting options.

2 Installation

2.1 Prerequisites

Grapa was developed using the Winpython environment, with versions corresponding to python 3.4 and 3.5. Retrocompatibility with earlier versions of python is not ensured.

Grapa was developed using matplotlib 1.5 and should be compatible with version 2.0 and later.

2.2 Installation

The installation procedure is quite simple. Download grapa and place its content in a folder named “grapa” somewhere on your hard drive.

The user interface can be started by executing the file GUI.py using your favorite python distribution.

Grapa can also be used in scripts. To access its code you should include in your sys.path the folder containing the folder “grapa”.

Enjoy!

3 General organization of the package

The grapa packages comes with a graphical user interface (GUI), which allows handling most features of the package. Most matplotlib plotting methods are supported, as well as insets and subplots.

Using grapa in python code allows further customization of the produced graphs. Notably some matplotlib functions are not fully supported by the package. The Graph.plot() method returns the produced figure and axes for further modifications. Existing figures and axes can as well be provided.

The central object of the package is the Graph object. A graph is stored in a Graph object. The Graph class can be thought as a combination of a dict storing plotting information (axes, text annotations, etc.) together with a list of Curves which stores the data and the corresponding plotting information (color, aspect, etc.).

The interfaces are organized as to handle pairs of (x,y) vector data, although this might not always be true.

4 Graph object

The Graph object can notably plot and export itself.

As provided grapa can open a range of file formats. Just have a try on your own files, grapa might be able to parse it if the data are organized column-wise. Otherwise child classes of Graph can be implemented in order to read your own files (see the corresponding section of this howto)

4.1 Keywords to Graph

A Graph object notably contains a dict storing information related to plotting. Below is a list of possible keywords arranged bycategories, with a brief explanation of the possible values and examples.

== Figure ==		
figsize	Figure size (inch).	(6.0, 4.0)
subplots_adjust	Margins (relative to figure).	0.15 (bottom only), [0.125, 0.1, 0.9, 0.9] left,b,r,top, or [1,1,5,3.5,'abs']
dpi	Resolution dots-per-inch.	300
fontsize	Font size of titles, annotations, etc.	12
title	Graph title, based on ax.set_title().	My data ['A dataset', {'color':'r'}]
== Axes ==		
xlim	Limits of x axis, based on ax.set_xlim().	[2,9] ['',4]
ylim	Limits of y axis, based on ax.set_ylim().	[0,100] [0,'']
xlabel	Label of x axis, based on ax.set_xlabel().	Axis x [unit] ['My label', {'size':6, 'color':'r'}]
ylabel	Label of y axis, based on ax.set_ylabel().	Axis y [unit] ['My label', {'size':6, 'color':'r'}]
xticksstep	Value difference between ticks on x axis, or ticks positions. Loosely based on ax.xaxis.set_ticks()	0.01 [0,1,2]
yticksstep	Value difference between ticks on y axis, or ticks positions. Loosely based on ax.yaxis.set_ticks()	0.01 [0,1,2]
xtickslabels	Customized ticks. First is a list of values, then a list of labels, then possibly options. Loosely based on plt.xticks().	[[0,1],['some','value']] [None, None, {'rotation':45, 'size': 6, 'color':'r'}]"
ytickslabels	Customized ticks. First is a list of values, then a list of labels, then possibly options. Loosely based on plt.yticks().	[[0,1],['some','value']] [None, None, {'rotation':45, 'size': 6, 'color':'r'}]
xlabel_coords	Position of xlabel, based on ax.xaxis.set_label_coords().	-0.1 [0.5,-0.15]"
ylabel_coords	Position of ylabel, based on ax.yaxis.set_label_coords().	-0.1 [-0.1,0.5]
== Legends ==		
legendproperties	Position, or keywords to ax.legend().	best sw {'bbox_to_anchor':(0.2,0.8), 'ncol':2, 'fontsize':8}
legendtitle	Legend title	Some title ['Some title', {'size':25}]
== Annotations ==		

axhline	Horizontal line(s), based on ax.axhline().	0 [-1, 1]
axvline	Vertical line(s), based on ax.axvline().	0 [-1, 1]
text	Annotations, use GUI window if possible. Text, textxy, textargs must be elements, or list with same number of elements.	Some text ['Here', 'There']
textxy	Use GUI window if possible.	(0.05, 0.95) [(0.2, 0.3), (0.8, 0.9)]
textargs	Use GUI window if possible.	{'fontsize':15} [{'horizontalalignment': 'right', 'xytext': (0.4, 0.65), 'arrowprops': {'shrink': 0.05}, 'xy': (0.46, 0.32)}, {}]
== Secondary axes ==		
twiny_xlabel	Secondary x axis label.	Other axis x [unit] ['My label', {'size':6, 'color':'r'}]
twinx_ylabel	Secondary y axis label.	Other axis x [unit] ['My label', {'size':6, 'color':'r'}]
twiny_xlim	Secondary x axis limits.	[2,9] ['',4]
twinx_ylim	Secondary y axis limits.	[0,100] [0,'']
== Misc ==		
alter	Data transform keyword, specific to the type of manipulated data.	Linear ['nmeV', 'tauc']
typeplot	General graph plotting instruction, based on ax.set_xscale() and ax.set_yscale().	plot semilogx etc.
arbitraryfunctions		[['xaxis.set_ticks',[[1.5, 5.5]],{'minor':True}], ['set_xticklabels',[['a','b']],{'minor':True}]]

5 Curve object

A Curve object contains the data as well as a dict storing information dedicated to the Curve, notably plotting information.

Child classes of Curve can be implemented in order to provide you the suitable data analysis tools.

A list of possible keywords to the Curves is presented below.

5.1 Provided Curve child classes

Parsing of several file formats is provided, and data types are supported for processing of specific types of measurements.

Every Curve type provides dedicated functions (for analysis, plotting, etc.) as well as specific data visualizations accessible from a drop-down menu in the GUI.

The provided Curve types can be organized in two groups. First the Curve types related to data manipulation and presentation:

	Functions
Subplot Used to create an array of axes in the figure, organized in a grid paving the space available based on the gridspec formalism of matplotlib. Unless specified otherwise, the next Curves will be displayed in the newly created axes.	
	- File subplot: a path to a graph file to be inserted as subplot. If not provided, the Curve's data will be showed in the created axis. - Colspan, rowpan: how many grid locations the subplot will fill - Update subplot: a dict whose elements will to given to the graph loaded by file subplot - Number cols: how many columns there are in the grid - Transpose: by default the grid is filled row-by-row. If set the grid is filled column-by-column - Show id: displays small (a), (b) etc. on top-left corner of each created axes. Coordinates relative to that corner can also be provided (in figure fraction) - Width ratios, height ratios: list of relative width/heights for the axes in the grid - Subplots_adjust: margins and spacings of the graphs: [left, bottom, right, top, wspace, hspace]
Inset Creates new axes in the figure and plot an external graph file inside. When combining subplots and insets the inset curves must be placed after the subplots.	
	- Fileinset: a path to a graph file to be inserted as inset. If not provided, this and the next curve's data will be showed in the created axis. - Coords: where to place the new axis - Update inset: a dict whose elements will to given to the graph loaded by fileinset

Image handles pictures as well as display of matrices	
	- Plot type: imshow, contour, contour. Choose imshow for pictures. - Imagefile: indicate path to a picture (as of v0.4.2.1 not implemented for matrices) - Transpose, rotate: seld-explaining. Rotation in degrees for image, by 90° steps for matrices - Extent: allow zooming on the image. See matplotlib help for imshow for more details - cmap, vmin, vmax: provides a colormap, and min and max values for matrices aspect ratio, interpolation
Math operations	
	- Addition, subtraction, multiplication and division with a scalar or a Curve. Operations with another Curve perform an interpolation of both Curves on the different x values. - Negation (0 - Curve) and inversion (1 / Curve) operations Swap x and y series

The second group of Curve types is related to data analysis, with a focus on photovoltaics and material science.

EQE external quantum efficiency		
	- Bandgap from Tauc (E * EQE)^2, manual tuning of the curve fit possible - Bandgap from derivative - Bandgap from exact Tauc method (E * ln(1-EQE))^2 - EQE current, using the AM1.5G solar spectrum - Fit of low-energy side exponential decay (Urbach tail) - Loading of the Empa 20.4% cell EQE	- EQE vs eV - Tauc plot (E*EQE)^2 vs E
JV, temperature-dependent JV		
	- Area correction - Calculation of Voc, Jsc, FF, Eff, MPP - Fit of the JV curve using diode with 2 resistors model - Manual tuning of the curve fit possible, as well as resampling to higher number of points.	- logarithmic view log(abs(J - Jsc))
Jsc – Voc pairs Jsc-Voc pairs at different light intensities and temperatures		
	- Area correction - Fit of Voc vs Jsc: fit the diode ideality factor and J0, while separating the data according the the temperature. Limits to Jsc and Voc can be set. - Separate according to temperature: separate the data source according to the temperature	- Log10 of absolute value
PL spectrum		
	- Add arbitrary offset (a dark is probably more suited)	- nm to eV

	- Substract dark: dark correction (curve subtraction) - Convert x axis data from nm to eV	- nm to cm-1
SIMS		
	- Some curves are automatically hidden at data loading. - Edge detection on elemental traces - Depth calibration - Adjustment of elemental yield coefficients to match known elemental ratios over some data range - Computation of arbitrary elemental ratios - Generation of arbitrary elemental ratio curves (e.g. GGI, Ga+In, etc.)	- Sputter time or depth (once calibrated)
C-V capacitance-voltage		
	- Fit Mott-Schottky	- Mott-Schottky plot (1/C^2 vs V) - Carrier density N vs V - N vs depth
C-f capacitance-frequency Further data processing to be performed by peak selection in derivative visualization and Arrhenius fitting.		
		- Semilogx - Derivative
TRPL time-resolved photoluminescence decays		
	- Add temporal offset, add vertical (background level) offset - Fit decay with constant plus arbitrary number of exponentials functions, with possibility of fixing some parameters. The residuals can also be shown.	- semilogy
XRF MCA		
		- Channel to Energy [keV] conversion

5.2 Keywords to Curve

A Curve object contains a dict storing information notably related to plotting. The keywords are not restricted, and only the keywords corresponding to the signature of the matplotlib plotting function will be given to the plotting method.

Please have a look to the matplotlib documentation for a list of possible keywords.

Keyword	**Details**	**Examples**
type	Plotting method of Axes.	== usual methods == 'plot', 'fill', 'errorbar', 'scatter', 'boxplot'

	Only a few have been extensively tested, but most should work as far as a specific argument order is not required.	== similar to plot == 'semilogx', 'semilogy', 'loglog', 'plot_date', 'stem', 'step', 'triplot',
		== (no linespec) == 'bar', 'barbs', 'barh', 'cohere', 'csd', 'fill_between', 'fill_betweenx', 'hexbin', 'hist2d', 'quiver', 'xcorr',
		== 1D vector data == 'acorr', 'angle_spectrum', 'eventplot', 'hist', 'magnitude_spectrum', 'phase_spectrum', 'pie', 'psd', 'specgram',
		== other == 'spy', 'stackplot', 'violinplot', 'imshow', 'contour', 'contourf'
linespec	Format string.	'--or'
label	Curve label to be shown in legend.	'Experiment C'
== Display ==		
color	Curve color.	'r', 'purple', '[0.5,0,0]'
alpha	Transparency.	'0.5', '1'
linewidth	Linewidth in points.	'1.5'
marker		'o', 's'
markersize	Size of marker in points.	'2.5'
markerfacecolor	Marker inner color.	'r', '[0.5,0,0]'
markeredgecolor	Marker border color.	'r', '[0.5,0,0]'
markeredgewidth	Marker border with in points.	'1.5'
== Offsets ==		
offset	Offset to data.	'-10', '[2,20]'
muloffset	Multiplitcative offset to data.	'0.01', '[10, 1e2]'
== For specific curve types ==		
facecolor	Color of "fill" Curve types.	'fill', 'r', '[0.5,0,0]'
cmap	Colormap, for Curve types which accept this keyword (scatter, etc.)	'afmhot', 'inferno', "[[0.91,0.25,1], [1.09,0.75,1], 'hls']"
vminmax	Bounds values for cmap.	'[0,7]', "[3, '']"
colorbar	If not empty display a colorbar according to keyword cmap.	'1', "{'ticks': [-1, 0, 2]}", "{'orientation':'horizontal', 'adjust':[0.1, 0.1, 0.7, 0.1]}"
xerr	x error for curve type "errorbar".	'errorbar', '1', '5'
yerr	y error for curve type "errorbar".	'errorbar', '1', '5'
== Misc ==		
labelhide	Use "1" to hide label in the graph.	'1'
ax_twinx	Plot curve on secondary axis.	'True'
ax_twiny	Plot curve on secondary axis.	'True'
linestyle	Use "none" to hide a Curve.	'none'
['key', value]	User-defined keyword-values pairs. Will be fed to the plotting method if possible. Only necessary in the GUI.	['fillstyle', 'top'], "['comment', 'a valuable info']"

6 Automated data processing scripts

A few scripts are provided for data processing. Here is a short description of the required input and of the effect of the scripts.

6.1 Load all files from one folder

Quite obvious

6.2 JV data fit cell-by-cell

- asks for a folder,
- loads all JV data in that folder,
- perform area correction if 2-column cell/area file is found (a search for such file is reported in console at beginning of script execution),
- for each cell, select one dark and one illuminated curve, if possible,
- fit the data with the 1 diode with 2 resistors model

The output the following:

- For each cell, compiled dark+illuminated plots. Have a look to the quality of the fit.
- Compiled database files of the cell properties (all, dark only, illuminated only).
- Graphical sample maps of the different JV parameters values.

6.3 JV data fit separately

Similar the JV data fit cel-by-cell. All JV files in a folder are fitted and the results are compiled in a database file.

6.4 JV sample maps:

Asks for a JV database file (either output of JV setup, or of cell-by-cell script above)

Output a graphical representation of each cells JV parameters

6.5 JV boxplots

Creates boxplot graphs from column-organized data files in a given folder.

- asks for a folder,
- open each JV database files present and create boxplots for each relevant JV parameter.

If no file is recognized as a JV database file, tries to aggregate column-wise the data found in each of the data file.

6.6 Processing of C-V data temperature series

Processes a folder containing C-V data as function of temperature (1 curve per file).

Produces a series of graphs (only the last is shown in the user interface)

- asks for a folder,
- generates the following graphs: C vs V, 1/C^2 vs V (Mott-Schottky), apparent doping vs depth, built-in voltage versus temperature.

6.7 Processing of C-f data temperature series

Processes a folder containing C-f data as function of temperature (1 curve per file).

Produces a series of graphs (only the last is shown in the user interface)

- asks for a folder,
- generates the following graphs: C vs f, derivative(C, log(f)) vs f, impedance angle.

From the aggregated data the inflection points can be selected using the data picker, and the defect energy can be extracted from an Arrhenius relation.

7 Configuration file

Some user preferences can be stored in a configuration file and indicated to the Graph object at creation. A personalized configuration file can also be provided to the GUI when using a application launcher as a second argument (it will be retrieved using sys.argv[1])

The configuration file will be opened by grapa as a standard Graph. Whenever needed the software will query for provided values, and use default settings if no suitable keyword is found.

A configuration file may look like this one:

```
# only keywords (first word) matters: comment line are read, maybe stored with
corresponding keyword (e.g. '#'), but never refered to
# repeating a keyword will overwrite the content of first instance

# graph labels default unit presentation: [unit], (unit), / unit, or [], (), /
graph_labels_units  []
# graph labels presence of symbols (ie: $C$ in 'Capacitance $C$ [nF]')
graph_labels_symbols      False

# path to inkscape executable, to export in .emf image format. Can be a string, or a
list of strings
inkscape_path["C:\Program Files\Inkscape\inkscape.exe"]

# GUI default colorscales. Each colorscale is represented as a string (matplotlib
colorscales), or a list of colors.
# Each color can be a [r,g,b] triplet, a [r,g,b,a] quadruplet, or a [h,l,s,'hls']
quadruplet. rgb, rgba, hls, hsv colorscape are supported.
GUI_colorscale00    [[1,0,0], [1,1,0.5], [0,1,0]]
GUI_colorscale01    [[1,0.3,0], [0.7,0,0.7], [0,0.3,1]]
GUI_colorscale02    [[1,0.43,0], [0,0,1]]
GUI_colorscale03    [[0.91,0.25,1], [1.09,0.75,1], 'hls']
GUI_colorscale04    [[0.70,0.25,1], [0.50,0.75,1], 'hls']
GUI_colorscale05    [[1,0,1], [1,0.5,1], [0.5,0.75,1], 'hls']
GUI_colorscale07    'inferno'
GUI_colorscale10    'gnuplot2'
GUI_colorscale11    'viridis'
GUI_colorscale12    'afmhot'
GUI_colorscale13    'YlGn'

# default saving image format
save_imgformat      .png
```

8 How to read my own data files ?

You will need to create in folder datatypes a file with name graphMyfile.py (must start with graph with a small g). This file contains a class GraphMyfile (capital G, same ending as file name) that inherits from class Graph.

The class must contain at least 1 attribute and 2 methods:

- FILEIO_GRAPHTYPE: some string identifier, for internal use.
- **isFileReadable**(cls, fileName, fileExt, line1='', line2='', line3='', **kwargs)
  Must return True is the file can be opened using your class, otherwise False
- **readDataFromFile**(self, attributes, **kwargs)
  Actually opens the file, which can be access from self.filename.
  attributes is a dict which can be given to the constructor of the Curves.

8.1 Example:

Let us assume your data file is organized as follows (file howto/example_sinx_x.txt):

```
This is my own data format
My label
-29.9  -0.033394483
-29.7  -0.033316063
-29.5  -0.03189937
-29.3  -0.029182928
-29.1  -0.025257719
-28.9  -0.020264268
-28.7  -0.014387719
-28.5  -0.007851075
```

Such a column-organized file can very likely be opened using the provided methods. Still, you may want to add your own stuff there.

In file howto/graphMydata.py is provided an example how such a file can be parsed. To have to example work just copy the file into the datatypes folder. The class is organized as follows:

```
import numpy as np
from grapa.graph import Graph
from grapa.curve import Curve

class GraphMydata(Graph):

    # will be used by grapa code
    FILEIO_GRAPHTYPE = 'MyCurve'

    # 3-element list, default x and y axis labels: [text, symbol, unit]
    AXISLABELS = [['theta', '\theta', 'unit'], ['value', '', 'unit']]

    @classmethod
    def isFileReadable(cls, fileName, fileExt, line1='', line2='', line3='',
**kwargs):
        """
        This function returns True if the file can be opened with this class.
        Please be selective so as not to open other file formats by mistake.
        Are provided the file name, extension, and the 3 first lines of the file
        """
        if fileExt == '.txt' and line1 == 'This is my own data format':
            return True
        return False

    def readDataFromFile(self, attributes, **kwargs):
        """
        This function parses your files
        The file can be opened using open(self.filename, 'r')
        """
        # interpret headers - here only attribute 'label'
        attrs = {}
        f = open(self.filename, 'r')
        f.readline()
        attrs['label'] = f.readline().strip()
        f.close()
        # read data
        data = np.array([[np.nan], [np.nan]])
        kw = {'skip_header': 2, 'usecols': [0, 1]}
        try:
            data = np.transpose(np.genfromtxt(self.filename, **kw))
        except Exception as e:
            if not self.silent:
                print('WARNING cannot read file', self.filename)
                print(type(e), e)
        # create Curve object(s)
        try:
            from grapa.datatypes.curveMycurve import CurveMycurve
            self.append(CurveMycurve(data, attributes))
        except ImportError:
            self.append(Curve(data, attributes))
        self.curve(-1).update(attrs) # file content may override default
attributes

        # cosmetics
        # some default settings
        self.update({'ylim': [-1.2, 1.2],
                     'xlabel': self.formatAxisLabel(GraphMydata.AXISLABELS[0]),
                     'ylabel': self.formatAxisLabel(GraphMydata.AXISLABELS[1])})
```

9 How to define my own Curve type ?

You'll need to create in folder datatypes a file named curveMycurve.py (file name must start with curve with a small c). The file must contain a class CurveMycurve (capital C, same ending as the file name) inheriting from class Curve.

9.1 Example

An example is provided in file howto/curveMycurve.py. To have the example work just place a copy of the example files in the folder datatypes. The structure is as follows:

```
 import numpy as np
from grapa.curve import Curve

class CurveMycurve(Curve):

    # so the Curve class information is retrieved when opening an existing file
    CURVE = 'Curve mycurve'

    def __init__(self, data, attributes, silent=False):
        """
        Constructor with minimal structure: Curve.__init__, and then set the
        'Curve' parameter.
        """
        Curve.__init__(self, data, attributes, silent=silent)
        self.update ({'Curve': CurveMycurve.CURVE})


    # GUI RELATED FUNCTIONS
    def funcListGUI(self, **kwargs)
        """ Fills in the Curve actions specific to the Curve type """

    def alterListGUI(self)
        """ Determines the possible data visualisations. """

    # data transform function
    def y_derivative(self, index=np.nan, xyValue=None)

    # data transform function
    def y_abs(self, index=np.nan, xyValue=None)

    # function called by user
    def statsAndAddition(self, what)

    # a Curve type specific function printing help for the user
    def printHelp(self)
```