ONERA

THE FRENCH AEROSPACE LAB

tkPlotXY Documentation

Release 3.1

/ELSA/MU-10020/V3.1

May 28, 2020

CONTENTS

CHAPTER

ONE

PREAMBLE

tkPlotXY is a 2D plotting library based on Matplotlib. The aim of tkPlotXY is to provide to users an easier scriptable interface and a useful graphical interface in the mean time. This documentation focuses only on the scriptable interface. To know more about its graphical interface, some tutos will soon be available.

tkPlotXY uses preferentially 1D-data from pyTrees but in the scriptable interface, some other ways to define datas are available and will be exposed in this document.

This module is part of Cassiopee, a free open-source pre- and post-processor for CFD simulations.

For use in a python script, you have to import tkPlotXY module:

```
import tkPlotXY
```

CHAPTER

TWO

LIST OF CLASSES

tkPlotXY is based on classes. Some of them are internal classes used for display. They are not documented here. It has to be remarked that some classes have a 'TK' suffix at the end of their name. These classes are equivalent to the one without suffix, but they have been developped to work inside the tkInter context. It means that for python scripting only classes without the suffix 'TK' should be used.

– Classes

class tkPlotXY.**GraphEditor**(*display*)
The class GraphEditor is an encapsulation of class Desktop.

class tkPlotXY.**Desktop**
An object of class Desktop allows you to create all your graphs.

class tkPlotXY.**Graph**(*parent*, *name*, *conf*, *dpi=None*, *figsize=None*)
An object of class Graph corresponds to a window where plots are drawn. A graph window can manage several plots.

class tkPlotXY.**Axis**(**args*, ***kwargs*)
An Axis object contains the X-DirAxis and the Y-DirAxis of a given plot inside a Graph object. Multiple axis are available for a single plot.

class tkPlotXY.**DirAxis**(*axis_logscale*, *axis_autoscale*, *axis_min*, *axis_max*, *axis_label*, *axis_inverted*, *axis_visible*, *axis_position*, *axis_offset*, *axis_label_fontsize*, *axis_label_format*)
DirAxis contains the settings of the X or Y axis. This settings can directly be accessed from the class Axis.

class tkPlotXY.**Legend**(**args*, ***kwargs*)
An object of class Legend configures the legend for a given plot inside a Graph window.

class tkPlotXY.**AxisGrid**(*display*, *grid_color*, *grid_style*, *grid_width*, *grid_tick_number*, *grid_tick_size*)

class tkPlotXY.**Grid**(**args*, ***kwargs*)
Grid contains the main grid and the second grid. They can be configured directly by

accessing to Grid or to the proper GridLevel object. In case of a multiple axis usage, then multiple Grid objects can be attached to a given plot.

class tkPlotXY.**LevelGrid**(*x_display*, *x_grid_color*, *x_grid_style*, *x_grid_width*, *x_grid_tick_number*, *x_grid_tick_size*, *y_display*, *y_grid_color*, *y_grid_style*, *y_grid_width*, *y_grid_tick_number*, *y_grid_tick_size*)

class tkPlotXY.**Curve**(**args*, ***kwargs*)

Curve class describes all the settings concerning a given curve itself.

class tkPlotXY.**SubPlotParams**(**args*, ***kwargs*)

SubPlotParams is one way (TightLayout) to set margin, padding for plots positionning inside the Graph window.

class tkPlotXY.**TightLayout**(**args*, ***kwargs*)

TightLayout is one way (SubPlotParams) to set margin, padding for plots positionning inside the Graph window.

class tkPlotXY.**Movie**(*fig*, *filename*, *fps=10*)

Class Movie can be used to generate a movie in case of a dynamic plot (Co-processing for instance)

CHAPTER

THREE

GRAPHEDITOR

class tkPlotXY.**GraphEditor**

An object of class GraphEditor allows you to create a Desktop. Accessing to the Desktop will give you the possibility to plot all your graphs. Moreover, the Desktop contains all the data that can be used to generate plots. For python scripting interface, the first step is to create an object of this class GraphEditor and to access its Desktop. This is performed by the function openGraphEditor that directly returns the Desktop. Then the data can be added to this Desktop object. Finally it is used to generate all the graphs.

The first step is to create a graphEditor and to get its Desktop using :

tkPlotXY.**openGraphEditor**()

tkPlotXY.openGraphEditor(display)	Create an object of class GraphEditor and returns its Desktop.

```
import tkPlotXY as tkP
# Create a graphEditor
graphDesktop = tkP.openGraphEditor(None)
```

CHAPTER

FOUR

DESKTOP

class tkPlotXY.**Desktop**

The Desktop deals with the data management and the graph plotting.

4.1 Data management

The data can be loaded by a Desktop object from a pyTree or from a dictionnary. Only the 1D-array data from a pyTree will be loaded while the data loaded from a dictionnary has to be compliant with the following structure : {Base/Zone (string) : {Variable name (string): data (array)}}

Several methods are available to set, update or even remove data :

Desktop.**addZone**(*data*, *zoneName*, *baseName='.*'*)
: Add a specific zone to the set of data. If pyTree format is used as input, then 'basename' can be specified to add only the zone from a specific base. If 'basename' is not specified in case of pyTree format, then the specified zone for all the bases from the pyTree will be added.

Desktop.**setData**(*data*)
: Set all the data from the input pyTree or dictionnary to the Desktop data manager.

Desktop.**replaceZone**(*data*, *oldZoneName*, *newZoneName*, *oldBaseName=''*, *newBaseName=''*)
: Allows you to replace a given zone by a new one. If some data from the old zone are plotted, then the plot remains but the data are updated with the data loaded from the new zone. According to the method addZone, old basename and new basename can be given to specify the targeted base.

Desktop.**deleteZoneFromData**(*zoneName*, *oldBaseName=''*)
: Simply delete data from a given zone and base to the set of data from the Desktop object.

```
import numpy as np
import tkPlotXY as tkP
# Create a graphEditor
graphDesktop = tkP.openGraphEditor(None)
# Generate data
t = np.arange(0., 5., 0.002)
dataFromDict = {'Zone1':
                        {
                            'Iteration':t,
                            'Residual':np.sin(t),
                            'Cf':np.sin(t/2),
                            'Debit':t*t
                        }
                }
# Set data
graphDesktop.setData(dataFromDict)
# Display data
for zone in graphDesktop.data.keys():
    for var in graphDesktop.data[zone].keys():
        print zone, ' : ', var, ' : ',graphDesktop.data[zone][var]
```

```
import tkPlotXY as tkP
# Create a graphEditor
graphDesktop = tkP.openGraphEditor(None)
# Generate data with a Lamb vortex (pyTree) -
import Generator.PyTree as G
import Initiator.PyTree as I
import Post.PyTree as P
import Converter.PyTree as C

NI = 100; NJ = 100
HI = 50./(NI-1); HJ = 50./(NJ-1)
tree = G.cart((0.,0.,0.), (HI,HJ,1.), (NI,NJ,2))
tree = I.initLamb(tree, position=(7.,7.), Gamma=2., MInf=0.8, loc='centers')
tree = P.isoSurfMC(tree, 'CoordinateZ', 0.5)
tree = P.isoSurfMC(tree, 'CoordinateY', 0.7)

# Save generated data as cgns
C.convertPyTree2File(tree,'vortex_slice.hdf')
# Load data as pyTree
tree=C.convertFile2PyTree('./vortex_slice.hdf')

# Set data
graphDesktop.setData(tree)
# Display data
for zone in graphDesktop.data.keys():
```

(continues on next page)

(continued from previous page)

```
    for var in graphDesktop.data[zone].keys():
        print zone, ' : ', var, ' : ',graphDesktop.data[zone][var]
```

4.2 Graph creation

Once data have been loaded into the Desktop, you can create as many graphs as you need with the Desktop object. After that, all the drawing will be driven by the graph object itself. This is performed by the method :

Desktop.**createGraph**(*name*, *conf*, *dpi=None*, *figsize=None*)

Create a window where the plots will be drawn. A matricial description is used to define this window. For instance, here are described some settings for 'conf' variable:

- '1:1' : a single plot in this graph window
- '2:2' : 4 plots (2 rows and 2 columns)
- '2:1' : 2 plots (2 rows and 1 column)
- '1:2' : 2 plots (1 row and 2 columns)

figsize and dpi can configured to adapt the size and the resolution of the graph window if needed. You can for instance use figsize=(12,3) to enlarge your image.

```
# Create First Graph
graph_0 = graphDesktop.createGraph('MyFirstGraph','1:1')
# Create Second Graph
graph_1 = graphDesktop.createGraph('MySecondGraph','2:1')
```

CHAPTER

FIVE

GRAPH

class tkPlotXY.**Graph**(*parent*, *name*, *conf*, *dpi=None*, *figsize=None*)

An object of class Graph corresponds to a window where plots are drawn. A graph window can manage several plots.

Creating a Graph object will automatically generate an Axis, a Grid and a Legend objects for each plots on the graph. Only curves have to be created and then attached to a given graph.

Then each object can be configured. To do so, it is mandatory to access these objects thanks to the graph.

All of these actions are described in the concerned item section (Curve, Axis, Grid or Legend).

5.1 Configure the Graph object

Some times, using a matricial Graph (for instance '2:2') will provide you an unacceptable drawing. Indeed, the axis label of a plot may be overlapped by the plot below. For all these reasons, you may be interested in advanced configuration for your Graph object such as positionning, padding, margin ...

Two possibilities are available: TightLayout or SubPlotParams.

In order to improve your drawing using SubPlotParams, please use the method:

Graph.**updateSubPlotParams**(*params*)

where *params* is a dictionnary such that:

```
params = {'left':...,'right':...,'top':...,'bottom':...,'hspace':...,'wspace':...,
↪'isActive':True}
```

For example, let us create a Graph object *graph_3* with 4 plots inside ('2:2') and let us try to improve the poisitionnment of this graph with SubPlotParams:

```
# Graph creation
graph_3 = graphDesktop.createGraph('MyThirdGraph','2:2')
# Improving drawing of the Graph thanks to SubPlotParams
graph_3.updateSubPlotParams({'isActive':True,'right':0.97,'top':0.97,'wspace':0.3})
```

The other way to improve this kind of drawing is to use TightLayout:

Graph.**updateTightLayout**(*params*)

where:

```
params = {'isActive':True, 'pad':..., 'hpad':..., 'wpad':...}
```

For example, let us create a Graph object *graph_3* with 4 plots inside ('2:2') and let us try to improve the poisitionnment of this graph with TightLayout.

```
# Graph creation
graph_3 = graphDesktop.createGraph('MyThirdGraph','2:2')
# Improving drawing of the Graph thanks to TightLayout
graph_3.updateTightLayout({'isActive':True,'pad':1.1,'hpad':0.1,'wpad':0.1})
```

CHAPTER

SIX

CURVE

class tkPlotXY.**Curve**(**args*, ***kwargs*)

Curve class describes all the settings concerning a given curve itself.

6.1 Creating a curve

To create a curve, one has just to create an object of class Curve. All the settings, listed in the section 'Editing a curve', can be already configured during the creation of the object. For instance:

```
curve_0 = Curve(zone=['Base/cart'],varx='CoordinateX',vary='Density_FlowSolution' ,␣
↪line_color='#7f00ff' , marker_face_color='#7f00ff' ,marker_edge_color='#7f00ff' )
```

6.2 Editing a curve

To edit a curve, you can use the method :

Curve.**setValue**(*variable*, *value*)

with *variable* according to the following tab:

Table 1: **Available variables to set a curve**

Variable	Allowed values	Description
zone	*List of zones*	*List of zones that should be plotted*
varx	*Variable name (string)*	*X-coordinate variable name (LaTeX available with $$)*
vary	*Variable name (string)*	*Y-coordinate variable name (LaTeX available with $$)*
line_color	*Html color code (string)*	*Color used to plot the line*
line_style	*'solid', 'dashed', 'dashdot', 'dotted', 'None'*	*Style of line to use*
line_width	*(float)*	*Width of the line*
marker_style	*'none', 'plus', 'star', 'pixel', 'point', 'star3_down', 'star3_up', 'star3_left', 'star3_right', 'triangle_left', 'triangle_right', 'diamond', 'hexagon2', 'triangle_up', 'hline', 'thin_diamond', 'hexagon1', 'circle', 'pentagon', 'square', 'triangle_down', 'x'*	*Type of marker to use*
marker_size	*(float)*	*Size of the marker*
marker_edge_color	*Html color code (string)*	*Color of the edge of the marker*
marker_edge_width	*(float)*	*Width of the edge of the marker*
marker_face_color	*Html color code (string)*	*Color of the face marker*
marker_sampling_start	*(int)*	*Index on data to start plotting markers*
marker_sampling_end	*(int)*	*Index on data to stop plotting markers*
marker_sampling_step	*(int)*	*Step between index to plot markers*
legend_label	*(string)*	*Name of the curve to display in the legend*
legend_display	*(bool)*	*Display the current curve in the legend*
visible	*(bool)*	*Hide or show the curve*
axis	*(int)*	*Axis in which the curve has to be plotted*

For instance to edit a curve to a dashed curve:

```
curve_0.setValue('line_style','dashed')
```

A curve can be edited all the time. The graph has just to be updated after the modification of the curve property.

6.3 Adding a curve to a given plot on a given graph

To attach a curve to a given plot inside a given graph, use the method:

`Graph.`**`addCurve`**(*iCurSubGraph*, *curve*)

where *iCurSubGraph* identifies the plot inside the graph thanks to its matricial position. For instance, to add a curve on the second line and first column of the graph *graph_1*:

```
graph_1.addCurve('2:1',curve_0)
```

CHAPTER

SEVEN

AXIS

class tkPlotXY.**Axis**(**args*, ***kwargs*)

An Axis object contains the X-DirAxis and the Y-DirAxis of a given plot inside a Graph object. Multiple axis are available for a single plot.

class tkPlotXY.**DirAxis**(*axis_logscale*, *axis_autoscale*, *axis_min*, *axis_max*, *axis_label*, *axis_inverted*, *axis_visible*, *axis_position*, *axis_offset*, *axis_label_fontsize*, *axis_label_format*)

DirAxis contains the settings of the X or Y axis. This settings can directly be accessed from the class Axis.

While a Graph object is created, an axis system (X and Y DirAxis) is generated for each plot on the graph. This system of X and Y DirAxis is an Axis object.

7.1 Access the Axis system

To get the Axis system of given plot on a given graph, use the method on your graph:

Graph.**getAxis**(*iCurSubGraph*, *ind=0*)

where *iCurSubGraph* identifies the plot inside the graph thanks to its matricial position. Moreover, in case of a multiple axis plot (2 or more Y DirAxis on the same plot for instance, see *Multiple axis system section*), you can specify the number identifying your axis system using *ind*. Note that the original axis system has the index 0 and then the index is increased for each new axis system.

```
axis_2 = graph_1.getAxis('2:1',ind=0) # Equivalent to axis_2 = graph_1.getAxis('2:1')
```

7.2 Multiple axis system

On the same plot, you can use multiple axis system. You can decide to twin your X or Y DirAxis or even to create a new independant axis system. to generate your new axis system

(twin or independant), use the method:

`Graph.`**`addAxis`**(*iCurSubGraph*, *shared=None*, *ind=0*, *axis=None*)

where *iCurSubGraph* identifies the plot inside the graph thanks to its matricial position, *shared* can take the value : 'x','X','y','Y' or None. If None is used, then an independant axis system will be created. If an other value is used, then *axis* allows you to specify the index of the axis system you want to clone. Remember, this index starts at 0 for each plot and is then locally increased for each new axis system in a plot. Or you can directly give the object Axis you want to clone by using the parameter 'axis'. This function returns the newly created axis object.

```
# Get axis_2
ind_axis_2 = axis_2.getInd() # returns the index of the current axis
axis_2 = graph_1.getAxis('2:1', ind_axis_2) # Equivalent to axis_2 = graph_1.getAxis(
↪'2:1') because here ind_axis_2 = 0 !

# Twin X DirAxis of axis_2
axis_3 = graph_1.addAxis('2:1',shared='x',axis=axis_2) # equivalent to "axis_3 =␣
↪graph_1.addAxis('2:1',shared='x',ind=ind_axis_2)"
ind_axis_3 = axis_3.getInd()
```

7.3 Changing the Axis of a curve

Once a curve has been added to a given plot on a given graph and that this plot is composed of several axis, then it is possible to change the axis where the curve will be drawn into the given plot. To do so, you just need to edit the attribute *axis* of your Curve object. You can either use the axis object itself or its index.

```
curve_3.setValue('axis',axis_3)
# equivalent to
curve_3.setValue('ind_axis',ind_axis_3)
```

7.4 Editing the Axis system

You can edit it by accessing to the Axis object and using the method:

`Axis.`**`setValue`**(*axis*, *variable*, *value*)

where *axis* has to be 'x' or 'y'.

Or you can directly access the X (resp. Y) DirAxis object by using the attribute *x* (resp. *y*) of the class Axis that will return the X DirAxis (resp. Y DirAxis) and then you can use the method:

`DirAxis.`**`setValue`**(*variable*, *value*)

with *variable* according to the following tab:

Table 1: **Available variables to set a DirAxis**

Variable	Allowed values	Description
axis_logscale	*(bool)*	*Use logscale for selected DirAxis*
axis_autoscale	*(bool)*	*Use auto-scaling for selected DirAxis*
axis_min	*(float)*	*Minimum range to plot for the selected DirAxis*
axis_max	*(float)*	*Maximum range to plot for the selected DirAxis*
axis_label	*Label name (string)*	*Label for the selected DirAxis (LaTeX formula are available with $$)*
axis_inverted	*(bool)*	*Invert the orientation for the selected DirAxis*
axis_visible	*(bool)*	*Show or hide the selected DirAxis*
axis_position	*For X-DirAxis : 'top','bottom','both' and for Y-DirAxis : 'left','right','both'*	*Position where to plot the axis line, its ticks and the label for selected DirAxis*
axis_offset	*(float)*	*Introduce an offset for the axis line, its ticks and the label for selected DirAxis*
axis_label_fontsize	*(float)*	*Set the size of the label font for the selected DirAxis*

For instance, setting logscale on the Y-DirAxis of the *axis_3* previously defined as a twin X-DirAxis of *axis_2* (See *Multiple axis system* section)

```
axis_3.y.setValue('axis_logscale',True) # equivalent to axis_3.setValue('y','axis_
↪inverted',True)
```

CHAPTER

EIGHT

GRID

class tkPlotXY.**Grid**(**args*, ***kwargs*)

Grid contains the main grid and the second grid. They can be configured directly by accessing to Grid or to the proper GridLevel object. In case of a multiple axis usage, then multiple Grid objects can be attached to a given plot.

class tkPlotXY.**LevelGrid**(*x_display*, *x_grid_color*, *x_grid_style*, *x_grid_width*, *x_grid_tick_number*, *x_grid_tick_size*, *y_display*, *y_grid_color*, *y_grid_style*, *y_grid_width*, *y_grid_tick_number*, *y_grid_tick_size*)

class tkPlotXY.**AxisGrid**(*display*, *grid_color*, *grid_style*, *grid_width*, *grid_tick_number*, *grid_tick_size*)

Once an axis system is created, a Grid object is attached to this new axis system. It means that for each Axis object there exists a unique associated Grid object. This Grid object is composed of two LevelGrid objects : *major* and *minor* which corresponds to the main grid and the second grid. Each LevelGrid object contains two AxisGrid : *X* and *Y*. To put it in a nutshell, a Grid object describes 4 AxisGrid objects : *major X*, *major Y*, *minor X* and *minor Y*.

Since Grid objects are automatically generated during the creation of Axis object, there is no need to create Grid object. It is just needed to be able to access it.

8.1 Access a Grid object

To access the associated Grid object to a given Axis on a given plot inside a given Graph, use the method:

Graph.**getGrid**(*iCurSubGraph*, *ind=0*, *axis=None*)

where *iCurSubGraph* identifies the plot inside the graph thanks to its matricial position. Moreover, in case of a multiple axis plot (2 or more Y DirAxis on the same plot for instance, see *Multiple axis system section*), you can specify the number identifying your axis system

using *ind* or directly specify the axis object using *axis*. Note that the original axis system has the index 0 and then the index is increased for each new axis system.

For instance, to get the Grid associated to the Axis *axis_3* previously defined:

```
grid_3 = graph_1.getGrid('2:1',ind=1)
# is equivalent to
grid_3 = graph_1.getGrid('2:1',axis=axis_3)
```

If needed, you can access directly the LevelGrid object using the attributes *minor* and *major* of class Grid and then you can get the *AxisGrid* using the attributes *x* and *y* of class LevelGrid.

```
grid_3 = graph_1.getGrid('2:1',axis=axis_3)
grid_3_majorX = grid_3.major.x
grid_3_majorY = grid_3.major.y
grid_3_minorX = grid_3.minor.x
grid_3_minorY = grid_3.minor.y
```

8.2 Editing a Grid object

You can edit a Grid object using the method:

`Grid.`**`setValue`**(*level*, *direction*, *variable*, *value*)

where *level* ('major' or 'minor' expected) and *direction* ('x' or 'y' expected) identify the AxisGrid to edit

or the LevelGrid object using the method:

`LevelGrid.`**`setValue`**(*direction*, *variable*, *value*)

where *direction* ('x' or 'y' expected) identifies the AxisGrid to edit

or directly the AxisGrid object using the method:

`AxisGrid.`**`setValue`**(*variable*, *value*)

Authorized *variable* and *value* are described in the following tab:

Table 1: **Available variables to set an AxisGrid**

Variable	Allowed values	Description
display	*(bool)*	*Show or hide the AxisGrid*"
grid_color	*Html color code (string)*	*Modify the color of the AxisGrid*"
grid_style	*'solid', 'dashed', 'dashdot', 'dotted', 'None'*	*Modify the line style for the AxisGrid*"
grid_width	*(float)*	*Modify the line width for the AxisGrid*"
grid_tick_number	*(int)*	*Change the number of ticks on the AxisGrid*"
grid_tick_size	*(float)*	*Change the size of the ticks*"

For instance, to add the major grid as dashed lines for X and Y axis on the Grid grid_3 previously defined:

```
grid_3.setValue('major','x','display',True)
grid_3.setValue('major','x','grid_style','dashed')

grid_3.setValue('major','y','display',True)
grid_3.setValue('major','y','grid_style','dashed')
```

CHAPTER

NINE

LEGEND

class tkPlotXY.**Legend**(**args*, ***kwargs*)

An object of class Legend configures the legend for a given plot inside a Graph window.

While a Graph object is created, a Legend object is associated to each plot of the Graph. There is no need to create manually a Legend object. You just need to access it in order to edit it.

9.1 Accessing a Legend object

To access a Legend object of a given plot inside a given Graph, use the method:

Graph.**getLegend**(*iCurSubGraph*)

where *iCurSubGraph* identifies the plot inside the graph thanks to its matricial position.

For example, to get the Legend of the plot on the second line of graph_1:

```
legend_2 = graph_1.getLegend('2:1')
```

9.2 Editing a Legend object

To edit a Legend object, you can use the method:

Legend.**setValue**(*variable*, *value*)

Authorized *variable* and *value* are described in the following tab:

Table 1: **Available variables to set a Legend**

Variable	Allowed values	Description
legend_display	*(bool)*	*Show or hide the legend box*
legend_title	*(string)*	*Title of the legend*
legend_border_width	*(float)*	*Width of the legend box border*
legend_border_color	*Html color code (string)*	*Color of the border of the legend box*
legend_background_color	*Html color code (string)*	*Background color of the legend box*
legend_background_color_active	*(bool)*	*Use or not transparency as background for the box of the legend*
legend_position	*'best', 'upper left', 'upper center', 'upper right', 'center left', 'center', 'center right', 'lower left', 'lower center', 'lower right'*	*Position of the legend box*
legend_ncol	*(int)*	*Number of columns to display the legend*
legend_label_weight	*'normal','bold'*	*Use bold font for the curves name*
legend_label_style	*'normal','italic'*	*Use italic font for the curves name*
legend_label_size	*(float)*	*Font size for the curves name*
legend_label_color	*Html color code (string)*	*Font color used for the name of the curves*
legend_title_weight	*'normal','bold'*	*Use bold font for the legend title*
legend_title_style	*'normal','italic'*	*Use italic font for the legend title*
legend_title_size	*(float)*	*Font size for the legend title*
legend_title_color	*Html color code (string)*	*Font color used for the legend title*

For example, the following code set the title of Legend legend_2, that has been previously defined, with a bold font:

```
legend_2.setValue('legend_title_weight','bold')
```

CHAPTER

TEN

UPDATE, VIEW AND SAVE YOUR GRAPH

Now that ve have listed all elements that can be used to configure your plots, let us address the main objective of your python script: visualize your plot. First of all, you will need to update them to take into account all the modifications that have been added. Then you can either display on screen or save the figure.

10.1 Update figures

To update a given plot on a given Graph, use the method:

`Graph.`**`updateGraph`**`(`*iCurSubGraph*`)`

where *iCurSubGraph* identifies the plot inside the graph thanks to its matricial position.

For instance, in order to update the plot on second line of *graph_2*:

```
graph_2.updateGraph('2:1')
```

If you want to update all the plots of a given Graph, you can use the method:

`Graph.`**`drawFigure`**`()`

For example, to update the two plots on our *graph_2*:

```
graph_2.drawFigure()
```

Please, do not be confused, the method **drawFigure** does not display the figure !

10.2 Display on screen

In order to display on screen a Graph, use the method:

`Graph.`**`showFigure`**`()`

Do not forget to add a call to **time.sleep()** after this command in order to let the display active more longer than just a pop-up !

```
import time
####
#...
####
graph_2.showFigure()
time.sleep(5.)  # stop for 5 sec. here
```

10.3 Save figures

Use the method **save** of the class Graph to save your drawing in a file:

Graph.**save**(*path*, *format=None*)

where *path* is the output file path and format is the format you wish to use to save your figure (available formats are : 'emf', 'eps', 'pdf', 'png', 'ps', 'raw', 'rgba', 'svg', 'svgz').

```
graph_2.save('/home/User/Example/myNiceGraph.png')
```

CHAPTER

ELEVEN

EXTRA USAGES

11.1 Load & Save configurations script

Configurations scripts are python scripts automatically generated by the GUI. They can be loaded to ease the proccess of tunning your plot. For instance, if you have a plot that you often draw, instead of re-creating each time your drawing, you can simply load your configuration and use it as a base. Moreover, this configurations scripts are created by the graphical user interface and you will only need to adapt a few elements (removing the 'TK' suffix of all the classes basically) to use them directly as script without the GUI.

CHAPTER

TWELVE

COMPLETE EXAMPLE

```
import tkPlotXY as tkP
# Create a graphEditor
graphDesktop = tkP.openGraphEditor(None)
# Generate data with a Lamb vortex (pyTree) -
import Generator.PyTree as G
import Initiator.PyTree as I
import Post.PyTree as P
import Converter.PyTree as C
import time,os
cwd = os.getcwd()

DEBUG_CHECKDATA = True

# Creating a vortex
NI = 100; NJ = 100
HI = 50./(NI-1); HJ = 50./(NJ-1)
tree = G.cart((0.,0.,0.), (HI,HJ,1.), (NI,NJ,2))
tree = I.initLamb(tree, position=(7.,7.), Gamma=2., MInf=0.8, loc='centers')
tree = P.isoSurfMC(tree, 'CoordinateZ', 0.5)
tree = P.isoSurfMC(tree, 'CoordinateY', 0.7)

# Save generated data as cgns
C.convertPyTree2File(tree,'vortex_slice.hdf')
# Load data as pyTree
tree=C.convertFile2PyTree('./vortex_slice.hdf')

############################ Set data
graphDesktop.setData(tree)
if DEBUG_CHECKDATA:
    for z in graphDesktop.data.keys():
        print '*-'*15
        print 'Zone : ',z
        for k in graphDesktop.data[z].keys():
            print '---> Var : ',k
```

(continues on next page)

(continued from previous page)

```
############################ Graph creation
# Create First Graph
graph_0 = graphDesktop.createGraph('MyFirstGraph','1:1')
# Create Second Graph
graph_1 = graphDesktop.createGraph('MySecondGraph','2:1')

############################ Curve creation
# Create the first curve
curve_0 = tkP.Curve(zone=['Base/cart'],varx='CoordinateX',vary='Density@FlowSolution
↪' , line_color='#7f00ff' , marker_face_color='#7f00ff' ,marker_edge_color='#7f00ff
↪' )

# Create the second curve
curve_1 = tkP.Curve(zone=['Base/cart'],varx='CoordinateX',vary=
↪'MomentumZ@FlowSolution' , line_color='#0404B4' , marker_face_color='#0404B4' ,
↪marker_edge_color='#0404B4' )

# Create the third curve
curve_2 = tkP.Curve(zone=['Base/cart'],varx='CoordinateX',vary=
↪'MomentumX@FlowSolution' , line_color='#FF00FF' , marker_face_color='#FF00FF' ,
↪marker_edge_color='#FF00FF' )

# Create the fourth curve
curve_3 = tkP.Curve(zone=['Base/cart'],varx='CoordinateX',vary=
↪'MomentumY@FlowSolution' , line_color='#FFBF00' , marker_face_color='#FFBF00' ,
↪marker_edge_color='#FFBF00' )

############################ Attaching curves to graph
# First curve to graph_0
graph_0.addCurve('1:1',curve_0)
# Second curve to graph_1 first line
graph_1.addCurve('1:1',curve_1)
# Third curve to graph_1 second line
graph_1.addCurve('2:1',curve_2)
# Fourth curve to graph_1 second line
graph_1.addCurve('2:1',curve_3)

############################ Editing curves
## Name for the legend
curve_0.setValue('legend_label','Density')
curve_1.setValue('legend_label','MomentumZ')
curve_2.setValue('legend_label','MomentumX')
curve_3.setValue('legend_label','MomentumY')
```

(continues on next page)

(continued from previous page)

```
## curve_3 : dashed
curve_3.setValue('line_style','dashed')

## curve_2 : add markers
curve_2.setValue('marker_style','circle')
curve_2.setValue('marker_sampling_step',20) # 1 marker over 20

############################ Axis properties
## 1/- Get axis
axis_0 = graph_0.getAxis('1:1') # ind = 0
ind_axis_0 = axis_0.getInd()
# #
axis_1 = graph_1.getAxis('1:1') # ind = 0
ind_axis_1 = axis_1.getInd()
#
axis_2 = graph_1.getAxis('2:1') # ind = 0
ind_axis_2 = axis_2.getInd()
## 2/- Twining X axis for plot 2:1 on graph_1
axis_3 = graph_1.addAxis('2:1',shared='x',axis=axis_2) # equivalent to "axis_3 =␣
↪graph_1.addAxis('2:1',shared='x',ind=ind_axis_2)""
ind_axis_3 = axis_3.getInd()

# Set the position of axis label
axis_2.setValue('y','axis_position','left')
axis_3.setValue('y','axis_position','right')
# Change the label text
axis_1.setValue('y','axis_label','$\\rho W$')
axis_2.setValue('y','axis_label','$\\rho U$')
axis_3.setValue('y','axis_label','$\\rho V$')

# ############################ Changing axis for curve_3
curve_3.setValue('axis',axis_3) # equivalent to "curve_3.setValue('ind_axis',ind_
↪axis_3)"

############################ Editing Grid properties
# Get the grid objects
grid_0 = graph_0.getGrid('1:1',ind=ind_axis_0)   # equivalent to "grid_0 = graph_0.
↪getGrid('1:1',axis=axis_0)"
grid_1 = graph_1.getGrid('1:1',ind=ind_axis_1)   # equivalent to "grid_1 = graph_1.
↪getGrid('1:1',axis=axis_1)"
grid_2 = graph_1.getGrid('2:1',axis=axis_2)   # equivalent to "grid_2 = graph_1.
↪getGrid('2:1',ind=ind_axis_2)"
grid_3 = graph_1.getGrid('2:1',axis=axis_3)   # equivalent to "grid_3 = graph_1.
↪getGrid('2:1',ind=ind_axis_3)"
# Display a solid grid for the major grids on X & Y axis_2
```

(continues on next page)

(continued from previous page)

```
grid_2.major.x.setValue('grid_style','solid')
grid_2.major.y.setValue('grid_style','solid')

########################### Editing Legend properties
# Get the legend objects
legend_0 = graph_0.getLegend('1:1')
legend_1 = graph_1.getLegend('1:1')
legend_2 = graph_1.getLegend('2:1')
# Hide legend_1
legend_1.setValue('legend_display',False)
# Reduce legend label size
legend_2.setValue('legend_label_size',8)
# Increase legend title size and set its font as bold
legend_2.setValue('legend_title_size',10)
legend_2.setValue('legend_title_weight','bold')
# Position legend in the lower right corner
legend_2.setValue('legend_position','lower right')

########################### Use SubPlotParams
params = {'left':0.12,'right':0.87,'top':0.90,'bottom':0.12,'isActive':True,'hspace
↪':0.3}
graph_1.updateSubPlotParams(params)

########################### Update, view & save
# Update
graph_1.drawFigure()
# Display
graph_1.showFigure()
# Wait
time.sleep(5.)
# Save
graph_1.save(os.path.join(cwd,'MyNiceFigure.png'))
```

CHAPTER

THIRTEEN

INDEX

- genindex
- modindex
- search