xfbp Documentation

Release 22.00-62

Klaus Koepernik

Jul 06, 2022

CONTENTS

CHAPTER ONE

START HERE

Non-python related help topics start *here*. These are the old help screens including the *native scripting*. Python scripting is explained *here*.

CHAPTER

TWO

XFBP PYTHON BINDINGS

Author Klaus Koepernik

2.1 General

The python binding of **xfbp** is hard coded into the program. There is no source code you can look at. Because of this tight connection *pyxfbp* can only be used from within **xfbp**. For convenience we have decided to setup the python environment in the following way:

The pyxfbp module is already imported as:

```
from pyxfbp import *
```

which injects all classes into the namespace. We also created an instance of the class `pyxfbp.Xfbp` called *xfbp*. From this some methods/properties are loaded into the namespace

- `G` represents all Graphs
- `killall` clean slate
- `printto` guess what
- `paper` setup paper dimensions
- `arrange` arranges graphs
- `setMouseHook` set a mini-script to be executed on left mouse click
- `cursor` the cursor position on mouse click

All other properties/classes can be accessed hierarchically from `G`. Many classes can be used standalone, whatever is more convenient. E.g.:

```
Title(1).text='some text'
#and
G[1].title.text='some text '
```

both change the text of the graph with id 1. However, indexing only works via the corresponding properties as in:

```
# copy set 1 into set 1
G[1].Gr[1].S[2]=G[1].Gr[1].S[1]
```

For a better understanding of the indexing read the doc for `Graphs`. Some simple classes like `FontStyle` behave differently in that they either represent the fontstyle of some object or can be created for assignment reasons:

```
# Here font returns a FontStyle object representing
# the title's font
Title(1).font.color=0xff

```

(continues on next page)

(continued from previous page)

```
# Here we create a free stranding FontStyle
fst=FontStyle(color=0xff)

# and assign it to all textboxes (assuming some exist)
for t in G[1].textboxes:
    t.font=fst
```

In the first case *.font* returns a FontStyle object representing the titles font and in the second the textboxes font is hard copied from *fst*.

Note, that many data are implemented as properties and some as functions/methods. Properties can be gotten and set, functions must be called:

```
G[1].title.text='some text' # set the property text

ti=G[1].title.text # get the property text

G[1].title.off() # call the method `off`.
```

2.2 Examples

There are examples of `.xpy` scripts scattered through out the **pyfplo** examples in `FPLO.../DOC/pyfplo/Examples` and in `FPLO.../DOC/Xfbp/Examples`.

2.3 Editor

The code editor has two modes, the native xfbp script mode with (file extension `.cmd`) which is still available but not as versatile and the python mode (file extension `.xpy` or `.py`). The `.xpy` extension is usefull for easy file association e.g. in midnight-commander or desktop tools. It also signals that these python scripts need xfbp to run. They cannot run standalone (yet).

A new script will have a file type marker at the beginning. This is only ment to tell you , which mode you are in. It is not used for identification. The file extensions decide how the script is parsed.

Python uses indentation to mark blocks. The editor supports python style, where you use the tab key to indent by 4 spaces. You can select a bunch of lines and indent or unindent them via editing menu commands. If for some reasons tab-characters find their way into the script they are highlighted to find them easier. You should use block unindent and indent to convert them into space. Otherwise, python might consider wrong indentation, which can change the meaning of the code. It might run but does not do what you expect.

Note, that if you copy text from some other source (help) the indentation might be wrong. If this happens select the block, indented/unindent it with the edit commands as needed.

There is quite some code insertion available. The fastest is to adjust you textboxes, shapes and world and view and then to insert the corresponding code into the script via the editors insert menu. Code insertion often uses direct object access in contrast to hierarchical access for brevity. You will get the hang of it. **After insertion select the whole block and use the edit->indent functionality for proper python indentation, if needed.**

2.4 Help

There are two versions of inbuilt help. The first is called when a python help command is written into the editor and the script is executed while the editor is open. This will either display the content from the python help system if the item does not belong to `pyxfbp` or display the `pyxfbp` help. The other version is just pressing *F1* in the editor, which displays `pyxfbp` help. The help browser has a search function (*Ctrl-F*). Type the search string. Hit

Ctrl-F or *ENTER* to continue searching. If the search reaches the end of the file hit *Ctrl-F*/*ENTER* again to start from the beginning. Hit *ESCAPE* to leave the search. Hit *Ctrl-F* twice to redo the previous search.

How to use it: if you don't know what comes next do the following:

```
g=G[1] # g represents graph with id 1
help(g)
```

will tell you that g is a `Graph` instance and from there you can pick a property, say *Gr* and write instead:

```
help(g.Gr[1])
```

Yet another way is to type in help for a class member:

```
help(Graph.active)    # help on property active in class Graph
```

which is not the same as:

```
help(G[1].active)     # G[1] is a Graph instance and G[1].active an int
                      # so you get help for the int class of python
```

You get the idea. If you know your way around just hit *F1* and use the search function.

If you want help on python keywords like `for` type:

```
help('for') # yes as string!
```

On last note … the functions which where loaded into the namespace from *xfbp* will be treated like python internal help. If you type:

```
help(Xfbp.killall)
```

you get the expected result.

CHAPTER

THREE

COMMMAND LINE PARAMETERS

In order to write generic scripts, command line parameters are implemented the following way. On the command line you give something like:

```
xfbp -a filename:+band -a col:0xff -a unit:0.529177 -a i:3
```

where the part before the colon is the variable name and the part after the colon is the value. **There shoudn't be a space before and after the colon!** You can us quotes for longer `str` values. If the `str` value contains $-formating use single quotes! The program will determine the easiest type of all values by trying to cast it in the following order: hex(`int`), `int`, `float` and `str`. If any of the casts succeeds the parameter in python will have this type, e.g. `-a i:42` will be an `int`. If inside the script you want it to be a `str` use `str(i)`. Please note, that these parameters become normal variables in the script. You should be carefull with the names. Simple example script called t.py:

```
killall()
G[1].read("band",filename)
G[1].Gr[1].line.color=col
G[1].title.text=title
G[1].yaxislabel.text='Energy [{}]'.format(sunit)
for s in G[1].Gr[1].S:
    s.y/=unit
G[1].world.ymin/=unit
G[1].world.ymax/=unit
```

is called e.g. like this:

```
xfbp t.py -a filename:+band -a col:0xff -a title:'This works: FeO$_2$.' \
   -a unit:13.60569193 -a sunit:Ry
```

More command line parameters are found under *Command line options*.

CHAPTER

FOUR

TEXT FORMATING

The various properties which can hold text can be formated in the following way.

- **`$~`** next character is from the symbol font (greek) (hopefully works on your system)
- **`$i`** switch to italic font
- **`$n`** switch to regular font
- **`$_`** switch to next subscript level
- **`$^`** switch to next superscript level
- **`$.`** switch to normal level
- **`$x{real-number}`** shift the current position (for the following characters) to the right (positive number) or left (negative number). Note that the shift scale is different before or after a sub/super script marker (`$^`, `$_`). Try:

  ```
  X$_ij$.$x{-0.5}$^ab$.
  ```

  or:

  ```
  X$_a$_i$.$x{-0.8}$^$~m$.
  ```

  versus:

  ```
  X$_ij$.$^ab$.
  ```

  or:

  ```
  X$_a$_i$.$^$~m$.
  ```

- **`$y{real-number}`** vertical shift (see `$x{}` above)
- **`$arrowup`, `$arrowdown`, `$arrowleft`, `$arrowright`, `$angstroem`, `$infinity`, `$nabla`** some special characters (hopefully works)
- **`$$`** a $ sign

Note, that some special characters (e.g. `$arrowup`) switch the font to regular after they were printed (bug). To continue with italic, use another `$i`. On some systems the display of symbol characters is wrong.

CHAPTER

FIVE

COLORS

There are several classes which have colors. A color is given as a hex(int) constant. A color is coded as rgb (red/green/blue) value. The format is as follows: There are three bytes (value 0..255) in a color the first byte represents the red value, the second green and the third blue. Using hex notation the bytes are forming two digit hex numbers. Hence a color contains 6 hex digits. The brightest colors are the highest value (ff). If red is zero it does not have to be written explicitly. A hex constant starts with `0x` The bytes are in the order left to right: r g b. If a color is returned by *pyxfbp* it is returend as `int` (base 10) .To print it in hex form use `print hex(c)`. Examples:

hex color	red	green	blue	color
`0xff00aa`	`ff/255`	`0`	`aa/170`	pink
`0xffffff`	`ff/255`	`ff/255`	`ff/255`	white
`0xff`	`0`	`0`	`ff/255`	blue
`0xff0000`	`ff/255`	`0`	`0`	red
`0xff00`	`0`	`ff/255`	`0`	green
`0xaa00`	`0`	`aa/170`	`0`	darker green
`0x0`	`0`	`0`	`0`	black

Usefull defaults

hex	amount (1==100%)
00	0
33	0.2
40	1/4
55	1/3
66	0.4
80	1/2
99	0.6
aa	2/3
c0	3/4
cc	0.8
ff	1

CHAPTER

SIX

MODULES

6.1 pyxfbp

This is the one and only module to manipulate the data of xfbp.

6.1.1 Xfbp

class Xfbp

This is the main class of `pyxfbp`. It provides access to a few basic functions and to all `Graphs` via the property `G`. An instance of this class called *xfbp* and all members of *xfbp* are loaded into the namespace when the editor is activated (see *Sec General*). Note, that `G` should not be overwritten by doing something like:

```
G="Helloworld" # now G is no longer a Graphs object but xfbp.G still is
```

killall()

remove everything and reset to initial state (a single empty graph)

printto(*filename*, *dpi=300*, *quality=0.85*)

print to file called *filename* with dot-per-inch set to *dpi* and with *quality*. Two file types are available: `eps` and `png`. *dpi* and *quality* only apply to `png`-files.

Parameters

- **filename** (*str*) – a filename including extension: `png` or `eps`
- **dpi** (*int*) – dots per inch
- **quality** (*float*) – compression quality in [0,1]

Returns

self to allow call chaining as in

```
Xfbp.printto(...).setSomethingElse(...)
```

Return type `Xfbp`

arrange(*paperwidth=800.0*, *Nx=1*, *leftgap=0.1*, *rightgap=0.1*, *hgap=0.1*, *Ny=1*, *topgap=0.1*, *bottomgap=0.1*, *vgap=0.1*, *aspectratio=1.5*, *commonxaxis=True*, *commonyaxis=True*, *commontitle=True*)

Arrange the first *Nx* * *Ny* graphs in a grid. The graphs can already exist. Otherwise they are created. The graph ids run row by row. *leftgap/rightgap* are the spaces left/right of the first/last viewbox (`View`) in percent*100 of the *paperwidth*. *topgap/bottomgap* work similar. *hgap/vgap* are the in-between gaps in percent*100 of the individual viewbox width/height. *aspectratio* determines the individual viewbox's aspect ratio. The paperheight depends on all these settings.

Note, that the point scale of the graphs depends on the *paperwidth*. You need to experiment with the gap values a bit to get the labels/ticmarks/titles properly displayed in the page. If *commonxaxis* is `True` the in-between-viewboxes tic labels and xaxis labels are switched off and in each column the world x axis is made equivalent. Similarily, for *commonyaxis*. For this to work the arrange command must be issued after setting up the graphs. If *commontitle* is `True` the titles in between rows are switched off. Example:

```
killall()
arrange(800,2,0.12,0.05,0.1,   2,0.1,0.12,0.1,   1, 1,1,1)
```

Parameters

- **paperwidth** (*float*) – in pt (A4 is 595 x 842)
- **Nx** (*int*) – number of columns
- **leftgap** (*float*) – in percent*100 of paperwidth
- **rightgap** (*float*) – in percent*100 of paperwidth
- **hgap** (*float*) – in percent*100 of viewbox width
- **Ny** (*int*) – number of rows
- **topgap** (*float*) – in percent*100 of paperheight
- **bottomgap** (*float*) – in percent*100 of paperheight
- **vgap** (*float*) – in percent*100 of viewbox height
- **aspectratio** (*float*) –
- **commonxaxis** (*int*) – 0 or 1 (`False` or `True`)
- **commonyaxis** (*int*) – 0 or 1 (`False` or `True`)
- **commontitle** (*int*) – 0 or 1 (`False` or `True`)

Returns

self to allow call chaining as in

`Xfbp.arrange(...).setSomethingElse(...)`

Return type *Xfbp*

setMouseHook (*button*, *script*)

setup a miniscript to be executed at mouse click of *button* Currently *button* is always 'left'. *script* should contain valid python code and it must be valid in the context it is exceuted in (objects refered to must exist) (see *Xfbp.cursor*)

Parameters

- **button** (*str*) – on which mouse button click to install the hook currently only 'left' works
- **script** (*str*) – a valid python mini script

Returns

self to allow call chaining as in

`Xfbp.setMouseHook(...).setSomethingElse(...)`

Return type *Xfbp*

G
return a Graphs object, which provides access to all graphs. The following two are nearly equivalent: `G[1]` and `Graph(1)` The former allows deletion, E.g. `del G[1]` will delete the graph with id 1. Use it as in:

```
G[1].world.x=(-2,3)
```

Type *Graphs*

paper
return a Paper object to set its dimensions.

Type *Paper*

cursor
return current cursor position as a tuple (x,y). This is usefull inside a mouse click hook. Note, that the graph must be the current graph to get expected results:

```
setMouseHook('left','''
G[1].title.toggle()
c=cursor
gr=G[1].Gr[1].on()
s=gr.S[4].on()
s.x=[c[0],c[0]]
s.y=[G[1].world.ymin,G[1].world.ymax]
print s.x
print s.y
''')
```

Type 2-tuple

6.1.2 Graphs

class Graphs
Graphs represents all *Graph* s. *Xfbp* returns an instance of *Graphs* via the property *G*. *G* is also loaded into the namespace such that we can just use it without the *Xfbp* object. It behaves a bit like a python `dict` with `int` keys and a bit like a `list`.

`G[id]` represents a reference to graph with *id*, it does not mean that this graph exists. On usage of `G[i]` errors are raised when the graph was not yet created. A graph can be created by using a *read* command (`G[1].read(...)`) or by switching its visibility as in `G[i].active=1` or `G[i].on()`. Generally, *on*, *off*, *toggle* and the property *active* will physically create a graph.

`len(G)` is the number of graphs not the highest id. Graphs can be created in any order of ids. The order of graphs determines the plotting order.

You can do the following:

```
# this iterates over all EXISTING graphs
for g in G:
   g.title.off()

G[4].active=1 # switch on graph number 4


g=G[6] # g represents a reference to graph 6

# if it does not exist yet errors will be raise when you use it
# but you can write
g.active=True # now it exists and is visible
```

(continues on next page)

(continued from previous page)

```
# iterate via index
for i in range(len(G)):
    print G.at(i).id


#iterate over certain graph ids
for id in [1,4,6]:
    G[id].title.off()


# delete graph with id 4
del G[4]
```

`__delitem__`(*id*)

you can delete a graph with a certain id:

```
del G[2]
```

`__len__`()

`len(G)` returns the number of graphs not the highest id

`at`(*i*)

For index (not id) based iteration, use as in:

```
for i in range(len(G)):
    G.at(i)...
```

Parameters **i** (*int*) – graph index (not id)

Returns graph at index i

Return type *Graph*

`__getitem__`()

`G[id]` returns the graph with a certain id:

```
killall() # only graph 1 will exist after killall
G[4].on() # switch on graph with id 4
G[2].on() # switch on graph with id 2
for g in G:
   print g   #now we have graphs 1,4 and 2
```

`G[i] <==> G.__getitem__(i)`

`__iter__`()

you can iterate over all graphs:

```
for g in G:
    print g.id
```

`next`()

for the iterator interface, see *`__iter__`*

`lastid`

the highest id among all existing graphs (not the number of graphs):

```
G[G.lastid+1].on()# definitely a new graph
```

Type int

6.1.3 Graph

class Graph(*graphid*)

This class represents a single graph. This class can be used by itself to address a particular graph with id *graphid*:

```
Graph(1) # represents graph 1
```

Another way would be:

```
G[1]
```

To delete a graph there is only one way:

```
del G[1]
```

Parameters **graphid** (*int*) – the graphid as shown in the GUI

read(*type*, *filename*, *groupid=0*)

read file called *filename* of intended type *type* into the graph (and optionally into a specific group in this graph) and return a `list` of *NewGroup* object, which provide easy access to all newly created groups and sets. The return value can be ignored. If a *groupid* was given the group will be created if it does not exist.

The *type* can be:

'xny' Data sets are read, assuming that an empty line starts a new data block. In each multi column block with N columns the first column is x and the other $N-1$ columns are y_i, resulting in $N-1$ sets for each data block in the file. All sets end up in a new group.

'xynw' The first column of each block is x. The second is y and the following columns are weights.

'xynz' First comes a block of N_x x-values, each line one value, followed by a blank line. Then comes a similar block of N_y y-values followed by a blank line. Finally a block of $N_x \cdot N_y$ z-values, one value per line. The resulting plot will be a density plot where the z-values define the color.

'band' An FPLO band structure file.

'bandweight' or 'bandweights' An FPLO band weights file.

'akbl' An FPLO Bloch-Spectral-Density file. (CPA FPLO5, pyfplo.Slabify)

Examples for return value:

```
gr=G[1].read('xny','grid.dat')[0].group
# new gr is a Group object


ret=G[1].read('xny','grid.dat')
for r in ret:
    for s in r.sets:
        print 'new group',r.group.id,'new set',s.id
```

Parameters

- **type** (*str*) – file type marker: 'xny', 'band', 'bandweight', 'xynw' ,'xynz' 'akbl'
- **filename** (*str*) – the filename
- **groupid** (*int*) – optionally, read into a specific group

Returns a list of all newly created groups and sets

Return type list of *NewGroup*

on()

switch graph on

Returns

self to allow call chaining as in

`Graph.on(...).setSomethingElse(...)`

Return type *Graph*

off()

switch graph off

Returns

self to allow call chaining as in

`Graph.off(...).setSomethingElse(...)`

Return type *Graph*

toggle()

toggle graph visibility

Returns

self to allow call chaining as in

`Graph.toggle(...).setSomethingElse(...)`

Return type *Graph*

autoscale(*what='all'*)

autoscale graph using the currently set *World.offset*

Parameters **what** (*str*) – 'all' ,'x' or 'y'

Returns

self to allow call chaining as in

`Graph.autoscale(...).setSomethingElse(...)`

Return type *Graph*

id

the graph id as shown in the GUI

Type int

active

get/set if the graph is visible. A nonzero value is True zero is False.

Type bool

Gr

return a Groups object, which provides access to all groupsof this graph.

Type *Groups*

linewidthscale

get/set an overall line width scale for the graph. This affects all line widths.

Type float

pointsizescale

get/set an overall point size scale for the graph. This affects font and symbol sizes and line widths.

Type float

title
return a Title object

Type *Title*

subtitle
return a SubTitle object

Type *SubTitle*

xaxislabel
return a XAxisLabel object

Type *XAxisLabel*

yaxislabel
return a YAxisLabel object

Type *YAxisLabel*

oppositexaxislabel
return a OppositeXAxisLabel object

Type *OppositeXAxisLabel*

oppositeyaxislabel
return a OppositeYAxisLabel object

Type *OppositeYAxisLabel*

textboxes
return a TextBoxes object, which provides access to all textboxes of this graph.

Type *TextBoxes*

legend
get/set a Legend object which represents the legendbox.

Type *Legend*

world
get/set the World object of this graph.

Type *World*

view
get/set the View object of this graph.

Type *View*

xaxis
get/set the x-axis object of this graph.

Type *Axis*

yaxis
get/set the y-axis object of this graph.

Type *Axis*

xtics
return the x-ticmarks of this graph

Type *TicMarks*

ytics
return the y-ticmarks of this graph

Type *TicMarks*

usertics
return the irregular user defined tickmarks of this graph

Type *UserTics*

lines
return a Lines object, which provides access to all Line shapes of this graph.

Type *Lines*

ellipses
return an Ellipses object, which provides access to all ellipses shapes of this graph.

Type *Ellipses*

6.1.4 Groups

class Groups(*graphid*)
The Groups object represents all `Group` s. It is organized id-based like `Graphs`. The groups need not be ordered with monotonous ids. The group order determines the plotting order. IDs are just the names of the groups in the GUI. Names should not change, thats why they cannot behave like linear indices. A groups object can be used standalone or probably better be obtained via the class hierarchy:

```
G[1].Gr   # this is the Groups object of graph(id 1)
Groups(1) # and this too


G[1].Gr[2] # and this is the Group with id 2 in graph(id 1)
Group(1,2) # and this too


# lets do useful things, assuming we loaded some data
gr=G[1].Gr[1]
gr.useattributes=True
gr.line.color=0xff # all sets in group(id 1) will be blue

# We cannot straight forwardly copy groups, since there is an issue with
# the way weightlabels are handled internally.
# But we can do this:
gr=G[1].Gr[1]
G[1].Gr[2].on()
for s in gr.S:
   G[1].Gr[2].S.append(s)
# A note on weightlabels.
# weightlabels are shared among the set of a bandweights/xynw plot
# after our copy above we need to set weightlabels to a consistent state
G[1].Gr[2].unifyWeightLabels()
# this will make sure that the original group and the new group
# have different weightlabel-data and that all sets in the group
# have the same.
```

Parameters **graphid** (*int*) – the graph id

__delitem__(*id*)
you can delete a group with a certain id:

```
del G[1].Gr[2]
```

__len__()
`len(G[1].G)` returns the number of groups not the highest id

at(*i*)
For index (not id) based iteration, use as in:

```
for i in range(len(G[1].Gr)):
    G[1].Gr.at(i)...
```

Parameters **i** (*int*) – group index (not id)

Returns group at index i

Return type *Group*

`__getitem__`()

`G[1].Gr[id]` returns the group with a certain id

`G[1].Gr[i] <==> G[1].Gr.__getitem__(i)`

`__iter__`()

you can iterate over all groups:

```
for gr in G[1].Gr:
    print gr.id
```

next()

for the iterator interface, see *`__iter__`*

new()

create and return a new Group

Returns a new Group

Return type *Group*

len

return the number of groups not the highest id

Type int

lastid

the highest id among all existing groups of this graph (not the number of groups)

Type int

6.1.5 Group

class Group(*graphid*, *groupid*)

A group is a collection of *Sets*. They allow easy application of collective properties on many sets as it occurs in band structure plots. The *Graph.read* method will create *Group* s for the new *Set* s in a reasonable fashion. If needed all sets can be displayed with identical properties via the *useattributes* flag. Similarly, *Weight* properties can also be accessed From groups. You can access a group through the *Groups* property of a *Graph*:

```
G[1].Gr[2] # this is group(id 2) in graph(id 1)
Group(1,2) # is the same
```

Parameters

- **graphid** (*int*) – the graph id
- **groupid** (*int*) – the group id

on()

switch group on

Returns

self to allow call chaining as in

```
Group.on(...).setSomethingElse(...)
```

Return type *Group*

off()

switch group off

Returns

self to allow call chaining as in

```
Group.off(...).setSomethingElse(...)
```

Return type *Group*

toggle()

toggle the group's visibility

Returns

self to allow call chaining as in

```
Group.toggle(...).setSomethingElse(...)
```

Return type *Group*

setWeightsStyle(*style='dots'*, *factor=1*, *min=0.2*, *max=6*, *showinlegend=True*)

this is another interface to the weight… properties. A prototype of this method can be inserted from the editor insert menu.

Parameters

- **style** – 'dots', 'connected' or 'individual' see(*weightstyle*)
- **factor** (*float*) – see *weightfactor*
- **min** (*float*) – see *weightmin*
- **max** (*float*) – see *weightmax*
- **showinlegend** (*int*) – see *showinlegend*

Returns

self to allow call chaining as in

```
Group.setWeightsStyle(...).setSomethingElse(...)
```

Return type *Group*

unifyWeightLabels()

This function will only be needed in rare cases, since all *Graph.read* commands already call this internally. One case, in which it is needed is when a `xynw` sets are created by script.

If all sets in this group have the same number of weights we can unify the group such that the groups and all it's sets refer to the same weight labels.

Returns

self to allow call chaining as in

```
Group.unifyWeightLabels(...).setSomethingElse(...)
```

Return type *Group*

id

the group's id

Type int

graph

return the graph, the group is in

Type *Graph*

active
get/set if the group is visible. A nonzero value is True, zero is False.

Type bool

S

this gives access to all sets of the group

```
G[1].Gr[1].S[2].line.color=0xff0000
G[1].Gr[1].S[2].line.width=2
#
G[1].Gr[1].S[2].line=LineStyle(color=0xff0000,width=2,style='Dot')
```

Type *Sets*

W

this gives acces to all weights. This is usefull in conjunction with *useattributes*. See also `Sets.W`.

```
G[1].Gr[1].W[1].off()
G[1].Gr[1].W[5].on().setStyle(...)
G[1].Gr[1].W[5].color=0xff
```

Type *Weights*

useattributes
if nonzero/`True`, the group's attributes will be used for each set of the group. When switched off, each set's individual properties will be used. This flag is used e.g. when band structures are loaded.

Type bool

line
get/set the linestyle.

Type *LineStyle*

legend
a group can be displayed as a single legend entry. This is usefull in conjunction with *useattributes*.

Type str

comment
get/set the 'set-comment' of the group. Set-comments denote, where the data came from. This is mostly set by file-loading routines.

Type str

showinlegend
get/set if this group is shown in the legend box.

Type bool

symbol
get/set the symbolstyle. Not used for *Weight* s, which have their own limited settings.

Type *SymbolStyle*

weightstyle
get/set the style of the weights

'dots'	one filled circles for each data point
'connected'	connect the circles by linear intepolation
'individual'	each weight can have its own symbol

Type str

`weightfactor`
get/set the weight factor. All weights are scaled by this factor, before applying *`weightmax`* and *`weightmin`*. (This is somehow superfluous, but usefull anyway).

Type float

`weightmin`
get/set the weight symbol size, under which no symbols will be plotted. This depends on *`weightmax`* since it scales everything up. It actually also depends on other scales, so just do trial & error for the desired result.

Type float

`weightmax`
get/set the symbol size, which represents a weight value of 1. weightmax is in a scale like font sizes.

Type float

`showweightsinlegend`
get/set if weight legend entries are shown.

Type bool

`usertics`
irregular user defined tickmarks

Type *`UserTics`*

6.1.6 Sets

class `Sets`(*graphid*, *groupid*)
The Sets object gives access to all sets of a *`Group`*. It is ID bases as *`Graphs`*. The Sets class allows access to the sets, deletion and copying of sets:

```
# access to set 1
G[1].Gr[1].S[1].line.color=0xff

# copy set 1 to set 3
G[1].Gr[1].S[3]=G[1].Gr[1].S[1]

# delete set 1
del G[1].Gr[1].S[1]
```

Parameters

- **graphid** (*int*) – the graph id
- **groupid** (*int*) – the group id

`__delitem__`(*id*)
you can delete a set with a certain id:

```
del G[1].Gr[1].S[1]
```

`__len__`()
`len(G[1].Gr[1].S)` returns the number of sets not the highest id

`at`(*i*)
For index (not id) based iteration, use as in:

```
for i in range(len(G[1].Gr[1].S)):
    G[1].Gr[1].S.at(i)...
```

Parameters **i** (*int*) – set index (not id)

Returns set at index i

Return type *Set*

__getitem__()

use as in in: print x[i]

__iter__()

you can iterate over all sets:

```
for s in G[1].Gr[1].S:
    s.line.color=0xff
```

next()

for the iterator interface, see *__iter__*

new(*s=None*)

create and return a new set. If argument *s* is a *Set*, copy it:

```
killall()
gr=G[1].Gr[1]
s=gr.S[1].on()
s.x=[-2,-1,0,1,2,3,4]
s.y=s.x**2
s.line.color=0xff
# new set - copy
s=gr.S.new(s)
s.x-=1
s.line.color=0xcc00
G[1].autoscale()
```

Parameters **s** (*Set*) – an optional set to copy into the new set

Returns a new set

Return type *Set*

append(*s*)

Parameters **s** (*Set*) – an existing set

Returns

self to allow call chaining as in

`Sets.append(...).setSomethingElse(...)`

Return type *Sets*

append set *s* to the sets, the set is hard copied, such that the original and the new set are physically distinct with the exception of weightlabel-data. The copied set shares the same weightlabels-data with the source set. This means that changing the name of a weight in the newly copied set will change the name in the source set as well. See *Group.unifyWeightLabels*.

len

return the number of sets not the highest id

Type int

lastid

the highest id among all existing sets of this group (not the number of sets)

Type int

6.1.7 Set

class Set(*graphid*, *groupid*, *setid*)

The Set class represents an actual data set. It can be accessed from the *S* property of a *Group* or directly:

```
G[1].Gr[2].S[3].color=0xff
# or
Set(1,2,3).color=0xff
```

Parameters

- **graphid** (*int*) – the graph id
- **groupid** (*int*) – the group id
- **setid** (*int*) – set set id

on()

switch set on

Returns

self to allow call chaining as in

`Set.on(...).setSomethingElse(...)`

Return type *Set*

off()

switch set off

Returns

self to allow call chaining as in

`Set.off(...).setSomethingElse(...)`

Return type *Set*

toggle()

toggle the set's visibility

Returns

self to allow call chaining as in

`Set.toggle(...).setSomethingElse(...)`

Return type *Set*

setWeightsStyle(*style='dots'*, *factor=1*, *min=0.2*, *max=6*, *showinlegend=True*)

this is another interface to the weight… properties. A prototype of this method can be inserted from the editor insert menu.

Usually you will use *Group.useattributes* and set weight properties from the group.

Parameters

- **style** – 'dots', 'connected' or 'individual' see(*weightstyle*)
- **factor** (*float*) – see *weightfactor*
- **min** (*float*) – see *weightmin*
- **max** (*float*) – see *weightmax*
- **showinlegend** (*int*) – see *showinlegend*

Returns

self to allow call chaining as in

```
Set.setWeightsStyle(...).setSomethingElse(...)
```

Return type *Set*

convolute(*width*)

replace current set data by a convolution with Gaussians of a given width.

Parameters **width** (*float*) – the Gaussin width

Returns

self to allow call chaining as in

```
Set.convolute(...).setSomethingElse(...)
```

Return type *Set*

integrate()

replace the current set by a running integral of the data. The last value of the y-data is the total integral (if needed):

```
s=G[1].Gr[1].S[1].integrate() # returns the set itself
print 'integral',s.y[-1]
```

Returns

self to allow call chaining as in

```
Set.integrate(...).setSomethingElse(...)
```

Return type *Set*

moment(*order*, *interval=None*, *normalized=True*)

calculate the *order*-th moment of the y-data over the whole x-data interval (if *interval* is `None` or missing) or over a given interval.

If *normalized* is `True` these are normalized moments $M_n = \frac{\int_{x_0}^{x_1} f(u)u^n du}{\int_{x_0}^{x_1} f(u)du}$, hence $M_1 \equiv 1$. The second moment corrected for the center of gravity is defined as $\frac{\langle (x-M_1)^2 \rangle}{\langle \rangle} = M_2 - M_1^2$.

Otherwise, they are unnormalized $M_n = \int_{x_0}^{x_1} f(u)\, u^n du$,

The unnormalized moment of order 0 is the same as the integral:

```
gr=G[1].Gr[1]
s=gr.S[1] # assume it exists
s2=gr.S.new(s)
s.integrate()
m=s2.moment(0,normalized=False)
print s.y[-1],m
```

Example:

```
import numpy as np

killall()
g=G[1]
gr=g.Gr[1].on()
N=1000
a=0.5
x0=-0.6
x1=1.6
w=0.2
s=gr.S[1].on()
# s1 length N
s.x=Vector(N,x0,x1)
```

(continues on next page)

(continued from previous page)

```
s.y=np.exp(-((s.x-a)/w)**2/2)*3
m1=s.moment(1,[x0,x1])
m2=s.moment(2,[x0,x1])
# calculate the normalized width
wi=np.sqrt(m2-m1**2)
t=g.textboxes.new()
t.text='calculated width={} input width={}'.format(wi,w)

gr.usertics.on()
gr.usertics.append(Tic(type='xmajor',position=m1,label='m1',
        labelside='Opposite',length=1,line=LineStyle(style='Solid')))\
    .append(Tic(type='xmajor',position=m1+wi,label='wi',
        labelside='Opposite',length=1,line=LineStyle(style='Dash')))\
    .append(Tic(type='xmajor',position=m1-wi,label='wi',
        labelside='Opposite',length=1,line=LineStyle(style='Dash')))



g.title.text='Moment example'
g.title.restriction=('y',0.02)
g.autoscale()
```

Parameters

- **`order`** (*`int`*) – the requested order
- **`interval`** (*`2-tuple or 2-sequence`*) – x-interval on which to calculate
- **`normalized`** (*`int`*) – normalized or not

Returns the moment of requested order

Return type float

`bspline` (*order*, *derivorder*, *xvector=None*)

Replace this set's y-data with the *derivorder*-th derivative of the bspline of order *order*, evaluated at the set's x-data if *xvector* is missing (or `None`), or on the provided *xvector*, in which case the set's x-data is set to *xvector*

Parameters

- **`order`** (*`int`*) – the requested spline order
- **`derivorder`** (*`int`*) – the requested derivative order
- **`xvector`** (None, seqence(list, *`Vector`*, tuple, numpy.ndarray) of float) – a possible alternative x-mesh

Returns

self to allow call chaining as in

```
Set.bspline(...).setSomethingElse(...)
```

Return type *`Set`*

`adjustDensPlot` ()

adjust data range settings according to a guess for the current *`zcomponent`*. Affected are *`scalemax`* and *`z0`*. See *Set Dialog*.

Returns

self to allow call chaining as in

```
Set.adjustDensPlot(...).setSomethingElse(...)
```

Return type *`Set`*

id
the set's id

Type int

group
return the group, the set is in

Type *Group*

graph
return the graph, the set is in

Type *Graph*

active
get/set if the set is visible. A nonzero value is `True`, zero is `False`.

Type bool

line
get/set the linestyle.

Type *LineStyle*

legend
get/set what is displayed in the legend.

Type str

comment
get/set the "set-comment" of the set. Set-comments denote, where the data came from. This is mostly set by file-loading routines.

Type str

showinlegend
get/set if this set is shown in the legend box.

Type bool

symbol
get/set the symbolstyle. Not used for *Weight* s, which have their own limited settings.

Type *SymbolStyle*

interpolationdepth
get/set the interpolation depth for density plots (xynz plots). A density plot can have a limited number of data points (pixels) due to resource restrictions. These data can be interpolated to give a smoother picture. *interpolationdepth* determines on how many sub pixels the data re-interpolated. See *Set Dialog*.

Type int

scalemin
set the lower cutoff for density plots. (see *Set Dialog*)

Type float

scalemax
set the upper cutoff for density plots. (see *Set Dialog*)

Type float

z0
data below this z-value will be considered background in grid/density plots (see *Set Dialog*)

Type float

databackgroundcolor
the data background color in grid/density plots. (see *Set Dialog*)

Type int

zpower

set the power law for mapping of z-values onto colormaps for grid/density plots (see *Set Dialog*)

Type int

colormap

get/set the colormap for density plots. (see *Set Dialog*)

A colormap can be a

a name (str) of a predefined map

'Terrain'	'RainBow'	'Magma'	'Inferno'	'Hot'	'Heat'
'Spring'	'Summer'	'Autumn'	'Winter'	'Gnuplot'	'Seismic'
'Rainbow2	'Rainbow3'				

a tuple of two colors which are RGB-interpolated between e.g. `(0xff00,0xff0000)`

a colormap which has the shape `[ [z0,color0], [z1,color1] ,...,interpolrgb]` where zi should be in [0,1] and monotonous and colori are hex numbers representing rgb colors. interpolrgb must evaluate to `bool` and determines the interpolation space.

The getter returns a full colormap. Example:

```
s.colormap=[
  [ 0.0 , 0x0ff ] ,
  [ 0.333333333333 , 0xff00dd ] ,
  [ 0.666666666667 , 0xff0000 ] ,
  [ 1.0 , 0xffff00 ] ,
  True
 ]
```

Type str or sequence

weightstyle

get/set the style of the weights

Usually you will use `Group.useattributes` and set weight properties from the group.

'dots'	individual filled circles for each data point
'connected'	connect the circles by linear intepolation
'individual'	each weight can have its own symbol

weightfactor

get/set the weight factor. Usually you will use `Group.useattributes` and set weight properties from the group.

All weights are scaled by this factor, before applying `weightmax` and `weightmin`. (This is somehow superfluous, but usefull anyway).

Type float

weightmin

get/set the weight symbol size, under which no symbols will be plotted. Usually you will use `Group.useattributes` and set weight properties from the group.

This depends on `weightmax` since it scales everything up. It actually also depends on other scales, so just do trial & error for the desired result.

Type float

weightmax
get/set the symbol size, which represents a weight value of 1. weightmax is in a scale like font sizes. Usually you will use `Group.useattributes` and set weight properties from the group.

Type float

showweightsinlegend
get/set if weight legend entries are shown. Usually you will use `Group.useattributes` and set weight properties from the group.

Type bool

W

this gives acces to all weights. Usually you will use `Group.useattributes` and set weight properties from the group.

```
G[1].Gr[1].S[1].W[1].off()
G[1].Gr[1].S[1].W[5].on().setStyle(...)
G[1].Gr[1].S[1].W[5].color=0xff
```

Type `Weights`

type
get/set data type. Possible values:

'xy' y=function(x), x can be non monotonous

'xynw' y=function(x), additional n weights w=weightfunc(x) are defined

'xynz'. z=function(x,y), there can be several components i.e. several z(x,y)

Warning: usually one does not need to set this. It can leave a confusing state.

Type str

zcomponent
get/set which zcomponent (densplot, xynz type) is plotted. This is a zero-based index. The first component is `zcomponent==0`. See *Set Dialog*.

Type int

x

get/set the vector of the abscissa or x-values. You can use `Vector`, which is used to allow standard arithmetic. It is mostly a class for intermediate results:

```
s=G[1].Gr[1].S[1]
s.x=Vector(100,-2.,10.) # !!! note, that N is first argument not last
                        # as for numpy.linspace !!!

s.x*=0.529177
s.y=((s.x+2)*3)**2  # intermediate results are Vector instances

#or
import math as m
s.y=map(m.sin,s.x) # awkward but working if numpy is not available

# or do this
for i,x in enumerate(s.x):
   s.y[i]=m.sin(x)
```

You can freely convert to and from numpy.ndarray:

```
import numpy as np
arr=np.array(G[1].Gr[1].S[1].x)
```

set from numpy array like this:

```
import numpy as np
G[1].Gr[1].S[1].x=arr
```

For example

```
import numpy as np
s=G[1].Gr[1].S[1]
s.x=np.linspace(-2.,10.,100)
s.y=s.x**2  # intermediate Vector instance

#or simply

s.x*=0.529177

s.y=np.sin(s.x)

#or
for i,x in enumerate(s.x):
   s.y[i]=np.sin(x)
```

Type `SetVector`

y

return the vector of the ordinate or y-values. See `x`.

Type `SetVector`

z

get/set the z-components (densplot (xynz type) or band weights (xynw type)). You can set z-components in a block or access its individual data via `S[i].z[iw]`. ZComponents behaves differently for xynw and xynz data types.

For xynw, each zcomponent represents one weight. So in general you have *Nband* sets and each set has *Norbital* weights. This is reflected by *Norbital* z-components for each set. `S[3].z[iw]` is the vector of weigths for set 3 and weight *iw*. `S[3].z[iw,ik]` is the weigth for set 3 and weight *iw* at k-point *ik*.

You can set the z-components in bulk `S[3].z=arr`, where `arr` must be a sequence of *Norbital* sequences of length `len(S[i].x)` (*Nk*). All sets should have the same length. A 2d numpy array can be used for `arr`. In general the outer (first) dimension is *Norbital* and the second (inner) *Nk*: `S[3].z=[ [weight1],[weight2],...]`, where `weight1 ...` are sequences of *Nk* weights.

If you construct xynw sets by yourself you need to call `Group.unifyWeightLabels` after the act, if you want the result to behave like a bandweights plot.

Example:

```
import numpy as np

killall()

gr=G[1].Gr[1].on()

N=100
x0=0.
x1=np.pi

for id in [1,2,3]: # 3 sets
    s=gr.S[id].on()
    s.x=np.linspace(x0,x1,N)
    s.y=np.cos(s.x*id)
    # set weight plot type
```

(continues on next page)

(continued from previous page)

```
    s.type='xynw'
    # three weights
    s.z.len=3 # this is important to set up internal weightlabels
    # set at once
    s.z=[3*s.x,2*s.x,1*s.x] # assign list of three weight vectors
    # or
    w=np.array([3*s.x,2*s.x,1*s.x]) # we got some numpy array
    s.z=w # and assign it

    # or via indices
    for i in range(N):
        s.z[0,i]=3*np.abs(s.x[i]) # weight 1
        s.z[1,i]=2*np.abs(s.x[i]) # weight 2
        s.z[2,i]=1*np.abs(s.x[i]) # weight 3

gr.unifyWeightLabels()
gr.useattributes=1
gr.setWeightsStyle(style='connected',factor=1,min=0.2,max=6,
 showinlegend=True)

for w in gr.W:  w.on()

gr.W[1].color=0xaa00
gr.W[2].color=0xff00ff
gr.W[3].color=0xffff00

G[1].autoscale()
```

For the xynz data type the ZComponents represent *Nc* z(x,y) functions, where *Nc* is *`z.len`*. Only one can be plotted at once, which is given by *`zcomponent`*. Some FPLO programs produce multiple z-components (ugly, I know). Each z-component contains `s.x.len*s.y.len` values z(x,y). The data can be accessed in bulk: s.z returns a list of *Nc* flat sequences, where each sequence is one component. In these flat sequences the x-coordinate runs first. `s.z` can be set from such a list or from a numpy array with `shape=(Nc,len(s.x)*len(s.y))`.

Another option is to obtain the ic-th zcomponent (still flat) via `s.z[ic]` this returns or assigns one whole z(x,y) from a flat sequence, where x runs first.

The last option is to access each element separately via `s.z[ic,ix,iy]`. This option is the slowest. To illustrate we give an example, where all possibilities are shown:

```
import numpy as np

def f(x,y):
    return np.cos(x)+np.cos(x*(x**2+y**2))

killall()

g=G[1]
gr=g.Gr[1].on()
s=gr.S[1].on()
s.type='xynz' # set the data type

Nx=200
Ny=100
# set the x and y values from numpy arrays
s.x=np.linspace(-np.pi*3,np.pi*3,Nx)
s.y=np.linspace(-np.pi*3,np.pi*3,Ny)

# the number of components
s.z.len=1
```

(continues on next page)

(continued from previous page)

```
# construct a flat numpy array containing z(x,y)
# x runs first
w=np.array([f(x,y) for x in s.x for y in s.y ])

# assign in bulk
s.z=[w]

# or assign single component
s.z[0]=w

# or assign each data point separately
for i,xx in enumerate(s.x):
    for j,yy in enumerate(s.y):
        s.z[0,i,j]=f(xx,yy)

# which component to plot
s.zcomponent=0
# setup some default scalemax, z0
s.adjustDensPlot()
# not good enough
s.z0=-2
# pick colormap
s.colormap='heat'

# make it smoother
s.interpolationdepth=2

#and
g.autoscale()

g.title.text='Illusion'
```

Type `ZComponents`

6.1.8 SetVector

class SetVector(*graphid, groupid, setid, xyz, ic=-1*)

A SetVector object is returned by `Set.x`, `Set.y` and `z[]` for some versions of indexing. It represents the x-data, y-data or z-component data (depending on the data `type`). You can assign a list or 1d `numpy.ndarray` or change single values. You can get/set the size via `len`. In this case there are two cases. For `Set.type` 'xy' and 'xynw' the size of `Set.x` and `Set.y` are the same since it represents y=f(x). Chaning the `len` of either changes the length of the other. For 'xynz' data the `len` of x and y can be different, since both are independent variables.

If the SetVector represent a z-component of a 'xynw' type, the length is the same as `x.len` and `y.len`. Setting the `x.len` is automatically setting the z-component length.

If the SetVector represents a z-component of a 'xynz' type (density plot) the `s.z.len` is `s.x.len*s.y.len`.

Instead of individual element access `x[12]=7` one can assign a sequence (see `Set.x`). In the latter case automatic resizing will take place.

Parameters

- **graphid** (*int*) – the graph id
- **groupid** (*int*) – the group id
- **setid** (*int*) – the set id

- **xyz** (*str*) – axis id 'x' or 'y' or 'z'
- **ic** (*int*) – if used: which z-component

__len__()
`len(...S[1].x)` returns the size of the vector (see also *len*)

__iter__()
you can iterate over a vector:

```
su=0
for d in G[1].Gr[1].S.x:
    su+=d
su/=G[1].Gr[1].S.x.len
```

This is **NOT** usefull for assignment.

next()
for the iterator interface, see *__iter__*

__getitem__()
return the i-th element:

```
...S[1].x[2]=...S[1].x[2]+0.1
```

__setitem__()
assign the i-th element:

```
...S[1].x[2]=...S[1].x[2]+0.1
```

__add__()
enable adding as in:

```
s=G[1].Gr[1].S[1]
s.x+=0.1
s.x=s.x+2
```

__sub__()
enable sutraction as in:

```
s=G[1].Gr[1].S[1]
s.x-=0.1
s.x=s.x-2
```

__mul__()
enable multiplication as in:

```
s=G[1].Gr[1].S[1]
s.x*=0.529177
s.y=(s.x+2)*3
```

__div__()
enable division as in:

```
print s.x/5 # same as [x/5 for x in s.x]
print 5/s.x # same as [5/x for x in s.x]
```

__pow__()
enable power as in:

```
s=G[1].Gr[1].S[1]
s.y=s.x**2
```

__neg__()
enable unary minus:

```
s=G[1].Gr[1].S[1]
s.y=-s.x
```

__pos__()
enable unary plus:

```
s=G[1].Gr[1].S[1]
s.y=+s.x
```

__abs__()
abs(objects)

len
get/set the length of the vector

Type int

6.1.9 Vector

class Vector(*size=0*, *x0=0*, *x1=0*)
Create a new Vector with *size* linearly spaced elements reaching from *x0* to *x1*. This is a minimal implementation of a float vector, to obtain simple element wise list math for `SetVector` without compiling with numpy. It should be more or less invisible to the user that this class appears here and there.

The point is that expressions like `s.x+2` must return a list with element wise arithmetic capabilities in order to be able to write `(s.x+2)*7`. Here the `s.x` is a `SetVector`. SetVector itself is not suited since it is a reference to real data and not data itself. We can implement *__add__* for SetVector to return the expression in parentheses as a `list`, but it must return more than a python `list` since `(..)*7` should also work. Ideally, we would return a `numpy` array, but this requires more dependencies. Numpy can still be used on this object as in

```
X=Vector(10,0,1) #instead of np.linspace, X is a Vector
Y=np.cos(X*2+3)
```

However, it also works without numpy:

```
from math import *
s.x=Vector(10,0,1)
s.y=map(cos,s.x+2)

s.y+=0.1
s.y=-((s.x+2)*7)**2
```

Manipulate vector elements in the following way:

```
v=Vector() # zero length
v.len=10 # lenght 10
v[2]=42  # element wise assignment
print v[2] # __getitem__

v=Vector() # zero length
v.append(1)
v.append(2)
v.append(4) # v contains [1,2,4]
```

Parameters

- **size** (*int*) – number of elements

- **x0** (*float*) – first value
- **x1** (*float*) – last value

__add__()
enable adding as in:

```
s=G[1].Gr[1].S[1]
s.x+=0.1
s.x=s.x+2
```

__sub__()
enable sutraction as in:

```
s=G[1].Gr[1].S[1]
s.x-=0.1
s.x=s.x-2
```

__mul__()
enable multiplication as in:

```
s=G[1].Gr[1].S[1]
s.x*=0.529177
s.y=(s.x+2)*3
```

__div__()
enable division as in:

```
v=Vector(5,1,5) # v=[1,2,3,4,5]
print v/5 # [0.2,0.4,0.6,0.8,1]
print 5/v # [5,2.5,1.666..,1.25,1]
```

__pow__()
enable power as in:

```
s=G[1].Gr[1].S[1]
s.y=s.x**2
```

__neg__()
enable unary minus:

```
s=G[1].Gr[1].S[1]
s.y=-s.x
```

__pos__()
enable unary plus:

```
s=G[1].Gr[1].S[1]
s.y=+s.x
```

__abs__()
enable absolute value:

```
s=G[1].Gr[1].S[1]
s.y=abs(s.x)
```

__len__()
`len(v)` returns the size of the Vector.

__getitem__()
return i-th item (float)

`__setitem__`()
assign a float to the i-th item

`__iter__`()
you can iterate over a Vector:

```
v=Vector(11,0.,10.)
su=0
for x in v:
    su+=x
su/=v.len
```

`next`()
for the iterator interface, see `__iter__`

`append`(*d*)
append *d* to the end of the Vector.

Parameters **d** (*float*) – a new value

Returns

self to allow call chaining as in

`Vector.append(...).setSomethingElse(...)`

Return type *Vector*

`len`
get/set the size of the vector. After setting a larger len some elements can be uninitialized.

Type int

6.1.10 ZComponents

class `ZComponents`(*graphid*, *groupid*, *setid*)
This class is returned by *z*. It can be used to manipulate the weigths (xynw type) or densplot functions z(x,y) (xynz type). Please read the doc for *z*.

Parameters

- **graphid** (*int*) – the graph id
- **groupid** (*int*) – the group id
- **setid** (*int*) – the setid

`__len__`()
`len(s.z)` returns the number of z-components.

xynw Number of weigths

xynz Number of independent z(x,y) functions stored in the data

`__getitem__`()
indexing works the following way:

xynw

`z[iw]` : the `iw`-th weight vector (length `s.x.len==s.y.len`) Note, that here `iw` is zero based, while weight ids start with id 1 (Sorry).

`z[iw,ix]` : the `iw`-th weight at `xvalue[ix]` (a number)

xynz

`z[ic]` : the whole `z[: , :]` for the `ic`-th component as flat vector, where `ix` runs first” `z[ic,ix,iy]` : `z[ix,iy]` for the `ic`-th component (a number)

`__setitem__`()

see *`__getitem__`*. When a certain index structure for __getitem__ returns a sequqence, assignment has also to occur from a sequence. If it returns a number you must assign a number.

As an exception from this rule you can always assign a single number, which sets the whole sequence to this number if the index structure result in a sequence:

```
s.z=0
```

`len`

get/set the number of z-components.

xynw this is the number of weights

xynz this is the number of independent z(x,y) functions stored in the data.

see examples in *`Set.z`*.

Type int

6.1.11 Weights

class `Weights`(*graphid*, *groupid*, *setid=0*)

The Weights class gives access to all available `Weigth` objects, which in turn determine the properties of plotted weights. As usual there is standalone and hierarchical acces. Weigths can be accessed from *`Group`* s or *`Set`* s:

```
G[1].Gr[2].W # Weigths class of grap(id 1), group(id 2)
Weights(1,2) # the same

G[1].Gr[2].S[3].W # Weigths class of grap(id 1), group(id 2), set(id3)
Weights(1,2,3)     # same
```

A single *`Weight`* can be obtained by indexing, either with an `int` index or a `str` index. `int` indices are the weight number (one-based counting). `str` indices are the name of a weight. The name has to match exactly:

```
G[1].Gr[2].W[5] # weight number 5
G[1].Gr[2].W['Al(001)3p+0'] # a specific weight
```

Note, that you can change the weight *`name`* in which case the new name can be used as index (the old no longer).

Parameters

- **`graphid`** (*`int`*) – the graph id
- **`groupid`** (*`int`*) – the group id
- **`setid`** (*`int`*) – missing (group weigth access) or set id

`__delitem__`(*id*)

you can delete a weight with a certain id:

```
del G[1].Gr[2].W[11]
```

`__len__`()

`len(G[1].Gr[2].W)` returns the number weights

`__getitem__`()

A single *`Weight`* can be obtained from an `int` index or a `str` index. `int` indices are the weight number (one-based counting). `str` indices are the name of a weight. The name has to match exactly.

__iter__()

you can iterate over all weights:

```
for w in G[1].Gr[1].W:
    w.skip=3
```

next()

for the iterator interface, see *__iter__*

6.1.12 Weight

class Weight(*graphid*, *groupid*, *setorweightid*, *weightid=1*)

A funtion or set of functions can have associated weigths, which can be used for a fat-band plot. The Weight class determines the appearance of the individual weights in such a plot. If all sets of a group share the same weightlabel set the weight properties can be set from the group for all sets at once if *Group.useattributes* is `True`. This sharing is setup by the *Graph.read* methods. If the user creates weights on the fly *Group.unifyWeightLabels* will setup this sharing (see *Set.z*). Normal usage is:

```
gr=G[1].Gr[1]
gr.W[1].off()
for id in [5,6,7]:
   gr.W[id].on()
   gr.W[id].skip=4
gr.W[5].color=0xff0000
gr.W[6].color=0xaa00
gr.W[7].color=0xff
```

If the class is accessed directly (`Weight(1,2,...)`) and called with three arguments they are: *graphid*, *groupid*, *weightid*. If called with four: *graphid*, *groupid*, *setid*, *weightsid*.

Parameters

- **graphid** (*int*) – the graph id
- **groupid** (*int*) – the group id
- **setorweightid** (*int*) – set/weight id
- **weightid** (*int*) – weightid

on()

switch the weight on

Returns

self to allow call chaining as in

`Weight.on(...).setSomethingElse(...)`

Return type *Weight*

off()

switch the weight off

Returns

self to allow call chaining as in

`Weight.off(...).setSomethingElse(...)`

Return type *Weight*

toggle()

toggle the weight's visibility

Returns

self to allow call chaining as in

```
Weight.toggle(...).setSomethingElse(...)
```

Return type *Weight*

setStyle(*style='square'*, *color=255*, *skip=0*, *linewidth=1*, *fill=1*)
convenience method to set the style at once. *name* and *plotorder* must be set separately.

Parameters

- **style** – see *style*
- **color** (*int*) – see *color*
- **skip** (*int*) – see *skip*
- **linewidth** (*int*) – see *linewidth*
- **fill** (*int*) – see *fill*

Returns

self to allow call chaining as in

```
Weight.setStyle(...).setSomethingElse(...)
```

Return type *Weight*

id
the weight id

Type int

active
get/set if the Weight is visible

Type bool

name
get/set the name of the weight.

Type str

plotorder
get/set the plot order. Weights with higher plot order are plotted later.

Type int

color
get/set the color as int. To print in hex form use e.g. `hex(...W[1].color)`. To set, use `W[1].color=0xff00ff` (magenta) (see *Colors*)

Type int

skip
get/set how many data points are skiped before the next weight symbol is plotted. If there are many data points in the sets, which carry weights it might be advantageous to only plot a diluted number of weight symbols.

Type int

style
get/set the weight style either from `int` or from `str` (If 'individual' weight style is set via *Group.weightstyle*/*Set.weightstyle*) The `str` version has short forms.

str	short	int	str	short	int	str	short	int
'none'	' '	0	'triangleup'	'^'	4	'plus'	'+'	8
'circle'	'o'	1	'triangleleft'	'<'	5	'cross'	'x'	9
'square'	'q'	2	'triangledown'	'v'	6	'star'	'*'	10
'diamond'	'd'	3	'triangleright'	'>'	7			

Type str/int

linewidth
get/set the line width of a weight symbol (If 'individual' weight style is set via `Group.weightstyle`/`Set.weightstyle`)

Type float

fill
get/set if the the weight symbols are filled (If 'individual' weight style is set via `Group.weightstyle`/`Set.weightstyle`)

Type bool

6.1.13 NewGroup

class NewGroup (*graphid*, *groupid*)
The NewGroup object is an element of the list returned by `Graph.read` and provides access to newly read `Group` s and `Set` s. This way one does not need not know, which groups/sets where created:

```
# a simple case: non-spinpolarized or relativistic bandstructure
#  (only 1 group)
gr=G[1].read('band','+band')[0].group
gr.line.color=0xff00

# same but spin polarized
grps=G[1].read('band','+band')
gr=grps[0].group
gr.line.color=0xff
gr=grps[1].group
gr.line.color=0xff00

# another example
grps=G[1].read('xny','+band') # note the xny type
for r in grps:
    for s in r.sets:
       s.line.width=2
       s.legend='group {grid} set {sid}'.format(grid=r.group.id,sid=s.id)
```

Parameters

- **graphid** (*int*) – the graph id
- **groupid** (*int*) – the group id

group
the new `Group` object

Type `Group`

sets
the list of new `Set` objects

Type list of `Set`

6.1.14 LineStyle

class LineStyle (*width=1*, *color=0*, *style='solid'*, *extracolor=1*)
A LineStyle object is returned from other object's properties such that we can change it:

```
G[1].Gr[1].S[1].line.color=0xff00 # here line returns LineStyle
```

Or we can create it separately and use it:

```
l=LineStyle(width=1.5,color=0xff,style='Dash')

for s in G[1].Gr[1].S:
   s.line=l
```

Parameters

- **width** (*float*) – the line width
- **color** (*int*) – the line color
- **style** – see *style*
- **extracolor** (*int*) – see *extracolor*

width
get/set the line width

Type float

color
get/set the line color as int(hex.) To print in hex form use e.g. `hex(...line.color)`. To set the color use `rim.color=0xff00ff` (see *Colors*).

Type int

style
get/set the line style as int or str. Invalid numbers are folded back into the allowed range (modulus). Valid styles are defined below.

'none'	0	'solid'	1	'dash'	2
'dot'	3	'dashdot'	4	'dashdotdot'	5
'dotdashdash'	6	'longdash'	7	'longdashdot'	8
'longdashdotdot'	9				
'dotlongdashlongdash'	10				

Type str/int

extracolor
get/set if separate line color is used for symbols. (in *SymbolStyle*)

Type bool

6.1.15 FillStyle

class FillStyle(*active=1*, *color=16777215*, *extracolor=1*)
A FillStyle object is returned from other object's properties such that we can change it:

```
G[1].legend.frame.fill.color=0xff00 # here fill returns FillStyle
```

Or we can create it separately and use it:

```
fs=FillStyle(active=True,color=0xffffff,extracolor=True)

for s in G[1].Gr[1].S:
   s.symbol.style='d'
   s.symbol.fill=fs
```

Parameters

- **active** (*int*) – fill or not
- **color** (*int*) – fill color
- **extracolor** (*int*) – see *extracolor*

on()

switch filling on

Returns

self to allow call chaining as in

`FillStyle.on(...).setSomethingElse(...)`

Return type *FillStyle*

off()

switch filling off

Returns

self to allow call chaining as in

`FillStyle.off(...).setSomethingElse(...)`

Return type *FillStyle*

active

get/set if filling is active.

Type bool

extracolor

get/set if separate fill color is used. (not used by all objects) If this property is used `extracolor=False` will trigger the use of the line color for filling. Otherwise *color* will be used.

Type bool

color

get/set the fill color. To print in hex form use e.g. `hex(...fill.color)` Set it via `fill.color=0xff00ff` (see *Colors*)

Type int/hex

6.1.16 FontStyle

class FontStyle(*size=12*, *subscriptscale=0.75*, *color=0*)

A FontStyle object is returned from other object's properties such that we can change it:

```
G[1].title.font.size=18
```

Or we can create it separately and use it:

```
ft=FontStyle(size=18,subscriptscale=0.75,color=0xff)

for t in G[1].textboxes:
   t.font=ft
```

Parameters

- **size** (*int*) – font size
- **subscriptscale** (*float*) – relative subscript font size scale
- **color** (*int*) – font color

color
get/set the color as int. To print in hex form use e.g. `hex(...rim.color)` Set color form as `font.color=0xff00ff` (see *Colors*)

Type int/hex

size
get/set the font size.

Type float

subscriptscale
get/set the relative scale of the subscripts font size compared to normal font size.

Type float

6.1.17 SymbolStyle

class SymbolStyle(*style='none'*, *size=12*, *line=None*, *fill=None*)
A SymbolStyle object is returned from *Set*/*Group* object's such that we can change it:

```
G[1].Gr[1].S[2].symbol.style='circle'
```

Or we can create it separately and use it:

```
sy=SymbolStyle(style='',size=12,
  line=LineStyle(style='Solid',width=1,color=0x0,extracolor=False),
  fill=FillStyle(active=False,color=0xffffff,extracolor=False))



for s in G[1].Gr[1].S:
   s.symbol=sy
```

Parameters

- **style** (`str`) – see *style*
- **size** (`float`) – symbol size
- **line** (*LineStyle*) – symbol's line style
- **fill** (*FillStyle*) – symbol's fill style

style
get/set the symbol style either as `int` or as `str`

str	short	int	str	short	int	str	short	int
'none'	''	0	'triangleup'	'^'	4	'plus'	'+'	8
'circle'	'o'	1	'triangleleft'	'<'	5	'cross'	'x'	9
'square'	'q'	2	'triangledown'	'v'	6	'star'	'*'	10
'diamond'	'd'	3	'triangleright'	'>'	7			

Type str/int

size
get/set the symbol size.

Type float

line
get/set the line style of the symbol. If *LineStyle.extracolor* is `False` the line color will be the same as the color of the *Set.line* style.

Type *LineStyle*

fill
get/set the fill style of the symbol. If *FillStyle.extracolor* is `False` the fill color will be the same as the color of the *Set.line* style.

Type *FillStyle*

6.1.18 Frame

class Frame(*active=1*, *borderspacing=0.2*, *rim=None*, *fill=None*)
A Frame object is returned e.g. by *Legend* or *TextBox*. It can be manipulated or use standalone for assignment:

```
G[1].legend.frame.off()

Legend(1).on()\
.setFrame(Frame(active=1,borderspacing=0.4,
    rim=LineStyle(style='Solid',width=1,color=0x0,extracolor=True),
    fill=FillStyle(active=True,color=0xffffff,extracolor=True)))
```

Parameters

- **active** (*int*) – is the frame visible?
- **borderspacing** (*float*) – see *borderspacing*
- **rim** (*LineStyle*) – rim line style
- **fill** (*FillStyle*) – frame fill style

on()
show the frame

Returns

self to allow call chaining as in

`Frame.on(...).setSomethingElse(...)`

Return type *Frame*

off()
don’t show the frame

Returns

self to allow call chaining as in

`Frame.off(...).setSomethingElse(...)`

Return type *Frame*

toggle()
toggle the frame’s visibility

Returns

self to allow call chaining as in

`Frame.toggle(...).setSomethingElse(...)`

Return type *Frame*

active
get/set if the frame is visible

Type bool

borderspacing
get/set the space between the rim and the content.

Type float

rim
get/set the rim line style. The getter returns a reference object such that changing it changes the object which owns rim. The setter makes a hard assignment of the rim data of the underlying object. The reference stays intact:

```
fr=G[1].textboxes[1].frame # fr refers to the textbox frame

ls=fr.rim # ls and fr are references ...

ls.width=2 # changes width in ls, fr and textbox(id 1)

fr.rim=LineStyle() #will hard set new rim data

ls.width=3 # now ls,fr and textbox have width 3
```

Type *LineStyle*

fill
get/set the fill style. See rim for reference logic.

Type *FillStyle*

6.1.19 Paper

class Paper
This class represents the paper which is the white area in the GUI containing everything. It is returned by global namespace *paper* or *Xfbp.paper*. Examples:

```
paper.size='a4'
paper.orientation='portrait'
# or
paper.size=(500,800) # not the same size but close
```

That's all there is to it.

size
get/set the paper size. There are two options. First one can give the paper size as a sequence (tuple,list,. . .) of two int:

```
paper.size=(500,800)
```

or one can give a paper size str

a0	. . .	a10
b0	. . .	b10
c1	. . .	c7
isob0	. . .	isob10
letter	legal	. . .

Type str or 2-sequence of int

orientation
get/set the orientation to 'landscape' or 'portrait'

Type str

6.1.20 World

class World(*graphid*)

The World object determines the part of the data range, which is shown inside the `Graph`'s viewport (`View`). A free standing object can be created or the hierarchy can be used:

```
World(1).x=[-1,11] # the world object of the graph with id 1

G[1].world.x=[-1,11] # the same thing

g=G[1]
# do somthing with graph 1
....
w=g.world
w.x=[-1,11] # yet the same thing
w.offset=(0,0.1) # leave some space in y-direction
w.autoscale('y') # let it be done
#or
g.autoscale('y') # let it be done, its all the same

print w.xmin,w.xmax # printing will be shown in the console (xterm)
print w.ymin,w.ymax
```

Parameters **graphid** (*int*) – the graph id

autoscale(*what='all'*)

autoscale graph using the currently set `offset`

Parameters **what** (*str*) – 'all' ,'x' or 'y'

Returns

self to allow call chaining as in

`World.autoscale(...).setSomethingElse(...)`

Return type `World`

setX(*x0=0, x1=0*)

convenience function to set the `x`-world extend

Parameters

- **x0** (*float*) – leftmost x-value
- **x1** (*float*) – righttmost x-value

Returns

self to allow call chaining as in

`World.setX(...).setSomethingElse(...)`

Return type `World`

setY(*y0=0, y1=0*)

convenience function to set the `y`-world extend

Parameters

- **y0** (*float*) – lowest y-value
- **y1** (*float*) – highest y-value

Returns

self to allow call chaining as in

`World.setY(...).setSomethingElse(...)`

Return type *World*

setOffset (*x=0*, *y=0*)
convenience function to set the autoscale `offset`

Parameters

- **x** (`float`) – the relative offset in x
- **y** (`float`) – the relative offset in y

Returns

self to allow call chaining as in

```
World.setOffset(...).setSomethingElse(...)
```

Return type *World*

x
get/set the x-world extend

Type 2-tuple of float

y
get/set the y-world extend

Type 2-tuple of float

xmin
get/set the leftmost x-value

Type float

ymin
get/set the lowest y-value

Type float

xmax
get/set the rightmost x-value

Type float

ymax
get/set the highest y-value

Type float

offset
get/set autoscale offset in percent*100 of the autoscale interval. The setter argument is a sequence of 2 float: (offsetx,offsety).

Type 2-tuple or 2-list of float

6.1.21 View

class View (*graphid*)
The View object describes the viewport of a *Graph*, the area which is usually inside a frame with ticmarks and such stuff. It is where the data are plotted in on the screen. A free standing object can be created or the hierarchy can be used:

```
View(1).x0=0.18 # dont ask why we use x0,y0,width,height here
View(1).width=0.77  # it's historical
View(1).y0=0.166
View(1).height=0.668

# or the same
```

(continues on next page)

(continued from previous page)

```
G[1].view.setGeometry(0.18,0.166,0.77,0.668)
# or more verbose
G[1].view.setGeometry(x0=0.18,y0=0.166,width=0.77,height=0.668)

G[1].view.setFrame(Frame(active=1,
  rim=LineStyle(style='Solid',width=1,color=0x0),
  fill=FillStyle(active=True,color=0xffffff)))
```

Parameters **graphid** (*int*) – the graph id

setFrame (*frame*)
convenience function for setting the *frame*

Parameters **frame** (*Frame*) – the frame

Returns

self to allow call chaining as in

`View.setFrame(...).setSomethingElse(...)`

Return type *View*

setGeometry (*x0=0.18*, *y0=0.166*, *width=0.7*, *height=0.668*)
convenience function for setting the *x0*, *y0*, *width* and *height*

Parameters

- **x0** (*float*) – relative left border
- **y0** (*float*) – relative top border
- **width** (*float*) – relative width
- **height** (*float*) – relative height

Returns

self to allow call chaining as in

`View.setGeometry(...).setSomethingElse(...)`

Return type *View*

frame
get/set the frame. The getter returns a reference object such that changing the returned frame changes the object which owns frame. The setter makes a hard assignment of the frame data of the underlying object while not changeing the reference.

Type *Frame*

x0
get/set the border left of the view frame in percent*100 of the paper width

Type float

y0
get/set the border above of the view frame in percent*100 of the paper height

Type float

width
get/set the width of the view frame in percent*100 of the paper width

Type float

height
get/set the height of the view frame in percent*100 of the paper height

Type float

6.1.22 Axis

class Axis(*graphid*, *xy*)

The Axis object controls the axis scaling of the *Graph*:

```
G[1].xaxis.scaling='log'
#or
Axis(1,'x').scaling='log'
```

Note, that we use a trick to allow for logarithmic axes with negative world values. In a case where e.g. `World.xmin<0` we adjust `xmin` to take the smallest representable positive number. This way no error message pops up but your graph looks weird.

Parameters

- **graphid** (*int*) – graph id
- **xy** (*str*) – axis id 'x' or 'y'

scaling

get/set the axis scaling 'lin' or 'log'

Type str

6.1.23 Legend

class Legend(*graphid*)

The Legend object represents the legend box of the *Graph*. The legend entries are not defined here. They are *Set*/*Group* properties:

```
G[1].legend.on()
G[1].legend.symbolwidth=2
G[1].legend.frame.borderspacing=0.2

#or
Legend(1).symbolwidth=2
...
```

Parameters **graphid** (*int*) – the graph id

on()

switch legend box on

Returns

self to allow call chaining as in

`Legend.on(...).setSomethingElse(...)`

Return type *Legend*

off()

switch legend box off

Returns

self to allow call chaining as in

`Legend.off(...).setSomethingElse(...)`

Return type *Legend*

toggle()
toggle legend box visibility

Returns

self to allow call chaining as in

```
Legend.toggle(...).setSomethingElse(...)
```

Return type *Legend*

setText(*font=None*, *linespacing=0.2*)
convenience function to compactly set the text related properties

Parameters

- **font** (*FontStyle*) – the *font* style
- **linespacing** (*float*) – the spacing between *lines*

Returns

self to allow call chaining as in

```
Legend.setText(...).setSomethingElse(...)
```

Return type *Legend*

setGeometry(*position=(0.98, 0.02)*, *origin=(0.98, 0.02)*)
convenience function to compactly set the geometry of the box

Parameters

- **position** (*sequence (list,tuple,..)*) – the *position* of the boxes origin
- **origin** (*sequence (list,tuple,..)*) – the boxes *origin*

Returns

self to allow call chaining as in

```
Legend.setGeometry(...).setSomethingElse(...)
```

Return type *Legend*

setFrame(*frame*)
convenience function to compactly set the *frame* properties

Parameters **frame** (*Frame*) – the *frame* settings

Returns

self to allow call chaining as in

```
Legend.setFrame(...).setSomethingElse(...)
```

Return type *Legend*

setSymbol(*spacing=0.5*, *width=3.0*)
convenience function to compactly set the symbol properties

Parameters

- **spacing** (*float*) – symbol-text *spacing*
- **width** (*float*) – symbol *width*

Returns

self to allow call chaining as in

```
Legend.setSymbol(...).setSomethingElse(...)
```

Return type *Legend*

active
get/set if the legend box's visiblity. A nonzero value is True zero is False.

Type bool

font
get/set the font. The getter returns a reference object such that changing the returned font changes the object which owns font. The setter makes a hard assignment of the font data of the underlying object while not changeing the reference.

Type *FontStyle*

position
the legendbox has an *origin*, which is considered the point in the box which is pinned to a position in the viewport of the graph. Ths position is relative to the viewport. So, `position=(1,0)` puts the origin of the box at the upper right corner of the viewport.

Type 2-tuple or 2-sequence of float

origin
the legendbox has an origin, which is considered the point in the box which is pinned to a *position* in the viewport of the graph. Ths origin is relative to the legendbox. So, `origin=(1,0)` puts the origin of the box at the upper right corner of the box.

Type 2-tuple or 2-sequence of float

symbolspacing
the symbol spacing is the distance between the symbol marker and the legend entry text. It is in units of the font size.

Type float

symbolwidth
the symbol width determines the width of the symbol marker in units of the font size.

Type float

linespacing
the line spacing determines the distance between the individual lines of legend entries in units of the font height. Note, that the font height is taller than the a capital letter. Hence linespacing=0 does not close the visible gap between the lines, negative values do.

Type float

frame
get/set the frame. The getter returns a reference object such that changing the returned frame changes the object which owns frame. The setter makes a hard assignment of the frame data of the underlying object while not changeing the reference.

Type *Frame*

6.1.24 Title

class Title(*graphid*)
Bases: *TextBox*

Title is a *TextBox* and only here for convenience (editor code insert functionality). Access the title of a *Graph* in the following way:

```
G[1].title.text="A title"
#or
Title(1).text="A title"
```

Parameters graphid (*int*) – the graph id

6.1.25 SubTitle

class SubTitle(*graphid*)

Bases: *TextBox*

SubTitle is a *TextBox* and only here for convenience (editor code insert functionality). Access the subtitle of a *Graph* in the following way:

```
G[1].subtitle.on()
G[1].subtitle.text="A subtitle"
#or
SubTitle(1).on().text="A subtitle"
```

Parameters graphid (*int*) – the graph id

6.1.26 XAxisLabel

class XAxisLabel(*graphid*)

Bases: *TextBox*

XAxisLabel is a *TextBox* and only here for convenience (editor code insert functionality). Access the xaxis label of a *Graph* in the following way:

```
G[1].xaxislabel.text="moment [$~m$_B$.]"
#or
XAxisLabel(1).text="moment [$~m$_B$.]"
```

Parameters graphid (*int*) – the graph id

6.1.27 YAxisLabel

class YAxisLabel(*graphid*)

Bases: *TextBox*

YAxisLabel is a *TextBox* and only here for convenience (editor code insert functionality). Access the yaxis label of a *Graph* in the following way:

```
G[1].yaxislabel.text="Energy [eV]"
#or
YAxisLabel(1).text="Energy [eV]"
```

Parameters graphid (*int*) – the graph id

6.1.28 OppositeXAxisLabel

class OppositeXAxisLabel(*graphid*)

Bases: *TextBox*

OppositeXAxisLabel is a *TextBox* and only here for convenience (editor code insert functionality). Access the opposite xaxis label of a *Graph* in the following way:

```
G[1].oppositexaxislabel.on()
G[1].oppositexaxislabel.text="occupation"
#or
OppositeXAxisLabel(1).on().text="occupation"
```

Parameters graphid (*int*) – the graph id

6.1.29 OppositeYAxisLabel

class OppositeYAxisLabel(*graphid*)

Bases: `TextBox`

OppositeYAxisLabel is a `TextBox` and only here for convenience (editor code insert functionality). Access the opposite yaxis label of a `Graph` in the following way:

```
G[1].oppositeyaxislabel.active=True
G[1].oppositeyaxislabel.text="Volume [a$_B$.$x{-0.6}$^$y{-0.4}3$.]"
#or
OppositeYAxisLabel(1).on().text="Volume [a$_B$.$x{-0.6}$^$y{-0.4}3$.]"
```

Parameters graphid (`int`) – the graph id

6.1.30 TextBoxes

class TextBoxes(*graphid*)

The TextBoxes class represents all user defined `text boxes` of the `Graph`. This class can be indexed in a similar way as `Graphs` with IDs which are not necessarily contiguous indices. The standard way to access a textbox would be:

```
G[1].textboxes[2].on().text="(A)"
G[1].textboxes[2].on().frame.off()
G[1].textboxes[2].on().position=(0.1,0.05)
```

One can use the class on it's own as with many others:

```
TextBoxes(1)[2].on()  # same thing as before
```

On can delete a textbox:

```
del G[1].textboxes[2]
#or
#del TextBoxes(1)[2]
```

and one can copy one textbox onto another if ever needed:

```
G[1].textboxes[1].on().text="(a)"
G[2].on()
G[2].view.frame.fill.off()
G[2].textboxes[5]=G[1].textboxes[1];
G[2].textboxes[5].position=(0.2,0.15)
```

Parameters graphid (`int`) – the graph id

__delitem__(*id*)

you can delete a textbox with a certain id:

```
del G[1].textboxes[2]
```

__len__()

`len(G[1].textboxes)` returns the number of textboxes not the highest id

at(*i*)

For index (not id) based iteration, use as in:

```
for i in range(len(G[1].textboxes)):
    G[1].textboxes.at(i).text='T{I}'.format(I=i+1)
```

Parameters **i** (*int*) – textbox index (not id)

Returns textbox at index i

Return type *TextBox*

__getitem__()

`G[1].textboxes[id]` returns the textbox with a certain id:

```
g=G[1]
tbs=g.textboxes
for i in range(1,6,2):
   t=tbs[i].on()
   t.text='Textbox id={0}'.format(t.id)
   t.position=[t.position[0]+i*0.03,t.position[1]+i*0.05]
   t.font.color=0xff
```

`G[1].textboxes[i] <==> G[1].textboxes.__getitem__(i)`

__setitem__()

Assign one textbox to another:

```
G[1].textboxes[2]=G[1].textboxes[1]
G[]1.textboxes[2].position=(0.5,0.5)
```

`G[1].textboxes[i]=x <==> G[1].textboxes.__setitem__(i,x)`

__iter__()

you can iterate over all textboxes:

```
for t in G[1].textboxes:
    print t.id
```

next()

for the iterator interface, see *__iter__*

new()

create and return a new TextBox

Returns a new *TextBox*

len

`G[1].textboxes.len` returns the number of textboxes not the highest id

Type int

lastid

the highest id among all existing textboxes (not the number of textboxes)

Type int

6.1.31 TextBox

class TextBox(*graphid*, *textboxid*)

TextBox represents a text box with *textboxid* in the graph with *graphid*. TextBox IDs are not necessarily ordered. You access a textbox from a *TextBoxes* object:

```
t=G[1].textboxes[2] # textbox 2 in graph 1
# or directly
t=TextBox(1,2)
```

The acces via *TextBoxes* allows deletion.

Use the GUI to set up the line and then use the menu->insert->textboxes-> to get something like:

```
TextBox(1,1).on()\
  .setText("$isymbols: $arrowup, $arrowdown, $arrowleft, "
           "$arrowright, $angstroem, "
        "$infinity $imore italic, $nnormal",
    font=FontStyle(size=12,subscriptscale=0.75,color=0xff))\
  .setFrame(Frame(active=1,borderspacing=0.2,
    rim=LineStyle(style='Solid',width=1,color=0x0,extracolor=True),
    fill=FillStyle(active=True,color=0xffffff,extracolor=True)))\
  .setGeometry(system='View',position=(0.03,0.25),angle=0,
    origin=(0,0),  restriction=None,offsets=[])
```

Parameters

- **`graphid`** (*`int`*) – the graph id
- **`textboxid`** (*`int`*) – the textbox id

`on`()

switch textbox on

Returns

self to allow call chaining as in

`TextBox.on(...).setSomethingElse(...)`

Return type *`TextBox`*

`off`()

switch textbox off

Returns

self to allow call chaining as in

`TextBox.off(...).setSomethingElse(...)`

Return type *`TextBox`*

`toggle`()

toggle textbox visibility

Returns

self to allow call chaining as in

`TextBox.toggle(...).setSomethingElse(...)`

Return type *`TextBox`*

`setText`(*text='some text', font=None*)

convenience function to set text related properties.

Parameters

- **`text`** (*`str`*) – the textbox' *`text`*
- **`font`** (*`FontStyle`*) – the *`font`* style

Returns

self to allow call chaining as in

`TextBox.setText(...).setSomethingElse(...)`

Return type *`TextBox`*

`setFrame`(*frame*)

convenience function to set the frame

Parameters **`frame`** (*`Frame`*) – the *`frame`*

Returns

self to allow call chaining as in

```
TextBox.setFrame(...).setSomethingElse(...)
```

Return type `TextBox`

`setGeometry`(*system='view'*, *position=(0, 0)*, *origin=(0, 0)*, *angle=0.0*, *restriction=None*, *offsets=[]*)

convenience function to set the geometry

Parameters

- **`system`** (*`str`*) – *`coordinate system`*, 'view' or 'world'
- **`position`** (*`sequence (list,tuple,..)`*) – the *`position`*: (x,y)
- **`origin`** (*`sequence (list,tuple,..)`*) – the box *`origin`*: (x,y)
- **`angle`** (*`float`*) – the rotation *`angle`*
- **`restriction`** – *`restrictions`*
- **`offsets`** (*`list`*) – additional *`offsets`*

Returns

self to allow call chaining as in

```
TextBox.setGeometry(...).setSomethingElse(...)
```

Return type `TextBox`

`active`

get/set if the textbox is visible. A nonzero value is True, zero is False.

Type bool

`text`

get/set the actual text of the box. Use *text formating* if needed.

Type str

`id`

the textbox id

Type int

`font`

get/set the font. The getter returns a reference object such that changing the returned font changes the object which owns font. The setter makes a hard assignment of the font data of the underlying object while not changeing the reference.

Type `FontStyle`

`position`

the textbox has an *`origin`*, which is considered the point in the box which is pinned to a position in the viewport of the graph. Ths position is relative to the viewport. So, `position=(0,0)` puts the origin of the box at the upper left corner of the viewport.

Type 2-tuple or 2-sequence of float

`origin`

the textbox has an origin, which is considered the point in the box which is pinned to a *`position`* in the viewport of the graph. Ths origin is relative to the textbox. So, `origin=(0,0)` puts the origin of the box at the upper left corner of the box.

Type 2-tuple or 2-sequence of float

`angle`

the rotation angle of the whole box

Type float

`frame`
get/set the frame. The getter returns a reference object such that changing the returned frame changes the object which owns frame. The setter makes a hard assignment of the frame data of the underlying object while not changeing the reference.

Type *`Frame`*

`coordinatesystem`
the *`position`* of the box is either specified in *`viewport`* coordinates or in *`world`* coordinates. In the latter case the box moves when the world is moved. Possible values: 'view' or 'world'.

Type str

`restriction`
a textbox can have restrictions, which pin it in relation to other objects (e.g. used for the title and axis lables). A restriction is either `None` or a tuple `(str,float)` where the `str` can be 'x'or 'y'. A restriction moves the box from its position into the specified direction by a viewport-relative amount given by the `float` (which can be negative).

Type tuple

`offsets`
additional offsets, which take into account the space occupied by other objects. If the `list` is empty no offsets are considered. Otherwise it can contain any of the following `int`:

1	opposite x tic label height
2	opposite x tic label offset
3	normal x tic label height
4	normal x tic label offset
5	opposite y tic label width
6	opposite y tic label offset
7	normal y tic label width
8	normal y tic label offset
9	subtitle offset
10	subtitle height
11	opposite x-axis label offset
12	opposite x-axis label height

Type list of int

6.1.32 TicMarks

class `TicMarks`(*graphid*, *xy*)
TicMarks represents the default tic marks and tic labels of the *`Graph`* s *`xaxis`* and *`yaxis`*. For user defined *`Tic`* s look into *`Graph.usertics`* and *`Group.usertics`*. You can access individual properties via the object hierarchy:

```
G[1].ytics.labels.decimals=2
G[1].ytics.major.line.color=0xff0000
G[1].ytics.minor.line.width=2
```

To setup handmade tic spacings/subdivisions you need to switch off the auto-tic production:

```
G[1].xtics.auto=False
```

Note, that for logarithmic axis scaling *`labels.decimals`* and *`major.spacing`* will be ignored, since they only make sense for linear scales.

It might probably be the easiest to use the editor insert menu to get and edit this (**After insertion select the whole block and use the edit->indent functionality for proper python indentation.**):

```
G[1].xtics.auto=True
G[1].xtics.side='Normal'
G[1].xtics.setLabels(side='Normal',offset=0.03,decimals=-1,
 font=FontStyle(size=16,subscriptscale=0.75,color=0x0))\
 .setMajor(active=1,spacing=1,length=0.02,separatelength=0,
 line=LineStyle(style='Solid',width=1,color=0x0,extracolor=True))\
 .setMinor(active=1,subdiv=2,length=0.01,separatelength=0,
 line=LineStyle(style='Solid',width=1,color=0x0,extracolor=True))
G[1].ytics.auto=True
G[1].ytics.side='Normal'
G[1].ytics.setLabels(side='Normal',offset=0.03,decimals=-1,
 font=FontStyle(size=16,subscriptscale=0.75,color=0x0))\
 .setMajor(active=1,spacing=1,length=0.02,separatelength=0,
 line=LineStyle(style='Solid',width=1,color=0x0,extracolor=True))\
 .setMinor(active=1,subdiv=2,length=0.01,separatelength=0,
 line=LineStyle(style='Solid',width=1,color=0x0,extracolor=True))
```

Parameters

- **graphid** (*int*) – the graph id
- **xy** (*str*) – axis id ‘x’ or ‘y’

setLabels (*side=’normal’*, *offset=0.03*, *decimals=-1*, *font=None*)

convenience function to set the tic label properties. This is another way of doing the following:

```
G[1].xtics.labels.side='both'
G[1].xtics.labels.offset=0.02
G[1].xtics.labels.decimals=2
G[1].xtics.labels.font.color=0xff #...
```

Parameters

- **side** (*str*) – ‘none’, ‘normal’, ‘opposite’ or ‘both’ (*TicLabels.side*)
- **offset** (*float*) – see *TicLabels.offset*
- **decimals** (*int*) – see *TicLabels.decimals*
- **font** (*FontStyle*) – see *TicLabels.font*

Returns

self to allow call chaining as in

`TicMarks.setLabels(...).setSomethingElse(...)`

Return type *TicMarks*

setMajor (*active=1*, *spacing=1.0*, *length=0.02*, *separatelength=False*, *line=None*)

convenience function to set the major tic mark properties. This is another way of doing the following:

```
G[1].xtics.auto=False
G[1].xtics.major.spacing=2
G[1].xtics.major.length=0.05
```

Parameters

- **active** (*int*) – see *TicMajor.active*
- **spacing** (*float*) – see *TicMajor.spacing*
- **length** (*float*) – see *TicMajor.length*

- **separatelength** (*int*) – see *TicMajor.separatelength*
- **line** (*LineStyle*) – see *TicMajor.line*

Returns

self to allow call chaining as in

`TicMarks.setMajor(...).setSomethingElse(...)`

Return type *TicMarks*

setMinor (*active=1, subdiv=2, length=0.01, separatelength=False, line=None*)

convenience function to set the minor tic mark properties. This is another way of doing the following:

```
G[1].xtics.auto=False
G[1].xtics.minor.subdiv=5
G[1].xtics.minor.length=0.03
```

Parameters

- **active** (*int*) – see *TicMinor.active*
- **subdiv** (*int*) – see *TicMinor.subdiv*
- **length** (*float*) – see *TicMinor.length*
- **separatelength** (*int*) – see *TicMinor.separatelength*
- **line** (*LineStyle*) – see *TicMinor.line*

Returns

self to allow call chaining as in

`TicMarks.setMinor(...).setSomethingElse(...)`

Return type *TicMarks*

auto

get/set if the tic mark positions shall be automatically determined.

Type bool

side

get/set the side on which to plot the tics. Can be 'none', 'normal', 'opposite' and 'both'

Type str

labels

get/set a TicLabels object which gives access to the label properties.

This is intended for hierarchical use to change a few properties only as in:

```
G[1].ytics.labels.decimals=2
```

Type *TicLabels*

major

get/set a TicMajor object which gives access to the major tic mark properties.

This is intended for hierarchical use to change a few properties only as in:

```
G[1].ytics.major.line.color=0xff0000
```

Type *TicMajor*

minor

get/set a TicMinor object which gives access to the minor tic mark properties.

This is intended for hierarchical use to change a few properties only as in:

```
G[1].ytics.minor.line.width=2
```

Type *TicMinor*

6.1.33 TicMajor

class TicMajor(*graphid*, *xy*)

TicMajor is returned by *G[1].xtics.major* or *G[1].ytics.major* and allows hierarchical access:

```
G[1].xtics.major.line.color=0xff
```

Parameters

- **graphid** (*int*) – the graph id
- **xy** (*str*) – axis id ‘x’ or ‘y’

on()

switch major tics/labels on

Returns

self to allow call chaining as in

`TicMajor.on(...).setSomethingElse(...)`

Return type *TicMajor*

off()

switch major tics/labels off

Returns

self to allow call chaining as in

`TicMajor.off(...).setSomethingElse(...)`

Return type *TicMajor*

toggle()

toggle major tics/labels visibility

Returns

self to allow call chaining as in

`TicMajor.toggle(...).setSomethingElse(...)`

Return type *TicMajor*

active

get/set if the major tics and tic labels are visible. A nonzero value is True, zero is False.

Type bool

spacing

get/set the spacing between major tics in *world* units. This takes effect, if *TicMarks.auto* is switched off. Note, that for logarithmic axis scaling the spacing will be ignored.

Type float

length
get/set the major tic's length in relative viewport units. (1 means the whole viewport width/height). See *separatelength*.

Type float

separatelength
(get/set) normally this is `False` such that the physical tic length of both axes is the same. In this case the xaxis determines the tic length. If *separatelength* is `True`, the tic length for both axes can be set separately. Note, that due to bad design both x- and y-tics have this property. So, if you have:

```
G[1].xtics.major.separatelength=True
G[1].ytics.major.separatelength=False
```

in your script, the second line will win.

Type bool

line
get/set the *line* (LineStyle). The getter returns a reference object such that changing it changes the *TicMajor*-object which owns *line*. The setter makes a hard assignment of the *TicMajor*-objects data. The reference of the assigned *LineStyle* stays intact.

Type *LineStyle*

6.1.34 TicMinor

class TicMinor(*graphid*, *xy*)
TicMinor is returned by *G[1].xtics.minor* or *G[1].ytics.minor* and allows hierarchical access:

```
G[1].xtics.minor.line.color=0xff
```

Parameters

- **graphid** (*int*) – the graph id
- **xy** (*str*) – axis id 'x' or 'y'

on()
switch minor tics/labels on

Returns

self to allow call chaining as in

`TicMinor.on(...).setSomethingElse(...)`

Return type *TicMinor*

off()
switch minor tics/labels off

Returns

self to allow call chaining as in

`TicMinor.off(...).setSomethingElse(...)`

Return type *TicMinor*

toggle()
toggle minor tics/labels visibility

Returns

self to allow call chaining as in

```
TicMinor.toggle(...).setSomethingElse(...)
```

Return type *TicMinor*

active
get/set if the minor tics and tic labels are visible. A nonzero value is True, zero is False.

Type bool

subdiv
get/set the number of subdivions-intervals between major tics between which to plot a minor tic.

Type int

length
get/set the major tic's length in relative viewport units. (1 means the whole viewport width/height). See *separatelength*.

Type float

separatelength
(get/set) normally this is `False` such that the physical tic length of both axes is the same. In this case the xaxis determines the tic length. If *separatelength* is `True`, the tic length for both axes can be set separately. Note, that due to bad design both x- and y-tics have this property. So, if you have:

```
G[1].xtics.minor.separatelength=True
G[1].ytics.minor.separatelength=False
```

in your script, the second line will win.

Type bool

line
get/set the *line* (LineStyle). The getter returns a reference object such that changing it changes the *TicMinor*-object which owns *line*. The setter makes a hard assignment of the *TicMinor*-objects data. The reference of the assigned *LineStyle* stays intact.

Type *LineStyle*

6.1.35 TicLabels

class TicLabels(*graphid*, *xy*)
TicLabels is returned by *G[1].xtics.labels* or *G[1].ytics.labels* and allows hierarchical access:

```
G[1].xtics.labels.decimals=2
```

Parameters

- **graphid** (*int*) – the graph id
- **xy** (*str*) – axis id 'x' or 'y'

side
get/set the side on which to plot tic labels. It can be 'none', 'normal', 'opposite' and 'both'

Type str

offset
get/set the offset between the tic labels and the axis in relative viewport units.

Type int

decimals
get/set how many decimals will be plotted. -1 means automatic. Note, that for logarithmic axis scaling *decimals* will be ignored.

Type int

font

get/set the font. The getter returns a reference object such that changing the returned font changes the object which owns font. The setter makes a hard assignment of the font data of the underlying object while not changeing the reference.

Type *FontStyle*

6.1.36 UserTics

class UserTics(*graphid*, *groupid=0*)

UserTics allow to define irregular individual tic marks/labels. They can be *Graph* specific: *Graph.usertics* and *Group* specific: *Group.usertics*. The difference is that their visibility is coupled to the respective parent object's visibility.

Unfortunately, tics are indexed by zero based indices as normal lists in contrast to all other objects in *pyxfbp*. The easiest way to deal with them when creating is:

```
G[1].usertics.clear().on()\
     .append(Tic(label="G",position=0.2,length=1,type='xmajor'))\
     .append(Tic(label="X",position=0.6,length=1,type='xmajor'))\
     .append(Tic(label="M",position=0.9,length=1,type='xmajor'))
```

When changing existing tics do this (you will anyway have looked at the GUI to figure out which tics where created by a *read* command):

```
G[1].usertics[1].label="P" # change label

del G[1].usertics[5] # dont want this
```

Some more explanantions: usertics[2] is the 2nd tic and not the tic with id 2 as for graphs… Deleting tics changes the index of all tics with higher index by -1. Furthermore, if we have reference objects they are now dangling:

```
G[1].usertics.clear().on()\
     .append(Tic(label="G",position=0.2,length=1,type='xmajor'))\
     .append(Tic(label="X",position=0.6,length=1,type='xmajor'))\
     .append(Tic(label="M",position=0.9,length=1,type='xmajor'))

t=G[1].usertics[0]
t.label="GG"
del G[1].usertics[0]

t.line.color=0xff0000 # this is allowed


for u in G[1].usertics:
    print u.label

# now t is a dangling Tic. You can change it without any visible
# changes in the GUI, well it's dangling.
# But you can assign it to another tic.

G[1].usertics.new()
G[1].usertics[2]=t # assign to new 3rd tic

# Now the tics are there in a different order

for u in G[1].usertics:
    print u.label
```

Parameters

- **graphid** (*int*) – the graph id
- **groupid** (*int*) – the group id

__delitem__(*i*)
you can delete a tic with index i:

```
del G[1].usertics[2]
```

on()
switch usertics on

Returns

self to allow call chaining as in

`UserTics.on(...).setSomethingElse(...)`

Return type *UserTics*

off()
switch usertics off

Returns

self to allow call chaining as in

`UserTics.off(...).setSomethingElse(...)`

Return type *UserTics*

toggle()
toggle usertics visibility

Returns

self to allow call chaining as in

`UserTics.toggle(...).setSomethingElse(...)`

Return type *UserTics*

__len__()
`len(G[1].usertics)` returns the number of usertics

__getitem__()
`G[1].usertics[i]` returns the usertic with index i

`G[1].usertics[i] <==> G[1].usertics.__getitem__(i)`

__setitem__()
Assign one usertic to another:

```
G[1].usertics[2]=G[1].usertics[1]
G[1].usertics[2].position=0.67
G[1].usertics[2].label="M"
```

`G[1].usertics[i]=x <==> G[1].usertics.__setitem__(i,x)`

__iter__()
you can iterate over all usertics:

```
for t in G[1].usertics:
    t.line.color=0xff00
```

next()
for the iterator interface, see *__iter__*

clear()
delete all usertics

Returns

self to allow call chaining as in

```
UserTics.clear(...).setSomethingElse(...)
```

Return type *UserTics*

new()
create and return a new Tic

Returns a new *Tic*

append(*tic*)
append *Tic* to the list of usertics.

Parameters tic (*Tic*) – a tic object

Returns

self to allow call chaining as in

```
UserTics.append(...).setSomethingElse(...)
```

Return type *UserTics*

active
get/set if these usertics (Graph or Group owned) are visible. A nonzero value is True, zero is False.

Type bool

6.1.37 Tic

class Tic(*type='xmajor'*, *position=0*, *length=0.02*, *label='label'*, *ticside='normal'*, *labelside='normal'*, *labeloffset=0.03*, *line=None*)
The Tic object is used to either append a new Tic to *UserTics* or to follow the object hierarchical:

```
G[1].usertics.\
  append(Tic(type='xmajor',position=0.1,label='M',
         ticside='Normal',labelside='Normal',length=1,labeloffset=0.03,
         line=LineStyle(style='Solid',width=1,color=0x0,extracolor=True)))


G[1].usertics[0].label='M' #  here usertics[1] returend a Tic object
```

Parameters

- **type** (*str*) – 'xmajor', 'xminor', 'ymajor' or 'yminor' *type*
- **position** (*float*) – see *position*
- **length** (*float*) – see *length*
- **label** (*str*) – see *label*
- **ticside** (*str*) – 'none', 'normal', 'opposite' and 'both', see *ticside*
- **labelside** (*str*) – 'none', 'normal', 'opposite' and 'both', see *labelside*
- **labeloffset** (*float*) – see *labeloffset*
- **line** (*LineStyle*) – see *line*

type
get/set the type of the tic, 'xminor', 'xmajor', 'yminor' or 'ymajor'

Type str

`position`
get/set the tic position in world units.

Type float

`label`
get/set the tic label.

Type str

`ticside`
get/set the tic side, 'none', 'normal', 'opposite' and 'both'

Type str

`labelside`
get/set the label side, 'none', 'normal', 'opposite' and 'both'

Type str

`length`
get/set the tic length in relative viewport units

Type float

`labeloffset`
get/set the label offset between the tic labels and the axis in relative viewport units.

Type float

`line`
get/set the linestyle. The getter returns a reference object such that changing it changes the `Tic`-object which owns *line*. The setter makes a hard assignment of the `Tic`-objects data. The reference of the assigned `LineStyle` stays intact.

Type `LineStyle`

6.1.38 Lines

`class Lines`(*graphid*)
Lines represent all `line` shapes in a `Graph`. Lines work ID-based, the same way as `Graphs`. You can delete line shapes and copy them onto each other:

```
G[1].lines[1].on()
# do some settings
G[1].lines[5]=G[1].lines[1]
del G[1].lines[1]
```

You can directly access Lines via:

```
lns=Lines(1)
lns[1].on()
```

Parameters **`graphid`** (*`int`*) – the graph id

`__delitem__`(*id*)
you can delete a line shape with a certain id:

```
del G[1].lines[2]
```

`__len__`()
`len(G[1].lines)` returns the number of line shapes not the highest id

at (*i*)

For index (not id) based iteration, use as in:

```
for i in range(len(G[1].lines)):
    G[1].lines.at(i).line.color=0xff
```

Parameters **i** (*int*) – line shape index (not id)

Returns line shape at index i

Return type *Line*

__getitem__()

`G[1].lines[id]` returns the line shape with a certain id:

```
g=G[1]
lns=g.lines
for i in range(1,6,2):
   t=lns[i].on()
   t.line.color=0xff0000
   t.startat=(0.2,0.2)
   t.endat=[t.endat[0]+i*0.03,t.endat[1]+i*0.0]
```

`G[1].lines[i] <==> G[1].lines.__getitem__(i)`

__setitem__()

Assign one line shape to another:

```
G[1].lines[2]=G[1].lines[1]
G[1].lines[2].startat=(0.9,0.5)
```

`G[1].lines[i]=x <==> G[1].lines.__setitem__(i,x)`

__iter__()

you can iterate over all line shapes:

```
for l in G[1].lines:
    l.line.color=0xff00
```

next()

for the iterator interface, see *__iter__*

new()

create and return a new Line

Returns a new *Line*

len

`G[1].lines.len` returns the number of line shapes not the highest id

lastid

the highest id among all existing line shapes (not the number of line shapes)

Type int

6.1.39 Line

class Line(*graphid*, *lineid*)

Line is a single line shape with optional arrows belonging to a particular *graph*. Access lines through the *lines* object or directly:

```
l=G[1].lines[4].on() # graph1 line4 now exists and is visible and
                     # assigned to variable l
Line(1,4).on()       # does the same thing

G[1].lines[1]=G[1].lines[4] # now we have two lines

del G[1].lines[4] # and delete the first

l.line.color=0xff0000  # error: line4 does not exist, we deleted it

# but

l.on().line.color=0xff0000 # back alive
```

Use the GUI to set up the line and then use the menu->insert->shapes to get something like:

```
Line(1,1).on().setName('')\
 .setGeometry(startat=(0.2,0),endat=(0.8,1),system='View')\
 .setLine(capat='None',style=LineStyle(style='Solid',width=1,\
     color=0x0,extracolor=True))\
 .setArrow(at='End',style='Closed',size=16,sharpness=2,
     fill=FillStyle(active=False,color=0x0,extracolor=True))
```

Parameters

- **graphid** (*int*) – graph owning the line
- **lineid** (*int*) – line shape id

on()

switch line shape on

Returns

self to allow call chaining as in

`Line.on(...).setSomethingElse(...)`

Return type *Line*

off()

switch line shape off

Returns

self to allow call chaining as in

`Line.off(...).setSomethingElse(...)`

Return type *Line*

toggle()

toggle line shape visibility

Returns

self to allow call chaining as in

`Line.toggle(...).setSomethingElse(...)`

Return type *Line*

setArrow(*at='end'*, *style='closed'*, *size=16*, *sharpness=2*, *fill=None*)

convenience function to set arrow related properties.

Parameters

- **at** (*str*) – where does the arrow sit (*arrowat*). 'none', 'end', 'start' or 'both'

- **style** (*str*) – 'open' or 'closed' (*arrowstyle*)
- **size** (*float*) – the *arrowsize*
- **sharpness** (*float*) – the *arrowsharpness*
- **fill** (*FillStyle*) – the *arrowfill*

Returns

self to allow call chaining as in

```
Line.setArrow(...).setSomethingElse(...)
```

Return type *Line*

setName (*name*)

convenience function to set the line shape *name*

Parameters **name** (*str*) – a name

Returns

self to allow call chaining as in

```
Line.setName(...).setSomethingElse(...)
```

Return type *Line*

setGeometry (*startat=(0.2, 0), endat=(0.8, 1), system='view'*)

convenience function to set the line geometry

Parameters

- **startat** (*2-sequence (tuple,list,..)*) – see *startat*
- **endat** (*2-sequence (tuple,list,..)*) – see *endat*
- **system** (*str*) – 'view' or 'world' (*coordinatesystem*)

Returns

self to allow call chaining as in

```
Line.setGeometry(...).setSomethingElse(...)
```

Return type *Line*

setLine (*capat='none', style=None*)

convenience function to set the line's properties

Parameters

- **capat** – 'none', 'end', 'start' or 'both' (*capat*)
- **style** (*LineStyle*) – see *line*

Returns

self to allow call chaining as in

```
Line.setLine(...).setSomethingElse(...)
```

Return type *Line*

id

the line shape's id

Type int

active

get/set if the line shape is visible. A nonzero value is True, zero is False.

Type bool

name
get/set a name for the line shape for easier orientation

Type str

line
get/set the linestyle. The getter returns a reference object such that changing it changes the `Line`-object which owns *line*. The setter makes a hard assignment of the `Line`-objects data. The reference of the assigned `LineStyle` stays intact.

Type `LineStyle`

coordinatesystem
get/set/ the coordinate system. `startat` and `endat` of the line shape are either specified in `viewport` coordinates or in `world` coordinates. In the latter case the line moves when the world is moved. Possible values: 'view' or 'world'.

Type str

startat
get/set the line starting position:

```
l=G[1].lines[1].on()
l.coordinatesystem='world'
l.startat=(0.1,0.2)
l.endat=(0.8,0.6)
G[1].world.setX(-1,1.5).setY(-0.5,1.3)
# the line is still at the intended coordinates
```

Type 2-tuple or 2-sequence of float

endat
get/set the line end position.

Type 2-tuple or 2-sequence of float

capat
get/set the line cap. A line can have a cap at either end. A cap is a little round thing, which makes the line end look smoother. Possible options are:

'none', 'end', 'start' or 'both'

Type str

arrowat
get/set the arrow. A line can have an arrow at either end. Possible options are:

'none', 'end', 'start' or 'both'

Type str

arrowstyle
get/set the arrow style. There are two styles: 'open' and 'closed'.

Type str

arrowsize
get/set the arrow size in point units (as font size).

Type float

arrowsharpness
get/set the arrow sharpness. This is accuteness of the opening.

Type float

arrowfill

get/set the *arrowfill* (`FillStyle`). If `arrowfill.active` is `False/0` and the style is 'closed' a full arrow head with the `line` color is drawn, if `arrowfill.active` is `True/!=0` the arrow head has a separate fill color.

The getter returns a reference object such that changing it changes the `Line`-object which owns *arrowfill*. The setter makes a hard assignment of the `Line`-objects data. The reference of the assigned `FillStyle` stays intact.

6.1.40 Ellipses

class Ellipses(*graphid*)

Ellipses represent all `ellipse` shapes in a `Graph`. Ellipses work ID-based, the same way as `Graphs`. You can delete ellipse shapes and copy them onto each other:

```
G[1].ellipses[1].on()
# do some settings
G[1].ellipses[5]=G[1].ellipses[1]
del G[1].ellipses[1]
```

You can directly access Ellipses via:

```
els=Ellipses(1) # all ellipses of graph with id 1
els[1].on()
```

Parameters graphid (`int`) – the graph id

__delitem__(*id*)

you can delete an ellipse shape with a certain id:

```
del G[1].ellipses[2]
```

__len__()

`len(G[1].ellipses)` returns the number of ellipse shapes not the highest id

at(*i*)

For index (not id) based iteration, use as in:

```
for i in range(len(G[1].ellipses)):
    G[1].ellipses.at(i).line.color=0xff
```

Parameters i (`int`) – ellipse shape index (not id)

Returns ellipse shape at index i

Return type `Ellipse`

__getitem__()

`G[1].ellipses[id]` returns the ellipse shape with a certain id:

```
g=G[1]
els=g.ellipses
for i in range(1,6,2):
   e=els[i].on()
   e.line.color=0xff0000
   e.radii=0.03*(i+1)
   e.center=(e.center[0],e.center[1]+i**2*0.01)
```

`G[1].ellipses[i] <==> G[1].ellipses.__getitem__(i)`

`__setitem__``()`
Assign one ellipse shape to another:

```
G[1].ellipses[2]=G[1].ellipses[1]
G[1].ellipses[2].center=(0.9,0.5)
```

```
G[1].ellipses[i]=x <==> G[1].ellipses.__setitem__(i,x)
```

`__iter__``()`
you can iterate over all ellipse shapes:

```
for l in G[1].ellipses:
    l.line.color=0xff00
```

`next``()`
for the iterator interface, see *`__iter__`*

`new``()`
create and return a new ellipse

Returns a new *`Ellipse`*

`len`
`G[1].ellipses.len` returns the number of ellipse shapes not the highest id

`lastid`
the highest id among all existing ellipse shapes (not the number of ellipse shapes)

Type int

6.1.41 Ellipse

class `Ellipse`(*graphid*, *ellipseid*)
Ellipse is a single ellipse shape belonging to a particular *`graph`*. Access ellipses through the *`ellipses`* object or directly:

```
e=G[1].ellipses[4].on() # graph1 ellipse4 now exists and is visible and
                        # assigned to variable e
Ellipse(1,4).on()       # does the same thing

G[1].ellipses[1]=G[1].ellipses[4] # now we have two ellipses

del G[1].ellipses[4] # and delete the first

e.line.color=0xff0000  # error: ellipse4 does not exist, we deleted it

# but

e.on().line.color=0xff0000 # back alive
```

Use the GUI to set up the ellipse and then use the menu->insert->shapes to get something like:

```
Ellipse(1,4).on().setName('')\
 .setGeometry(center=(4.47214,60),radii=(1.11474,18),angle=0,system='World')\
 .setLine(LineStyle(style='Dot',width=4,color=0xff0000,extracolor=True))\
 .setFill(FillStyle(active=True,color=0xaa00,extracolor=True))
```

Parameters

- **`graphid`** (*`int`*) – graph owning the ellipse
- **`ellipseid`** (*`int`*) – ellipse shape id

on()
switch ellipse shape on

Returns

self to allow call chaining as in

```
Ellipse.on(...).setSomethingElse(...)
```

Return type `Ellipse`

off()
switch ellipse shape off

Returns

self to allow call chaining as in

```
Ellipse.off(...).setSomethingElse(...)
```

Return type `Ellipse`

toggle()
toggle ellipse shape visibility

Returns

self to allow call chaining as in

```
Ellipse.toggle(...).setSomethingElse(...)
```

Return type `Ellipse`

setName(*name*)
convenience function to set the ellipse shape `name`

Parameters **name** (`str`) – a name

Returns

self to allow call chaining as in

```
Ellipse.setName(...).setSomethingElse(...)
```

Return type `Ellipse`

setGeometry(*center=(0, 0)*, *radii=(0.1, 0.2)*, *angle=0.0*, *system='view'*)
convenience function to set the line geometry

Parameters

- **center** (`2-sequence (tuple,list,..)`) – see `center`
- **radii** (`float or 2-sequence (tuple,list,..)`) – see `radii`
- **angle** (`float`) – see `angle`
- **system** (`str`) – 'view' or 'world' (`coordinatesystem`)

Returns

self to allow call chaining as in

```
Ellipse.setGeometry(...).setSomethingElse(...)
```

Return type `Ellipse`

setLine(*style*)
convenience function to set the ellipse's linestyle

Parameters **style** (`LineStyle`) – : see `line`

Returns

self to allow call chaining as in

```
Ellipse.setLine(...).setSomethingElse(...)
```

Return type *Ellipse*

setFill (*fill*)
convenience function to set the ellipse's fillstyle

Parameters **fill** (*FillStyle*) – : see *fill*

Returns

self to allow call chaining as in

```
Ellipse.setFill(...).setSomethingElse(...)
```

Return type *Ellipse*

id
the ellipse shape's id

Type int

active
get/set if the ellipse shape is visible. A nonzero value is True, zero is False.

Type bool

name
get/set a name for the ellipse shape for easier orientation

Type str

line
get/set the linestyle. The getter returns a reference object such that changing it changes the *Ellipse*-object which owns *line*. The setter makes a hard assignment of the *Ellipse*-objects data. The reference of the assigned *LineStyle* stays intact.

Type *LineStyle*

fill
get/set the fillstyle. The getter returns a reference object such that changing it changes the *Ellipse*-object which owns *fill*. The setter makes a hard assignment of the *Ellipse*-objects data. The reference of the assigned *FillStyle* stays intact.

Type *FillStyle*

center
get/set the ellipse starting position:

```
e=G[1].ellipses[1].on()
e.coordinatesystem='world'
e.center=(0.1,0.2)
e.radii=(0.2,0.1)
G[1].world.setX(-1,1.5).setY(-0.5,1.3)
# the ellipse is still at the intended coordinates
```

Type 2-tuple or 2-sequence of float

radii
get/set radii/radius. The 2-sequence (tuple,list,...) denotes [x-radius,y-radius]. If a circle is needed just us a single float (no sequence):

```
ellipse.radii=0.2 # a circle
```

Type float or 2-tuple of float

`angle`
get/set rotation angle of the ellipse

Type float

`coordinatesystem`
get/set/ the coordinate system. *`center`* and *`radii`* of the ellipse shape are either specified in *`viewport`* coordinates or in *`world`* coordinates. In the latter case the ellipse moves when the world is moved. Possible values: 'view' or 'world'.

Type str

CHAPTER

SEVEN

XFBP

Author Klaus Koepernik

7.1 Python scripting

If the program was compiled with python support the *pyxfbp help* is displayed when *F1* is hit in the script/transform dialog if the editor is set to python mode. The python scripts should have the extension `.xpy` since they do not work in other python shells.

7.2 Native Scripting

The Transform Dialog is actually a *script editor*, where **xfbp** commands from the file description language and some more commands can be used. For python mode go *here*.

The commands can be saved to file or loaded from file (extension `.cmd`). To get familiar with the commands you can use the *insert* functionality of the editor. We also recommend to look at the `.xfp` files: save the current plot (`.xfp`) (not the current script) and look at the `.xfp` file.

We try to describe the scripting language in the following, including examples. To copy these examples into the script editor, select and copy them via *Ctrl-C* and paste them (*Ctrl-V*) into the script editor.

The language is made of statements. Each statement is a single line.

In the following optional parts of a statement are enclosed in square brackets `[]`. A construct like `(a|b|c|...)` means that at this position in the command either `a` or `b` or `c ...` can be used. If parentheses `(` and `)` appear without a `|` between them they are literals, which means you have to type them (example: `sin(x)`). They will be set in bold face in the command description. Similarly square brackets `[` and `]` can appear in vector element constructs (example: `s1.x[6]=12.5`).

In the following key words are denoted by **keyword**. They are not case sensitive. Values are denoted by *value*. There are different values:

- sub commands like *linestylecommands*
- expressions like *exp*, which can be numbers, variables, parameters and some other constructs. They evaluate to scalars or vectors

- *strings*, which are used for file names and text. They must be enclosed within double quotes, example:

```
read xny "+dos.total"
read bandweight "+bweights"
```

- *parameters* are values defined on the command line via option *-a* and are available in the script editor via *$parametername*. Example: Create a command file `t.cmd` with the content:

```
kill all
read xny $pp
with g1.gr1.s1
line width $w
legend $leg
title "Density of states"
```

  From the command line call:

```
xfbp t.cmd -a pp:"+dos.sort001" -a leg:"Fe" -a w:2
```

  Of course, you have to make sure that there is actually a file `+dos.sort001` for this to work.

Important: when graphs, groups, sets or weights are referenced in the script, they will be made current, which means that they will be remembered until the next such explicit reference makes another graph… the current object. This allows to skip these specifiers in many contexts.

Content

7.2.1 Comments

- # blah blah

  Full line comment. There are no inline comments!

top

7.2.2 File loading

- **read** *filetype* (*string* | *parameter*) [(**into** (*graphdesc* | *groupdesc*) | **into new graph**)]

  Read file with name *string* or with its name provided in a *parameter* into current graph (or a specific *graph/group*) enforcing data type *filetype*. (*string* | *parameter*) is something like "myfile.dat". Band structure data cannot be read into specified groups, since they are organized into groups by the programm.

  Examples:

```
# killall and initialize graph 1
killall
# read band strcture into current graph (graph 1)
read band "+band"
# read two files into graph 2
read xny "+dos.total" into g2
read xny "+dos.sort001" into g2.gr5
```

top

7.2.3 Print commands

Currently there is only two options to print: `png` and `eps` files. The printed `eps` format is not yet fully *eps* standard, but works, especially embedded in LaTeX.

- **print to** (*string* | *parameter*) [**dpi** *int* (**quality** *real*) | **quality** *real* (**dpi** *int*)]

  export plot to file named *string* in `eps` or `png` format (the file extension determines which). Example:

  ```
  print to "bands.png" dpi 300 quality 0.95
  ```

- **print filename** (*string* | *parameter*)

  set the the file name for print commands to *string* or *parameter*.

- **print to file**

  export plot in `eps` or `png` format to file. (The filename must have been defined before.)

top

7.2.4 Paper commands

- **paper size** *int* , *int*

  set paper size from width and height (integer). Example:

  ```
  paper size 400,200
  ```

- **paper size** *papersizes*

  set paper size from paper size names, e.g. a1. . . a10, letter, lettersmall, legal, statement, tabloid, ledger, folio, quarto. Example:

  ```
  paper size a4
  paper orientation portrait
  ```

- **paper orientation** (**portrait** | **landscape**)

  set orientation. Only for predefined paper sizes.

- **arrange** (*paperwidth* , *Nx* , *xgap0* , *xgap1* , *deltawidth* , *Ny* , *ygap0* , *ygap1* , *deltaheight* , *aspectratio*, *commonxaxis*, *commonyaxis*, *commontitle*)

  arrange *Nx* * *Ny* graphs on a grid. The graphs can already exist. Otherwise they are created. The index runs row by row. *xgap0*/*xgap1* are the spaces left/right of the first/last viewbox in percent/100 of the paper width. Similarily *ygap0*/*ygap1*. *deltawidth*/*deltaheight* are the in-between gaps in percent/100 of the individual viewbox width/height. *aspectratio* determines the individual viewbox aspect ratio. Note, that the point scale of the graphs depends on the paperwidth. You need to experiment with the gap values a bit to get the labels/ticmarks/titles properly displayed in the page. If *commonxaxis* is nonzero the in-between-viewboxes tic labels and xaxis labels are switched off and in each column the world x axis is made equivalent. Similarily, for *commonyaxis*. For this to work the arrange command must be issued after setting up the graphs. If *commontitle* is nonzero the titles in between rows are switched off:

```
arrange(800,   3,0.12,0.03,0.1,   2,0.12,0.15,0.1,    1.0,   1,1,1)
```

top

7.2.5 Graph/Group/Set/Weight descriptors

Data belong to graphs and are organized in groups and sets. *graph/group/set/weight* s have descriptors, which are explained here. Note, that there is the concept of a current *graph/group/set/weight*. The current object is memorized after a descriptor appeared in the script until another object becomes current by the appearance of another descriptor . Hence we can write:

```
g1.gr2.s1 line color 0xff
line width 2.5
line symbol ...
```

to change the settings of set 1 in group2 in graph 1.

- **g***int*

  graph *int*: g4 is graph 4.

- **gr***int*

  group *int* in the current graph (if valid)

- **s***int*

  set *int* in the current graph and current group

- **g***int1* . **gr***int2*

  group *int2* in graph *int1*

- [[**g***int1* .]**gr***int2* .]**s***int3*

  set *int3* in current group or group *int2* in current graph or graph *int1*

- [(*groupdesc* | *setdesc*) .](**x** | **y**)

  vector of x/y values in groupdesc/setdesc or current group/set. Example: `g1.gr1.x=x+1`, will increase all x values of all sets in `g1.gr1` by one. The full command would be `g1.gr1.x=g1.gr1.x+1`, but this is not necessary. And `g1.gr1.s2.y=y*2`, will multiply all y values of set `g1.gr1.s2` by two.

Note, that each descriptor resets the higher level to invalid, such that following works:

```
g1.gr1.s3 line color 0x00
# now  current set is s3, current group is gr1
line width 2.4
# it is still set 3
g1.gr1.x=mesh(...)
# after g1.gr1 the current set is invalid and
# hence all x of group1 get assigned a mesh
line width 2.4
# now we just set the line width for all sets in group 1
```

Each set can have a number of weights associated with them. If all sets of a group refer to the same set of weights (same set of weight labels.) the group can be used to manipulate the weight appearance. (bandweight plots)

- [(*groupdesc* | *setdesc*) .]**w***int*

  weight number *int* in groupdesc/setdesc or current group or set (whichever was defined/refered-to previously). Example:

  ```
  g1.gr1.w8 off
  # turns off weight 8 in group 1.
  ```

- [(*groupdesc* | *setdesc*) .]**w***string*

  weight with name *string* in specified or current group/set. The names must match the existing weight labels exactly. Example:

  ```
  w"Cu(001)3s+0" off.
  ```

top

7.2.6 With command

The current *graph/group/set/weight* descriptors can be set without reference to a particular commmand.

- **with** (*graphdesc* | *groupdesc* | *setdesc* | *weightdesc*)

  set the current . Example:

  ```
  with g1.gr1.s3
  line color 0xff0000
  symbol style circle
  # now weigths with implicit with (remembered from the last weightdescriptor)
  w1 off
  w"Cu(001)3d+0" on
  # set this weights color
  color 0xff0000
  # and the skippage
  skip 5
  ```

top

7.2.7 World commands

World commands refer to a certain graph, which either is specified explicitly or was set as current graph before.

- [*graphdesc*] **world** (**xmin** | **xmax** | **ymin** | **ymax**) *exp*

  set world $x_{\min}$, $x_{\max}$, $y_{\min}$ or $y_{\max}$ for *graph*. Example:

  ```
  world xmin -1
  ```

- [*graphdesc*] **world** (**x** | **y**) *exp* , *exp*

  set the x or y interval of the world coordinates. Example:

  ```
  world x -1,10
  ```

- **autoscale** [(**x** | **y**)]

  autoscale all, x-only or y-only. Example:

  ```
  autoscale x
  ```

- **autoscale offset** *exp* , *exp*

  define the autoscale offset for both directions. This is the space in percent*100 of the total world interval, which will be added on both sides of the intervall. If non-zero, an autoscale will leave the specified percentage of space at either side of the plotted curves.

The current settings can be loaded from the insert menu in the script editor.

top

7.2.8 View commands

The view is the area spanned by the frame of the graph. It is defined in relative coordinates with respect to the paper. A view of width 1 and x-origin 0 would produce a graph frame spanning the whole breadth of the paper. In the following graphdesc can be left out if previously a graph was referenced (**with g***int*, or other references).

- [*graphdesc*] **view** *exp* , *exp* , *exp* , *exp*

  set x-origin, width, y-origin and height of the view. x,y=(0,0) is the upper left corner and x,y=(1,1) the lower right corner. *exp* must evaluate to a scalar.

- [*graphdesc*] **view frame** (**on** | **off**)

  switch frame of view on or off

- [*graphdesc*] **view frame rim** *linestylecommands*

  use *linestylecommands* for the rim of the view frame

  [*graphdesc*] **view frame fill** *fillstylecommands*

  use *fillstylecommands* for the view frame filling

The current settings can be loaded from the insert menu in the script editor.

top

7.2.9 Legend box commands

The legend box belongs to a particular graph. The following commands refer to the current graph. They can also be prefixed with a *graphdesc* or follow after **with g***int* to set the current graph (as indicated in *graphcommands*).

- **legend** (**on** | **off**)

  switch legend box on or off

- **legend font** *fontstylecommands*

  use *fontstylecommands* for text in legendbox

- **legend line spacing** *exp*

  set spacing between legend box entries (lines). *exp* must evaluate to a scalar.

- **legend symbol marker spacing** *exp*

  set spacing between symbol and text

- **legend symbol marker width** *exp*

  set width of symbol before the text

- **legend frame** (**on** | **off**)

  switch frame on or off

- **legend frame rim** *linestylecommands*

  use *linestylecommands* for the rim

- **legend frame border spacing** *exp*

  set spacing between rim (border) and content (symbols, text)

- **legend frame fill** *fillstylecommands*

  use *fillstylecommands* for the frame

- **legend position** *exp* , *exp*

  set legend position relative to the view frame (y=0 is on the top). The position refers to the point of the legend box given as origin (see below)

- **legend origin** *exp* , *exp*

  define the point of the legend box, which is considered its origin. (the **position** command places this point.)

The current settings can be loaded from the insert menu in the script editor.

top

7.2.10 Graph commands

Graph commands will manipulate graphs. A graph is a single view-frame/plot with axes, which can contain several groups/sets. Graphs have (irregular) tic mark settings and can contain shapes (lines/ellipses). Graphs can be scaled.

- *graphdesc* (**on | off | toggle**)

  switch *graph* on or off or toggle on/off (for *hookcommands*). Example:

  ```
  g2 on
  ```

- **new graph**

  create a new graph. More control is achieved via switching a particular graph on or off.

- [*graphdesc*] *textboxcommands*

  use *textboxcommands* for the current/specified graph. Example:

  ```
  g1 subtitle on
  subtitle "some subtitle"
  subtitle font color 0xff
  ```

- [*graphdesc*] *legendboxcommands*

  use *legendboxcommands* for the current/specified graph.

- [*graphdesc*] *ticmarkcommands*

  use *ticmarkcommands* for the current/specified graph.

- [*graphdesc*] *irregularticmarkcommands*

  use *irregularticmarkcommands* for the current/specified graph.

- [*graphdesc*] **graph line width scale** *exp*

  set an overall line width scale for the current/specified graph. This affects all line widths.

- [*graphdesc*] **graph point size scale** *exp*

  set an overall point size scale for the current/specified graph. This affects font and symbol sizes and line widths.

- [*graphdesc*] *shapecommands*

  manipulate shapes (arrows/lines/ellipses), see *shapecommands*. Example:

  ```
  g1 line1 on
  line1 line width 2
  ```

top

7.2.11 Shape commands

In the moment line shapes (lines/arrows) and ellipses (circles) can be added to a graph. First let us define shape descriptors:

- **ellipse**[ ]*int*

  reference to ellipse number *int*. The space between the keyword and the number is optional. Example:

  ```
  ellipse3
  ```

- **line**[ ]*int*

  reference to line shape number *int*. This can be an arrow not just a line. The space between the keyword and the number is optional. Example:

  ```
  line 2
  ```

Now, we give the details for each shape type. Some commands are specific for the particular shape kind others are common.

- *shapedesc* (**on** | **off**)

  switch on (create it of not yet existing) or off a *shape*. Example:

  ```
  line1 on
  ```

- *shapedesc* **name** (*string* | *parameter*)

  give the *shape* a name. This is usefull in the GUI, where all shapes appear in a list. Example:

  ```
  line1 name "my-fancy-line-name"
  ```

- *shapedesc* **line** *linestylecommands*

  use *linestylecommands* for the *shape*'s rim/line.Example:

  ```
  line1 line color 0xff
  line1 line style dot
  ```

- *shapedesc* **coordinatesystem** (**view** | **world**)

  use this coordinate system's units for the *shape*'s size/positional settings. World coordinates refer to the physical coordinates of the underlying plot. Hence, changing the zoom of the plot will move the shape with it. View coordinates refer to the graph's view/frame. These are rleative coordinates, where `x,y=(0,0)` is the upper left corner and `x,y=(1,1)` is the lower right corner of the view.

- *ellipsedesc* **fill** *fillstylecommands*

  use *fillstylecommands* for the *ellipse*'s interior. Example:

  ```
  ellipse1 fill color 0xff
  ellipse1 fill on
  ```

- *ellipsedesc* **angle** *exp*

  set the angle about which the *ellipse* is rotated.

- *ellipsedesc* **center** *exp* , *exp*

  set the center of the *ellipse*. This refers either to world or view coordinates.

- *ellipsedesc* **radii** *exp* , *exp*

  set the two radii of the *ellipse*. This refers either to world or view coordinates.

- *ellipsedesc* **radius** *exp*

  set both radii of the *ellipse* to the same value (making it a circle). This refers either to world or view coordinates.

- *linedesc* **arrow fill** *fillstylecommands*

  use *fillstylecommands* for the *line*. Note, that only the fill color command does work here, i.e. closed arrow heads allways have `fill on`. If an empty head is required, use `fill color 0xffffff` (white).

- *linedesc* (**arrow** | **cap**) **position** (**none** | **start** | **end** | **both**)

  set the position of the arrow head or cap on the *line* shape. A cap is a little runding off at the end of the line. better visible if the line width is larger.

- *linedesc* **arrow style** (**open****| **closed**)

  open arrows are made of lines, closed ones are haveing a filling.

- *linedesc* **arrow** (**size** | **sharpness**) *exp*

  set the size or sharpness of the arrow head.

- *linedesc* (**start** | **end**) *exp* , *exp*

  set the start and the end of the *line* shape. (refers to the specified *coordinatesystem*.)

The current settings of existing shapes can be loaded from the insert menu in the script editor.

top

7.2.12 Text box commands

Text boxes are used for axis labels but can also be placed freely in the graphs. They belong to a particular graph. The special text boxes like axes- and tic labels can impose restrictions on each others placement, which are used to keep them at reasonable distance from each other. Text boxes have an origin, with respect to which the position is defined.

Textboxes are identified by a descriptor:

- **textbox**[ ]*int*

  the general text box number *int*. The space is optional. Example:

  ```
  textbox1
  #or
  textbox 1
  ```

- **(title | subtitle | xaxislabel | yaxislabel | oppositexaxislabel | oppositeyaxislabel)**

  one of the special textboxes

Now, the commands:

- *textboxdesc* (**on** | **off**)

  switch *textbox* on or off

- *textboxdesc* (*string* | *parameter*)

  set the content of *textbox* to *string* or *parameter*. The *string* can contain *formating*. Example:

  ```
  title "N$_0$.$x{-0.7}$^$iFe$n$."
  ```

  will produce N_0^{Fe}

- *textboxdesc* **font** *fontstylecommands*

  use *fontstylecommands* for *textbox*. Example:

  ```
  title font color 0xff00ff
  ```

- *textboxdesc* **frame** (**on** | **off**)

  switch this *textbox*'s frame on or off.

- *textboxdesc* **frame rim** *linestylecommands*

  use *linestylecommands* for the frame rim of *textbox*

- *textboxdesc* **frame fill** *fillstylecommands*

  use *fillstylecommands* for the filling of *textbox*'s frame

- *textboxdesc* **frame border spacing** *exp*

  define the space between the frame rim and the text.

- *textboxdesc* **coordinatesystem** (**view** | **world**)

  Set the *coordinatesystem* for the text position.

- *textboxdesc* **position** *exp* , *exp*

  set the position of the *textbox*. This referes to the specified coordinate system. See origin below.

- *textboxdesc* **origin** *exp* , *exp*

  The *textbox* has an origin, which is positioned accoding to the **position** command.

- *textboxdesc* **angle** *exp*

  set the rotation angle of the *textbox*

- *textboxdesc* **restriction** (**+x** | **-x** | **+y** | **-y**) *exp*

  restrict the movement of this *textbox* to the positive/negative x/y-axis and shift it in this direction by the amount *exp*. Example:

  ```
  title  restriction -Y 0.03
  ```

- *textboxdesc* **restriction none**

  define that there are no additional placement restrictions for this textbox

- *textboxdesc* **restriction additional** [*int* [*int* [*int*. . .]]]

  define a list of *int* values, encoding various additional restrictions refering to the special text boxes. The list can be empty.

x-tic labels	int	y-tic labels	int	sub-title/axis-labels	int
opposite x tic label height	1	opposite y tic label width	5	subtitle offset	9
opposite x tic label offset	2	opposite y tic label offset	6	subtitle height	10
normal x tic label height	3	normal y tic label width	7	opposite x-axis label offset	11
normal x tic label offset	4	normal y tic label offset	8	opposite x-axis label height	12

The current settings of existing textboxes can be loaded from the insert menu in the script editor.

top

7.2.13 Tic mark commands

Tics come in two varieties: regular tics, which are graph specific and *irregular tics* which can belong to a graph or a group.

7.2.13.1 Regular tic commands:

- (**x** | **y**) **auto tic**[s] (**on** | **off**)

  auto tics on means that the tic spacing is determined automatically, off means that the user has to set the major tic spacing and the minor tic subdivisions. Example:

  ```
  x auto tics off
  ```

- **(x | y) axis scaling (log[arithmic] | lin[ear]**)

  set the axis scaling to linear or logarithmic. Also read *Logarithmic plots*. Example:

  ```
  x axis scaling log
  ```

- (**x** | **y**) **tic**[s] **side** (**none** | **normal** | **opposite** | **both**)

  on which side to place the tics. Example:

  ```
  x tic side opposite
  ```

- (**x** | **y**) (**major** | **minor**) **tic**[s] (**on** | **off**)

  switch the drawing of x/y major/minor tics on or off. Example:

  ```
  x minor tics off
  ```

- (**x** | **y**) **major tic**[s] **spacing** *exp*

  the distance of major tics in world units. For logarithmic axes the spacing between two major tics is determined by multiplication with this number instead. Example:

  ```
  x major tic spacing 0.1
  ```

- **y** (**major** | **minor**) **separate tic**[s] **length** (**on** | **off**)

  if switched on, the length of the y axis tics is set separatly. Otherwise it is the same physcial length as for the x-axis tics. Example:

  ```
  y major separate tic length on
  x major tic length 0.03
  y major tic length 0.05
  ```

- (**x** | **y**) (**major** | **minor**) **tic**[s] **length** *exp*

  set the length of the tics in view units.

- (**x** | **y**) **minor tic**[s] **subdiv** *int*

  set the number of subdivisions of a major tic interval to define the position of the minor tics.Example:

  ```
  x minor tic subdiv 2
  ```

- (**x** | **y**) (**major** | **minor**) **tic**[s] **line** *linestylecommands*

  use *linestylecommands* for the tics. Example:

  ```
  x major tic line color 0xff00
  ```

- (**x** | **y**) **tic**[s] **label**[s] **decimals** *int*

  set the number of decimals to be printed in the tic labels. A negative number signals to use the automatic default. Example:

  ```
  x tic label decimals 2
  ```

- (**x** | **y**) **tic**[**s**] **label**[**s**] **offset** *exp*

  the label offset from the axis in view units. Example:

  ```
  x tic label offset 0.1
  ```

- (**x** | **y**) **tic**[**s**] **label**[**s**] **side** (**none** | **normal** | **opposite** | **both**)

  where to draw the tic labels. Example:

  ```
  x tic label side both
  ```

- (**x** | **y**) **tic**[**s**] **label**[**s**] **font** *fontstylecommands*

  use *fontstylecommands* for label's text. Example:

  ```
  x tic label font color 0xffff
  ```

top

7.2.13.2 Irregular tic commands

Irregular tics can be owned by graphs and groups. Group owned irregular tics will be visible if the group is visible. These commands must follow after a particular graph/group was specified by a previous command (*graphcommands*, *groupcommands*) or via a **with**-command. The group irregular tic mark commands always need a *groupdesc* prefix.

- **irregular tic**[**s**] (**on** | **off**)

  switch on irregular tics. Example:

  ```
  gr1 on
  g1.gr1 irregular tics on
  ```

The irregular tics get descriptors formed in the following way:

- **itic**[ ]*int*

  an irregular tic descriptor. The space is optional. Example:

  ```
  itic1
  ```

Now the commands

- *iticdesc* **type** (**x** | **y**) (**major** | **minor**)

  define if *itic* is an x or a y tic and if it is major or minor. Major tics can have labels. Example:

  ```
  itic1 type y major
  ```

- *iticdesc* **length** *exp*

  define the tic length of *itic*. This is in view units. a length of 1 creates a line spanning the whole view area. Example:

  ```
  # span the full
  itic1 length 1
  ```

- *iticdesc* **position** *exp*

  the tic position in world units. Example:

  ```
  itic1 position 0
  ```

- *iticdesc* **label**[s] (*string* | *parameter*)

  the *itic*’s label. Only for major labels. Example:

  ```
  itic1 label "E$_F$."
  ```

- *iticdesc* **tic**[s] **side** (**none** | **normal** | **opposite** | **both**)

  at which side to draw the *itic*

- *iticdesc* **label**[s] **side** (**none** | **normal** | **opposite** | **both**)

  at which side to draw the *itic* label:

  ```
  itic1 tic side normal
  itic1 label side opposite
  ```

- *iticdesc* **label**[s] **offset** *exp*

  the label offset from the axis in view units. Example:

  ```
  itic1 label offset 0.07
  ```

- *iticdesc* **line** *linestylecommands*

  use *linestylecommands* for the *itic*. Example:

  ```
  itic1 line style dot
  ```

Example: the Fermi level can be done like this:

```
gr1 on
g1.gr1 irregular tics on
itic1 type y major
itic1 length 1
itic1 position 0
itic1 label "$~e$_F$."
itic1 label side opposite
itic1 line style dot
```

top

7.2.14 Group commands

Group commands manipulate groups. They act on the current *group*, which can be set either explicitly or implicitly as explained in so many other places.

- [*groupdesc*] **use attributes** (**on** | **off**)

  if on, the group attributes will be used for each set of the current or specified *group*. When switched off, each *set*’s individual properties will be used.

- [*groupdesc*] *setattribcommands*

  use *setattribcommands* for the current or specified *group*

- [*groupdesc*] *weightsettings*

  use *weightsettings* for the current or specified *group*

- *groupdesc irregularticmarkcommands*

  define *irregularticmarkcommands* for this *group*.

top

7.2.15 Set commands

Set commands are commands, which modify a single set (sometimes all sets of a group). The set can be made current via the **with** command or via at least one explicit *set* descriptor.

Example:

```
# we assume that the current graph and group are already set

s1 on
# alternatively
with s1
# now the properties
line color 0xff0000
symbol style circle
convolute(s1,0.5)
```

- [*setdesc*] *setattribcommands*

  use *setattribcommands* for this *set*

- [*setdesc*] *weightsettings*

  use *weightsettings* for this *set*

- [(*setdesc* | *groupdesc*) .](**x** | **y**) = *exp*

  assign *exp* to the x/y-vector of this *group/set*. *exp* can be a vector or scalar expression. Example:

  ```
  # this will create a parabola by setting y[i]=x[i]^2
  # for each point i in the set
  g1.gr1.s1.y=x^2

  # or

  with g1.gr1.s1

  y=x^2
  ```

- **convolute (** *setdesc* , *exp* **)**

  convolute the *set* with a Gaussian of half width *exp*.

- *setdesc* **length** *exp*

  set the length of the x and y vectors. (The set must by of xy-type.)

- **bspline (** *setdesc* , *int1* , *int2* **)**

  construct the *int2*-th derivative of the B-spline interpolation of order *int1* of the *set* and apply it to the set. B-splines are spline interpolations of arbitrary order. Note, that there must be a minimum number of data points to construct it. Zeroth order means histogram. First order means linear interpolation second order quadratic spline-interpolation and so on. Note, that *int2*=0 means zeroth derivative and that nothing changes, since the bspline is an exact interpolation at the data points and a zeroth derivative is an identity. To calculate derivatives you chose a spline order, which does not lead to too many wiggles but is high enough to smoothly represent the data. Example:

  ```
  killall
  with g1.gr1
  s1 on
  N=100
  m=mesh(0,10,N)
  s1 length N
  x=m
  y=sin(x)
  line color 0xff
  ```

(continues on next page)

(continued from previous page)

```
copy s1 to s2
s2.y=cos(x)
line color 0xff0000
#first derivative
bspline(s1,2,1)
copy s1 to s3
#difference
s3.y=y-s2.y
line color 0xff00
legend "error"
autoscale
```

- **bspline** (*setdesc* , *int1* , *int2* , *identifier*)

  construct the *int2*-th derivative of the B-spline of order *int1* and interpolate it onto the x-vector pointed to by *identifier*, which must be a vector. The result is saved in *set*.y. Example

```
kill all
g1.gr1.s1 on
data xy
0 0
1 1
2 4
end data
legend "raw data\char"
line color 0xf00
symbol style circle
symbol fill on
line style none
N=100
m=mesh(-1,3,N)
copy s1 to s2
bspline(s2,1,0,m)
s2 legend "spline order 1"
line color 0xff0000
copy s1 to s3
bspline(s3,2,0,m)
s3 legend "spline order 2"
line color 0xff00
autoscale
```

- **integrate** (*setdesc*)

  running sum integral of *set*. This produces the curve $I(x) = \int_{x_0}^{x} y(u)\,du$ for each point x_i in the set (upper boundary indefinite integral). For derivatives see *bspline*. If you need the value of the integral in a script do something like:

```
integrate(s1)
sum=y[length-1]
```

- **data** *datatype*

  datalines

  end data

  define a block of data of *datatype*

datatype	explanantion
xy	standard y(x)
grid/xynz	z(x,y)
xynw	y(x) and w(x,y)

The *datalines* depend on the *datatype*. They may contain *identifier* and things like **world**.**xmin**. *Example*. Also look at saved `.xfp` files.

- [*setdesc*] **weightlabel**[s] **reference**[ ]*int*

  set the weight label reference of the current set to *int*. This refers to a particular *weightlabeldefinitions*, defined elswhere. Sets with the same reference are treated as having the same set of weights. This allows to manipulate the weights settings of all these sets together (especially when grouped). The number of weights and dimensions of x/y of all these sets must be the same!

top

7.2.16 Set attribute commands

Set attributes can be applied to groups or sets. The last *group/set* descriptor or **with** command will decide which object's attributes are defined. You can also prefix each command or at least the first of a series of such commands with an explicit *group/set* descriptor.

- (**on** | **off**)

  switch the current *group/set* on or off.

- **toggle**

  toggle the current *group/set*. (For *hookcommands*)

- **line** *linestylecommands*

  use *linestylecommands* for the current *group/set*.

- **symbol** *symbolstylecommands*

  use *symbolstylecommands* for the current *group/set*.

- **legend** (*string* | *parameter*)

  set the legend string for the current *group/set*.

- **showin legend** (**on** | **off**)

  decide if *group/set* this is shown in the legend box.

- **interpolationdepth** *int*

  set the interpolation depth for grid/density plots.

- **max cutoff** *exp*

  set the upper cutoff for density grid/density plots. see "scalemax" in Section *SetDialog*.

- **min cutoff** *exp*

  set the lower cutoff for density grid/density plots. see "scalemin" in Section *SetDialog*.

- **min color** *hexconstant*

  set the data-background color for density grid/density plots. see "z0" in Section *SetDialog*.

- **null cutoff** *exp*

  set the z-value under which data background color is gradually inserted for density grid/density plots. see "z0" in Section *SetDialog*.

- **zpower** *exp*

  set the power law for mapping of z-values onto colormaps for density grid/density plots. see "zpower" in Section *SetDialog*.

- **zcomponent** *exp*

  set which z-component is ploted for density grid/density plots. This is usually 0. see Section *SetDialog*.

- **colormap from** *exp* , *exp*

  exp must be hex constants and represent colors. The resulting colormap is interpolated between the two.

- **colormap name** *string*

  select a predefined colormap

Terrain	Hot	Autumn
RainBow	Heat	Winter
Magma	Spring	Gnuplot
Inferno	Summer	Seismic
RainBow2	RainBow3	

- **colormap** *exp*
  real : *hexconstant real* : *hexconstant* . . .
  real : *hexconstant real* : *hexconstant* . . .
  . . .
  colormap end

  This is a list of pairs of z-values in `[0,1]` and hex-colors. Each line can contain arbitrary number of pairs. There can be arbitrary number of lines. There are **no** commas. This defines a set of mappings of z-values from the standard interval onto colors. If *exp* is nonzero the interpolation is done in RGB space, otherwise in the circular hue-saturation space.

- **use secondderivative** (**on** | **off**)

  (**deprecated does not work**) decide the use of second derivatives for grid/density plots.

- **setcomment** *string*

  set the "set-comment" of the current *group/set*. Set-comments denote, where the data came from. This is mostly used in the *.xfp files, which get saved and loaded by xfbp.

top

7.2.17 Weight commands

Weight commands determine the appearance of individual weights of groups or sets. The internal organization of weights into groups is a bit confusing. As long as your weights came from an fplo bandweights file it should work alright. Weight descriptors can be made current via a **with** command or by using a *weightdesc* (at least for the first command which belongs to a particular weight) Examples:

```
w11 on
plotorder 1
name "yx"
# or
with w11
on
plotorder 1
name "yx"
```

- [*weightdesc*] (**on** | **off**)

  switch the *weight* on or off. Note, that if the current *weightdesc* is invalid but some other descriptor is valid, this command switches the currently valid *graph/group/set*. Example:

  ```
  with g1
  on
  # we switched graph 1
  with gr1.w2
  # implicit g1 explicit gr1 and w2
  ```

(continues on next page)

(continued from previous page)

```
on
# now, we switched weight 2
```

- [*weightdesc*] **plotorder** *int*

  the weight with the higher plotorder gets plotted later.

- [*weightdesc*] **name** (*string* | *parameter*)

  rename the weight. This affects the legend entry. Note, that a name change also means that the *weightdesc* `w"weigthname"` changes as well. Furthermore the weight label/name is unique in that it is a property of the weight itself such that if one changes the name of a weight in a group it also changes the name of this weight in all sets, which are assoziated to this weight. i.e. all sets of all groups, which are belonging together, like spin up and down groups of a +bweights plot.

- [*weightdesc*] **color** *hexconstant*

  set the weights color to *hexconstant*

- [*weightdesc*] **skip** *exp*

  skip as many symbols between plotted symbols (unless *weight style* is connected). skip 0 means: plot all symbols.

- [*weightdesc*] **symbol fill** (**on** | **off**)

  fill symbols or not. (only if is *weight style* individual).

- [*weightdesc*] **symbol line width** *exp*

  for open symbols the symbol line width matters. (only if *weight style* is individual).

- [*weightdesc*] **symbol style** *symboltype*

  set the *symboltype*. (only if *weight style* is individual).

top

7.2.18 Weight settings

Weight settings can be applied to groups or sets. The last *group/set* descriptor or **with** command will decide which object's settings are defined. Weight settings affect the appearance of all visible weights of a group or set.

- **weight max** *exp*

  set the symbol size, which represents a weight value of 1. *exp* is in a scale like font sizes.

- **weight min** *exp*

  set the weight symbol size, under which no symbols will be plotted. This depends on **max** since it scales everything up.

- **weight factor** *exp*

  scale all weights by this factor, before applying **max** and **min**. (This is somehow superfluous).

- **weight style** (**dots** | **connected** | **individual**)

  set the style of the weights.

dots	individual filled circles for each data point
connected	connect the circles by linear intepolation
individual	each weight can have its own symbol

top

7.2.19 Weightlabel definitions

This is special stuff to define tables of weightlabels especially in saved files.

- **weightlabel definition**[s]
  weightlabels[ ]*int1*
  stringlist
  end weightlabel[s]
  ...
  weightlabels[ ]*intn*
  stringlist
  end weightlabel[s]

  define tables of weightlabels with references *int1* through *intn*. See *weightlabel references*. *stringlist* is a list of weight labels (strings), each on a separate line.

top

7.2.20 Line style commands

Line style commands are used in several other commands.

- **color** *hexconstant*

  define the color using a hex number, which encodes the RGB (red-green-blue) color components. The constant contains three bytes (leading zeros need not be specified) the left-most is red, then green and right-most is blue:

hex	color	red	green	blue
0x000000=0x0	black	0	0	0
0xff0000	red	255 (ff)	0	0
0x00ff00=0xff00	green	0	255 (ff)	0
0x0000ff=0xff	blue	0	0	255 (ff)
0x800000	darkred	128 (80)	0	0
0xffffff	white	255 (ff)	255 (ff)	255 (ff)

- **width** *exp*

  set line with

- **style** (*linestyle* | *int*)

  set line style via name or number

linestyle	int	linestyle	int	linestyle	int
none	0			long dash	7
solid	1	dash dot	4	long dash dot	8
dash	2	dash dot dot	5	long dash dot dot	9
dot	3	dot dash dash	6	dot long dash long dash	10

  Example:

```
s1 line style dash
# or
s1 line style 2
```

top

7.2.21 Fill style commands

These commands appear in various other commans and usually cannot stand by themselfs (they are preceded by … **fill**).

- **color** *hexconstant*

  define fill color using a *hexconstant*

- (**on** | **off**)

  switch filling on or off

top

7.2.22 Font style commands

These commands are used in several commands after **font**.

- **size** *exp*

  set the font size to *exp*, which must evaluate to a number.

- **subscriptscale** *exp*

  set the scaling down ratio of subscript

- **color** *hexconstant*

  define font color using a *hexconstant*

top

7.2.23 Symbol style commands

These commands are used in other commands and preceded by **symbol**.

- **style** (*symboltype**int*)

  set symbol style by name or number

symboltype	int	symboltype	int	symboltype	int
none	0	triangleup	4	plus/+	8
circle	1	triangleleft	5	cross/x	9
square	2	triangledown	6	star/*	10
diamond	3	triangleright	7		

  Example:

```
s1 symbol style square
# or
s1 symbol style 2
```

- **size** *exp*

  set the symbol size to *exp*, which must evaluate to a number. The symbol sizes have the same scale as font sizes.

- **fill** *fillstylecommands*

  use *fillstylecommands* for the symbol.

- **line** *linestylecommands*

  use *linestylecommands* for the rim

Example:

```
g1.gr1.s1 symbol style diamond
symbol size 18
symbol fill  on
symbol fill color 0xff0000
symbol line color 0x0000ff
symbol line width 2
```

top

7.2.24 Kill commands

- **kill**[ ]**all**

  kill everything (clean slate) and initialize graph 1 in default state. The space between the keyword and the number is optional.

- **kill** (*graphdesc* | *groupdesc* | *setdesc*)

  remove *graph/group/set*. If the last graph was killed graph 1 is initialized in a default state. Example:

  ```
  kill g1.gr12
  ```

top

7.2.25 Copy/Move commands

Use these to move sets around or copy them.

- (**copy** | **move**) *setdesc1* **to** *setdesc2*

  copy/move *set*1 to *set*2. Moving is essentially renaming. Note, that you can move a set to a different graph and/or group:

  ```
  # here I shall have a g1.gr1.s1
  g2 on
  # now we have graph 2
  gr3 on
  # now we have a group3 in graph 2
  # in order to move g1.gr1.s1 there, we have to explicitly spell out g1.gr1,
  # since the last command defined the current descriptor as g2.gr3, hence:
  move g1.gr1.s1 to g2.gr3.s11
  # now we have moved the set into s11 in graph 2 group 3

  # in order to move the s11 to s1 in g2.gr3 we do not have to change groups or␣
  ↪graphs

  # hence,

  move s11 to s1

  # will work
  ```

top

7.2.26 Hook commands

Sometimes it is usefull to have the program do something when the mouse is clicked on a certain point in a graph. This can be done by ”hooking” a particular command to the mouseclick.

- **hook mouseclick left** *command-as-string*

  set the command in *command-as-string* to be hooked onto the left mouse click. In the moment only left mouse click is implemented. When hooking is active the current world point (at clicking) is written into the file `+currentpoint`. *command-as-string* can stretch over multiple lines to allow for several commands. Example:

  ```
  hook mouseclick left "
   with g1.gr10
    s1 on
    s1 length 2
    x[0]=cursor.x
    x[1]=cursor.x
    y[0]=world.ymin
    y[1]=world.ymax
    line color 0xff0000
  "
  ```

  This will draw a vertical red line at the mouse position when clicked.

top

7.2.27 Cursor reference

If **hook** commands are active the world coordinates pointed to by the mouse cursor can be references.

- **cursor** . (**x** | **y**)

  reference the world coorinate under the mouse cursor. Example:

  ```
  hook mouseclick left "world xmin cursor.x
  world ymin cursor.y"
  ```

  Note, the newline within the string!

top

7.2.28 Assignments/Definitions

One can in a limited way define variables: scalars and vectors.

- *identifier* = *exp*

  define a variable name *identifier* and assign it *exp*. Note, that “length” cannot be an *identifier* name. Example:

  ```
  N=100
  m=mesh(0.1,12.3,N)
  ```

- [*setdesc* .](**x** | **y**) **[** *exp1* **]** = *exp2*

  this assigns the *exp2* to a value x[*exp1*] or y[*exp1*] in the current or specified *set*. Note, that the square brackets are for real. It is a vector element reference. The vector assignment is explained *here* . Example:

  ```
  s1.y[0]=0
  s1.y[length-1]=0
  ```

top

7.2.29 Expressions

exp can be anything of the following:

- **world** . (**xmin** | **xmax** | **ymin** | **ymax**)

  reference to the current world limits. (For *hookcommands* and other usefull stuff.)

- **mesh** (*exp1* , *exp2* , *exp3*)

  define a vector of length *exp3* made of equidistant points between *exp1* and *exp2*. Example:

```
s1 on
N=100
s1 length N
x=mesh(0.1,12.3,N)
y=x^2
```

- *real*

  any real or integer number.

- *identifier*

  any previously defined *identifier*

- *parameter*

  any parameter defined on the command line. See *parameter*.

- **cursor** . (**x** | **y**)

  a reference to the mouse cursor position. (For *hookcommands*) See *Cursor reference*.

- [*setdesc* .] **length**

  the length of the *set*'s x/y-vectors. Example:

```
with s1

x[length-1]=100
```

- [*setdesc* .] (**x** | **y**)

  the *set*'s x/y-vector. *Example*.

- [*groupdesc* .] (**x** | **y**)

  the *group*'s x/y-vectors

- *setdesc* . (**x** | **y**) **[** *exp* **]**

  the *exp*-th element of the x/y-vector of *set*. The square brackets are for real here. Example:

```
s1.x[10]
```

- **moment** (*setdesc* , *exp1* , *exp2* , *exp3*)

  the *exp1*-th moment of *set*.y in the x-interval [*exp2*,*exp3*]. These are normalized moment $M_n = \frac{\int_{x_0}^{x_1} f(u)u^n du}{\int_{x_0}^{x_1} f(u) du}$, hence $M_1 \equiv 1$. The second moment corrected for the center of gravity is defined as $\frac{\left\langle (x-M_1)^2 \right\rangle}{\langle\rangle} = M_2 - M_1^2$. Example:

```
killall
N=1000
a=0.5
x0=-0.6
x1=1.6
w=0.2
```

(continues on next page)

(continued from previous page)

```
g1.gr1.s1 on
s1 length N
x=mesh(x0,x1,N)
y=exp(-((x-a)/w)^2/2)*3
m1=moment(s1,1,x0,x1)
m2=moment(s1,2,x0,x1)
# calculate the normalized width
wi=sqrt(m2-m1^2)
echo "wi=",wi
gr1 irregular tics on
itic1 type x major
itic1 position m1
itic1 length 1
itic1 label side opposite
itic1 label "m1"

itic2 type x major
itic2 position m1+wi
itic2 length 1
itic2 label side opposite
itic2 label "wi"
itic2 line style dash
autoscale
```

- (*exp*)

  we can use parentheses around any expression. Example:

```
a*(b+c)
```

- *function* (*exp*)

  These functions can be used on scalars or vectors

function	meaning	function	meaning	function	meaning
sqrt(x)	$\sqrt{x}$	sin(x)	$\sin(x)$	asin(x)	$\arcsin(x)$
abs(x)	$\|x\|$	cos(x)	$\cos(x)$	acos(x)	$\arccos(x)$
exp(x)	e^x	tan(x)	$\tan(x)$	atan(x)	$\arctan(x) \in [-\frac{\pi}{2}, \frac{\pi}{2}]$
log(x)	$\log_e(x)$	cot(x)	$\frac{1}{\tan(x)}$		
log10(x)	$\log_{10}(x)$	theta(x)	$\theta(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$		

- *function* (*exp* , *exp*)

  These functions can be used on scalars and vectors, and mixed arguments

function	meaning
min(x1,x2)	$\min(x_1, x_2)$
max(x1,x2)	$\max(x_1, x_2)$
sign(x1,x2)	$\|x_1\|\text{sign}(x_2)$
atan2(x1,x2)	$\arctan(\frac{x_2}{x_1}) \in [-\pi, \pi]$

- (**min** | **max**) (*vectorexp*)

  return a scalar containing the maximum or minimum of all values in the *vectorexp*, which must be a vector. Example:

```
m=min(g1.gr1.s1.y)
# or
```

(continues on next page)

(continued from previous page)

```
with g1.gr1

m=max(min(s1.y),1.0e-5)
# mixed scalar and vector expression: each element of s1.y will be the larger␣
↪of
# either the element or m.
y=max(y,m)
```

- *exp1* **^** *exp2*

  exp1 raised to the power of *exp2*.

- *exp1* (***** | **/**) *exp2*
- (**+** *exp* | **-** *exp*)
- *exp* (**+** | **-**) *exp*
- **format(** *string*, *exp* **)**
  format(*string*, *exp*, *exp* **)**
  format(*string*, *exp*, *exp*, *exp* **)**
  format(*string*, *exp*, *exp*, *exp*, *exp* **)**

  string must be a format string, as in C, where each "%" marker refers to one of the other arguments. There must be as many markers as *exp* arguments and the type must match.

  Typical markers are %s for strings, %ld for integer, %g for real, %10.5f for fixed format real with width 10 and 5 digits after the decimal point. Example:

```
i=42
suffix="x1"
print to format("file\%ld_\%s.png",i,suffix)
```

top

7.3 GUI

7.3.1 Plotting window

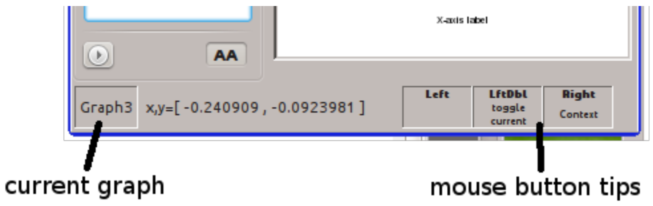
Use the right mouse key on an objects and double click (depends on where you do this). Have a look at the *mouse button tips*.

If the zoom buttons on the left are used, some of them change the cursor into a cross hair shape. This means that the corresponding function is switched on. To cancel the function use a right mouse click.

Hotkeys:

- **zoom in** Ctrl-+
- **zoom out** Ctrl- -
- **autoscale all** Ctrl-a
- **autoscale x** Ctrl-x
- **autoscale y** Ctrl-y
- **scroll** Ctrl-left, Ctrl-right, Ctrl-up, Ctrl-down

Hover with the mouse pointer over GUI elements to get tooltips.



To make a graph the current graph double click on empty space in the graph far enough away from sets and other objects. If several graphs overlap where you double click the current graph is changeing everytime you double click (loop through all graphs under the mouse cursor). You can also double click on empty space outside of any graph to loop through all the graphs. The current graph is displayed at the *buttom of the window*.

7.3.2 Scripting window

Have a look at the *edit* and *insert* menus for usefull functions.

Searching: Start search via *Ctrl-F*, type the search term, use *backspace* for corrections. Hit *Ctrl-F* to find next hit. If end of script is reached (colored search bar) hit *Ctrl-F* again to go back to start searching at the beginning of script. Use any cursor movement to cancel search mode. If *Ctrl-F* is hit to initiate search, hit it again to bring back the previous search string (if it exists).

7.4 Logarithmic plots

In log scale invalid data points are converted into very small numbers before plotting. This way the data sets are never invalid but the world scale will look weird.

For ellipses in world coordinate units the radii for the y/x direction denote the upper/right radius in world scale when the ellipse were placed at 1,1. Example:

```
killall
g1 on
x axis scaling log
y axis scaling log
world x 0.1 , 100
world y  0.1 , 100
Ellipse1 on
Ellipse1 name ""
Ellipse1 line style solid
Ellipse1 line width 1
Ellipse1 line color 0x0
Ellipse1 fill on
Ellipse1 fill color 0xeeeeee
Ellipse1 coordinate system World
Ellipse1 center 1,1
Ellipse1 radii 9,9
Ellipse1 angle  0
with g1.gr1
s1 on
data xy
10 0.1
10 100
end data
s2 on
data xy
0.1 10
100 10
end data
```

(continues on next page)

(continued from previous page)

```
textbox1 on
textbox1 coordinate system World
textbox1 "radius 9"
textbox1 position 11 , 1
textbox2 on
textbox2 coordinate system World
textbox2 "center at 1,1"
textbox2 position 11 , 9
Line1 on
Line1 coordinate system World
Line1 start  11.2658,5.96763
Line1 end  1.01985,0.796076
```

7.5 Data

The data are organized in groups and sets. Groups and sets have attributes. The group attributes can be set to hold for each set of the group irrespective of the individual set's settings.

7.6 Files

xfbp saves files in its own format (which is a subset of the *scripting language*). The extension is `.xfp`. It can also save and load the scripts, usually with the extension `.cmd`. It can load a set of *data files*.

If compiled with python support, it loads `.xpy` (and `.py`) files from the command line and executes them. The `.xpy` extension denotes a python script, which is using *pyxfbp bindings*. These files cannot be executed in a normal python shell, since they need **xfbp** to run properly.

7.7 Command line options

The program can be called with filenames as argument. If the file type has to be specified explicitely a *file type flag* has to preceed the filename. Most common files are automatically recognized via some heuristics. These are **fplo** band and bandweight files and **xfbp**'s own files: `.cmd`, `.xfp` and `.xpy` (or `.py` if you insist). If no file type is specified, data files are loaded with an implicit `-xny` flag (blocks of columnar data). In general each file is read into its own group. Spin polarized band structures create two groups if there are two spin directions (not full relativistic):

-cwd make the directory of following file current directory in the application. (I do not know, what this was for.)

-oi **open in observe mode, to monitor calculation** progress
(especially usefull for an on-machine **fplo** run).

-a namevalue specify a parameter name – value pair, separated by a colon, e.g:

```
-a p1:"filename1"
```

in generic command/script files, where the parameter `$p1` (or `p1` in python) will be available in the script and contain `"filename1"`. Also see python specifiaction *Commmand line parameters*.

-die in connection with a script use this option to quit **xfbp** after script execution:

```
xfbp -die script.xpy
```

7.7.1 File type flags

A file on the command line can be preceeded by a file type flag to force a certain file type. For more details on the data strutures see *data files*.

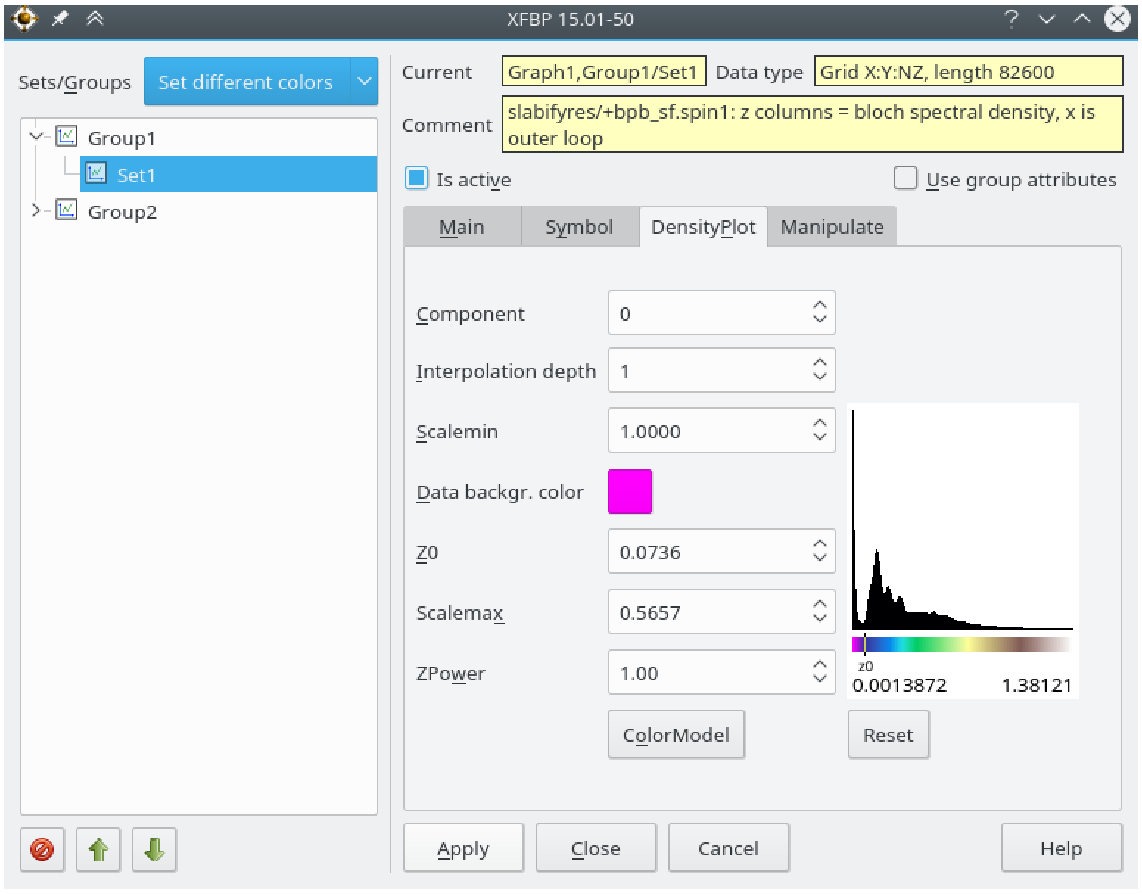
-xny	xny data
-xynw	xnyw data
-xynz	xynz gridded data
-band	band structure data
-bandweight	band weights data
-akbl	Bloch-Spectral-Density data
-p	a parameter file, containing scripting commands (only native).

7.8 Data file types

- **xny** Data sets are read, assuming that an empty line starts a new data block. In each multi column block with N columns the first column is x and the other $N-1$ columns are y_i, resulting in $N-1$ sets for each data block in the file. All sets end up in a new group.
- **xynw** The first column of each block is x. The second is y and the following columns are weights.
- **xynz** First comes a block of N_x x-values, each line one value, followed by a blank line. Then comes a similar block of N_y y-values followed by a blank line. Finally a block of $N_x \cdot N_y$ z-values, one value per line. The resulting plot will be a density plot where the z-values define the color.
- **band** An FPLO band structure file.
- **bandweight or bandweights** An FPLO band weights file.
- **akbl** An FPLO Bloch-Spectral-Density file. (CPA FPLO5, pyfplo.Slabify)

7.9 Set Dialog

For `xynz` functions a density plot is performed, which plots a color code for each $z(x, y)$. If you double click on a density plot in the viewport area this *dialog* opens. The tab “DensityPlot” controls the appearance of the plot.


XFBP 15.01-50

Sets/Groups: Set different colors

Group1
Set1
Group2

Current: Graph1,Group1/Set1 Data type: Grid X:Y:NZ, length 82600

Comment: slabifyres/+bpb_sf.spin1: z columns = bloch spectral density, x is outer loop

Is active ✓ Use group attributes ☐

Main | Symbol | DensityPlot | Manipulate

Component	0
Interpolation depth	1
Scalemin	1.0000
Data backgr. color	🟪
Z0	0.0736
Scalemax	0.5657
ZPower	1.00

Fig. 1: The densityplot tab of the set dialog.

- An `xynz` file can contain several components, e.g. the Bloch spectral function of the CPA modul (old) produced a gross (component 0) and net (component 1) spectral density in the same file.
- By default component 0 is plotted unless it is chosen differently (*component-spinbox* in the *dialog*, *zcomponent* or `pyxfbp.Set.zcomponent`). If the underlying data are coarse, use *interploation depth* in the *dialog* (*interpolationdepth* or `pyxfbp.Set.interpolationdepth`) to smoothly add more data points (dont use overly large values!).
- Chose a *colormodel* (*colormap* . . . or `pyxfbp.Set.colormap`).
- Look at the histogram, which shows the number of data points for each z-value and the z-interval which is mapped to the color map. By default the whole interval is mapped with a small modification. On loading the file (and through `pyxfbp.Set.adjustDensPlot`) an automatic detection of background data and upper data cut-off is attempted This can be changed and often needs to be changed.
  - if *scalemax* is decreased (from default 1 – no upper cut-off) more values at the higher end of z-values get painted with the maximum-value color. Basically all z values right of the right rim of the color bar have maximum-value color.
  - if *z0* (*null cutoff* or `pyxfbp.Set.z0`) is larger than the minimum z-value the lowest color value of the color model is interpolated into the *data background color* for z-values lower than *z0*. This can be used to plot values below a certain z-value with another color (to clean a data background). The default *data-background color* (*min color* or `pyxfbp.Set.databackgroundcolor`) is magenta to alert the user that something might need adjustment.
  - if *scalemin* (*min cutoff* or `pyxfbp.Set.scalemin`) is smaller than 1 the interpolation reaches full data-background-color already for z-values larger than `min(z)` but smaller than *z0*. This way the background cleaning color spreads more aggresively.
  - *zpower* (*zpower* or `pyxfbp.Set.zpower`) is used to map the z-values in a non-linear fashion onto the colormap.

Just try it. For completeness the scalar x which maps linearly to the colormap is defined as

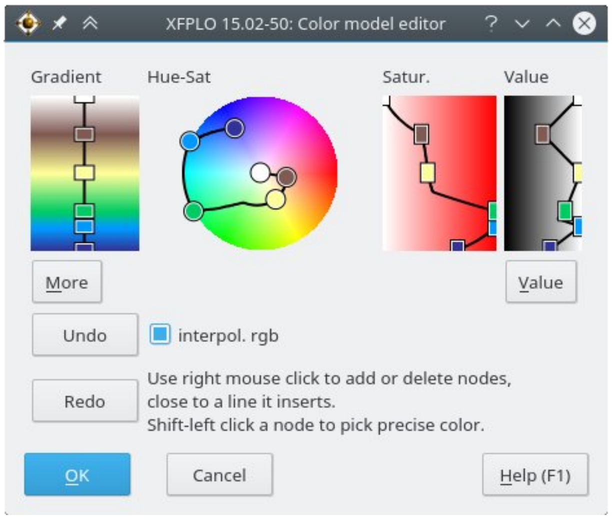
$$x = \max\left(\min\left(\frac{z - z_0}{(z_{max} - z_0)\,\text{scalemax}}, 1\right), 0\right)^{\text{zpower}}, \quad z > z_0$$

and the x which interpolates between the lower end of the colormap and the data-background color is defined as

$$x = \max\left(\min\left(\frac{(z_0 - z)}{(z_0 - z_{min})\,\text{scalemin}}, 1\right), 0\right)^{\text{zpower}}, \quad z < z_0$$

7.10 Color Model dialog

The *color map editor* has a "more" button from which one can choose pre-made color maps.



☑ interpol. rgb

Use right mouse click to add or delete nodes, close to a line it inserts.
Shift-left click a node to pick precise color.

Fig. 2: The colormap editor.

A map consists of a chain of nodes, each with a certain hue-saturation combination and a certain value (darkness). These nodes are interpolated to obtain all colors. The interpolation happens either on straight lines in the *Hue-Sat* space or in rgb space. The latter will happen if the *interpol. rgb* checkbox is on. The difference is most visible if two colors are chosen such that their connection line crosses the white center of the circle.

Right click somewhere and a node gets added.

- If there is no node yet, a single node appears.
- If there is one node a second appears and will be connected to the first node by a line.
- If there are more than two nodes ,the new node will be connected to the closest end node, or it will be inserted if the mouse click was close to a connection line. (Currently, a straight line between neighboring nodes is considered and not the curved lines which appear in rgb interpolation scheme. If you click on the visible line close to a node it should work in almost all cases)

Imediately after the right click the new node can be dragged around.

Right click on a node and it gets deleted.

Left click on a node the move it around.

Shift-left click a node to set a precise color.

Any of the 4 color widgets can be clicked on.

If you prefer to wrap your own color model start with a two color map. Choose more->clear to remove all prior nodes. Right click into the round Hue-Sat widget and drag the new node to your color of choice. Right click somewhere else in the same widget and drag to the desired color. This is a colormap.

Now, optionally, switch on/off the *interpol. rgb* checkbox. This will interpolate between the two end nodes either in RGB or Hue-Saturation colorspace. These two-end-color maps are usually rather usefull, especially with RGB interpolation.

You can invert the node sequence (more->invert)

The more->set-equal-distance function will make the gradient nodes equidistant.

PYTHON MODULE INDEX

INDEX