gtk3-matplotlib-cookbook Documentation

Release 0.1

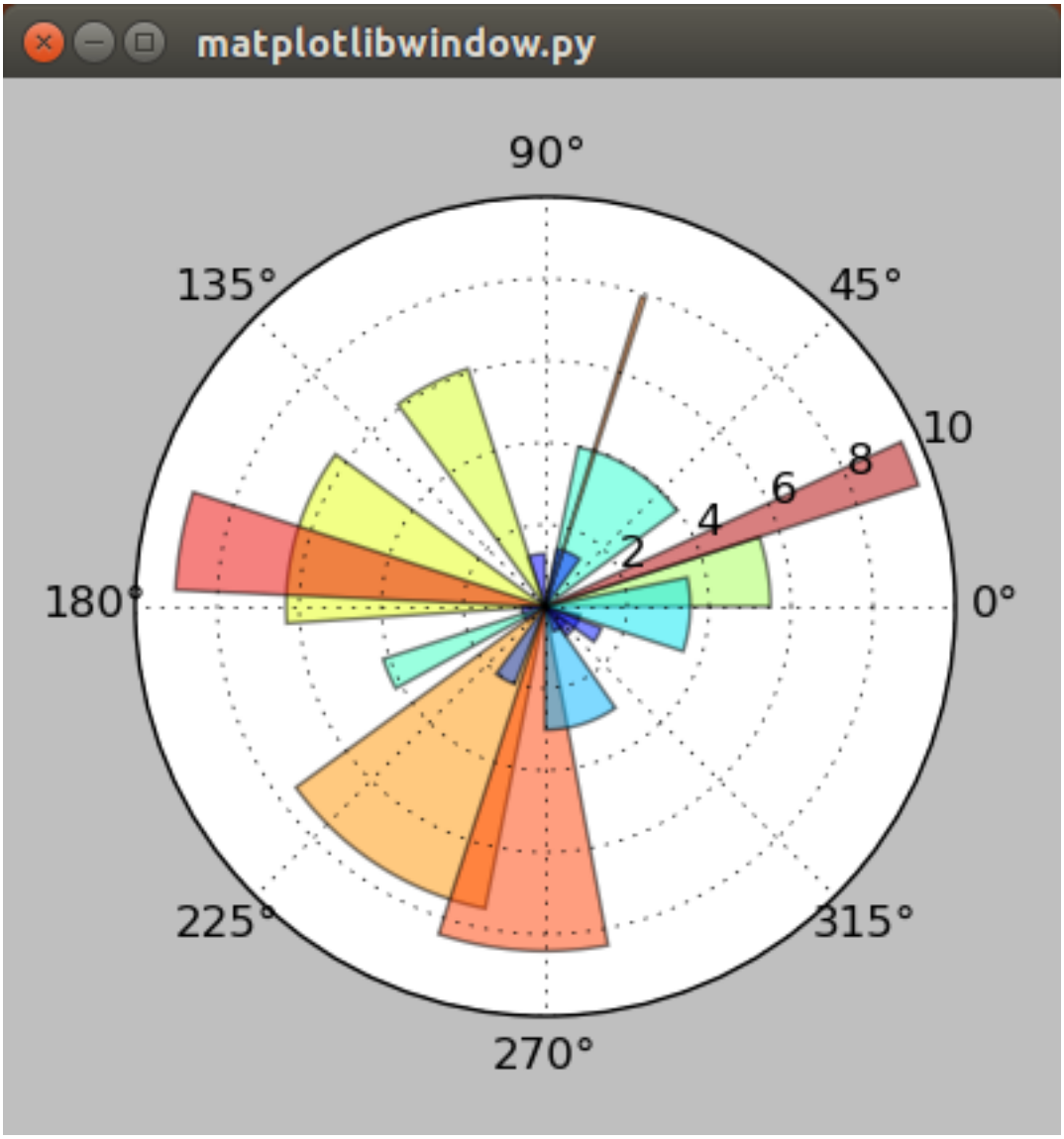
Tobias Schönberg

February 03, 2017

Contents

This Cookbook explains how to embed Matplotlib into GTK3 using Python 3. The tutorials will start out simple, and slowly increase in difficulty. The examples will focus on education and application of mathematics and science. The goal is to make the reader a independent developer of scientific applications that process and graph data.

If you would like to add your own examples to this tutorial, please create an issue or pull-request at the github repository (https://github.com/spiessbuerger/GTK3-Matplotlib-Cookbook).



CHAPTER 1

Requirements

To follow the examples you should have access to GTK+ 3.x, Python 3.x, Matplotlib 1.3.x and Glade 3.16.x or a more recent version.

CHAPTER 2

Recommended reading

In order to follow along with the Cookbook it is recommended to go through a Python 3.x tutorial (e.g. https://docs.python.org/3.4/tutorial/) and a tutorial about the Python bindings of GTK 3.x (e.g. http://python-gtk-3-tutorial.readthedocs.org/).

Additional reading:

- Matplotlib
  - http://wiki.scipy.org/Cookbook/Matplotlib
- Scientific Python
  - http://scipy-lectures.github.io/
- GTK3 Reference
  - https://lazka.github.io/pgi-docs/Gtk-3.0/index.html

CHAPTER 3

News and Comments

3.1 2014-12-26

I have been busy on some other projects lately. Revisiting this project, I had some trouble running my examples on Ubuntu 14.04 and the current master of Matplotlib (1.5.dev1). The problem seems to be in the *gtk3agg* backend. I get the following error: *NotImplementedError: Surface.create_for_data: Not Implemented yet.*. Not sure how many people have trouble with this, but it is mentioned in different places (e.g. http://matplotlib.1069221.n5.nabble.com/Matplotlib-py3-gtk3-td42953.html). Getting the plots to work again just requires you to switch rendering backends:

```
from matplotlib.backends.backend_gtk3cairo import FigureCanvasGTK3Cairo as FigureCanvas
```

3.2 2014-07-26

The fourth chapter about entering data is finished. I also decided to start code reviewing with Landscape. The score was 63 % on standard settings and 23 % on high (meaning very strict code reviewing). The low score is mostly caused by the *conf.py* file from Sphinx, my usage of "old classes" and Landscape complaining about the Gtk specific code. I will try to deal with the issues first, before uploading more chapters.

3.3 2014-06-27

Third chapter about zooming closer to datapoints is done. Currently I am just adding examples when I'm pleased with them. I will reorganize the book once the content is done. The new example relies heavily on classes. Does anybody reading this think that the amount of explanation is sufficient?

3.4 2014-06-19

This Cookbook is not finished and will be gradually developed in the coming months. If you would like to participate you can mail me or fork the repository. I am a beginning programmer, so the presented code might not be optimal. If you find any sub-optimal sections please write me, or create an issue or pull request on Github.

CHAPTER 4

Directory

4.1 Hello plot!

The first chapter will explain how to open an empty GTK3-window and then how to embed Matplotlib into it.

For small applications the GTK3-code can be easily integrated into the Python-code. Building the interface with Glade is a little more complicated in the beginning. With increasing size though, the usage of Glade will become more useful.

4.1.1 Empty GTK 3 window

Let's start with the code that will open an empty window.

```
#!/usr/bin/python3

from gi.repository import Gtk

myfirstwindow = Gtk.Window()

myfirstwindow.connect("delete-event", Gtk.main_quit)
myfirstwindow.show_all()
Gtk.main()
```

These are all the lines that are required for a fully functional window. This is what they do:

The first line helps Unix operating systems to recognize the file format of a file. In this case we wan't the operating system to know that the file should be excecuted with Python 3.x:

```
#!/usr/bin/python3
```

Then the program needs to import the gui-framework (or gui-toolkit). Older Gtk 2.x applications used (*import gtk*), but for Python 3 and Gtk 3.x applications (i.e. this tutorial) we need:

```
from gi.repository import Gtk
```

We can then define an object for a *Gtk.Window()*, which can have any name:

```
myfirstwindow = Gtk.Window()
```

Next the we have to connect our program with the quit-button (x-button) of the window. Otherwise closing the window will not terminate the application:

```
myfirstwindow.connect("delete-event", Gtk.main_quit)
```

The next line ensures that the program window is shown. Excluding this line will mean that the program start, but no window is displayed:

```
myfirstwindow.show_all()
```

The last line starts the main program loop with all functions. Without this line no loop is started and the program will not do anything:

```
Gtk.main()
```

Empty window with Glade

Opening an empty window with Glade takes a little more effort. First we need to open the Glade interface designer. Then drag a *GtkWindow* into the workspace. By default the window will be named *"window1"*, which we can keep. Then we have to set a signal for that window, so we can later close it. The signal we need is *"GtkWidget –> destroy"* and in the column process we can set *"on_window1_destroy"*. This will also be the name of the function in the Python code.

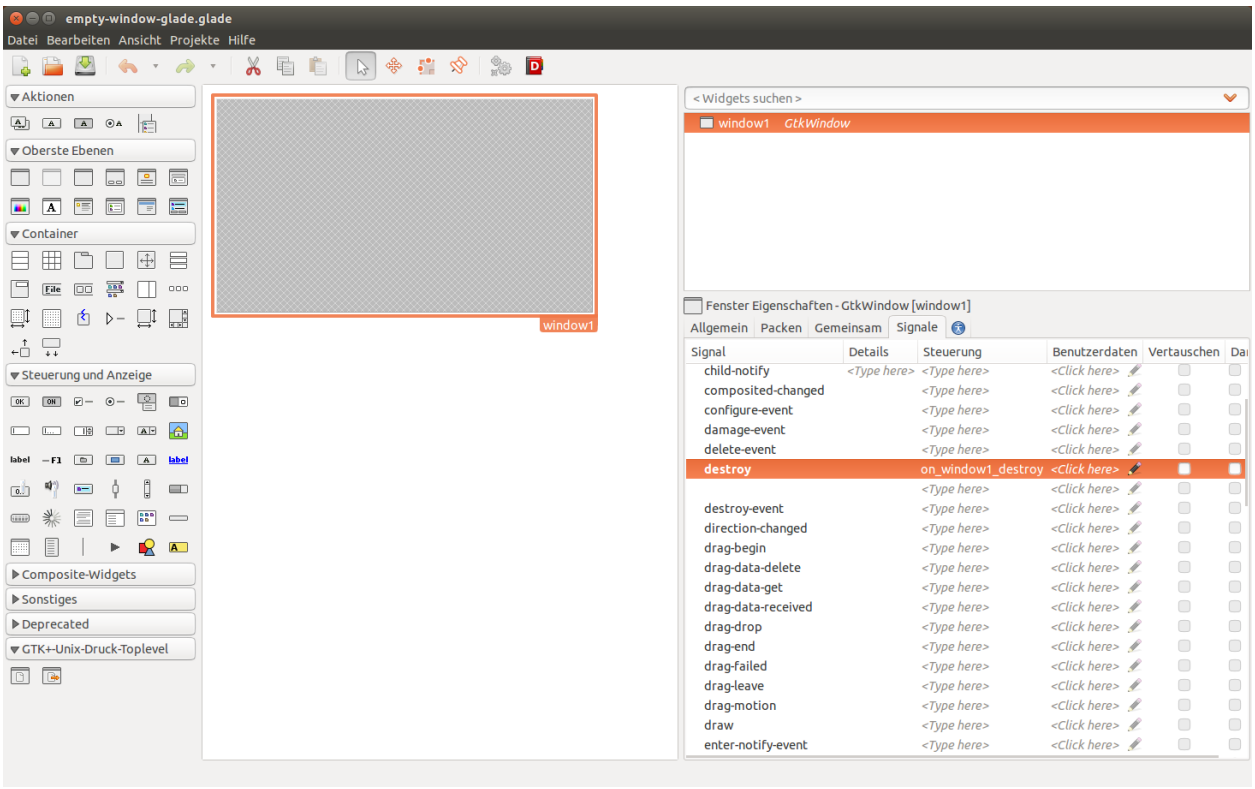


Fig. 4.1: The steps in Glade are: Create a *GtkWindow*, then open the Signals tab and enter *"on_window1_destroy"* for *"GtkWidget –> destroy"*.

This is all we need for an empty (and closable) window. Then we can save the file with the extension *".glade"*. The finished XML-code of that file looks like this and should be fairly easy to understand:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Generated with glade 3.16.1 -->
<interface>
  <requires lib="gtk+" version="3.10"/>
  <object class="GtkWindow" id="window1">
```

```
    <property name="can_focus">False</property>
    <signal name="destroy" handler="on_window1_destroy" swapped="no"/>
    <child>
      <placeholder/>
    </child>
  </object>
</interface>
```

Now we have to create a separate file to hold the python code that will call the Glade-file. The finished code looks like this:

```
#!/usr/bin/python3

from gi.repository import Gtk

class Signals:
    def on_window1_destroy(self, widget):
        Gtk.main_quit()

builder = Gtk.Builder()
builder.add_objects_from_file('empty-window-glade.glade', ('window1', '') )
builder.connect_signals(Signals())

myfirstwindow = builder.get_object('window1')

myfirstwindow.show_all()
Gtk.main()
```

In comparison to the previous approach a few lines of code have changed. First we call the *Gtk.Builder()* function:

```
builder = Gtk.Builder()
```

Then we use the *Gtk.Builder()* to add the objects from the Glade-file. In the bracket we first need to specify the Glade-file, and then a list of objects even if we just want to call one object (*Thankyou errol from http://www.gtkforums.com for this tip*):

```
builder.add_objects_from_file('empty-window-glade.glade', ('window1', '') )
```

Next the builder needs to connect the signals that we defined in the Glade file. The easiest way of doing this is to place the Signals in their own *Class*. We only defined one signal in Glade which was *"on_window1_destroy"*:

```
builder.connect_signals(Signals())

class Signals:
    def on_window1_destroy(self, widget):
        Gtk.main_quit()
```

The last two lines of the program are the same as for the previous example.

Further Reading

- Python GTK+ 3 Tutorial: Getting started
- GTK+ 3 Reference Manual
- GTK+ 3 Reference Manual: GtkBuilder
- Glade - A user interface designer

4.1.2 Embedding Matplotlib

Now that we have an empty window we will learn how to place Matplotlib into it. The main differences are that we need to import Matplotlib-specific packages, insert our Matplotlib-code and place the resulting *FigureCanvas* in a *Gtk.ScrolledWindow* (which is a child-element of the *Gtk.Window*).

We will look at an example that will produce a random radial plot on each application start (adapted from http://matplotlib.org/dev/examples/pie_and_polar_charts/polar_bar_demo.html). The finished Python-code is:

```
#!/usr/bin/python3

from gi.repository import Gtk

from matplotlib.figure import Figure
from numpy import arange, pi, random, linspace
import matplotlib.cm as cm
#Possibly this rendering backend is broken currently
#from matplotlib.backends.backend_gtk3agg import FigureCanvasGTK3Agg as FigureCanvas
from matplotlib.backends.backend_gtk3cairo import FigureCanvasGTK3Cairo as FigureCanvas

myfirstwindow = Gtk.Window()
myfirstwindow.connect("delete-event", Gtk.main_quit)
myfirstwindow.set_default_size(400, 400)

fig = Figure(figsize=(5,5), dpi=100)
ax = fig.add_subplot(111, projection='polar')

N = 20
theta = linspace(0.0, 2 * pi, N, endpoint=False)
radii = 10 * random.rand(N)
width = pi / 4 * random.rand(N)

bars = ax.bar(theta, radii, width=width, bottom=0.0)

for r, bar in zip(radii, bars):
    bar.set_facecolor(cm.jet(r / 10.))
    bar.set_alpha(0.5)

ax.plot()

sw = Gtk.ScrolledWindow()
myfirstwindow.add(sw)

canvas = FigureCanvas(fig)
canvas.set_size_request(400,400)
sw.add_with_viewport(canvas)

myfirstwindow.show_all()
Gtk.main()
```

As you probably noticed we imported a few more modules. The module *matplotlib.figure* is required to render the graph. We need some functions from *NumPy* for evenly dividing an interval (*numpy.arange*), the value for the constant pi (*numpy.pi*), a function for random numbers (*numpy.random*) and a function that returns evenly spaced numbers in an interval (*numpy.linspace*). We also need the colormap function (*matplotlib.cm*). The container in which the graph is rendered is the *FigureCanvasGTK3Agg* or *FigureCanvasGTK3Cairo*.

```
from matplotlib.figure import Figure
from numpy import arange, pi, random, linspace
import matplotlib.cm as cm
```

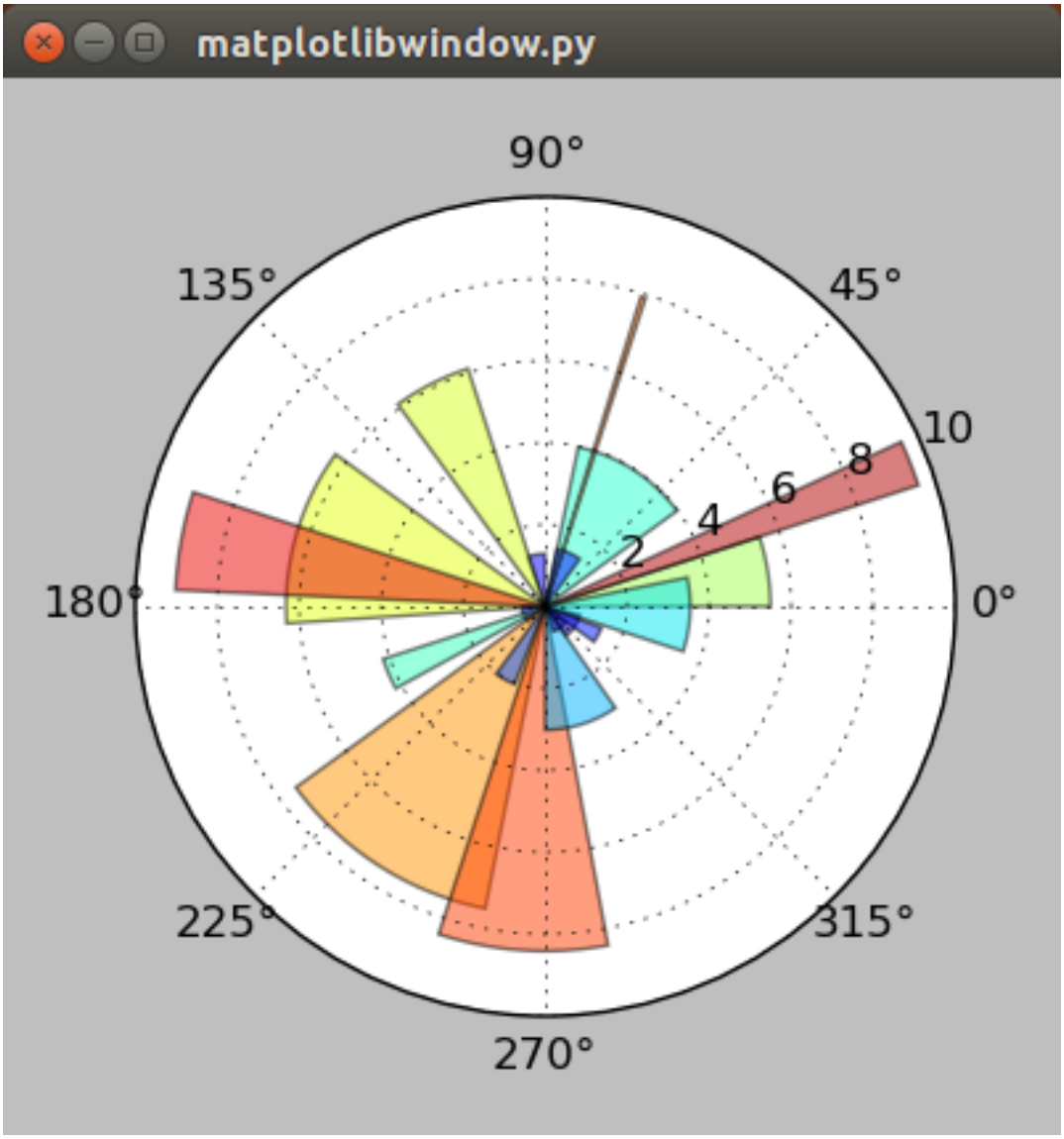


Fig. 4.2: The first window with an embedded Matplotlib-graph as it renders in Ubuntu 14.04.

```
#Possibly this rendering backend is broken currently
#from matplotlib.backends.backend_gtk3agg import FigureCanvasGTK3Agg as FigureCanvas
from matplotlib.backends.backend_gtk3cairo import FigureCanvasGTK3Cairo as FigureCanvas
```

In order to give the plot a sufficient default size we add this line:

```
myfirstwindow.set_default_size(400, 400)
```

Next we create an instance of a *matplotlib.figure* and define its size and resolution. The resolution is a fixed value, which means that your plots will probably not look the same on newer display with very high pixel-densities. 100 dpi ("dots per inch", which is a very annoying non-metric unit) works well for regular screens which is about 40 pixels per cm.

```
fig = Figure(figsize=(5,5), dpi=100)
```

Then we add a subplot to the plot (This sounds a little weird for only one graph, but you can acutally add many subplots to one plot). 111 means that we have a 1 x 1 grid and are putting the subplot in the 1st cell. Because we want to create a polar plot, we have to set *"projection='polar"'*.

```
ax = fig.add_subplot(111, projection='polar')
```

Next we will define the number of intervals, and divide a full circle (in radian-units: 2*pi) by that number. Then we create random arrays for 20 radiis and 20 widths for the histogram bars. You can test if the program uses an array by trying *"radii = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]"* to create bars that have radii from 1 to 20.

```
N = 20
theta = linspace(0.0, 2 * pi, N, endpoint=False)
radii = 10 * random.rand(N)
width = pi / 4 * random.rand(N)
```

Then we assign the newly created bars to our plot and store them in a variable:

```
bars = ax.bar(theta, radii, width=width, bottom=0.0)
```

We can use this variable *"bars"* to customize the plot. We will do this for the color in the next piece of code. This algorithm assigns each radius a different color using the *"cm.jet"* color scheme (See Matplotlib: Colormaps reference for more color maps). Additionaly the alpha of each histogram-bar is set to "0.5".

```
for r, bar in zip(radii, bars):
    bar.set_facecolor(cm.jet(r / 10.))
    bar.set_alpha(0.5)
```

In the next step we plot the generated graph:

```
ax.plot()
```

Then we just have to create a *GtkScrolledWindow*, and add it to our *GtkWindow*.

```
sw = Gtk.ScrolledWindow()
myfirstwindow.add(sw)
```

Finally we can create an instance of a *FigureCanvasGTK3Agg* or *FigureCanvasGTK3Cairo* with our figure included in it. Then we set the size of the canvas and add embed it into the *GtkScrolledWindow*.

```
canvas = FigureCanvas(fig)
canvas.set_size_request(400,400)
sw.add_with_viewport(canvas)
```

Embedding Matplotlib with Glade

Recreating the previous example with Glade requires just some minor changes to the Glade-file and a slightly different Python-3-code.

First we need to add a *GtkScrolledWindow* to our empty window. Although the name has "window" in it, it actually is more like a canvas for other widgets. In the Python code we will place in the next step a *FigureCanvas* into the *GtkScrolledWindow*. In order to give the plot more space, we can also set the default width and height of *"window1"* to 400 px each.

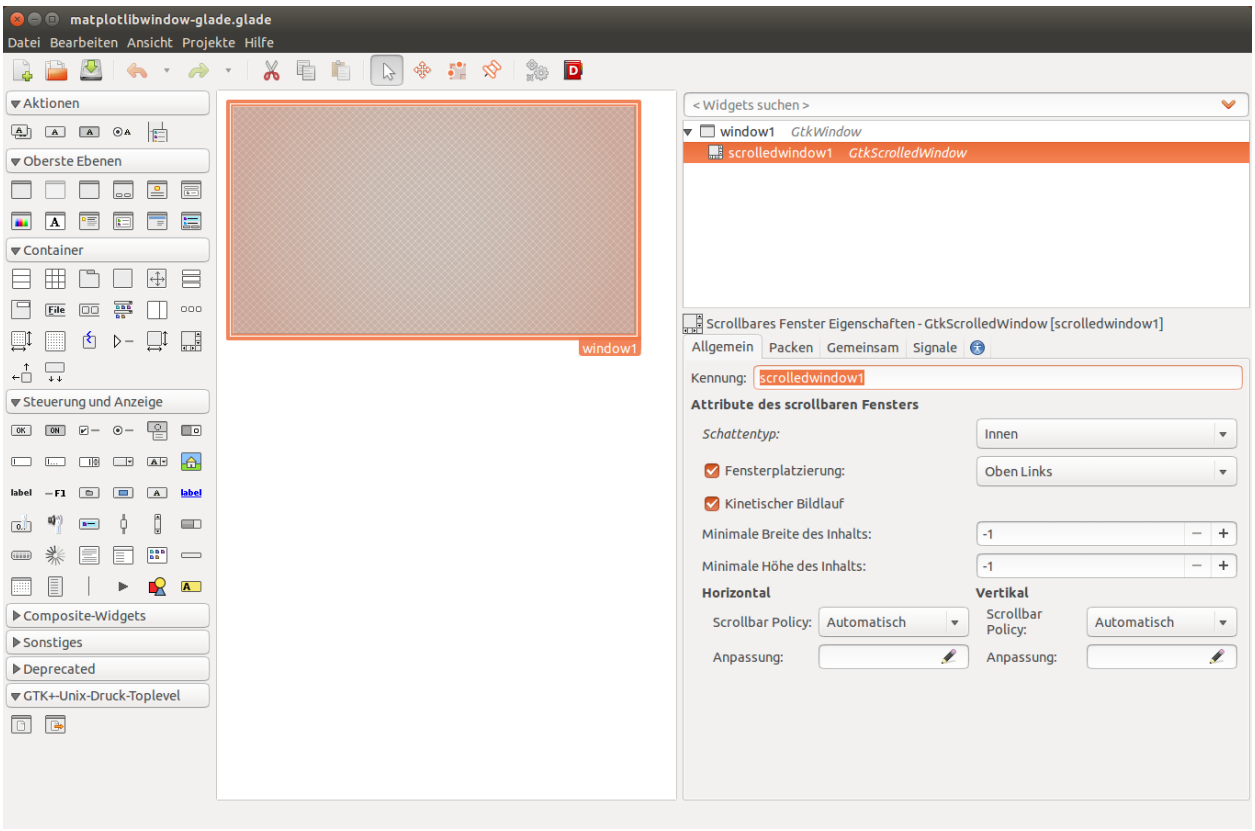


Fig. 4.3: Starting with the previous example all we need to add is a *GtkScrolledWindow*.

The XML-code of the Glade-file after the modifications looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Generated with glade 3.16.1 -->
<interface>
  <requires lib="gtk+" version="3.10"/>
  <object class="GtkWindow" id="window1">
    <property name="can_focus">False</property>
    <property name="default_width">400</property>
    <property name="default_height">400</property>
    <signal name="destroy" handler="on_window1_destroy" swapped="no"/>
    <child>
      <object class="GtkScrolledWindow" id="scrolledwindow1">
        <property name="visible">True</property>
        <property name="can_focus">True</property>
        <property name="shadow_type">in</property>
        <child>
```

```
          <placeholder/>
        </child>
      </object>
    </child>
  </object>
</interface>
```

Starting with the code from the previous examples we only have to make slight changes to port this example. This is the final result:

```
#!/usr/bin/python3

from gi.repository import Gtk

from matplotlib.figure import Figure
from numpy import arange, pi, random, linspace
import matplotlib.cm as cm
#Possibly this rendering backend is broken currently
#from matplotlib.backends.backend_gtk3agg import FigureCanvasGTK3Agg as FigureCanvas
from matplotlib.backends.backend_gtk3cairo import FigureCanvasGTK3Cairo as FigureCanvas

class Signals:
    def on_window1_destroy(self, widget):
        Gtk.main_quit()

builder = Gtk.Builder()
builder.add_objects_from_file('matplotlibwindow-glade.glade', ('window1', '') )
builder.connect_signals(Signals())

myfirstwindow = builder.get_object('window1')
sw = builder.get_object('scrolledwindow1')

fig = Figure(figsize=(5,5), dpi=100)
ax = fig.add_subplot(111, projection='polar')

N = 20
theta = linspace(0.0, 2 * pi, N, endpoint=False)
radii = 10 * random.rand(N)
width = pi / 4 * random.rand(N)

bars = ax.bar(theta, radii, width=width, bottom=0.0)

for r, bar in zip(radii, bars):
    bar.set_facecolor(cm.jet(r / 10.))
    bar.set_alpha(0.5)

ax.plot()

canvas = FigureCanvas(fig)
sw.add_with_viewport(canvas)

myfirstwindow.show_all()
Gtk.main()
```

The most important difference to the non-Glade code is that we need to get our *GtkWindow* and *GtkScrolledWindow* from Glade using the *GtkBuilder*:

```
myfirstwindow = builder.get_object('window1')
sw = builder.get_object('scrolledwindow1')
```

Further Reading

- The Python Standard Library: Built-in Functions: zip
- Numpy
- The Matplotlib API: pyplot
- The Matplotlib API: figure
- The Matplotlib API: cm (colormap)
- Matplotlib: Colormaps reference
- Pyplot tutorial
- Numpy: linspace
- Numpy: arange
- Numpy: random
- Numpy: random.rand
- FigureCanvasGTK3Agg documentation [[2014-06-21 Find link for documentation]]
- FigureCanvasGTK3Cairo documentaion [Link needed]
- Stackoverflow question about subplot grids

4.2 Matplotlib-Toolbar

In the second chapter we will design a window that includes the Matplotlib-Toolbar (or Navigation-Toolbar) that is a part of the Matplotlib-API. It has a few very useful functions, for manipulating the view of the plot, cursor tracking, and most importantly a save-button, that allows the user to save a "png" or "svg" of the plot. As a bonus we will customize the window with an icon and a title, and customize the plot with some annotations.

In order to add the toolbar we have to subdivide our *GtkWindow* into layout containers. One of the containers is going to hold the graph, while the other one will hold the Navigation-Toolbar. If you want to do the layout inside of the code you should familarize yourself with GTK layout containers (Python GTK+ 3 Tutorial: Layout Containers). In Glade, setting up the layout can be done graphically, and the required containers are called with the *GtkBuilder* function. In both cases the Navigation-Toolbar can be imported like this:

```
from matplotlib.backends.backend_gtk3 import NavigationToolbar2GTK3 as NavigationToolbar
```

4.2.1 Matplotlib-Toolbar using GTK3

Assuming that you worked through the first chapter, you should already be slightly familar with the structure of the program. First we need to import the required packages and functions:

```
from gi.repository import Gtk

from matplotlib.figure import Figure
from numpy import sin, cos, pi, linspace
#Possibly this rendering backend is broken currently
```

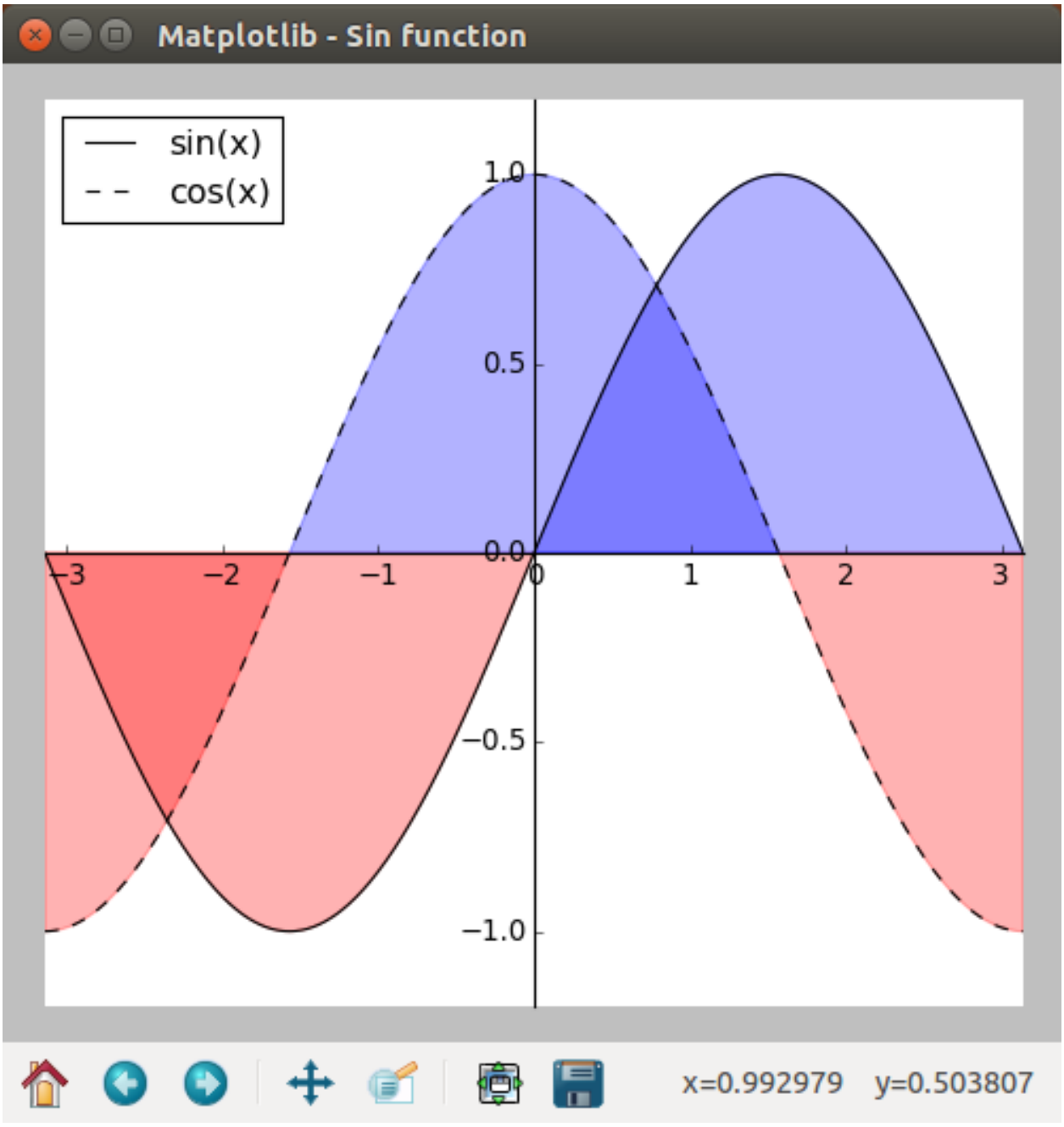


Fig. 4.4: In this chapter we will program an example that plots a sine and cosine function and includes the Navigation-Toolbar.

```
#from matplotlib.backends.backend_gtk3agg import FigureCanvasGTK3Agg as FigureCanvas
from matplotlib.backends.backend_gtk3cairo import FigureCanvasGTK3Cairo as FigureCanvas    from matpl
```

The next lines of code create a *GtkWindow* and in addition to lines from the first chapter we will also define a title and a icon (from a file in the same directory) for our program window:

```
myfirstwindow = Gtk.Window()
myfirstwindow.connect("delete-event", Gtk.main_quit)
myfirstwindow.set_default_size(500, 500)
myfirstwindow.set_title('Matplotlib - Sin function')
myfirstwindow.set_icon_from_file('testicon.svg')
```

Because we need multiple containers for this example we will create a vertical Gtk.Box. Then we can add it to the main window:

```
box = Gtk.Box(orientation=Gtk.Orientation.VERTICAL)
myfirstwindow.add(box)
```

Note that the Gtk.Box is an empty. Later in the code we will add individual containers to it.

Then we can start with the Matplotlib-specific code. First we create a figure, this time choosing a lower resolution of 80 dpi (About 30 pixel per cm) to make the numbers a little smaller than the example from the first chapter. We also create a single subplot inside that figure:

```
fig = Figure(figsize=(5,5), dpi=80)
ax = fig.add_subplot(111)
```

Next we divide the distance between -pi and pi into 1000 steps and calculate sin and cos for them:

```
n = 1000
xsin = linspace(-pi, pi, n, endpoint=True)
xcos = linspace(-pi, pi, n, endpoint=True)
ysin = sin(xsin)
ycos = cos(xcos)
```

Within the subplot we plot two function. One for sin and one for cos. Both functions are labelled for the plot-key and the cos-function gets a dashed line:

```
sinwave = ax.plot(xsin, ysin, color='black', label='sin(x)')
coswave = ax.plot(xcos, ycos, color='black', label='cos(x)', linestyle='--')
```

Then we can define the limits of the view we will see when the window opens. A good value should be close to the range of the data:

```
ax.set_xlim(-pi,pi)
ax.set_ylim(-1.2,1.2)
```

For both functions we want to shade the area between the function and the x-xis. This can be done with the *"fill_between"* function that can be easily adjusted:

```
ax.fill_between(xsin, 0, ysin, (ysin - 1) > -1, color='blue', alpha=.3)
ax.fill_between(xsin, 0, ysin, (ysin - 1) < -1, color='red',  alpha=.3)
ax.fill_between(xcos, 0, ycos, (ycos - 1) > -1, color='blue', alpha=.3)
ax.fill_between(xcos, 0, ycos, (ycos - 1) < -1, color='red',  alpha=.3)
```

Next we need to find a place for our legend and add it to "ax":

```
ax.legend(loc='upper left')
```

In this example we wan't to move the x-axis and y-axis to the center of the plot. Similar to how graphs are often seen in mathematics. First we excecute "fig.gca()" which "gets the current axis" of the figure. Then we can format the spines to intersect at the the (0|0) point of the graph:

```
ax = fig.gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))
```

The final Matplotlib-specific line helps to correct the layout of the whole figure. The function is still experimental, will sometimes produce a "UserWarning", but will often improve the problems with sizing and overlap:

```
fig.tight_layout()
```

Then we can build a FigureCanvas from our figure and add it to a *GtkBox*:

```
canvas = FigureCanvas(fig)
box.pack_start(canvas, True, True, 0)
```

The second *GtkBox* contains the Navigations-Toolbar. The *"box.pack_start"* will get the second argument "expand=False" which ensures that the box containg the graph will get all the space it needs:

```
toolbar = NavigationToolbar(canvas, myfirstwindow)
box.pack_start(toolbar, False, True, 0)
```

The last line of the code just show the window and start the main program loop. The complete code is:

```
#!/usr/bin/python3

from gi.repository import Gtk

from matplotlib.figure import Figure
from numpy import sin, cos, pi, linspace
#Possibly this rendering backend is broken currently
#from matplotlib.backends.backend_gtk3agg import FigureCanvasGTK3Agg as FigureCanvas
from matplotlib.backends.backend_gtk3cairo import FigureCanvasGTK3Cairo as FigureCanvas    from matpl

myfirstwindow = Gtk.Window()
myfirstwindow.connect("delete-event", Gtk.main_quit)
myfirstwindow.set_default_size(500, 500)
myfirstwindow.set_title('Matplotlib')
myfirstwindow.set_icon_from_file('testicon.svg')

box = Gtk.Box(orientation=Gtk.Orientation.VERTICAL)
myfirstwindow.add(box)

fig = Figure(figsize=(5,5), dpi=80)
ax = fig.add_subplot(111)

n = 1000
xsin = linspace(-pi, pi, n, endpoint=True)
xcos = linspace(-pi, pi, n, endpoint=True)
ysin = sin(xsin)
ycos = cos(xcos)

sinwave = ax.plot(xsin, ysin, color='black', label='sin(x)')
coswave = ax.plot(xcos, ycos, color='black', label='cos(x)', linestyle='--')
```

```
ax.set_xlim(-pi,pi)
ax.set_ylim(-1.2,1.2)

ax.fill_between(xsin, 0, ysin, (ysin - 1) > -1, color='blue', alpha=.3)
ax.fill_between(xsin, 0, ysin, (ysin - 1) < -1, color='red',  alpha=.3)
ax.fill_between(xcos, 0, ycos, (ycos - 1) > -1, color='blue', alpha=.3)
ax.fill_between(xcos, 0, ycos, (ycos - 1) < -1, color='red',  alpha=.3)

ax.legend(loc='upper left')

ax = fig.gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))

fig.tight_layout()

canvas = FigureCanvas(fig)
box.pack_start(canvas, True, True, 0)

toolbar = NavigationToolbar(canvas, myfirstwindow)
box.pack_start(toolbar, False, True, 0)

myfirstwindow.show_all()
Gtk.main()
```

4.2.2 Matplotlib-Toolbar with Glade

The same example using Glade requires a vertical box with two containers. Each container gets one *GtkScrolledWindow*:

Then we still have to set the packing of the *"ScrolledWindow1"* to "Expand = Yes". This will again ensure that the graph gets all the space it needs and the toolbar will be resized to a minimum.

The final XML-code from the Glade-file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Generated with glade 3.16.1 -->
<interface>
  <requires lib="gtk+" version="3.10"/>
  <object class="GtkWindow" id="window1">
    <property name="can_focus">False</property>
    <property name="title" translatable="yes">Matplotlib</property>
    <property name="default_width">400</property>
    <property name="default_height">400</property>
    <property name="icon">testicon.svg</property>
    <signal name="destroy" handler="on_window1_destroy" swapped="no"/>
    <child>
      <object class="GtkBox" id="box1">
        <property name="visible">True</property>
        <property name="can_focus">False</property>
        <property name="orientation">vertical</property>
        <child>
          <object class="GtkScrolledWindow" id="scrolledwindow1">
```

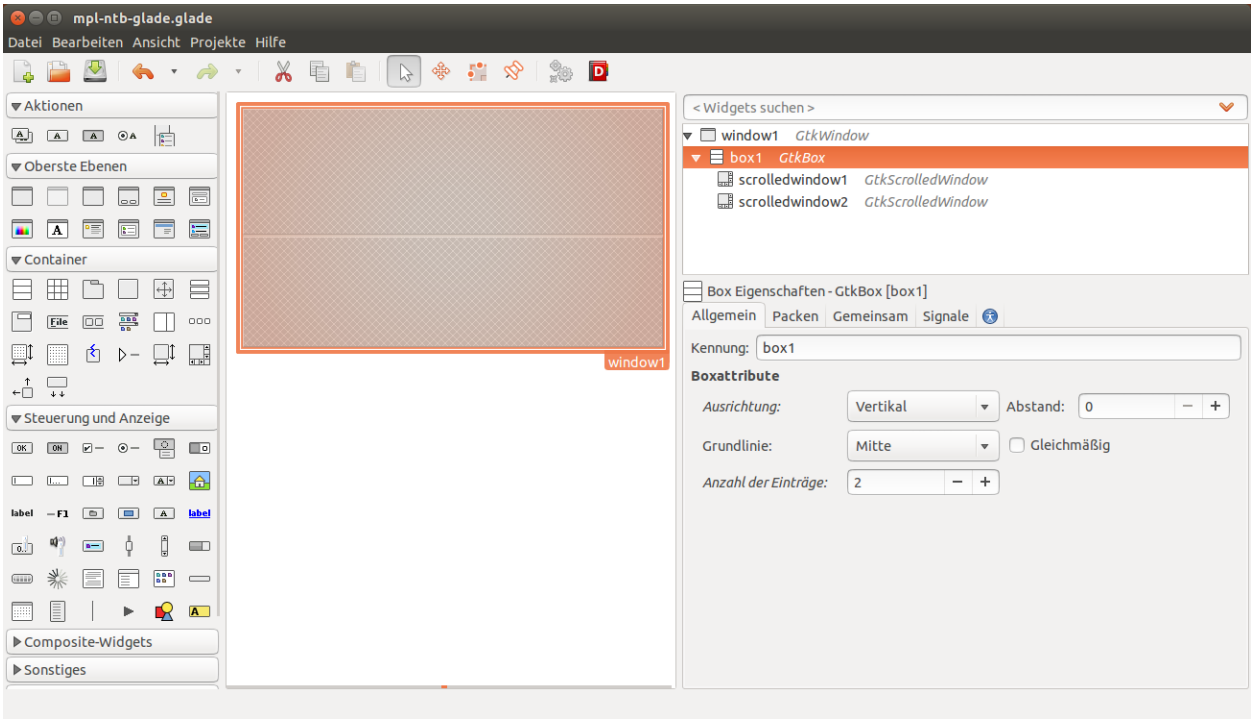


Fig. 4.5: This is the layout we have to create in Glade in order to make room for the graph and the Navigation-Toolbar.

```
            <property name="visible">True</property>
            <property name="can_focus">True</property>
            <property name="shadow_type">in</property>
            <child>
              <placeholder/>
            </child>
          </object>
          <packing>
            <property name="expand">True</property>
            <property name="fill">True</property>
            <property name="position">0</property>
          </packing>
        </child>
        <child>
          <object class="GtkScrolledWindow" id="scrolledwindow2">
            <property name="visible">True</property>
            <property name="can_focus">True</property>
            <property name="hscrollbar_policy">never</property>
            <property name="vscrollbar_policy">never</property>
            <property name="shadow_type">in</property>
            <child>
              <placeholder/>
            </child>
          </object>
          <packing>
            <property name="expand">False</property>
            <property name="fill">True</property>
            <property name="position">1</property>
          </packing>
        </child>
```

```
      </object>
    </child>
  </object>
</interface>
```

The Python code from above only needs minor adjustments. The two containers (*GtkScrolledWindow*) are called by the *GtkBuilder*:

```
#!/usr/bin/python3

from gi.repository import Gtk

from matplotlib.figure import Figure
from numpy import sin, cos, pi, linspace
#Possibly this rendering backend is broken currently
#from matplotlib.backends.backend_gtk3agg import FigureCanvasGTK3Agg as FigureCanvas
from matplotlib.backends.backend_gtk3cairo import FigureCanvasGTK3Cairo as FigureCanvas    from matpl

class Signals:
    def on_window1_destroy(self, widget):
        Gtk.main_quit()

builder = Gtk.Builder()
builder.add_objects_from_file('mpl-ntb-glade.glade', ('window1', '') )
builder.connect_signals(Signals())

myfirstwindow = builder.get_object('window1')
sw = builder.get_object('scrolledwindow1')
sw2 = builder.get_object('scrolledwindow2')

fig = Figure(figsize=(5,5), dpi=80)
ax = fig.add_subplot(111)

n = 1000
xsin = linspace(-pi, pi, n, endpoint=True)
xcos = linspace(-pi, pi, n, endpoint=True)
ysin = sin(xsin)
ycos = cos(xcos)

sinwave = ax.plot(xsin, ysin, color='black', label='sin(x)')
coswave = ax.plot(xcos, ycos, color='black', label='cos(x)', linestyle='--')

ax.set_xlim(-pi,pi)
ax.set_ylim(-1.2,1.2)

ax.fill_between(xsin, 0, ysin, (ysin - 1) > -1, color='blue', alpha=.3)
ax.fill_between(xsin, 0, ysin, (ysin - 1) < -1, color='red',  alpha=.3)
ax.fill_between(xcos, 0, ycos, (ycos - 1) > -1, color='blue', alpha=.3)
ax.fill_between(xcos, 0, ycos, (ycos - 1) < -1, color='red',  alpha=.3)

ax.legend(loc='upper left')

ax = fig.gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
ax.yaxis.set_ticks_position('left')
```

```
ax.spines['left'].set_position(('data',0))

fig.tight_layout()

canvas = FigureCanvas(fig)
sw.add_with_viewport(canvas)

toolbar = NavigationToolbar(canvas, myfirstwindow)
sw2.add_with_viewport(toolbar)

myfirstwindow.show_all()
Gtk.main()
```

4.2.3 Further reading

- Matplotlib: Interactive navigation
- Python GTK+ 3 Tutorial: Layout Containers
- GTK3 API: GtkBox
- Python Scientific Lecture Notes: Matplotlib plotting: Annotate some points
- Matplotlib API: Spines

4.3 Zooming in on data

This chapter will show how you can zoom in on your data. This is a common feature of programs for data analysis. This is the first tutorial that will make use of *classes*. If you are new to Python you might want to review some simple examples first before reading on (See *Further Reading* at bottom of page).

4.3.1 Simple Zoom-Subplot

In the first example we will create a simple collection of random points, with random size and random color. Zooming on some of the points will create a rectangle that highlights the zoomed area. The zoom-graph itself will show the points enlarged, as we don't want to produce mileading precision (*This will be useful in the example with error bars*).

To get you motivated or if you want to try to build the program yourself the next figure will show the finished example I came up with.

At the beginning of the code we will import the usual modules. New is the import of the *Rectangle* function which we will use to plot a rectangle within the graph.

```
#!/usr/bin/python3

from gi.repository import Gtk
from matplotlib.figure import Figure
from numpy import random
#Possibly this rendering backend is broken currently
#from matplotlib.backends.backend_gtk3agg import FigureCanvasGTK3Agg as FigureCanvas
from matplotlib.backends.backend_gtk3cairo import FigureCanvasGTK3Cairo as FigureCanvas    from matpl
from matplotlib.patches import Rectangle
```

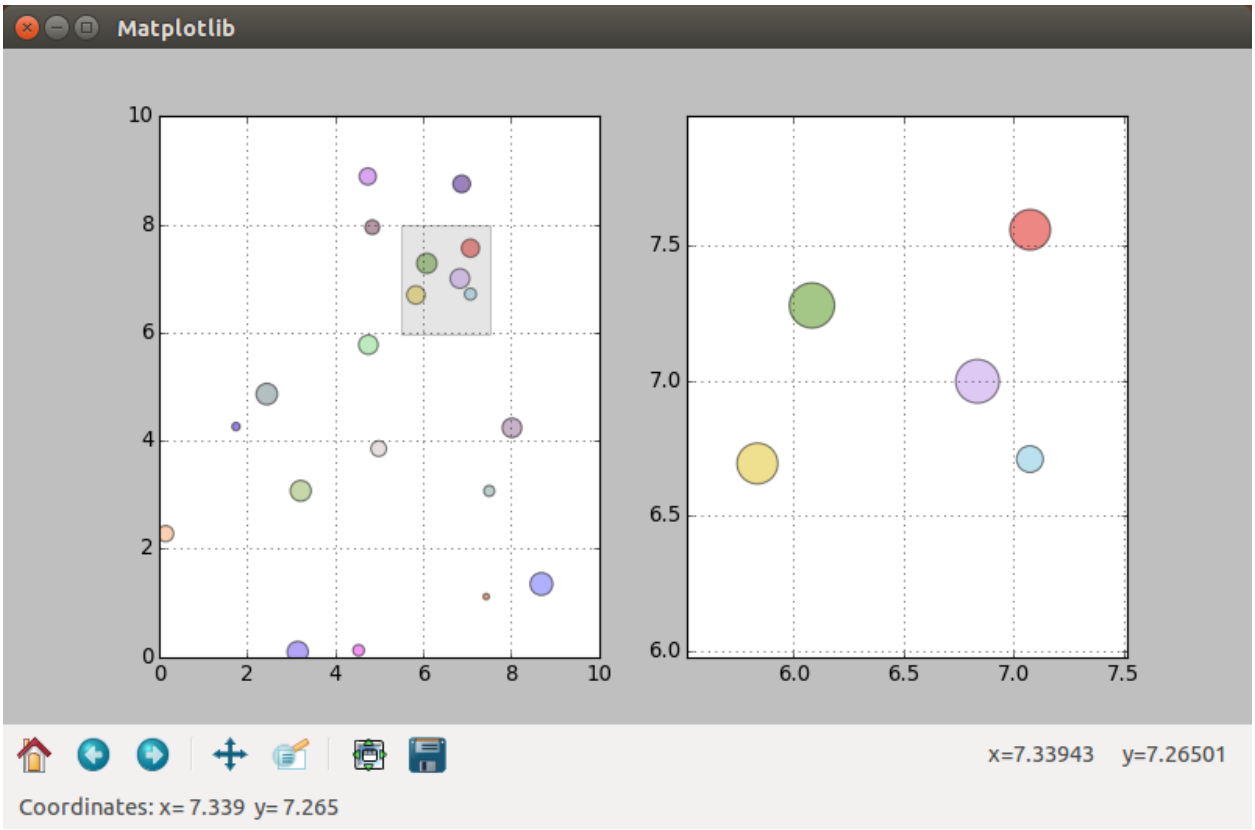


Fig. 4.6: The left window shows the original data and the right window the zoomed data. The zoomed area is highlighted and the points are enlarged according to the zoomfactor.

The *DrawPoints* class will do most of the work in this example. On init, we will define some variables and set up the 2 subplots. Each subplot has its own drawing function, which clears the subplot, before drawing. The *zoom* function recycles the *draw* and *drawzoom* function.

```
class DrawPoints:
    '''Creates random points, 2 axis on 1 figure on 1 canvas on init. Allows for drawing and zooming
    def __init__(self):
        self.n = 20
        self.xrand = random.rand(1,self.n)*10
        self.yrand = random.rand(1,self.n)*10
        self.randsize = random.rand(1,self.n)*200
        self.randcolor = random.rand(self.n,3)

        self.fig = Figure(figsize=(10,10), dpi=80)
        self.ax = self.fig.add_subplot(121)
        self.axzoom = self.fig.add_subplot(122)
        self.canvas = FigureCanvas(self.fig)
    def draw(self):
        '''Draws the ax-subplot'''
        self.ax.cla()
        self.ax.grid(True)
        self.ax.set_xlim(0,10)
        self.ax.set_ylim(0,10)
        self.ax.scatter(self.xrand, self.yrand, marker='o', s=self.randsize, c=self.randcolor, alpha=
    def drawzoom(self):
        '''Draws the axzoom-suplot'''
        self.axzoom.cla()
        self.axzoom.grid(True)
        self.axzoom.set_xlim(self.x-1, self.x+1)
        self.axzoom.set_ylim(self.y-1, self.y+1)
        self.axzoom.scatter(self.xrand, self.yrand, marker='o', s=self.randsize*5, c=self.randcolor,
    def zoom(self, x, y):
        '''Adds a rectangle to the zoomed area of the ax-graph and updates the axzoom-graph'''
        self.x = x
        self.y = y
        self.draw()
        self.drawzoom()
        self.ax.add_patch(Rectangle((x - 1, y - 1), 2, 2, facecolor="grey", alpha=0.2))
        self.fig.canvas.draw()
```

The function *updatecursorposition* is activated when the mouse moves over the graph. The x and y coordinates are then send to the *GtkStatusbar*.

```
def updatecursorposition(event):
    '''When cursor inside plot, get position and print to statusbar'''
    if event.inaxes:
        x = event.xdata
        y = event.ydata
        statbar.push(1, ("Coordinates:" + " x= " + str(round(x,3)) + "  y= " + str(round(y,3))))
```

The *updatezoom* function is activated when the mouse is clicked inside the graph. It then checks for the left mouse button, and if the event is a number (meaning not the Python None-Type). Then the coordinates are forwarded to the *zoom* function (Later in the program we will create an instance of the class called *points()* and that is why we call *points.zoom()*).

```
def updatezoom(event):
    '''When mouse is right-clicked on the canvas get the coordiantes and send them to points.zoom'''
    if event.button!=1: return
    if (event.xdata is None): return
```

```
    x,y = event.xdata, event.ydata
    points.zoom(x,y)
```

The rest of the program goes through the usual setting up, of the Gtk window and the containers. We also define an instance of the *DrawPoints* class and call *points.draw() to draw the initial graph. Two event triggers are connected with the canvas (*points.fig.canvas*) and the appropriate functions are called (*updatecursorposition* for mouse motion and *updatezoom* for mouse clicks)

```
window = Gtk.Window()
window.connect("delete-event", Gtk.main_quit)
window.set_default_size(800, 500)
window.set_title('Matplotlib')

box = Gtk.Box(orientation=Gtk.Orientation.VERTICAL)
window.add(box)

points = DrawPoints()
points.draw()

box.pack_start(points.canvas, True, True, 0)

toolbar = NavigationToolbar(points.canvas, window)
box.pack_start(toolbar, False, True, 0)

statbar = Gtk.Statusbar()
box.pack_start(statbar, False, True, 0)

points.fig.canvas.mpl_connect('motion_notify_event', updatecursorposition)
points.fig.canvas.mpl_connect('button_press_event', updatezoom)

window.show_all()
Gtk.main()
```

This is the code of the whole example:

```
#!/usr/bin/python3

from gi.repository import Gtk
from matplotlib.figure import Figure
from numpy import random
#Possibly this rendering backend is broken currently
#from matplotlib.backends.backend_gtk3agg import FigureCanvasGTK3Agg as FigureCanvas
from matplotlib.backends.backend_gtk3cairo import FigureCanvasGTK3Cairo as FigureCanvas    from matpl
from matplotlib.patches import Rectangle

class DrawPoints:
    '''Creates random points, 2 axis on 1 figure on 1 canvas on init. Allows for drawing and zooming
    def __init__(self):
        self.n = 20
        self.xrand = random.rand(1,self.n)*10
        self.yrand = random.rand(1,self.n)*10
        self.randsize = random.rand(1,self.n)*200
        self.randcolor = random.rand(self.n,3)

        self.fig = Figure(figsize=(10,10), dpi=80)
        self.ax = self.fig.add_subplot(121)
        self.axzoom = self.fig.add_subplot(122)
        self.canvas = FigureCanvas(self.fig)
    def draw(self):
```

```
        '''Draws the ax-subplot'''
        self.ax.cla()
        self.ax.grid(True)
        self.ax.set_xlim(0,10)
        self.ax.set_ylim(0,10)
        self.ax.scatter(self.xrand, self.yrand, marker='o', s=self.randsize, c=self.randcolor, alpha=
    def drawzoom(self):
        '''Draws the axzoom-suplot'''
        self.axzoom.cla()
        self.axzoom.grid(True)
        self.axzoom.set_xlim(self.x-1, self.x+1)
        self.axzoom.set_ylim(self.y-1, self.y+1)
        self.axzoom.scatter(self.xrand, self.yrand, marker='o', s=self.randsize*5, c=self.randcolor,
    def zoom(self, x, y):
        '''Adds a rectangle to the zoomed area of the ax-graph and updates the axzoom-graph'''
        self.x = x
        self.y = y
        self.draw()
        self.drawzoom()
        self.ax.add_patch(Rectangle((x - 1, y - 1), 2, 2, facecolor="grey", alpha=0.2))
        self.fig.canvas.draw()

def updatecursorposition(event):
    '''When cursor inside plot, get position and print to statusbar'''
    if event.inaxes:
        x = event.xdata
        y = event.ydata
        statbar.push(1, ("Coordinates:" + " x= " + str(round(x,3)) + "  y= " + str(round(y,3))))

def updatezoom(event):
    '''When mouse is right-clicked on the canvas get the coordiantes and send them to points.zoom'''
    if event.button!=1: return
    if (event.xdata is None): return
    x,y = event.xdata, event.ydata
    points.zoom(x,y)

window = Gtk.Window()
window.connect("delete-event", Gtk.main_quit)
window.set_default_size(800, 500)
window.set_title('Matplotlib')

box = Gtk.Box(orientation=Gtk.Orientation.VERTICAL)
window.add(box)

points = DrawPoints()
points.draw()

box.pack_start(points.canvas, True, True, 0)

toolbar = NavigationToolbar(points.canvas, window)
box.pack_start(toolbar, False, True, 0)

statbar = Gtk.Statusbar()
box.pack_start(statbar, False, True, 0)

points.fig.canvas.mpl_connect('motion_notify_event', updatecursorposition)
points.fig.canvas.mpl_connect('button_press_event', updatezoom)
```

```
window.show_all()
Gtk.main()
```

Simple Zoom-Subplot with Glade

This example doesn't require any additional Glade structures than the examples before. Just another container for the *GtkToolbar*. You should have no trouble converting the example yourself.

4.3.2 Further Reading

- The Python Tutorial: Classes
- Matplotlib API Patch collection example

4.4 Entering data

In this chapter we will build the first program that will allow data to be entered and immedietly graphed. The data will be (temporarily) stored in a *Gtk.ListStore* and viewed and changed by using *Gtk.TreeView*. A program like this might be useful for class-room demonstrations, but in the next chapters we will add data input and output to these examples to build fully fledged data processing programs.

4.4.1 Point plotting with Gtk.TreeView

The first example we will build will consist of a table (a *GTKTreeView*) and a Matplotlib-graph. The table has two columns for x-y-coordinates which will be plotted in the graph.

The program requires the following imports:

```
#!/usr/bin/python3

from gi.repository import Gtk
from matplotlib.figure import Figure
#Possibly this rendering backend is broken currently
#from matplotlib.backends.backend_gtk3agg import FigureCanvasGTK3Agg as FigureCanvas
from matplotlib.backends.backend_gtk3cairo import FigureCanvasGTK3Cairo as FigureCanvas
```

I decided to put the whole program into a class. First we have to set up the window, the layout and a *Gtk.Toolbar*. The two lines starting with *self.context* ensure that the toolbar will be styled similar to your operating system.

```
class MainClass():
    def __init__(self):
        self.window = Gtk.Window()
        self.window.set_default_size(800, 500)
        self.boxvertical = Gtk.Box(orientation=Gtk.Orientation.VERTICAL)
        self.window.add(self.boxvertical)

        self.toolbar = Gtk.Toolbar()
        self.context = self.toolbar.get_style_context()
        self.context.add_class(Gtk.STYLE_CLASS_PRIMARY_TOOLBAR)
        self.boxvertical.pack_start(self.toolbar, False, False, 0)
```

Then we have to create two buttons and add them to the toolbar on position 0 and 1. Then add the toolbar to our layout. The two buttons are also connected to their respective function when they are clicked.

```
self.addbutton = Gtk.ToolButton(Gtk.STOCK_ADD)
self.removebutton = Gtk.ToolButton(Gtk.STOCK_REMOVE)

self.toolbar.insert(self.addbutton, 0)
self.toolbar.insert(self.removebutton, 1)

self.addbutton.connect("clicked", self.addrow)
self.removebutton.connect("clicked", self.removerow)

self.box = Gtk.Box()
self.boxvertical.pack_start(self.box, True, True, 0)
```

Next we set up the figure and add it to the layout.

```
self.fig = Figure(figsize=(10,10), dpi=80)
self.ax = self.fig.add_subplot(111)
self.canvas = FigureCanvas(self.fig)
self.box.pack_start(self.canvas, True, True, 0)
```

Next we will set up the *Gtk.ListStore* and the *Gtk.TreeView*, connect the columns to their respective function. We can also add two values to the columns, just that the graph and table won't be empty on startup.

```
self.liststore = Gtk.ListStore(float, float)
self.treeview = Gtk.TreeView(model=self.liststore)
self.box.pack_start(self.treeview, False, True, 0)

self.xrenderer = Gtk.CellRendererText()
self.xrenderer.set_property("editable", True)
self.xcolumn = Gtk.TreeViewColumn("x-Value", self.xrenderer, text=0)
self.xcolumn.set_min_width(100)
self.xcolumn.set_alignment(0.5)
self.treeview.append_column(self.xcolumn)

self.yrenderer = Gtk.CellRendererText()
self.yrenderer.set_property("editable", True)
self.ycolumn = Gtk.TreeViewColumn("y-Value", self.yrenderer, text=1)
self.ycolumn.set_min_width(100)
self.ycolumn.set_alignment(0.5)
self.treeview.append_column(self.ycolumn)

self.xrenderer.connect("edited", self.xedited)
self.yrenderer.connect("edited", self.yedited)

self.liststore.append([2.35, 2.40])
self.liststore.append([3.45, 4.70])
```

The *resetplot* function clears the axis, resets the limits and recreates the grid:

```
def resetplot(self):
    self.ax.cla()
    self.ax.set_xlim(0,10)
    self.ax.set_ylim(0,10)
    self.ax.grid(True)
```

The *plotpoints* function calls the *resetplot* function and iterates over the rows of the liststore. For each row one point is created. Then the *fig.canvas.draw()* command updates the plot.

```
def plotpoints(self):
    self.resetplot()
```

```
    for row in self.liststore:
        self.ax.scatter(row[:1], row[1:], marker='o', s=50)
    self.fig.canvas.draw()
```

The *xedited* and *yedited* functions first ensure that the comma is conveerted to a ".", so Python can work with them. This is important so your program will work independent of the format of the decimal point. The value is then added to the *Gtk.ListStore*. Then the *plotpoints* function is called.

```
def xedited(self, widget, path, number):
    self.liststore[path][0] = float(number.replace(',', '.'))
    self.plotpoints()

def yedited(self, widget, path, number):
    self.liststore[path][1] = float(number.replace(',', '.'))
    self.plotpoints()
```

The *addrow* and *removerow* function appends or removes a row from the *Gtk.ListStore*. To remove a row, we first have to query which row is currently selected. Either action calls the *plotpoints* funtion.

```
def addrow(self, widget):
    self.liststore.append()
    self.plotpoints()

def removerow(self, widget):
    self.select = self.treeview.get_selection()
    self.model, self.treeiter = self.select.get_selected()
    if self.treeiter is not None:
        self.liststore.remove(self.treeiter)
    self.plotpoints()
```

The end of the code creates an instance of the *MainClass*, sets up the plot using *resetplot* and plots the initial points. We also have to connect the window to the *delete-event*, calls the window and starts the Gtk main loop.

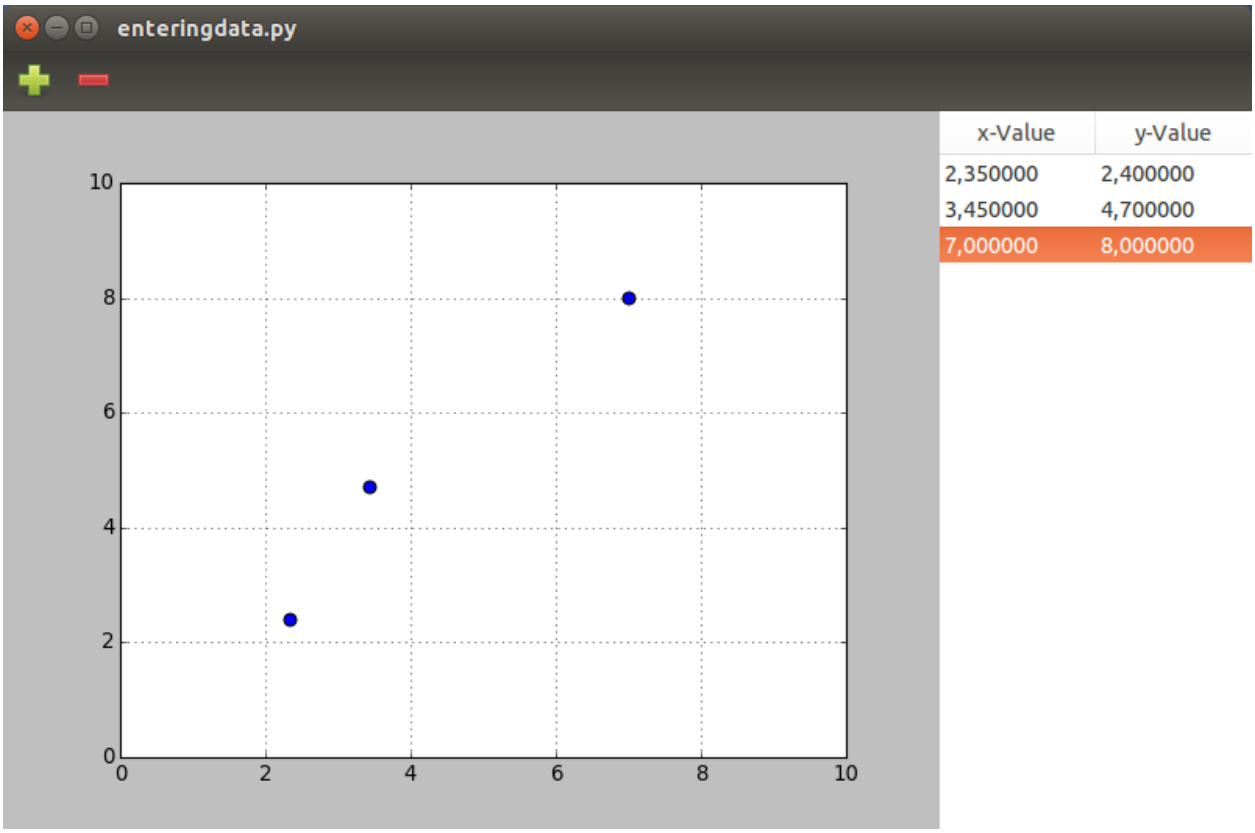
```
mc = MainClass()
mc.resetplot()
mc.plotpoints()

mc.window.connect("delete-event", Gtk.main_quit)
mc.window.show_all()
Gtk.main()
```

The finished program under Ubuntu looks like this:

4.4.2 Further Reading

- Python GTK+ 3 Tutorial: Tree and List Widgets
- AskUbuntu: What is the black header widget in some programs?


enteringdata.py
x-Value
y-Value
2,350000
2,400000
3,450000
4,700000
7,000000
8,000000
10
8
6
4
2
0
0
2
4
6
8
10

CHAPTER 5

Indices and tables

-* genindex -* modindex * search