The Python Papers Monograph, Vol. 1 (2009)
Available online at http://ojs.pythonpapers.org/index.php/tppm

Python Data Plotting and Visualisation Extravaganza

Guy K. Kloss
Computer Science
Institute of Information & Mathematical Sciences
Massey University at Albany,
Auckland, New Zealand
G.Kloss@massey.ac.nz

This paper tries to dive into certain aspects of graphical visualisation of data. Specifically it focuses on the plotting of (multi-dimensional) data using 2D and 3D tools, which can update plots at run-time of an application producing or acquiring new or updated data during its run time. Other visualisation tools for example for graph visualisation, post computation rendering and interactive visual data exploration are intentionally left out.

Keywords: Linear regression; vector field; affine transformation; NumPy.

1 Introduction

Many applications produce *data.* Data by itself is often not too helpful. To generate *knowledge* out of data, a user usually has to digest the information contained within the data. Many people have the tendency to extract patterns from information much more easily when the data is visualised. So data that can be visualised in some way can be much more accessible for the purpose of understanding.

This paper focuses on the aspect of *data plotting* for these purposes. Data stored in some more or less structured form can be analysed in multiple ways. One aspect of this is post-analysis, which can often be organised in an interactive exploration fashion. One may for example import the data into a spreadsheet or otherwise suitable software tool which allows to present the data in various ways. A user may "play" with different views, plots, scales, etc. to do so. Another way of accessing information from data visually is to plot it at the time of computation at a much earlier stage of the process. An engineer may for example be interested in the change of a parameter during the long numerical solution of a problem, so that the process can be interrupted early to avoid an unnecessary waste of time and computational resources in case the solver diverges rather than converges.

To get to the point ... the aspects discussed here are centering on displaying and updating data in plots at runtime of a computation. Whether it is a single scalar value plotted over time, or multi-dimensional vector fields are displayed.

Simple one-dimensional data is boring, as it may just be represented by a single bar or "gauge" like display (e. g. a "speedometer"). So what we are going to look at are two- (see Sect. 2) and three-dimensional plotting tools (see Sect. 3) used for forcing numerical content onto us through a GUI plot pane. In the minimum these are then scalar values that change over time/iterations, through two- and three-dimensional data sets, up to snapshots of higher-dimensional data sets.

Some of the tools that were originally intended to find their (stronger) representation in this paper had to be "kept short" and may only be mentioned. Unfortunately these had some more or less severe problems being installed on the author's system(s) at the time of writing (Ubuntu Jaunty and Karmic). Nonetheless, these are very much "worthy" of the inclusion and closer discussion in this list, but their code base, packaging and/or installer needs more time to mature towards current "painless" usage.

2 Two Dimensional Tools

The world of two dimensional plotting is (much more than the three dimensional tool set) generously populated by numerous tools. Many of them unmaintained, with no community or otherwise "esoteric". The tools discussed here do not claim to be the "ideal" tools, but they have proven to have a good persistence over time and feature a larger community of users. The first one is the "godfather" of plotting tools *Gnuplot,* which is still largely in use for the purpose of "academically credible" plotting purposes. In many aspects it is in the Python world nowadays superseded by the more modern *matplotlib.*

2.1 Gnuplot.py

Gnuplot [1,2] is most undoubtedly regarded by many as the original and most widely used plotting tool in the field. It has a usage history well exceeding a decade, and it is still maintained and enhanced. It is possible to use Gnuplot also for live plotting purposes, and Michael Haggerty has developed Python bindings for it [3]. These bindings have come a bit of age, but are still very well and easily usable. One just has to avoid minor pitfalls due to the introduction of the `with` key word in Python, but that can be very easily and elegantly circumvented by using dictionaries for plotting attributes. Fig. 1 shows a simple example on how to engage in an updatable data plotting session.

Another point to mention is, that it uses temporary files for storage of the plotting data, rather than passing the data straight into Gnuplot. However, as Gnuplot is very fast, this does not introduce any noticeable delays in plotting compared to the

```
import Gnuplot
import math
import time

SIZE = 100
INCREMENT = 10.0 * math.pi / SIZE

class MyPlotter(object):
    def __init__(self):
        self.values = []
        self.my_plot = Gnuplot.Gnuplot()
        self.current_x = 0.0

    def update(self):
        self.current_x += INCREMENT
        self.values.append(math.sin(self.current_x) / self.current_x)
        self.my_plot.plot(self.values)

    def run(self):
        for i in range(SIZE):
            self.update()
            time.sleep(0.1)

if __name__ == '__main__':
    plotter = MyPlotter()
    plotter.run()
    raw_input('Please press return to finish ...\n')
```

Figure 1: Gnuplot example, adding values on every call to `update()` to a list for plotting.

"matplotlib" (see next section).

2.2 matplotlib

matplotlib [4,5] is a more recent plotting package that has gained significant traction in the Python community. It provides a very powerful and feature rich environment for plotting. The resulting plot panels as well as the integrated GUI elements look much more modern, and the plotting panel can also be integrated into various other GUI applications, rather than "living" in a separate window (as compared to Gnuplot). For the GUI it can use various generic GUI toolkits (wxPython, Qt or GTK).

One of the features of matplotlib is, that it offers next to the object-oriented API an additional procedural interface (the `pylab` interface) that is closely modelled to resemble that of MATLAB. The combination of matplotlib with NumPy [6] therefore can be used as a free, modern and easy to learn replacement for the commercial MATLAB package(s).

In Fig. 2 the Gnuplot example is picked up and implemented using matplotlib.

```python
import math
import matplotlib
matplotlib.use('GTKAgg') # do this before importing pylab or pyplot
from matplotlib import pyplot
import gobject

SIZE = 100
INCREMENT = 10.0 * math.pi / SIZE

class MyPlotter(object):
    def __init__(self):
        self.current_x = 0.0
        self.x_values = []
        self.y_values = []
        self.my_figure = pyplot.figure()
        self.my_plot = self.my_figure.add_subplot(111)
        self.my_curve = self.my_plot.plot(self.x_values, self.y_values)

    def update(self):
        self.current_x += INCREMENT
        self.x_values.append(self.current_x)
        self.y_values.append(math.sin(self.current_x) / self.current_x)
        self.my_curve[0].set_xdata(self.x_values)
        self.my_curve[0].set_ydata(self.y_values)
        self.my_plot.relim()
        self.my_plot.autoscale_view()
        self.my_figure.canvas.draw()

    def run(self):
        for i in range(SIZE):
            self.update()

if __name__ == '__main__':
    plotter = MyPlotter()
    gobject.idle_add(plotter.run)
    pyplot.show()
```

Figure 2: Example for matplotlib, similar to the previous Gnuplot example.

Matplotlib in contrast to Gnuplot runs within the same process. As it comes along with its own GUI event loop, a little more boiler plate code is needed to keep the main Python script “alive” to feed further data.

2.3 Honorary Mentions

As mentioned in the introduction to this section, the user is facing a multitude of available plotting packages for two dimensions which would clearly exceed the scope of this paper. Some of these, however, deserve a mention without too much further description.

RPy is a a programming language and software environment for statistical computing and graphics [7,8]. *RPy* [9] provides a Python interface to the R programming language. Through RPy it is possible to use the features or R almost seamlessly, including graph plotting. In terms of plotting, R is comparable to Gnuplot, each having advantages and disadvantages to each other in different places.

Chaco is an open source Python plotting application toolkit for all types of 2D plotting [10]. It features a huge set of capabilities – both for live plotting, as well as for an interactive data visualization and exploration. Its capabilities go well beyond those of matplotlib. It builds on the very powerful Enthought Tool Suite and Traits by the company Enthought [11], who are also sponsoring and supporting the open source SciPy tools [12]. Unfortunately at the time of writing it failed to install and/or run properly both under Ubuntu Jaunty as well as Karmic. Although an excellent tool, it can be rather difficult to match all dependencies and get everything right, even for the distributor's packages.

GracePlot A Python interface [13] to another grandfather of plotting tools, "xmgrace." Both, GracePlot as well as xmgrace, did not experience much developer attention in the past. But this year GracePlot development has been picked up (again) with a release of a 2.0 version of the libraries. GracePlot owns its popularity for be very easy to use, but does not nearly offer the features of the tools mentioned above.

3 Three Dimensional Tools

Three- and higher-dimensional plotting challenges the plotting tool in a different way. These tools usually all *have to* create the impression of three-dimensionality on a 2D screen. Usually this is supported by the use of hardware accelerated OpenGL based libraries. Perspective and shading as well as transparency are tricks used to give the impression of spatiality.

This works still quite well for plotting 3D data sets like surface plots in a 2D domain. Unfortunately this is in many cases not enough. For example when a scalar field within a 3D space is to be visualised, or a 2D vector field in a 2D plane. Developers of plotting tools have put much effort into enabling the visualisation of these. These are some of the tricks used to achieve this:

- *Quiver Plots*
  little "vector arrows" at (regular) intervals within the plot

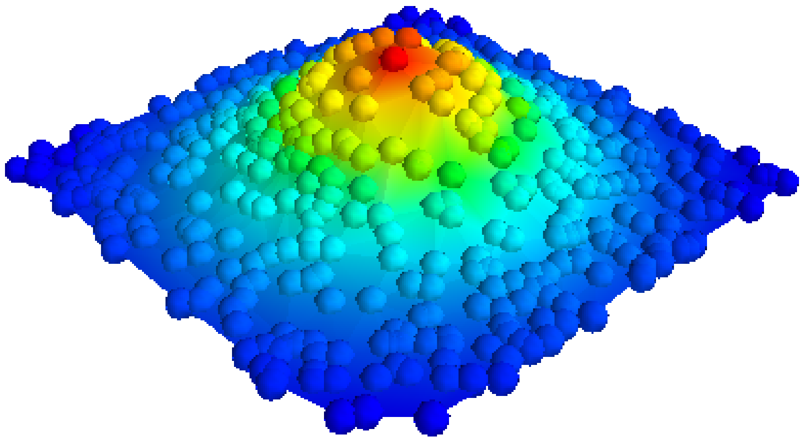

Figure 3: Surface plot from irregularly sampled data created with Mayavi from script in Fig. 4.

- *ISO Surfaces*
  regions of equal values in a 3D space are represented as (partly translucent) 2D membranes
- *Colour Shading*
  using colour encoding on a surface to indicate a value
- *Cutting Planes*
  using (movable) planes cross-secting a volume to render the "inside" of a volume on it
- *Stream Lines*
  showing virtual paths of a number of particles as if they were moved through space by the forces of a field

3.1 Mayavi

Mayavi [14] is today – just like Chaco – a highly sophisticated plotting tool built on top of the Enthought Tool Suite and Traits, and also supported by Enthought. For the 3D rendering purposes it uses TVTK, a Pythonic API to the Visualization Toolkit (VTK), which can be considered as being one of the de-facto tools for 3D data rendering.

Mayavi features all the techniques mentioned above and is particularly well suited for these tricks. For multi-dimensional plotting in scientific applications Mayavi is a tool definitely worth evaluating, and will probably score at least among the first places in any evaluation for Python based tools. A simple sample plot is shown in Fig. 3 with the corresponding script in Fig. 4.

```python
from enthought.mayavi import mlab
import numpy

# Create data with x and y random in the [-2, 2] segment, and z a
# Gaussian function of x and y.
numpy.random.seed(12345)
x = 4 * (numpy.random.random(500) - 0.5)
y = 4 * (numpy.random.random(500) - 0.5)

def f(x, y):
    return numpy.exp(-(x ** 2 + y ** 2))

z = f(x, y)

# Create a figure.
mlab.figure(1, fgcolor=(0, 0, 0), bgcolor=(1, 1, 1))

# Visualize the points.
points = mlab.points3d(x, y, z, z, scale_mode='none', scale_factor=0.2)

# Create and visualize the mesh.
mesh = mlab.pipeline.delaunay2d(points)
surface = mlab.pipeline.surface(mesh)

# Set viewing direction/distance and show.
mlab.view(47, 57, 8.2, (0.1, 0.15, 0.14))
mlab.show()
```

Figure 4: Example of a surface plot from irregularly sampled data using Mayavi.

3.2 Visual Python

Visual Python [15] – or short VPython – is actually *not* a plotting tool. VPython is a tool that aims at making it "easy to create navigable 3D displays and animations, even for those with limited programming experience." It is originally developed by David Scherer and now for several years maintained by Bruce Sherwood. It is mostly used for educational purposes (at universities) for teaching without the need to get deeper into programming.

And it was exactly this fact of ease of use that has helped the author many times to "knock up" easily visualisation tools to gain better insight into multi-dimensional data sets. Particularly for two reasons:

1. It is "easy as" to come up with a fast, hardware accelerated visualisation.
2. As it is not "plotting" in the general sense, one has got more influence on the exact representations.

Some of the data sets faced were dealing with colour spaces, often encoded with three channels (e. g. RGB). This way objects for colour transformations could be

```python
import visual
import numpy

SIZE = 4

class PointCloud(object):
    def __init__(self):
        self.iterator = None
        self.balls = numpy.zeros([SIZE] * 3, dtype=object)
        for x in range(SIZE):
            for y in range(SIZE):
                for z in range(SIZE):
                    coords = numpy.array([x, y, z], dtype=float)
                    new_sphere = visual.sphere(pos=coords,
                                               radius=0.25,
                                               color=tuple(coords / (SIZE - 1)))
                    self.balls[x, y, z] =  new_sphere

    def update_balls(self):
        for x in range(SIZE):
            for y in range(SIZE):
                for z in range(SIZE):
                    offset = numpy.random.normal(loc=0.0,
                                                 scale=0.01,
                                                 size=3)
                    pos = self.balls[x, y, z].pos
                    self.balls[x, y, z].pos = pos + offset

    def run(self):
        for i in range(10000):
            self.update_balls()

if __name__ == '__main__':
    foo = PointCloud()
    foo.run()
```

Figure 5: Example simulating a Brownian point cloud using VPython.

assigned *directly* the appropriate colour (3D) to points in 3D space, yielding a 6D space that was to explore and analyse (see some samples in Fig. 6). VPython renders fast … very fast! With the help of Python bindings to OpenCV [16] and NumPy it was possible to capture and analyse frames off a web cam every 1–2 seconds, and display three dimensional colour space distributions (histograms) live. The scene can be rotated, zoomed and panned with no visible performance impact during this live process.

During simulations for example it is possible to update the `pos` attribute of rendered objects, and without further ado VPython would reposition and update the scene with minimal time impact on the computation. This even works very well

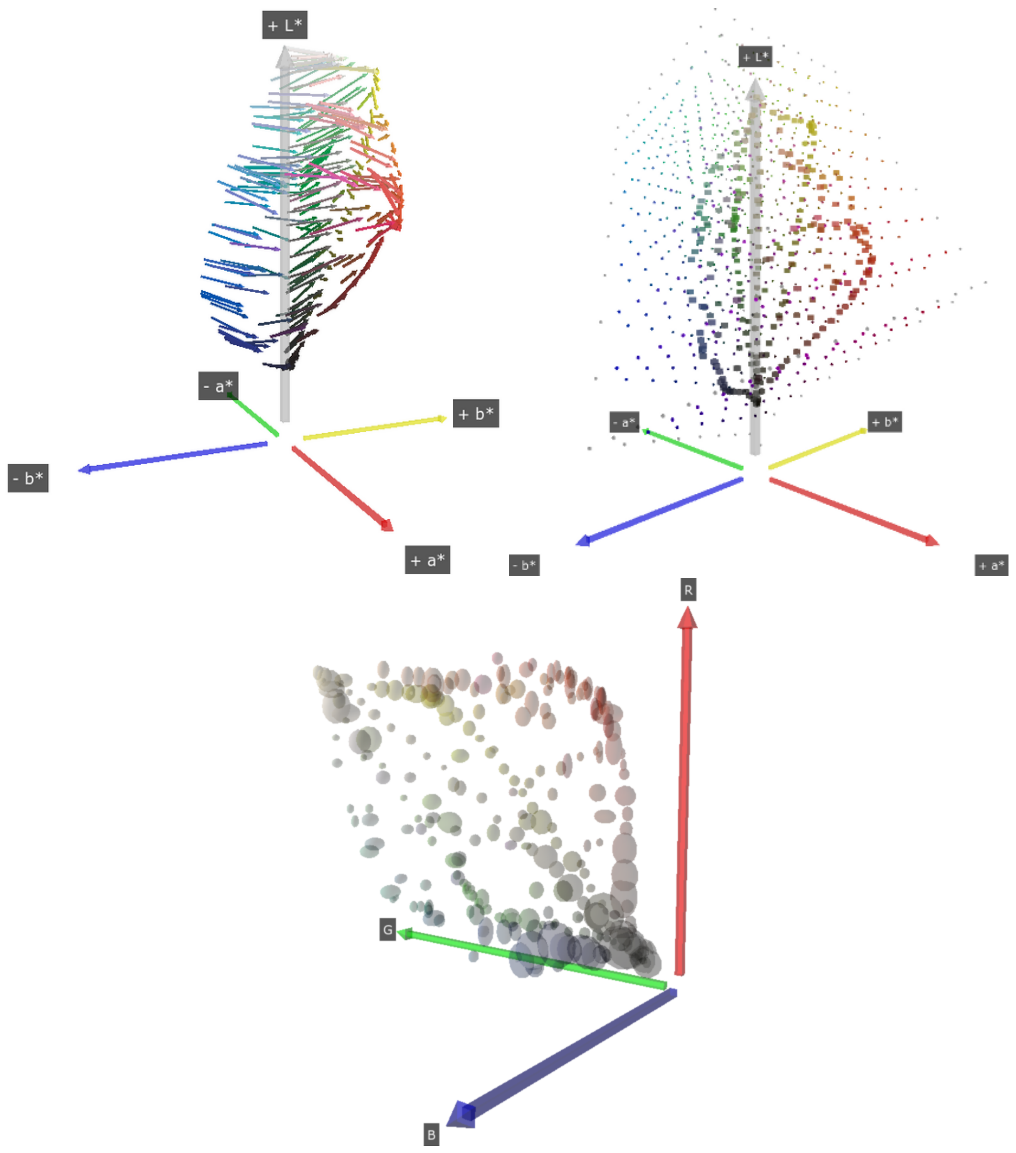

Figure 6: Some sample plots using VPython. (top left) Colour shifts through arrows (quiver plot). (top right) Cubic point clouds of interpolation volume and data points. (bottom) Colour measurement values in RGB space with measurement errors indicated through ellipsis sizes.

with scenes containing more than 5000 nodes rendered as spheres in a cubic mesh.

Another very nice feature is that a POVray export module is available for VPython. With this one 3D scenes can be rendered in much superior quality over the raw OpenGL window by ray tracing output for highest publication quality rendering. The POVray output is not 100 % feature complete to VPython, but it does handle the basic object geometries very reliably and well.

The biggest drawback currently is that VPython is *not* installable on recent versions of Python, as a change in Python 2.6.3 has introduced a compatibility problem with all stable versions of Boost.Python, which are needed to bind the native C++ code base to the Python API. Future versions of the Boost libraries (> 1.40) or Python ($> 2.6.4$) may fix this problem.

3.3 Mayavi "visual" Module

VTK through TVTK features similar possibilities to render 3D objects, as VPython does. A first attempt of a compatibility module `enthought.tvtk.tools.visual` has been made by Raashid Baig, a student of Prabhu Ramachandran, the original author of Mayavi. Some aspects of VPython are covered by this `visual` module due to the attempt to retain the API.

This `visual` module therefore can partly be used as a replacement for VPython. But of course one must be aware of the differences:

- Once rendered, it is just as fast as VPython in the direct mouse interaction with the scene, but rendering can be a lot slower, particularly when using the default *wxWindows* GUI back end. This can be partly improved by switching to the *Qt4* back end (by setting the environment variable `ETS_TOOLKIT` to the value of `qt4`).
- The compatibility API is not yet complete, and some things are handled differently.
- The implementation was developed with compatibility of VPython 3.x (current is version 5.x).
- Range checking through the "traits model" (a trait is a type definition that can be used for normal Python object attributes, giving the attributes some additional characteristics) based approach imposes some not so sensibles limitations to object sizing.
- One can use the very convenient "Traits" based GUI dialogues to alter object properties.
- The implementation of the module in `visual.py` is quite easy to understand and extend.
- The Mayavi `visual` module does currently *not* perform all kinds of wild card imports internally (Which is a good thing! The visual name space is "polluted" with wild card imports of `numpy` and `math`.).

```
from enthought.tvtk.tools import visual

# Rest of class definition clipped ...

    def run(self):
        self.iterator = visual.iterate(30, self.update_balls)
        visual.show()
```

Figure 7: Adaptations to imports and the `run()` method from listing in Fig. 5 for the Mayavi `visual` module.

3.4 Honorary Mentions

Grand dad Gnuplot is also capable of plotting in 3D to some extent. But that is mainly reduced to some rather simple surface plots or curves and scattered points in 3D. It is therefore not further discussed here. The access to Gnuplot's 3D features works the same way just using the Gnuplot.py bindings as used for 2D, just the usual 3D features (e. g. "splot") are used.

4 Conclusion

This paper shows by no means an exhaustive picture of plotting tools. But it shows that data plotting and visualisation, particularly for live plotting, is possible for two and more dimensions. The tools presented give some good examples for people who want to engage in scientific data plotting and visualisation, and the features discussed just barely scratch the surface of what is possible. Everybody has to evaluate and study each tool for personal needs to come up with a good solution for the current problem at hand. However, all of the discussed tools are worth keeping in one's personal "tool chest" for the point of time when they become useful.

The newer and more fully featured plotting tools (matplotlib, Mayavi, Chaco) commonly run within the same process, and they can be embedded within other (GUI) applications. Unfortunately these features come at a cost by introducing further steps in order to make them "play nice" with the general core application. This is also mainly one of the reasons why the classics as Gnuplot still have a very strong stand in the scientific community where the goal is functionality over "pretty applications."

All the discussed tools in this paper (as well as Chaco) provide the feature to export rendered frames to image files. These can be used for documentation purposes, or to produce more advanced output compilations in the form of movie sequences compiled from them.

References

[1] J. K. Philipp, *Gnuplot in Action: Understanding Data with Graphs.* Manning Publications, 2009.

[2] "Gnuplot Homepage," http://www.gnuplot.info/.

[3] M. Haggerty, "Gnuplot.py Web Site," http://gnuplot-py.sourceforge.net/.

[4] J. Hunter, "matplotlib Project Web Site," http://matplotlib.sourceforge.net/.

[5] "SciPy's matplotlib Cookbook," http://www.scipy.org/Cookbook/Matplotlib/.

[6] T. E. Oliphant, *Guide to NumPy*, T. E. Oliphant, Ed. Trelgol Publishing, 2006.

[7] R. Ihaka and R. Gentleman, "R: A language for data analysis and graphics," *Journal of Computational and Graphical Statistics*, vol. 5, pp. 299–314, 1996.

[8] "R Main Project Web Page," http://www.r-project.org/.

[9] "RPy Project Web Site," http://rpy.sourceforge.net/.

[10] "Chaco Project," http://code.enthought.com/projects/chaco/.

[11] T. Vaught and E. Jones, "Enthought Scientific Computing Solutions, Inc." http://www.enthought.com/.

[12] "SciPy – Scientific Python Project," http://www.scipy.org/.

[13] "GracePlot Web Site," http://graceplot.sourceforge.net/.

[14] P. Ramachandran and G. Varoquaux, "Mayavi Project," http://code.enthought.com/projects/mayavi/.

[15] B. Sherwood and D. Scherer, "Visual Python Project," http://vpython.org/.

[16] "OpenCV Project," http://opencv.willowgarage.com/.