

# PyPlotter

## A Python/Jython Graph Plotting Package

### Manual

Eckhart Arnold

September, 6th 2015

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>License</b>	<b>2</b>
<b>3</b>	<b>Screenshots</b>	<b>3</b>
<b>4</b>	<b>Quick Tutorial</b>	<b>3</b>
4.1	Example 1: Plotting a Graph . . . . .	4
4.2	Example 2: Plotting a simplex diagram . . . . .	4
<b>5</b>	<b>Reference</b>	<b>5</b>
5.1	Overview . . . . .	5
5.2	Class <code>Graph.Cartesian</code> . . . . .	6
5.3	Class <code>Simplex.Diagram</code> . . . . .	9
<b>6</b>	<b>Implementing a new device driver for PyPlotter</b>	<b>11</b>

# 1 Introduction

PyPlotter is a 2D graph plotting package for Python and Jython (the java version of Python). It contains classes for drawing graphs on a cartesian coordinate plain (with linear or logarithmic scale) and for plotting 2D simplex diagrams. PyPlotter supports different GUI libraries and can easily adapted to other GUIs or output devices by implementing a very simple driver interface. Currently (Version 0.9.2), tk, gtk, qt, wxWidgets, java awt and postscript are supported as output devices.

Since Version 0.9.2 PyPlotter is Python 3 compatible. However, the drivers gtkGfx.py and wxGfx.py are not (yet) Python 3 compatible.

# 2 License

The MIT License (MIT)

Copyright (c) 2004 Eckhart Arnold (eckhart\_arnold@yahoo.de, www.eckhartarnold.de)

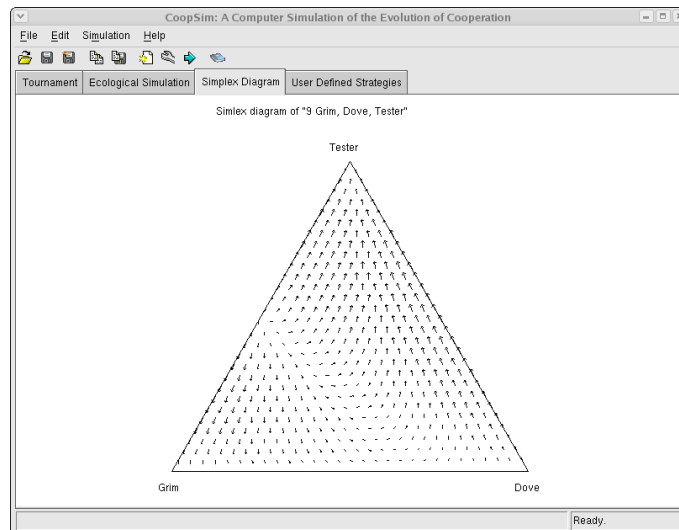
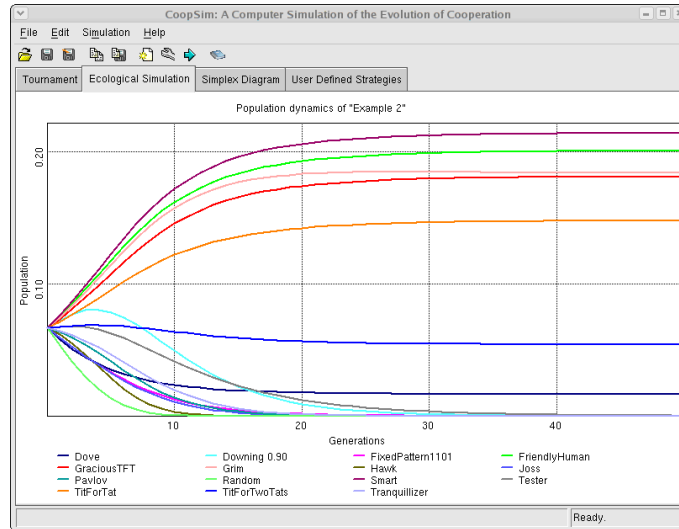
Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

### 3 Screenshots

Here are two sample screenshots form a program using PyPlotter:



### 4 Quick Tutorial

While the classes `Graph.Cartesian` and `Simplex.Diagram` are quite versatile, it was a major aim of their development to make the usage for beginners as simple as possible. To show you how to use these classes, this tutorial contains two commented example programs.

## 4.1 Example 1: Plotting a Graph

In order to see the results of this example, either run the file “Example1.py” from the PyPlotter directory or enter the following lines at the python command prompt.

```
1: import math
2: from PyPlotter import tkGfx as GfxDriver # 'awtGfx' for jython
3: from PyPlotter import Graph, Gfx
4:
5: gfx = GfxDriver.Window(title="Function Plotter")
6: gr = Graph.Cartesian(gfx, -4.0, -2.0, 4.0, 2.0)
7: gr.addPen("sin(x)", Gfx.RED_PEN)
8: for x in gr.xaxisSteps(-4.0, 4.0):
9:     gr.addValue("sin(x)", x, math.sin(x))
10: gfx.waitUntilClosed()
```

Thats all! If everything went right you should have seen a nice sine curve on your display. Here is an explanation of what the program does. Line 5 opens a window for graphical output. Then a new cartesian graph is being created in this window. In line 7 a new pen is added to the graph. Before you can draw anything onto the graph, you have to add one or more pens. Every pen is identified by its unique name. By default the graph as a caption where all pens are listed by their names. To actually draw something on the graph, you have to add one or more coordinate pairs to the graph with a given pen. The coordinate pairs of a pen will then be connected with a continuous line in the order they where added to the graph. This is done in line 8 and 9. In line 8 the method `xaxisSteps` is called, which returns a list of x values for a given range, each of which corresponds to exactly one pixel on the screen.

Since Version 0.8.7 of PyPlotter the same can be done even simpler:

```
1: import math
2: from PyPlotter import Graph
3: gr = Graph.Cartesian(Graph.AUTO_GFX, -4.0, -2.0, 4.0, 2.0)
4: for x in gr.xaxisSteps(-4.0, 4.0):
5:     gr.addValue("sin(x)", x, math.sin(x))
6: gr.gfx.waitUntilClosed()
```

## 4.2 Example 2: Plotting a simplex diagram

Here is a short code snippet to demonstrate the use of a simplex diagram. For the sake of brevity, the actual population dynamical function is not contained. See file “Example2.py” for the full program.

```

1: from PyPlotter import tkGfx as GfxDriver
2: from PyPlotter import Simplex
3:
4: gfx = GfxDriver.Window(title="Demand Game")
5: dynamicsFunction = lambda p: PopulationDynamics(p,DemandGame,
6:                                                  e=0.0,noise=0.0)
7: diagram = Simplex.Diagram(gfx, dynamicsFunction, "Demand Game",
8:                          "Demand 1/3", "Demand 2/3", "Demand 1/2")
9: diagram.show()
10: gfx.waitUntilClosed()

```

In order to draw a simplex diagram, you need to instantiate class `Simplex.Diagram` (line 7) with a suitable population dynamical function. Class `Simplex.Diagram` is specifically designed for visualizing population dynamics. If you want to use simplex diagrams for another purpose, you should use the lower level class `Simplex.Plotter` instead. The simplex diagram will not be drawn, unless the `show` method of class `Simplex.Diagram` is called, as it is done in line 9 of this example.

## 5 Reference

This reference of the `PyPlotter` package does only cover the most high level classes and functions of `PyPlotter`. For a description of the lower level classes and functions, see the doc strings in the source code.

### 5.1 Overview

`PyPlotter` basically consists of two parts, a front end part and a back end part. The front end part comprises the high level classes to plot cartesian graphs or simplex diagrams. These are the class `Cartesian` from the `Graph` module and class `Diagram` from the `Simplex` module. The backend part is a simple driver interface that is defined in the `Gfx` module. There exist several implementations of this driver interface for different graphical user interfaces. They are located in the modules named `**Gfx`.

Package `PyPlotter` consists of the following Modules:

**Compatibility** A helper module to ensure compatibility with different Python versions (Versions 2.1 through to Version 2.4) as well as compatibility with Jython 2.1 .

**Colors** A helper module for dealing with colors. It contains a list of well distinguishable colors (useful if drawing many graphs on one single plain) and a few filter functions that help assigning similar color shades to graphs that belong to the same of several groups.

**Gfx** This module defines the driver interface (`class Driver`). It also contains `class Pen` to store a set of graphical attributes such as color, line width etc.

**\*\*Gfx** These modules contain implementations of `Gfx.Driver` for different GUIs. There are drivers for the following GUI toolkits:

- `tkGfx` for the *tkinter* GUI toolkit that comes with the Python standard distribution.
- `qtGfx` for the *qt* GUI toolkit ([www.riverbankcomputing.co.uk/software/pyqt/](http://www.riverbankcomputing.co.uk/software/pyqt/)). `qtGfx` tries to import qt version 4, but falls back on version 3, if version 4 of qt is not present.
- `wxGfx` for the *wxWidgets* GUI toolkit ([www.wxwidgets.org](http://www.wxwidgets.org)).
- `gtkGfx` for the *gtk* GUI toolkit ([www.pygtk.org](http://www.pygtk.org)).
- `awtGfx` for the Java *awt/swing* GUI toolkit under Jython, the Python version running under the Java JVM.
- `tt psGfx` for *postscript* output that can be written to a file.

**Graph** Contains the high level class `Cartesian` for drawing graphs on a cartesian plain. It also contains a number of intermediate level classes for mapping virtual to screen coordinates etc.

**Simplex** Contains the high level class `Diagram` for drawing simplex diagrams of population dynamics. Within in this module also some intermediate classes for simplex drawing and coordinate transformation are implemented.

## 5.2 Class `Graph.Cartesian`

Class `Graph.Cartesian` is versatile high level class for drawing graphs on a cartesian plain. It supports linear and logarithmic scales and automatic adjustment of the coordinate range as well as automatic captioning.

```
__init__ (self, gfx, x1, y1, x2, y2,  
         title = "Graph", xaxis="X", yaxis="Y",  
         styleFlags = DEFAULT_STYLE,
```

```
axisPen = Gfx.BLACK_PEN, labelPen = Gfx.BLACK_PEN,  
titlePen = Gfx.BLACK_PEN, captionPen = Gfx.BLACK_PEN,  
backgroundPen = Gfx.WHITE_PEN,  
region = REGION_FULLSCREEN)
```

Initializes the class with the following parameters:

**gfx** Gfx.Driver: The Gfx drivers used for drawing the graph. Use AUTO\_PEN if you want the Graph.Cartesian object to find a suitable driver (depending on the installed widget toolkits) on its own.

**x1,y1,x2,y2** floats: Coordinate range.

**title** string: Title string.

**xaxis, yaxis** strings: Axis descriptions.

**styleFlags** integer: Interpreted as a bitfield of flags that define the style of the graph. The following flags can be set:

**AXISES, AXIS\_DIVISION, FULL\_GRID** Draw axes, axis divisions and (or) a full grid.

**LABELS, CAPTION, TITLE** Draw axis labels, a caption with descriptions (generated from the pen names) below the graph, a title above the graph.

**SHUFFLE\_DRAW, EVADE\_DRAW** Two different algorithms to allow for the visibility of overlapping graphs.

**LOG\_X, LOG\_Y** Use a logarithmic scale for the x or y axis respectively.

**KEEP\_ASPECT** Keep the aspect ratio of the coordinates.

**AUTO\_ADJUST** Automatically adjust the range of the graph when a point is added that falls outside the current range.

**axisPen, labelPen, titlePen, captionPen, backgroundPen**  
Gfx.Pen: Pens (sets of graphical attributes) for the respective elements of the graph.

**region** 4-tuple of floats. The part of the screen that is used for the graph. Example: (0.05, 0.05, 0.95, 0.95) would leave a border of 5 % of the screen size on each side.

**adjustRange** (self, x1, y1, x2, y2) - Adjusts the range of the coordinate plane.

**setStyle** (self, styleFlags=None, axisPen=None, labelPen=None, titlePen=None, captionPen=None, backgroundPen = None) - Changes the style of the graph. Only parameters that are not **None** will be changed.

**setTitle** (self, title) - Changes the title of the graph.

**setLabels** (self, xaxis=None, yaxis=None) - Changes the labels of the graph.

**resizedGfx** (self) - Takes notice of a resized window.

**changeGfx** (self, gfx) - Switch to another device context. This can be useful if you want to draw the current graph into a buffered image that you want to save on a disk. In this case you have to create the buffered image, create the Gfx driver for your buffered image, call changeGfx and then redraw. After that you can call changeGfx to switch back to the former output device.

**redrawGraph** (self) - Redraws the graph, but not the caption, title or labels.

**redrawCaption** (self) - Redraw only the caption of the graph.

**redraw** (self) - Redraws the whole graph including, title, labels and the caption.

**reset** (self, x1, y1, x2, y2) - Restarts with a new empty graph of the given range. All pens are removed.

**addPen** (self, name, pen=AUTO\_GENERATE\_PEN, updateCaption=True) - Adds a new pen with name “name” and attributes “pen” to the graph.

**removePen** (self, name, redraw=True) - Removes a pen from the graph. All coordinate pairs associated with this pen will be discarded.

**addValue** (self, name, x, y) - Add the point (x,y) to the graph drawn with pen “name”.

**peek** (self, x, y) - Returns the graph coordinates of the screen coordinates (x,y)

**xaxisSteps** (self, x1, x2) - Returns a list of virtual x-coordinates in the range [x1,x2] with one point for each screen pixel. This is especially useful when working with large range logarithmic scales.



**yaxisSteps** (self, y1, y2) - Returns a list of virtual x-coordinates in the range [y1,y2] with one point for each screen pixel. This is especially useful when working with large range logarithmic scales.

### 5.3 Class Simplex.Diagram

Class `Simplex.Diagram` is a class for drawing simplex diagrams of population dynamics of populations of three species. For simplex diagrams dedicated to other purposes it is recommended to use the lower level class `Simplex.Plotter` instead.

```
__init__ (self, gfx, function, title="Simplex Diagram",  
          p1="A", p2="B", p3="C", styleFlags = VECTORS,  
          raster = RASTER_DEFAULT, density = -1,  
          color1 = (0.,1.,0.), color2 = (1.,0.,0.),  
          color3 = (0.,0.,1.), colorFunc = scaleColor,  
          titlePen = Gfx.BLACK_PEN, labelPen = Gfx.BLACK_PEN,  
          simplexPen=Gfx.BLACK_PEN, backgroundPen=Gfx.WHITE_PEN,  
          section=Graph.REGION_FULLSCREEN)
```

Initializes the class with the following parameters:

**gfx** `Gfx.Driver`: The Gfx drivers used for drawing the simplex diagram.

**function**  $f(p) \rightarrow p^*$ , where  $p$  and  $p^*$  are 3 tuples of floats that add up to 1.0: Population dynamics function to be displayed in the simplex diagram.

**title, p1, p2, p3** strings: Strings to mark the title and the three corners of the diagram with.

**styleFlags** integer, interpreted as a bitfield of flags: The style or rather flavour of the simplex diagram. Presently three flavours are possible: `VECTORS` for drawing the diagram as a vector field with many little arrows; `TRAJECTORIES` for drawing pseudo trajectories; `PATCHES` for drawing a patched diagram, where each point in the diagram has a unique color in the beginning. From generation to generation, however, colors are adjusted such that every point ("patch") takes the color of the point it has moved to. This exposes areas of attraction in the diagram.

**raster** list of points (3-tuples of floats that add up to 1.0): The point raster of the simplex diagram. Suitable point rasters of varying density can be produced with the functions `Simplex.GenRaster` and `Simplex.RandomGrid`.

**density** integer > 2: The density of the points of the simplex diagram.

This is mainly useful in combination with style `PATCHES`, because this style does not use a raster.

**color1, color2, color3** (r,g,b)-tuples, where r,g and b are floats in range of [0.0, 1.0]: The three color parameters have a different meaning depending on the diagram style used. For patch diagrams these are the edge colors of the three edges of the diagram. For trajectory diagrams color1 is the starting color and color2 is the color towards which later steps of the trajectory are shaded. For vector fields the range between color1 and color2 is used to indicate the strength of the vector field.

**colorFunc** f(ca, cb, strength) -> c, where ca and cb are colors and strength is a float from [0, infinity]: This function produces a color shade from 'ca', 'cb' and 'strength', usually somewhere on the line between 'ca' and 'cb'. The parameter `colorFunc` is not used for patches diagrams.

**titlePen, labelPen, simplexPen, backgroundPen** Gfx.Pen: Pens for the respective parts of the simplex diagram.

**section** 4-tuple of floats from then range [0.0, 1.0]: the part of the screen to be used for the diagram.

**setStyle** (self, styleFlags=None, titlePen=None, labelPen=None, simplexPen=None, backgroundPen=None) - Changes the style of the simplex diagram. It is not necessary to assign a value to all arguments of the functions. Those arguments that no value is assigned to will leave the respective class attributes untouched.

**setFunction** (self, function) - Changes the population dynamics function that is visualized by the diagram. The change will only be visible after the method `show` has been called.

**setRaster** (self, raster) - Changes the raster of sample points. The change will only be visible after the method `show` has been called.

**setDensity** (self, density) - Generates a raster of uniformly distributed sample points (population distributions) with the given density. The change will only be visible after the method `show` has been called.

**changeColors** (self, color1 = (0.,1.,0.), color2 = (1.,0.,0.), color3 = (0.,0.,1.), colorFunc=scaleColor) - Changes the colors of diagram, including a color modifying function. Note: The semantics

of these parameters may differ depending on the visualizer used. The change will only be visible after the method `show` has been called.

**show** (self, steps=-1) - Shows the diagram calculating 'steps' generations for dynamic diagrams (style `TRAJECTORIES` or `PATCHES`).

**showFixedPoints** (self, color) - Shows candidates(!) for fixed points (only if style is `PATCHES`).

**redraw** (self) - Redraws the diagram.

**resizedGfx** (self) - Takes notice of a resized graphics context and redraws the diagram.

## 6 Implementing a new device driver for Py-Plotter

Adapting `PyPlotter` to a new GUI environment or to a new output device is very easy. You only have to implement a class for the driver itself that is derived from `Gfx.Driver` and, optionally, also another very simple standardized window class to open an output window (or context) on your GUI or device. The latter class must be derived from class `Gfx.Window`.

The driver class must implement the following methods from its parent class `Gfx.Drivers`: `__init__`, `resizedGfx`, `getSize`, `getResolution`, `setColor`, `setLineWidth`, `setLinePattern`, `setFillPattern`, `setFont`, `getTextSize`, `drawLine`, `fillPoly`, `writeStr`. Overriding the other methods or adding further methods is optional and may lead to increased performance.

The window class must implement all methods of class `Gfx.Window`, that is: `__init__`, `refresh`, `quit`, `waitUntilClosed`.

The already implemented drivers in modules `awtGfx`, `wxGfx`, `tkGfx`, `gtkGfx`, `qtGfx` and `psGfx` may serve as examples for implementing new drivers.