The Python Graphics Interface, Part II

Object-Oriented Graphics Manual

Written by

Zane C. Motteler
Lee Busby
Fred N. Fritsch

This work was produced at the University of California, Lawrence Livermore National Laboratory under contract no. W-7405-ENG-48 between the U.S. Department of Energy and The Regents of the University of California for the operation of UC LLNL.

DISCLAIMER

Table of Contents

CHAPTER 1: The Python Graphics Interface

1.1 Overview of the Python Graphics Interface

The Python Graphics Interface (abbreviated PyGraph) provides Python users with capabilities for plotting curves, meshes, surfaces, cell arrays, vector fields, and isosurface and plane cross sections of three dimensional meshes, with many options regarding line widths and styles, markings and labels, shading, contours, filled contours, coloring, etc. Animation, moving light sources, real-time rotation, etc., are also available. PyGraph is intended to supply a choice of easy-to-use interfaces to graphics which are relatively independent of the underlying graphics engine, concealing the technical details from all but the most intrepid users. Obviously different graphics engines offer different features, but the intention is that when a user requests a particular type of plot which is not available on a particular engine, the low level interface will make an intelligent guess and give some approximation of what was asked for.

There are two such graphics packages which are relatively independent of the underlying plotting library. The Object-Oriented Graphics (OOG) Package defines geometric objects (Curves, Surfaces, Meshes, etc.), Graph objects which can be given one or more geometric objects to plot, and Plotter objects, which receive geometric objects to plot from Graph objects, and which interface with the graphics engine(s) to do the actual plotting. A Graph can create its own Plotter, or the more capable user can create one or more, handy when one wishes (for instance) to plot on a remote machine, or to open graphics windows of different types at the same time. The second such package is called EZPLOT; it is built on top of OOG, and provides an interface similar to the command-line interface of the Basis EZN package. Some of our long-time users may be more comfortable with this package, until they have mastered the concepts of object-oriented design.

As mentioned above, a Graph object needs at least one Plotter object to plot itself; only the Plotter objects need know about graphics engines. At present we have two types of Plotter objects, one which knows about Gist and one which knows about Narcisse. Some power users may prefer to use the lower-level library-specific function calls, but most users will use EZPLOT or OOG.

Gist is a scientific graphics library written in C by David H. Munro of Lawrence Livermore National Laboratory. It features support for three common graphics output devices: Xwindows, (color) PostScript, and ANSI/ISO Standard Computer Graphics Metafiles (CGM). The library is small (written directly to Xlib), portable, efficient, and full-featured. It produces x-vs.-y plots with ‘‘good’’ tick marks and tick levels, 2-D quadrilateral mesh plots with contours, vector fields, or pseudocolor maps on such meshes. 3-D plot capabilities include wire mesh plots (transparent or opaque), shaded and colored surface plots, isosurface and plane cross sections of meshes containing data, and real-time animation (moving light sources and rotations). The Python Gist module `gist.py` and the associated Python extension `gistCmodule` provide a Python interface to this library (referred to as PyGist).

Narcisse is a graphics library developed at out sister laboratory at Limeil in France. It is especially strong in high-quality 3-D surface rendering. Surfaces can be colored in a variety of ways, including colored wire mesh, colored contours, filled contours, and colored surface cells. Some combinations of these are also possible. We have also added the capability of doing isosurfaces and plane sections of meshes, which is not available in the original Narcisse. The Python Narcisse module `narcissemodule` (referred to as PyNarcisse) provides a low-level Python interface to this library. Unlike Gist, Narcisse does not currently write automatically to standard files such as PostScript or CGM, although it writes profusely to its own type of files unless inhibited from doing so, as described below. However, there is a "Print" button in the Narcisse graphics window, which opens a dialog that allows you to write the current plot to a postscript file or to send it to a postscript printer.

1.2 Using the Python Graphics Interface

In order to use PyGraph, you first need to have Python installed on your system. If you do not have Python, you can obtain it free from the Python pages at `http://www.python.org`. You may need the help of your system administrator to install it on your machine. Once you have Python, you have to know at least a smattering of the language. The best way to do this is to download the excellent tutorial from the Python pages, sit down at your computer or terminal, and work your way through it.

Before using the Python Graphics Interface, you should set some environment variables as follows.

- Your `PATH` variable should contain the path to the `python` executable.
- You should set a `PYTHONPATH` variable to point to all directories that contain Python extensions or modules that you will be loading, which may include the OOG modules, `ezplot`, and `narcissemodule` or `gistCmodule`. Check with your System Manager for the exact specifications on your local systems.
- Unless you create your own plotter objects, PyGraph will create a default Gist Plotter which will plot to a Gist window only. If you want your default Plotter to be a Narcisse Plotter, then set the variable `PYGRAPH` to `Nar` or `Narcisse`.

A Gist Plotter object automatically creates its own Gist window and then plots to that window. Narcisse, however, works differently. Narcisse is established as a separately running process, to which the Plotter communicates via sockets. Thus, to run a Narcisse Plotter, you must first open a Narcisse.[1] To

1. I am going to assume that you already have Narcisse installed on your system, and its directory path in your `PATH` variable.

do so, you need to go through the following steps:

1. Set your environment variable `PORT_SERVEUR`[1] to 0.
2. Start up Narcisse by typing in the command `Narcisse &`. It will take a few moments for the Narcisse GUI to open, then immediately afterwards it will be covered by an annoying window which you can eliminate by clicking its `OK` button.
3. You will note that there is a server port number given on the GUI. Set your `PORT_SERVEUR` variable to this value.
4. Narcisse has an annoying habit of saving everything it does to a multitude of files, and notifying you on the fly of all its computations. If you do a lot of graphics, these files can quickly fill up your quota. In addition, the running commentary on file writing and computation on the GUI is time-consuming and slows Narcisse down to a truly glacial pace. To avoid this, you need to turn off a number of options via the GUI before you begin. They are all under the `STATE` submenu of the `FILE` menu, and should be set as follows: set ‘‘`Socket compute`’’ to “`no`,’’ set “`File save`’’ to “`nothing`,’’ set ‘‘`Config save`’’ to “`no`,’’ and set “`Ihm compute`’’ to “`no`.’’ (“IHM’’ are the French initials for ‘‘GUI.’’)

1.3 About This Manual

This manual is part of a series of manuals documenting the Python Graphics Interface (PyGraph). They are:

- **I.** EZPLOT User Manual
- **II.** Object-Oriented Graphics Manual
- **III.** Plotter Objects Manual
- **IV.** Python Gist Graphics Manual
- **V.** Python Narcisse Graphics Manual

EZPLOT is a command-line oriented interface that is very similar to the EZN graphics package in Basis. The Object-Oriented Graphics Manual provides a higher-level interface to PyGraph. The remaining manuals give low-level plotting details that should be of interest only to computer scientists developing new user-level plot commands, or to power users desiring more precise control over their graphics or wanting to do exotic things such as opening a graphics window on a remote machine.

PyGraph is available on Sun (both SunOS and Solaris), Hewlett-Packard, DEC, SGI workstations, and some other platforms. Currently at LLNL, Narcisse is installed only on the X Division HP and Solaris boxes, however, and Narcisse is not available for distribution outside this laboratory. Our French col-

1. We did tell you that Narcisse was French, didn’t we?

leagues are going through the necessary procedures for public release, but these have not yet been crowned with success. Gist, however, is publicly available as part of the Yorick release, and may be obtained by anonymous ftp from `ftp-icf.llnl.gov`; look in the subdirectory `/ftp/pub/Yorick`.

A great many people have helped create PyGraph and its documentation. These include

- Lee Busby of LLNL, who wrote `gistCmodule`, and wrought the necessary changes in the Python kernel to allow it to work correctly;
- Zane Motteler of LLNL, who wrote `narcissemodule`, `ezplot`, the OOG, and some other auxiliary routines, and who wrote much of the documentation, at least the part that was not blatantly stolen from David Munro and Steve Langer (see below);
- Paul Dubois of LLNL, who wrote the `PDB` and `Ranf` modules, and who worked with Konrad Hinsen (Laboratoire de Dynamique Moleculaire, Institut de Biologie Structurale, Grenoble, France) and James Hugunin (Massachusetts Institute of Technology) on `NumPy`, the numeric extension to Python, without which this work could not have been done;
- Fred Fritsch of LLNL, who produced the templates and did some of the writing of this documentation;
- Our French collaborators at the Centre D'Etudes de Limeil-Valenton (CEL-V), Commissariat A L'Energie Atomique, Villeneuve-St-Georges, France, among whom are Didier Courtaud, Jean-Philippe Nomine, Pierre Brochard, Jean-Bernard Weill, and others;
- David Munro of LLNL, the man behind Yorick and Gist, and Steve Langer of LLNL, who collaborated with him on the 3-D interpreted graphics in Yorick. We have also shamelessly stolen from their Gist documentation; however, any inaccuracies which crept in during the transmission remain the authors' responsibility.

The authors of this manual stand as representative of their efforts and those of a much larger number of minor contributors.

Send any comments about these documents to ‘‘`support@icf.llnl.gov`’’ on the Internet or to ‘‘`support`’’ on Lasnet.

CHAPTER 2: Introduction to Object-Oriented Graphics

Graphics objects consist of instances of one or more of the geometric objects (`Curve`, `Surface`, `Mesh3d`, etc.), and of objects to which they can be given to create a potential plot (`Graph2d` for `Curve`s, `Graph3d` for `Surface`s and/or `Mesh3d`s). A `Graph` object containing at least one geometric object needs to hand itself over to a third kind of object, a `Plotter` object, in order for the actual plot to appear somewhere (in an Xwindow or in a file, for example).

2.1 Object Oriented Graphics

The idea behind object oriented graphics (OOG) is to supply the user with classes of geometric objects and graph objects which are completely independent of the underlying graphics engine, making it unnecessary for the user to have to learn details of low level interfaces to graphics. Most users do not wish to be bothered with the low-level and often arcane methods of dealing with a graphics engine, let alone having to know the properties of more than one graphics engine, since typically they differ so radically from one another. We believe that the typical user would like to do something like the following: take the results of some calculations and use them to specify geometric objects; hand the geometric objects to graph objects; ask the graph objects to plot themselves.

Unfortunately the goal of a set of high-level graphics objects which are independent of the underlying graphics engines is difficult (nearly impossible) to reach. This is particularly true of the two graphics engines, Gist and Narcisse, which currently underlie the OOG. Gist has far more and better capabilitiues for 2-D graphics than does Narcisse. This means that to supply relatively equivalent 2-D graphics with Narcisse, it would be necessary to write a Python or C wrapper for Narcisse which does the necessary computations. Likewise, although there is considerable overlap, each engine supplies some 3-D capabilities that the other does not, so wrappers supplying extensions to each must be written. At the time of the writing of this manual, only a small part of this work has been done, but we hope to proceed with this work in the future.

Another intrinsic difficulty is that Narcisse is much slower than Gist, so, in particular, real-time animations involving complex figures are simply not feasible in Narcisse. Part of this slowness is due to the fact that the user program and Narcisse (a separate process) communicate data back and forth via sockets, and part is simply that Narcisse internal computations, for whatever reasons, are very slow.

A third problem is that plotting solely to a file is impossible in Narcisse; it is designed to be used interactively. Narcisse plots can be sent to either a binary or ascii file in addition to being sent to a window, but these files are in a format peculiar to Narcisse. A particular Narcisse plot can be sent to a Post-

Script file only by clicking a button in the Narcisse GUI currently displaying that plot. On the other hand, Gist plots can be sent to an arbitrary choice of windows, PostScript files, and CGM files without interctive intervention.

The tables below indicate to the user which capabilities are available in PyGist and PyNarcisse currently, and what types of devices can be plotted to. We use the term ‘‘not yet’’ for features which will someday be implemented, and ‘‘never’’ for those which are essentially impossible.

TABLE 1 Geometry Capabilities of PyGist and PyNarcisse

	PyGist	**PyNarcisse**
curves, including multiple	yes	yes
multiple disjoint lines	yes	not yet
quadrilateral mesh--line plot	yes	not yet
quadrilateral mesh--contour plot	yes	not yet
quadrilateral mesh--filled contour plot	not yet	yes
region plots	yes	not yet
filled polygons	yes	not yet
cell arrays	yes	not yet
2-D animation, real time	yes	never
color bar	yes	yes
axes in 3d plots	gnomon[a] only	yes
surfaces--wire mesh, monochrome	yes	yes
surfaces--wire mesh, colored by data	never	yes
surfaces--flat (color filled cells)	yes	yes
surfaces--contours, filled contours	not yet	yes
surfaces--shaded by light source	yes	not yet
3-D mesh--complete cells	never	yes
3-D mesh--isosurface and plane slices	yes	yes
3-D mesh--isosurface and plane slices, split palette	yes	not yet
3-D realtime animation--moving light source	yes	maybe someday
3-D realtime animation--rotation	yes	yes (slow)

a. The gnomon is a small representation of the coordinate axes at the lower left of the picture. The name of an axis is reverse video if it points into the plane of the graph.

TABLE 2 Device Capabilities of PyGist and PyNarcisse

	PyGist	PyNarcisse
Xwindow	yes	yes
multiple Xwindows	yes	yes
Xwindow on remote machine	yes	yes
file(s) only, no Xwindow	yes	never
CGM file(s)	yes	never
PostScript file(s)	yes	only from GUI
multiple files	yes	never
file in self-specific format	no	yes

2.2 Running OOG

Please read Chapter 1 first and follow the instructions there regarding the setting of various environment variables before running Python and PyGraph. Then, once you have fired up python, you need to execute `import` statements for each component of the OOG which you intend to use. There are two forms of the `import` statement.

- `from xxxx import *`

  (`xxxx` is the name of the file imported, but without the ‘‘`.py`’’ suffix.) This form imports the name space from file `xxxx` into the name space where the import statement is executed. Thus, if `foo` is a name in `xxxx`’s name space, then it may be referred to simply as `foo`.

- `import xxxx`

  This form imports only the name `xxxx`, so that if `foo` is a variable in the `xxxx` name space, then it must be referred to as `xxxx.foo`.

Following is a list of the OOG files available in PyGraph, and the names of the classes (capitalized) and functions (lower case) which are declared in the files which you may want to use:

```
curve.py: Curve
lines.py: Lines
quadmesh.py: QuadMesh
region.py: Region
polymap.py: Polymap
cellarray.py: CellArray
surface.py: Surface
mesh3d.py: Mesh3d, Slice, slice
plane.py: Plane
graph.py: Graph (not normally instantiated alone)
```

```
graph2d.py: Graph2d
graph3d.py: Graph3d
animation2d.py: Animation2d
Nar.py: Plotter
Gist.py: Plotter
```

Note that if you want to instantiate both a PyNarcisse and a PyGist Plotter, you must use the ‘‘`import xxxx`’’ form of the `import` statement.

2.3 Class Summary

Here is a summary of the PyGraph classes which are described in the remainder of this manual.

- Two-dimensional geometric objects (CHAPTER 3: “Two-Dimensional Geometric Objects”)

```
c1 = Curve ( <keylist>)
l1 = Lines ( <keylist>)
qm = QuadMesh ( <keylist>)
rg = Region ( <keylist>)
pm = Polymap ( <keylist>)
ca = CellArray ( <keylist>)
```

- Three-dimensional geometric objects (CHAPTER 4: “Three-Dimensional Geometric Objects”)

```
sf = Surface ( <keylist>)
m3 = Mesh3d ( <keylist>)
pl = Plane ( <normal>, <point>)
sl = slice (m, val [, varno])    # slice is a function
sl = slice (m, plane [, varno])  # slice is a function
sl = slice (s, plane [, nslices])# slice is a function
sl = Slice (nv, xyzv [, val [, plane [, iso]]])
```

- Graph objects (CHAPTER 5: “Graph Objects”)

```
g2 = Graph2d ( <object list>, <keylist>)
g3 = Graph3d ( <object list>, <keylist>)
```

- Animation objects (CHAPTER 6: “Animation2d Objects”)

```
anim = Animation2d ( <keylist>)
```

- Plotter objects (CHAPTER 7: “Plotters: A Brief Primer”)

```
pl = Nar.Plotter ( [ <filename>] [, <keylist>])
pl = Gist.Plotter ( [ <filename>] [, <keylist>])
```

CHAPTER 3: Two-Dimensional Geometric Objects

Two-dimensional geometric objects available in OOG include: `Curve`, `Lines` (a collection of disjoint lines), `QuadMesh` (as its name implies, a quadrilateral mesh), `Region` (a sub-part of a `QuadMesh`), `PolyMap` (a two-dimensional layout of polygons, each with an associated color), `CellArray` (a two-dimensional array of rectangular cells, each with an associated color), and `Animation2d` (a specification of an animation, which includes initialization, calculation, and update functions). All of these objects are available in PyGist, but PyNarcisse supports only `Curve` objects in two dimensions (Narcisse is primarily a three and four dimensional plotting engine).

`Animation2d` objects are the subject of a separate chapter; see CHAPTER 6: “Animation2d Objects” on page 109.

3.1 `Curve` Objects

To use `Curve` objects, you must import the Python module contained in file `curve.py`.

Instantiation

```
from curve import *
c1 = Curve ( <keylist>)
```

Description

A `Curve` object consists of the coordinates and other characteristics of a geometric curve. You use ‘‘`Curve`’’ to create one, and the other methods of the `Curve` class to make a new `Curve` out of the old one or change `Curve` characteristics. Here is a short description of the methods of the Curve class:

`set`: used to set one or more keyword arguments to new values. Warning--very little error checking is done; it may be possible to set keywords to conflicting values using this method.

`new`: reinitializes a `Curve` object for reuse. The arguments are the same as for `Curve`.

The keyword arguments are all of the form ‘‘`keyword = <value>`’’. Most are optional and will be assigned sensible values if omitted.

Keyword Arguments

The following keyword arguments can be specified for a `Curve` object:

`y`, `x`, `color`, `axis`, `label`, `type`, `marks`, `marker`, `width`, `hide`

Descriptions of the keywords are as follows:

`y = <sequence of floating point values>` (required): the y coordinates of the curve.

`x = <sequence of floating point values>` (optional): the x coordinates of the curve. If not specified, y will be plotted versus its subscript range.

`color = <value>` where `<value>` is an integer from 0 to 63 (PyNarcisse) or 0 to 199 (PyGist) representing an entry in a color chart, or else a string giving a common color name. The common color names refer to colors on the default color card ‘‘`rainbowhls`’’ (PyNarcisse) or ‘‘`rainbow.gp`’’ (PyGist). The allowed names are `"background"`, `"foreground"` (the default), `"blue"`, `"green"`, `"yellow"`, `"orange"`, `"red"`, `"purple"`, `"cyan"`, `"magenta"`, `"gold"`, `"yellowgreen"`, `"orangered"`, `"redorange"`, `"black"`, and `"white"`. The abbreviations `"fg"` and `"bg"` are also allowed. On this color card the numbers and their corresponding colors for PyNarcisse are roughly: 10-23: dark shading to light blue; 24-39: greens; 40-43: yellow to gold; 44-47: oranges; 48-57: reds; and 58-63: purples. PyGist shades will be similar but scaled from 0 to 199.

`axis = "left"` or `"right"` tells whether the left or right y axis will be assigned to this curve. (Narcisse allows two y axes with different scales, one on the left of the plot and one on the right; this option is not available in PyGist.)

`label = <string>` represents the label of this curve. In PyGist, the label will be a single character appearing periodically along the curve. In PyNarcisse, the label may be more than one character, and will appear opposite the right end of the curve.

`type = <value>` tells how the curve will be plotted: `"line"`, `"solid"` (same as `"line"`), `"step"`, `"dash"`, `"dashdot"`, `"dashdotdot"`, `"none"`, `"+"`, `"*"`, `"o"`, `"x"`, and `"."` are allowed. If the option is not available in a particular graphics package, a good guess will be substituted. If `type = "none"` and `marks = 1`, the plot will be a polymarker plot, if supported by the graphics. Note that because of disparities among graphics packages supported, you can specify plotting a curve pointwise with symbols like `"+"`, `"*"`, etc., either by use of the `type` variable or by using `marks` and `markers` in conjunction with `type = "none"`.

`marks = 0` or `1`; select unadorned lines (0) or lines with occasional markers (1). PyNarcisse does not support this option. The markers default to letters of the alphabet, but can be changed by the `marker` keyword.

`marker =` character or integer value for character used to mark this curve if `marks = 1`. Special values `'\1'`, `'\2'`, `'\3'`, `'\4'`, and `'\5'` stand for point, plus, asterisk, circle, and cross, which sometimes look prettier than characters on some devices. `"."`, `"+"`, `"*"`, `"o"`, and `"x"` are also allowed.

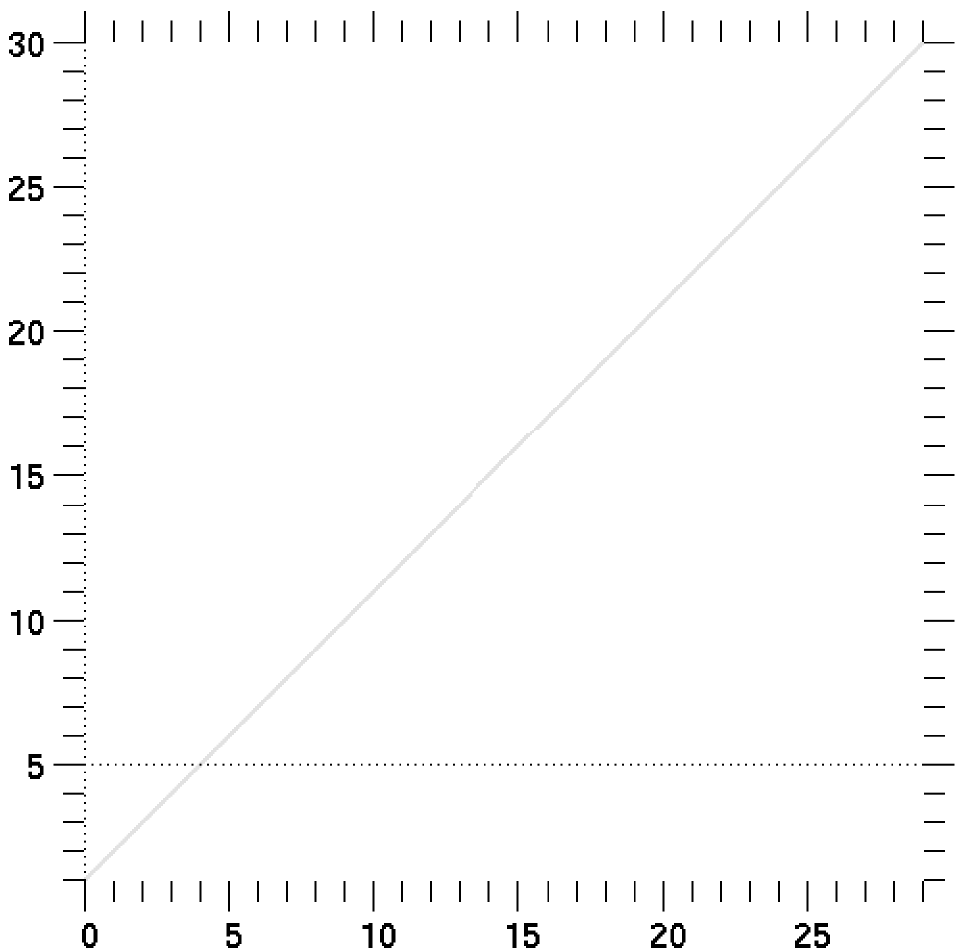
`width =` real number; specifies the width of a curve if this is supported by the graphics. 1.0 gives a finely drawn curve and is the default.

`hide = 0` or `1`; if set to 1, this curve will be hidden on the plot.

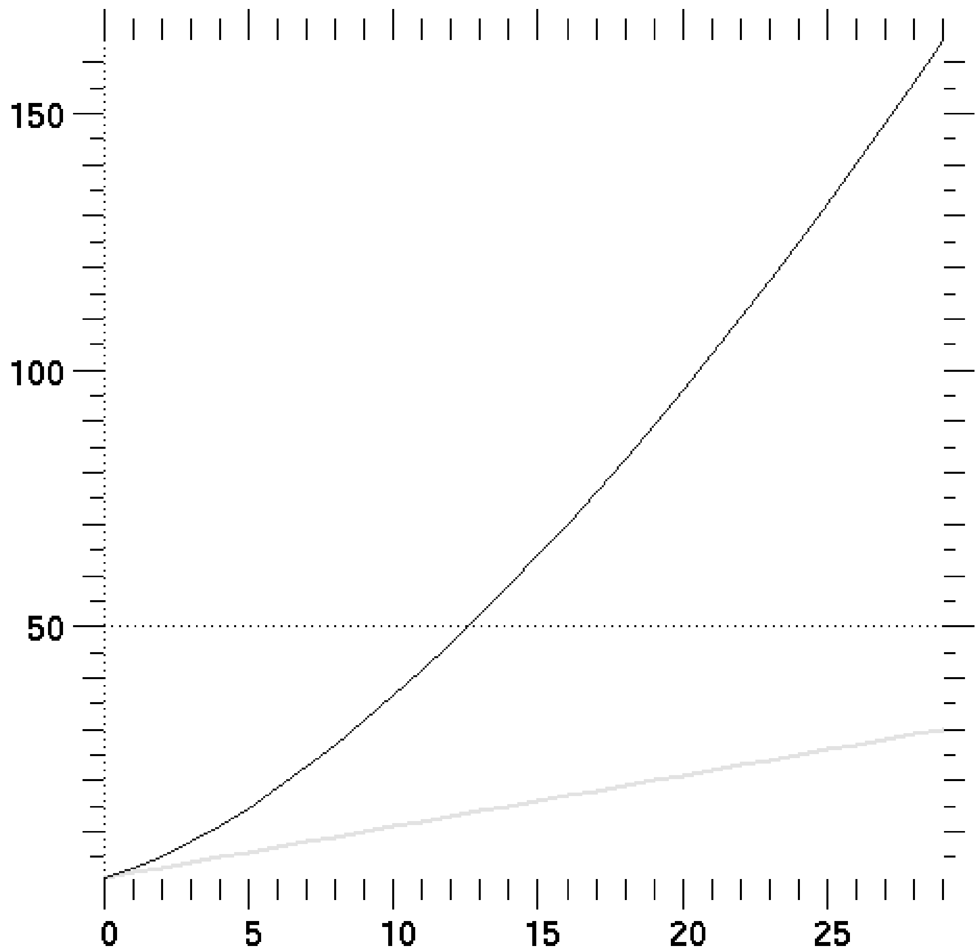
Examples

In the following example, two curves with different characteristics are created and plotted. The comments in the code explain what is going on. We use only the simplest (and minimal) properties of a `Graph2d` object for this example.

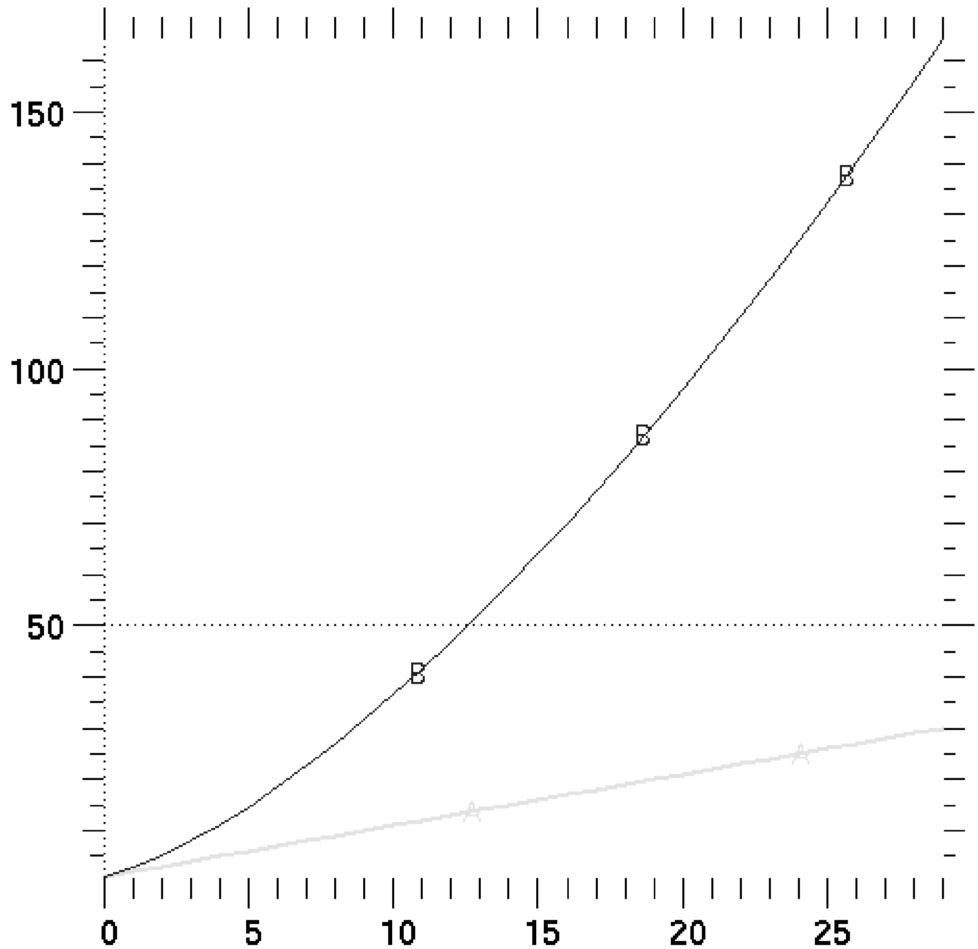
```
from curve import *
from graph2d import *
# instantiate first Curve:
c1 = Curve ( y = arange (1, kmax+1, typecode = Float) ,
    color = "yellow" , width = 4)
# create a Graph2d containing this curve:
g2 = Graph2d ( c1 )
# plot this curve (Graph2d creates a default Plotter):
g2.plot ( )
```



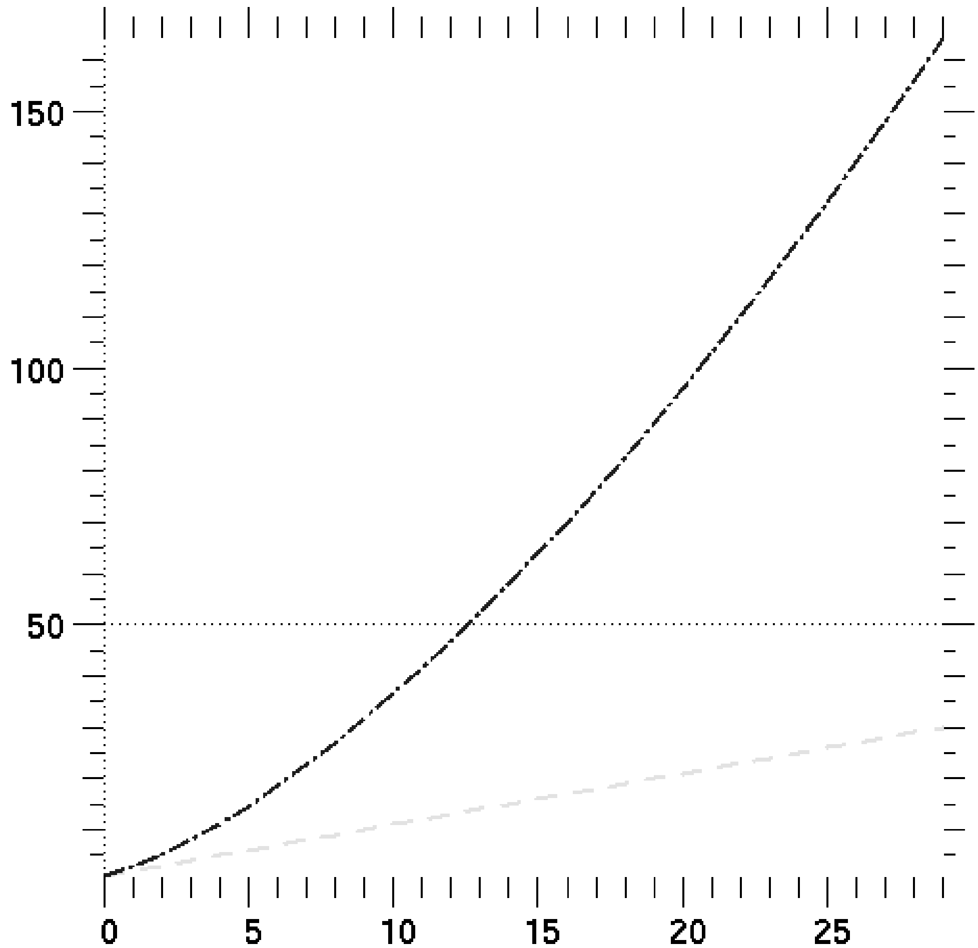
```
# Create a second Curve:
c2 = Curve (
   y = sqrt (arange (1, kmax+1, typecode = Float)**3) ,
   color = "blue")
# Add it to the Graph:
g2.add ( c2 )
# Plot the two curves:
g2.plot ( )
```



```
# Change the two curves to have markers:
c1.set (marks = 1, marker = "A")
c2.set (marks = 1, marker = "B")
# Replot with new characteristics:
g2.plot ( )
```



```
# change yet again:
c1.set (marks = 0, width = 3.0, type = "dash")
c2.set (marks = 0, width = 4.0, type = "dashdot")
# Replot:
g2.plot ( )
```



Note that the changes we made to curve instances `c1` and `c2` did not need to be transmitted to the `Graph2d` instance `g2`. `g2` has *references* to `c1` and `c2`, not *copies* of them; hence any changes made to the curves will be known to `g2`. This is characteristic of Python: it passes objects by reference rather than by value, which, particularly for large objects, saves a lot of copying overhead.

At this point we used only very simple `Graph2d` properties so as not to distract from the fact that we are currently emphasizing curves. For thorough discussions and examples of `Graph2d`, Section 5.1 on page 79.

3.2 Lines Objects

This class is not currently supported by PyNarcisse.

Instantiation

```
from lines import *
l1 = Lines ( <keylist>)
```

Description

A `Lines` object contains the specifications for a set of disjoint lines. It has only keyword arguments, in the form ''`keyword = <value>`''. It has methods `set` and `new`, which function the same as the `Curve` methods by the same name (Section on page 9). The following keywords arguments are allowed:

`x0, y0, x1, y1, color, hide, width, type`

These keywords are described in the next subsection.

Keyword Arguments

The following keyword arguments can be specified for a `Lines` object:

`x0 = <sequence of floating point values>`

`y0 = <sequence of floating point values>`

`x1 = <sequence of floating point values>`

`y1 = <sequence of floating point values>`

`x0`, `y0`, `x1`, and y1 can actually be scalars, but if arrays must match in size and shape. `(x0[i], y0[i])` represents the starting point of the `i`th line, and `(x1[i], y1[i])` represents its endpoint.

`color =` one of the legal values for PyGist (currently the only package supporting `Lines`). See `gist.help` for details.

`hide = 0/1` (`1` to hide this part of the graph)

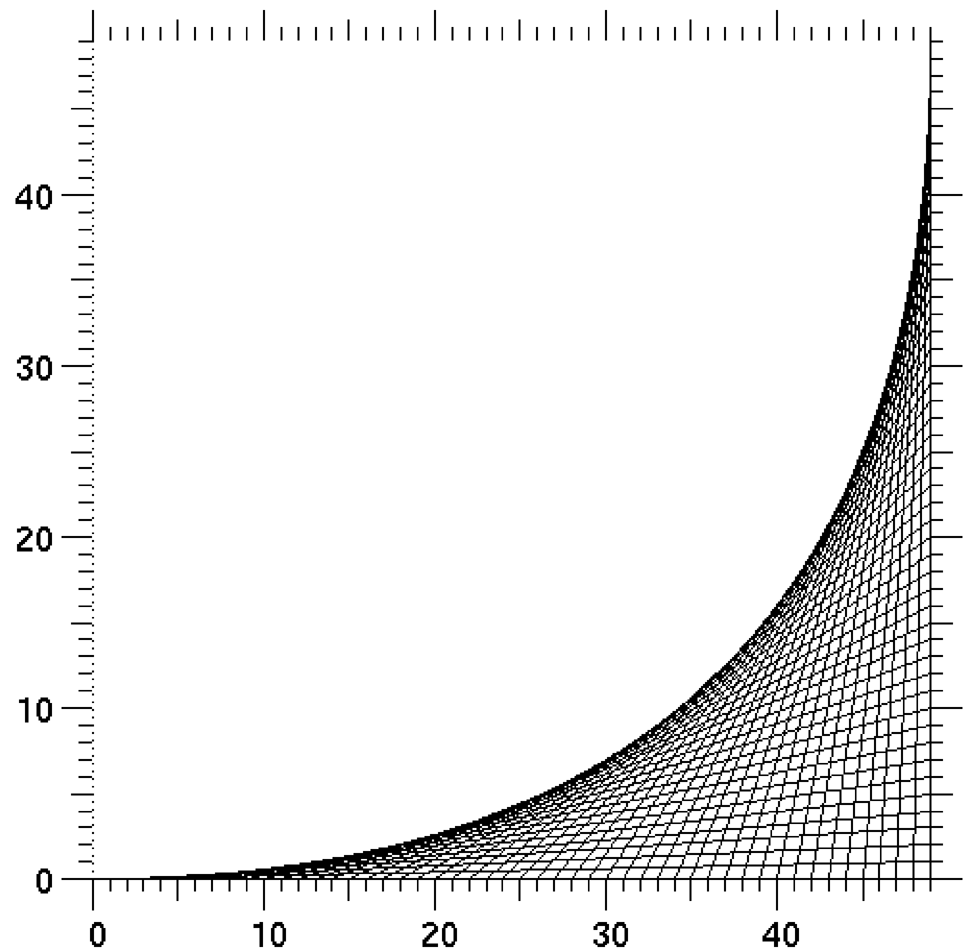
`width =` width of the lines. 1.0 (pretty narrow) is the default. Successive values 2.0, 3.0, ... roughly represent width in pixels.

type = `"solid"`, `"dash"`, `"dot"`, `"dashdot"`, `"dashdotdot"`, and `"none"` (in which case the lines will be plotted as characters).

Example 1

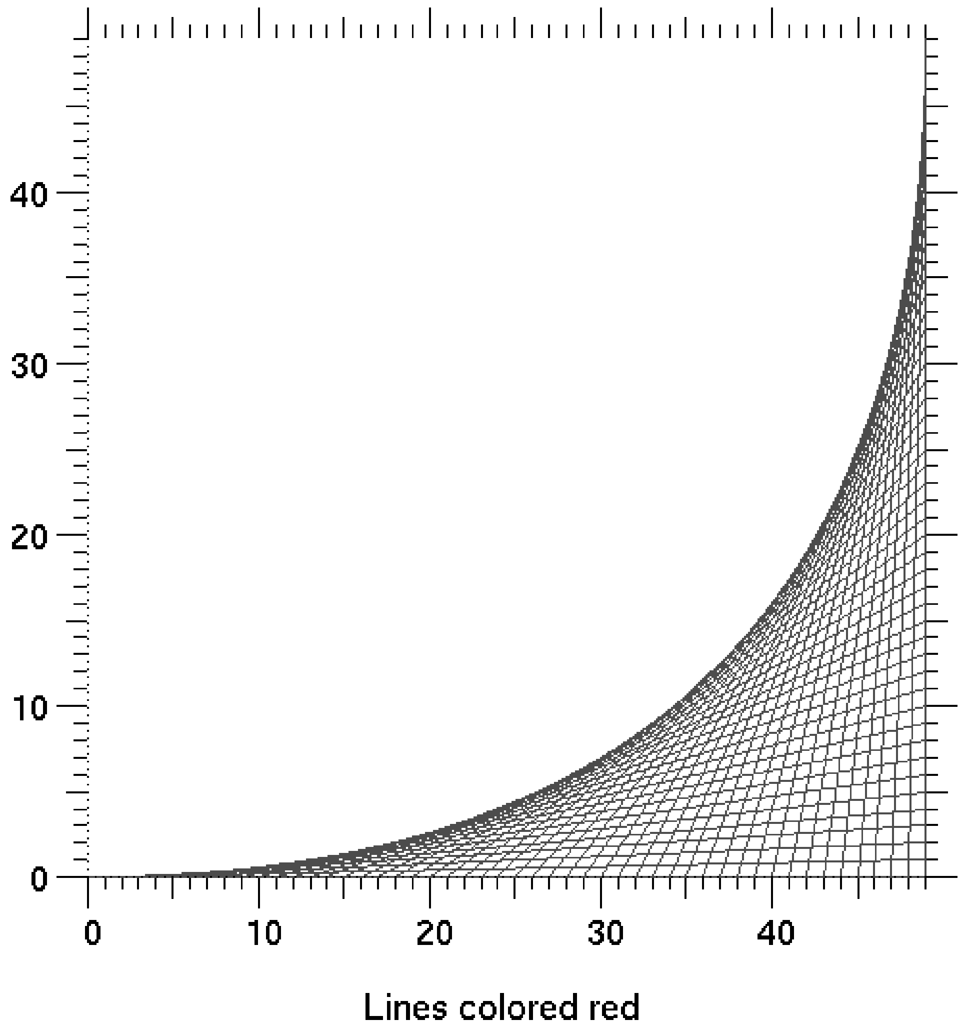
The first example draws a series of lines starting at a set of equally spaced points along the x axis and ending at a set of equally spaced points along the vertical line `x = 49`. Subsequent commands change the plot as explained in the comments.

```
from lines import *
# Set up the points along x axis:
x0 = arange(50, typecode = Float)
y0 = zeros(50, Float)
# Set up the points along the line x = 49:
x1 = 49 * ones(50, Float)
y1 = arange(50, typecode = Float)
# Instantiate the Lines object ly:
ly = Lines (x0 = x0 ,y0 = y0, x1 = x1, y1 = y1)
# Instantiate a graph2d containing ly, with
# (bottom) title "Just Lines":
g0 = Graph2d ( ly , titles = "Just Lines")
# Plot the graph:
g0.plot ()
```

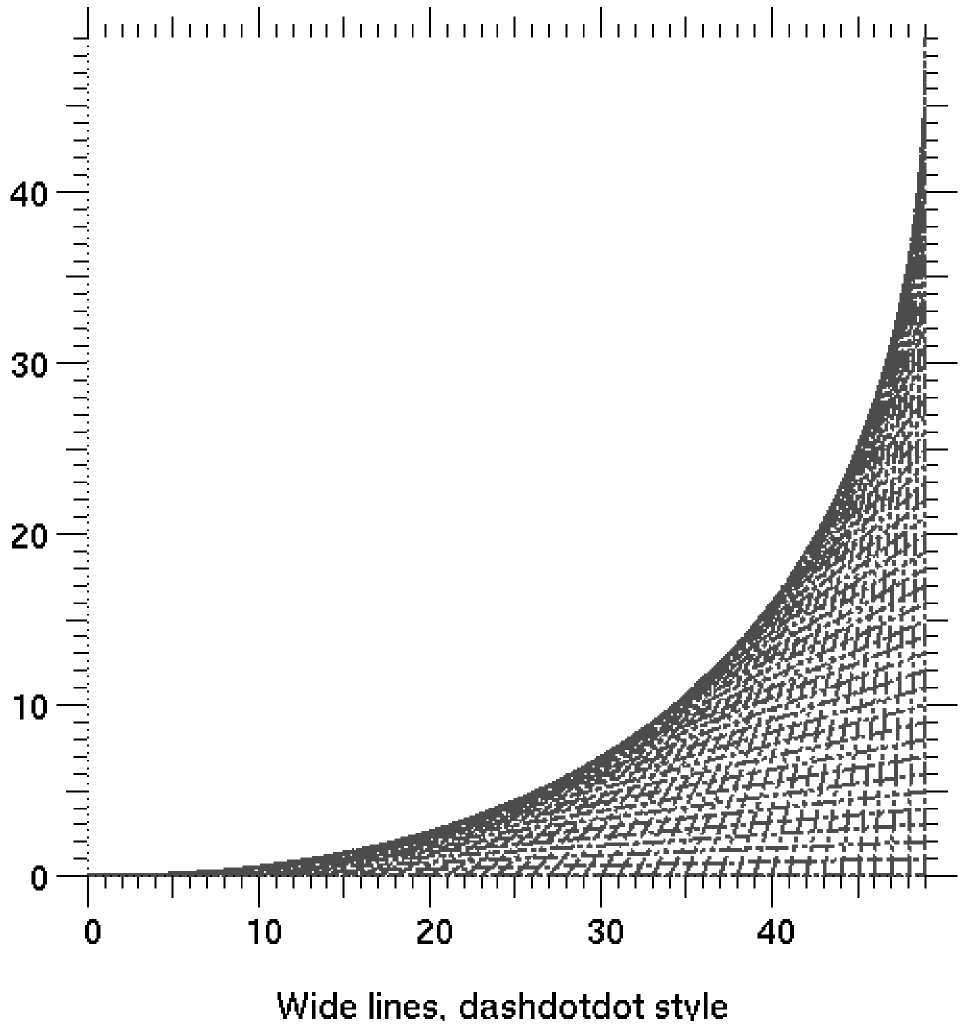


Just Lines

```
# Set the color of the lines to red; note that Lines,
# like Curve, has a method set:
ly.set (color = "red")
# Change the title to reflect this:
g0.change (titles = "Lines colored red")
# Plot the new graph:
g0.plot ()
```



```
# Now make the lines wider and change their type:
ly.set (width=4.0,type="dashdotdot")
# Change the title to reflect this:
g0.change(titles = "Wide lines, dashdotdot style")
# Plot the new graph:
g0.plot ()
```



Note once again that the `Graph2d` object `g0` contains a *reference* to `ly`; hence when we change some of the characteristics of `ly`, `g0` will know about these changes.

Example 2

The second example is more complicated, but is worth studying. It uses two functions to compute the endpoints of the lines; let us examine these functions first. Note that function `a2` assumes that the `Numeric` module's name space has been imported.

```
def a2 (lb, ub, n) :
   return reshape (array ((n - 1) * spanz (lb, ub, n),
      Float), (n-1, n-1))
```

This function takes what `spanz` returns (which is a sequence of `n - 1` items, as we shall see below), concatenates it to itself `n - 1` times (the "`*`" is *not* multiplication, but replication), turns it into an array, then reshapes it into a two-dimensional array `n - 1` by `n - 1`.

The `spanz` function is as follows:

```
def spanz (lb, ub, n) :
   if n < 3 : raise ValueError, \
      "3rd argument must be at least 3"
   c = 0.5 * (ub - lb) / (n - 1.0)
   b = lb + c
   a = (ub -c -b) / (n - 2.0)
   return map (lambda x, A = a, B = b:
    A * x + B, range (n - 1))
```

The `spanz` function divides the interval from `lb` to `ub` into `n` parts, such that the interior subintervals are of equal length, and the two end subintervals are each half of that length.and returns the sequence of `n - 1` equally spaced points which divide it into these `n` parts. If you do not understand how this function works, we recommend it as an excellent exercise to learn about the application of the `map` function and the `lambda` operator in Python.[1]

Although this manual is not a Python text, it might be instructive to study the following version of function `a2`, which does the same thing as the above `a2` and `spanz` together:

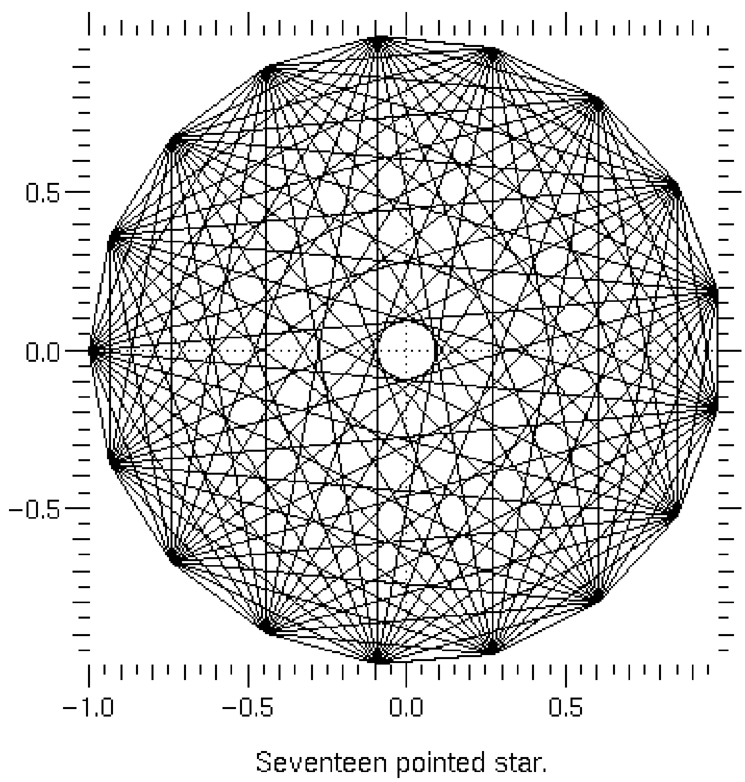
```
def a2 (lb, ub, n) :
   return multiply.outer (ones (17, Float),
      arange (n - 1, typecode = Float) * (ub - lb) / (n - 1) +
     (ub - lb) / (2 * (n - 1)))
```

With the help of these two auxiliary functions, or just the latter one, if you prefer, the following code will draw the graph of an interesting seventeen-pointed star:

```
# Create the endpoints:
theta = a2 (0, 2*pi, 18)
x = cos (theta)
```

1. See the *Python Library Reference*, page 17, for a description of `map`. The *Python Reference Manual*, p. 29, describes `lambda` forms.

```
y = sin (theta)

from lines import *
# Instantiate the lines object:
ln = Lines (x0 = x, y0 = y, x1 = transpose (x),
    y1 = transpose (y))
# Instantiate a Graph2d object containing ln, with the x
# and y axes in equal scale, and an informative title:
g1 = Graph2d (ln, xyequal = 1,
    titles = "Seventeen pointed star")
# Plot the graph:
g1.plot ()
```



3.3 QuadMesh Objects

Currently only PyGist supports `QuadMesh`es.

Instantiation

```
from quadmesh import *
qm = QuadMesh ( <keylist>)
```

Description

The `QuadMesh` class provides a means of encapsulating information about two-dimensional, quadrilateral meshes and plotting the information connected with these meshes in various ways. Information can be plotted as contour lines, filled contours, or filled cells; different regions of the mesh can be plotted with different characteristics; and vector fields can be plotted on all or part of the mesh. The keyword arguments for QuadMesh objects are:

> **`x, y, ireg, boundary, boundary_type, boundary_color, regions, region, inhibit, tri, z, levels, filled, contours, edges, ecolor, ewidth, vx, vy, type, color, hide, width, marks, marker`**

`QuadMesh`, like all other 2d classes, also has methods `set` and `new`.

Keyword Arguments

`x` and `y`, matching two-dimensional sequences of floating point values. These arguments are required and give the coordinates of the nodes of the mesh.

`ireg`, the region map: optional two-dimensional sequence of integer values with the same dimensions as `x` and `y`, giving positive region numbers for the cells of the mesh, zero where the mesh does not exist. The first row and column of `ireg` should be zero (although these values will be ignored), since there are one fewer cells in each direction than there are nodes.

`boundary` = `0/1`; `0`: plot entire mesh; `1`: plot only the boundary of the selected region(s).

`boundary_type`, `boundary_color`: these matter only if `boundary` = `1`, and tell how the boundary will be plotted (`"solid"`, `"dash"`, `"dot"`, `"dashdot"`, `"dashdotdot"`, or `"none"`) and what its color will be.

`region` = `n:` if `n` = `0`, plot entire mesh; if any other number, plot the region specified (according to the settings in `ireg`).

`regions` = [r_1, r_2, ...]: this option allows the user to specify to a `QuadMesh` a list of `Region` objects (Section 3.4 on page 34) to plot. Each object may have different plotting characteristics (Section on page 35). Only regions r_1, r_2, ... will be plotted.

`regions` = `"all"` (the default): plot all regions of the mesh.

`inhibit` = `0/1/2`; `0`: Plot all mesh lines. `1`: Do not plot the `(x [:, j], y [:, j])` lines; `2`: Do not plot the `(x  [i,:], y[i,:])` lines; `3`: If `boundary` = `1`, do not plot boundaries. (Default: 0.) `0`, `1`, and `2` only apply if `edges` = `1`.

`tri`, optional two-dimensional sequence of values with the same dimensions as `ireg`, triangulation array used for contour plotting.

`z` = optional two-dimensional sequence of floating point values. Has the same shape as `x` and `y`. If present, the contours of `z` will be plotted (default: 8 contours unless `levels` (see below) specifies otherwise), or a filled mesh will be plotted if `filled` = `1`. In the latter case, `z` may be one smaller than `x` and `y` in each direction, and represents a zone-centered quantity.

`levels` = either:

(1) optional one-dimensional sequence of floating point values. If present, a list of the values of `z` at which you want contours; or

(2) if a single integer, represents the number of contours desired. They will be computed (at equal levels) by the graphics.

`filled` = `0/1`: If `1`, plot a filled mesh using the values of `z` if `contours = 0`, or plot filled contours if `contours = 1` (this option is not available at the time of writing of this manual, but will be added soon). If `z` is not present, the mesh zones will be filled with the background color, which allows plotting of a wire frame. (default value: `0`.)

`contours` = `0/1`: if `1`, contours will be plotted if `filled = 0`, and filled contours will be plotted if `filled = 1` (this option is not available at the time of writing of this manual, but will be added soon). `contours` normally defaults to `0`, but will default to `1` if `edges = 0` and `filled = 0`.

The table below summarizes the effects of these two keywords (assuming `z` is present):

TABLE 3 `filled` and `contours`

	`contours = 0`	`contours = 1`
`filled = 0`	`k` and/or `l` lines if `edges = 1`	contour lines
`filled = 1`	filled mesh ("bg" fill if `z = None`)	filled contours

`edges`, if nonzero, draw a solid edge around each zone. If `edges = 0` and `filled = 0`, draw contour lines. (Default value: `0`.)

`ecolor`, `ewidth`--the color and width of the mesh lines when `filled = 1` and `edges` is nonzero.

`vx`, `vy`--optional two-dimensional sequences of floating point values. Has the same shape as `x` and `y`. If present, represents a vector field to be plotted on the mesh.

`scale` = floating point value. When plotting a vector field, a conversion factor from the units of `(vx, vy)` to the units of `(x, y)`. If omitted, `scale` is chosen so that the longest vectors have a length comparable to a ‘‘typical’’ zone size.

`z_scale` = specifies `"log"`, `"lin"`, or `"normal"` for how `z` is to be plotted.

`type`, `color`, `width`, `label`, `hide`, and `marks` are as for curves.

`marker` is different, since you would not want to specify the same marker for all contours in a contour plot. Instead, you can use `marker` to designate the letter (or number) which you want to mark the lowest contour curve; then the remaining contours will be lettered or numbered consecutively from that point on.

Methods `new` and `set` are as in the `Curve` class. Remember the warning about `set`: very little error checking is done, so if you are not careful, you could assign conflicting values to keywords.

Examples

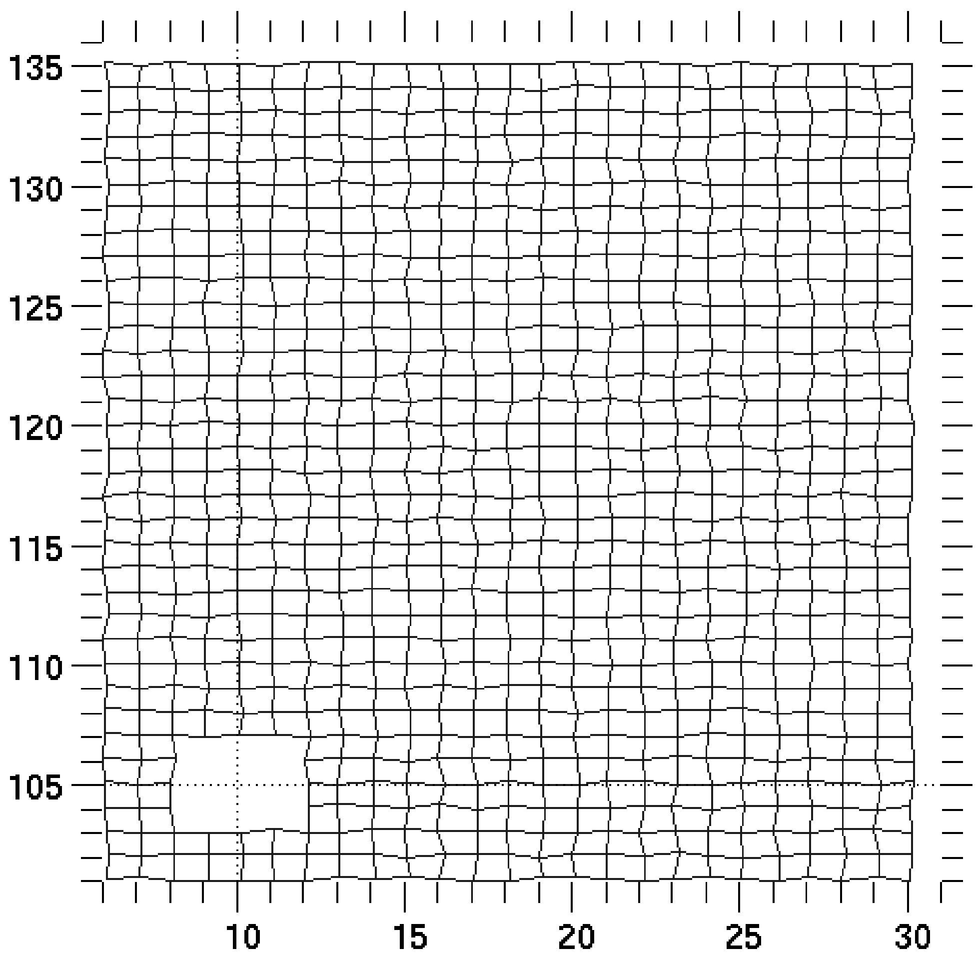
The following Python code computes a mesh and some data on the mesh to be used in the `QuadMesh` examples which follow. This same code will be assumed in the examples given in the next section on `Region`s. Note the `import` statements, which bring in the necessary name spaces to do the computations.

```
from quadmesh import *
from graph2d import *
from Ranf import *
from Numeric import *
from shapetest import *
s = 1000.
kmax = 25       # The mesh is going to be 25 by 35
lmax = 35       # (24 cells by 34)
xr = multiply.outer ( arange (1, kmax + 1, typecode = Float),
   ones (lmax))
yr = multiply.outer ( ones (kmax), arange (1, lmax + 1,
   typecode = Float))
zt = 5. + xr + .2 * random_sample (kmax, lmax)
rt = 100. + yr + .2 * random_sample (kmax, lmax)
z = s * (rt + zt)
z = z + .02 * z * random_sample (kmax, lmax)
z [3:10, 3:12] = z [3:10, 3:12] * .9
z [5, 5] = z [5, 5] * .9
z [17:22, 15:18] = z [17:22, 15:18] * 1.2
z [16, 16] = z [16, 16] * 1.1
# Define a vector field on the mesh:
ut = rt/sqrt (rt ** 2 + zt ** 2)
vt = zt/sqrt (rt ** 2 + zt ** 2)
# Define the region map:
ireg =  multiply.outer ( ones (kmax), ones (lmax))
# The first row and column should be 0:
ireg [0:1, 0:lmax]=0
ireg [0:kmax, 0:1]=0
ireg [1:15, 7:12]=2
ireg [1:15, 12:lmax]=3
# Create a void in the mesh:
ireg [3:7, 3:7]=0
```

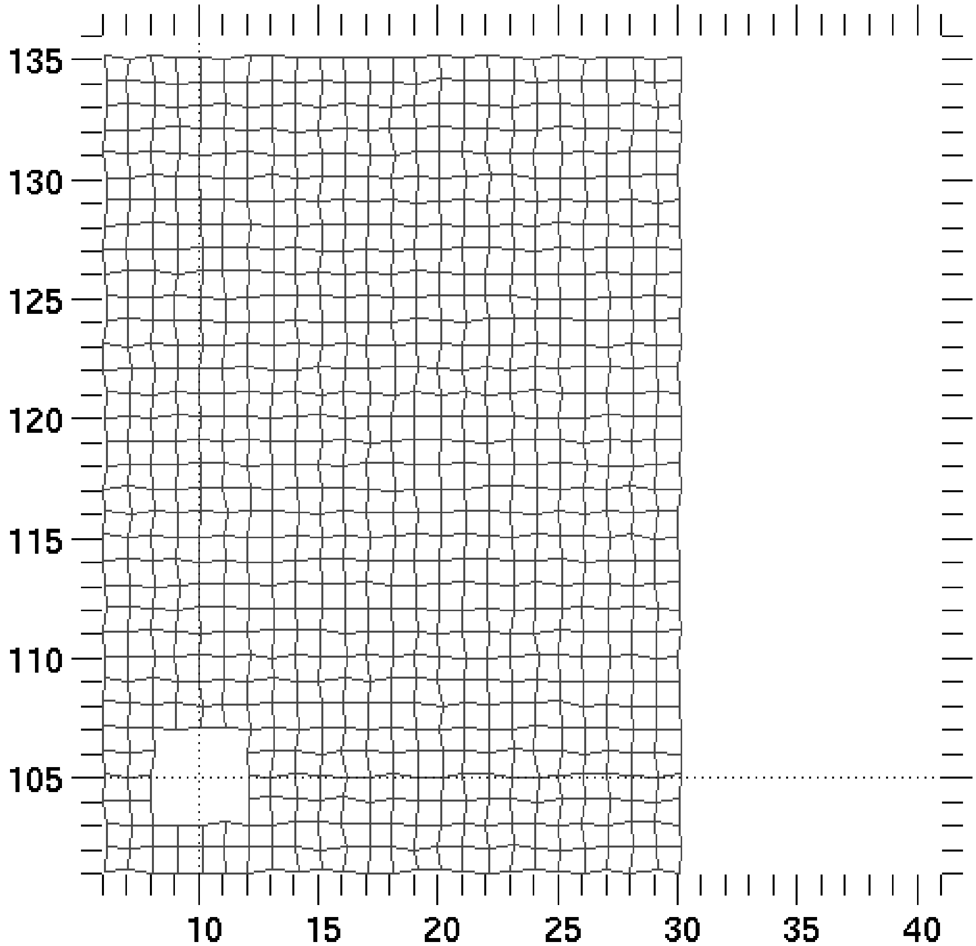
3.3.1 Plots of Mesh Lines

The following code plots the mesh lines in three different ways, as described in the comments:

```
# Instantiate a QuadMesh object qm with the mesh defined
# by zt, rt, and ireg; its lines to be of width 1,
# and blue in color:
qm = QuadMesh (x = zt, y = rt, ireg = ireg, width = 1.,
   color = "blue")
# Create a Graph2d object gr with a reference to qm:
gr = Graph2d (qm)
# Plot the graph. Note the void area in the graph.
gr.plot ()
```

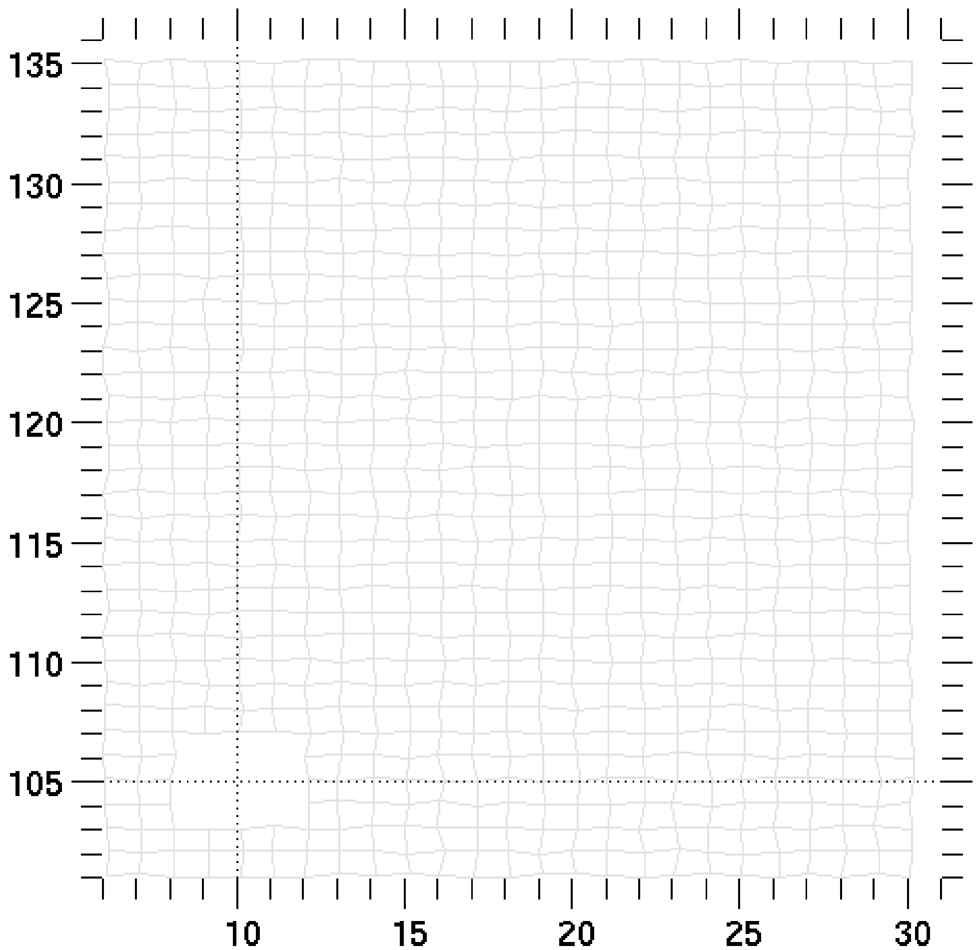


```
# Change to a red-colored mesh:
qm.set (color = "red", width = 1.)
# Change the plot so that the x and y axes have the same
# scale (the mesh will appear narrower)1:
gr.change_plot(xyequal = 1)
```



1. Note: the `Graph` method `change_plot` changes the appearance of the existing plot; there is no need to issue another plot call. See “Description” on page 79.

```
# Change the color of the mesh to yellow:
qm.set (color = "yellow")
# Let Gist calculate the x and y scales depending
# on the data:
gr.change_plot(xyequal = 0)
```

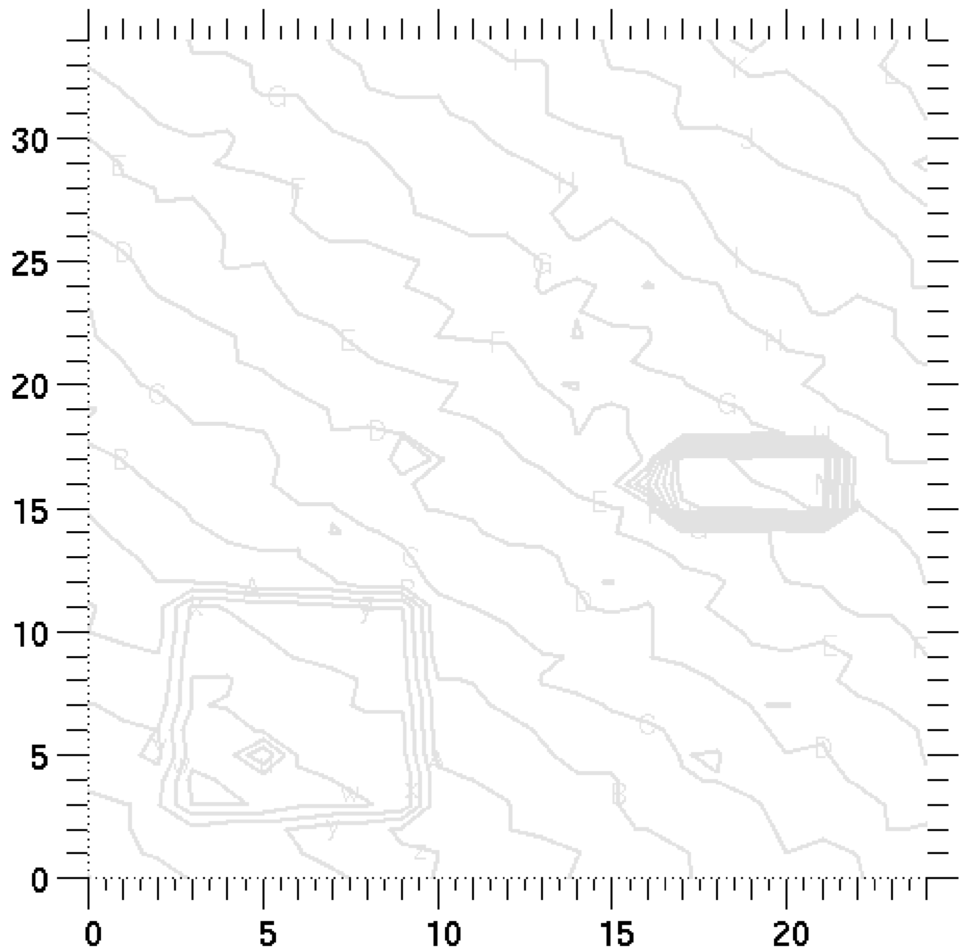


3.3.2 Contour Plots

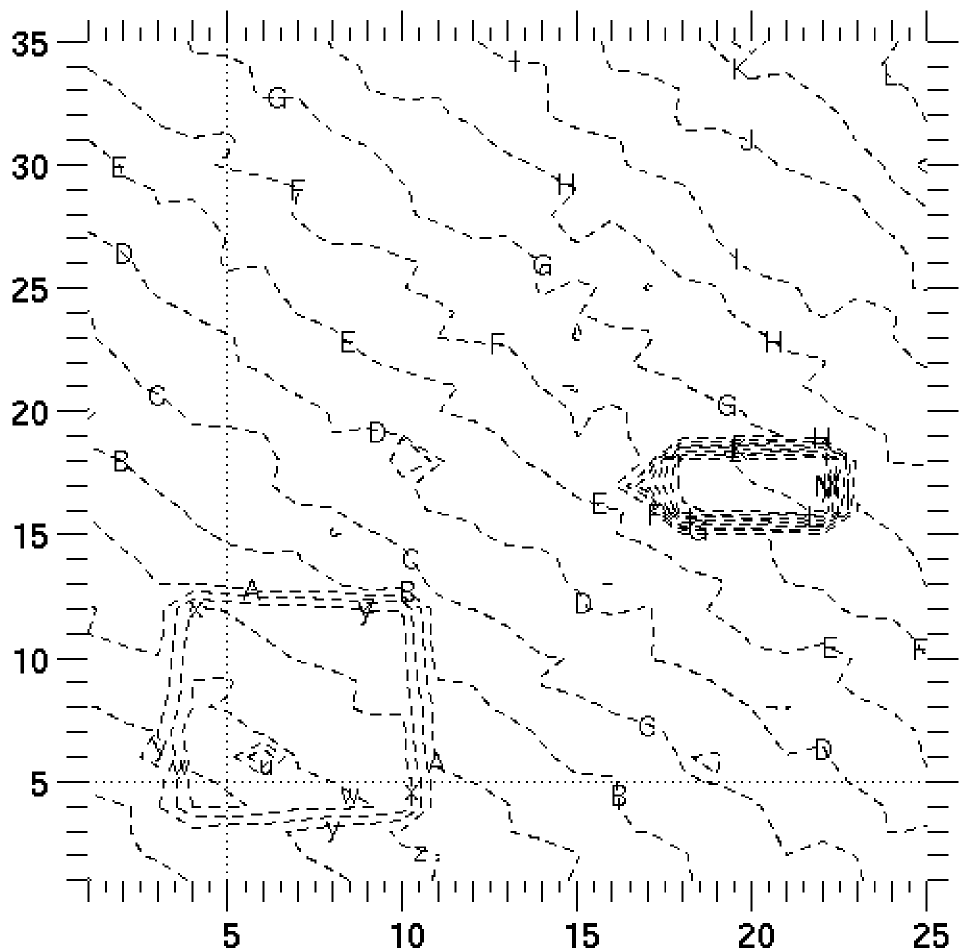
In this example, we create a uniform 25 by 35 mesh, and do a contour plot of the values of z computed above. Then we go back to the `(xr, yr)` mesh used above. We use the same `Graph2d` object as the preceding example, deleting object 1 and then adding the new `QuadMesh` object each time.

```
sh = shape (z)
sh1 = sh[0]
```

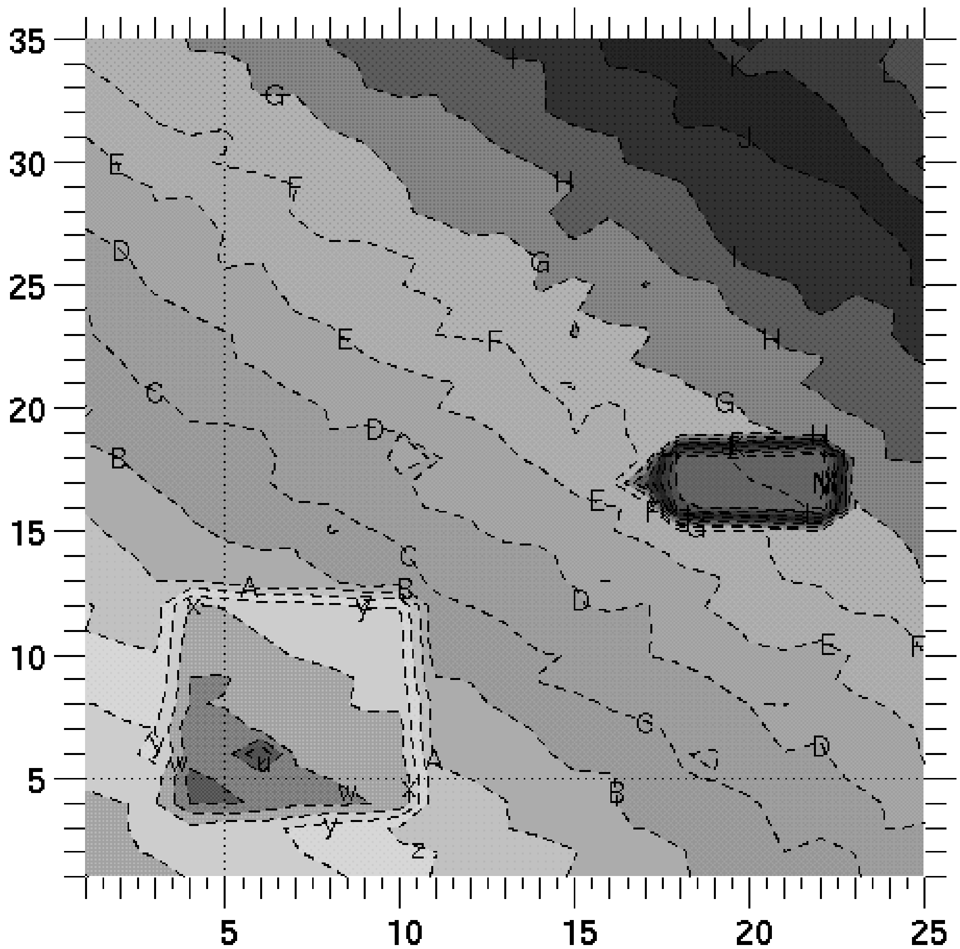
```
sh2 = sh[1]
x = multiply.outer (arange (sh1, typecode = Float),
   ones (sh2, Float))
y = multiply.outer (ones (sh1, Float),
   arange (sh2, typecode = Float))
# qm2 will have twenty yellow contour levels
# with default labels (capital letters):
qm2 = QuadMesh (z = z, y = y, x = x, color = "yellow",
   width = 3., levels = 20, marks = 1)
# Delete object 1 (the only one) from gr:
gr.delete (1)
# Add the new object to gr, and plot it:
gr.add (qm2)
gr.plot ()
```



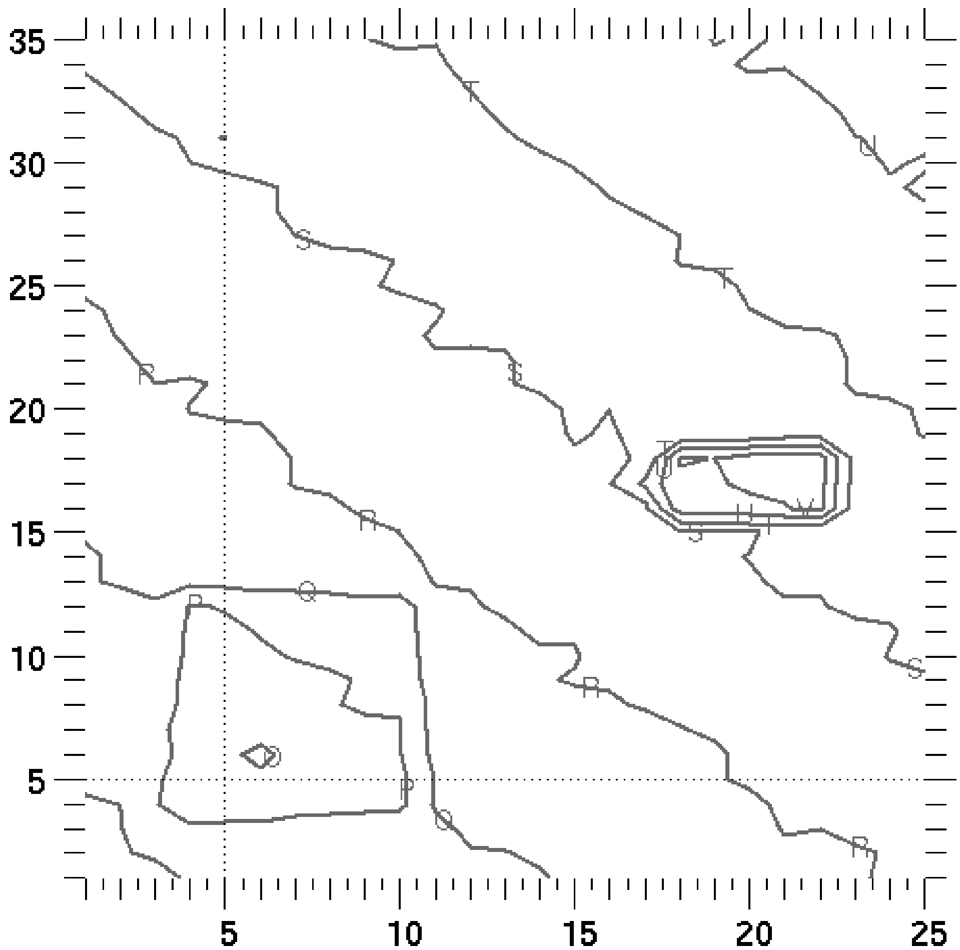
```
# Now change back to (xr, yr), mesh plotted with dashes
# in the foreground color:
qm2 = QuadMesh (z = z - z[kmax / 2, lmax / 2] ,
   y = yr, x = xr, type = "dash",
   color = "fg", levels = 20, width = 2., marks = 1)
gr.delete (1)
gr.add (qm2)
gr.plot ()
```



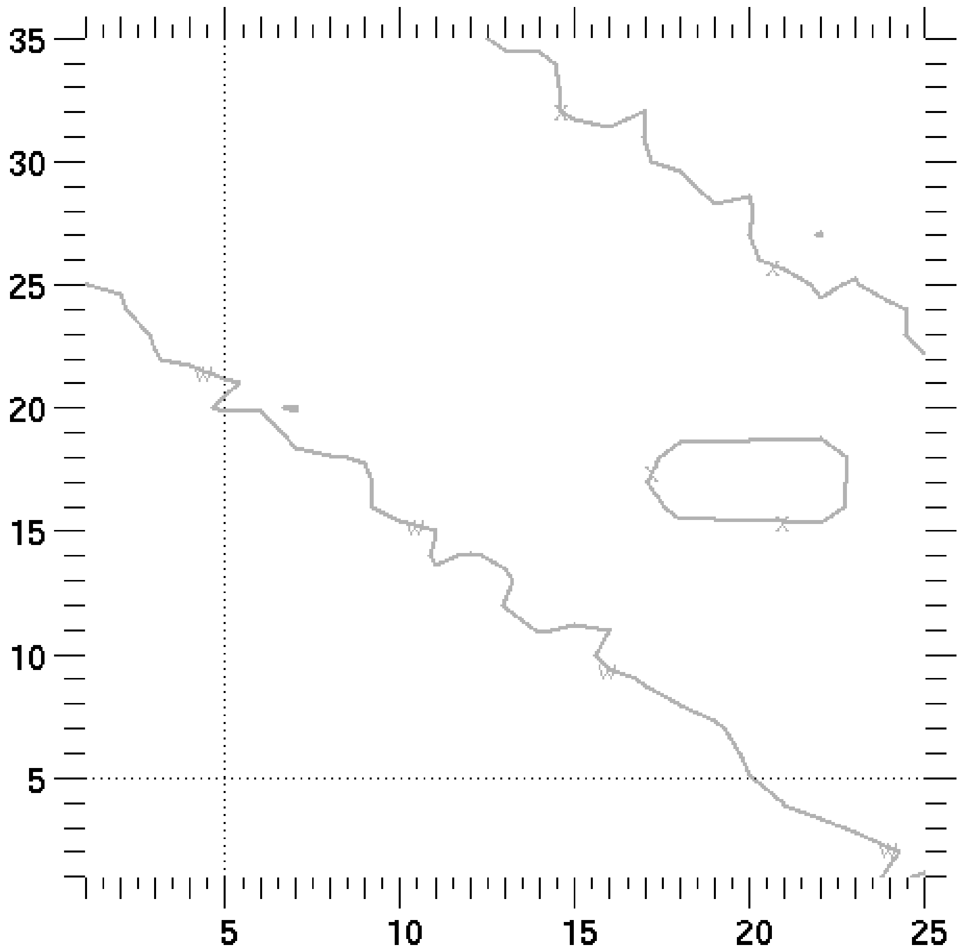
```
# Now plot the same thing as a filled contour plot:
qm2.set(filled=1)
gr.plot ()
```



```
# Next, plot the default number of contours (8) in
# purple with width 3; start marking with letter "O":
qm2 = QuadMesh (z = z, y = yr, x = xr , color = "purple" ,
   marks = 1, marker = "O", width = 3.)
gr.delete (1)
gr.add (qm2)
gr.plot ()
```



```
# Finally, plot four specified contour levels (three
# contours) in cyan, width 3:
qm2 = QuadMesh (z = z, y = yr, x = xr , color = "cyan" ,
   marks = 1, width = 3., levels = [0., max (ravel (z)) / 4.,
   3. * max (ravel (z)) / 4., 7. * max (ravel (z)) / 8.])
gr.delete (1)
gr.add (qm2)
gr.plot ()
```



3.4 **`Region`** Objects

Currently only PyGist supports `Region`s.

Instantiation

```
from region import *
rg = Region ( <keylist>)
```

Description

`Region` objects are used to specify graphing modes for some or all of the regions in a `QuadMesh` plot. As we shall show in the examples later in this section, subsets of the regions in a `QuadMesh` can be selected for plotting, and different regions can be plotted with different keyword options. The `QuadMesh` keyword `regions` is used to specify a list of `Region` objects to a `QuadMesh`. If such a list of `Region` objects is given, then only those regions on the list will be plotted, even though the `QuadMesh` may contain others.

The keywords arguments recognized are:

> **`number, boundary, boundary_type, boundary_color, inhibit, levels, filled, contours, vectors, z_scale, edges, type, color, width, label, hide, marks, marker`**

Note that there are no keywords for specifying the mesh itself. `Region`s are never plotted unless they belong to an already-defined `QuadMesh`, which has all the necessary information.

`Region`, like other 2d classes, also has the methods `set` and `new`.

Keyword Arguments

The keyword arguments for Region object instantiation are as follows:

`number` = `<positive integer>`: the number of the `Region` being specified. must correspond to one or more entries in the `ireg` array of the `QuadMesh` to which this `Region` belongs.

`boundary` = `0/1`--`0`: plot portion of mesh for the selected region; `1`: plot only the boundary of the selected region.

`boundary_type`, `boundary_color`: these matter only if `boundary` = `1`, and tell how the boundary will be plotted and what its color will be.

`inhibit` = `0/1/2`--`0`: plot both sets of mesh lines; `1`: do not plot the `(x [:, j], y [, j])` lines; `2`: do not plot the `(x [i,:], y[i,:])` lines; `3`: if `boundary` = `1`, do not plot the boundary (default 0). Only applies if `edges` = 1.

`levels` = either:

(1) optional one-dimensional sequence of floating point values. If present, a list of the values of `z` at which you want contours; or

(2) a single integer specifying the number of contours, in which case the graphics will compute the contour levels.

`filled` = `0/1`--If `1`, plot a filled mesh using the values of `z` if `contours = 0`, or plot filled contours if `contours = 1` (this option is not available at the time of writing of this manual, but will be added soon). If `z` is not present, the mesh zones will be filled with the background color, which allows plotting of a wire frame. (default value: `0`.)

`contours` = `0/1`--if `1`, contours will be plotted if `filled = 0`, and filled contours will be plotted if `filled = 1` (this option is not available at the time of writing of this manual, but will be added soon). `contours` normally defaults to `0`, but will default to `1` if `edges = 0`, `filled = 0`, and `vectors = 0` on the theory that you must want to plot something.

The user should Table 3, “filled and contours,” on page 24 to understand how these last two parameters relate.

`z_scale` = `"lin"` (default), `"log"`, or `"normal"` specifies how the contours are to be computed.

`vectors` = `0/1`--This keyword is only meaningful if the `QuadMesh` containing this `Region` has the vectors `vx` and `vy` defined. If 0, the vectors defined on this `Region` will not be plotted; if `1`, they will be. (default: `1`)

`edges`, if nonzero when `filled = 1`, draw a solid edge around each zone, as controlled by keyword `inhibit`. `ewidth` and `ecolor` may also be used.

`type`, `color`, `width`, `label`, `hide`, `marks`, `marker` as for `QuadMesh.` Remember that a `marker` specified for a contour plot represents the first of a consecutive series of markers for the contours.

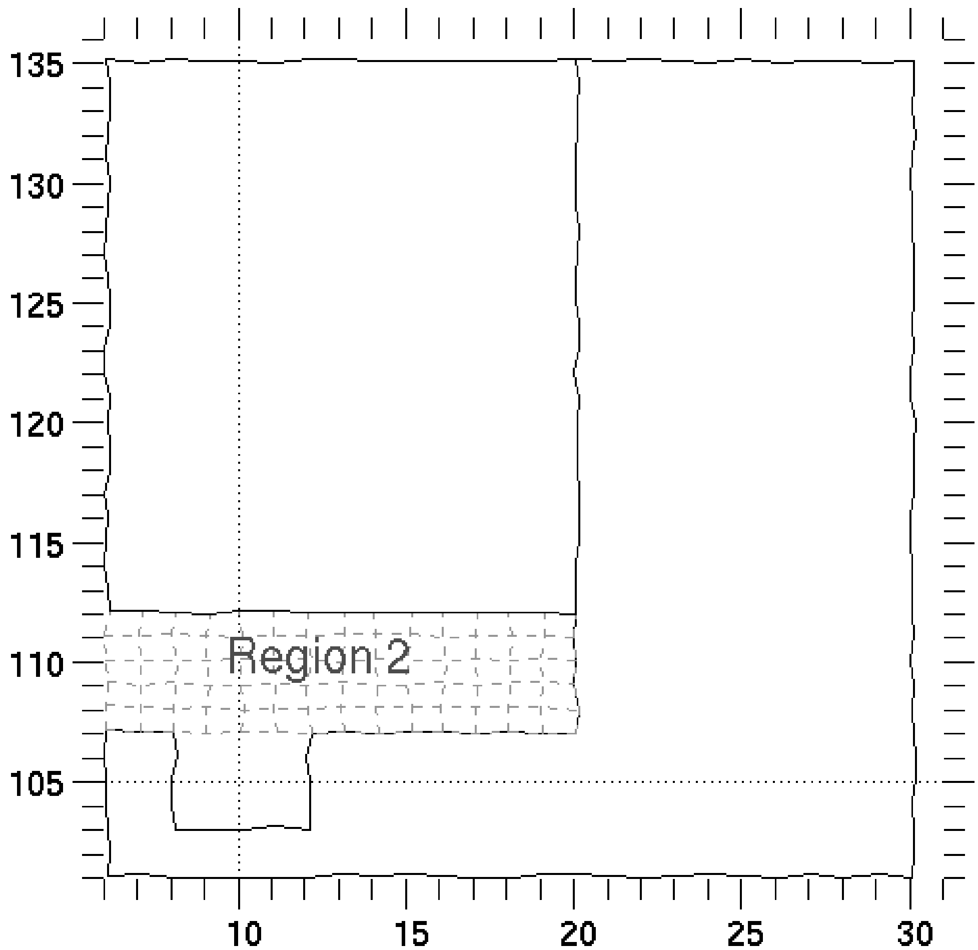
Methods `new` and `set` are as in the `Curve` and `QuadMesh` classes. Remember to beware of setting conflicting values for keywords with `set`.

Examples

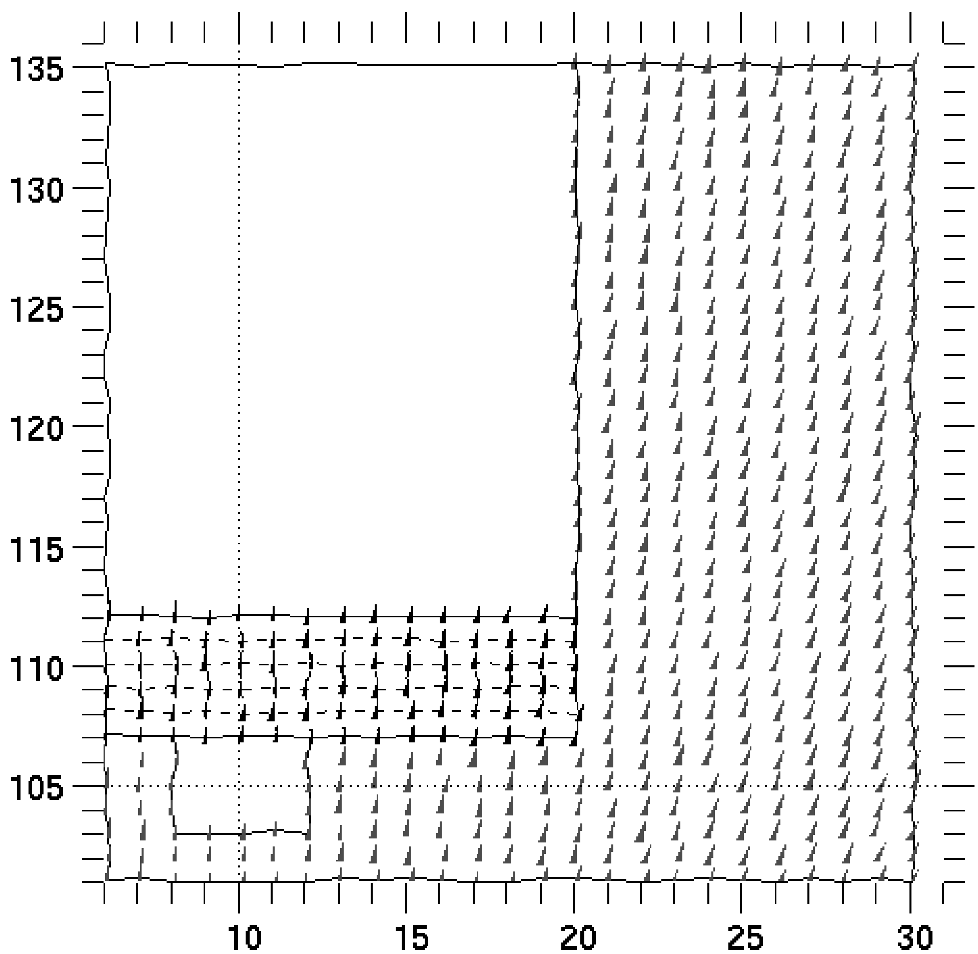
The following examples illustrate (on the same mesh as before) how you can plot the regions of the mesh in differing styles. Study the code and comments carefully, and run the examples yourself.

```
from region import *
# Region 1 will have a solid, foreground-colored boundary:
r1 = Region (number = 1, width = 1., color = "fg",
   boundary = 1)
# Region 2 will be plotted with no boundary and with its
# mesh lines colored green and dashed in appearance:
r2 = Region (number = 2, width = 1., color = "green",
   type = "dash")
# Region 3 will be plotted in the same style as Region 1:
r3 = Region (number = 3, width = 1., color = "fg",
```

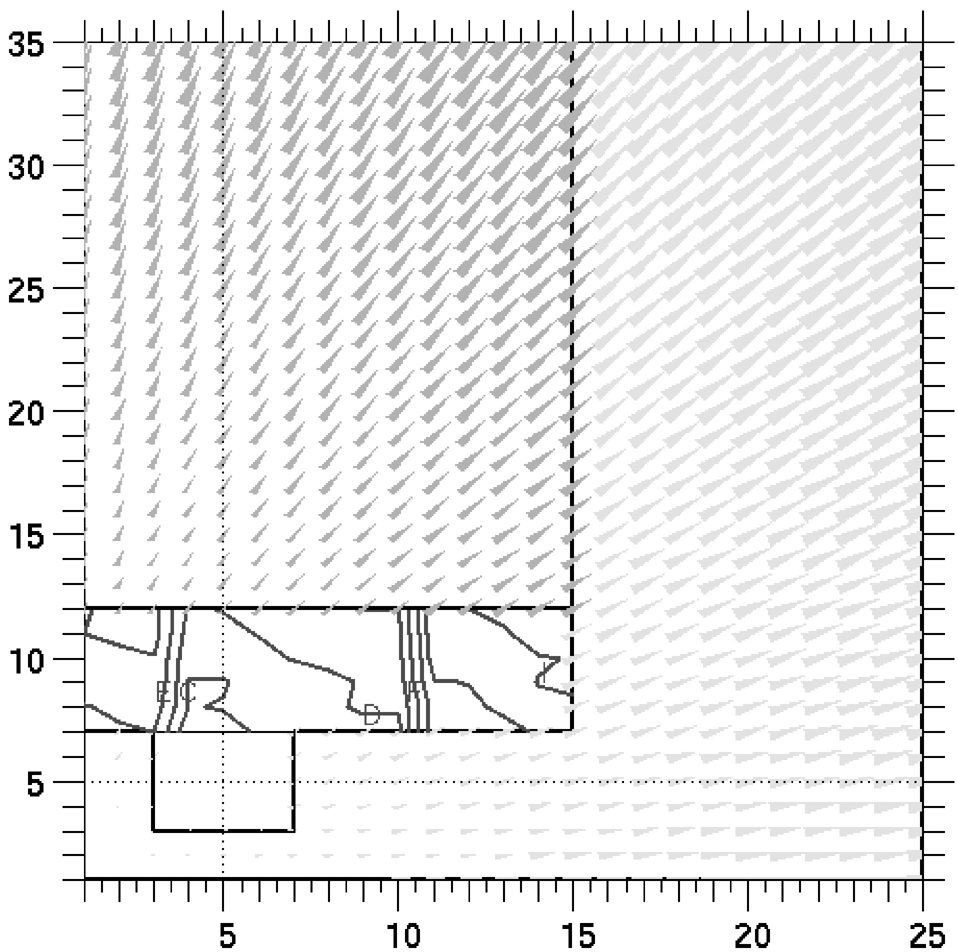
```
   boundary = 1)
# We now send the region list to the existing QuadMesh qm:
qm.set (regions = [r1, r2, r3])
# Change the graph to print Region 2 in red on top
# of Region 2, then plot the graph:
gr.change(text = "Region 2", text_pos = [0.25,0.54],
   text_size = 18, text_color = "red")
gr.plot ()
```



```
# The next plot will be a vector plot, so send vectors
# to the QuadMesh qm:
qm.set (vx = vt, vy = ut, scale = 1.)
# Plot r1's vectors in red:
r1.set (color = "red")
# Change the color of r2's mesh to foreground, and
# give it a solid boundary. Its vectors will also be
# foreground.
r2.set (color = "fg", type = "solid", boundary = 1)
# Suppress the plotting of vectors over r3
r3.set (vectors = 0)
# Erase the text and plot:
gr.change(text = "")
gr.plot ()
```



```
# Change qm back to the rectangular mesh, and change
# the vector field:
qm.set(z = z, x = xr, y = yr, vx = xr + yr/5.,
   vy = yr + xr/10., scale = .05)
# change r1 to have orange vectors:
r1.set (color = "orange", width = 3.)
# r2 will have red contours, no vectors:
r2.set (color = "red", width = 3., vectors = 0,
   contours = 1, type = "solid", levels = 20)
# r3 will have cyan colored vectors:
r3.set (color = "cyan", width = 3., vectors = 1)
gr.plot ()
```



3.5 `Polymap` Objects

Currently `Polymap` objects are only available in PyGist.

Instantiation

```
from polymap import *
pm = Polymap ( <keylist>)
```

Description

A `Polymap` is a set of arbitrary color-filled polygons. The allowed keywords are

`x, y, n, z, hide, label`

In addition, like all 2d geometric classes, `Polymap`s have the methods `set` and `new`.

Keyword Arguments

The following keyword arguments can be specified for `Polymap`s:

`x = <sequence of floating point values>`

`y = <sequence of floating point values>`

These are the coordinates of the vertices of the polygons. (The way this data is set up, vertices of adjacent polygons will be repeated.)

`n = <sequence of integer values>`

Entry `n [i]` in this array tells how many vertices polygon `i` has. Thus the first `n [0]` entries in `x` and `y` are the vertices of the first polygon, the next `n [1]` entries, of the second, etc. The sum of all the entries in `n` is the length of vectors `x` and `y`.

`z = <sequence of numerical or unsigned character values>` (this vector is the same length as `n`) tells how to color the polygons. Numbers are interpolated into a palette; the integer values of unsigned characters (Python typecode `'b'`)are used as indices into the palette.

`hide = 0/1`--(`1` to hide this part of the graph)

`label` = `<string>`--label for this part of the graph.

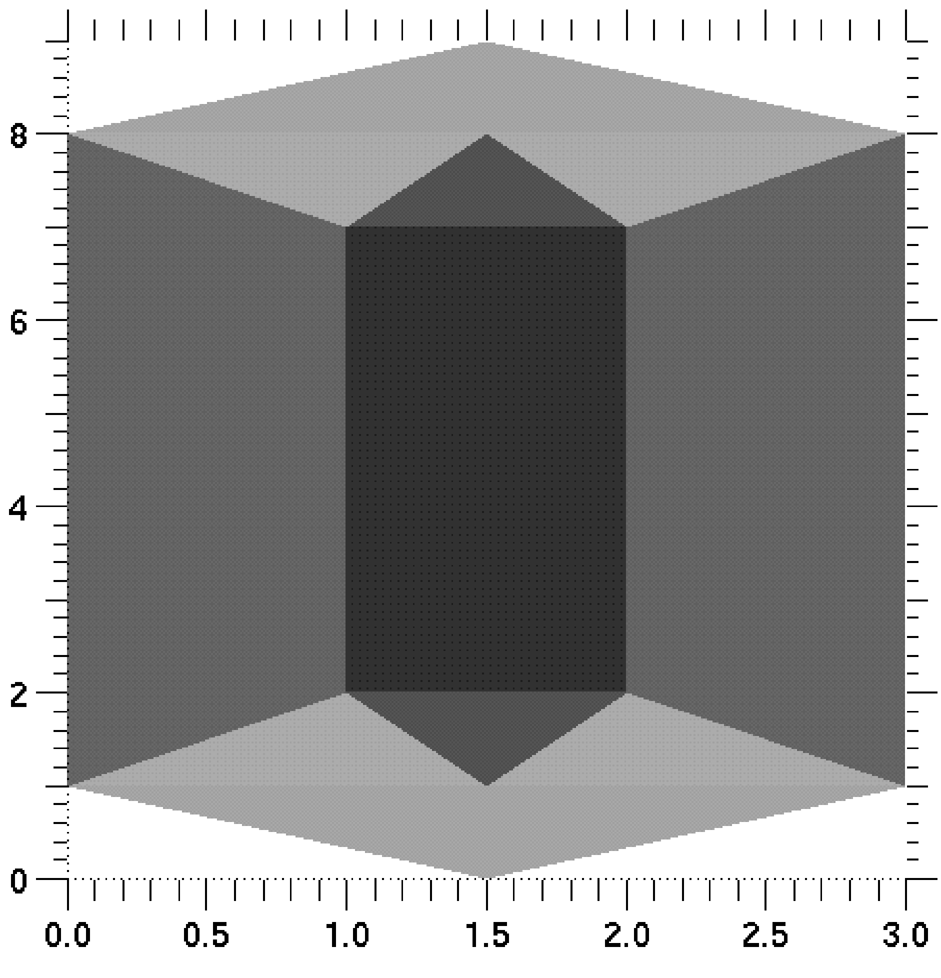
Methods `new` and `set` have the same function as in the other 2d classes.

Example

The following simple polymap example shows something like a stained glass window.

```
from polymap import *
```

```
n = array ( [4, 3, 3, 3, 3, 4, 4, 3, 3, 3, 3])
x = array ( [0., 1., 1., 0., 0., 1.5, 1., 1.5, 3., 0., 1.5,
   3., 2., 1.5, 2., 1., 2., 3., 3., 2., 1., 2., 2., 1., 2.,
   3., 1.5, 1., 2., 1.5, 1., 1.5, 0., 0., 3., 1.5])
y = [1., 2., 7., 8., 1., 1., 2., 0., 1., 1., 1., 1., 2., 1.,
   2., 2., 2., 1., 8., 7., 2., 2., 7., 7., 7., 8., 8., 7.,
   7., 8., 7., 8., 8., 8., 8., 9.])
z = array ( [2.5, 1.2, 1.5, 1.2, .5, 2.5, 2., 1.2, .5, 1.2,
   1.5])
p1 = Polymap (x = x, y = y, z = z, n = n)
g0 = Graph2d (p1, titles = "Stained glass window")
g0.plot ()
```



3.6 `CellArray` Objects

Currently, `CellArray` objects are only available in PyGist.

Instantiation

```
from cellarray import *
ca = CellArray ( <keylist>)
```

Description

A `CellArray` is a regular two dimensional rectangular mesh whose cells are color filled as specified by the keyword argument `z`. The keywords accepted by `CellArray` are:

`z, x0, y0, x1, y1, hide, label`

In addition, `CellArray` objects have the methods `new` and `set`, like all other 2d geometric objects.

Keyword Arguments

The following keyword arguments can be specified for `CellArray`s:

`z = <2d sequence of numeric or unsigned character values>`

specifies the coloring to be given to the `CellArray`. If numeric, the values are interpolated onto a palette. If unsigned character, the values are used to index into the palette. **The argument `z` is required.** If the dimensions of `z` are n_1 and n_2, then the cell array will be n_1 by n_2 cells in size.

`x0`, `y0` -- floating point scalars; if present, the coordinates of the lower left corner of the cell array. The default is `(0., 0.)`.

These coordinates are optional, but if they are present then x1, y1 must be also (see below).

`x1`, `y1` -- floating point scalars; if present, the coordinates of the upper right corner of the cell array. If these optional keywords are missing, then `x0`, `y0` must be also missing, and their default values `(1.0, 1.0)` will be used.

`hide` = `0/1` (`1` to hide this part of the graph)

`label` = `<string>` label for this part of the graph.

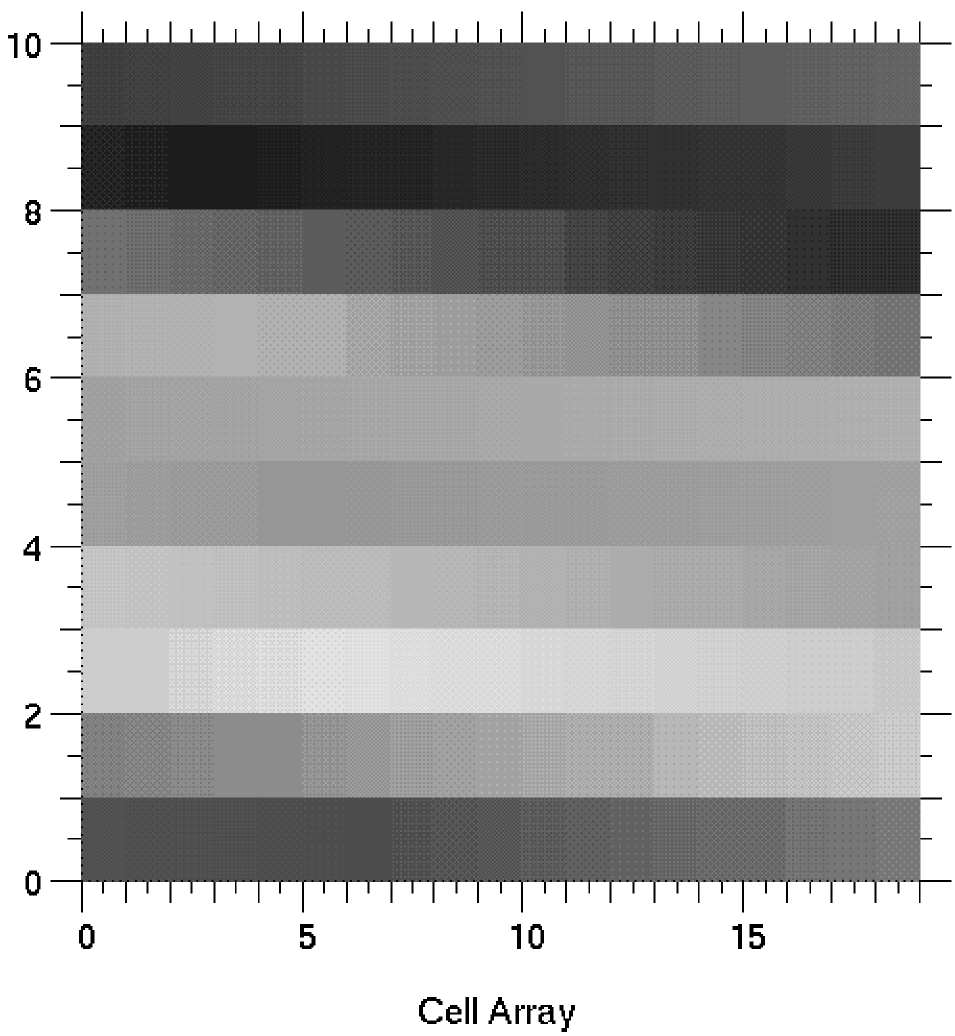
Methods `new` and `set` are as in the other 2d classes.

Example

The following simple example creates a 10 by 19 `CellArray` object and plots it.

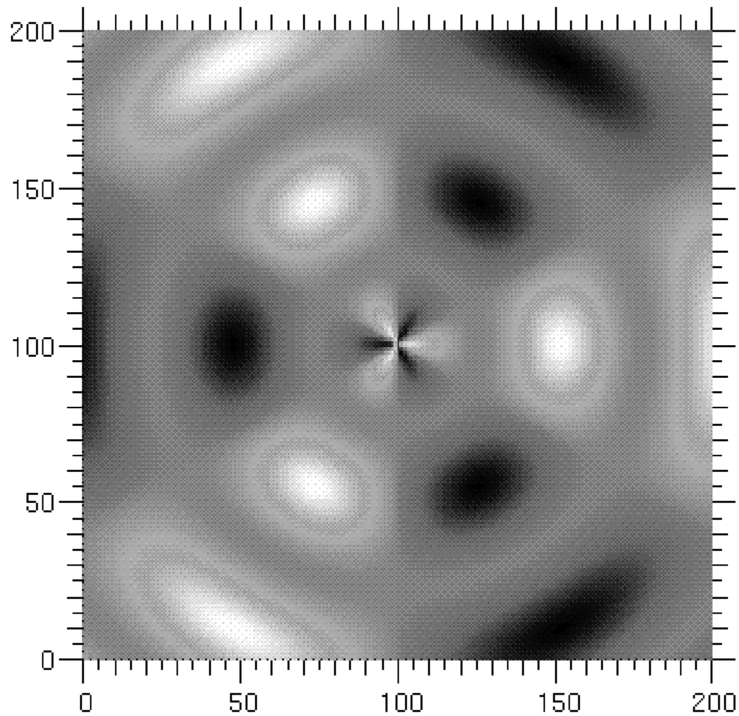
```
from cellarray import *
```

```
nx= 10
ny= 19
# 'b' (byte) is the typecode used by Python for
# unsigned character:
ndx = reshape (arange (nx * ny, typecode = 'b'), (nx, ny))
# Instantiate a CellArray object:
cla = CellArray ( z = ndx )
# Create a Graph2d object containing it:
gca = Graph2d ( cla , titles = "Cell Array",
   axis_scales = "linlin" ) # (axis scales is redundant)
# Plot it:
gca.plot ( )
```



Another (and more interesting) example of a `CellArray` is given below. First we show the functions `mag` and `a3`, which are used to calculate the data plotted.

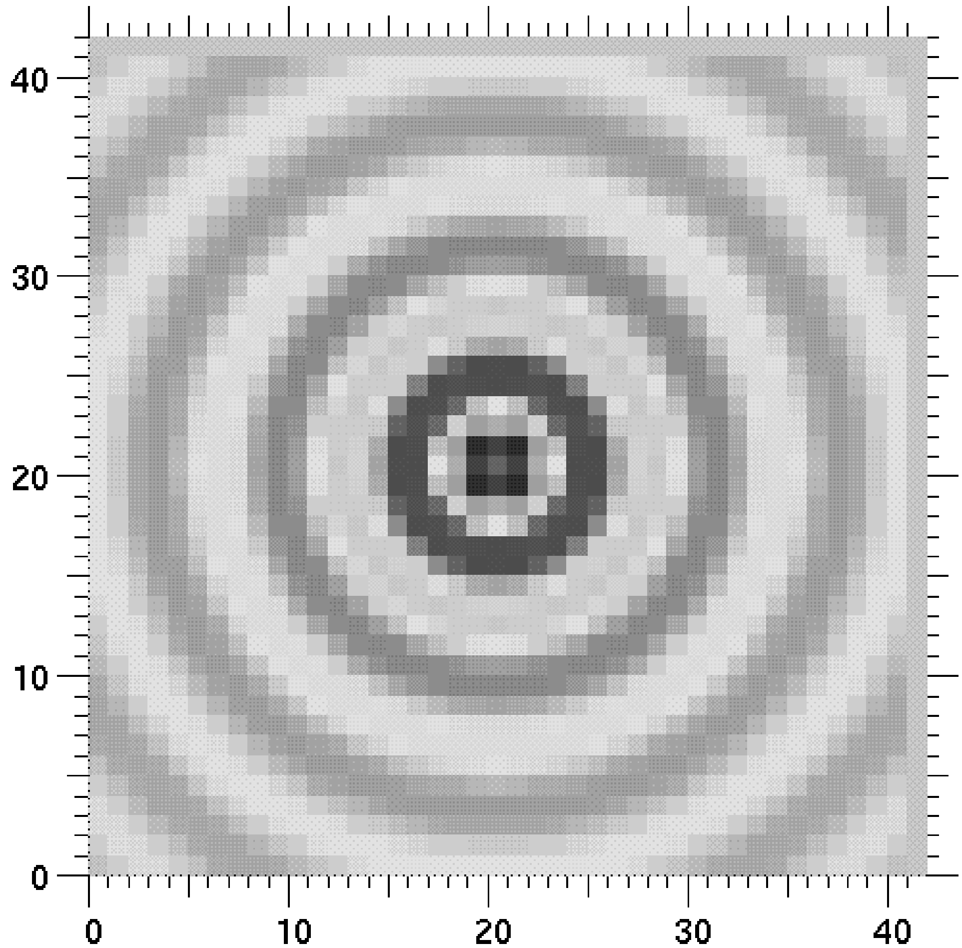
```
def mag ( *args ) :
   r = 0
   for i in range (len (args)) :
      r = r + args[i] * args[i]
   return sqrt (r)
def a3 (lb, ub, n) :
   return reshape (array(n*span(lb,ub,n), Float), (n,n))
# The following computation produces the plot
x=a3(-6,6,200)
y=transpose (x)
r=mag(y,x)
theta=arctan2(y,x)
funky=cos(r)**2*cos(3*theta)
from cellarray import *
c1 = CellArray(z=funky)
g1 = Graph2d (c1, color_card = "earth.gp",
           titles ="Cell array, three cycles in theta,r",
           axis_limits = "defaults")
g1.plot()
```



Cell array, three cycles in theta,r

A final example of the CellArray is the sombrero function.

```
nz = 20
x = arange (-nz, nz+1, typecode = Float )
y = x
z = zeros ((2*nz + 2, 2*nz + 2), Float)
for i in range ( len (x) ) :
   for j in range ( len (y)) :
      r = sqrt ( x [i] * x[i] + y [j] * y [j] ) + 1.e-12
      z [i, j] = sin (r) / r
# cell array plot
cla = CellArray ( z = z)
gca = Graph2d ( cla , titles = "Sombrero Function",
      color_card = "rainbow.gp", axis_limits="defaults")
gca.plot ( )
```



Sombrero Function

CHAPTER 4: Three-Dimensional Geometric Objects

Three dimensional objects are instantiated similarly to two dimensional objects, and have many similar sounding keywords and methods. However, concatenating or linking multiple 3d objects on the same graph--sometimes with different 3d options, color cards, etc.--is more complicated.

4.1 `Surface` Objects

Instantiation

```
from surface import *
sf = Surface ( <keylist>)
```

Description

A `Surface` represents a two-dimensional object in three-dimensional space. It may be purely geometric, or there may be a function defined on the `Surface` which needs representation too, in which case we have essentially a four dimensional object. The `Surface` itself is projected on the plane of the graph from some angle; its third dimension may be represented by shading (as if it is shiny and there is a light source from some direction), by superimposing a wire mesh on the `Surface`, or by coloring it according to height (when there isn't a function on it which needs representation). A function defined on the `Surface` may have its values denoted by coloring the `Surface` or by drawing contours on the `Surface`.

The following keyword arguments may be used in the instantiation of a `Surface`:

`z, x, y, c, color_card, opt_3d, mesh_type, mask, z_c_switch, z_contours_scale, c_contours_scale, z_contours_array, c_contours_array, number_of_z_contours, number_of_c_contours`

In addition, `Surface`s have two methods `new` (for clearing out a used `Surface` to an empty shell and redefining its geometry) and `set` (for changing the value of arbitrary keywords). These methods work exactly as they do for two dimensional objects.

Keyword Arguments

The following keyword arguments can be specified for a `Surface` object. Note that not all keywords are available in both PyGist and PyNarcisse. Generally, using an inapproriate keyword will not cause an error; it will be ignored or else the graphics engine will make a clever guess.

`z` = `<value>` (required). `z` is a two dimensional array. If `x` and `y` are not specified, then `z` will be graphed on equally spaced mesh points.

`x` = `<value>`, `y` = `<value>` (if one is present, then so must the other be.) If `c` (below) is not present, this represents a 3d plot. Either `x` and `y` have dimensions matching `z`, or else they are one-dimensional and `x`'s length matches the first dimension of `z`, and `y`'s length matches the second.

`c` = `<value>` If present, then the Surface will be colored according to the values of `c`. (This is a so-called four-dimensional graph.) `c` must have the same dimensions as `z`.

`color_card` = `<value>`

specifies which color card (another name for palette) you wish to use, e. g., `"rainbowhls"` (the default), `"random"`, etc. Although a characteristic of a `Graph2d`, it can be a `Surface` characteristic since `'link'`ed surfaces can have different color cards (valid for Narcisse only). Following is a list of color cards available in Narcisse and Gist, with a brief description of each. The graphics interface is intelligent enough to make a good guess if you specify a Gist color card to Narcisse or vice versa; and if there is no near equivalent, it will simply assign the default color card.

First we list the Narcisse Color Cards. Narcisse color cards contain 64 colors. The first ten, in order, are always bg, fg, blue, green, yellow, orange, red, purple, black, and white. The other 54 are described roughly in the table, starting with the lowest index.

TABLE 4 Narcisse Color cards

absolute	from black to light grey in the middle of the palette, then back down to dark grey at the end.
binary	repeatedly runs through the colors light blue, blue, cyan, green, purple, red, yellow, and orange.
bluegreen	continuously shading from light green to deep blue.
default	grey scale, from black at the low end to white at the top
negative	first half is black; second is grey scale from white to dark grey
positive	first half shades from dark grey to white; second is black
rainbow	shades through the rainbow colors from purple at the low end through red at the high.
rainbowhls	low end is blue, shades through rainbow colors to red, then purple.
random	different every time you use it.
redblue	shades from blue at the low end to red at the high end.
redgreen	shades from light green at the low end to red at the high end.
shifted	shades from medium grey at the low end to white in the middle, then from black in the middle to medium grey at the high end.

Next we describe the Gist color cards. Gist color cards contain 200 colors. There are no reserved spots at the start for special colors.

TABLE 5 Gist Color Cards

earth.gp	black, dark to light blues and then greens, all brown-tinged, tan, beige, some grey, pink, ivory, and white at the top.
stern.gp	black, grey, red, magenta, purple, lightening into blue, bluegreen, green, ivory, light grey, white.
rainbow.gp	red through purple, in the normal rainbow order.
heat.gp	very dark red, lightens up through shades of red to orange, yellow, ivory, and white.
gray.gp	grey scale running from black at the low end to white at the high.
yarg.gp	same, but white at the bottom to black at the top.

`opt_3d` = `<value>` where `<value>` is a string or a sequence of strings giving the 3d or 4d surface characteristics. A surface is colored by height in `z` if a 3d option is specified, and by the value of the function `c` if a 4d option is specified. With a wire grid option, the grid is colored (Narcisse only); with a flat option, the quadrilaterals set off by grid points are colored; with a smooth option, the surface itself is colored by height (filled contours); and with an iso option, the contour lines are colored (Narcisse only). flat and iso options may be used together in any combination. wire grid options are independent of the other options. Legal arguments for `opt_3d` are:

`'wm'`--monochrome wire grid (the default); `'w3'` and `'w4'`--3d and 4d coloring of wire grid. `'w3'` and `'w4'` are not currently available in Gist.

`'f3'` and `'f4'`--flat 3d and 4d coloring options.

`'i3'` and `'i4'`--3d and 4d isoline (contour line) options. Colored isolines are not currently available in Gist.

`'s3'` and `'s4'`--3d and 4d smooth coloring (filled contour) options.

`mesh_type` = `<string>` in one of the wire modes, tells what form the wire grid takes: `"x"`: x lines only; `"y"`: y lines only; `"xy"`: both x lines and y lines (the default). Only the latter is available in Gist.

`mask` = `<string>`: specifies whether hidden lines will be eliminated, and if so, how complex the algorithm that will be used to determine what is hidden. `"none"` : transparent wire grid (the default); `"min"`: simple masking; `"max"` : better masking; `"sort"`: slowest but most sophisticated. Only `"none"` and `"sort"` are available in Gist.

`z_c_switch` = `0` or `1` : set to `1` means switch `z` and `c` in the plot.

`z_contours_scale`, `c_contours_scale` = `"lin"` or `"log"`.

`z_contours_array`, `c_contours_array` = actual array of numbers to use for contours, if you don't want them computed automatically.

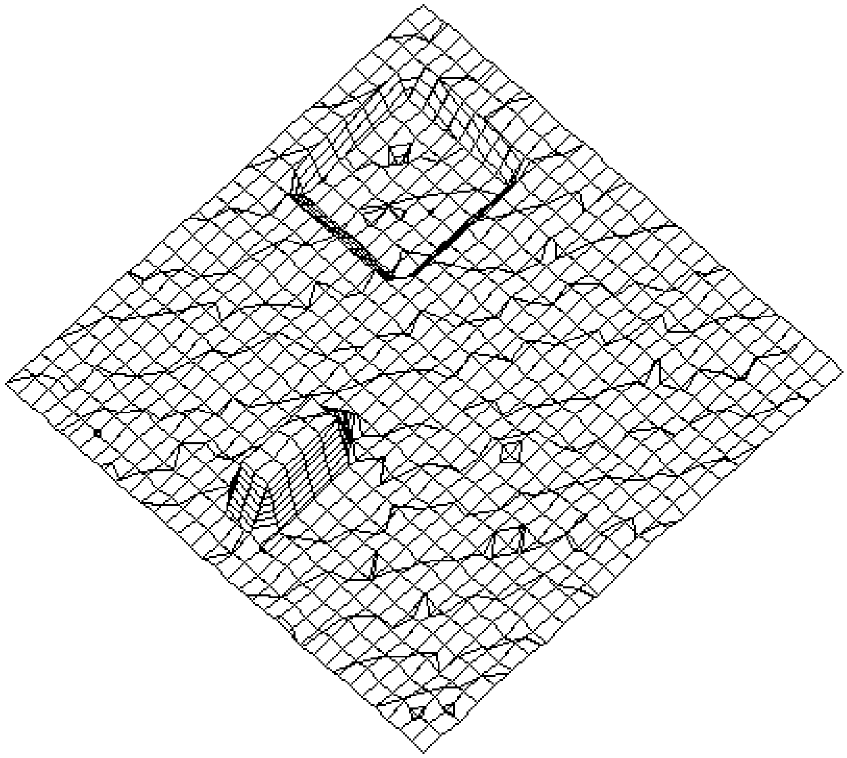
`number_of_z_contours`, `number_of_c_contours` = `<integer>` specifies how many contours to use; they will be computed automatically depending on the data.

Examples

The following set of computations defines a surface and functional values on the surface, which will be used in the subsequent plots. Note that this is very similar to the `QuadMesh` example. (See “Examples” on page 25..) However, now we shall see the surface in three-dimensional space, with contours and contour lines. We will do many plots of this surface, in order to show the many available options.

```
s = 1000.
kmax = 25
lmax = 35
xr = multiply.outer (arange (1, kmax + 1, typecode = Float),
   ones (lmax, Float))
yr = multiply.outer (ones (kmax, Float), arange (1, lmax + 1,
   typecode = Float))
zt = 5. + xr + .2 * random_sample (kmax, lmax)
rt = 100. + yr + .2 * random_sample (kmax, lmax)
z = s * (rt + zt)
z = z + .02 * z * random_sample (kmax, lmax)
ut = rt / sqrt (rt ** 2 + zt ** 2)
vt = zt / sqrt (rt ** 2 + zt ** 2)
ireg =  multiply.outer ( ones (kmax), ones (lmax))
ireg [0:1, 0:lmax] = 0
ireg [0:kmax, 0:1] = 0
ireg [1:15, 7:12] = 2
ireg [1:15, 12:lmax] = 3
ireg [3:7, 3:7] = 0

freg = ireg.astype (Float) + .2 * (1. -
   random_sample (kmax, lmax))
z [3:10, 3:12] = z [3:10, 3:12] * .9
z [5, 5] = z [5, 5] * .9
z [17:22, 15:18] = z [17:22, 15:18] * 1.2
z [16, 16] = z [16, 16] * 1.1
s1 = Surface (z = z, mask = "max", opt_3d = ["wm", "i3"])
g1 = Graph3d ( s1 , titles = "Surface with contour lines",
               xyequal = 1.,
               theta = 45., phi = 10., roll = 0.)
g1.plot ( )
```

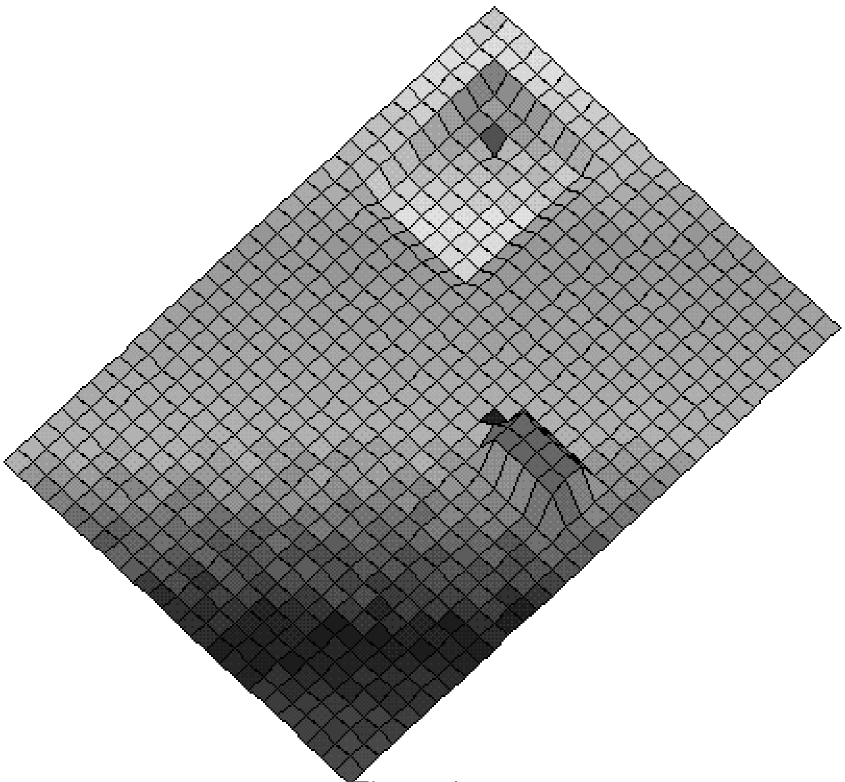
The plot appears on the next page.



Surface with contour lines

In the following plot, we change the 3d options to `"wm"` (wire mode, i. e., mesh lines are plotted) and `"f3"` (flat 3d, meaning cells are colored according to their average height).

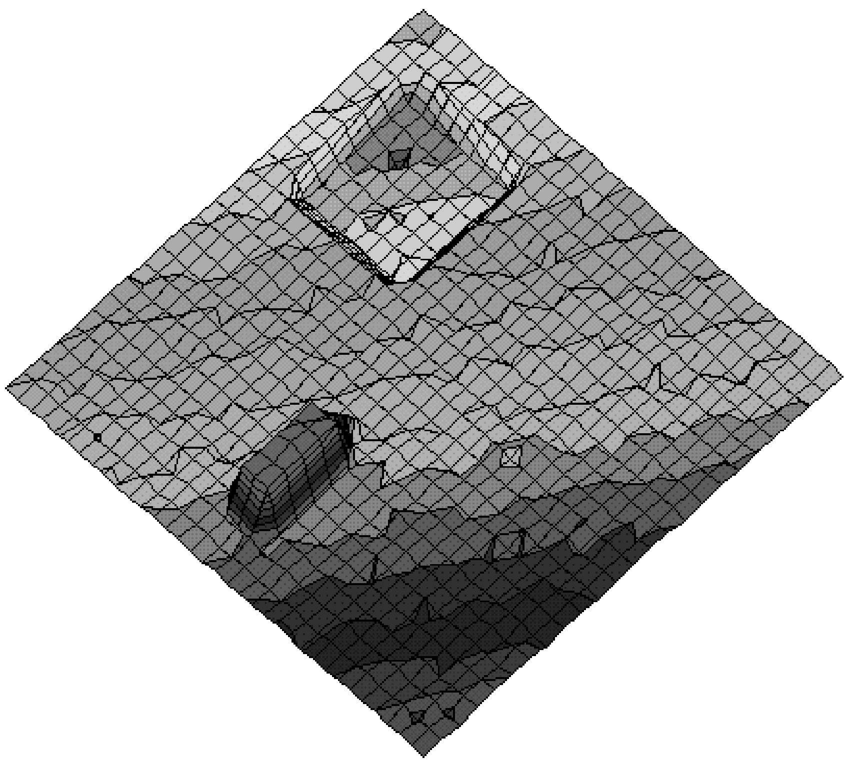
```
s1.set (opt_3d = ["wm", "f3"])
g1.change (titles = "Flat mode")
g1.plot ()
```



Flat mode

Now let us leave "wm" set, and switch to "s3" (smooth 3d, i. e., the surface is drawn with contours filled with color according to height.

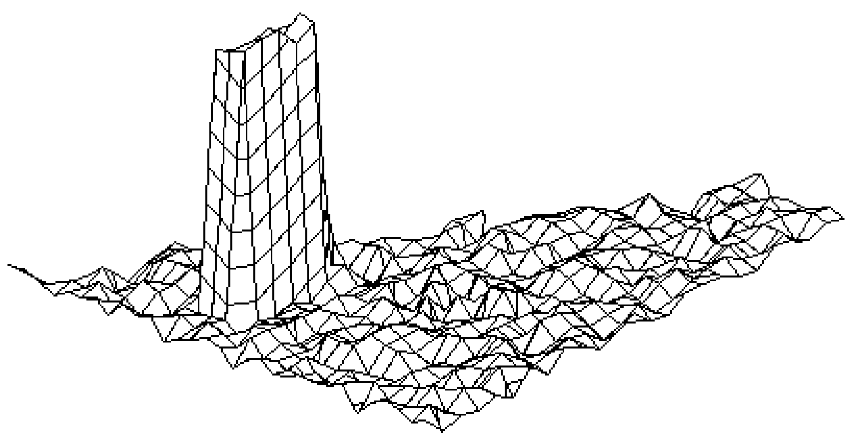
```
s1.set (opt_3d = ["wm", "s3"])
g1.change (titles = "Smooth mode")
g1.plot ()
```



Smooth mode

The next plot illustrates how we can use the `axis_limits` keyword to trim off a portion of the figure, when plotting contours. If both axis limits are given as 0.0, then PyGist takes this as a signal to compute limits based on the data. if either or both limits are nonzero, then PyGist will not display parts of the graph whose `z` values fall outside the limits. In this particular example, we have set the minimum `z` value to 0.0, so the part of the surface below the xy plane will be suppressed. The same thing may be done with 4d plots, by specifying a fourth set of limits, which apply to the variable being plotted. The scale is exaggerated in the `z` direction; a larger `y_factor` might ameliorate this problem.

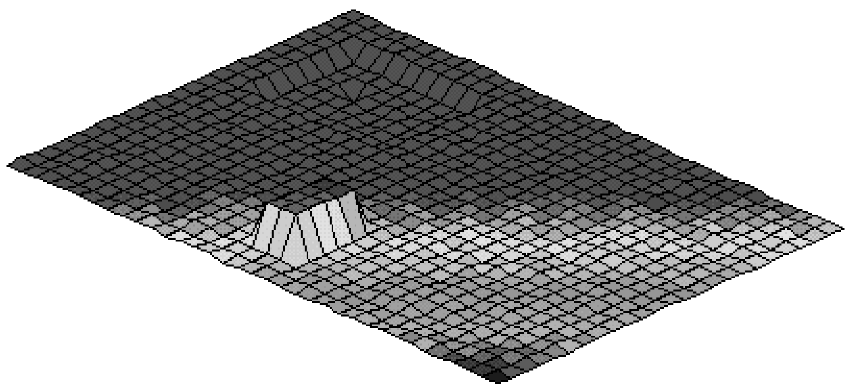
```
s1.new (x = xr, y = yr, z = z - z [kmax/2, lmax/2],
   mask = "max", opt_3d = ["wm" , "i3"])
g1.change ( titles = "Part of surface above xy plane",
   phi = 30., y_factor = 2.0,
   axis_limits = [[0., 0.],[0., 0.], [0., 100000.]])
```



Part of surface above xy plane

If we try to do the same plot in flat mode, the `z` axis limits do not work as advertised. Plots can be trimmed as above only in one of the contour plotting modes: `"i3"`, `"i4"`, `"s3"`, or `"s4"`. We are going to use the same `Surface` and `Graph3d` objects, changing only the `Graph3d`'s title, but therefore leaving the `z` axis limits unchanged. Thus we have the following:

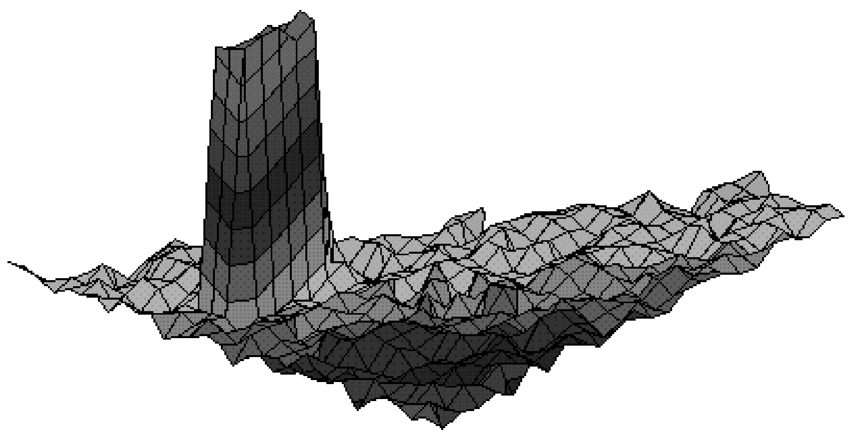
```
s1.set (opt_3d = ["wm", "f3"])
g1.change (titles = "Flat mode")
g1.plot ()
```



Flat mode

Next we go to smooth (filled contour) mode. The contours are colored based on the maximum and minimum `z` values taken over the whole surface, not on the ones plotted.

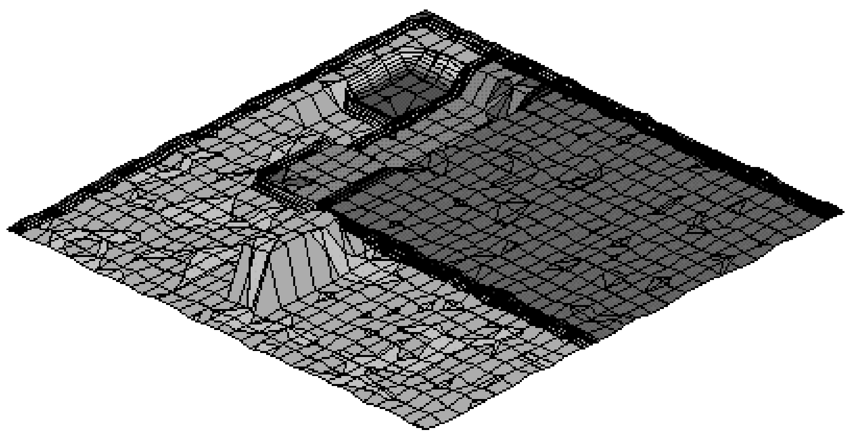
```
s1.set (opt_3d = ["wm", "s3"])
g1.change (titles = "Smooth mode")
g1.plot ()
```



Smooth mode

Below is a plot of the same surface, but this time it is a so-called 4d plot, meaning that contours are drawn and filled according to the value of a variable on the mesh, rather than its height. In this case the variable is `freg`, defined a few pages previously (see page 50). Note that all axis limits are set back to defaults.

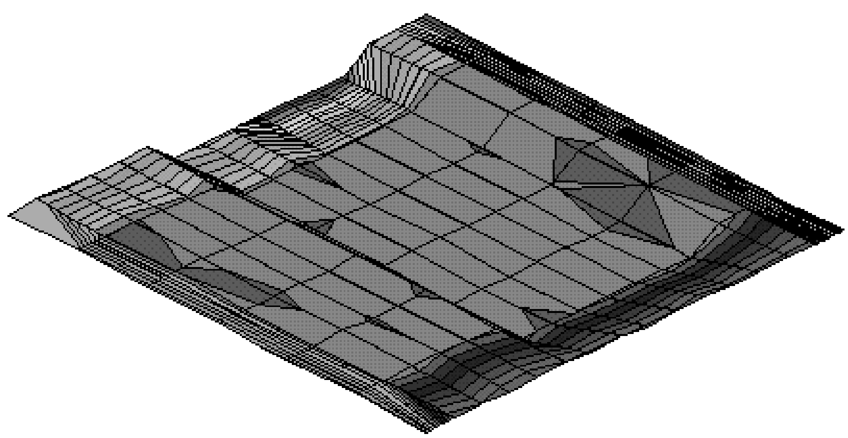
```
s1.set (z = z, c = freg, opt_3d = ["wm", "s4"])
g1.change ( titles = "Surface colored by mesh values",
   phi = 20., xyequal = 1,
   axis_limits = [[0., 0.], [0., 0.], [0., 0.], [0., 0.]])
g1.plot ( )
```



Surface colored by mesh values

Here is an illustration of a plot of a single region of the previous plot, namely region 2.

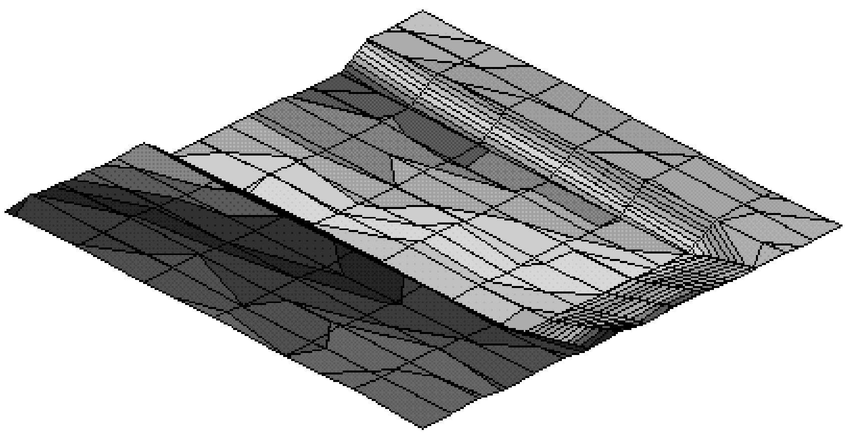
```
xr1 = xr [0:16, 6:13]
yr1 = yr [0:16, 6:13]
z1 = z [0:16, 6:13]
zs1 = freg [0:16, 6:13]
s1.set (x = xr1, y = yr1, z = z1, c = zs1)
g1.change ( titles = "Region 2 colored by mesh values",
   phi = 10.)
g1.plot ( )
```



Region 2 colored by mesh values

Here is the same geometric object plotted with height contours:

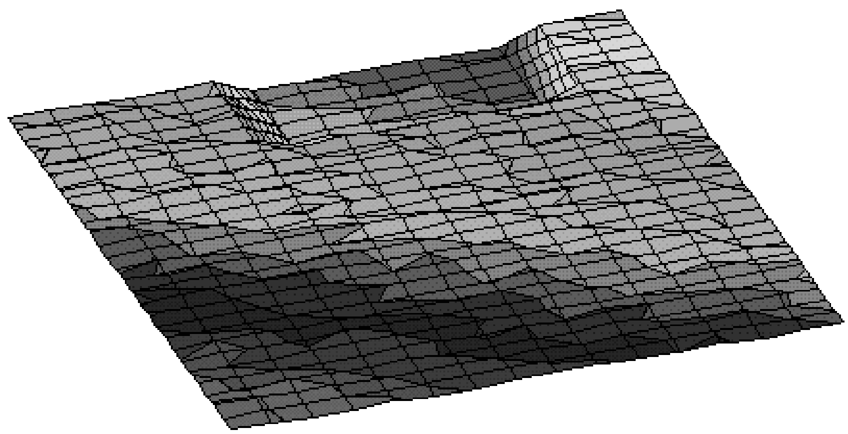
```
s1.new (x = xr1, y = yr1, z = z1, opt_3d = ["wm", "s3"],
   mask = "max")
g1.change ( titles = "Region 2 with mesh and contours",
   phi = 10.)
g1.plot ( )
```



Region 2 with mesh and contours

Our final example is a plot of regions 2 and 3, with height contours.

```
zs1 = z [0:16, 6:lmax - 1]
s1.new (z = zs1, opt_3d = ["wm", "s3"], mask = "max")
g1.change ( titles = "Regions 2 and 3, mesh and contours",
   theta = 70., phi = 10., roll = 0.)
g1.plot ( )
```



Regions 2 and 3, mesh and contours

4.2 `Mesh3d` Objects

`Surface` and `Mesh3d` objects differ in that a `Surface` is really the two-dimensional boundary of a three-dimensional object (if it is closed) or the topological equivalent of a plane (if not closed). In other words, it is a two dimensional object which has been twisted, bent, or deformed through a third dimension. In contrast, a `Mesh3d` consists of a partition of a three-dimensional object into smaller three-dimensional objects called cells. With a `Surface`, only what is happening on the two dimensions of the `Surface` itself is of interest to us, whereas with a `Mesh3d`, what is happening inside the three dimensional object is of interest.[1] This leads to a problem in visualization, because how can you see the inside of an object? The answer, usually, is that cells have to be stripped away to view what is going on underneath them. Alternatively, we can take sections through the `Mesh3d`, the most common of these being plane slices and isosurface slices. (Isosurfaces are slices upon which some specified function is equal to some constant.)

At any rate, a `Mesh3d` is a generalization of a `Surface`, and in fact a `Mesh3d` is a derived class of a `Surface`.

4.2.1 Structured vs. Nonstructured Meshes

There are two kinds of `Mesh3d` objects:

- A structured `Mesh3d` consists of rectangular hexahedra with sides parallel to the axes, and is specified by three one dimensional vectors of coordinates, `x`, `y`, and `z`. Associated with each `Mesh3d` point is a component of a three-dimensional array of data called `c`.
- A nonstructured `Mesh3d` in principle could consist of cells of arbitrary shape, but we limit ourselves to the four standard shapes: hexahedra, tetrahedra, pyramids (with square bases) and prisms (with triangular bases). A nonstructured `Mesh3d` is specified by one-dimensional arrays of `x`, `y`, and `z` coordinates, the i^{th} component of `x`, `y`, and `z` being the coordinates of the i^{th} node in the mesh. There is an associated one-dimensional array `c` of data, one value for each node point. Naturally the points alone are not sufficient to specify the connectivity of the mesh. Hence we need configuration information which, for each cell in the mesh, tells which nodes belong to the cell. The `Mesh3d` class accepts two formats, the Narcisse format peculiar to itself which we will not go into here (See “The Narcisse Format and Keywords” on page 68.), and the AVS format. In the case of the AVS format, for each shape of cell in the `Mesh3d`, the user must supply a count of the cells, and a one-dimensional array of integer node numbers for each of the cells, in a standard order, as follows:

  tetrahedra--apex, then base nodes, in inward normal order.

  pyramids--apex, then base nodes, in inward normal order.

  prisms--one triangular face, in outwards normal order, then the corresponding nodes of the opposite face, in inward normal order.

  hexahedra--one face, in outwards normal order, then the corresponding nodes of the opposite face, in inward normal order.

All `Mesh3d` objects are instantiated as described below; the keyword parameters are how PyGist distingushes what kind of `Mesh3d` it is.

Instantiation

1. Normally there is a function defined on the mesh--e. g., a physical quantity such as pressure, density, or velocity--that we want to visualize. These function values really add a fourth dimension to the plot.

```
from mesh3d import *
m3 = Mesh3d ( <keylist>)
```

Description

The list of keywords recognized by all types of `Mesh3d` objects are as follows:

`color_card, opt_3d, mesh_type, mask, z_c_switch, z_contours_scale, c_contours_scale, z_contours_array, c_contours_array, number_of_z_contours, number_of_c_contours`

Since a `Mesh3d` is a `Surface`, it also accepts all the keywords that define a `Surface` object, ignoring any that might not be sensible (Section on page 47).

In addition, the `Mesh3d` class has two methods `set` (inherited from `Surface`) and `new` (not inherited, but having exactly the same functionality).

Keyword Arguments

The following keyword arguments can be specified for a `Mesh3d` object. Note that not all keywords are available in both PyGist and PyNarcisse. Generally, using an inapproriate keyword will not cause an error; it will be ignored or else the graphics engine will make a clever guess.

`color_card = <value>` specifies which color card (another name for palette) you wish to use, e. g., `"rainbowhls"` (the default), `"random"`, etc. Although a characteristic of a `Graph2d`, it can be a `Surface` characteristic since `'link'`ed surfaces can have different color cards (valid for Narcisse only). For a full description of available color cards, see “color_card = <value>” on page 48. The graphics interface is intelligent enough to make a good guess if you specify a Gist color card to Narcisse or vice versa; and if there is no near equivalent, it will simply assign the default color card.

`opt_3d = <value>` where `<value>` is a string or a sequence of strings giving the 3d or 4d surface characteristics. A surface is colored by height in `z` if a 3d option is specified, and by the value of the function `c` if a 4d option is specified. With a wire grid option, the grid is colored; with a flat option, the cells set off by grid lines are colored; with a smooth option, the surface itself is colored by height; and with an iso option, the contour lines are colored. Flat and iso options may be used together in any combination. Wire grid options are independent of the other options. Legal arguments for `opt_3d` are:

- `'wm'`--monochrome wire grid (the default); `'w3'` and `'w4'`--3d and 4d coloring of wire grid. The latter two are not currently available in Gist.
- `'f3'` and `'f4'`--flat 3d and 4d coloring options.
- `'i3'` and `'i4'`--3d and 4d isoline (contour line) options. Colored isolines are currently not available in Gist.
- `'s3'` and `'s4'`--3d and 4d smooth coloring options (filled contours).

`mesh_type` = `<string>` in one of the wire modes, tells what form the wire grid takes: `"x"`: x lines only; `"y"`: y lines only; `"xy"`: both x lines and y lines (the default). Gist currently supports only the default.

`mask` = `<string>`: specifies whether hidden lines will be eliminated, and if so, how complex the algorithm that will be used to determine what is hidden. Allowed values are `"none"` : see-through wire mesh (the default); `"min"`: simple masking; `"max"` : better masking; `"sort"`: slowest but most sophisticated. Gist currently supports only `"none"` and `"sort"`; spefications of `"min"` and `"max"` are equivalent to `"sort"`.

`z_c_switch` = `0` or `1`: set to `1` means switch `z` and `c` in the plot.

`z_contours_scale`, `c_contours_scale` = `"lin"` or `"log"`.

`z_contours_array`, `c_contours_array` = actual array of numbers to use for contours, if you don't want them computed automatically.

`number_of_z_contours`, `number_of_c_contours` = `<integer>` specifies how many contours to use; they will be computed automatically based on the data.

4.2.2 Regular (or Structured) Meshes

Instantiation

```
from mesh3d import *
m3 = Mesh3d ( <keylist>)
```

Where *`<keylist>`* contains keywords peculiar to regular meshes.

Description

A structured `Mesh3d` consists of rectangular hexahedra with sides parallel to the axes, and is specified by three arrays of coordinates, `x`, `y`, and `z`. Associated with each `Mesh3d` point is a component of a three-dimensional array of data called `c`. Thus the keywords uniquely associated with structured (or regular) `Mesh3d`s are:

`x, y, z, c`

Keyword Arguments

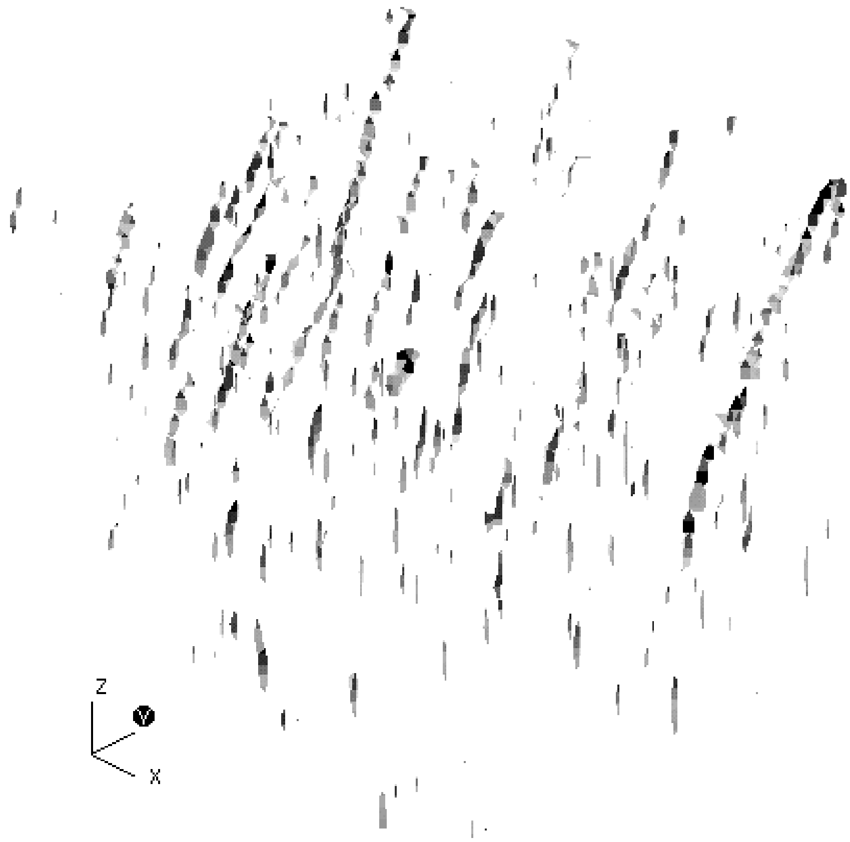
`x` = `<values>` , `y` = `<values>` , `z` = `<values>` To establish notation, assume that the mesh is `k` by `l` by `m` (i. e., there are `k` nodes in the `x` direction, `l` nodes in the `y` direction, and `m` nodes in the `z` direction.) Then there are three options for these keywords: (1) `x`, `y`, and `z` equally spaced: `x` is a vector consisting of the three integers `k - 1`, `l - 1`, `m - 1` (the cell dimensions), `y` is a vector of three `Float`s giving `dx`, `dy`, `dz` (the increments in each direction), and `z` is an array of three `Float`s giving `x0`, `y0`, `z0` (the starting values of `x`, `y`, and `z`); (2) `x`, `y`, and `z` not equally spaced: `x`, `y`, and `z` are one dimensional arrays of type `Float` specifying a `k` by `l` by `m` mesh (`k` = `len (x)`, `l` = `len (y)`, `m` = `len (z)`; or (3) `x`, `y`, and `z` are each `k` by `l` by `m`, specifying a completely general hexahedral mesh.

`c = <values>` , a three-dimensional array dimensioned `k` by `l` by `m`, whose `[i, j, k]` element gives the associated data value at the `(i, j, k)`th point of the mesh. `c` may also be one less in each direction, giving a cell-centered quantity. `c` may also be a list of such arrays, when isosurfaces of more than one function are to be plotted.

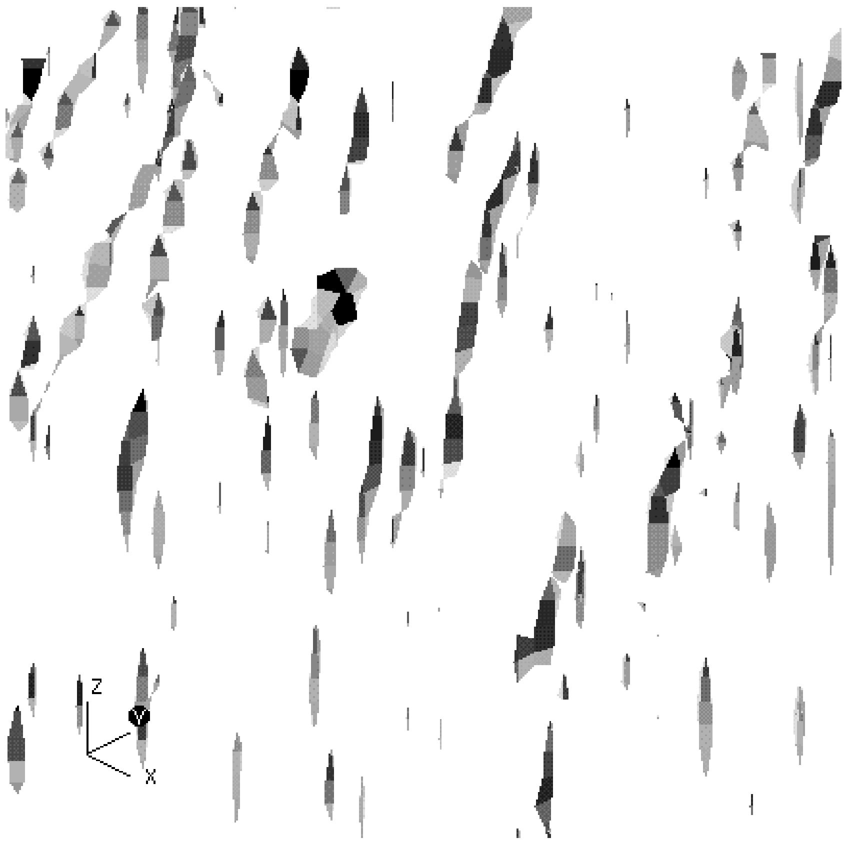
Examples

A pdb file named `berts.py` contains temperature data for filamentary flow in a plasma on a regular mesh. In this example we illustrate how to plot an isosurface slice trhrough the mesh, illustrating the filaments at a constant temperature. In order to get the isosurface slice, we use the function sslice, which is described later in the chapter (see 4.4 “Slice objects” on page 73). In this plot, we use the rainbow palette to shade the surface as if a light source were shining from behind and slightly to the right of the viewer. Polygons facing or nearly facing the viewer will be at the blue-violet end of the spectrum, and closer to the red end the closer they get to facing perpendicular to the line of sight. The code to produce this plot is as follows:

```
f = PR ('./berts_plot')
x = -80.0 + arange (64, typecode = Float) * 2.5
y = -80.0 + arange (64, typecode = Float) * 2.5
z = arange (50, typecode = Float) * 10.
c = f.c
m3 = Mesh3d (x = x, y = y, z = z, c = transpose (c))
s3 = sslice (m3, 6.5, opt_3d = ["none"])
g3 = Graph3d (s3, color_card = "rainbow.gp", gnomon = 1,
  xyequal = 1, diffuse = .2, specular = 1)
g3.plot ()
```



Note the use of `opt_3d = "none"`. Isosurfaces are shaded, so we use none of the usual 3d options. The plot looks a bit chaotic; it is possible that the PyGist sorting algorithm was a bit puzzled by the complexity of this plot. At any rate, if the user is interested in getting a closer look at this plot (or any PyGraph plot), place the cursor within the window and click the left mouse button a couple of times. Doing so causes the graph to zoom in, and you get something like the following:



You can zoom back out by clicking the third mouse button. To shift the plot around the window, click and drag with the middle button.

4.2.3 Irregular (Unstructured) Meshes

Instantiation

```
from mesh3d import *
m3 = Mesh3d ( <keylist>)
```

Where *<keylist>* contains keywords peculiar to irregular meshes.

Description

A nonstructured `Mesh3d` in principle could consist of cells of arbitrary shape, but we limit ourselves to the four standard shapes: hexahedra, tetrahedra, pyramids (with square bases) and prisms (with triangular bases). A nonstructured `Mesh3d` is specified by one-dimensional arrays of `x`, `y`, and `z` coordinates of the same length, the $\mathtt{i}^{\mathtt{th}}$ component of `x`, `y`, and `z` being the coordinates of the $\mathtt{i}^{\mathtt{th}}$ node in the mesh. There is an associated one-dimensional array `c` of data, one value for each node point. Naturally the points alone are not sufficient to specify the connectivity of the mesh. Hence we need configuration information which, for each cell in the mesh, tells which nodes belong to the cell. The Mesh3d class accepts two formats, the Narcisse format peculiar to itself which we will not go into here (See “The Narcisse Format and Keywords” on page 68.), and the AVS format. In the case of the AVS format, for each shape of cell in the Mesh3d, the user must supply a count of the cells, and a one-dimensional array of integer node numbers for each of the cells, in a standard order, as follows:

- tetrahedra--apex, then base nodes, in inward normal order
- pyramids--apex, then base nodes, in inward normal order
- prisms--one triangular face, in outwards normal order, then the corresponding nodes of the opposite face, in inward normal order
- hexahedra--one face, in outwards normal order, then the corresponding nodes of the opposite face, in inward normal order

The allowed keywords for irregular meshes are:

x, y, z, c

The following keywords apply if the mesh is in AVS format:

avs = 1, hex, tet, prism, pyr

The following keywords apply if the mesh is given in Narcisse internal format:

avs = 0, no_cells, cell_descr

Keyword Arguments

The following explains the keyword arguments in detail:

Description of example(s).

x = <values> , y = <values> , z = <values>: three vectors of equal lengths giving the coordinates of the nodes of a nonstructured Mesh3d.

c = <values> : a vector of the same size as x, y, and z giving a data value at each of the node points. c could also be an array of such vectors, when isosurfaces of more than one function are to be plotted. c is also allowed to be one smaller than x, y, and z in each dimension, for cell-centered values.

avs = 0 or 1: if 1, the input data represents a nonstructured Mesh3d in a sort of AVS format, which will be explained in more detail below. The data will be translated into the Narcisse format prior to being sent to Narcisse.

cell_descr = <integer array>: if present, this keyword signifies a nonstructured Mesh3d submitted in the Narcisse format, also explained in more detail below. avs must be zero (or absent) if this keyword is present.

The AVS Format and Keywords

If `avs = 1`, then one or more of the following keywords must also be present; these are used to specify the types of cells present, and their node coordinates, in a standard order. These keywords are:

`hex = [ <list of hexahedral cell data> ]` The two entries in the list (in order) must be: (1) an integer number `n_zones`, which is the number of hex cells in the `Mesh3d`; and (2) a matrix `nz` whose dimensions are `n_zones` by `8`; `nz [i][0]`, `nz [i][1]`, ... `nz [i][7]` give the indices of the 8 vertices of the i^{th} zone in canonical order (one side in the outward normal direction, then the corresponding vertices of the opposite side in the inward normal direction).

`tet = [ <list of tetrahedral cell data> ]` The list is the same format as for `hex` data. The matrix `nz` will now be `n_zones` by `4`, and each row gives the indices of the apex and then the base in inward normal order.

`prism = [ <list of prismatic cell data> ]` The list is the same format as for `hex` data. The matrix `nz` will now be `n_zones` by `6`, and each row gives the indices of one of the triangular sides in the outward normal direction, then the corresponding vertices of the opposite side in the inward normal direction.

`pyr = [ <list of pyramidal cell data> ]` The list is the same format as for `hex` data. The matrix `nz` will now be `n_zones` by `5`, and each row gives the indices of the apex and then the base in inward normal order.

Warning--your numbering of cells must be consistent: all cells of a particular type must be listed together; the actual ordering of the four cell types, however, is irrelevant.

The Narcisse Format and Keywords

The special Narcisse keywords, which apply when `avs = 0`, and their descriptions are:

`no_cells = <integer value>`, the total number of two-dimensional cells in the `Mesh3d`. (Here ‘‘cells’’ really refers to faces of 3d cells).

`cell_descr = <integer array>` , a vector of integers giving the description of the cells of the Mesh3d, as follows:

`cell_descr [0]`, `call_descr [1]`, ... , `cell_descr [no_cells - 1]` tell how many vertices `cell [0]`, `cell [1]` , ... , `cell [no_cells - 1]` have.

`cell_descr [no_cells]` through `cell_descr [no_cells + cell_descr [0] - 1]` are the subscripts of the vertex coordinates of `cell [0]`; `cell_descr [no_cells + cell_descr [0]]` through `cell_descr [no_cells + cell_descr [0] + cell_descr [1] - 1]` are the subscripts of the vertex coordinates of `cell [1]`, etc. Cell vertices must be given in the outward normal order.

Example 1 (a PyNarcisse plot):

In general, PyGist does not support graphing an entire mesh. Instead, PyGist includes support for graphing plane cross-sections and isosurface slices of meshes, which we will discuss later in the chapter.

The following example first reads data from a pdb file called `bills_plot`[1]. The object partitioned by the mesh is an imploding sphere, and the intent is to graph the `z` component of the velocity of implosion. Although the mesh is unstructured, it has only hexahedral cells, and the data is already in an order accepted by PyNarcisse, so need not be rearranged. When PyNarcisse is given an entire mesh to plot, it will plot every face of every cell from front to back; if the mask is other than `"none"`, then the front faces will cover the back ones. If one plots the entire sphere, all that will remain visible at the end is the portion of the exterior of the sphere facing the observer. Therefore in this example we strip away the ‘‘front’’ half of the sphere so that we can observe a cross section.

```
f = PR ('./bills_plot')

n_nodes = f.NumNodes
n_z = f.NodesOnZones
x = f.XNodeCoords
y = f.YNodeCoords
z = f.ZNodeCoords
c = f.ZNodeVelocity
n_zones = f.NumZones

# Now we're going to plot it with all cells missing which
# have an x coordinate greater than 0.005.
n_zones_used = 0
zones_not_used = [] # subscripts of zones to ignore
zones_used = []
for i in range (n_zones) :
   nz = n_z [i]
   used = 1
   for j in range (8) :
      if x [nz [j]] > 0.005 :
         zones_not_used.append (i)
         used = 0
         break
   if used == 1 :
      zones_used.append (i)
      n_zones_used = n_zones_used + 1

new_n_z = zeros ( (n_zones_used, 8), Int32)
for i in range (n_zones_used) :
   new_n_z [i] = n_z [zones_used [i]]

m1 = Mesh3d (x = x, y = y, z = z, c = c, avs = 1,
```

1. `bills_plot` and other files mentioned in this chapter are available on kristen in `/home/cs/motteler/wrk/EB.KEEP`.

```
            hex = [n_zones_used, new_n_z],
            mask = "max", opt_3d = "s4")
    # Uncomment below when we take the front face away
    g2 = Graph3d (m1,
         titles = ["Vertical component of velocity",
         "Imploding Sphere"])
    g2.plot ( )
```

Example 2 (A PyNarcisse Plot):

In the next example, we read in imploding sphere data from file "ball.s0001", which is represented by an unstructured mesh containing all four kinds of cells. The data is not supplied in AVS order, so it is necessary to convert it into AVS format. We then do a number of plots of the data.

```
f = PR ("ball.s0001")
ZLss = f.ZLstruct_shapesize
ZLsc = f.ZLstruct_shapecnt
ZLsn = f.ZLstruct_nodelist
x = f.sap_mesh_coord0
y = f.sap_mesh_coord1
z = f.sap_mesh_coord2
c = f.W_vel_data
# Now we need to convert this information to avs-style data
istart = 0 # beginning index into ZLstruct_nodelist
NodeError = "NodeError"
ntet = 0
nhex = 0
npyr = 0
nprism = 0
nz_tet = []
nz_hex = []
nz_pyr = []
nz_prism = []
for i in range (4) :
   if ZLss [i] == 4 : # TETRAHEDRON
      # put node coords into 4 by no_tet_nodes array
      nz_tet = reshape (ZLsn [istart: istart + ZLss [i] *
               ZLsc [i]], (ZLsc [i], ZLss [i]))
      ntet = ZLsc [i]
      istart = istart + ZLss [i] * ZLsc [i]
   elif ZLss[i] == 5 : # PYRAMID
      # put node coords into 5 by no_pyr_nodes array
      nz_pyr = reshape (ZLsn [istart: istart + ZLss [i] *
               ZLsc [i]], (ZLsc [i], ZLss [i]))
      npyr = ZLsc [i]
      # Now reorder the points (data has the apex last
```

```
        # instead of first)
        for ip in range (npyr) :
           tmp = nz_pyr [ip, 4]
           for jp in range (4) :
              nz_pyr [ip, 4 - jp] = nz_pyr [ip, 3 - jp]
           nz_pyr [ip, 0] = tmp
        istart = istart + ZLss [i] * ZLsc [i]
     elif ZLss[i] == 6 : # PRISM
        # put node coords into 6 by no_prism_nodes array
        nz_prism = reshape (ZLsn [istart: istart + ZLss [i] *
                 ZLsc [i]], (ZLsc [i], ZLss [i]))
        nprism = ZLsc [i]
        # now reorder the points (data has a square face first)
        for ip in range (nprism) :
           tmp = nz_prism [ip, 1]
           tmpp = nz_prism [ip, 2]
           nz_prism [ip, 1] = nz_prism [ip, 4]
           nz_prism [ip, 2] = nz_prism [ip, 3]
           nz_prism [ip, 3] = tmp
           nz_prism [ip, 4] = nz_prism [ip, 5]
           nz_prism [ip, 5] = tmpp
        istart = istart + ZLss [i] * ZLsc [i]
     elif ZLss[i] == 8 : # HEXAHEDRON
        # put node coords into 8 by no_hex_nodes array
        nz_hex = reshape (ZLsn [istart: istart + ZLss [i] *
                 ZLsc [i]], (ZLsc [i], ZLss [i]))
        # hex points are in proper avs order
        nhex = ZLsc [i]
        istart = istart + ZLss [i] * ZLsc [i]
     else :
        raise NodeError, `ZLss[i]` + "is an incorrect number of
nodes."
# Create entire mesh, then create one mesh for each cell type
m1 = Mesh3d (x = x, y = y, z = z, c = c, avs = 1,
           hex = [nhex, nz_hex] ,
           pyr = [npyr, nz_pyr] ,
           tet = [ntet, nz_tet] ,
           prism = [nprism, nz_prism] , mask = "max",
           opt_3d = ["s4","wm"])
m2 = Mesh3d (x = x, y = y, z = z, c = c, avs = 1,
           hex = [nhex, nz_hex] , mask = "max",
           opt_3d = ["s4","wm"])
m3 = Mesh3d (x = x, y = y, z = z, c = c, avs = 1,
           pyr = [npyr, nz_pyr] , mask = "max",
           opt_3d = ["s4","wm"])
```

```
m4 = Mesh3d (x = x, y = y, z = z, c = c, avs = 1,
           tet = [ntet, nz_tet] ,  mask = "max",
           opt_3d = ["s4","wm"])
m5 = Mesh3d (x = x, y = y, z = z, c = c, avs = 1,
           prism = [nprism, nz_prism] , mask = "max",
           opt_3d = ["s4","wm"])
# Now we graph the cells of each type, and then draw the
# whole sphere. N. B. "paws" is a function which halts
# until user enters a carriage return.
g1 = Graph3d (m5)
g1.plot () # draw prisms
paws ()
g1 = Graph3d (m4)
g1.plot () # draw tetrahedra
paws ()
g1 = Graph3d (m3)
g1.plot () # draw pyramids
paws ()
g1 = Graph3d (m2)
g1.plot () # draw hexahedra
paws ()
g1 = Graph3d (m1)
g1.plot () # draw the entire mesh
```

4.3 `Plane` objects

Instantiation

```
from plane import *
pl = Plane ( <normal>, <point>)
```

Description

A `Plane` object is used as an auxiliary geometric object to enable plane slices through structured and unstructured meshes, as described in the next section. `Plane`s cannot be directly passed to a `Graph` object to be plotted; they can, however, be plotted if they are a plane `Slice` object. The two positional arguments used to instantiate a `Plane` are:

<normal>: the direction numbers of the normal to the plane. If both arguments are omitted, this defaults to the positive x axis.

<point>: coordinates of a point through which the plane passes. If this argument is omitted, then the origin is the default.

A `Plane` object's data is actually stored as the coefficients of the plane's equation.

4.4 `Slice` objects

A `Slice` object is created by taking a slice through a `Mesh3d` object or perhaps an earlier-created `Slice`. There are two types of `Mesh3d` `Slice`s: an isosurface slice (i. e., a surface where some specified function on the `Mesh3d` has a constant value), and a plane slice (as created by slicing with a `Plane` object). A pre-existing `Slice` can be sliced only by a `Plane`, and the user has the option of retaining both slices, or of discarding one or the other (useful for seeing inside closed isosurfaces, for example).

The user will not normally instantiate a `Slice` directly, but rather, by invoking the `sslice` function, which does all the work and returns the resulting `Slice`.

Creation of a `Slice`

Isosurface Slice

```
from mesh3d import *
sl = sslice (m, val [, varno])
```

The arguments are as follows:

m: a `Mesh3d` object to be sliced.

val: the value of the function on the isosurface.

varno: the number of the variable for the isosurface; defaults to `1` if not specified. (Recall that the argument `c` to a `Mesh3d` can be a vector of values, in which case isosurfaces for several different functions can be plotted on the same graph.)

Upon return from function `sslice`, `sl` will be assigned the specified `Slice` object, or `None`, if it does not exist.

Plane Slice of Mesh3d

```
from mesh3d import *
sl = sslice (m, plane [, varno])
```

The arguments are as follows:

m: a `Mesh3d` object to be sliced.

plane: a `Plane` object by which to slice the specified `Mesh3d`.

varno: the number of the variable used to color the slicing plane; defaults to `1` if not specified. (Recall that the argument `c` to a `Mesh3d` can be a vector of values, in which case isosurfaces for several different functions can be plotted on the same graph.)

Upon return from function `sslice`, `sl` will be assigned the specified `Slice` object, or `None`, if it does not exist.

Plane Slice of a Slice

```
from mesh3d import *
sl = sslice (s, plane [, nslices])
```

The arguments are as follows:

`s`: a `Slice` object to be sliced.

`plane`: a `Plane` object by which to slice the specified `Slice`:

`nslices`: if `nslices = 1` (the default) then return the piece in front of the `Plane`; if `nslices = 2`, return the pair `[`*`front`*, *`back`*`]` of slices. (If you want just the `"back"` surface, you can achieve this by calling slice with `nslices = 1` and `- plane` instead of `plane`.)

Upon return from function `sslice`, `sl` will be assigned the specified `Slice` object(s), or `None`, or `[None`, `None]`, if it (they) does (do) not exist.

Instantiation

The user will most likely use the `sslice` function to create `Slice` objects, rather than instantiating them directly; but here, for the sake of completeness, is direct instantiation.

```
from mesh3d import *
sl = Slice (nv, xyzv [, val [, plane [, iso]]])
```

Description

The arguments are as follows:

- `nv` is a one-dimensional integer array whose i^{th} entry is the number of vertices of the i^{th} face of the object being sliced.
- `xyzv` is a two-dimensional array dimensioned `sum (nv)` by `3`. The first `nv [0]` triples in `xyzv` are the coordinates of the vertices of `face [0]`, the next `nv [1]` triples are the coordinates of the vertices of `face [1]`, etc.
- `val` (if present) is an array the same length as `nv` whose i^{th} entry specifies a color for face `i`.
- `plane` (if present) says that this is a plane slice, and all the vertices `xyzv` lie in this plane.
- `iso` (if present) says that this is the isosurface for the given value.

A `Slice` object or two `Slice` objects are created by a call to the function `sslice` (See “Creation of a Slice” on page 73). The function `sslice` accepts either a mesh and a specification of how to slice it (isosurface or plane), or else a previously created slice, a plane to slice it with, and whether you want to keep the resulting `"front"` slice or both slices.

Example 1 (a PyGist plot):

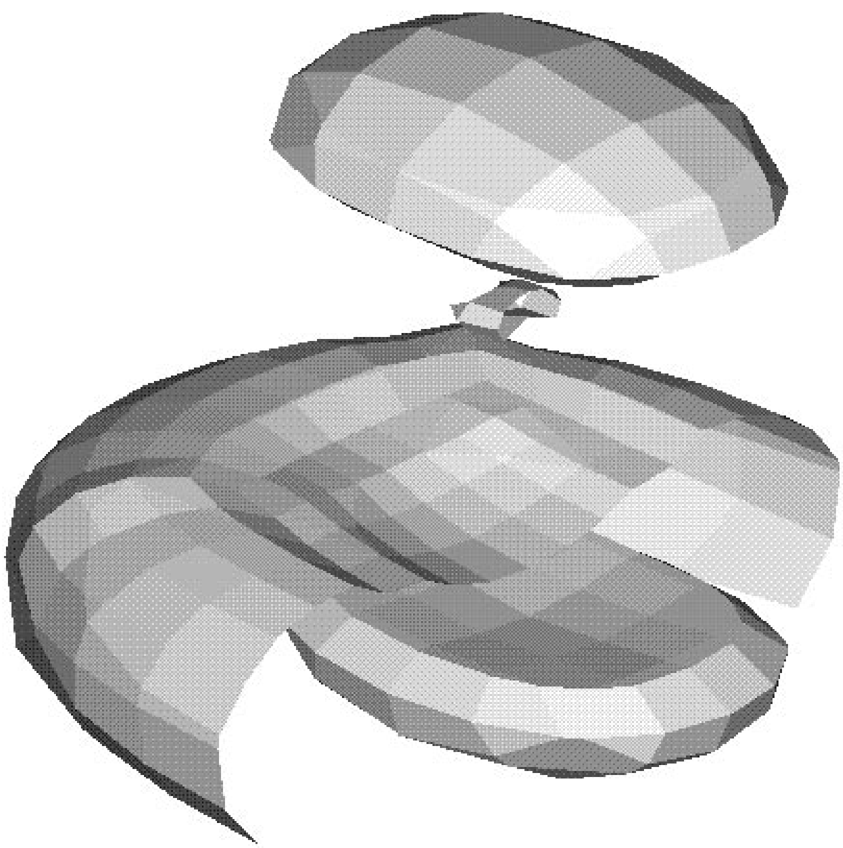
This example reads in the same data as the first PyNarcisse example (“Example 1 (a PyNarcisse plot):” on page 68). The data is already in AVS order, so it does not need to be rearranged. This example takes three plane sections through the imploding sphere, and graphs them with color-filled contours and a color bar. The actual plot is shown in Chapter 5, page 102.

```
from mesh3d import *
from plane import *
from graph3d import *
f = PR ('./bills_plot')
n_nodes = f.NumNodes
n_z = f.NodesOnZones
x = f.XNodeCoords
y = f.YNodeCoords
z = f.ZNodeCoords
c = f.ZNodeVelocity
n_zones = f.NumZones
m1 = Mesh3d (x = x, y = y, z = z, c = c, avs = 1,
   hex = [n_zones, n_z])
# Now define the three planes:
pyz = Plane (array ([1., 0., 0.], Float ),
   array ( [0.0001, 0., 0.], Float))
pxz = Plane (array ([0., 1., 0.], Float ),
   array ( [0., 0.0001, 0.], Float))
p2 = Plane (array ([1., 0., 0.], Float ),
   array ( [0.35, 0., 0.], Float))
# Now define the three slices
s2 = sslice (m1, pyz, varno = 1, opt_3d = ["wm", "s4"])
s22 = sslice (m1, p2, varno = 1, opt_3d = ["wm", "s4"])
s23 = sslice (m1, pxz, varno = 1, opt_3d = ["wm", "s4"])
# Create the graph
g1 = Graph3d( [s2, s22, s23], color_card = "rainbow.gp",
        opt_3d = ["wm", "s4"], mask = "min", color_bar = 1,
        split = 0, hardcopy = "talk.ps")
# plot the graph
g1.plot ()
```

Example 2 (a PyGist plot):

Now we shall plot three isosurfaces of the same sphere shaded by a light source (`opt_3d` = "`none`"). The isosurfaces are nested and one will block our view of another, so we slice it for better visibility. Note that the slices isosurface is disconnected, because you see two sliced pieces!

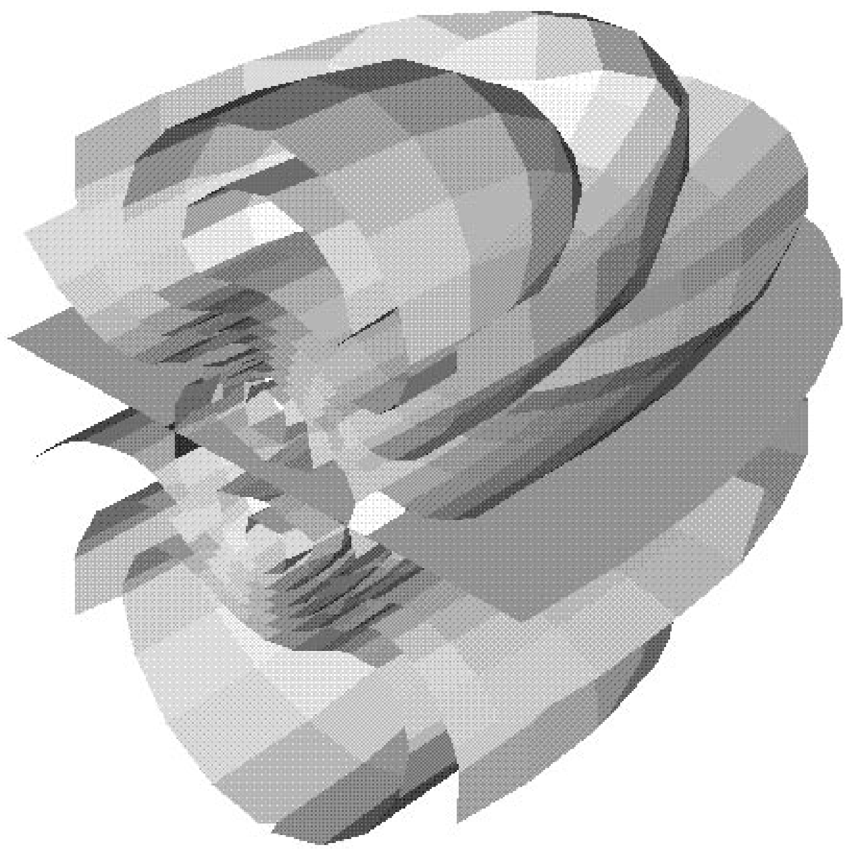
```
s1 = sslice (m1, .9 * max (c), varno = 1)
s2 = sslice (m1, .9 * min (c), varno = 1, opt_3d = "none")
s5 = sslice (m1, .5 * max (c), varno = 1, opt_3d = "none")
s6 = sslice (s5, -pyz, opt_3d = "none")
g1.set_surface_list ( [s1, s2, s6])
g1.plot ()
```



Example 3 (a PyGist plot):

The next plot consists of a number of isosurfaces of the above imploding sphere shaded by a light source (`opt_3d` = "`none`"). The isosurfaces are nested and some are closed, so we slice all of them in half and keep the ‘‘back’’ halves for visibility.

```
for i in range (8) :
   sl = sslice (m1, .9 * min (c) + i * (.9 * max (c) - .9 *
      min (c)) / 8., varno = 1, opt_3d = "none")
   slice_list.append (sslice (sl, pxz))
g1.set_surface_list ( slice_list)
g1.plot ()
```



4.5 3D Animation

Graph3d objects have two methods that enable realtime animation of 3D plots. These are described in the Graph3d chapter; see “3d Animation Methods” on page 93.

CHAPTER 5: Graph Objects

A `Graph` object is defined as a container for geometric objects which also contains the type of information common to all graphs (for example, titles, axis labels and scales, and the like). A `Graph` object can be asked to plot itself; the user can supply one or more `Plotter` objects to the `Graph`, or else leave it up to the `Graph` to try to obtain its own default `Plotter` object. The base class `Graph` is not normally instantiated as is. Instead, the user will normally instantiate its derived classes, `Graph2d` and `Graph3d`.

5.1 Graph2d Objects

Instantiation

```
from graph2d import *
g2 = Graph2d ( <object list>, <keylist>)
```

Description

A `Graph2d` is a two-dimensional graphics object which contains one or more 2d geometric objects plus a global environment. It will accept one or a list of `Plotter` objects or plotter identifiers, or will try to complete generic connection(s) of its own if asked to plot without having been given a plotter specification.

<object list> is one or a sequence of 2d geometric objects. It makes sense sometimes to graph several `Curve` objects on one plot; the user can specify several 2d objects of other types, or even of mixed types, but does so at his/her own risk.

A list of keyword arguments accepted by `Graph2d` is:

plotter, filename, display, graphics, style, label_type, titles, title_colors, grid_type, axis_labels, x_axis_label, y_axis_label, yr_axis_label, axis_limits, x_axis_limits, y_axis_limits, yr_axis_limits, axis_scales, x_axis_scale, y_axis_scale, yr_axis_scale, text, text_color, text_size, text_pos, color_card, xyequal, sync, color_bar, color_bar_pos

There are a number of methods available in `Graph2d` to enable the user to reconfigure an existing object. Let's say that `g2` is a `Graph2d` object.

`g2.new (` *<object list>*, *<keylist>*`)`: has the same arguments as `Graph2d`, and simply reinitializes an existing `Graph2d` object, instead of instantiating a separate one.

`g2.add ( <2d object>)` adds the specified 2d object to the others already in the `Graph2d`. (2d objects are numbered in the order that they are put into the graph, beginning with 1.)

`g2.delete (n)`: deletes the nth 2d object from the `Graph2d`.

`g2.replace (n, <2d object>)`: replaces the nth 2d object in the `Graph2d` with the one specified.

`g2.change_plot ( <keyword arguments>)`: used to change any `Graph2d` characteristics except the 2d objects being graphed. Use the `add`, `delete`, and/or `replace` methods to do that. `change_plot` will draw the graph without sending object coordinates, unless keyword `send` is `1`. Generally, `change_plot` should be used when the graph needs to be recomputed, and `quick_plot` (below) when it does not. `change_plot` does no error checking and does not convert user-friendly names of colors and such into numbers.

`g2.quick_plot ( <keyword arguments>)` is used to change some Graph2d characteristics which do not demand that the graph be recomputed. You can change the characteristics of a 2d object in the graph by specifying its number (`curve = n`) and any combination of the traits `type`, `color`, and `label`. Or you can change such overall graph characteristics as `label_type`, `titles`, `title_colors`, `text`, `text_color`, `text_size`, `text_pos`, `color_card`, `grid_type`, `sync`, and `axis_labels`. The changes will be effected and the graph redrawn.

Things that you cannot change include axis limits and scales, and the coordinates of a curve. Use `change_plot` if axis limits and scales are among the things you want to change, and use `add`, `delete`, or `replace` followed by a call to `plot`, if you wish to change the 2d object list.

`g2.plot ( )` plots a 2d graph. If the user has not by now specified `Plotter`(s) or `filename`(s) then a generic `Plotter` object will be created, if it is possible to find a local Graphics routine.

`Graph2d` objects inherit from base class `Graph`, as does `Graph3d`. The following methods are inherited from `Graph`:

`g2.add_file ("filename")` allows the user to add a `Plotter` contacted via `"filename"` to the list of `Plotter`s being used to draw the current `Graph` object.

`filename` has different (and incompatible) meanings for PyNarcisse and PyGist, because the two graphics packages are so fundamentally different in the way that the user program communicates with them. Please consult the discussion of keyword arguments in the following section.

`g2.delete_file ("filename")` allows the user to delete a `Plotter` contacted via `"filename"` from the list of `Plotter`s being used by this object.

`g2.add_display ("host")` and `g1.delete_display ("host")` adds or deletes a `Plotter` displaying on the specified `host`. Currently, these are the same as the `add_file` and `delete_file` for PyGist.

g2.add_plotter (pl) allows the user to add the specified Plotter to the list of Plotters being used by this object.

g2.delete_plotter (pl) allows the user to delete the specified Plotter from the list of Plotters being used by this object.

g2.change (*<keyword arguments>*) allows some of the graph's generic characteristics to be changed. These are sync, titles, title_colors, style (PyGist only), grid_type, axis_labels (and x_axis_labels, etc.), axis_limits (and x_axis_limits, etc.), axis_scales (and x_axis_scales, etc.), text, text_color, text_size, and text_pos.

Keyword Arguments

This section describes the Graph2d keyword arguments. The first three keywords are used to identify where and how you want the graph plotted. A Graph2d will create a default plotter if none of these keywords is specified; see below.

plotter = <Plotter object> or a sequence of <Plotter object>s if you have a Plotter object or a sequence of them that you want the Graph2d to use when plotting itself. In particular, if you want to plot only to CGM or PostScript files and not interactively (i. e., in batch mode), then you will need to instantiate your own Plotter and give it to the Graph2d object. Currently Graph objects cannot instantiate their own batch Plotters, although this feature will eventually be added.

filename = <string> or a sequence of <string>s specifies plotting associated with a particular filename. PyGist and PyNarcisse differ dramatically in the meaning of this keyword, as explained below.

PyGist: currently, filename is synonymous with the display keyword, which specifies a host where the PyGist window is to be displayed. If the user wants a plotter which plots to a given CGM or PostScript file (or one of each), then the user must instantiate one or more Plotter objects and hand them to the Graph2d via the plotter keyword. Eventually this will be changed to make it easier on the user. But for future compatibility, use the display keyword to specify where PyGist will open its window, and instantiate a Plotter yourself if you wish to have PyGist send plots to a file. (See “Plotters: A Brief Primer” on page 113..)

PyNarcisse: filename can be used in two different ways.

(1) As a way to specify a Narcisse process to connect to. In this case, the filename should be in the form "machine+port_serveur++user@ie.32" where machine specifies where the display is to take place (e. g., "icf.llnl.gov:0.0"), port_serveur is the port number displayed on the Narcisse GUI, and user is the userid of the person running.

(2) As a filename where Narcisse is to dump its plots. In this case, use the file suffix ".spx" to specify a binary file, or ".spc" for an ascii dump file. If a filename of this form is specified, PyNarcisse attempts a connection to Narcisse using the value of the DEST_SP3 environment variable, if it exists, and if not, attempts to construct a connection filename using DISPLAY environment variable for machine, the value of the PORT_SERVEUR environment variable (or the default 2101 if PORT_SERVEUR is not defined) for port_serveur, and the value of

USER for user.

display = <string> or a sequence of <string>s if you want to display on the named hosts. (This keyword is ignored by PyNarcisse, since the desired display is specified in the filename keyword.) The form of <string> is the usual

"hostname:server.screen"

The purpose of this argument is to allow you to continue without exiting Python if you have to open a Gist window without the DISPLAY environment variable having been set, or if you want to open Gist windows on more than one host.

If none of the above three keywords is specified, then when asked to plot, a Graph2d will attempt to connect to a plotter as follows:

- It first examines the environment variable PYGRAPH to see what type of graphics is desired (the allowed values are "Gist" and "Nar"). If this variable is undefined, then the default is "Gist".
- If the graphics is "Gist", it attempts to open a Gist graphics window on the host specified by the DISPLAY environment variable.
- If the graphics is "Narcisse", it attempts a connection using the value of the DEST_SP3 environment variable, if it exists, and if not, attempts to construct a connection filename using DISPLAY environment variable for machine, the value of the PORT_SERVEUR environment variable (or the default 2101 if PORT_SERVEUR is not defined) for PORT_SERVEUR, and the value of USER for user.

The remaining Graph2d keyword arguments are as follows:

graphics = <string> or a sequence of <string>s if you want to specify which graphics the Plotter or Plotters associated with this Graph2d will connect to. Currently the values allowed are "Nar" and "Gist". This argument is meaningless if you supply one or a list of Plotters via the plotter keyword. If <string> is a scalar and you have supplied a list of filenames, then all Plotters opened will be that type. If it is a vector, then it must match the list of filenames in size and correspond to the filename, i. e., don't give a Narcisse-style filename and specify a "Gist" Plotter for it.

style = one of "vg.gs", "boxed.gs", "vgbox.gs", "nobox.gs", "work.gs". For Gist only, and only if a Plotter has not already been specified, then each Plotter opened will have axes plotted in the specified style. The default is "work.gs".

grid_type = <string>: where "none" means no axis grid, "axes" means a pair of axes with tick marks, "wide" means a widely spaced 2d grid, and "full" means a closely spaced 2d grid. (By a ‘‘grid’’ here, we mean a set of lines parallel to the coordinate axes which overlay the graph when plotted.)

label_type = "end" (to label the curve at its end), "box" (to put the labels in a box) Applicable to PyNarcisse only.

titles = <value> where <value> is a string or a sequence of up to four strings, giving the titles in the order bottom, top, left, right.

title_colors = <value> where value is an integer or string or a sequence of up to four integers or strings giving the colors of the titles.

axis_labels = <value> where <value> is a string or sequence of up to three strings representing the labels of the x axis, the y axis, and the right y axis.

x_axis_label, y_axis_label, and yr_axis_label may be used to label individual axes.

axis_limits = <value> where <value> is a pair [xmin, xmax] or a sequence of up to three pairs, where the second would be the y limits, and the third the yr limits.

x_axis_limits, y_axis_limits, and yr_axis_limits may be used to specify limits on individual axes.

axis_scales = "linlin", "linlog", "loglin", or "loglog" or, if all three axes are to be specified, a triple of the values "lin" and "log".

x_axis_scale, y_axis_scale, and yr_axis_scale may be used to specify individual axis scales.

text = <value> where <value> is a string or a sequence of strings representing texts to be placed on the plot.

text_color = <value> where <value> is a color number or name, or a sequence of color numbers or names giving colors for the texts.

text_size = <value> where <value> is an integer or a sequence of integers giving (roughly) the number of characters in a line on the graph (PyNarcisse) or the point size (PyGist).

text_pos = <value> where <value> is a pair or a sequence or reals between 0. and 1.0 giving the relative position of the lower left corner of a text in the graphics window.

color_card = <value> specifies which color card you wish to use, e. g., "rainbowhls" (the default), "random", etc. Note that for Graph2d, color_card is a keyword, since it is not possible to specify different color cards on the same 2d graph, whereas linked 3d and 4d graphs can have different color cards. For details on color cards, See “Narcisse Color cards” on page 48. and See “Gist Color Cards” on page 49..

xyequal = 0/1: If 1, the axis limits will be adjusted so that both axes are to the same scale.

sync = 0 or 1: (1 to synchronize before sending a plot) defaults to 1, otherwise plots may get garbled. Only applicable to PyNarcisse.

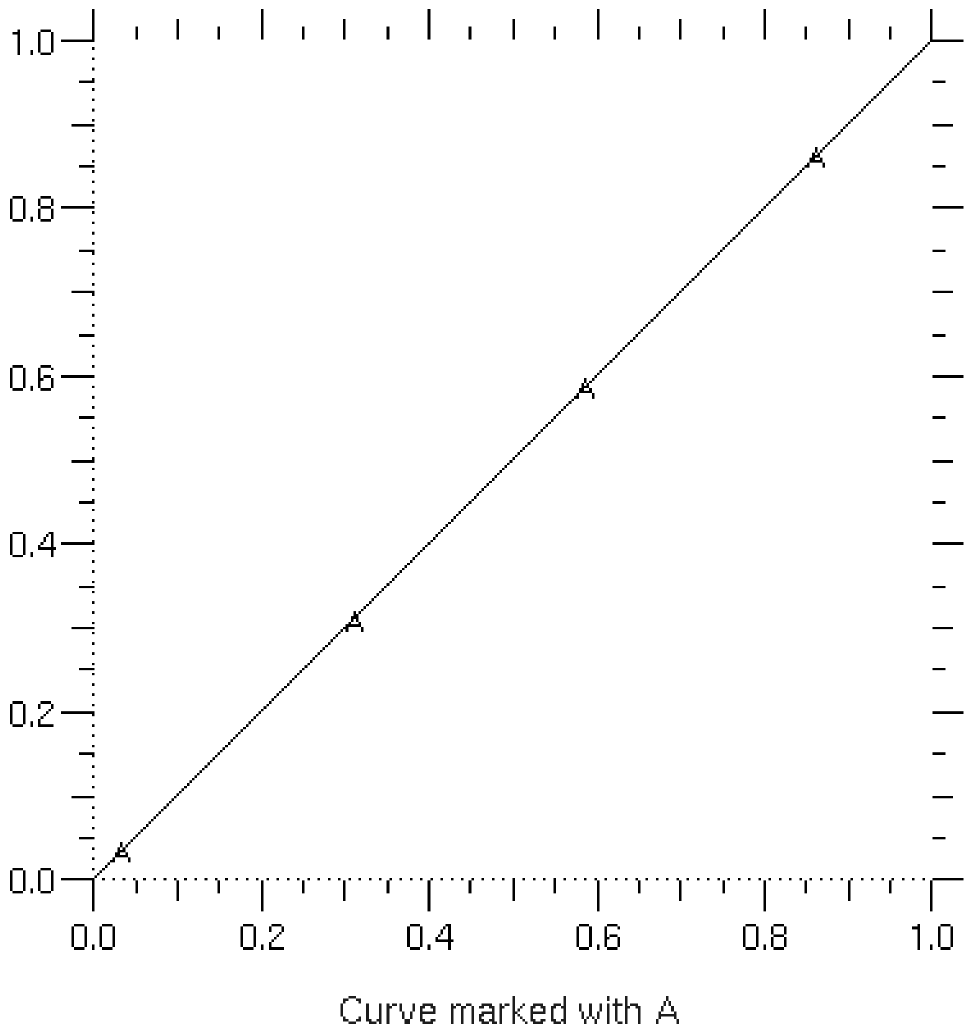
color_bar = 0 or 1: (1 enables plotting of a color bar on any graphs for which it is meaningful (colored contour plots, filled contour plots, cell arrays, filled meshes and polygons).

color_bar_pos (ignored unless a color bar is actually plotted) is a 2d array [[xmin, ymin], [xmax, ymax]] specifying where (in window coordinates) the diagonally opposite corners of the color bar are to be placed.

Examples

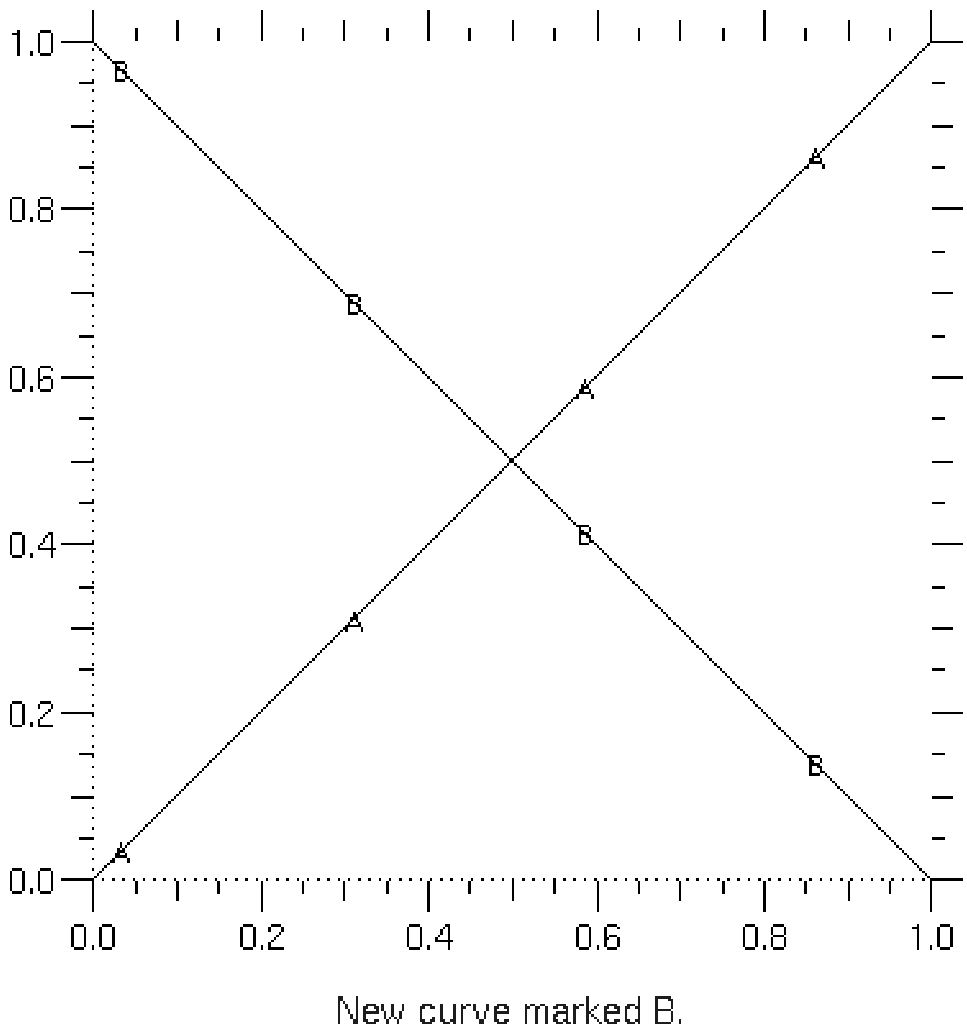
The following sequence of instructions creates a simple curve (a straight line), a 75 dpi plot window, passes both objects to a new `Graph2d` object with a blue title, and does the plot:

```
c1 = Curve ( y = [0,1] , marks = 1 , marker = "A" )
pl = Plotter ( dpi = 75 )
g1 = Graph2d ( c1, plotter = pl , titles =
     "Curve marked with A" , title_colors = "blue")
g1.plot ()
```



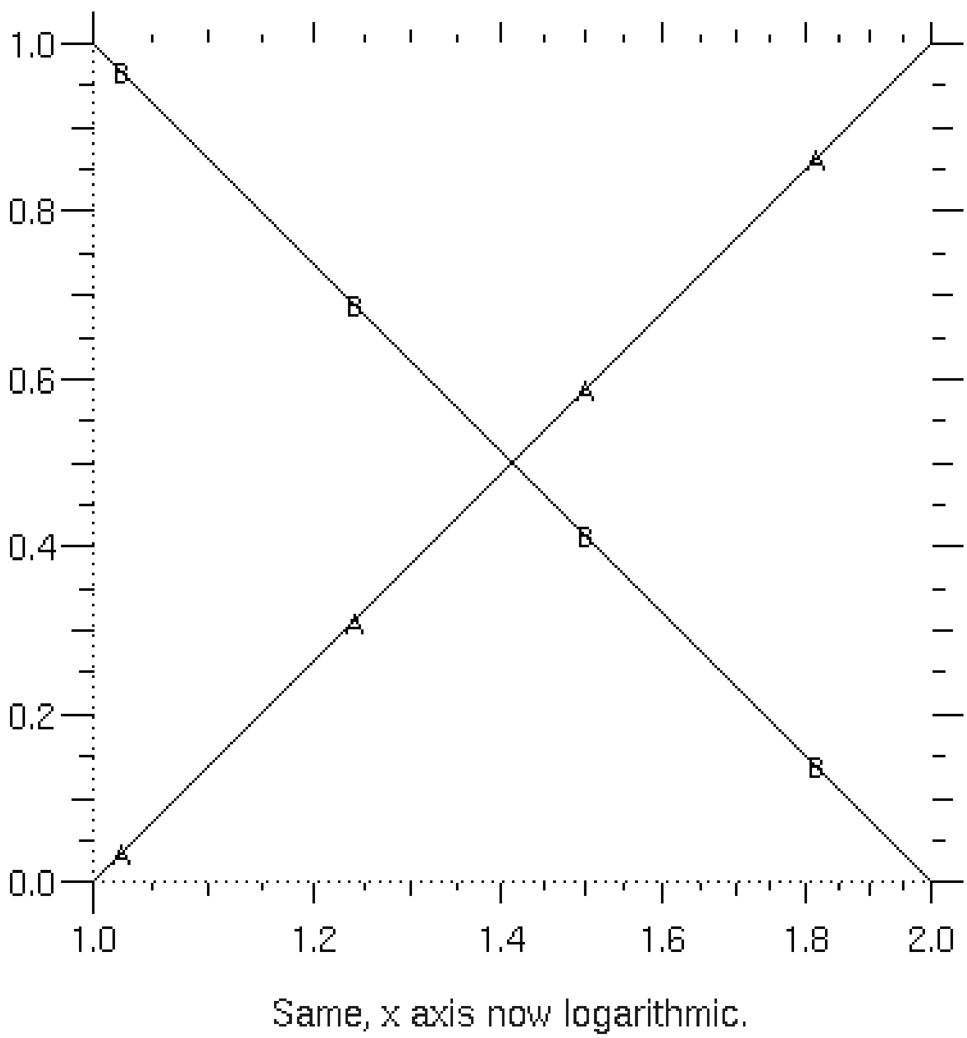
This next sequence adds a second curve to the above `Graph2d`, and changes the title:

```
c2 = Curve ( y = [1,0] , marks = 1 , marker = "B")
g1.add (c2)
g1.change (titles = "New curve marked B.")
g1.plot ()
```



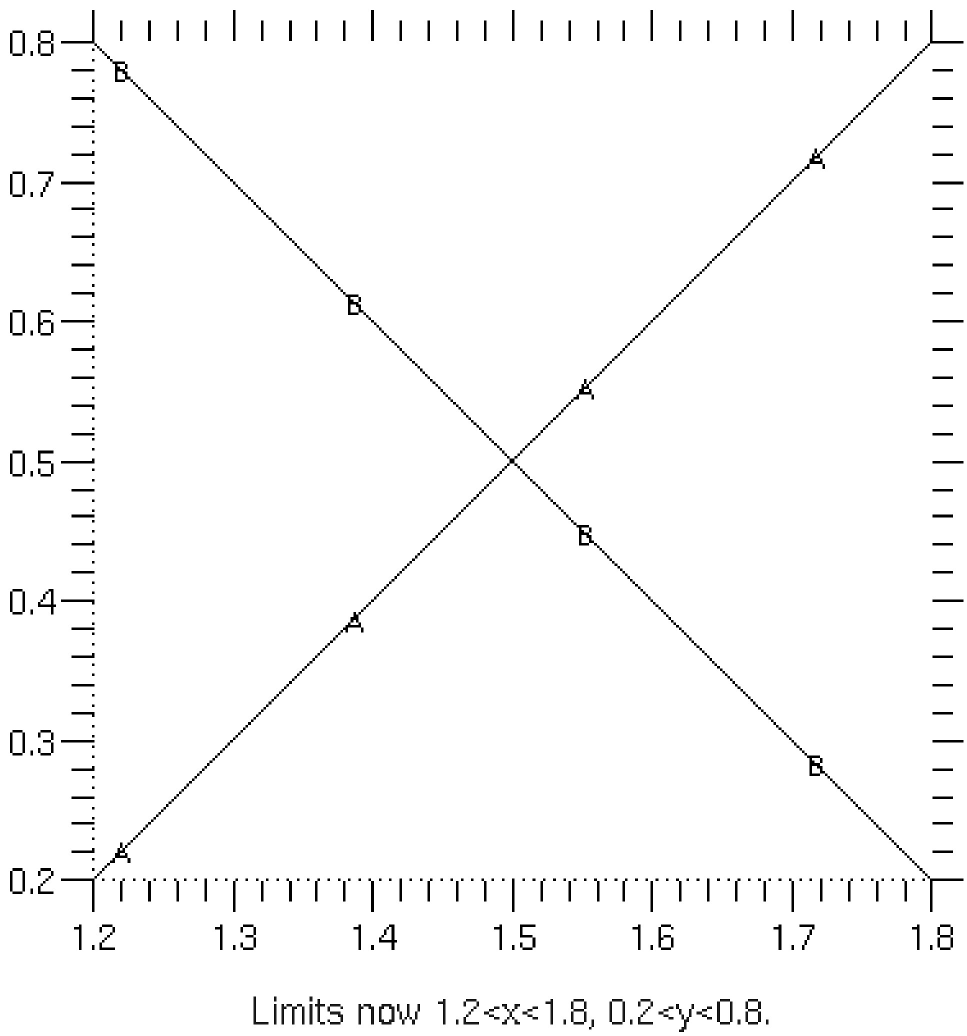
Now we set the x coordinates of the two curves. g1 has already been given references to c1 and c2, so the changes will be visible to g1 and will be reflected in the new plot. g1's title is changed, and the x scale is changed to logarithmic.

```
c1.set ( x = [1,2] )
c2.set ( x = [1,2] )
g1.change (axis_scales = "loglin",
   titles = "Same, x axis now logarithmic.")
g1.plot ( )
```



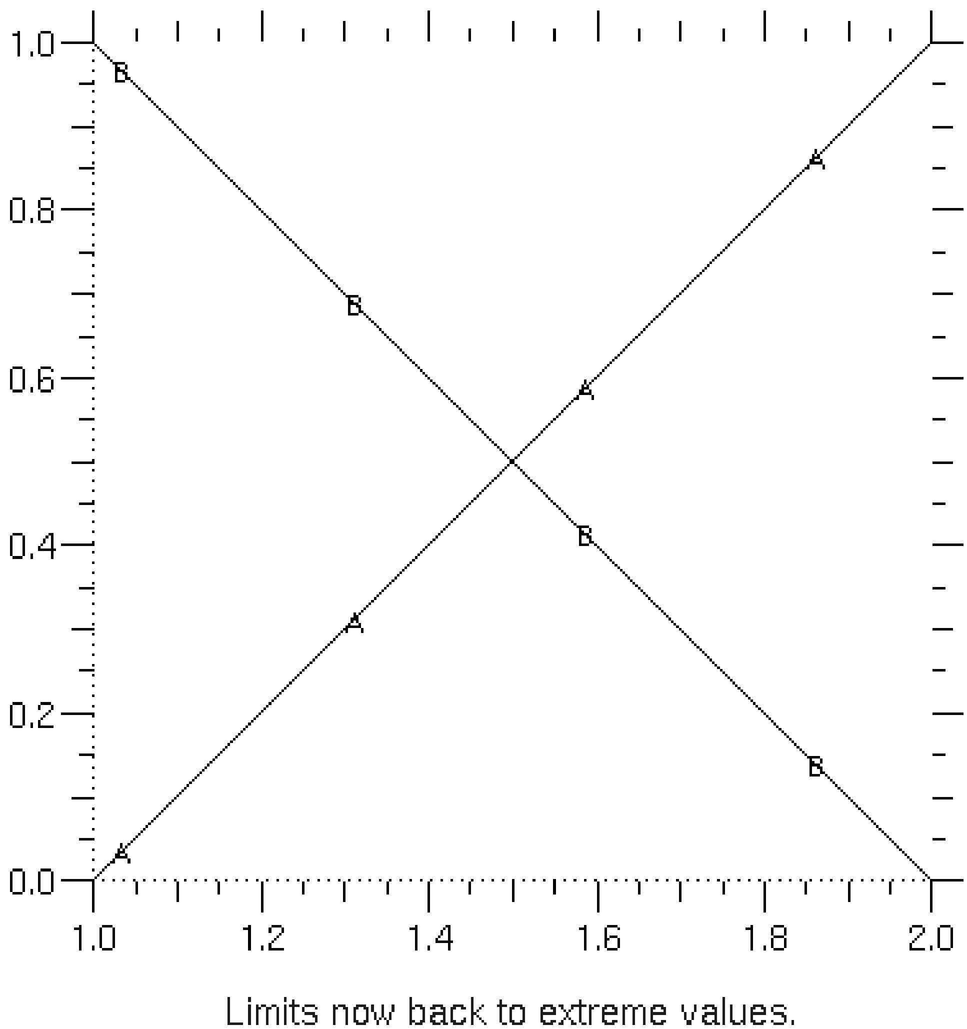
Change the `x` axis scale back to linear, and change the axis limits to show only a part of the graph:

```
g1.change (x_axis_scale = "lin",
           axis_limits=[[1.2,1.8],[0.2,0.8]],
           titles="Limits now 1.2<x<1.8, 0.2<y<0.8.")
g1.plot ()
```



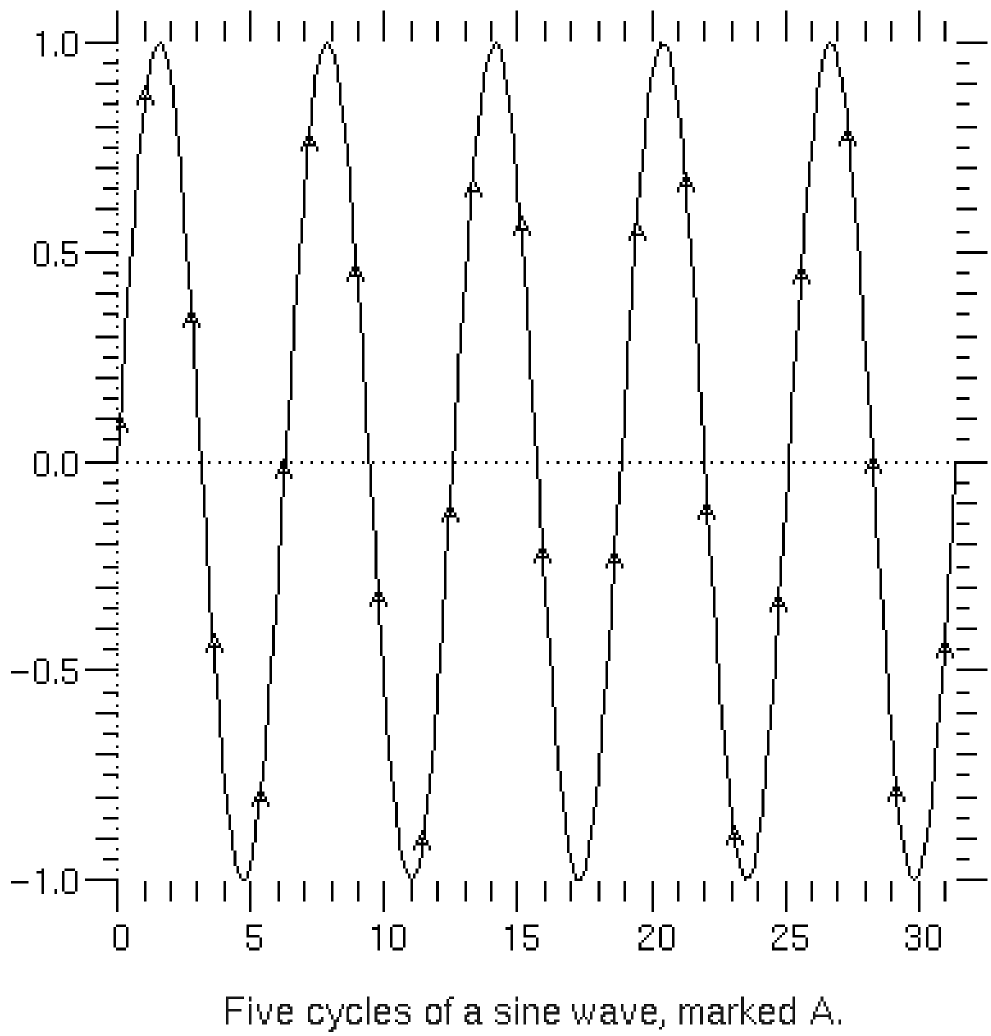
Change the axis limits back to defaults, i. e., values computed from the data by PyGist:

```
g1.change(axis_limits="defaults",
          titles="Limits now back to extreme values.")
g1.plot ()
```



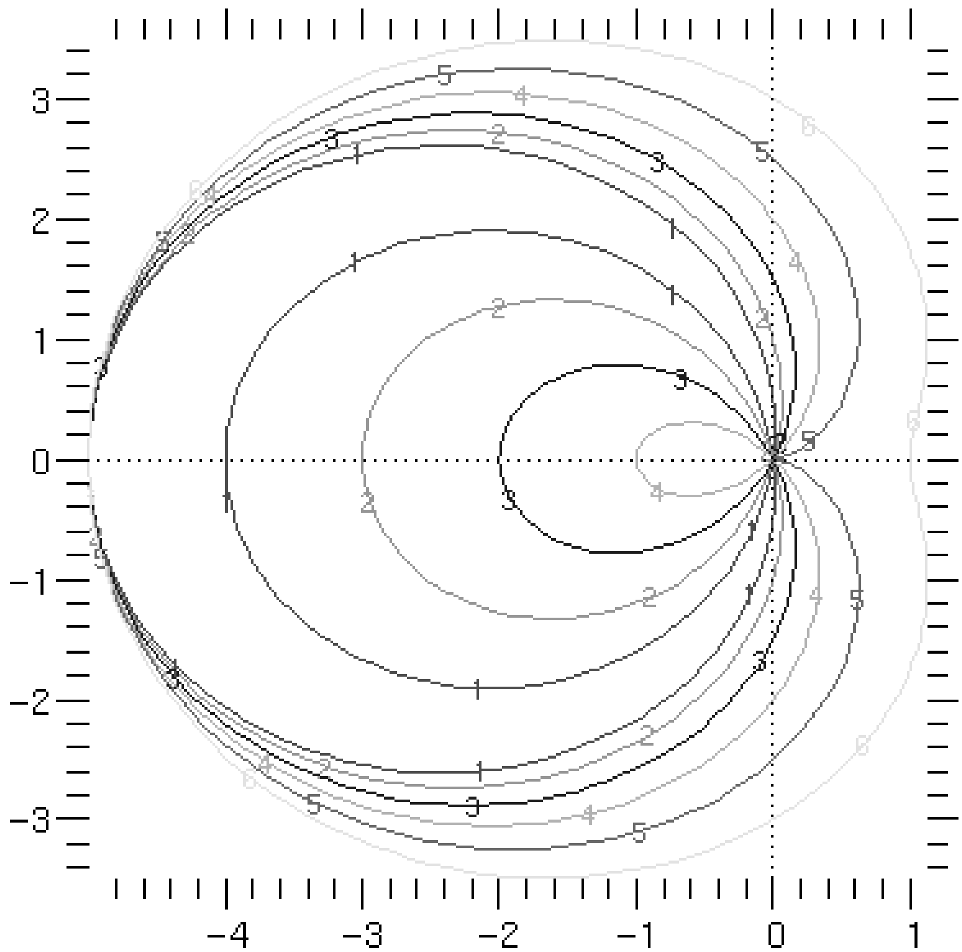
The next example shows how you can change the curve or curves associated with a `Graph2d` object. Here we delete the two curves associated with `g1`, then add the new one, change the title, and plot.

```
x=10*pi*arange(200, typecode = Float)/199.0
c1 = Curve ( x = x , y = sin(x),marks = 1, marker= "A")
g1.delete (2)
g1.delete (1)
g1.add (c1)
g1.change (titles = "Five cycles of a sine wave, marked A.")
g1.plot ( )
```



The next sequence of code creates a list of `Curve` objects which are nested cardioids with different parameters. It then creates a new `Graph2d` containing these `Curve`s, and plots them in different colors, labeling each with a number.

```
x=2*pi*arange(200, typecode = Float)/199.0
crvs = []
for i in range (1,7) :
   r = 0.5*i -(5-0.5*i)*cos(x)
   s = `i`
   crvs.append(Curve(y=r*sin(x),x=r*cos(x),marks=0,
      color=-4-i,label=s))
g1=Graph2d(crvs,plotter = pl,
   titles="Nested cardioids in colors")
g1.plot ( )
```



Nested cardioids in colors

5.2 Graph3d Objects

Instantiation

```
from graph3d import *
g3 = Graph3d ( <object list>, <keylist>)
```

Description

A `Graph3d` is a container for one or more three dimensional geometric objects (`Surface`s, `Mesh3d`s, and/or `Slice`s) as well as global information about the graph. It will accept one or a list of `Plotter` objects or `Plotter` identifiers, or will try to complete generic connection(s) of its own if asked to plot without having been given a plotter specification.

<object list> is one or a sequence of 3d geometric objects. It makes sense sometimes to graph several such objects on one plot. By means of `link`ing two or more objects (see description below), it is possible though somewhat difficult in PyNarcisse to plot two or more objects with different 3d/4d options, palettes, etc. on the same graph. PyGist does not allow this; however, in mesh plots which mix isosurfaces and plane slices, PyGist allows a split palette option, which shades the isosurfaces as if from a light source, but colors plane slices according to the specified function.

A list of keyword arguments accepted by `Graph3d` is:

> **plotter, filename, display, titles, title_colors, grid_type, axis_labels, x_axis_label, y_axis_label, z_axis_label, c_axis_label, yr_axis_label, axis_limits, x_axis_limits, y_axis_limits, z_axis_limits, c_axis_limits, yr_axis_limits, axis_scales, x_factor, y_factor, x_axis_scale, y_axis_scale, z_axis_scale, c_axis_scale, yr_axis_scale, text, text_color, text_size,text_pos, phi, theta, roll, distance, link, connect, sync, ambient, diffuse, specular, spower, sdir, color_bar, color_bar_pos**

`Graph3d` objects inherit from base class `Graph`, as does `Graph2d`. The following methods are inherited from `Graph`: `add_file`, `delete_file`, `add_plotter`, `delete_plotter`, and `change`. See "Description" on page 79. for details. In addition, a `Graph3d` has the following methods which, except where noted, are similar to the `Graph2d` methods with the same names: `new`, `add`, `delete`, `replace`, `change_plot`, `quick_plot`, and `plot`.

Notes:

- `new` has the same arguments as `Graph3d.new`.
- `add`, `delete`, and `replace` have the same calling sequence as the same-named methods in `Graph2d`, except, of course, that the number refers to a 3d object in the `Graph`.

- `change_plot` carries the same caveats as the `Graph2d` method by the same name.
- `quick_plot` is used to change some `Graph3d` characteristics which do not demand that the graph be recomputed. You can change the characteristics of a `Surface` (or other object) in the graph by specifying its number (`surface = n`) and any combination of the traits `color_card`, `opt_3d`, `mesh_type`, or `mask`. Or you can change such overall graph characteristics as `titles`, `title_colors`, `text`, `text_color`, `text_size`, `text_pos`, `color_card`, `grid_type`, `sync`, `theta`, `phi`, `roll`, and `axis_labels`. Or you can do both. The changes will be effected and the graph redrawn. Things that you cannot change include axis limits and scales, and the coordinates of a `Surface`. Use `change_plot` if axis limits and scales are among the things you want to change, and use `add`, `delete`, or `replace` followed by a call to `plot`, if you wish to add, delete, or change a `Surface`. `quick_plot` will not work right for `link`ed Surfaces. Once the changes have been made, you will have to call `plot`.

3d Animation Methods

Finally, `Graph3d` has two methods which have to do with real time animation of 3d plots. These methods are as follows:

`move_light_source (`*`<keylist>`*`)` The keyword arguments are:

`nframes` (default `30`): the number of frames in the proposed movie.

`angle` (default `360 / nframes`): the angle (in degrees) through which the light source rotates for each frame.

This method is not yet implemented in PyNarcisse.

`rotate (`*`<keylist>`*`)` The keyword arguments are:

`axis` (default `[-1., 1., 0.]`): the direction numbers of the axis about which the graph is rotated.

`nframes` (default `30`): the number of frames in the proposed movie.

`angle` (default `360 / nframes`): the angle (in degrees) through which the graph is rotated for each frame.

This method is not yet implemented in PyNarcisse.

Keyword Arguments

The following keywords inherited from `Graph` have exactly the same behavior as described under `Graph2d` (See “Keyword Arguments” on page 81.):

`plotter, filename, display, titles, title_colors, grid_type` (PyNarcisse only)**`, text, text_color, text_size, text_pos, color_bar, color_bar_pos`**

Up to four axes are possible in 3d and 4d plots : `x`, `y`, `z` or `c` (depending on whether we chose the option of switching z and c), and the right y axis (when the left and right sides of the plot have different y axis scales, with some objects plotted on one and some on another), so the specifications of axis characteristics are different from those for Graph2d. The axis characteristic keywords (primarily applicable to PyNarcisse, but see the next paragraph) are:

axis_labels, axis_limits, axis_scales

Each should be specified as a list of up to five items, in the order x, y, z, c, yr; items omitted from the right will be defaulted. axis_labels are strings; axis_limits are pairs of floats; and axis_scales are one of the two strings "lin" or "log".

PyGist does not currently support full axes display in 3d. Instead, it is capable of displaying a *gnomon* in the lower left corner of a 3d plot, i. e., a small representation showing the orientation of the three coordinate axes, with the labels in reverse video if they are pointed ‘‘into’’ the plane of the plot. These labels default to ‘‘**X**’’, ‘‘**Y**’’, and ‘‘**Z**’’, but the defaults can be overruled by the axis_labels keyword. PyGist will only use the first letter of each specified label, if longer than one letter. The keyword gnomon (if set to nonzero) turns on the gnomon display.

Another peculiarity of PyGist is its tendency to stretch the plotted surface so that it extends from edge to edge of the plotting area. The keywords x_factor and y_factor can be used to force the display to appear in proper perspective; in most cases leave x_factor alone, and set y_factor to 2.0. Both keywoirds default to 1.0.

Other keywords which are peculiar to Graph3d objects are:

phi = <integer value> specifies the angle that the line from the view point to the origin makes with the positive z axis. The angle is in degrees.

theta = <integer value> specifies the angle made by the projection of the line of view on the xy plane with the positive x axis. The angle is in degrees.

roll = <integer value> specifies the angle of rotation of the graph around the line from the origin to the view point. The angle is measured in the positive direction, i. e., if your right thumb is aligned outwards along the line from the origin to the view point, then your fingers curl in the positive direction. The angle is in degrees. (This keyword is not available in PyGist, and will be ignored if supplied.)

distance = <integer value> specifies the distance of the view point from the origin. This is an integer between 0 and 20. 0 makes the distance infinite; otherwise the smaller the number, the closer you are. This number does not affect the size of the graph, but rather the amount of distortion in the picture (the closer you are, the more distortion).

The following keywords are applicable only in PyNarcisse:

link = 0 or 1: Used to link surfaces of different 3d options. normally all surfaces in a graph will have the same 3d options. This value should be set to 1 if you want to graph two or more surfaces with different 3d options. otherwise multiple surface graphs will appear with the options of the last surface specified. This may not always work as expected, since successive objects in a `link`ed `Graph` are plotted on top of whatever is in the current window. That may not be where they are positioned; e. g., it would be easy to have an object that is really behind another be drawn on top of it.

`connect` = `0` or `1:` set to `1` for graphs of more than one 3d object to provide better hidden line removal. Must not be used with `link`.

`sync` = `0` or `1`: set to `1` to synchronize with Narcisse before plotting the next graph. Keeps graphs sent in rapid succession from becoming garbled. Defaults to `1`; set it to `0` if you don't have a timing problem.

The following lighting keywords are applicable only in PyGist:

`ambient` = `<value>` is a light level (arbitrary units) that is added to every surface independently of its orientation. High values of this argument cause the surface to appear to glow with its own light, making it so bright as to lose contrast. Low values of this argument mean that reflected and diffuse light are more important in visualizing the surface.

`diffuse` = `<value>` is a light level which is proportional to `cos (theta)`, where `theta` is the angle between the surface normal and the viewing direction, so that surfaces directly facing the viewer are bright, while surfaces viewed edge on are unlit (and surfaces facing away, if drawn, are shaded as if they faced the viewer, so that if we are looking at the inside of a surface, it will look properly three-dimensional).

`specular` = `<value>`

`spower` = `<value>`

`sdir` = `<value>`

`specular` = `S_LEVEL` is a light level proportional to a high power `spower` = `N` of `1 + cos (alpha)`, where `alpha` is the angle between the specular reflection angle and the viewing direction. The light source for the calculation of `alpha` lies in the direction `sdir` = `XYZ` (a 3 element vector) in the viewer's coordinate system at infinite distance. You can have `ns` light sources by making `S_LEVEL`, `N`, and `XYZ` (or any combination) be vectors of length `ns` (`ns`-by-`3` in the case of `XYZ`).

The four parameters `ambient`, `diffuse`, `specular`, and `spower` act together to produce interesting effects. if `diffuse` and `specular` are both 0, then the surface will not be reflective, and all three dimensional appearance will be lost. `specular` and `spower` together determine how reflective the surface is; large `spower` with `specular` not `0` gives small, bright highlights with most of the surface appearing black. As `spower` decreases, the highlights become somewhat larger and darker portions of the surface become lighter. If `diffuse` is not zero but `specular` and `ambient` are zero, then the surface will appear shaded gently, brighter on the side(s) toward the light source(s), but not highly reflective. The user is encouraged to experiment to find the desired effect.

`split` = `0` or `1` (default `1`) If `1`, causes the palette to be split when both planes and isosurfaces are present in a graph, so that isosurfaces are shaded according to current light settings, while plane sections of the mesh are colored according to a specified function. (The lower half of the palette is grey scale, and the upper half is (usually) rainbow.

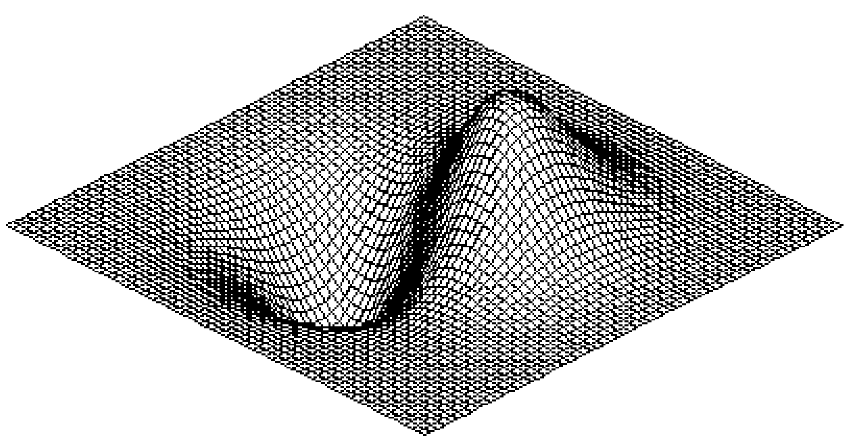
Example 1. Surface plots.

All of the plots illustrated in this example are of the following surface; it is an interesting symmetric surface with a peak and a valley.

```
x = span (-1, 1, 64, 64)
y = transpose (x)
z = (x + y) * exp (-6.*(x*x+y*y))
s1 = Surface (z = z, opt_3d = "wm", mask = "sort")
```

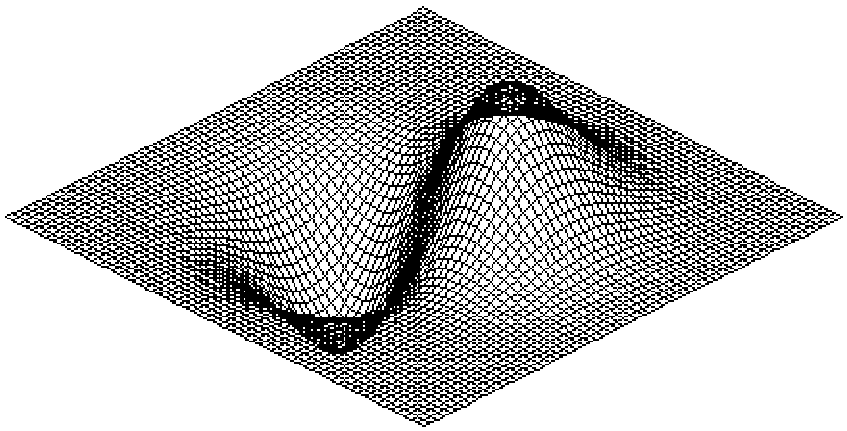
In each case, the title describes how the surface is displayed. We have set the `y_factor` keyword to 2.0 so that the surface will show in proper perspective; otherwise it would be stretched out from border to border in the vertical direction.

```
g1 = Graph3d (s1, color_card = "gray.gp",
   titles = "opaque wire mesh", y_factor = 2.)
g1.plot ()
paws ()
```



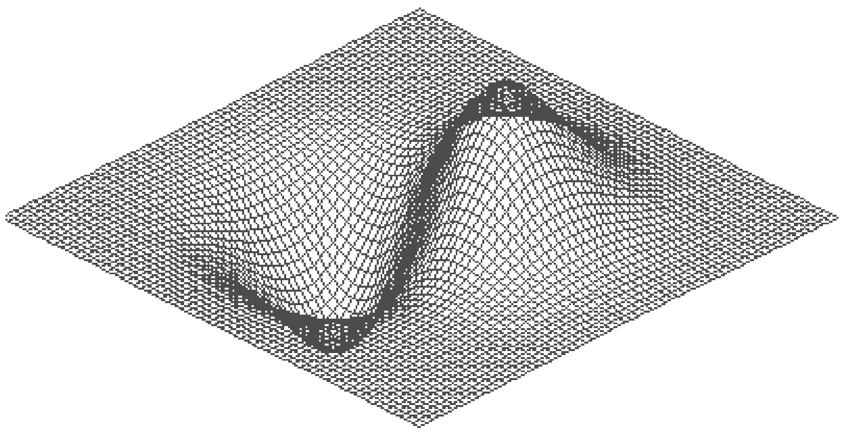
opaque wire mesh

```
s1.set (mask = "none")
g1.change (titles = "transparent wire mesh")
g1.plot ()
paws ()
```



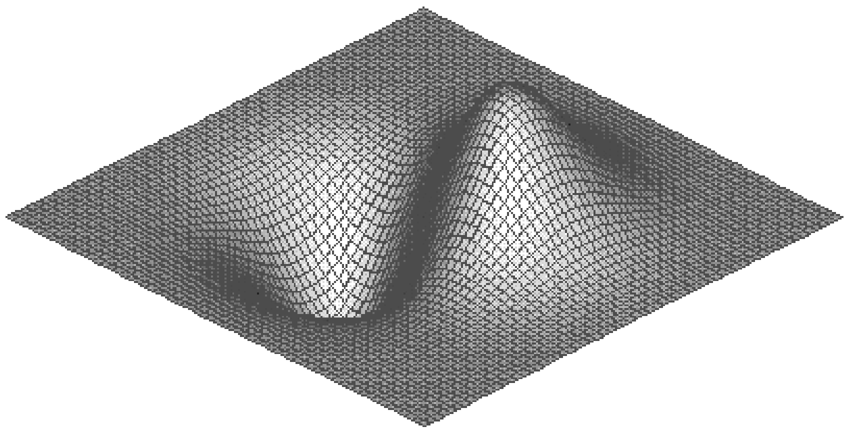
transparent wire mesh

```
s1.set (ecolor = "red")
g1.change (titles = "transparent wire mesh in red")
g1.plot ()
paws ()
```



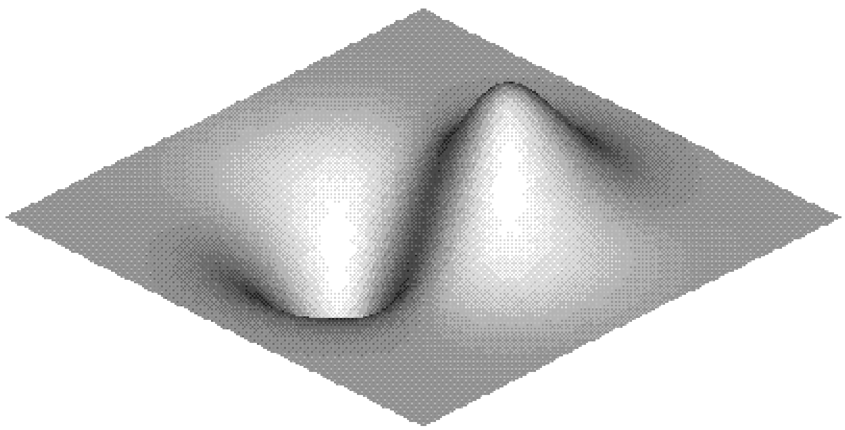
transparent wire mesh in red

```
s1.set (mask = "sort", shade = 1)
g1.change (titles = "opaque shaded mesh with red lines")
g1.plot ()
paws ()
```



opaque shaded mesh with red lines

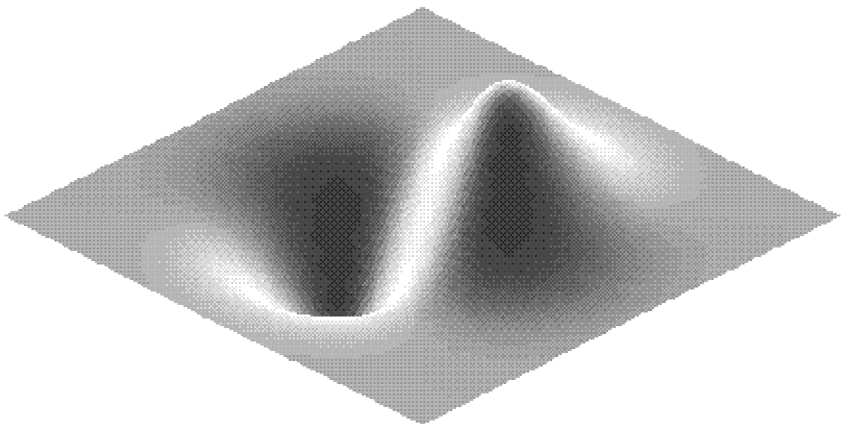
```
s1.set (opt_3d = "none")
g1.change (titles = "opaque shaded mesh with no lines")
g1.plot ()
paws ()
```



opaque shaded mesh with no lines

The next example is interesting in that it shows a back-lit surface.

```
g1.change (titles = "same with different lighting")
g1.quick_plot (diffuse=.1, specular = 1.,
   sdir=array([0,0,-1]))
paws ()
```



same with different lighting

Example 2. Plane cross sections of imploding sphere.

The user may recall this example. An imploding sphere has been decomposed into an unstructured (but hexahedral) mesh. The data is read in from a pdb file as follows:

```
f = PR ('./bills_plot')
n_nodes = f.NumNodes
n_z = f.NodesOnZones
x = f.XNodeCoords
y = f.YNodeCoords
z = f.ZNodeCoords
c = f.ZNodeVelocity
n_zones = f.NumZones
```

Now we build a `Mesh3d` object from the data:

```
m1 = Mesh3d (x = x, y = y, z = z, c = c, avs = 1,
   hex = [n_zones, n_z])
```

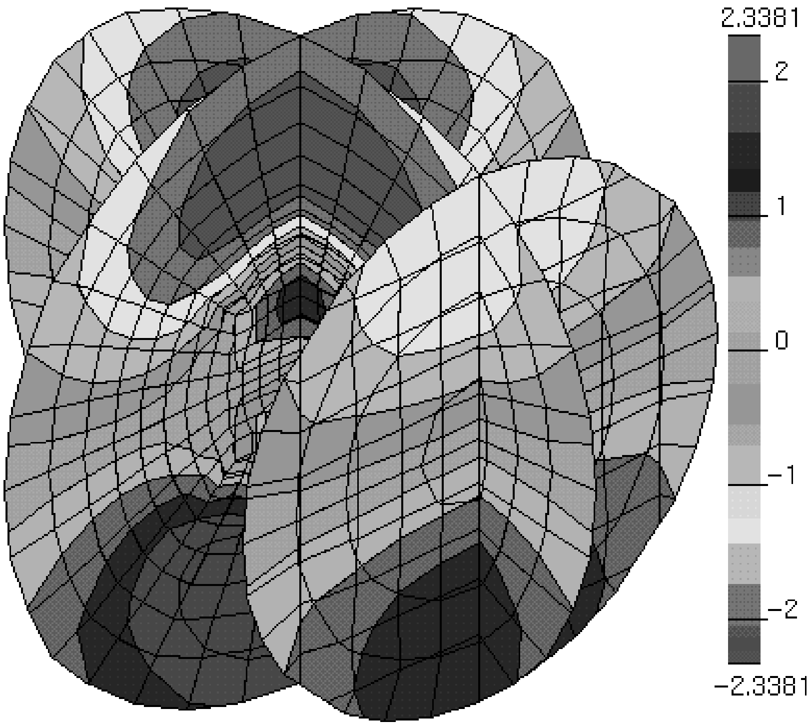
Create three `Plane` objects with which to perform cross sections:

```
pyz = Plane (array ([1., 0., 0.], Float ),
   array ( [0.0001, 0., 0.], Float))
pxz = Plane (array ([0., 1., 0.], Float ),
   array ( [0., 0.0001, 0.], Float))
p2 = Plane (array ([1., 0., 0.], Float ),
   array ( [0.35, 0., 0.], Float))
```

Slice the mesh three times:

```
s2 = sslice (m1, pyz, varno = 1, opt_3d = ["wm", "s4"])
s22 = sslice (m1, p2, varno = 1, opt_3d = ["wm", "s4"])
s23 = sslice (m1, pxz, varno = 1, opt_3d = ["wm", "s4"])

g1 = Graph3d( [s2, s22, s23], color_card = "rainbow.gp",
   opt_3d = ["wm", "s4"], mask = "min", color_bar = 1,
   split = 0, hardcopy = "talk.ps")
g1.plot ()
```

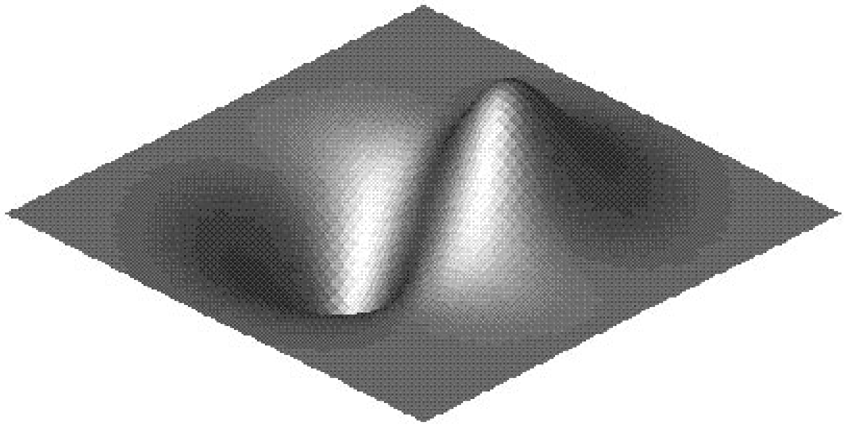
The resulting graph is shown below.

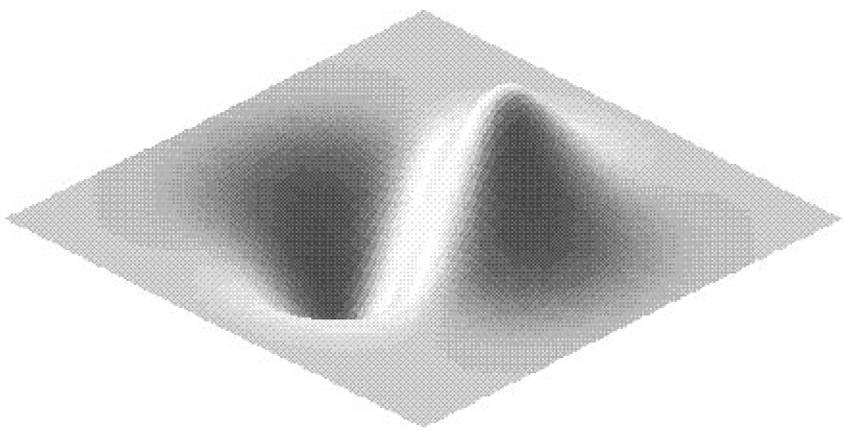


Example 3. Moving light source on surface.

In this example, we will illustrate how to set up a graph with a moving light source. The light source will apparently move over the surface in real time. You will have to take our word for this; the next two figures show different views of the surface as the light progresses.

```
s1 = Surface (z = z, opt_3d = "none", mask = "sort",
  shade = 1) # Same surface as Example 1

g1 = Graph3d (s1, ambient = 0.2, diffuse = .2, specular = 1.,
  color_card = "gray.gp", titles = "moving light source",
  y_factor = 2.)

g1.move_light_source ()
```



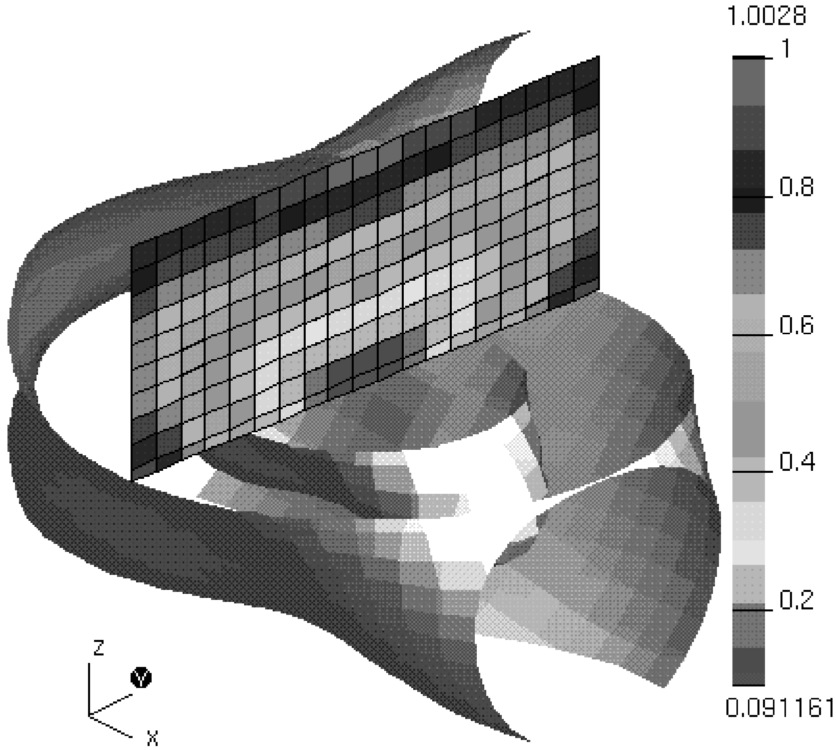


Imagine the light source as moving from right to left just behind the viewer.

Example 4. Rotating isosurfaces and cutting plane.

We cannot show you the actual rotation in these pages, but we shall show you a couple of different snapshots of the rotating surface. This example consists of a couple of isosurfaces in a mesh, each sliced horizontally and vertically, with parts discarded so that we can see inside the figure, and a portion of one of the slicing planes. The isosurfaces are shaded in greyscale as if by a light shining over the viewer's right shoulder, and the polygons of the portion of the slicing plane are colored using the rainbow palette by the values of the same function that was used to perform the isosurface slicing. This figure illustrates the so-called ‘‘split palette’’, where half of the palette is set to greyscale colors and is used to shade isosurfaces, while the other half is set to colors used to plot function values on plane slices.

The edges of the polygons on the plane slice are also shown. The figure and the code generating it begin below.



The following code computes the coordinates of the mesh, the function `c` defined on it, and then creates the `Mesh3d` object.

```
nx = 20
ny = 20
nz = 20
xyz = zeros ( (3, nx, ny, nz), Float)
xyz [0] = multiply.outer ( span (-1, 1, nx),
   ones ( (ny, nz), Float))
xyz [1] = multiply.outer ( ones (nx, Float),
   multiply.outer ( span (-1, 1, ny), ones (nz, Float)))
xyz [2] = multiply.outer ( ones ( (nx, ny), Float),
   span (-1, 1, nz))
r = sqrt (xyz [0] ** 2 + xyz [1] **2 + xyz [2] **2)
theta = arccos (xyz [2] / r)
phi = arctan2 (xyz [1] , xyz [0] + logical_not (r))
```

```
y32 = sin (theta) ** 2 * cos (theta) * cos (2 * phi)

m1 = Mesh3d (x = span (-1, 1, nx), y = span (-1, 1, ny),
             z = span (-1, 1, nz), c = r * (1. + y32))
```

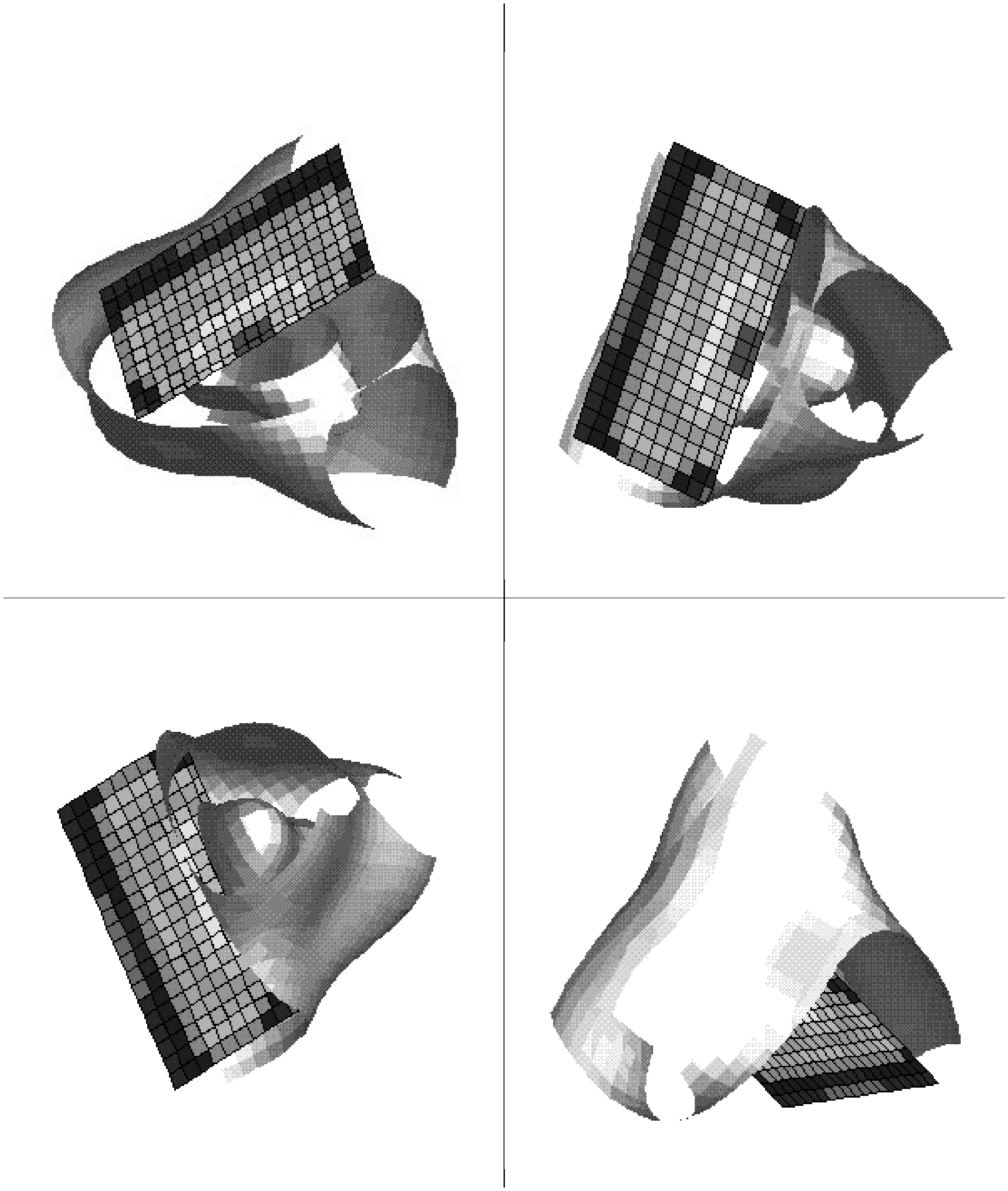
The following code sequence performs the slicing. We do not specify `opt_3d` for the isosurfaces, since with the split palette option they will automatically be shaded.

```
s1 = sslice (m1, .50, varno = 1) # (inner isosurface)
s2 = sslice (m1, 1.0, varno = 1) # (outer isosurface)

pxy = Plane (array ([0., 0., 1.], Float ), zeros (3, Float))
pyz = Plane (array ([1., 0., 0.], Float ), zeros (3, Float))

# create a pseudo-colored plane slice, then cut it in half
# and save only the front half. "f4" specifies that the
# cells be colored by the function assigned to the c
# keyword of the mesh m1. "wm" (wire monochrome) causes the
# edges of the cells to be shown.
s3 = sslice (m1, pyz, opt_3d = ["wm", "f4"])
s3 = sslice (s3, pxy, nslices = 1, opt_3d = ["wm", "f4"])

# cut the inner isosurface in half so that we can slice the
# top off one of the halves and discard it:
[s1, s4] = sslice (s1, pxy, nslices = 2)
# Note the use of - pyz to keep the "bottom" slice:
s1 = sslice (s1, - pyz)

# do the same with the outer isosurface:
[s2, s5] = sslice (s2, pxy, nslices = 2)
s2 = sslice (s2, - pyz)

# Create Graph object with split palette (rainbow/greyscale)
g1 = Graph3d ([s3, s1, s4, s2, s5], gnomon = 1,
   color_card = "rainbow.gp", diffuse = .2, specular = 1,
   mask = "min", split = 1)
g1.plot ()
```

The code which generates the rotating figure is given below. We change the `x_factor` and `y_factor` of `g1` so that the figure will appear smaller.

```
g1.change (x_factor = 2., y_factor = 2.)
g1.rotate ()
```

Snapshots of the rotating figure are shown on the next page.



CHAPTER 6: Animation2d Objects

An `Animation2d` object is a container for the controls for a two dimensional animation. The user supplies these controls, which are functions (written in Python) that initialize internal variables in the object, compute the coordinates for each frame, and update the internal variables. To see the animation performed, give the object to a `Graph2d` and ask the `Graph` to plot itself.

Currently `Animation2d` is not implemented in PyNarcisse.

Instantiation

```
from animation2d import *
anim = Animation2d ( <keylist>)
```

Description

`Animation2d` accepts the following keyword arguments:

`initialize, calculations, update, animation, nsteps, color`

It also has methods `new` and `set`, which work the same as the methods with the same names in other 2d objects. See “Description” on page 9., for instance.

Keyword Arguments

The following keyword arguments can be specified with `Animation2d`:

`initialize = <name of an initialization function>`. This function should have one argument, the name of an `Animation2d` instantiation, say ‘`anim`’, and when called should initialize any of `anim`’s internal variables needed before beginning to compute the animation.

`calculations = <calculation function(s) for coordinates>`: the value of this keyword is the name of a function, or a list of names of functions. Each of the calculations routines should have ‘`anim`’ as the argument. This routine (or these routines) are called from within a loop in the `Plotter`(s) associated with `anim`. They should compute the current values of `anim.x` and `anim.y`, the coordinates of the curve(s) in this step of the animation. The first frame starts with the results of `initialize`, then in subsequent calls, use the results of `update` (below). If more than one calculation is specified, then a plot command will be issued after each one.

`update = <function to update the variables used in calculations>`. This function, when called with ‘`anim`’ as its sole argument, updates (increments, decrements) variables used in calculating the frames.

`animation` = `0/1` (If `1`, supplies a smoother, less jerky animation. Default value `1`.)

`nsteps` = number of animation steps desired. Default: `100`.

`color` = `<value>` where `<value>` is an integer representing an index into a color chart, or a common color name like `"red"`, `"blue"`, `"background"`, etc. In the interest of speed, other keywords relating to curve `type`, `thickness`, etc., are currently not allowed.

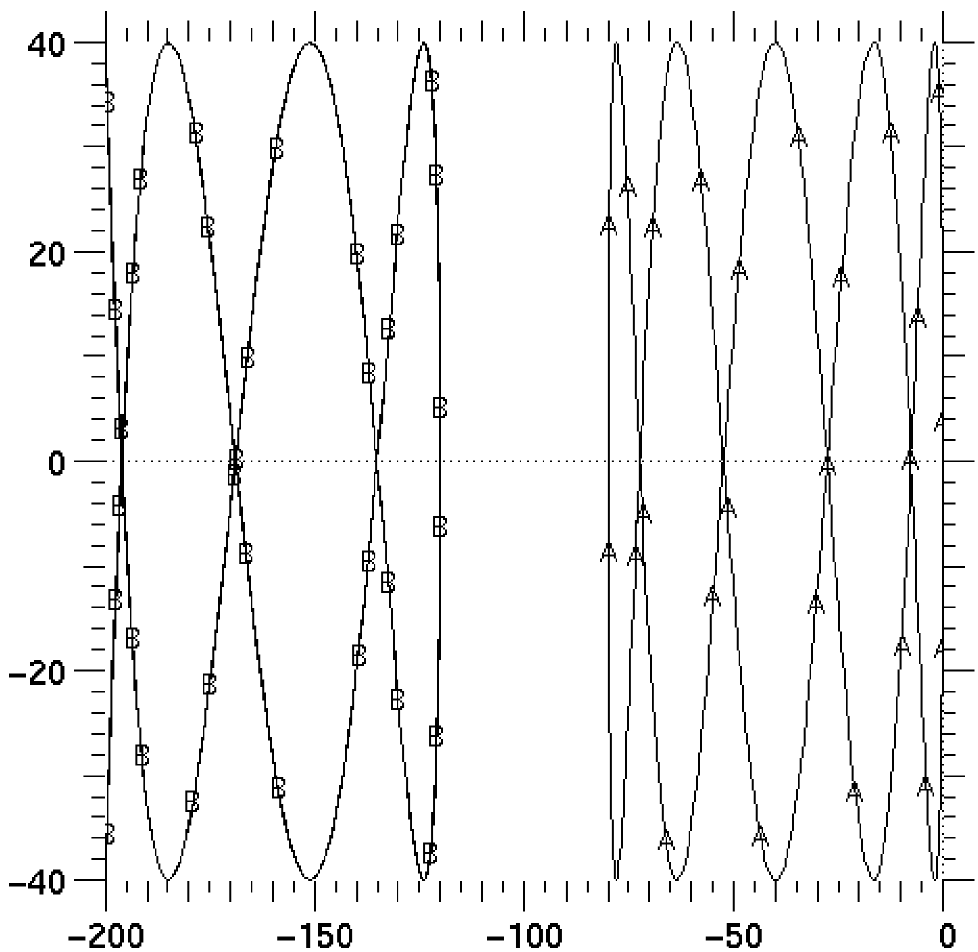
Examples

The following is an interesting example of ‘‘dancing curves’’, sine waves which appear to jump up and down and go around in circles.

```
def init ( self ) :
    self.t = 2*pi*arange (400, typecode = Float) / 399.0
    self.na1 = 1
    self.nb1 = 5
    self.na2 = 2
    self.nb2 = 7
    self.rc1 = 40.
    self.rc2 = 160.
    self.size = 40.
    self.phase = self.theta = 0.
    self.dtheta = pi / (self.nsteps - 1)
    self.dphase = 2 * pi / (self.nsteps - 1)
def calc1 ( self ) :
    self.cost = cos (self.theta)
    self.sint = sin (self.theta)
    self.x = self.rc1 * self.cost + \
       self.size * cos (self.na1 * self.t)
    self.y = self.rc1 * self.sint + \
       self.size * sin (self.nb1 * self.t + self.phase)
def calc2 ( self ) :
    self.x = self.rc2 * self.cost + \
       self.size * cos (self.na2 * self.t)
    self.y = self.rc2 * self.sint + \
       self.size * sin (self.nb2 * self.t + self.phase)
def incr  ( self ) :
    self.theta = self.theta + self.dtheta
    self.phase = self.phase + self.dphase

from animation2d import *
# instantiate an Animation2d without smoothness
anim = Animation2d ( initialize = init,
   calculations = [calc1, calc2], update = incr,
```

```
    animation = 0, nsteps = 200 )

g1 = Graph2d ( anim )
g1.plot ( )

# Now animate smoothly to see the difference.
anim.set (animation = 1)
g1.plot ( )
```

We have been unable to capture steps in the animation for this document; below is a pisture of the two curves after the animation has finished. You will have to try this example yourself to see the incredible effects!



CHAPTER 7: `Plotter`s: A Brief Primer

The purpose of this chapter is to give a quick and dirty introduction on how to instantiate a `Plotter` object and use it. It is currently not possible to induce a `Graph` object to create `Plotter`s of every conceivable type; the user who may not be satisfied with what is supplied can use this chapter to learn how to create `Plotter`s which can be passed as the values of keyword arguments to `Graph` objects upon instantiation.

In general we recommend against anybody using the full capability of `Plotter` objects who is not on the computer science team. They are a low-level interface and require a lot of work and knowledge of low-level graphics engine intrinsics on the part of the user. If there is some capability not currently offered by `Graph` objects, then rather than using a `Plotter`, I recommend that you contact a member of the computer science team to add the capability which you desire.

Instantiation

```
# Uncomment one of the following depending on which
# graphics you are going to use (or both if you want
# both kinds of plotter)
# For a Narcisse plotter use:
# import NarPlotter
# plN = NarPlotter.Plotter ( [ <filename>] [, <keylist>])
# For a Gist plotter use:
# import GistPlotter
# plG = GistPlotter.Plotter ( [ <filename>] [, <keylist>])
```

Description

The only argument to instantiate a PyNarcisse `Plotter` is *`<filename>`*; it is also the first argument to instantiate a PyGist `Plotter`. *`<filename>`* is a string which specifies plotting associated with a particular filename. PyGist and PyNarcisse differ dramatically in the meaning of this argument, as explained below.

PyGist: *`<filename>`* specifies a host where the PyGist window is to be displayed (e. g., `"icf.llnl.gov:0.0"`). If the argument *`<filename>`* is missing or if the user specifies `" "` (a single blank) as the *`<filename>`*, then PyGist will attempt to obtain the user's `DISPLAY` environment variable; no window will be opened if this variable is undefined. Likewise, no window will be opened if the user specifies `""` (a blank), `"none"`, or `None` as the *`<filename>`*. If the user wants a `Plotter` which plots to a given CGM or PostScript file (or one of each), then the user must instantiate one or more `Plotter` objects using the keyword argument `hcp` (described below) and hand it to a `Graph` via the `plotter` keyword. Eventually this will be changed to make it easier on the user to ask the `Graph` to plot to a file only.

PyNarcisse: *<filename>* can be used in two different ways.

(1) As a way to specify a Narcisse process to connect to. In this case, the *<filename>* should be in the form `"machine+port_serveur++user@ie.32"` where `machine` specifies where the display is to take place (e. g., `"icf.llnl.gov:0.0"`), `port_serveur` is the port number displayed on the Narcisse GUI, and `user` is the userid of the person running.

(2) As a filename where Narcisse is to dump its plots. In this case, use the file suffix `".spx"` to specify a binary file, or `".spc"` for an ascii dump file. If a filename of this form is specified, PyNarcisse attempts a connection to Narcisse using the value of the `DEST_SP3` environment variable, if it exists, and if not, attempts to construct a connection filename using `DISPLAY` environment variable for `machine`, the value of the `PORT_SERVEUR` environment variable (or the default 2101 if `PORT_SERVEUR` is not defined) for `port_serveur`, and the value of `USER` for `user`.

Keyword Arguments

Currently only PyGist `Plotter`s accept keyword arguments; these arguments (with their default values, if not specified) are as follows:

`n (0)` -- the number of the graphics window (`0` to `7` are allowed). each `Plotter` object corresponds to a separate window.

`dpi (100` for 2d; `75` for 3d.`)` -- the size of the window wanted. `100` and `75` are allowed; `100` is the larger size. This does not affect the size of hardcopy plots.

`wait (1)` -- used to make sure everything is plotted before changing frames.

`private (0)` -- use a common colormap (palette) for all windows.

`hcp` -- if not present, or if set to `""`, there will be no hardcopy file. If present, names a file unique to this window. This will be PostScript if the *<filename>* ends in `".ps"` and CGM if the *<filename>* ends in `".cgm"`. Note that if both *<filename>* and `hcp` are `""`, then you will have a `Plotter` with no window and no file, a circumstance of doubtful utility.

`dump (0)` -- if `1`, dumps the color palette at the beginning of each page of hardcopy output, otherwise converts to grey scale.

`legends (0)` -- controls whether (`1`) or not (`0`) curve legends are dumped to the hardcopy.

`style ("work.gs"` for 2d; `"nobox.gs"` for 3d.`)` -- name of a Gist style sheet.

Example

The following will create a plotter with a window and a hardcopy file for color plots called `talk.ps`:

```
pl = GistPlotter.Plotter (" ", hcp = "talk.ps", dump = 1, dpi = 75)
```

Index

A

B

C

D

E

F

K

L

M

N

O

P

Q

R

X

Y

Z