PYGIST

Python Interface to Gist Graphics

Abstract

This document describes usage of the PYGIST and GIST3D Python modules. PYGIST enables users to access Yorick graphics functions from the Python scripting language. This module provides 2D plotting capability. The separate GIST3D modules provides a set of Python functions for interpreted 3D graphics.

Contents

1 Introduction

1.1   History

This documentation is a revision of the original documentation[1] with a new title.

The Python Gist Scientific Graphics Package, written by Lee Busby and Zane Motteler of Lawrence Livermore National Laboratory, is a set of Python modules for production of general scientific graphics. The Python Gist module `gist.py` and the associated Python extension `gistCmodule.c` provide a Python interface to the Gist library and is referred to as 'PyGist.'

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: X-Windows, (color) Postscript, and ANSI/ISO Standard Compute Graphics Metafile (CGM). The library is small (written directly to Xlib), portable, efficient, and full-featured. It produces x-y plots with "good" tick marks and tick labels, 2D quadrilateral mesh plots with contours, filled contours, vector fields, or pseudocolor maps on such meshes, and a selection of 3D plots, including 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 3D library is packaged separately as the 'Gist3D' module.

The original Python Gist module utilized the "Numerical" package due to J. Hugunin and others. PyGist is therefore fast and able to handle large datasets. The Gist module includes an X-windows event dispatcher which can be dynamically added to the Python interpreter. This makes fast mouse-controlled zoom, pan, and other graphic operations available to the researcher while maintaining the usual Python command-line interface.

1.2   The 1.5 Update

PyGist was unmaintained for a period of time, during which the three dependent components changed significantly:

- Yorick 1.3 evolved to Yorick 1.5 with major changes to the graphics and event handling.

- Python 1.x evolved to Python 2.x.

- Numerical package has also evolved under Paul Dubois.

The major issues addressed in bringing up PyGist 1.5 were the following:

- Upgrading the C extension to be compatible with Yorick version 1.5.

- Porting the code system to wider set of platforms of current interest: including Solaris, IRIX64, OSF/1, AIX, Linux, Cygwin, and Windows.

- Getting these two products (Python and Yorick) that support event handling to cooperate in this area.

- Simplifying the distribution of PyGist by:

  - Including the required portion of Yorick in the PyGist source.
  - Packaging the software using Python DistUtils.

1.3   Platforms

PyGist 1.5.x runs on the following platforms:

- AIX

- Intel Linux [1]

- alpha Linux

- Windows [2]

- Cygwin [3]

- alpha OSF/1

- Solaris

- IRIX64 (one issue)

---

[1] Michiel de Hoon has tested PyGist on i386, and Lila Chase on Pentium4 SMP (i686).

[2] Michiel de Hoon has tested PyGist on the following: DOS command prompt, Python for Windows, PyShell, PyCrust, PythonWin.

[3] Michiel de Hoon has tested PyGist on Cygwin with or without X11.

1.4 Current Maintenance

The following people are currently involved in the maintenance of PyGist:

- Dave Munro, LLNL: Gist, event handling interface

- Michiel de Hoon, University of Tokyo: Cygwin and Windows ports, build facility

- Dave Grote, LBNL: contributed bug fixes

- Lila Chase, LLNL: porting, testing, documentation

2 PyGist Installation

In general, the installation consists of running the following two steps:

```
python setup.py config
python setup.py install
```

This works on Linux, Unix, Mac OS X, Cygwin, and Windows (for the latter two, see below). Some Python versions older than 2.3 contain bugs in the distutils package, and may therefore need a different installation procedure:

```
cd src; make clean; make config; cd ..
python setup.py install
```

instead.

Alternatively, use the script provided:

```
./install      (for installation in a private directory)
./llnl_install (for installation in a public LLNL directory)
```

2.1 Windows

For use on Windows (from the DOS prompt or using a GUI such as IDLE, PythonWin, PyCrust, or PyShell), use the Windows installer. It is available at `http://bonsai.ims.u-tokyo.ac.jp/~mdehoon`.

If you want to recompile Pygist for Windows from source, you can do so using Cygwin/MinGW. To compile, run

```
/cygdrive/c/Python22/python setup.py config --compiler=mingw32
/cygdrive/c/Python22/python setup.py build --compiler=mingw32
```

Here, `/cygdrive/c/Python22/python` refers to Windows's Python as seen from Cygwin. The exact path may be different on your machine. To install Pygist for Python on Windows, run

```
/cygdrive/c/Python22/python setup.py install
```

If instead you want to create the Window installer, run

```
/cygdrive/c/Python22/python setup.py bdist_wininst
```

2.2 Cygwin

For Cygwin without XFree86, the usual installation procedure is used:

```
python setup.py config
python setup.py install
```

If you want to use Cygwin with the XFree86 implementation of the X Windows System, you need to add `--x11` to these commands:

```
python setup.py config --x11
python setup.py install --x11
```

For Python versions older than 2.3, for the config step it is necessary to specify the Cygwin compiler explicitly by adding `--compiler=cygwin`:

```
python setup.py config --compiler=cygwin
```

or

```
python setup.py config --compiler=cygwin --x11
```

The install command does not change.

3 User Setup and Demo

At LLNL, the public version of python is installed in `/usr/apps/python/opt/bin` on both open and closed Livermore Computing computers and on the local AX-net computers. Ensure that `python` is in your path; if it is not, add:

```
set path = ( /usr/apps/python/opt/bin $path )
```

to your `.cshrc` if your are a CSH user, and `source` the file.

There is a demonstration Python script installed in `${PYTHONHOME}/lib/python2.2/site-packages/gist/gistdemolow.py`, where `PYTHONHOME` is where your Python is installed; at LLNL, it is `/usr/apps/python/opt`.

To run the demo, type:

```
python
import gistdemolow
gistdemolow.run()
(ctrl-d)
```

3.1 GISTPATH

Gist has style files (`*.gs`) and palette files (`*.gp`), typically installed in a subdirectory `g` where Python is installed, and the user usually does not need to do anything else to access these files.

However, Zope users and others may require an extra setup step. For example, because Zope's Python is installed in a non-standard location, PyGist has problems finding these files. To solve this problem, set the environment variable GISTPATH to the correct location of these files.

4 Python Gist Graphics

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: X Windows, (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 labels, 2D quadrilateral mesh plots with contours, filled contours, vector fields, or pseudocolor maps on such meshes. Some 3D plot capabilities are also available. The Python Gist module gist. py and the Python extension gistCmodule provide a low-level Python interface to this library as far as 2D is concerned. In addition, there are several other Python modules which interface with the 2D graphics to produce 3D graphics and animation: movie.py (supporting animation), pl3d.py (basic 3D plotting algorithms), plwf.py (wire frame plotting), and slice3.py (providing mesh capability with isosurface and plane slicing). Collectively all of these interface modules are known as PyGist.

This chapter will summarize the plotting features that are available in PyGist, and list (in the final section) the functions that are to be described in future chapters.

4.1   PyGist 2D Graphics

In two dimensions, PyGist supplies functions to plot curves, meshes (with various combinations of contours, filled mesh cells, and vector fields on the mesh, with color-filled contours in the future), sets of filled polygons, cell arrays, sets of disjoint lines, text strings, and a title. These are all provided by the Python module gist.py.

We will show a couple of simple examples below to give the reader a flavor of the interface.

4.1.1   Example 1

In the first example we simply plot a straight line from $(1, 0)$ to $(2, 1)$. Note that only two coordinates are specified for y; x is not specified. In such a case, the values of x default to the integers from 1 to len (y).

```
from gist import *    # Put plot functions in name space.
pldefault (marks = 1, width = 0, type = 1, style = "work.gs",
```

```
          dpi = 100) # Set some defaults.
winkill (0)                 # Kill any existing window.
window (0, wait = 1, dpi = 75)
plg ([0, 1])                # The first positional argument is y.
```
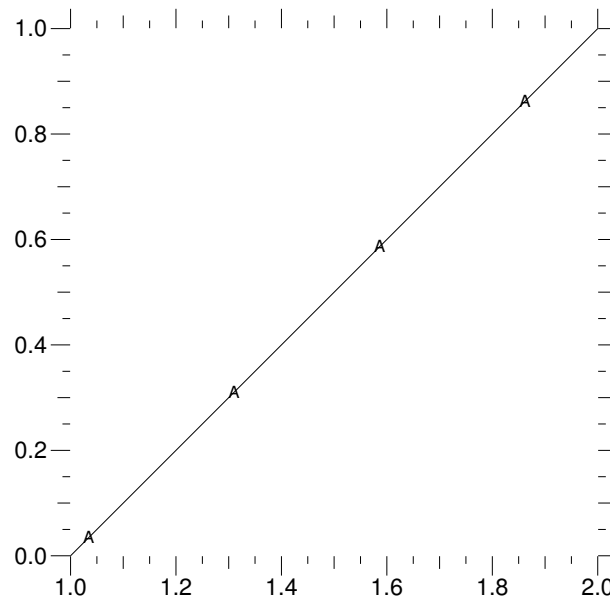

System 0

The first positional argument in PLG is y.

As can be deduced from this example, most PyGist function calls can be augmented with a number of optional keyword arguments. These can (usually) be supplied in any order, and each is of the form keyword= value. Throughout this manual, a list of the available keywords for a function is given with the description of the function.

4.1.2   Example 2

The next example computes and plots a set of nested cardioids in the primary and secondary colors.

```
fma()
x = 2 * pi * arange (200, typecode = Float) / 199.0
for i in range (1, 7):
```

```
r = 0.5 * i -(5 -0.5 * i) * cos (x)
s = 'curve ' + `i` #Backticks produce something printable.
plg (r * sin (x), r * cos (x), marks = 0, color = -4 -i,
legend = s) # Curves unmarked, in colors.
```
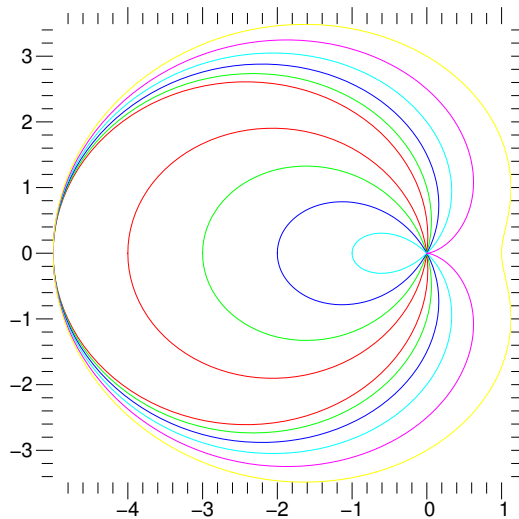


```
curve [1]
curve [2]
curve [3]
curve [4]
curve [5]
curve [6]
```

4.2   PyGist 3D Graphics

4.2.1   General overview of module pl3d

The Python module pl3d.py contains the basic 3D plotting algorithms and is the workhorse of the
PyGist 3D graphics. The philosophy behind 3D plotting is to instruct the 3D plotting functions to
accumulate information about the plot until such time as the information is complete, and then ask
that the picture be drawn. The information about the plot is stored in a Python list containing the
following information:

The orientation of the axes, the location of the origin, and the distance of the viewpoint; A set of
pairs of plot functions to call and their argument lists; and A collection of one or more quintuples
specifying the lighting (it is possible to specify multiple light sources). The first and third items
above default to reasonable values if the user does not call functions (e. g., `rot3`, `mov3`, `aim3`,

`set3_ light`., etc.) to set them. The list described in the second bullet is built by a set of one or more calls to the various plotting functions, which create the list of arguments for each call and then add the function name and argument list pair to the plot list for future execution. When the list is complete, a call to draw3 causes the list to be traversed, and at this point each plotting function on the list executes with the argument list that was built when it was first called.

4.2.2   Overview of module plwf

The main function of interest in plwf.py is the function plwf ("plot wire frame"), which enables the user to plot an arbitrary wire frame on a quadrilateral grid. The grid may be see-through or not (cells filled with the background color). In the latter case, the drawing order of the zones is determined by a simple "painter's algorithm", which works fairly well if the mesh is reasonably nearly rectilinear, but can fail even then if the viewpoint is chosen to produce extreme fisheye perspective effects. One must look at the resulting plot carefully to be sure the algorithm has correctly rendered the model in each case.

A 3D wire mesh can also be plotted using shading and lighting effects as determined by values set in the pl3d module; or the zones can be colored (using the current palette) by their average height or by the values of some function, which may be zone-centered or node-centered.

Examples

The following is a fairly simple example of a wire mesh plot.

```
from pl3d import *
set_draw3_(0)
x = span (-1, 1, 64, 64)
y = transpose (x)
z = (x + y) * exp (-6.*(x* x+ y* y))
orient3 ()
light3 ()
from plwf import *
plwf (z, y, x)
[xmin, xmax, ymin, ymax] = draw3(1)
limits (xmin, xmax, ymin, ymax)
```

Calling `set_draw3_` with argument zero tells the 3d plotting routines not to draw the graph until asked (by a call to draw3). orient3 and light3 set the orientation and lighting parameters to

default values when called with no arguments. (light3 is irrelevant for this durface, since it is not shaded.) The plwf call puts this surface on the drawing list (plwf = "plot wire frame.") The draw3 call then causes the drawing list to be plotted. draw3 returns the maxima and minima of the x and y variables, which must then be sent to the limits function to prevent the plot appearing distorted. (Ah, the perils of using low level graphics.)

4.2.3   Overview of module slice3

Module slice3.py contains two plotting functions of interest. First, pl3surf can be used for graphing surfaces on an arbitrary two-dimensional mesh with filled cells and no mesh lines. (Currently plwf can be used to do the same thing in the case of a mesh all of whose cells are quadrilateral, and has more flexibility, in that it allows mesh lines to be drawn and/ or allows for the mesh to be see-through.) Secondly, pl3tree is a plotting function that can be called multiple times in order to have several surfaces drawn on the same graph. pl3tree (as its name suggests) creates a tree of values sorted as to when they will be plotted on the screen; if the algorithm works correctly, then more distant cells are plotted first, then covered by closer cells which are plotted later, giving the surface the correct appearance.

Surfaces to be plotted by pl3surf or pl3tree can be generated by taking plane sections of an arbitrary mesh or by creating isosurfaces for some function or functions defined on the mesh. These planes and isosurfaces can themselves be sliced and portions discarded, to enhance visibility of the interior. The functions mesh3 and slice3mesh take raw input data and put it into the form accepted by slice3, which can form plane sections or isosurfaces through the mesh. Functions slice2 (which returns the portion of a surface in front of the slicing plane) and slice2x (which returns the two parts of a surface sliced by a plane) complete the triumvirate of slicing functions.

The algorithms in slice3 are independent of the underlying graphics. Thus slice3 may equally well be used with Narcisse graphics.

4.3   movie.py: PyGist 3D Animation

The module movie.py supports 3D real time animation. Function movie accepts as argument the name of a drawing function which has as its single argument a frame number; movie then calls this drawing function within a loop, halting when the function returns zero. The idea is that the drawing function increments from the previous frame and draws the new frame, returning zero when some predefined event takes place, e. g., some set number of frames has been drawn, or a certain amount of time has elapsed. The function `spin3` in module `pl3d` calls `movie`; the drawing function `_spin3` draws the successive frames of a rotating 3D plot. The demonstration

module demo5.py contains an example of a shaded surface with a moving light source; the drawing function, `demo5_light`, moves the light and draws the next frame.

4.3.1   Examples

The following example is explained by comments in the code. It is taken from demo5.py. (To repeat, `demo5_light` is a function which appears in demo5.py.)

```
# First we define the mesh and functions on it.
# (Note: nx == ny == 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)
# mesh3 creates an object which slice3 can slice. The
# isosurfaces will be with respect to constant values
# of the function r * (1. + y32)].
m3 = mesh3 (xyz, funcs = [r * (1. + y32)])
[nv, xyzv, dum] = slice3 (m3, 1, None, None, value = .50)
# (inner isosurface)
[nw, xyzw, dum] = slice3 (m3, 1, None, None, value = 1.)
# (outer isosurface)
pxy = plane3 (array ([ 0, 0, 1], Float), zeros (3, Float))
pyz = plane3 (array ([ 1, 0, 0], Float), zeros (3, Float))
[np, xyzp, vp] = slice3 (m3, pyz, None, None, 1)
# (pseudo-colored plane slice)
[np, xyzp, vp] = slice2 (pxy, np, xyzp, vp)
# (cut slice in half, discard "back" part)
[nv, xyzv, d1, nvb, xyzvb, d2] = \
slice2x (pxy, nv, xyzv, None) # halve inner isosurface
[nv, xyzv, d1] = \
slice2 (-pyz, nv, xyzv, None)
# (... halve one of those halves)
```

```
[nw, xyzw, d1, nwb, xyzwb, d2] = \
slice2x (pxy , nw, xyzw, None)
# (split outer isosurface in halves)
[nw, xyzw, d1] = \
slice2 (-pyz, nw, xyzw, None) # discard half of one half
fma () # frame advance
# split_palette causes isosurfaces to be shaded in grey
# scale, plane sections to be colored by function values
split_palette (" earth.gp")
gnomon (1) # show small set of axes
clear3 () # clears drawing list
set_draw3_(0) # Make sure we don't draw till ready
# Create a tree of objects and put on drawing list
pl3tree (np, xyzp, vp, pyz)
pl3tree (nvb, xyzvb)
pl3tree (nwb, xyzwb)
pl3tree (nv, xyzv)
pl3tree (nw, xyzw)
orient3 ()
# set lighting parameters for isosurfaces
light3 (diffuse = .2, specular = 1)
limits (square= 1)
demo5_light (1) # Causes drawing to appear
```

demo5.py also contains code which rotates the above object in real-time animation. It is not possible to illustrate that here.

4.4   Function Summary

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

Control functions See page 20.

```
    window ([n] [, <keylist>]) # open or select device n
        keywords: display, dpi, dump, hcp, legends, private, style, wait

    winkill ([n]) # delete device n

    n = current_window () # determine active device
```

```
fma () # frame advance
```

Plot limits and scaling See page 29.

```
old_limits = limits ()

old_limits = limits (xmin [, xmax[, ymin[, ymax]]]
                        [, <keylist>])
   keywords: square, nice, restrict

limits (old_limits)

ylimits (ymin[, ymax])

logxy (xflag[, yflag])

gridxy (flag)

gridxy (xflag, yflag)

zoom_factor (factor)

unzoom ()
```

Two-dimensional plotting functions See page 33.

```
plg (y [, x][, <keylist>]) # plot a graph
   keywords: legend, hide, type, width, color, closed,
             smooth, marks, marker, mspace, mphase, rays

plmesh ([ y, x][, ireg][, triangle= tri_array])
# set default mesh

plmesh () # delete default mesh

plm ([ y, x][, ireg][, <keylist>])
# plot mesh
   keywords: boundary, inhibit, legend, hide, type, width,
             color, region

plc (z[, y, x][, ireg][, <keylist>])
```

```
# plot contours
   keywords: levs, triangle, legend, hide, type, width,
             color, smooth, marks, marker, mspace, mphase,
             region

plfc (z[, y, x][, ireg][, <keylist>])
# plot filled contours
   keywords: contours, colors, region, triangle, scale

plv (vy, vx[, y, x][, ireg][, <keylist>])
# plot vector field
   keywords: scale, hollow, aspect, legend, hide, type,
             width, color, smooth, marks, marker, mspace,
             mphase, triangle, region

plf (z[, y, x][, ireg][, <keylist>])
# Plot a filled mesh
   keywords: edges, ecolor, ewidth, legend, hide, region,
             top, cmin, cmax

plfp (z, y, x, n[, <keylist>])
# Plot filled polygons
   keywords: legend, hide, top, cmin, cmax

pli (z[[, x0, y0], x1, y1][, <keylist>])
# Plot a cell array
   keywords: legend, hide, top, cmin, cmax

pldj (x0, y0, x1, y1[, <keylist>])
# Plot disjoint lines
   keywords: legend, hide, type, width, color

plt (text, x, y[, <keylist>])
   keywords: tosys, font, height, opaque, path, justify,
             legend, hide, color

pltitle (title) # Plot a title
```

Inquiry and Miscellaneous functions See page 53.

```
plq () # Query plot element status
```

```
legend_list = plq ()
**** RETURN VALUE NOT YET IMPLEMENTED ****

plq (n_element[, n_contour])

properties = plq (n_element[, n_contour])

pledit ([ n_element[, n_contour],] <keylist>)
# Change Plotting Properties of Current Element
```
The keywords can be any of the keywords that apply to the current element.

```
pldefault (key1= value1, key2= value2, ...)
# Set default values
```
The keywords can be most of the keywords that can be passed to the plotting
commands.
Also, one can specify a larger CGM file size:
```
pldefault (cgmfilesize=20)  # default is 10 MB; reset to 20 MB

bytscl (z[, top= max_byte][, cmin= lower_cutoff]
[, cmax= upper_cutoff])
# Convert data to color array

histeq_scale (z[, top= top_value][, cmin= cmin][,
cmax= cmax]) **** NOT YET IMPLEMENTED ****
# Histogram Equalized Scaling

mesh_loc (y0, x0[, y, x[, ireg]])
# Get zone index of (x0, y0)

result = mouse (system, style, prompt)
# Handle Mouse Click

moush ([ y, x[, ireg]])
# Return zone index of point clicked in mesh

pause (milliseconds) # self-explanatory
```

Three-dimensional plotting functions See page 59.

```
orient3 (phi = angle 1 , theta = angle 2)

rot3 (xa = angle x , ya = angle y , za = angle z)
```

```
mov3 (xa = val 1 , ya = val 2 , za = val 3)

aim3 (xa = val 1 , ya = val 2 , za = val 3)

light3 (ambient= a_level, diffuse= d_level,
        specular= s_level, spower= n, sdir= xyz)

clear3 ()

window3 ([n] [, dump = val] [, hcp = filename])

gnomon ([onoff] [, chr = <labels>])

set_default_gnomon ([onoff])

[lims = ] draw3 ([called_as_idler = <val>])

limits (lims [0], lims [1], lims [2], lims [3])

set_draw3_ (n)

n = get_draw3_ ()

clear_idler ()

set_idler (func_name)

set_default_idler ()

call_idler ()

plane3 (< normal>, <point>)

mesh3 (x, y, z)

mesh3 (x, y, z, funcs = [f1, f2, ...], [verts = <spec>])

mesh3 (xyz, funcs = [f1, f2, ...])

mesh3 (nxnynz, dxdydz, x0y0z0, funcs = [f1, f2, ...])
```

```
slice3mesh (z [, color])

slice3mesh (nxny, dxdy, x0y0, z [, color])

slice3mesh (x, y, z [, color])

slice3 (m3, fslice, nv, xyzv [, fcolor [, flg 1 ]]
        [, value = <val>] [, node = flg 2 ])
        [nverts, xyzverts, values] = slice2 (plane, nv, xyzv, vals)
        [nverts, xyzverts, values, nvertb, xyzvertb, valueb] =

slice2x (plane, nv, xyzv, vals)

plwf (z [, y, x] [, <keylist>])
   keywords: fill, shade, edges, ecolor, ewidth, cull, scale, cmax

pl3surf (nverts, xyzverts [, values] [, <keylist>])
   keywords: cmin, cmax

pl3tree (nverts, xyzverts [, values] [, <keylist>])
   keywords: plane, cmin, cmax, split
```

Animation functions See page 80.

```
movie (draw_frame [, time_limit = 120.]
       [, min_interframe = 0.0]
       [, bracket_time = array ([ 2., 2.], Float)]
       [, lims = None]
       [, timing = 0])

spin3 (nframes = 30,
       axis = array ([-1, 1, 0], Float),
       tlimit = 60.,
       dtmin = 0.0,
       bracket_time = array ([ 2., 2.], Float),
       lims = None,
       timing = 0,
       angle = 2. * pi)
```

Syntactic Sugar See page 83.

```
split_palette ([palette_name])
```

```
view = save3 ()

restore3 (view)
```

5 Control Functions

This chapter contains all the information you need to control PyGist devices. Device refers to an X Window or a hard copy file. In addition, we describe functions which control some aspects of the appearance of the graph.

5.1 Device Control

5.1.1 Window Control

Calling Sequences ...

```
window ([n] [, <keylist>])
winkill ([n])
n = current_window ()
fma()
```

Description The window function selects device n as the current graphics device. n may range from 0 to 7, inclusive. Each graphics device corresponds to an X window, a hardcopy file, or both, depending on the values of the keyword arguments described below. If n is omitted, it defaults to the current active device, if any. window returns the number of the currently active device. winkill deletes the current graphics device, or device n if n is specified. `current_window` returns the number of the current active device, or -1 if there is none. fma frame advances the current graphics device. The current picture remains displayed in the associated X window (if any) until the next element is actually plotted. An fma must be given after the last plot to a hardcopy file for that plot to appear when the file is printed.

The keywords accepted by the window function are

```
display, dpi, dump, hcp, legends, private, style, wait
```

and are described in the next subsection.

Keyword Arguments The following keyword arguments can be specified with this function. display A string of the form `host: server. screen` which tells where the X window will appear (for example, `icf.llnl.gov: 0.0`). If not specified, uses your default display (which it gets from your DISPLAY environment variable). Use `display = ""` (the null string) to create a graphics device which has no associated X window. (You should do this if you want to make plots in a non-interactive batch mode.)

dpi The allowed values for dpi are 75 and 100. The X window will appear on your default display at 75 dpi, unless you specify the display and/ or dpi keywords. A dpi = 100 X window is larger than a dpi = 75 X window; both represent the same thing on paper.

dump The dump keyword, if present, controls whether all colors are converted to a gray scale (dump = 0, the default), or the current palette is dumped at the beginning of each page of hardcopy output. Set dump to 1 if you are doing color plots. The dump keyword applies only to the specific hardcopy file defined using the hcp keyword (see below)—use the dump keyword in the `hcp_file` command to get the same effect in the default hardcopy file.

hcp The value of this keyword is a quoted string giving a file name. By default, a graphics window does NOT have a hardcopy file of its own—any requests for hardcopy are directed to the default hardcopy file, so hardcopy output from any window goes to a single file. By specifying the hcp keyword, however, a hardcopy file unique to this window will be created. If the hcp filename ends in `.ps`, then the hardcopy file will be a PostScript file; otherwise, hardcopy files are in binary CGM format. Use `hcp = ""` (the null string) to revert to the default hardcopy file (closing the window specific file, if any).

In the next section of this manual we shall consider the hardcopy and file functions. Note that the PyGist default is to write to a hardcopy file only on demand. (See function hcp on page 26)

legends The legends keyword, if present, controls whether the curve legends are (legends = 1, the default) or are not (legends = 0) dumped to the hardcopy file. The legends keyword applies to all pictures dumped to hardcopy from this graphics window. Legends are never plotted to the X window.

private By default, an X window will attempt to use shared colors, which permits several PyGist graphics windows (including windows from multiple instances of Python) to use a common palette. You can force an X window to post its own colormap (set its colormap attribute) with the private = 1 keyword. You will most likely have to fiddle with your window manager to understand how it handles colormap focus if you do this. Use private = 0 to return to shared colors.

style The style keyword, if present, specifies (as a quoted string) the name of a Gist stylesheet file; the default is `work.gs`. The style sheet determines the number and location of coordinate systems, tick and label styles, and the like. Here are brief descriptions of the available stylesheets:

axes.gs axes with labeled tick marks along bottom and left of graph. boxed.gs: lines all the way around the plot with tick marks, labeled along bottom and left.

boxed2.gs same as boxed.gs but no tick marks on the top and right sides.

l_nobox.gs no box, axes, or ticks; graph extends all the way to edge of window.

nobox.gs indistinguishable from `l_nobox.gs`. vg.gs: large tick marks all the way around the graph, but no lines, with large infrequent labels on the bottom and left.

vgbox.gs same as vg.gs except with lines all the way around as well

work.gs small tick marks with small, frequent labels on bottom and left, no lines.

work2.gs similar to work.gs, but no ticks along top and right. wait By default, Python will not wait for the X window to become visible. Code which creates a new window, then plots a series of frames to that window should use wait = 1 to assure that all frames are actually plotted.

Examples The first example ensures that an old window 0 is not hanging around, and then creates a new 100 dpi window.

```
winkill(0)
window (0, wait = 1, dpi = 100)
```

The second example changes the style sheet of window 2.

```
window (2, style = "vgbox.gs")
```

5.1.2 Changing the Window Style Interactively

Calling Sequences ...

```
style = get_style()
set_style(style)
```

Description The function `get_style()` returns a set of nested dictionaries that contain the style information that is currently used by the active window. The dictionary has the following structure:

```
style
```
    `['landscape']`: Set orientation to portrait (0) or landscape (1)
    `['legend']`:
      `['x']`: NDC horizontal location of the legend box
      `['y']`: NDC vertical location of the legend box
      `['dx']`: NDC horizontal offset to 2nd column
      `['dy']`: NDC vertical offset to 2nd column
      `['nchars']`: Maximum number of characters on a line
      `['nlines']`: Maximum number of lines
      `['nwrap']`: Maximum number of lines to wrap long legends
      `['textStyle']`:
        `['height']`: Character height in NDC, default 0.0156 (12pt)
        `['font']`: Text font, specified by an integer:

                      Courier: 0
                      Times: 4
                      Helvetica: 8
                      Symbol: 12
                      New Century: 16
                      (add 1 for bold, 2 for italic)

    ['color']: Text color
    ['orient']: Text path
                  ('right', 'up', 'left', 'down')
    ['alignH']: Horizontal alignment
                  ('normal', 'left', 'center', 'right')
    ['alignV']: Vertical alignment
                  ('normal', 'top', 'cap', 'half',
                   'base', 'bottom')

['contourlegend']:
  ['x']: NDC horizontal location of the contour legend box
  ['y']: NDC vertical location of the contour legend box
  ['dx']: NDC horizontal offset to 2nd column in the contour legend box
  ['dy']: NDC vertical offset to 2nd column in the contour legend box
  ['nchars']: Maximum number of characters on a line
  ['nlines']: Maximum number of lines
  ['nwrap']: Maximum number of lines to wrap long legends
  ['textStyle']:
    ['height']: Character height in NDC, default 0.0156 (12pt)
    ['font']: Text font, specified by an integer:
                  Courier: 0
                  Times: 4
                  Helvetica: 8
                  Symbol: 12
                  New Century: 16
                  (add 1 for bold, 2 for italic)
    ['color']: Text color
    ['orient']: Text path
                  ('right', 'up', 'left', 'down')
    ['alignH']: Horizontal alignment
                  ('normal', 'left', 'center', 'right')
    ['alignV']: Vertical alignment
                  ('normal', 'top', 'cap', 'half',
                   'base', 'bottom')

['systems']: returns a list of systems, each of which is a dictionary with the keys:
  ['legend']: default legend
  ['viewport']: viewport size, array([left, right, bottom, top])

```
['ticks']:
  ['frame']: Switch the frame on (1) or off (0)
  ['frameStyle']:
     ['color']: Color of the frame
     ['width']: Line width of the frame
     ['type']: Line style of the frame:
                 ('none', 'solid', 'dash', 'dot',
                  'dashdot', 'dashdotdot')
  ['horizontal']:
     ['nDigits']: Number of digits for the tick labels
     ['nMinor']: Number of minor tick marks
     ['nMajor']: Number of major tick marks
     ['logAdjMinor']: Adjustment factor for nMinor for a log scale
     ['logAdjMajor']: Adjustment factor for nMajor for a log scale
     ['flags']: Integer, given by the sum of
                    1: There are ticks at the bottom edge of the viewport
                    2: There are ticks at the upper edge of the viewport
                    4: Ticks are centered on the axis
                    8: Ticks are go inward from the axis
                   16: Ticks are go outward from the axis
                   32: There are labels at the bottom edge
                   64: There are labels at the upper edge
                  128: There is a full grid
                  256: There is a single grid line at the origin
                  512: Alternative tick generator is used
                 1024: Alternative label generator is used
     ['xOver']: Horizontal position of the overflow label
     ['yOver']: Vertical position of the overflow label
     ['labelOff']: Offset from the edge of the viewport to the labels
     ['tickOff']: Offset from the edge of the viewport to the ticks
     ['tickLen']: Tick lengths in NDC
     ['tickStyle']:
        ['color']: Color of the ticks
        ['width']: Line width of the ticks
        ['type']: Line type of the ticks
                    ('none', 'solid', 'dash', 'dot',
                     'dashdot', 'dashdotdot')
     ['gridLevel']: Level of the ticks at which the grid is drawn
     ['gridStyle']:
        ['color']: Color of the grid
        ['width']: Line width of the grid
        ['type']: Line type of the grid
```

```
                               ('none', 'solid', 'dash', 'dot',
                                'dashdot', 'dashdotdot')
```
['vertical']:
  ['nDigits']: Number of digits for the tick labels
  ['nMinor']: Number of minor tick marks
  ['nMajor']: Number of major tick marks
  ['logAdjMinor']: Adjustment factor for nMinor for a log scale
  ['logAdjMajor']: Adjustment factor for nMajor for a log scale
  ['flags']: Integer, given by the sum of

      1: There are ticks at the left edge of the viewport
      2: There are ticks at the right edge of the viewport
      4: Ticks are centered on the axis
      8: Ticks are go inward from the axis
     16: Ticks are go outward from the axis
     32: There are labels at the left edge
     64: There are labels at the right edge
    128: There is a full grid
    256: There is a single grid line at the origin
    512: Alternative tick generator is used
   1024: Alternative label generator is used

  ['xOver']: Horizontal position of the overflow label
  ['yOver']: Vertical position of the overflow label
  ['labelOff']: Offset from the edge of the viewport to the labels
  ['tickOff']: Offset from the edge of the viewport to the ticks
  ['tickLen']: Tick lengths in NDC
  ['tickStyle']:
    ['color']: Color of the ticks
    ['width']: Line width of the ticks
    ['type']: Line type of the ticks
```
                               ('none', 'solid', 'dash', 'dot',
                                'dashdot', 'dashdotdot')
```
  ['gridLevel']: Level of the ticks at which the grid is drawn
  ['gridStyle']:
    ['color']: Color of the grid
    ['width']: Line width of the grid
    ['type']: Line type of the grid
```
                               ('none', 'solid', 'dash', 'dot',
                                'dashdot', 'dashdotdot')
```

By changing the values of the appropriate keys in the dictionary `style`, and calling the routine `set_style(style)`, the style of the current window can be changed interactively. This is used in the `plh` routine (see page 48) to switch off the tick marks and labels below the horizontal axis in a histogram.

5.1.3   Hard Copy and File Control

Calling Sequences ...

```
eps (name)
hcp ()
hcp_file ([filename] [, dump = 0/1])
filename = hcp_finish ([n])
hcp_out ([n] [, keep = 0/ 1])
hcpon ()
hcpoff ()
```

Descriptions ...

eps(name) Write the picture in the current graphics window to the Encapsulated PostScript file name + .epsi (i. e., the suffix .epsi is added to name). The eps function requires the ps2epsi utility which comes with the project GNU Ghostscript program. Any hardcopy file associated with the current window is first closed, but the default hardcopy file is unaffected. As a side effect, legends are turned off and color table dumping is turned on for the current window. The environment variable PS2EPSI_FORMAT contains the format for the command to start the ps2epsi program.

hcp() The hcp function sends the picture displayed in the current graphics window to the hardcopy file. (The name of the default hardcopy file can be specified using hcp_file; each individual graphics window may have its own hardcopy file as specified by the window function.)

hcp_file([filename [,dump=0/1])]

Sets the default hardcopy file to filename. If filename ends with .ps, the file will be a PostScript file, otherwise it will be a binary CGM file. By default, the hardcopy file name will be Aa00.cgm, or Ab00.cgm if that exists, or Ac00.cgm if both exist, and so on. The default hardcopy file gets hardcopy from all graphics windows which do not have their own specific hardcopy file (see the window function). If the dump keyword is present and nonzero, then the current palette will be dumped at the beginning of each frame of the default hardcopy file. This is what you want to do when you want color plots. With dump = 0, the default behavior of converting all colors to a gray scale is restored.

filename=hcp_finish([n)] Close the current hardcopy file and return the filename. If n is specified, close the hcp file associated with window n and return its name; use hcp_finish(-1) to close the default hardcopy file.

hcp_out([n [,keep=0/1])] **** NOT YET IMPLEMENTED **** Finishes the current hardcopy file and sends it to the printer. If n is specified, prints the hcp file associated with

window n; use `hcp_out(-1)` to print the default hardcopy file. Unless the keep keyword is supplied and nonzero, the file will be deleted after it is processed by gist and sent to lpr.

hcpon() The hcpon function causes every fma (frame advance) function call to do an implicit hcp, so that every frame is sent to the hardcopy file.

hcpoff() The hcpoff command reverts to the default "demand only" mode.

5.2 Other Controls

5.2.1 animate: Control Animation Mode

Calling Sequence ...

```
animate([0/1])
```

Description Without any arguments, toggle animation mode; with argument 0, turn off animation mode; with argument 1 turn on animation mode. In animation mode, the X window associated with a graphics window is actually an offscreen pixmap which is bit-blitted onscreen when an fma() command is issued. This is confusing unless you are actually trying to make a movie, but results in smoother animation if you are. Generally, you should turn animation on, run your movie, then turn it off.

5.2.2 palette: Set or Retrieve Palette

Calling Sequence ...

```
palette (filename)
palette (source_window_number)
palette (red, green, blue[, gray][, query = 1]
         [, ntsc = 1/0])
```

Description Set (or retrieve with query = 1) the palette for the current graphics window. The filename is the name of a Gist palette file; the standard palettes are `earth.gp`, `stern.gp`, `rainbow.gp`, `heat.gp`, `gray.gp`, and `yarg.gp`. Use the maxcolors keyword in the pldefault command to put an upper limit on the number of colors which will be read from the palette in filename.

In the second form, the palette for the current window is copied from window number `source_window_number`. If the X colormap for the window is private, there will still be two separate X colormaps for the two windows, but they will have the same color values.

In the third form, red, green, and blue are 1D arrays of unsigned char (Python typecode `"b"`) and of the same length specifying the palette you wish to install; the values should vary between 0 and 255, and your palette should have no more than 240 colors. If ntsc= 0, monochrome devices (such as most laser printers) will use the average brightness to translate your colors into gray; otherwise, the NTSC (television) averaging will be used (`.30*red`.59*green+.11* blue+). Alternatively, you can specify gray explicitly.

Ordinarily, the palette is not dumped to a hardcopy file (color hardcopy is still rare and expensive), but you can force the palette to dump using the `window()` or `hcp_file()` commands.

5.2.3   plsys: Set Coordinate System

Calling Sequence ...

```
plsys(n)
```

Description Set the current coordinate system to number n in the current graphics window. If n equals 0, subsequent elements will be plotted in absolute NDC coordinates outside of any coordinate system. The default style sheet `work.gs` defines only a single coordinate system, so the only other choice is n equal 1.

You can make up your own style sheet (using a text editor) which defines multiple coordinate systems. You need to do this if you want to display four plots side by side on a single page, for example. The standard style sheets `work2.gs` and `boxed2.gs` define two overlayed coordinate systems with the first labeled to the right of the plot and the second labeled to the left of the plot. When using overlayed coordinate systems, it is your responsibility to ensure that the x-axis limits in the two systems are identical.

5.2.4   redraw: Redraw X Window

Calling Sequence ...

```
redraw ()
```

6 Plot Limits and Scaling

6.1 Setting Plot Limits

6.1.1 limits: Save or Restore Plot Limits

Calling Sequence ...

```
old_limits = limits()
old_limits = limits(xmin [, xmax[, ymin[, ymax]]]
                            [, <keylist>])
limits(old_limits)
```

Description In the first form, restore all four plot limits to extreme values, and save the previous limits in the tuple `old_limits`.

In the second form, set the plot limits in the current coordinate system to xmin, xmax, ymin, ymax, which may each be a number to fix the corresponding limit to a specified value, or the string `"e"` to make the corresponding limit take on the extreme value of the currently displayed data. Arguments may be omitted from the right end only. (But see ylimits, below, to set limits on the y-axis.)

`limits()` always returns a tuple of 4 doubles and an integer; `old_limits[0:3]` are the previous xmin, xmax, ymin, and ymax, and `old_limits[4]` is a set of flags indicating extreme values and the square, nice, restrict, and log flags. This tuple can be saved and passed back to `limits()` in a future call to restore the limits to a previous state.

In an X window, the limits may also be adjusted interactively with the mouse. Drag left to zoom in and pan (click left to zoom in on a point without moving it), drag middle to pan, and click (and drag) right to zoom out (and pan). If you click just above or below the plot, these operations will be restricted to the x-axis; if you click just to the left or right, the operations are restricted to the y-axis. A shift-left click, drag, and release will expand the box you dragged over to fill the plot (other popular software zooms with this paradigm). If the rubber band box is not visible with shift-left zooming, try shift-middle or shift-right for alternate XOR masks. Such mouse-set limits are equivalent to a limits command specifying all four limits except that the unzoom command (see "Zooming Operations" on page 31) can revert to the limits before a series of mouse zooms and pans.

The limits you set using the limits or ylimits functions carry over to the next plot; that is, an fma operation does not reset the limits to extreme values.

Keyword Arguments The following keyword arguments can be specified with this function.

```
square = 0/1
```

If present, the square keyword determines whether limits marked as extreme values will be adjusted to force the x and y scales to be equal (square= 1) or not (square= 0, the default).

```
nice = 0/ 1
```

If present, the nice keyword determines whether limits will be adjusted to nice values (nice= 1) or not (nice= 0, the default).

```
restrict = 0/ 1
```

There is a subtlety in the meaning of "extreme value" when one or both of the limits on the OPPOSITE axis have fixed values: does the "extreme value" of the data include points which will not be plotted because their other coordinate lies outside the fixed limit on the opposite axis (restrict= 0, the default), or not (restrict= 1)?

6.1.2   ylimits: Set y-axis Limits

Calling Sequence ...

```
ylimits (ymin[, ymax])
```

Description Set the y-axis plot limits in the current coordinate system to ymin, ymax, which may each be a number to fix the corresponding limit to a specified value, or the string "e" to make the corresponding limit take on the extreme value of the currently displayed data.

Arguments may be omitted only from the right. Use limits(xmin, xmax) to accomplish the same function for the x-axis plot limits.

Note that the corresponding Yorick function for ylimits is range. Since this word is a Python built-in function, the name has been changed to avoid the collision.

6.2   Scaling and Grid Lines

6.2.1   logxy: Set Linear/ Log Axis Scaling

Calling Sequence ...

```
logxy(xflag[, yflag])
```

Description logxy sets the linear/log axis scaling flags for the current coordinate system. xflag and
yflag may be 0 to select linear scaling, or 1 to select log scaling. yflag may be omitted (but
not xflag).

6.2.2 gridxy: Specify Grid Lines

Calling Sequence ...

```
gridxy(flag)
gridxy(xflag, yflag)
```

Description Turns on or off grid lines according to flag. In the first form, both the x and y axes
are affected. In the second form, xflag and yflag may differ to have different grid options for
the two axes. In either case, a flag value of 0 means no grid lines (the default), a value of
1 means grid lines at all major ticks (the level of ticks which get grid lines can be set in the
style sheet), and a flag value of 2 means that the coordinate origin only will get a grid line. In
styles with multiple coordinate systems, only the current coordinate system is affected. The
keywords can be used to affect the style of the grid lines.

You can also turn the ticks off entirely. (You might want to do this to plot your own custom
set of tick marks when the automatic tick generating machinery will never give the ticks you
want. For example a latitude axis in degrees might reasonably be labeled "0, 30, 60, 90", but
the automatic machinery considers 3 an "ugly" number—only 1, 2, and 5 are "pretty"—and
cannot make the required scale. In this case, you can turn off the automatic ticks and labels,
and use plsys, pldj, and plt to generate your own.)

To fiddle with the tick flags in this general manner, set the 0x200 bit of flag (or xflag or yflag),
and "or-in" the 0x1ff bits however you wish. The meaning of the various flags is described in
the `work.gs` Gist style sheet. Additionally, you can use the 0x400 bit to turn on or off the
frame drawn around the viewport. Here are some examples:

```
gridxy(0x233)    work.gs default setting
gridxy(0, 0x200) like work.gs, but no y-axis ticks or labels
gridxy(0, 0x231) like work.gs, but no y-axis ticks on right
gridxy(0x62b)    boxed.gs default setting
```

6.3 Zooming Operations

Calling Sequences ...

```
zoom_factor(factor)
unzoom()
```

Descriptions `zoom_factor` sets the zoom factor for mouse-click zoom in and zoom out operations. The default factor is 1.5; factor should always be greater than 1.0.

unzoom restores limits to their values before zoom and pan operations performed interactively using the mouse. Use

```
old_limits = limits()
...
limits(old_limits)
```

to save and restore plot limits generally.

7 Two-Dimensional Plotting Functions

This chapter describes the Gist output primitives are available for drawing two-dimensional plots. Keyword arguments that apply only to a single function are described with that function; those that apply to several are collected in a separate section at the end of the chapter.

7.1 Output Primitives

7.1.1 plg: Plot a Graph

Calling Sequence ...

```
plg(y [, x][, <keylist>])
```

Description Plot a graph of y versus x. y and x must be 1D arrays of equal length. If x is omitted, it defaults to [1, 2, ..., len(y)].

Keyword Arguments The following keyword argument(s) apply only to this function.

```
rspace = <float value>
rphase = <float value>
arroww = <float value>
arrowl = <float value>
```

Select the spacing, phase, and size of occasional ray arrows placed along polylines. The spacing and phase are in NDC units (0. 0013 NDC equals 1.0 point); the default rspace is 0.13, and the default rphase is 0.11375, but rphase is automatically incremented for successive curves on a single plot. The arrowhead width, arroww, and arrowhead length, arrowl are in relative units, defaulting to 1.0, which translates to an arrowhead 10 points long and 4 points in half-width.

The following additional keyword arguments can be specified with this function. legend, hide, type, width, color, closed, smooth, marks, marker, mspace, mphase, rays

See "Plot Function Keywords" on page 48 for detailed descriptions of these keywords.

Examples The following example simply plots two straight lines.

```
>>> from gist import *
>>> window (0, wait= 1, dpi= 75)
0
>>> plg([0,1])
>>> plg([1,0])
```
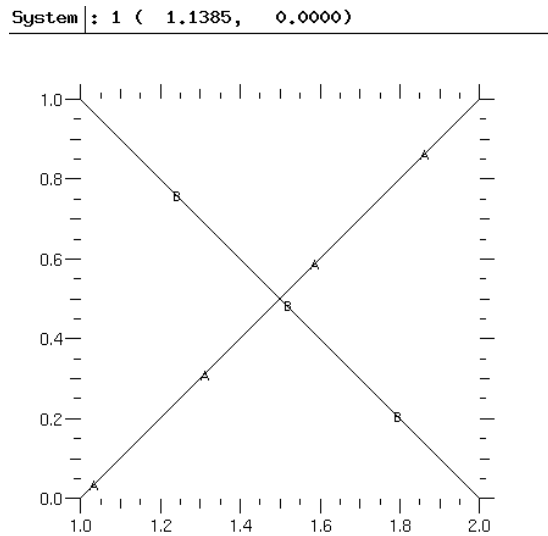

The following draws the graph of a sine curve:

```
fma()
x = 10* pi* arange(200, typecode = Float)/ 199.0
plg(sin(x), x)
```

7.1.2   plmesh: Set Default Mesh

Calling Sequence ...

```
plmesh([y, x][, ireg][, triangle= tri_array])
plmesh()
```

Description Set the default mesh for subsequent plm, plc, plv, plf, and plfc calls. In the second
form, plmesh deletes the default mesh (until you do this, or switch to a new default mesh, the
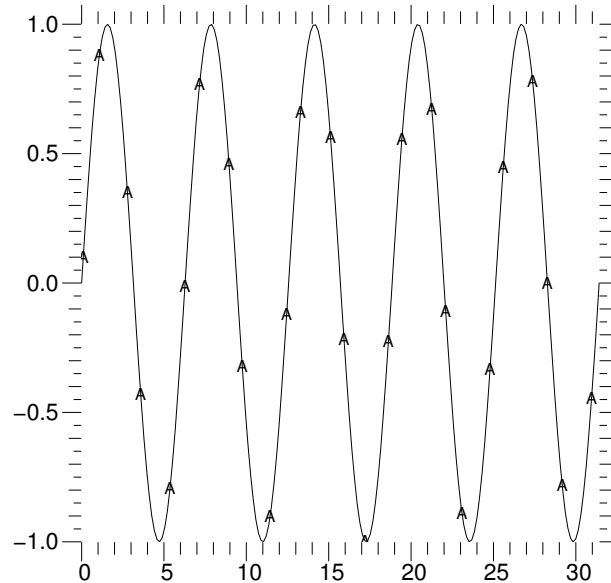
default mesh arrays persist and takes up space in memory). The y, x, and ireg arrays should all be the same shape; y and x will be converted to double, and ireg will be converted to int.

If ireg is omitted, it defaults to ireg(0,)= ireg(, 0)= 0, ireg(1:, 1:)= 1; that is, region number 1 is the whole mesh. The triangulation array `tri_array` is used by plc and plfc; the correspondence between `tri_array` indices and zone indices is the same as for ireg, and its default value is all zero. The ireg or `tri_array` arguments may be supplied without y and x to change the region numbering or triangulation for a given set of mesh coordinates. However, a default y and x must already have been defined if you do this. If y is supplied, x must be supplied, and vice-versa.

Example   The following example creates a mesh whose graph we will see later. For convenience, we show the functions span and a3, which are used to build the data.

```
def span(lb, ub, n):
if n < 2: raise ValueError, '3rd arg must be at least 2'
b = lb
a = (ub -lb)/(n -1.0)
return map(lambda x, A= a, B= b: A* x + B, range(n))
def a3(lb, ub, n):
return reshape (array(n* span(lb, ub, n), Float), (n, n))
fma()
limits()
x = a3(-1, 1, 26)
y = transpose (x)
```

```
z = x+ 1j* y
z = 5.* z/(5.+ z* z)
xx = z. real
yy = z. imaginary
plmesh(yy, xx)
```

7.1.3   plm: Plot a Mesh

Calling Sequence ...

```
plm([y,x][,ireg][,<keylist>])
```

Description Plot a mesh of y versus x. y and x must be 2D arrays with equal dimensions. If present, ireg must be a 2D region number array for the mesh, with the same dimensions as x and y. The values of ireg should be positive region numbers, and zero for zones which do not exist. The first row and column of ireg never correspond to any zone, and should always be zero. The default ireg is 1 everywhere else.

The y, x, and ireg arguments may all be omitted to default to the mesh set by the most recent plmesh call.

Keyword Arguments The following keyword argument(s) apply only to this function.

```
boundary = 0/ 1
```

If present, the boundary keyword determines whether the entire mesh is to be plotted (boundary= 0, the default), or just the boundary of the selected region (boundary= 1).

```
inhibit = 0/ 1/ 2/ 3
```

If present, the inhibit keyword causes the $(x(, j), y(, j))$ lines to not be plotted (inhibit= 1), the $(x(i,), y(i,))$ lines to not be plotted (inhibit= 2), or both sets of lines not to be plotted (inhibit= 3). By default (inhibit= 0), mesh lines in both logical directions are plotted.

The following additional keyword arguments can be specified with this function:

```
legend, hide, type, width, color, region
```

See "Plot Function Keywords" on page 48 for detailed descriptions of these keywords.

Example The mesh set by the plmesh function call in the preceding example may be plotted simply by calling plm with no arguments:

```
plm ()
```
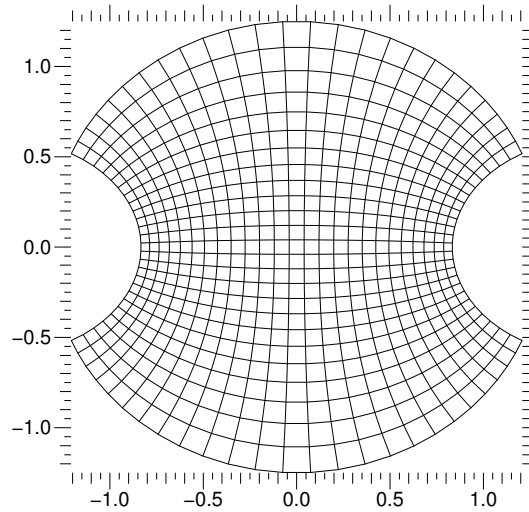
```
plm,
```

7.1.4   plc: Plot Contours

Calling Sequence ...

```
plc(z[, y, x][, ireg][, <keylist>])
```

Description Plot contours of z on the mesh y versus x. y, x, and ireg are as for plm. The z array
   must have the same shape as y and x. The function being contoured takes the value z at each
   point (x, y); that is, the z array is presumed to be point-centered. The y, x, and ireg arguments
   may all be omitted to default to the mesh set by the most recent plmesh call.

Keyword Arguments The following keyword argument(s) apply only to this function.

```
levs = z_values
```

The levs keyword specifies a list of the values of z at which you want contour curves. The
default is eight contours spanning the range of z.

```
triangle = triangle
```

Set the triangulation array for a contour plot. triangle must be the same shape as the ireg
(region number) array, and the correspondence between mesh zones and indices is the same
as for ireg. The triangulation array is used to resolve the ambiguity in saddle zones, in which
the function z being contoured has two diagonally opposite corners high, and the other two

corners low. The triangulation array element for a zone is 0 if the algorithm is to choose a triangulation, based on the curvature of the first contour to enter the zone. If zone (i, j) is to be triangulated from point (i-1, j-1) to point (i, j), then triangle(i, j)= 1, while if it is to be triangulated from (i-1, j) to (i, j-1), then triangle(i, j)=-1. Contours will never cross this "triangulation line".

You should rarely need to fiddle with the triangulation array; it is a hedge for dealing with pathological cases.

The following additional keyword arguments can be specified with this function. legend, hide, type, width, color, smooth, marks, marker, mspace, mphase, region

See "Plot Function Keywords" on page 48 for detailed descriptions of these keywords.

Examples The following example gives a contour plot of the same mesh used in the preceding two examples. Calling plm with boundary = 1 and region = 1 plots the boundary of the mesh (which, by default, is one region); then calling plc plots a default number of contours (8).

```
fma()
def mag(* args):
    r = 0
    for i in range(len(args)):
        r = r + args[i]* args[i]
    return sqrt(r)
plm(region= 1, boundary= 1)
plc (mag(x+. 5, y-. 5), marks= 1, region= 1)
plm(inhibit= 3, boundary= 1, region= 1)
plm(boundary= 1, region= 1)
```

7.1.5   plv: Plot a Vector Field

Calling Sequence ...

```
plv(vy, vx[, y, x][, ireg][, <keylist>])
```

Description Plot a vector field (vx, vy) on the mesh (x, y). y, x, and ireg are as for plm. The vy and vx arrays must have the same shape as y and x. The y, x, and ireg arguments may all be omitted to default to the mesh set by the most recent plmesh call.

Keyword Arguments The following keyword argument(s) apply only to this function.

```
scale = dt
```
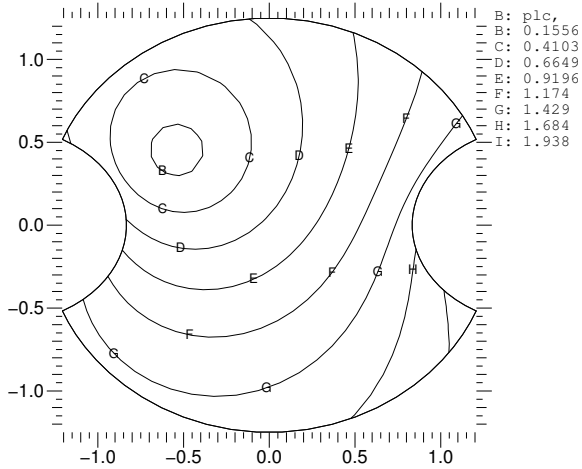
```
plm,
B: plc,
plm,
plm,
```
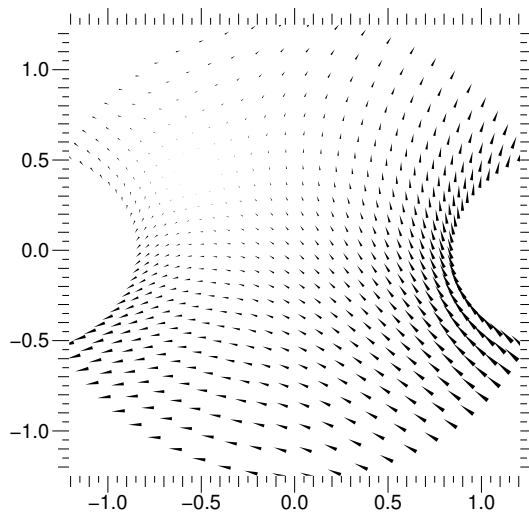


```
plv,
```

The scale keyword is the conversion factor from the units of (vx, vy) to the units of (x, y) — a time interval if (vx, vy) is a velocity and (x, y) is a position — which determines the length of the vector "darts" plotted at the (x, y) points.

If omitted, scale is chosen so that the longest ray arrows have a length comparable to a "typical" zone size. You can use the scalem keyword in pledit to make adjustments to the scale factor computed by default.

```
hollow = 0/ 1
aspect = <float value>
```

Set the appearance of the "darts" of a vector field plot. The default darts, hollow= 0, are filled; use hollow= 1 to get just the dart outlines. The default is aspect= 0.125; aspect is the ratio of the half-width to the length of the darts. Use the color keyword to control the color of the darts.

The following additional keyword arguments can be specified with this function. legend, hide, type, width, color, smooth, marks, marker, mspace, mphase, triangle, region

See "Plot Function Keywords" on page 48 for detailed descriptions of these keywords.

Example This example applies to the same mesh that we have considered in the last three examples.

```
plv(x+. 5, y-. 5)
```

The plot appears on the next page.

7.1.6   plf: Plot a Filled Mesh

Calling Sequence ...

```
plf(z[, y, x][, ireg][, <keylist>])
```

Description Plot a filled mesh y versus x. y, x, and ireg are as for plm. The z array must have the same shape as y and x, or one smaller in both dimensions. If z is of type unsigned char (Python typecode 'b'), it is used "as is"; otherwise, it is linearly scaled to fill the current palette, as with the bytscl function. The mesh is drawn with each zone in the color derived from the z function and the current palette; thus z is interpreted as a zone-centered array. The y, x, and ireg arguments may all be omitted to default to the mesh set by the most recent plmesh call.

A solid edge can optionally be drawn around each zone by setting the edges keyword non-zero. ecolor and ewidth determine the edge color and width. The mesh is drawn zone by zone in order from ireg(2+ imax) to ireg(jmax* imax) (the latter is ireg(imax, jmax)), so you can achieve 3D effects by arranging for this order to coincide with back-to-front order. If z is nil, the mesh zones are filled with the background color, which you can use to produce 3D wire frames.

Keyword Arguments The following keyword argument(s) apply only to this function.

```
edges = 0/ 1
ecolor = <color value>
ewidth = <float value>
```

Set the appearance of the zone edges in a filled mesh plot (plf). By default, edges= 0, and the zone edges are not plotted. If edges= 1, a solid line is drawn around each zone after it is filled; the edge color and width are given by ecolor and ewidth, which are "fg" and 1.0 by default.

The following additional keyword arguments can be specified with this function. legend, hide, region, top, cmin, cmax See "Plot Function Keywords" on page 48 for detailed descriptions of these keywords. (See the bytscl function description on page 93 for explanation of top, cmin, cmax.)

Examples The following gives a filled mesh plot of the same mesh we have been considering in the preceding examples.

```
plf (mag(x+. 5, y-. 5))
```
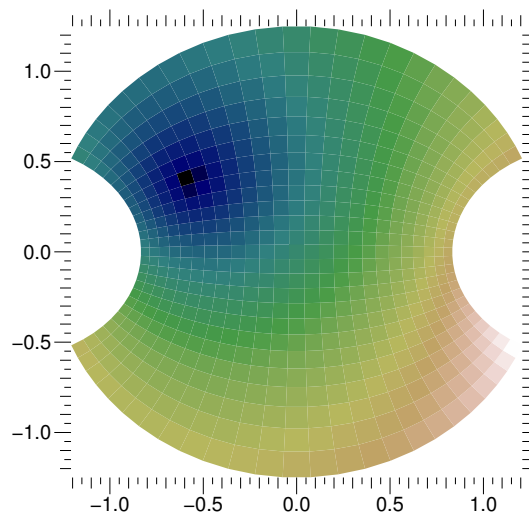


```
plf,
```

7.1.7 plfc: Plot filled contours

Calling Sequence ...

```
plfc (z, y, x, ireg, contours = 20, colors = None,
    region = 0, triangle = None, scale = "lin")
```

Description Unlike the other plotting primitives, plfc is implemented in Python code. It calls a C module to compute the contours, then uses plfp (described in the next subsection) to draw the

filled contour lines. It does not use the mesh plotting routines; hence the arguments z, y, x, and ireg must be given explicitly. They will not default to the values set by plmesh.

Keyword Arguments The values given above for the keyword arguments are the defaults. The meanings of the keywords are as follows:

`contours`

If an integer, specifies the number of contour lines desired. The contour levels will be computed automatically. If an array of floats, specifies the actual contour levels.

`colors`

An array of unsigned char (Python typecode 'b') with values between 0 and 199 specifying the indices into the current palette of the fill colors to use. The size of this array (if present) must be one larger than the number of contours specified.

`triangle`

As described for the mesh plots. scale If the number of contours was given, this keyword specifies how they are to be computed: `"lin"` (linearly), `"log"` (logarithmically) and `"normal"` (based on the normal distribution; the minimum and maximum contours will be two standard deviations from the mean).

Example In the following example, we have to explicitly compute and pass an ireg array. We plot filled contours and then plot contour lines on top of them. Note that the contour divisions do not coincide, since the two routines use different algorithms for computing contour levels. Perhaps someday this defect will be remedied.

```
ireg = ones (xx. shape, Int)
ireg [0, :] = 0
ireg [:, 0] = 0
plfc(mag(x+. 5, y-. 5), yy, xx, ireg, contours= 8)
plc (mag(x+. 5, y-. 5), marks= 1)
```

7.1.8   plfp: Plot a List of Filled Polygons

Calling Sequence ...

```
plfp(z, y, x, n[, <keylist>])
```

Description Plot a list of filled polygons y versus x, with colors z. The n array is a 1D list of lengths (number of corners) of the polygons; the 1D colors array z has the same length as n. The x and y arrays have length equal to the sum of all dimensions of n.

If z is of type unsigned char (Python typecode `"b"`), it is used "as is"; otherwise, it is linearly scaled to fill the current palette, as with the bytscl function.
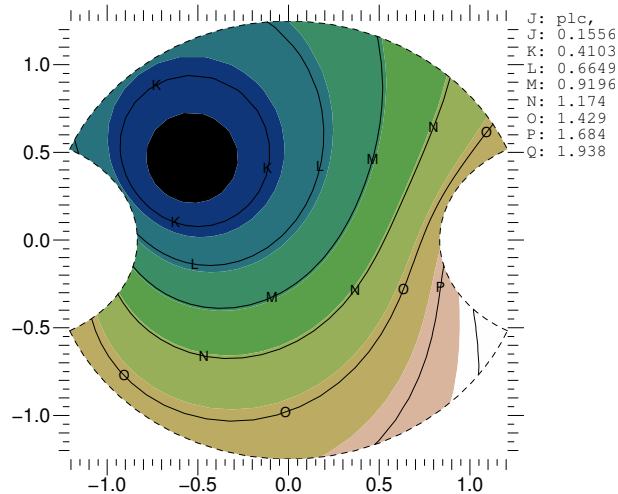
```
J: plc,
plm,
```

Keyword Arguments The following keyword arguments can be specified with this function. legend, hide, top, cmin, cmax See "Plot Function Keywords" on page 48 for detailed descriptions of these keywords. (See the bytscl function description on page 93 for explanation of top, cmin, cmax.)

Example This example gives a sort of "stained glass window" effect;.

```
z = array([190,100, 130,100,50,190,160,100,50,100,130], 'b')
y = array ([1.0, 2.0, 7.0, 8.0, 1.0, 1.0, 2.0, 0.0, 1.0, 1.0,
1.0, 1.0, 2.0, 1.0, 2.0, 2.0, 2.0, 1.0, 8.0, 7.0, 2.0, 2.0,
7.0, 7.0, 7.0, 8.0, 8.0, 7.0, 7.0, 8.0, 7.0, 8.0, 8.0, 8.0,
8.0, 9.0])
x = array ([0.0, 1.0, 1.0, 0.0, 0.0, 1.5, 1.0, 1.5, 3.0, 0.0,
1.5, 3.0, 2.0, 1.5, 2.0, 1.0, 2.0, 3.0, 3.0, 2.0, 1.0, 2.0,
2.0, 1.0, 2.0, 3.0, 1.5, 1.0, 2.0, 1.5, 1.0, 1.5, 0.0, 0.0,
3.0, 1.5])
n = array ([4, 3, 3, 3, 3, 4, 4, 3, 3, 3, 3])
plfp (z, y, x, n)
```

7.1.9   pli: Plot a Cell Array

Calling Sequence ...
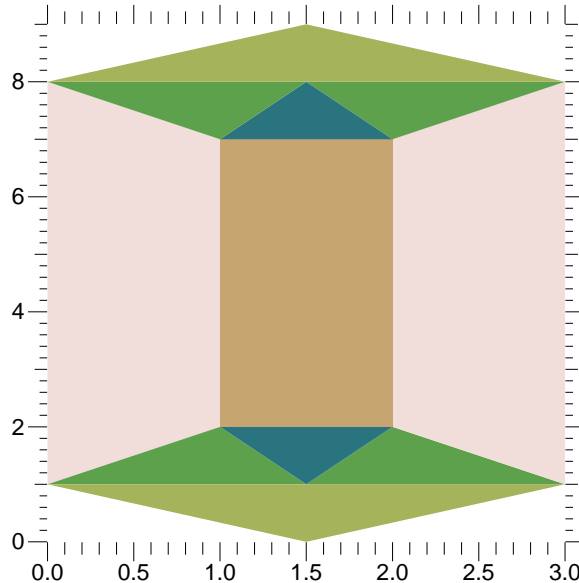
```
pli(z[[, x0, y0], x1, y1][, <keylist>])
```

Description Plot the image z as a cell array: an array of equal rectangular cells colored according to the 2D array z. The first dimension of z is plotted along x, the second dimension is along y. If z is of type unsigned char (Python typecode `"b"`), it is used " as is"; otherwise, it is linearly scaled to fill the current palette, as with the bytscl function.

If x1 and y1 are given, they represent the coordinates of the upper right corner of the image. If x0, and y0 are given, they represent the coordinates of the lower left corner, which is at (0, 0) by default. If only the z array is given, each cell will be a 1x1 unit square, with the lower left corner of the image at (0, 0).

Keyword Arguments The following keyword arguments can be specified with this function. legend, hide, top, cmin, cmax See "Plot Function Keywords" on page 48 for detailed descriptions of these keywords. (See the bytscl function description on page 93 for explanation of top, cmin, cmax.)

Example The following example computes and draws an interesting cell array.

```
fma()
unzoom()
x = a3 (-6,6,200)
y = transpose (x)
r = mag(y, x)
theta = arctan2 (y, x)
funky = cos(r)** 2 * cos(3* theta)
```

```
pli(funky)
```
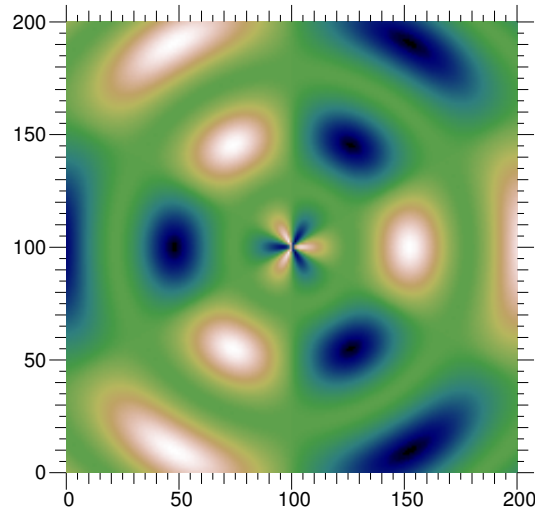


```
pli,
```

7.1.10   pldj: Plot Disjoint Lines

Calling Sequence ...

```
pldj(x0, y0, x1, y1[, <keylist>])
```

Description Plot disjoint lines from (x0, y0) to (x1, y1). x0, y0, x1, and y1 may have any dimensionality, but all must have the same number of elements.

Keyword Arguments The following keyword arguments can be specified with this function. legend, hide, type, width, color See "Plot Function Keywords" on page 48 for detailed descriptions of these keywords. (See the bytscl function description on page 93 for explanation of top, cmin, cmax.)

Example This example draws a set of seventeen-pointed stars.

```
theta = a2(0, 2* pi, 18)
x = cos(theta)
y = sin(theta)
pldj(x, y, transpose (x), transpose (y))
pltitle("Seventeen Pointed Stars")
limits(square = 1)
```

Seventeen Pointed Stars
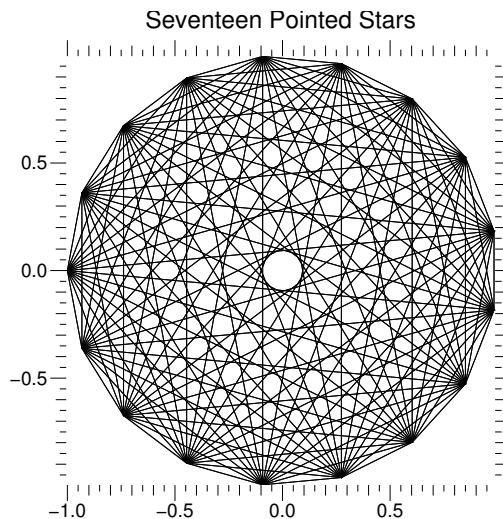
pldj,

7.1.11   plt: Plot Text

Calling Sequence ...

```
plt(text, x, y[, <keylist>])
```

Description   Plot text (a string) at the point (x, y). The exact relationship between the point (x, y) and the text is determined by the justify keyword. text may contain newline ("\n") characters to output multiple lines of text with a single call.

The coordinates (x, y) are NDC coordinates (outside of any coordinate system) unless the tosys keyword is present and non-zero, in which case the text will be placed in the current coordinate system. However, the character height is never affected by the scale of the coordinate system to which the text belongs.

Note that the pledit command (see "pledit: Change Plotting Properties" on page 53) takes dx and/ or dy keywords to adjust the position of existing text elements.

Keyword Arguments   The following keyword argument(s) apply only to this function. tosys = 0/ 1 Establish the interpretation of (x, y). If tosys= 0 (the default), use Normalized Device Coordinates; if nonzero, use the current coordinate system.

```
font = <font value>
height = <float value>
opaque = 0/ 1
```

```
path = 0/ 1
orient = <integer value>
justify = (see text description)
```

Select text properties. The font can be any of the strings `"courier"`, `"times"`, `"helvetica"` (the default), `"symbol"`, or `"schoolbook"`. Append B for boldface and `"I"` for italic, so `"courierB"` is boldface Courier, `"timesI"` is Times italic, and `"helveticaBI"` is bold italic (oblique) Helvetica. Your X server should have the Adobe fonts (available free from the MIT X distribution tapes) for all these fonts, preferably at both 75 and 100 dpi. Occasionally, a PostScript printer will not be equipped for some fonts; often New Century Schoolbook is missing. The font keyword may also be an integer: 0 is Courier, 4 is Times, 8 is Helvetica, 12 is Symbol, 16 is New Century Schoolbook, and you add 1 to get boldface and/ or 2 to get italic (or oblique).

The height is the font size in points; 14.0 is the default. X windows only has 8, 10, 12, 14, 18, and 24 point fonts, so don't stray from these sizes if you want what you see on the screen to be a reasonably close match to what will be printed. By default, opaque= 0 and text is transparent. Set opaque= 1 to white-out a box before drawing the text.

The default path (path= 0) is left-to-right text; set path= 1 for top-to-bottom text. The default text justification, `justify="NN"` is normal in both the horizontal and vertical directions. Other possibilities are `"L"`, `"C"`, or `"R"` for the first character, meaning left, center, and right horizontal justification, and `"T"`, `"C"`, `"H"`, `"A"`, or `"B"` for the second character, meaning top, capline, half, baseline, and bottom vertical justification. The normal justification `"NN"` is equivalent to `"LA"` if path= 0, and to `"CT"` if path= 1. Common values are `"LA"`, `"CA"`, and `"RA"` for garden variety left, center, and right justified text, with the y coordinate at the baseline of the last line in the string presented to plt. The characters labeling the right axis of a plot are `"RH"`, so that the y value of the text will match the y value of the corresponding tick. Similarly, the characters labeling the bottom axis of a plot are `"CT"`. The justification may also be a number, horizontal+ vertical, where horizontal is 0 for `"N"`, 1 for `"L"`, 2 for `"C"`, or 3 for `"R"`, and vertical is 0 for `"N"`, 4 for `"T"`, 8 for `"C"`, 12 for `"H"`, 16 for `"A"`, or 20 for `"B"`.

The integer value orient (default 0) specifies one of four angles that the text makes with the horizontal (0 is horizontal, 1 is ninety degrees, 2 is 180 degrees, and 3 is 270 degrees).

7.1.12   pltitle: Plot a Title

Calling Sequence ...

```
pltitle(title)
```

Description Plot title centered above the coordinate system for any of the standard Gist styles. You will need to customize this for other plot styles.

7.1.13   plh: Plot a Histogram

Calling Sequence ...

```
plh(y, [x, labels, <keylist>])
```

Description Draws a histogram, where the height of the bars is given by Y. If X is None, the bars
in the histogram have a width equal to unity. If X is a single real number, it is interpreted as
the width of the bars. If X is a one- dimensional array with an equal number of elements as
the Y array, then X is interpreted as the widths of the bars. In all of these cases, the histogram
starts at the origin. However, if X is a one-dimensional array with one element more than Y,
then X is interpreted as the locations of the start and end points of the bars in the histogram.
If X is a one-dimensional array with twice as many elements as Y, then X represents the start
and end points for each bar separately.

The keyword color is either a single color to fill the bars with, or a list of colors, one for
each bar. If the keyword labels is given, then the horizontal tick marks and numerical labels
are switched off. The keyword labels should then consist of a list of strings, with the same
number of elements as Y. These labels are drawn below the horizontal axis.

To switch the tick marks and labels back on for subsequent plots, you can execute

```
window(style="work.gs")
```

which will reset the window to the usual work.gs style sheet.

Keyword Arguments width, hide, color

7.2   Plot Function Keywords

In addition to the keyword arguments described above with individual Gist primitive plotting com-
mands, the following keywords are available to modify the details of the plots.

legend ...

```
legend = "text destined for the legend"
```

Set the legend for a plot. There are no default legends in PyGist. Legends are never plotted
to the X window; use the plq command to see them interactively. Legends will appear in
hardcopy output unless they have been explicitly turned off.

Plotting Commands: plg, plm, plc, plv, plf, pli, plt, pldj

See also: hide

hide ...

```
hide = 0/ 1
```

Set the visibility of a plotted element. The default is hide= 0, which means that the element will be visible. Use hide= 1 to remove the element from the plot (but not from the display list).

Plotting Commands: plg, plm, plc, plv, plf, pli, plt, pldj, plh

See also: legend

type ...

```
type = <line type value>
```

Select line type. Valid values are the strings `"solid"`, `"dash"`, `"dot"`, `"dashdot"`, `"dashdotdot"`, and `"none"`. The `"none"` value causes the line to be plotted as a poly-marker. The type value may also be a number; 0 is `"none"`, 1 is `"solid"`, 2 is `"dash"`, 3 is `"dot"`, 4 is `"dashdot"`, and 5 is `"dashdotdot"`.

Plotting Commands: plg, plm, plc, pldj

See also: width, color, marks, marker, rays, closed, smooth

width ...

```
width = <floating point value>
```

Select line width. Valid values are positive floating point numbers giving the line thickness relative to the default line width of one half point, which is width = 1.0.

Plotting Commands: plg, plm, plc, pldj, plv (only if hollow= 1), plh

See also: type, color, marks, marker, rays, closed, smooth

color ...

```
color = <color value>
```

Select line or text color. Valid values are the strings `"bg"`, `"fg"`, `"black"`, `"white"`, `"red"`, `"green"`, `"blue"`, `"cyan"`, `"magenta"`, `"yellow"`, or a 0-origin index into the current palette. The default is `"fg"`. Negative numbers may be used instead of the strings: -1 is `"bg"` (background), -2 is `"fg"` (foreground), -3 is black, -4 is white, -5 is red, -6 is green, -7 is blue, -8 is cyan, -9 is magenta, and -10 is yellow.

Plotting Commands: plg, plm, plc, pldj, plt, plh

See also: type, width, marks, marker, mcolor, rays, closed, smooth

marks ...

```
marks = 0/ 1
```

Select unadorned lines (marks= 0), or lines with occasional markers (marks= 1). Ignored if type is `"none"` (indicating polymarkers instead of occasional markers). The spacing and phase of the occasional markers can be altered using the mspace and mphase keywords; the character used to make the mark can be altered using the marker keyword.

Plotting Commands: plg, plc

See also: type, width, color, marker, rays, mspace, mphase, msize, mcolor

marker ...

```
marker = <character or integer value>
```

Select the character used for occasional markers along a polyline, or for the polymarker if type=`"none"`. The special values `'\ 1'`, `'\ 2'`, `'\ 3'`, `'\ 4'`, and `'\ 5'` stand for point, plus, asterisk, circle, and cross, which are prettier than text characters on output to some devices. The default marker is the next available capital letter: `'A'`, `'B'`, `...,` `'Z'`.

Plotting Commands: plg, plc

See also: type, width, color, marks, rays, mspace, mphase, msize, mcolor

mspace ...

```
mspace = <float value>
mphase = <float value>
msize = <float value>
```

Select the spacing, phase, and size of occasional markers placed along polylines. The msize also selects polymarker size if type is `"none"`. The spacing and phase are in NDC units (0. 0013 NDC equals 1.0 point); the default mspace is 0.16, and the default mphase is 0.14, but mphase is automatically incremented for successive curves on a single plot. The msize is in relative units, with the default msize of 1.0 representing 10 points.

Plotting Commands: plg, plc

See also: type, width, color, marks, marker, rays

mcolor ...

```
mcolor = <color value>
```

The mcolor keyword is the same as the color keyword, but controls the marker color instead of the line color. Setting the color automatically sets the mcolor to the same value, so you only need to use mcolor if you want the markers for a curve to be a different color than the curve itself.

Plotting Commands: plg, plc

See also: type, width, color, marks, marker, rays

rays ...

```
rays = 0/ 1
```

Select unadorned lines (rays= 0), or lines with occasional ray arrows (rays= 1). Ignored if type is `"none"`. The spacing and phase of the occasional arrows can be altered using the rspace and rphase keywords; the shape of the arrowhead can be modified using the arroww and arrowl keywords.

Plotting Commands: plg, plc

See also: type, width, color, marker, marks, rspace, rphase, arroww, arrowl

closed/smooth ...

```
closed = 0/ 1
smooth = 0/ 1/ 2/ 3/ 4
```

Select closed curves (closed= 1) or default open curves (closed= 0), or Bezier smoothing (smooth¿ 0) or default piecewise linear curves (smooth= 0). The value of smooth can be 1, 2, 3, or 4 to get successively more smoothing. Only the Bezier control points are plotted to an X window; the actual Bezier curves will show up in PostScript hardcopy files. Closed curves join correctly, which becomes more noticeable for wide lines; non-solid closed curves may look bad because the dashing pattern may be incommensurate with the length of the curve.

Plotting Commands: plg, plc (smooth only)

See also: type, width, color, marks, marker, rays

region ...

```
region = <region number>
```

Select the part of mesh to consider. The region should match one of the numbers in the ireg array. Putting region= 0 (the default) means to plot the entire mesh; that is, everything EXCEPT region zero (non-existent zones). Any other number means to plot only the specified region number; region= 3 would plot region 3 only.

Plotting Commands: plm, plc, plv, plf

8 Inquiry and Miscellaneous Functions

This chapter describes functions that are available to inquire about the state of PyGist control variables and set their values. It also describes other miscellaneous functions.

8.1   Inquiry and Editing Functions

8.1.1   plq: Query Plot Element Status

Calling Sequence ...

```
plq()
legend_list = plq() **** RETURN VALUE NOT YET IMPLEMENTED ****
plq(n_element[, n_contour])
properties = plq(n_element[, n_contour])
```

Description Called as a subroutine, plq prints the list of legends for the current coordinate system (with an (H) to mark hidden elements), or prints a list of current properties of element n_element (such as line type, width, font, etc.), or of contour number n_contour of element number n_element (which must be contours generated using the plc command). Elements and contours are both numbered starting with one; hidden elements or contours are included in this numbering.

The plq function always operates on the current coordinate system in the current graphics window; use window and plsys to change these.

8.1.2   pledit: Change Plotting Properties

Calling Sequence ...

```
pledit([n_element[, n_contour],] <keylist>)
```

where, as usual, <keylist> has the form key1=value1, key2=value2, ...

Description pledit changes some property or properties of element number `n_element` (and con-
tour number `n_contour` of that element). If `n_element` and `n_contour` are omitted,
the default is the most recently added element, or the element specified in the most recent plq
query command.

The keywords can be any of the keywords that apply to the current element. These are: plg:
color, type, width, marks, mcolor, marker, msize, mspace, mphase, rays, rspace, rphase, ar-
rowl, arroww, closed, smooth plm: region, boundary, inhibit, color, type, width plc: region,
color, type, width, marks, mcolor, marker, msize, mspace, mphase, smooth, levs (For con-
tours, if you aren't talking about a particular `n_contour`, any changes will affect ALL the
contours.) plv: region, color, hollow, width, aspect, scale plf: region pldj: color, type, width
plt: color, font, height, path, justify, opaque

A plv (vector field) element can also take the scalem keyword to multiply all vector lengths
by a specified factor.

A plt (text) element can also take the dx and/ or dy keywords to adjust the text position by
(dx, dy).

8.1.3   plremove: Remove Plot Element from Display

Calling Sequence ...

```
plremove(n_element)
```

Description Removes the element number `n_element` from the display. If `n_element` is omit-
ted, the default is the most recently added element, or the element specified in the most recent
`plq` query command.

8.1.4   pldefault: Set Default Values

Calling Sequence ...

```
pldefault(key1= value1, key2= value2, ...)
```

Description Set default values for the various properties of graphical elements. The keywords can
be most of the keywords that can be passed to the plotting commands:

```
plg: color, type, width, marks, mcolor, msize, mspace,
     mphase, rays, rspace, rphase, arrowl, arroww
plm: color, type, width
```

```
plc: color, type, width, marks, mcolor, marker, msize,
     mspace, mphase
plv: color, hollow, width, aspect
plf: edges, ecolor, ewidth
pldj: color, type, width
plt: color, font, height, path, justify, opaque
```

The initial default values are:

```
color="fg", type="solid", width= 1.0 (1/ 2 point),
marks= 1, mcolor="fg", msize= 1.0 (10 points), mspace= 0.16,
mphase= 0.14,
rays= 0, arrowl= 1.0 (10 points), arroww= 1.0 (4 points), rspace= 0.13,
rphase= 0.11375,
font="helvetica", height= 12.0, path= 0, justify="NN",
opaque= 0,
hollow= 0, aspect= 0.125,
edges= 0, ecolor="fg", ewidth= 1.0 (1/ 2 point)
```

Additional default keywords are:

```
dpi, style, legends (see window command)
palette      (to set default filename as in palette command)
maxcolors    (default 200)
cgmfilesize  (default 10 MB) in units of MBytes
```

8.2   Miscellaneous Functions

8.2.1   bytscl: Convert to Color Array

Calling Sequence ...

```
bytscl(z[, top= max_byte][, cmin= lower_cutoff]
       [, cmax= upper_cutoff])
```

Description bytscl returns an unsigned char array of the same shape as z, with values linearly scaled to the range 0 to one less than the current palette size. If max_byte is specified, the scaled values will run from 0 to max_byte instead.

If lower_cutoff and/ or upper_cutoff are specified, z values outside this range are mapped to the cutoff value; otherwise the linear scaling maps the extreme values of z to 0 and max_byte.

8.2.2   histeq_scale: Histogram Equalized Scaling

**** NOT YET IMPLEMENTED ****

Calling Sequence ...

```
histeq_scale(z[, top= top_value][, cmin= cmin][, cmax= cmax])
```

Description `histeq_scale` returns a byte-scaled version of the array z having the property that
each byte occurs with equal frequency (z is histogram equalized). The result bytes range from
0 to `top_value`, which defaults to one less than the size of the current palette (or 255 if no
pli, plf, or palette command has yet been issued).

If non-nil cmin and/or cmax is supplied, values of z beyond these cutoffs are not included in
the frequency counts.

8.2.3   mesh_loc: Get Mesh Location

Calling Sequence ...

```
mesh_loc(y0, x0[, y, x[, ireg]])
```

Description `mesh_loc` returns the zone index (= i+ imax*(j-1)) of the zone of the mesh (x, y)
(with optional region number array ireg) containing the point (x0, y0). If (x0, y0) lies outside
the mesh, returns 0. For example, `ireg(mesh_loc(x0,y0,y,x,ireg))` is the region
number of the region containing (x0, y0). If no mesh specified, uses default. x0 and y0 may
be arrays as long as they are conformable.

8.2.4   mouse: Handle Mouse Click

This function is useful in developing interactive graphics applications.

Calling Sequence ...

```
result = mouse(system, style, prompt)
```

Description mouse displays the specified prompt, then waits for a mouse button to be pressed, then
released. It returns a tuple of length eleven:

```
result = [x_pressed, y_pressed, x_released, y_released,
          xndc_pressed, yndc_pressed, xndc_released,
          yndc_released, system, button, modifiers]
```

If `system>=0`, the first four coordinate values will be relative to that coordinate system. For `system<0`, the first four coordinate values will be relative to the coordinate system under the mouse when the button was pressed.

The second four coordinates are always normalized device coordinates, which start at $(0, 0)$ in the lower left corner of the 8.5x11 sheet of paper the picture will be printed on, with 0.0013 NDC unit being 1/ 72.27 inch (1. 0 point). Look in the style sheet for the location of the viewport in NDC coordinates (see the style keyword).

If style is 0, there will be no visual cues that the mouse command has been called; this is intended for a simple click. If style is 1, a rubber band box will be drawn; if style is 2, a rubber band line will be drawn. These disappear when the button is released.

Clicking a second button before releasing the first cancels the mouse function, which will then return nil. Ordinary text input also cancels the mouse function, which again returns nil.

The left button reverses foreground for background (by XOR) in order to draw the rubber band (if any). The middle and right buttons use other masks, in case the rubber band is not visible with the left button.

result[8] is the coordinate system in which the first four coordinates are to be interpreted. result[9] is the button which was pressed, 1 for left, 2 for middle, and 3 for right (4 and 5 are also possible).

result[10] is a mask representing the modifier keys which were pressed during the operation: 1 for shift, 2 for shift lock, 4 for control, 8 for mod1 (alt or meta), 16 for mod2, 32 for mod3, 64 for mod4, and 128 for mod5.

8.2.5   moush: Mouse in a Mesh

Calling Sequence ...

```
moush([y,x[,ireg]])
```

Description moush returns the 1-origin zone index for the point clicked in for the default mesh, or for the mesh (x, y) (region array ireg).

8.2.6   pause: Pause

Calling Sequence ...

```
pause(milliseconds)
```

Description Pause for the specified number of milliseconds of wall clock time, or until input arrives from the keyboard. This is intended for use in creating animated sequences.

9 Three-Dimensional Plotting Functions

The PyGist 3D graphics uses the PyGist 2D graphics to draw its pictures; most of the 3D routines are computational, and take 3D data in one form or another and massage it until, when plotted, it will appear to be a correct two-dimensional projection of a three-dimensional graph. The usual order of operation in 3D PyGist is

- Retrieve or compute your data;

- Tell PyGist orientation and lighting information;

- Call the appropriate PyGist computational routines;

- Call one or more PyGist 3D plotting routines;

- Call the master function draw3, which actually displays the graph.

PyGist builds a list of information about the graph which you wish to plot, but in its normal operating mode, does not actually draw the graph until you ask it to do so, by invoking draw3. Meanwhile, it stores the information about the graph in a Python list. In this chapter we shall describe the contents of this list in general terms, and the commands which you use to build it (orientation and lighting functions); the setup functions for complicated 3D plots; and the plotting functions themselves. In a final section, for people who may some day be maintaining or adding to this code, we describe the auxiliary functions which everyday users will seldom if ever use.

9.1 Setting Up For 3D Graphics

9.1.1 The Plotting List

The 3D PyGist graphics keeps an internal list called `_draw3_list` containing complete information about the currently active frame (which may or may not be visible depending on whether draw3 has been invoked). Regular users should never need to access this list; however, there is an access function available called `get_draw3_list_` which code developers and maintainers may use to get at the list; `get_draw3_n_` returns the number of elements in the viewing and lighting portion of the list, described below. Likewise, ordinary users do not really need to know the structure of this list in detail; however, every user of the 3D graphics should be aware of the contents of the list, how it affects the graph, and what functions to use to alter it.

`_draw3_list` is a Python list, organized as follows:

```
[rotation, origin, camera_dist, ambient, diffuse, specular,
spower, sdir, fnc 1 , args 1 , fnc 2 , args 2 ,...]
```

The elements of this list are divided into the viewing transformation, lighting specifications, and display information, as follows:

Viewing: rotation: a 3-by-3 rotation matrix giving the angles of view. origin: a 3-vector giving the coordinates of the origin in the user's coordinate system.

   camera_dist: A real number giving the camera distance; the value None (the default) translates to infinity.

Lighting: ambient: a light level (in arbitrary units) that is added to every part of the surface regardless of its orientation. It might be said to be the amount of light which a surface exudes on its own. A surface with ambient of 0 is totally black unless illuminated. diffuse: 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). specular: a light level proportional to a high power spower 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 (a 3 element vector) in the viewer's coordinate system at infinite distance. You can have ns light sources by making specular, spower, and sdir (or any combination) be vectors of length ns (3-by-ns in the case of sdir).

Display: fnc1, fnc2, etc.: Plotting function(s) (whose argument lists are arg 1 , arg 2 , etc., respectively) defining the component(s) of this graph. During its normal operating mode, the 3D graphics accumulates information about calls to plotting functions until the user calls the function draw3. These calls are then executed when draw3 is invoked.

9.1.2   Functions For Setting Viewing Parameters

Angular orientation

Calling Sequences ...

```
    orient3 (phi = angle 1 , theta = angle 2 )
    rot3 (xa = angle x , ya = angle y , za = angle z )
```

Description Note that most of the functions in 3D PyGist accept keyword arguments. These arguments may be entered in any order; omitted arguments will default to a sensible value.

`orient3` sets the orientation of the object to (angle 1 , angle 2). Orientations are a subset of the possible rotation matrices in which the z axis of the object appears vertical on the screen (that is, the object z axis projects onto the viewer y axis). The theta angle is the angle from the viewer y axis to the object z axis, positive if the object z axis is tilted towards you (toward viewer +z). phi is zero when the object x axis coincides with the viewer x axis. If neither phi nor theta is specified, phi defaults to -pi / 4 and theta defaults to pi / 6. If only phi is specified, theta remains unchanged, unless the current theta is near pi / 2, in which case theta returns to pi / 6, or unless the current orientation does not have a vertical z axis, in which case theta returns to its default. If only theta is specified, phi retains its current value. Unlike rot3, orient3 is not a cumulative operation.

`rot3` rotates the current 3D plot by angle x about viewer's x axis, angle y about viewer's y axis, and angle z about viewer's z-axis.

Physical orientation

Calling Sequences ...

```
mov3 (xa = val 1 , ya = val 2 , za = val 3 )
aim3 (xa = val 1 , ya = val 2 , za = val 3 )
setz3 (zc = dist)
```

Description `mov3` moves the current 3D plot by val 1 along the viewer's x axis, val 2 along the viewer's y axis, and val 3 along the viewer's z axis. aim3 moves the current 3D plot to put the point (val 1 , val 2 , val 3) in object coordinates at the point (0, 0, 0)—the aim point—in the viewer's coordinates. In both functions, if any of the val 1 , val 2 , or val 3 is missing, it defaults to 0.

`setz3` sets the camera position to dist (x = y = 0) in the viewer's coordinate system. If dist is None or if zc is missing, set the camera to infinity (default).

9.1.3   Lighting Parameters

Calling Sequence ...

```
light3 (ambient= a_level, diffuse= d_level, specular= s_level,
spower= n, sdir= xyz)
```

This function is used to set the lighting parameters for the current drawing list.

9.1.4   Display List

Calling Sequences ...

```
<plot function> (arg 1 , arg 2 , arg 3 , ...)
clear3 ( )
```

When one of the plotting functions (plwf, pl3surf, pl3tree) is called and the internal variable
_draw3 has been set to zero, or else if it is nonzero and the idler is a do-nothing routine,
Then this plot call will add <plot function> to the display list, and will process the
arguments into a Python list, which will be added to the display list after the function name.

The function clear3 clears the display list of all plotting functions. It leaves orientation and
lighting information unchanged.

9.2   3D Graphics Control Functions

9.2.1   Getting a Window

Calling Sequence ...

```
window3 ( [n] [, dump = val] [, hcp = filename])
```

Description If n is specified, make window n the active window (open a window if necessary). If
n is not specified, connect to the active window, or open one if none is active. Associate the
hardopy file named filename with the window if hcp is specified; this will be postscript if the
name ends in .ps, or cgm if it ends in .cgm. The style sheet associated with the window will
be nobox. gs, i. e., a plain window with no axes (except possibly a gnomon). The dump
keyword, if 1, causes the color palette to be dumped to the hcp file with each frame that is
sent there (otherwise hardcopy plots will be in greyscale).

9.2.2   Displaying the Gnomon

Webster's defines a gnomon as "an object that by its position... serves as an indicator." In 3D PyGist,
the gnomon is a small diagram of the coordinate axes that appears in the lower left corner of a 3D
plot, if this capability has been turned on.

Calling Sequence ...

```
gnomon ( [onoff] [, chr = <labels>] )
set_default_gnomon ( [onoff])
```

Description gnomon toggles the gnomon display if onoff is omitted. If onoff is present and non-zero turn on the gnomon. If zero, turn it off. `set_default_gnomon` allows the user to specify what the default gnomon is to be when the default idler is called (see the discussion in "The variable `_draw3` and the idler" on page 64, and "The Default Idler" on page 64.)

The gnomon shows the x, y, and z axis directions in the object coordinate system. The directions are labeled. The labels default to X, Y, and Z, but may be specified to be something else by using the keyword chr. ¡labels¿ must be a Python list consisting of three character strings. The gnomon is always infinitely far behind the object (away from the camera).

There is a mirror-through-the-screen-plane ambiguity in the display which is resolved in two ways: (1) the (x, y, z) coordinate system is right-handed, and (2) If the tip of an axis projects into the screen, its label is drawn in opposite polarity to the other text in the screen.

9.2.3 Plotting the Display List

The only way that the display list can be plotted is by an invocation of the function draw3. The user may control when this function gets called. To have a new plot appear totally under user control, set _draw3 to 0 (i. e., execute `set_draw3_(0)`) and then call draw3 only when you want the plot to appear. To have a plot appear automatically after each plot command is given, `_draw3` should be set to 1 and the idler should be set to some function which calls draw3. The details are in "The variable `_draw3` and the idler" on page 64.

Calling Sequence ...

```
[lims = ] draw3 ( [called_as_idler = <val>])
limits (lims [0], lims [1], lims [2], lims [3])
```

Description The function `draw3` traverses the display list and executes each function on the list with the list of arguments supplied. Assuming that the list is not empty, this means that the frame specified by this list will be displayed. If the parameter `called_as_idler` is present and is nonzero, then a `fma` (frame advance) call will be made first, meaning that the current display will be erased before the new one is plotted. Otherwise the new display will appear on top of the old.

`draw3` always attempts to return a list of four items `[xmin, xmax, ymin, ymax]` which give the maximum and minimum of the x and y coordinates actually plotted to the PyGist window. Calling the limits function with these four values as limits will scale the graph properly. One could also perform computations with these limits (for example, to force

x and y to the same scale, or to shrink the graph a little to force it well inside the borders of
the window). If you like the way your graphs look, then there is no reason to deal with these
numbers.

We apologize for this messy kludge; we have encountered timing problems and other diffi-
culties with the Gist limits calculating process which we have not been able to solve except
by computing our own limits.

9.2.4   The variable ₋draw3 and the idler

`_draw3` is an internal 3D PyGist variable accessible to the user only by means of the access func-
tions described below. `_draw3`, in conjunction with a function called an idler, determines whether,
after a plot function and its arguments have been placed on the display list, some further action
takes place. The default idler (see below) will cause the graph to be plotted each time it is called;
and it will be called immediately after the plot function has been added to the display list, provided
`_draw3` is nonzero.

Calling Sequences ...

```
set_draw3_ (n)
n = get_draw3_ ( )
```

Description The first function is used to set _draw3 to n (default 0), and the second, to read its
current setting.

Calling Sequences ...

```
clear_idler ( )
set_idler (func_name)
set_default_idler ( )
call_idler ( )
```

Description The function `clear_idler` sets the idler function to a routine which does nothing.
It will be called after each plot function adds to the display list (if _draw3 is nonzero),
but will do nothing. `set_idler` allows the user to define an action for 3D PyGist to take
after each plot function call adds to the display list. `func_name` must be callable with no
arguments. It will be called only if _draw3 is nonzero. `set_default_idler` will set the
idler to call the function whose code is given below. `call_idler` gives you the capability
to call the idler yourself, if you wish.

The Default Idler

Below is the code for the default idler.

```
def _draw3_idler ( ) :
global _default_gnomon
orient3 ()
if current_window () == -1 :
window3 (0)
else :
window3 (current_window ())
gnomon (_default_gnomon)
lims = draw3 (1)
if lims == None :
return
else :
limits (lims [0], lims [1], lims [2], lims [3])
```

9.3   Data Setup Functions for Plotting

9.3.1   Creating a Plane

Calling Sequence ...

```
plane3 (< normal>, <point>)
```

Description This function returns the coefficients of the equation of a plane as a vector of length
four. This is the form of a plane argument as expected by the slicing functions. ¡normal¿ is
a vector of length three giving the direction numbers of the normal to the plane; ¡point¿ is a
vector of length three giving the coordinates of a point in the plane.

9.3.2   Creating a mesh3 argument

The function mesh3 is used to create a mesh3 object from your data. A mesh3 object is required as
an input to a number of routines, most importantly, the various slicing functions.

Calling Sequence ...

```
mesh3 (x, y, z)
mesh3 (x, y, z, funcs = [f1, f2, ...], [verts = <spec>])
```

Description mesh3 creates a mesh3 object as expected by the various functions slice3, xyz3, getv3, etc. The form of a mesh3 object will be described below (See "Description of a mesh3 object" on page 65). Note that Python is able to determine which of the above calls is intended because it can check for the presence of optional and keyword arguments and can check the dimensions and types of the arguments.

In the first two forms of the call, x, y, and z are coordinate arrays specifying the mesh. If x, y, and z are three dimensional of the same shape, then they represent the coordinates of the vertices of a regular rectangular mesh. If x, y, and z are one dimensional of the same size, then the keyword argument verts determines how they are interpreted. If verts is not present, then we have a structured rectangular mesh with unequally spaced nodes. If verts is present, then they represent the coordinates of an unstructured mesh, and the keyword argument verts must be used to pass information about the cells to the mesh3 function. ¡spec¿ can be either a single two dimensional array of integer subscripts into x, y, and z, or a Python list of up to four such objects, one for each type of cell in the mesh. The format of the two dimensional array for each type of cell shape is as follows: hexahedra: the array is `no_hex_cells` by 8. The first subscript is the hexahedron cell number; for each value of this subscript, the second indexes the vertices in canonical order (the first side in outward normal order, the opposite side's corresponding vertices in inward normal order).

prisms: the array is `no_prism_cells` by 6. The first subscript is the prism cell number; for each value of this subscript, the second indexes the vertices in canonical order (the first side in outward normal order, the opposite side's corresponding vertices in inward normal order).

pyramids: the array is `no_pyr_cells` by 5. The first subscript is the pyramid cell number; for each value of this subscript, the second indexes first the apex, then the vertices of the base in inward normal order.

tetrahedra: the array is `no_tet_cells` by 4. The first subscript is the tetrahedron cell number; for each value of this subscript, the second indexes first some arbitrary cell as the apex, then the vertices of the base in inward normal order. Each type of cell has a relative cell number running from 0 to `no_celltype_cells` -1. The cells are also assumed to have absolute cell numbers, which depend on the order in which the defining arrays appear, but will run consecutively starting from 0 in the first cell of the first type up to the total number of cells -1 for the last cell of the last type.

The optional keyword funcs defines f1, f2, etc., which are arrays of function values (e. g. density, temperature) defined on the mesh. In the case of a regular (or structured) rectangular mesh, these functions are 3D arrays. If they represent cell-centered data, they will have one less value along each dimension than the coordinate arrays. If they are vertex-centered data, then they will have the same dimensions. In the case of an unstructured mesh, f1, f2, etc. are one-dimensional arrays. If they represent cell-centered data, then they are indexed by the absolute cell number, and must be the same length as the number of cells. If they represent

vertex-centered data, then they are indexed the same as the vertices, and must be the same length as the vertex arrays.

Calling Sequence ...

```
mesh3 (xyz, funcs = [f1, f2, ...])
```

Description In this case xyz is a four dimensional array specifying the mesh; xyz [0] is the three dimensional x coordinate, xyz [1] is the three dimensional y coordinate, and xyz [2] is the three dimensional z coordinate. (mesh3 actually converts the x, y, z arguments of the first two calls into this xyz form in a mesh3 object; see "Description of a mesh3 object" on page 65. The funcs keyword operates as previously described.

Calling Sequence ...

```
mesh3 (nxnynz, dxdydz, x0y0z0, funcs = [f1, f2, ...])
```

Description nxnynz is a vector of 3 integers, specifying the number of cells of a uniform 3D mesh in the x, y, and z directions, respectively. dxdydz is an array of three reals, specifying the size of the entire mesh, not the size of one cell, in each of the three directions, and x0y0z0 is an array of three reals, representing the point of minimum x, y, and z where the mesh begins. The funcs keyword operates as previously described.

Description of a mesh3 object The form of a mesh3 object varies according to whether the mesh specified was uniform, structured, or unstructured.

Uniform case, node equally spaced: ...

```
[[ xyz3_unif, getv3_rect, getc3_rect, iterator3_rect],
[ (nxnynz [0], nxnynz [1], nxnynz [2]),
array ( [dxdydz, x0y0z0])], [f1, f2, ...]]
```

The four items in the first list are the names of functions. The details of these need not concern us at this time except in their broad outlines. The iterator3 function will split the mesh into chunks for processing by the slicing functions, if necessary, in order to save space. xyz3 returns the vertex coordinates of a chunk. getv3 returns the vertex values of a function on the chunk; getc3 returns cell values. Because these routines necessarily differ depending on the type of mesh, their names are passed along with the mesh specifications to that the appropriate ones can be called. The remainder of the items in the object specify the mesh and the function(s) defined on the mesh (if any; if there are none, the final list will be []).

Uniform case, nodes unequally spaced: ...

```
[[ xyz3_unif, getv3_rect, getc3_rect, iterator3_rect],
[ (len (x) -1, len (y) -1, len (z) -1),
array ( [x, y, z])], [f1, f2, ...]]
```

The functions' purpose is as described above. x, y, and z are one-dimensional arrays, possibly of different lengths, specifying the node coordinates of a uniform rectangular mesh, which might be unequally spaced. The triple consisting of the three array lengths minus one gives the size of the mesh in cells.

Structured case: ...

```
[[ xyz3_rect, getv3_rect, getc3_rect, iterator3_rect],
[dim_cell, xyz], [f1, f2, ...]]
```

The functions' purpose is as described above. `dim_cell` is an integer vector of length three giving the size of the mesh in cells, `dim_cell[0]` being the x direction size, `dim_cell[1]` the y, and `dim_cell[2]` the z. `xyz` is a four-dimensional array of coordinates; `xyz[0]` is the three dimensional x coordinate array, `xyz[1]` is the three dimensional y coordinate array, and `xyz[2]` is the three dimensional z coordinate array.

Unstructured case: ...

```
[[ xyz3_irreg, getv3_irreg, getc3_irreg, iterator3_irreg],
[dims, array ( [x, y, z]), sizes, totals], [f1, f2, ...]]
```

The functions' purpose is as described above. dims is the value of the keyword argument verts, i. e., it represents one array, or a list of up to four arrays, specifying the subscripts of the cell vertices into arrays x, y, and z in canonical order. If there is only one type of cell in the unstructured mesh, then sizes and totals will not be present; otherwise, they are used to help recover the absolute cell number from a cell's index in the list of cells of the same type. sizes [i] is the number of cells of type i; totals [i] is the total number of cells up to and including type i.

9.4   The Slicing Functions

The slicing functions must be called in order to create data appropriate for the pl3surf (plot a 3D surface) and pl3tree (add a plot to a tree) routines. In general, the slicing routines take a mesh specification of some sort and return a list of the form

```
[nverts, xyzverts, color]
```

which specifies a set of polygonal cells and how to color them.

nverts is `no_cells` long and the i th entry tells how many vertices the i th cell has.

xyzverts is sum (nverts) by 3 and gives the vertex coordinates of the cells in order, i. e., the first `nverts[0]` entries in `xyzverts` are the coordinates of the first polygon's vertices, the next `nverts[1]` entries are the coordinates of the second polygon's vertices, etc.

color , if present, is `no_cells` long and contains a color value for each cell in the mesh.

9.4.1 slice3mesh: Pseudo-slice for a surface

The function slice3mesh is designed specifically to produce an input argument for pl3surf, although if you want more than one surface in a picture, it can also be fed to pl3tree. It has several distinct calling sequences, which Python can distinguish with its type savvy.

Calling Sequence ...

```
slice3mesh (z [, color])
```

Description z is a two dimensional array of function values, assumed to be on a uniform mesh nx by ny cells (assuming z is nx by ny) nx being the number of cells in the x direction, ny the number in the y direction. color, if specified, is either an nx by ny array of cell-centered values by which the surface is to be colored, or an nx+1 by ny+1 array of vertex-centered values, which will be averaged over each cell to give cell-centered values.

Calling Sequence ...

```
slice3mesh (nxny, dxdy, x0y0, z [, color])
```

Description In this case, slice3mesh accepts the specification for a regular 2D mesh: nxny is the number of cells in the x direction and the y direction; `x0y0` are the initial values of x and y; and dxdy are the increments in the two directions. z is the height of a surface above the xy plane and must be dimensioned nx + 1 by ny + 1. color, if specified, is as above.

Calling Sequence ...

```
slice3mesh (x, y, z [, color])
```

Description z is as above, an nx by ny array of function values on a mesh of the same dimensions. There are two choices for x and y: they can both be one-dimensional, dimensioned nx and ny respectively, in which case they represent a mesh whose edges are parallel to the axes; or else they can both be nx by ny, in which case they represent a general quadrilateral mesh.

9.4.2   slice3: Plane and Isosurface Slices of a 3D mesh

Calling Sequence ...

```
[nverts, xyzverts, color] = \
slice3 (m3, fslice, nv, xyzv [, fcolor [, flg 1 ]]

[, value = <val>] [, node = flg 2 ])
```

Description Slice the 3D mesh m3 as specified by fslice, returning the list [nverts, xyzverts, color]. nverts is the number of vertices in each polygon of the slice, and xyzverts is the 3-by-sum (nverts) list of polygon vertices. If the fcolor argument is present, the values of that coloring function on the polygons are returned as the value of color. color will have the same size as nverts, i. e., the number of polygons in the slice, except that the keyword argument node, if present and nonzero, is a signal to return node-centered values rather than cell-centered values. In the latter case color will be sum (nverts) long and entries in color will be associated with the corresponding coordinates in xyzverts. nv and xyzv are not needed; None should be passed as their values (this is a leftover from an older version of the code).

fslice can be a function, a vector of 4 reals, or an integer number. If fslice is a function, it should be of the form:

```
def fslice (m3, chunk)
or, in the case of an isosurface slice,
def fslice (m3, chunk, iso_index, _value)
or for a plane slice,
def fslice (m3, chunk, normal, projection)
```

and should return a list of function values on the specified chunk of the mesh m3. Module slice3 offers plane and isosurface slicers (for descriptions, see page 70). If you wish to write your own slice routine, you should bear in mind that the format of chunk depends on the type of m3 mesh, so you should use only the appropriate mesh functions xyz3 and getv3 which take that type of m3 and chunk as arguments. The return value of fslice should have the same dimensions as the return value of getv3; the return value of xyz3 has an additional first dimension of length 3.

If fslice is a list of 4 reals, it is taken as a slicing plane as returned by plane3.

If fslice is a single integer, the slice will be an isosurface for the fslice th function associated with the mesh m3. In this case, the keyword value must also be present, representing the value of that function on the isosurface.

If fcolor is omitted or has value None, then slice3 returns None as the value of color.. If you want to color the polygons in a manner that depends only on their vertex coordinates (e. g., by a 3D shading calculation), use this mode.

fcolor can be a function or a single integer. If fcolor is a function, it should be of the form: def fcolor (m3, cells, l, u, fsl, fsu, ihist) and should return a list of function values on the specified cells of the mesh m3. If the optional argument flg 1 after fcolor is not missing or None and is non-zero, then the fcolor function is called with only two arguments:

```
def fcolor (m3, cells)
```

The cells argument will be the list of cell indices in m3 at which values are to be returned. l, u, fsl, fsu, and ihist are interpolation coefficients which can be used to interpolate from vertex centered values to the required cell centered values, ignoring the cells argument. See getc3 source code. The return values should always have the same size and shape as cells.

If fcolor is a single integer, then the slice will be an isosurface for the fcolor th variable associated with the mesh m3.

9.4.3   slice2 and slice2x: Slicing Surfaces with planes

The functions slice2 and slice2x allow one to slice surfaces specified by slice3-type output. slice2 will return the portion on one side of the slicing plane; slice2x will return both portions.

Calling Sequences ...

```
[nverts, xyzverts, values] = slice2 (plane, nv,
xyzv, vals)
[nverts, xyzverts, values, nvertb, xyzvertb, valueb] =
slice2x (plane, nv, xyzv, vals)
```

Description The argument plane can be either a scalar or a plane3 (see "Creating a Plane" on page 65); nv is an array of integers, the i th entry of which gives the number of vertices of the i-th polygonal cell; `xyzv` are the vertices of the coordinates of the cells, with each consecutive nv [i] entries representing the

vertices of the i th cell; and vals being a set of values as explained below. These arguments are the same format as returned by slice3 and slice3mesh.

If plane is a plane3, then vals (if not None) is a cell-centered set of values expressing the color of each cell, and the outputs nverts, xyzverts, and values represent the polygons and their colors (if any) describing the portion of the sliced surface that is on the positive side of the plane. That's all you get with slice2. With slice2x, you get in addition nvertb, xyzvertb, and valueb, which describe the part of the surface on the negative side of the slicing plane. Warning: one of these specifications could be None, None, None if the entire surface lies on one side of the plane.

If plane is a scalar value, then vals must be present and must be node-centered. In this case, the outputs nverts, xyzverts, and values represent the polygons and their colors (if any) describing the portion of the sliced surface where vals on the vertices are greater than or equal to the scalar value plane. (This actually allows you to form an arbitrary two-dimensional slice of a surface.) With slice2x, you get in addition nvertb, xyzvertb, and valueb, which describe the part of the surface where vals on the vertices are less than the scalar value plane.

9.5   The 3D Plotting Functions

9.5.1   plwf: plot a wire frame

Calling Sequence ...

```
plwf (z [, y, x] [, <keylist>] )
```

Description plwf plots a 3D wire frame of the given 2D array z. If x and y are given, then they must be the same shape as z or else len (x) should be the first dimension of z and len (y) the second. If x and y are not given, they default to the first and second indices of z, respectively. plwf calls clear3 before putting the plot command on the display list, which means that PyGist can only show one wire frame at a time using this function. (See pl3tree for graphs with multiple components).

plwf accepts the following keyword arguments:

fill, shade, edges, ecolor, ewidth, cull, scale, cmax A description of the keywords follows: fill: optional colors to use (default is to make zones have background color), same dimension options as for z argument to plf function, i. e., it should be the same dimension as the mesh (vertex-centered values) or one smaller in each dimension (cell-centered values).

shade: set non-zero to compute shading from the current 3D lighting sources.

edges: default is 1 (draw edges), but if you provide fill colors, you may set to 0 to suppress the edges.

ecolor, ewidth: color and width of edges. cull: default is 1 (cull back surfaces), but if you want to see the "underside" of the model, set to 0.

scale: by default, z is scaled to "reasonable" maximum and minimum values related to the scale of (x, y). This keyword alters the default scaling factor, in the sense that scale = 2.0 will produce twice the z-relief of the default scale = 1.0.

cmax: the ambient keyword in light3 can be used to control how dark the darkest surface is; use this to control how light the lightest surface is. The lightwf routine can change this parameter interactively.

Examples The following example computes the information for a surface with a peak and a valley,
and then plots the resulting wire frame with various options. In the first case, we see simply
an opaque wire frame.

```
set_draw3_(0)
x = span (-1, 1, 64, 64)
y = transpose (x)
z = (x + y) * exp (-6.*( x* x+ y* y))
orient3 ( )
light3 ( )
plwf (z, y, x)
[xmin, xmax, ymin, ymax] = draw3( 1)
limits (xmin, xmax, ymin, ymax)
plt(" opaque wire mesh", .30, .42)
```

Next, we see the same surface shaded from a default light source (roughly over the viewer's
right shoulder) and with the mesh lined in red.

```
plwf( z, y, x, shade= 1, ecolor=" red")
[xmin, xmax, ymin, ymax] = draw3( 1)
limits (xmin, xmax, ymin, ymax)
```

Finally, the following sequence plots the same surface with no edges, and with lighting com-
ing from the back.

```
plwf( z, y, x, shade= 1, edges= 0)
light3 ( diffuse=. 1, specular= 1., sdir= array([ 0,0,-1]))
[xmin, xmax, ymin, ymax] = draw3( 1)
limits (xmin, xmax, ymin, ymax)
```

9.5.2   pl3surf: plot a 3-D surface

Calling Sequence ...

```
pl3surf (nverts, xyzverts [, values] [, <keylist>])
```

Description Perform simple 3D rendering of an object created by slice3 (possibly followed by
slice2). nverts and xyzverts are polygon lists as returned by slice3, so xyzverts is sum (nverts)-
by-3, where nverts is a list of the number of vertices in each polygon. If present, the values
should have the same length as nverts; they are used to color the polygon. If values is not

specified, the 3D lighting calculation set up using the light3 function will be carried out. Keywords cmin and cmax as for plf, pli, or plfp are also accepted. (If you do not supply values, you probably want to use the ambient keyword to light3 instead of cmin here, but cmax may still be useful.)

pl3surf calls clear3 before putting the plot command on the display list, which means that PyGist can only show one surface at a time using this function. (See pl3tree below for graphs with multiple components).

Example The following example is the familiar sombrero function. The first few lines of code compute its value.

```
nc1 = 100
nv1 = nc1 + 1
br = -(nc1 / 2)
tr = nc1 / 2 + 1
x = arange (br, tr, typecode = Float) * 40. / nc1
y = arange (br, tr, typecode = Float) * 40. / nc1
z = zeros ( (nv1, nv1), Float)
r = sqrt (add. outer ( x ** 2, y ** 2)) + 1e-6
z = sin (r) / r
```

In order to use pl3surf, we need to construct a mesh using mesh3. The way we shall do that is to define a function on the 3d mesh so that the sombrero function is its 0-isosurface.

```
z0 = min (ravel (z))
z0 = z0 -.05 * abs (z0)
maxz = max (ravel (z))
maxz = maxz + .05 * abs (maxz)
zmult = max (max (abs (x)), max (abs (y)))
dz = (maxz -z0)
nxnynz = array ( [nc1, nc1, 1], Int)
dxdydz = array ( [1.0, 1.0, zmult* dz], Float )
x0y0z0 = array ( [float (br), float (br), z0* zmult], Float )
meshf = zeros ( (nv1, nv1, 2), Float )
meshf [:, :, 0] = zmult* z -(x0y0z0 [2])
meshf [:, :, 1] = zmult* z -(x0y0z0 [2] + dxdydz [2])
```

Finally, we create the mesh and call the plotting functions.

```
m3 = mesh3 (nxnynz, dxdydz, x0y0z0, funcs = [meshf])
fma ()
# Make sure we don't draw till ready
```

```
set_draw3_(0)
pldefault( edges= 0)
[nv, xyzv, col] = slice3 (m3, 1, None, None, value = 0.)
orient3 () # (default orientation)
pl3surf (nv, xyzv)
lim = draw3 (1)
dif = 0.5 * (lim [3] -lim [2])
# dif is used to compress the y scale a bit.
limits (lim [0], lim [1], lim [2] -dif, lim [3] + dif)
palette (" gray. gp")
```

The graph that results from this sequence of code is on the next page. This next sequence of functions uses slice3mesh to draw the same surface; this time the polygons that make up the surface are colored according to height (using the rainbow palette).

```
# Try new slicing function to get color graph
[nv, xyzv, col] = slice3mesh (nxnynz [0: 2], dxdydz [0: 2],
x0y0z0 [0: 2], zmult * z, color = zmult * z)
pl3surf (nv, xyzv, values = col)
lim = draw3 (1)
dif = 0.5 * (lim [3] -lim [2])
limits (lim [0], lim [1], lim [2] -dif, lim [3] + dif)
palette (" rainbow. gp")
```

9.5.3   pl3tree: add a surface to a plotting tree

pl3tree accepts surfaces and slices of surfaces in the slice2/ slice3 format, and, as its name suggests, builds a b-tree. Its purpose is to attempt to analyze multiple surface plots in such a way as to determine the order of plotting, so that hidden portions of the surfaces will be graphed first, and this covered by later portions. pl3tree may be called multiple times to build plots of arbitrary complexity.

Calling Sequence ...

```
pl3tree (nverts, xyzverts [, values] [, <keylist>])
```

Description pl3tree accepts the following keywords: plane, cmin, cmax, split pl3tree adds the polygon list specified by nverts (number of vertices in each polygon) and xyzverts (3-by-sum (nverts) vertex coordinates) to the currently displayed b-tree. If values is specified, it must have the same dimension as nverts, and represents the color of each polygon. If values is not specified, then the polygons are assumed to form an isosurface which will be shaded by the

current 3D lighting model; the isosurfaces are at the leaves of he b-tree, sliced by all of the planes. If plane (in the format returned by a call to plane3) is specified, then the xyzverts must all lie in that plane, and that plane becomes a new slicing plane in the b-tree.

Each leaf of the b-tree consists of a set of sliced isosurfaces. A node of the b-tree consists of some polygons in one of the planes, a b-tree or leaf entirely on one side of that plane, and a b-tree or leaf on the other side. The first plane you add becomes the root node, slicing any existing leaf in half. When you add an isosurface, it propagates down the tree, getting sliced at each node, until its pieces reach the existing leaves, to which they are added. When you add a plane, it also propagates down the tree, getting sliced at each node, until its pieces reach the leaves, which it slices, becoming the nodes closest to the leaves.

This structure is relatively easy to plot, since from any viewpoint, a node can always be plotted in the order from one side, then the plane, then the other side.

If keyword split is set nonzero (the default), then this routine assumes a "split palette"; the current palette will be "split" or truncated so that its colors are numbered 0 to 99, while colors 100 to 199 will be greyscale. Colors for the values will be scaled to fit from color 0 to color 99, while the colors from the shading calculation will be scaled to fit from color 100 to color 199. (If values is specified as an unsigned char array (Python typecode "b"), however, it will be used without scaling.) You may specify a cmin or cmax keyword to affect the scaling; cmin is ignored if values is not specified (use the ambient keyword from light3 for that case).

Example In the following example, nx, ny, and nz are each 20. First we compute the mesh and some data on the mesh.

```
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)
m3 = mesh3 (xyz, funcs = [r * (1. + y32)])
```

Next we construct two isosurfaces, an inner (function value .5) and an outer (function value 1.0) using slice3.

```
[nv, xyzv, dum] = slice3 (m3, 1, None, None, value = .50)
```

```
# (inner isosurface)
[nw, xyzw, dum] = slice3 (m3, 1, None, None, value = 1.)
# (outer isosurface)
```

Now we create two planes, use one to form a plane slice through the mesh, then the second to slice the first in half.

```
pxy = plane3 ( array ([ 0, 0, 1], Float ), zeros (3, Float))
pyz = plane3 ( array ([ 1, 0, 0], Float ), zeros (3, Float))
[np, xyzp, vp] = slice3 (m3, pyz, None, None, 1)
# (pseudo-colored plane slice)
[np, xyzp, vp] = slice2 (pxy, np, xyzp, vp)
# (cut slice in half)
```

Finally, we slice each isosurface in half, keeping both halves (slice2x calls), then slice the "top" half of each in half again, discarding the front of each (slice2 calls).

```
[nv, xyzv, d1, nvb, xyzvb, d2] = \
slice2x (pxy, nv, xyzv, None)
[nv, xyzv, d1] = slice2 (-pyz, nv, xyzv, None)
# (... halve one of those halves)
[nw, xyzw, d1, nwb, xyzwb, d2] = \
slice2x ( pxy , nw, xyzw, None)
# (split outer in halves)
[nw, xyzw, d1] = slice2 (-pyz, nw, xyzw, None)
```

Now, a sequence of calls to pl3tree sets up the graph, and a call to demo5_light actually plots it. For completeness, we give the function demo5_light first.

```
making_movie = 0
def demo5_light (i) :
global making_movie
if i >= 30 : return 0
theta = pi / 4 + (i -1) * 2 * pi/ 29
light3 (sdir = array ( [cos( theta), .25, sin( theta)],
Float))
draw3 ( not making_movie )
return 1
fma ()
split_palette (" earth. gp")
gnomon (1)
```

```
clear3 ()
# Make sure we don't draw till ready
set_draw3_(0)
pl3tree (np, xyzp, vp, pyz)
pl3tree (nvb, xyzvb)
pl3tree (nwb, xyzwb)
pl3tree (nv, xyzv)
pl3tree (nw, xyzw)
orient3 ()
light3 (diffuse = .2, specular = 1)
limits (square= 1)
demo5_light (1)
```

9.6   Contour Plotting on Surfaces: plzcont and pl4cont

Contour lines can be plotted on a surface, or filled contours can be drawn, or both, by means of the two functions plzcont (plot z contours, i. e., contours according to height in the z direction) and pl4cont (plot 4D contours, i. e., contours determined by some other function defined on the surface.

Calling Sequences ...

```
plzcont (nverts, xyzverts, contours = 8, scale = "lin",
clear = 1, edges = 0, color = None, cmin = None,
cmax = None, zaxis_min = None, zaxis_max = 0, split = 0)
pl4cont (nverts, xyzverts, values, contours = 8, scale =
"lin", clear = 1, edges = 0, color = None, cmin = None,
cmax = None, caxis_min = None, caxis_max = 0, split = 0)
```

Description plzcont plots z contours, and pl4cont plots contours derived from the function values. nverts and xyzverts specify the polygons which define the surface. nverts is an array of integers,

the i th entry of which gives the number of vertices of the i th polygonal cell; xyzverts are the vertices of the coordinates of the cells, with each consecutive nv [i] entries representing the vertices

of the i-th cell; and values (for pl4cont) being a set of values, one for each vertex. These arguments are the same format as returned by slice3 and slice3mesh (see Section "slice3: Plane and Isosurface Slices of a 3D mesh" on page 70). plzcont and pl4cont actually do repeated calls to slice2x (see Section "slice2 and slice2x: Slicing Surfaces with planes" on page 71) in order to obtain the contour curves.

Keyword Arguments contours can be one of the following: N, an integer: Plot N contours (therefore, N+ 1 colored components of the surface) CVALS, a vector of floats: draw the contours at the specified levels.

scale can be `"lin"`, `"log"`, or `"normal"` specifying the contour scale. (Only applicable if contours = N, of course).

clear If `clear == 1`, clear the display list first. Otherwise the current contour plot will be added to the display list.

edges If `edges == 1`, plot the edges.

color If `color == None`, then bytscl the palette into N + 1 colors and send each of the slices to pl3tree with the appropriate color. If `color=="bg"`, will plot only the edges. See also split (below).

cmin, cmax If `cmin` is given, use it instead of the minimum c actually being plotted in the computation of contour levels. If `cmax` is given, use it instead of the maximum c actually being plotted in the computation of contour levels. This is done so that a component of a larger graph will have the same colors at the same levels as every other component, rather than its levels being based on its own maximum and minimum, which may lie inside or outside those of the rest of the graph.

zaxis_min, zaxis_max `zaxis_min` and `zaxis_max` represent axis limits on z as expressed by the user. If present, `zaxis_min` will inhibit plotting of all lesser z values, and `zaxis_max` will inhibit the plotting of all greater z values.

caxis_min, caxis_max `caxis_min` and `caxis_max` represent axis limits on c as expressed by the user. If present, `caxis_min` will inhibit plotting of all lesser c values, and `caxis_max` will inhibit the plotting of all greater c values.

split If split == 1, then it is intended to plot this portion of the graph as if the palette has been split, so only colors 0-99 will be used to color the contours. If split == 0, then all colors from 0 to 199 will be used.

Example In the following example, we compute the sombrero function and then use plzcont to draw it with contours in `"normal"` scale. In `"normal"` scale, the top and bottom contours are two standard deviations away from the mean. Thus the peak of the sombrero is all the same color because its few points contribute very little to the standard deviation.

```
# compute sombrero function
x = arange (-20, 21, typecode = Float)
y = arange (-20, 21, typecode = Float)
z = zeros ( (41, 41), Float)
r = sqrt (add. outer ( x ** 2, y ** 2)) + 1e-6
z = sin (r) / r
fma ()
clear3 ()
```

```
gnomon (0)
# Make sure we don't draw till ready
set_draw3_(0)
palette (" rainbow. gp")
[nv, xyzv, dum] = slice3mesh (x, y, z)
plzcont (nv, xyzv, contours = 20, scale = "normal")
[xmin, xmax, ymin, ymax] = draw3 (1)
limits (xmin, xmax, ymin, ymax)
```

To draw the same function in `"lin"` scale, with edges visible, enter the following code:

```
plzcont (nv, xyzv, contours = 20, scale = "lin", edges= 1)
[xmin, xmax, ymin, ymax] = draw3 (1)
limits (xmin, xmax, ymin, ymax)
```

9.7   Animation: movie and spin3

9.7.1   The movie module and function

Calling Sequence ...

```
movie (draw_frame [, time_limit = 120.]
[, min_interframe = 0.0]
[, bracket_time = array ([ 2., 2.], Float )]
[, lims = None]
[, timing = 0])
```

Description   Note: All but the first argument are keyword arguments, with defaults as shown. This
function runs a movie based on the given `draw_frame` function. The movie stops after a
total elapsed time of `time_limit` seconds, which defaults to 60 (one minute), or when the
`draw_frame` function returns zero. (N. B. Currently the timing option described here and
in a subsequent paragraph is not completely implemented.)

```
draw_frame is a function described as follows:
def draw_frame (i) :
# Input argument i is the frame number.
# draw_frame should return non-zero if there are more
# frames in this movie. A zero return will stop the
# movie.
```

```
# draw_frame must NOT include any fma command if the
# making_movie variable is set (movie sets this variable
# before calling draw_frame)
```

If `min_interframe` is specified, a pause will be added as necessary to slow down the movie. `min_interframe` is a time in seconds (default 0). The keyword `bracket_time` (again a time in seconds) can be used to adjust the duration of the pauses after the first and last frames. It may also be a two element array [beg, end]. If the pause at the end is greater than five seconds, you will be prompted to explain that hitting ¡RETURN¿ will abort the final pause. (Well, the Python version does not currently have this capability due to the difficulty of implementing it consistently over various platforms.)

timing = 1 enables a timing printout for your movie. If every frame of your movie has the same limits, use the lims keyword argument to fix the limits during the movie.

Example In the following example, the movie demonstrates the effect of a moving light source on the currently drawn surface. (The plot functions creating the surface have not been shown; it is assumed that the data for the surface is on the current display list.)

The `draw_frame` function is as follows:

```
def demo5_light (i) :
global making_movie
if i >= 30 : return 0
theta = pi / 4 + (i -1) * 2 * pi/ 29
light3 (sdir =
array ( [cos( theta), .25, sin( theta)], Float))
# without an explicit call to draw3, the light3
# function would cause no changes until Python
# paused for input from the keyboard, since
# unlike the primitive plotting functions (plg, plf,
# plfp, ...) the fma call made by the movie function
# will not trigger the 3-D display list. any movie
# frame display function which uses the 3-D drawing
# functions in pl3d. py will need to do this. the
# !making_movie flag supresses the fma in draw3 if
# this function is called by movie (which issues
# its own fma), but allows it otherwise
draw3 ( not making_movie )
return 1
```

Here is the Python code necessary to run a movie. This particular animation shows the surface with a peak and valley which we saw earlier in this chapter (See "Examples" on page 73), with a moving light source. A few frames of the movie are shown on the next page.

```
set_draw3_(0)
x = span (-1, 1, 64, 64)
y = transpose (x)
z = (x + y) * exp (-6.*( x* x+ y* y))
orient3 ( )
light3 (diffuse=. 2, specular= 1 )
limits_( square = 1)
plwf (z, y, x, shade= 1, edges= 0)
[xmin, xmax, ymin, ymax] = draw3 (1)
limits (xmin, xmax, ymin, ymax)
making_movie = 1
movie( demo5_light, lims = [xmin, xmax, ymin, ymax])
making_movie = 0
```

9.7.2   The spin3 function

spin3 is a function which takes an existing 3D plot and spins it about an axis. It actually calls movie
for you, with a `draw_frame` function which is internal to the pl3d module and not available
outside this module, because its name begins with an underscore.

Calling Sequence ...

```
spin3 (nframes = 30,
axis = array ([-1, 1, 0], Float),
tlimit = 60.,
dtmin = 0.0,
bracket_time = array ([ 2., 2.], Float),
lims = None,
timing = 0,
angle = 2. * pi)
```

Description Spin the current 3D display list about axis (default `[-1,1,0]`) over nframes (default
30). Note that all arguments are keywords. Also note that the timing keywords are allowed
but are not currently implemented. Their meanings are:

tlimit: the total time allowed for the movie in seconds (default 60).

dtmin: the minimum allowed interframe time in seconds (default 0.0).

bracket_time: (as for movie function in movie.py).

lims: the axis limits, if you wish to specify them. timing = 1 if you want timing measured
and printed out, 0 if not.

angle: the total angle about the axis through which the object will be rotated. During each step of the rotation, the object will rotate angle / (nframes-1).

Example In this example, we take the surface discussed previously (see "Example" on page 81) and rotate it about an axis. Assume that the sequence of code given there has been executed, giving the figure shown there. Then we do the following to run the movie:

```
spin3 () # (lims = [l0, l1, l2, l3])
```

Four frames from the resulting movie are shown on the next page.

TABLE 2. Frames from Movie of Rotating Isosurfaces

9.8 Syntactic Sugar: Some Helpful Functions

9.8.1 Specifying the palette to be split: split_palette

Calling Sequence ...

```
split_palette ( [palette_name])
```

Description Split the current palette (if `palette_name` is not present) or the specified palette (if `palette_name` is present) into two parts; colors 0 to 99 will be a compressed version of the original, while colors 100 to 199 will be a gray scale. For details on the available palettes, see "palette: Set or Retrieve Palette" on page 27.

If you use `split_palette` to split the palette yourself, then be sure to call pl3tree with keyword split = 0, because otherwise pl3tree will split it again, with bizarre results. Alternatively, you can use the palette function referenced above to set the palette to your choice, then call pl3tree with split = 1.

9.8.2 Saving and restoring the view and lighting: save3, restore3

Calling Sequences ...

```
view = save3 ()
movie (_spin3, ... <other arguments>)
restore3 (view)
```

Description In the above, the `save3` function returns a copy of the current 3D viewing transformation and lighting, so that the user can put it aside in the variable view. The `_spin3` function does actually change the viewing transformation and lighting; the call to restore3 with argument view sets it back to its previous configuration.

References

[1] Lee Busby, Fred Fritsch, and Zane Motteler, "Manual 4, Chapter 1: The Python Graphics Interface," UCRL-MA-128569, November 23, 1998.

10 Acknowledgements

The following people contributed to the PyGist development over the years:

- **Lee Busby** (LLNL) wrote the original extension wrapping Gist for use from Python (`gistCmodule.c`). Work was based on Yorick 1.3.

- **Zane Motteler** (retired LLNL) was responsible for the object-oriented component of the original Python Graphics (PyGraph), EZPlot, and the Narcisse module (all three are no longer in use). Wrote much of the documentation external to documentation contributed from Steve Langer and Dave Munro for their software. Also maintained the entire graphics interface (PyGraph) until retirement.

- **Fred Fritsch** (retired LLNL) developed the template for the documentation and contributed to some of the original PyGraph documentation.

- **Dave Munro** (LLNL), author of Yorick (and Gist). Also, identified the subset of Yorick 1.5 required for PyGist. Helped identify the technical issues, particularly related to event handling, involved with interfacing Gist (1.5) to Python 2.x. Also contributes routinely by making Yorick changes to accommodate PyGist.

- **Steve Langer** (LLNL), Yorick developer. Collaborated with Munro on the interpreted 3D graphics. Also contributed to identification of issues in bringing up PyGist 1.3 to 1.5.

- **Paul Dubois** (LLNL) wrote the PDB and Ranf modules, and 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.

- **Michiel de Hoon** (University of Tokyo, Human Genome Center) contributed solutions to the problem of getting Python and Gist to work together in handling events. He rewrote the build procedure (setup.py) and was also responsible for the Cygwin port and collaborated with Munro on the Windows port to 1.5. Michiel also provided the Windows-related documentation in this manual.

- **Lila Chase** (LLNL) currently maintains PyGist (2001-). Gathered relevant pieces from PyGist 1.3, upgraded to Yorick 1.5 and latest Python and Numeric, created new infrastructure, ported, and tested on a wide variety of platforms.

A Useful Functions for Developers

November 23, 1998

In this chapter we describe more of the available functions in detail, for those who are really interested in plumbing the depths of the low-level 3D graphics.

A.1 Find 3D Lighting: get3_light

Calling Sequence ...

```
get3_light(xyz [, nxyz])
```

Description Return 3D lighting for polygons with vertices XYZ. If NXYZ is specified, XYZ should be sum(nxyz)-by-3, with NXYZ being the list of numbers of vertices for each polygon (as for the plfp function; see page 42). If NXYZ is not specified, XYZ should be a quadrilateral mesh, ni-by-nj-by-3 (as for the plf function; see page 40). In the first case, the return value is len (NXYZ) long; in the second case, the return value is (ni-1)-by-(nj-1).

The parameters of the lighting calculation are set by the light3 function (see "Lighting Parameters" on page 61).

A.2 Get Normals to Polygon Set: get3_normal

Calling Sequence ...

```
get3_normal(xyz [, nxyz])
```

Description Return 3D normals for polygons with vertices XYZ. If NXYZ is specified, XYZ should be sum(nxyz)-by-3, with NXYZ being the list of numbers of vertices for each polygon (as for the plfp function; see page 42). If NXYZ is not specified, XYZ should be a quadrilateral mesh, ni-by-nj-by-3 (as for the plf function; see page 40). In the first case, the return value is len(NXYZ)-by-3; in the second case, the return value is (ni-1)-by-(nj-1)-by-3.

The normals are constructed from the cross product of the lines joining the midpoints of two edges which as nearly quarter the polygon as possible (the medians for a quadrilateral). No check is made that these not be parallel; the returned "normal" is [0, 0, 0] in that case. Also, if the polygon vertices are not coplanar, the "normal" has no precisely definable meaning.

A.3   Get Centroids of Polygon Set: get3_centroid

Calling Sequence ...

```
get3_centroid(xyz [, nxyz])
```

Description Return 3D centroids for polygons with vertices XYZ. If NXYZ is specified, XYZ should be sum(nxyz)-by-3, with NXYZ being the list of numbers of vertices for each polygon (as for the plfp function; see page 42). If NXYZ is not specified, XYZ should be a quadrilateral mesh, ni-by-nj-by-3 (as for the plf function; see page 40). In the first case, the return value is len(NXYZ) in length; in the second case, the return value is (ni-1)-by-(nj-1)-by-3.

The centroids are constructed as the mean value of all vertices of each polygon.

A.4   Get Viewer's Coordinates: get3_xy

Calling Sequence ...

```
get3_xy(xyz [, 1])
```

Description Given 3-by-anything coordinates XYZ, return X and Y in viewer's coordinate system (set by rot3, mov3, orient3, etc.; see "Functions For Setting Viewing Parameters" on page 60). If the second argument is present and non-zero, also return Z (for use in sort3d or `get3_light`, for example; see "Sort z Coordinates: sort3d" on page 89 and "Find 3D Lighting: get3_light" on page 87.). If the camera position has been set to a finite distance with setz3 (see "Physical orientation" on page 61), the returned coordinates will be tangents of angles for a perspective drawing (and Z will be scaled by 1/zc). x, y, and z can be either 1D or 2D, so this routine is written in two cases.

A.5   Add object to drawing list: set3_object

Calling Sequence ...

```
set3_object(drawing_function, [arg1, arg2,...])
```

Description Set up to trigger a call to draw3, adding a call to the3D display list of the form:

```
DRAWING_FUNCTION ([ARG1, ARG2, ...]))
```

When draw3 calls `DRAWING_FUNCTION`, the external variable `draw3_` will be non-zero, so `DRAWING_FUNCTION` can be written like this:

```
def drawing_function(arg) : if (draw3_) :
arg1=arg[0]
arg1=arg[1]
...
...< calls to get3_xy, sort3d, get3_light, etc.>...
...< calls to graphics functions plfp, plf, etc.>...
return


...< verify args>...
...< do orientation and lighting independent calcs>...
set3_object (drawing_function, [arg1, arg2,...])
```

A.6   Sort z Coordinates: sort3d

Calling Sequence ...

```
sort3d(z, npolys)
```

Description Given Z and NPOLYS, with len(Z)== sum(npolys), return a 2-element list [LIST, VLIST] such that take(Z, VLIST) and take (NPOLYS, LIST) are sorted from smallest average Z to largest average Z, where the averages are taken over the clusters of length NPOLYS. Within each cluster (polygon), the cyclic order of take (Z, VLIST) remains unchanged, but the absolute order may change.

This sorting order produces correct or nearly correct order for a plfp call to make a plot involving hidden or partially hidden surfaces in three dimensions. It works best when the polygons form a set of disjoint closed, convex surfaces, and when the surface normal changes only very little between neighboring polygons. (If the latter condition holds, then even if sort3d misorders two neighboring polygons, their colors will be very nearly the same, and the mistake won't be noticeable.) A truly correct 3D sorting routine is impossible, since there may be no rendering order which produces correct surface hiding (some polygons may need to be split into pieces in order to do that). There are more nearly correct algorithms than this, but they are much slower.

A.7   Set the cmax parameter: lightwf

Calling Sequence ...

```
lightwf (cmax)
```

Description Sets the cmax parameter interactively, assuming the current 3D display list contains the result of a previous plwf call. This changes the color of the brightest surface in the picture. The darkest surface color can be controlled using the ambient keyword to light3 (see "Lighting:" on page 61).

A.8   Return a Wire Frame Specification: xyz_wf

Calling Sequence ...

```
xyz_wf (z, [y, x] [, scale = 1.0])
```

Returns a 3-by-ni-by-nj array whose 0 th entry is x, 1 th entry is y, and 2 th entry is z. z is ni-by-nj. x and y, if present, must be the same shape. If not present, integer ranges will be used to create an equally spaced coordinate grid in x and y. The function which scales the "topography" of z(x, y) is potentially useful apart from plwf.

For example, the xyz array used by plwf can be converted from a quadrilateral mesh plotted using plf to a polygon list plotted using plfp like this:

Description xyz= xyz_wf(z, y, x, scale= scale)
```
ni= z. shape[ 1]
nj= z. shape[ 2]
list = ravel (add. outer (
ravel(add. outer (adders, zeros(nj-1, Int))) +
arange((ni-1)*(nj-1), typecode = Int),
array ([[ 0, 1], [nj + 1, nj]])))
xyz= array([ take(ravel(xyz[ 0]), list),
take(ravel(xyz[ 1]), list),
take(ravel(xyz[ 2]), list)])
nxyz= ones((ni-1)*(nj-1)) * 4;
```

The resulting array xyz is 3-by-(4*(nj-1)*(ni-1)). xyz[0: 3,4* i: 4*(i+ 1)] are the clockwise coordinates of the vertices of cell number i.

A.9 Calculate Chunks of Mesh: iterator3

Calling Sequences ...

```
iterator3 (m3)
iterator3 (m3, chunk, clist)
iterator3_rect (m3)
iterator3_rect (m3, chunk, clist)
iterator3_irreg (m3)
iterator3_irreg (m3, chunk, clist)
```

Description The iterator3 functions combine three distinct operations: 1. If only the M3 argument is given, return the initial chunk of the mesh. The chunk will be no more than `chunk3_limit` cells of the mesh.

2. If only M3 and CHUNK are given, return the next CHUNK, or None if there are no more chunks.

3. If M3, CHUNK, and CLIST are all specified, return the absolute cell index list corresponding to the index list CLIST of the cells in the CHUNK. Do not increment the chunk in this case.

The form of the CHUNK argument and return value for cases (1) and (2) is not specified, but it must be recognized by the xyz3 and getv3 functions (see "Return Vertex Coordinates for a Chunk: xyz3" on page 94 and "Get Vertex Values of Function: getv3" on page 91) which go along with this iterator3. (For case (3), CLIST and the return value are both ordinary index lists.) In the irregular case, it is guaranteed that the returned chunk consists of only one type of cell (tetrahedra, hexahedra, pyramids, or prisms).

A.10 Get Vertex Values of Function: getv3

Calling Sequence ...

```
getv3(i, m3, chunk)
getv3_rect(i, m3, chunk)
getv3_irreg (i, m3, chunk)
```

Description getv3 returns vertex values of the I th function attached to 3D mesh M3 for cells in the specified CHUNK. The CHUNK may be a list of cell indices, in which case getv3 returns a 2x2x2x(CHUNK. shape) list of vertex coordinates. CHUNK may also be a mesh-specific data structure used in the slice3 routine (see "slice3: Plane and Isosurface Slices of a 3D

mesh" on page 70), in which case getv3 may return a (ni) x(nj) x(nk) array of vertex values. For meshes which are logically rectangular or consist of several rectangular patches, this is up to 8 times less data, with a concomitant performance advantage. Use getv3 when writing slicing functions for slice3.

`getv3_rect` does the job for a regular rectangular mesh. `getv3_irreg`, for an irregular mesh, returns a 3-list whose elements are:

1. the function values for the I th function on the vertices of the given CHUNK. (The function values must have the same dimension as the coordinates; there is no attempt to convert zone-centered values to vertex-centered values.)

2. an array of relative cell numbers within the list of cells of this type.

3. a number that can be added to these relative numbers to give the absolute cell numbers for correct access to their coordinates and function values

A.11   Get Cell Values of Function: getc3

Calling Sequence ...

```
getc3(i, m3, chunk)
getc3(i, m3, clist , l, u, fsl, fsu, cells)
```

Description Returns cell values of the I th function attached to 3D mesh M3 for cells in the specified CHUNK. The CHUNK may be a list of cell indices, in which case getc3 returns a (CHUNK. shape) array of vertex coordinates. CHUNK may also be a mesh-specific data structure used in the slice3 routine (see "slice3: Plane and Isosurface Slices of a 3D mesh" on page 70), in which case getc3 may return a (ni) x(nj) x(nk) array of vertex values. There is no savings in the amount of data for such a CHUNK, but the gather operation is cheaper than a general list of cell indices. Use getc3 when writing coloring functions for slice3.

If CHUNK is a CLIST, the additional arguments L, U, FSL, and FSU are vertex index lists which override the CLIST if the I th attached function is defined on mesh vertices. L and U are index arrays into the (CLIST. shape) x2x2x2 vertex value array, say vva, and FSL and FSU are corresponding interpolation coefficients; the zone centered value is computed as a weighted average of involving these coefficients. The CELLS argument is required by histogram to do the averaging. See the source code for details. By default, this conversion (if necessary) is done by averaging the eight vertex-centered values.

`getc3_rect` does the job for a regular rectangular mesh. `getc3_irreg`: Same thing as `getc3_rect`, i. e., returns the same type of data structure, but from an irregular mesh. m3[1] is a 2-list; m3[1][0] is an array whose i th element is an array of coordinate indices for the i th cell, or a list of up to four such arrays. m3 [1] [1] is the 3 by nverts array of

coordinates. m3 [2] is a list of arrays of vertex-centered or cell-centered data. chunk may be a list, in which case chunk [0] is a 2-sequence representing a range of cell indices; or it may be a one-dimensional array, in which case it is a nonconsecutive set of cell indices. It is guaranteed that all cells indexed by the chunk are the same type.

A.12   Controlling Points Close to the Slicing Plane: ₋slice2₋precision

Calling Sequences ...

```
precision = get_slice2_precision ()
set_slice2_precision (precision)
```

Description Internal variable `_slice2_precision` controls how slice2 (or slice2x) handles points very close to the slicing plane or surface. PRECISION should be a positive number or zero. Zero PRECISION means to clip exactly to the plane, with points exactly on the plane acting as if they were slightly on the side the normal points toward. Positive PRECISION means that edges are clipped to parallel planes a distance PRECISION on either side of the given plane. (Polygons lying entirely between these planes are completely discarded.)

Default value is 0.0.

A.13   Scale variables to a palette: bytscl, split₋bytscl

Calling Sequence ...

```
bytscl(z, top= max_byte, cmin= lower_cutoff,
cmax= upper_cutoff)
split_bytscl (x, upper, cmin = None, cmax = None)
```

Description bytscl returns an unsigned char array (Python typecode `"b"`) of the same shape as Z, with values linearly scaled to the range 0 to one less than the current palette size. If `MAX_BYTE` is specified, then the scaled values will run from 0 to `MAX_BYTE` instead. If `LOWER_CUTOFF` and/ or `UPPER_CUTOFF` are specified, Z values outside this range are mapped to the cutoff value; otherwise the linear scaling maps the extreme values of Z to 0 and `MAX_BYTE`.

`split_bytscl` is as the bytscl function, but scales to the lower half of a split palette (0-99, normally the color scale) if the second parameter is zero or nil, or the upper half (100-199, normally the gray scale) if the second parameter is non-zero.

A.14   Return Vertex Coordinates for a Chunk: xyz3

Calling Sequence ...

```
xyz3 (m3, chunk)
```

Description Return vertex coordinates for CHUNK of 3D mesh M3. The CHUNK may be a list
of cell indices, in which case xyz3 returns a (CHUNK. shape)) x3x2x2x2 list of vertex co-
ordinates. CHUNK may also be a mesh-specific data structure used in the slice3 routine
(see "slice3: Plane and Isosurface Slices of a 3D mesh" on page 70), in which case xyz3
may return a 3x(ni) x(nj) x(nk) array of vertex coordinates. For meshes which are logically
rectangular or consist of several rectangular patches, this is up to 8 times less data, with a
concomitant performance advantage. Use xyz3 when writing slicing functions or coloring
functions for slice3.

A.15   Find Corner Indices of List of Cells: to_corners3

Calling Sequence ...

```
to_corners3(list, nj, nk)
```

Description Convert an array of cell indices in an (ni-1)-by-(NJ-1)-by-(NK-1) logically rectangular
grid of cells into the array of len(LIST)-by-2-by-2-by-2 cell corner indices in the correspond-
ing ni-by-NJ-by-NK array of vertices. The algorithm used is described in section "More
slice3 details". Note that this computation in Yorick gives an absolute offset for each cell
quantity in the grid. In Yorick it is legal to index a multidimensional array with an absolute
offset. In Python it is not. However, an array can be flattened if necessary.

Other changes from Yorick were necessitated by row-major order and 0-origin indices, and
of course the lack of Yorick array facilities.

A.16   Timing: timer_, timer_print

Calling Sequences ...

```
timer_ (elapsed)
timer_ (elapsed, split)
timer_print (label1, split1 [, label2, split2, ...])
```

Description timer_ returns a triple consisting of the times [cpu, system, wall]. If argument split is present, a sequence is returned whose first element is [cpu, system, wall] and whose second element is the sum of split and the difference between the new and old values of 'elapsed. ' `timer_print` prints out a timing summary for splits accumulated by timer_.

B Maintenance: Things You Really Didn't Want to Know

In this chapter we discuss in even more gory detail how the PyGist graphics are put together.

B.1 The Workhorse: gistCmodule

The reader should be familiar with many of the functions in gistCmodule from the chapter "Two-Dimensional Plotting Functions" on page 33. Most of the gistCmodule functions discussed there, and many of the helper functions, are pretty close to literal translations of the equivalent functions in Gist, the main difference being the superstructure built on top of them in order to handle PyObjects. In addition to the plotting functions, a number of functions in gistCmodule are essential for maintenance of that module, and are discussed here. We also discuss briefly a few other functions which are not literal translations of Gist functions.

B.1.1 Memory Maintenance: PyObjects

One of the primary challenges facing the developer of Python extensions is correct management of the reference counting for Python objects. Memory leaks will result if the programmer fails to decrement the reference count in temporary objects. On the other hand, decrementing the reference count too early can cause an object to go away that is referred to later, which can cause a segmentation fault when it is referenced. We have semi-automated the process in gistCmodule by maintaining a list of all PyObjects created in the process of running one of the module's functions. This list is declared as follows:

```
#define ARRAY_LIST_SIZE 30
static PyObject * PyArrayList [ARRAY_LIST_SIZE];
static int array_list_length = 0;
```

There is a suite of functions for manipulating PyArrayList.

Function Prototypes

```
static int addToArrayList (PyObject * obj)
static void clearArrayList ()
```

```
static void removeFromArrayList (PyObject * obj)
static void takeOffArrayList (PyObject * obj)
```

addToArrayList places obj on `PyArrayList` and returns 1 if successful. If obj is NULL or the list is full, returns 0. `clearArrayList` DECREF's everything on the list, and sets `array_list_lengt` to 0. This needs to be done prior to any error return. `removeFromArrayList` DECREF's obj (if it is on the list), removes it from the list, and compresses the list. `takeOffArrayList` removes obj from the list and compresses the list, but does not DECREF obj. This is done, for example, when obj is to be returned to the caller.

addToArrayList occurs throughout gistCmodule primarily in macros which create arrays. All of these macros use the TRY macro, which is defined as follows:

```
#define TRY( e, m) do{ if(!( e)){ clearArrayList(); \
clearFreeList( 0); clearMemList(); return m;}} while( 0)
```

The idea behind TRY is that generally Python functions return 0 or NULL if an error occurred. In this case it is necessary to get rid of all temporary objects and memory which was allocated up to this point. clearArrayList was discussed above, clearFreeList and clearMemList are discussed later in the chapter.

The array creation macros are as follows:

```
#define GET_ARR( ap, op, type, dim, cast) \
TRY( addToArrayList(( PyObject *)( ap=( PyArrayObject *)\
PyArray_ContiguousFromObject( op, type, dim, dim))), \
(cast) PyErr_NoMemory ())
```

This macro is the usual protocol for creating a contiguous array from a PyObject which has been sent as an argument to a function.

```
#define NEW_ARR( ap, n, dims, type, cast) \
TRY( addToArrayList(( PyObject *)( ap=\
(PyArrayObject *) PyArray_FromDims( n, dims, type))), \
(cast) PyErr_NoMemory ())
```

This macro is used usually when creating an array whose dimensions are known and which is to be filled with computed data.

```
#define RET_ARR( op, ndim, dim, type, data, cast)\
TRY( addToArrayList( op=\
PyArray_FromDimsAndData( ndim, dim, type, data)), \
(cast) PyErr_NoMemory ())
```

This final macro is used when we have a block of data and we wish to create an array containing this data, usually as a return value from a function. In order to keep this object from being permanent, use the following macro:

```
#define SET_OWN( op) \
( (PyArrayObject *) op)-> flags |= OWN_DATA
```

This macro sets a flag in the PyObject which tells Python that it can be DECREF'ed.

B.1.2   Memory Management: ArrayObjects

Definition ArrayObjects are defined as follows:

```
typedef struct arrayobject {
void * data ;
int size ;
char typecode ;
} ArrayObject;
```

These objects are used primarily in the slice2 routines to store temporary results during the calculation. The final results are passed back in PyArrayObjects created by RET_ARR. Two lists of ArrayObjects are maintained by the slice2 suite: list 0 by slice2 itself, and list 1, which is used by _slice2_part, which is called by slice2. These lists are declared as follows:

```
#define MAX_NO_LISTS 2
#define MAX_LIST_SIZE 30

static ArrayObject * freeList [MAX_NO_LISTS] [MAX_LIST_SIZE];
static int freeListLen [MAX_NO_LISTS] = {0, 0};
```

Function Prototypes ...

```
static ArrayObject * allocateArray (int size, char tc,
int nlist)
static ArrayObject * copyArray (ArrayObject * a)
static ArrayObject * arrayFromPointer (int size, char tc,
void * data, int nlist)
static void freeArray (ArrayObject * a, int n)
static void clearFreeList (int n)
static int addToFreeList (ArrayObject * x, int n)
static void removeArrayOnly (ArrayObject * x, int n)
static void removeFromFreeList (ArrayObject * x, int n)
```

Description allocateArray allocates an appropriate amount of space for size items of type tc. It then creates an ArrayObject containing this data and puts it on freeList [nlist]. copyArray makes and returns a copy of a. It does not add a to any freeList. arrayFromPointer creates an arrayObject whose data pointer points to data; it is assumed that the caller has supplied correct size and tc arguments. The resulting object is placed on freeList [nlist]. freeArray frees a's data and then a itself, and removes it from freeList [n] if it is there. clearFreeList frees everything on freeList [n] and sets the list length to 0. addToFreeList adds x to freeList [n], if it can. removeArrayOnly removes the array from freeList [n], then frees x without freeing its data. This would most likely be done when RET_ARR creates a PyArrayObject which points to x's data. removeFromFreeList frees x's data, then x itself, and removes x from freeList [n].

B.1.3 Memory Management: naked memory

Occasionally in gistCmodule it is necessary to malloc a block of memory which is not contained inside some type of object.

Definition MemList is used to keep track of such memory:

```
#define MEM_LIST_SIZE 15

static void * PyMemList [MEM_LIST_SIZE];
static int mem_list_length = 0;
```

MemList is maintained by its own suite of functions.

Function Prototypes `static int addToMemList (void * addr)`
`static void clearMemList ()`

Description The first function adds an address to MemList; the second frees everything on Mem-List and sets its length back to 0.

B.1.4   Computing contour curves: contour

Calling Sequence ...

```
# Set mesh first
plmesh (y, x, ireg, triangle = triangle)
[nc, yc, xc] = contour (level, z)
```

Description The calling sequence given above emphasizes that mesh parameters should be set by
a call to plmesh prior to calling contour. plmesh arguments are explained in section "plmesh:
Set Default Mesh" on page 34. If level is a scalar floating point number, then the the returned
values are the points at that contour level. All such points lie on edges of the mesh. If a
contour curve closes, the final point is the same as the initial point (i. e., that point is included
twice in the returned list). If level is a sequence of two reals, then contour returns the points
of a set of polygons which outline the regions between the two contour levels. The returned
values are in the form required for arguments of plfp (see Section "plfp: Plot a List of Filled
Polygons" on page 42). These will include points on the mesh boundary which lie between
the levels, in addition to the edge points for both levels. The polygons are closed, simply
connected, and will not contain more than about 4000 points (larger polygons are split into
pieces with a few points repeated where the pieces join). The 2D filled contour plot routine
plfc (see Section "plfc: Plot filled contours" on page 41) operates by calling contour with pairs
of adjacent contour levels, and then calling plfp with the output and a single color, inside a
loop. contour needed to be programmed in C because it can be called many times to do a
single filled contour plot, and the calculations take too long to be performed in interpreted
code. contour calls lower level Gist routines that do most of the work.

B.1.5   Computing slices: slice2, slice2x, ␣slice2␣part

The 3D graphics in Gist itself is still experimental, and virtually all the computational functions are
written in Yorick, an interpreted language. Many of the PyGist 3D computations were translated
into Python from Yorick, originally including the slice2 and slice2x functions, and their auxiliary,
`_slice2_part`. When we implemented contours and filled contours on surfaces (which is cur-
rently not implemented in Gist itself), we used slice2 and slice2x to compute the contours and
the polygon lists enclosed within them. These computations were much too slow, so we rewrote
slice2 in C (slice2x remains in Python; it just calls slice2 with a parameter set) and put them into
the gistCmodule. The user interface to these functions has been discussed in a previous section
"slice2 and slice2x: Slicing Surfaces with planes", but we discuss them here from the viewpoint of
implementation. We also discuss the "hidden" function `_slice2_part` here for the first time.

Calling Sequences ...

```
[nverts, xyzverts, values] = slice2 (plane, nv,
xyzv, vals = None, _slice2x = 0)
[nverts, xyzverts, values, nvertb, xyzvertb, valueb] =
slice2x (plane, nv, xyzv, vals)
static int _slice2_part (ArrayObject * xyzc,
ArrayObject * keep, ArrayObject * next, ArrayObject * dp,
ArrayObject * prev, ArrayObject * last,
ArrayObject * valc, ArrayObject ** xyzc_new,
ArrayObject ** nvertc, ArrayObject ** valc_new,
int freexyzc, int freevalc)
```

Description The argument plane can be either a scalar or a plane3 (see "Creating a Plane" on page 65); nv is an array of integers, the i th entry of which gives the number of vertices of the i th polygonal cell; xyzv are the vertices of the coordinates of the cells, with each consecutive nv[i] entries representing the

vertices of the i-th cell; and vals being a set of values, one for each cell. These arguments are the same format as returned by slice3 and slice3mesh.

If plane is a plane3, then vals (if not None) is a cell-centered set of values expressing the color of each cell, and the outputs nverts, xyzverts, and values represent the polygons and their colors (if any) describing the portion of the sliced surface that is on the positive side of the plane. That's all you get with slice2. With slice2x, you get in addition nvertb, xyzvertb, and valueb, which describe the part of the surface on the negative side of the slicing plane. Warning: one of these specifications could be None, None, None if the entire surface lies on one side of the plane. If plane is a scalar value, then vals must be present and must be node-centered. In this case, the outputs nverts, xyzverts, and values represent the polygons and their colors (if any) describing the portion of the sliced surface where vals on the vertices are greater than or equal to the scalar value plane. (This actually allows you to form an arbitrary two-dimensional slice of a surface.) With slice2x, you get in addition nvertb, xyzvertb, and valueb, which describe the part of the surface where vals on the vertices are less than the scalar value plane.

The optional parameter _slice2x, if 1, tells slice2 to return slices on both sides of the slicing surface or plane; if not present, or 0, then the slice on "top" is returned. slice2 works by deciding which polygons lie entirely "above" the slicing surface, which ones lie entirely "below" the slicing surface, and which ones are cut by the surface. If _slice2x is 0, then the ones "below" the surface are discarded. slice2 then calls _slice2_part with the polygons to be cut by the plane; once to get the cut polygons "above" the surface, then, if _slice2x is 1, a second time to get the cut polygons "below" the surface. The list of uncut and cut polygons "above" the surface is concatenated and returned (_slice2x == 0); the list of uncut and cut polygons "below" the surface is concatenated and returned also if _slice2x is 1.

In the case of a plane slice, suppose that the equation of the slicing plane is

```
ax + by + cz = d
```

Then a point `(x1,y1,z1)` is considered to be on the positive side of the plane if

```
ax 1 + by 1 + cz 1 -d >= _slice2_precision
```

and on the negative side if

```
ax 1 + by 1 + cz 1 -d < _slice2_precision
```

For a discussion of `_slice2_precision`, and how to get and set its value, see Section "Controlling Points Close to the Slicing Plane: ˍslice2ˍprecision" on page 93.

In the case of a slicing surface, vertex i is considered to be above the surface if

```
vals [i] -plane >= _slice2_precision
```

and below it if

```
vals [i] -plane < _slice2_precision
```

For all intents and purposes, the user may assume that `_slice2_precision` is 0.0, as this is the default. However, we allow you to change this if you think you have good reason.

There is a conceptual difficulty for the case of a quad face all four of whose edges are cut by the slicing plane or surface. This can only happen when two opposite corners are above and the other two below the slicing plane. There are three possible ways to connect the four intersection points in two pairs: (1) // (2)
and (3) X. There is a severe problem with (1) and (2) in that a consistent decision must be made when connecting the points on the two cells which share the face—that is, each face must carry information on which way it is triangulated. For a regular 3D mesh, it is relatively easy to come up with a consistent scheme for triangulating faces, but for a general unstructured mesh, each face itself must carry this information. This presents a huge challenge for data flow, which we don't believe is worthwhile, because the X choice is unique, and we don't see why we shouldn't use it here. For contouring routines, we reject the X choice on aesthetic grounds, and perhaps that will prove to be the case here as well but we believe we should try the simple way out first. In this case, we are going to be filling these polygons with a color representing a function value in the cell. Since the adjacent cells should have nearly the same values, the X-traced polygons will have nearly the same color, and we doubt there will be an aesthetic problem. Anyway, our implementation of slice3, slice2, and `_slice2_part` produces the unique X (bowtied) polygons, rather than attempting to choose between // or (non-bowtied) alternatives. Besides, in the case of contours, the trivial alternating triangulation scheme is just as bad aesthetically as every zone triangulated the same way!

B.2 Some Yorick-like Functions: yorick.py

The module `yorick.py` contains a few functions similar to ones in Yorick, which perform array manipulations necessary in doing 3D graphics.

We shall depart from our usual format here, and just give the calling sequences followed by a short explanation for each of the functions.

```
zcen_(x, i = 0)
```

Returns an array whose i th dimension is one smaller than the i th dimension of x, with the elements along the i th being the averages of two adjacent elements in the original x. i cannot be larger than 5.

```
dif_(x, i = 0)
```

Returns an array whose i th dimension is one smaller than the i th dimension of x, with the elements along the i th being the differences of two adjacent elements in the original x. i cannot be larger than 5.

```
maxelt_(* x)
```

`maxelt_` accepts a sequence of one or more possible multi-dimensional numerical objects and computes their maximum. In principle these can be of arbitrary complexity, since the routine recurses.

```
minelt_(* x)
```

`minelt_` accepts a sequence of one or more possible multi-dimensional numerical objects and computes their minimum. In principle these can be of arbitrary complexity, since the routine recurses.

```
rem_0_(z)
```

`rem_0_(z)` returns a copy of array z after having replaced any zero elements with 1.e-35. Assumes z has one or two dimensions.

```
avg_(z)
```

avg_(z) returns the average of all elements of its array argument.

```
sign_(x)
```

Returns 1 if $x >= 0$, -1 otherwise.

```
timer_(elapsed, *split)
timer_print (label, split, *other_args)
```

see Section "Timing: timer_, timer_print" on page 94.

B.3   Additional Array Operations: arrayfnsmodule

Obsolete

A number of functions which emulate Yorick functions are used frequently by the 3D graphics, and in interpreted code simply run too slowly. These functions have been moved to arrayfnsmodule and written in C. Their descriptions form this section of the manual.

B.3.1   Counting Occurrences of a Value: histogram

Calling Sequence ...

```
histogram (list [, weight])
```

Description histogram accepts one or two arguments. The first is an array of non-negative integers and the second, if present, is an array of weights, which must be promotable to double. Call these arguments list and weight. Both must be one-dimensional with len (weight) ¿= max (list) + 1. If weight is not present:

histogram (list) [i] is the number of occurrences of i in list. If weight is present: histogram (list, weight) [i] is the sum of all weight [j] where list [j] == i.

B.3.2 Assigning to an Arbitrary Subset of an Array: array_set

Calling Sequence ...

```
array_set (vals1, indices, vals2)
```

Description `array_set` accepts three arguments. The first is an array of numerics (Python characters, integers, or floats), and the third is of the same type. The second is an array of integers which are valid subscripts into the first. The third array must be at least long enough to supply all the elements called for by the subscript array. (It can also be a scalar, in which case its value will be broadcast.) The result is that elements of the third array are assigned in order to elements of the first whose subscripts are elements of the second.

```
arr_array_set (vals1, indices, vals2)
```

is equivalent to the Yorick assignment

```
vals1 (indices) = vals2
```

We have generalized this so that the source and target arrays may be two dimensional; the second dimensions must match. Then the array of subscripts is assumed to apply to the first subscript only of the target. The target had better be contiguous.

B.3.3 Sorting an array: index_sort

Calling Sequence ...

```
index_sort (x)
```

Description `index_sort` accepts a one-dimensional array x of some numerical type and returns an integer array of the same length whose entries are the subscripts of the elements of the original array arranged in increasing order. We chose to use heap sort because its worst behavior is n* log(n), unlike quicksort, whose worst behavior is n** 2.

B.3.4 Interpolating Values: interp

Calling Sequence ...

```
interp (x, y, z)
```

Description `interp (y, x, z)` treats `(x, y)` as a piecewise linear function whose value is `y [0]` for `x < x [0]` and `y [len (y)-1]` for `x > x [len (y)-1]`. An array of floats the same length as `z` is returned, whose values are ordinates for the corresponding `z` abscissae interpolated into the piecewise linear function.

B.3.5   Digitizing an array: digitize

Calling Sequence ...

```
digitize (x, bins)
```

Description `bins` is a one-dimensional array of integers which is either monotonically increasing or monotonically decreasing. `digitize (x, bins)` returns an array of python integers the same length as `x` (if `x` is a one-dimensional array), or just an integer (if `x` is a scalar). The values i returned are such that `bins [i-1] <= x < bins [i]` if `bins` is monotonically increasing, or `bins [i-1] > x >= bins [i]` if `bins` is monotonically decreasing. Beyond the bounds of `bins`, returns either `i = 0` or `i = len (bins)` as appropriate.

B.3.6   Reversing a Two-Dimensional array: reverse

Calling Sequence ...

```
reverse (x, n)
```

Description reverse (x, n) returns a PyFloat matrix the same size and shape as x, but with the elements along the n th dimension reversed. x must be two-dimensional.

B.3.7   Obtaining an Equally-Spaced Array of Floats: span

Calling Sequence ...

```
span (lo, hi, num, d2 = 0)
```

Description span (lo, hi, num, d2 = 0) returns an array of num equally spaced PyFloats starting with lo and ending with hi. if d2 is not zero, it will return a two-dimensional array, each of the d2 rows of which is the array of equally spaced numbers.

B.3.8   Effective Length of an Array: nz

Calling Sequence ...

```
nz(x)
```

Description nz (x): x is an array of unsigned bytes (Python typecode `"b"`). If x ends with a bunch of zeros, this returns with the index of the first zero element after the last nonzero element. It returns the length of the array if its last element is nonzero. This is essentially the "effective length" of the array.

B.3.9 Finding Edges Cut by Isosurfaces: find_mask

Calling Sequence ...

```
find_mask (fs, node_edges)
```

Description This function is used to calculate a mask of integers whose corresponding entry is 1 precisely if an edge of a cell is cut by an isosurface or plane, i. e., if the function fs is one on one of the two vertices of an edge and zero on the other (fs = 1 represents where some function on the mesh was found to be negative by the calling routine). fs is ntotal by nv, where nv is the number of vertices of a cell (4 for a tetrahedron, 5 for a pyramid, 6 for a prism, 8 for a hexahedron). node_edges is a nv by ne array, where ne is the number of edges on a cell (6 for a tet, 8 for a pyramid, 9 for a prism, 12 for a hexahedron). The entries in each row are 1 precisely if the corresponding edge is incident on the vertex. The exclusive or of the rows which correspond to nonzero entries in fs contains 1 in entries corresponding to edges where fs has opposite values on the vertices. (The vertices and edges of a cell have a standard ordering which is discussed in "Standard ordering for the four types of mesh cells" on page 108.)

The mask returned by this function will be a one dimensional array ntotal * ne long. An entry [i * ne + j] in this mask will be 1 precisely if edge j of cell i is cut by the isosurface or plane.

B.3.10 Order Cut Edges of a cell: construct3

Calling Sequence ...

```
construct3 (mask, itype)
```

Description Computes how the cut edges of a particular type of cell must be ordered so that the polygon of intersection can be drawn correctly. itype = 0 for tetrahedra; 1 for pyramids; 2 for prisms; 3 for hexahedra. Suppose nv is the number of vertices of the cell type, and ne is the number of edges. mask has been ravelled so that it is flat; originally it had 2** nv-2 rows, each with ne entries. Each row is ne long, and has an entry of 1 corresponding to each edge that is cut when the set of vertices corresponding to the row index has negative values. (The binary number for the row index + 1 has a one in position i if vertex i has a negative value.) The return array permute is ne by 2** nv-2, and the columns of permute tell how the edges should be ordered to draw the polygon properly.

B.3.11 Expand cell-centered values to node-centered values: to_corners

Calling Sequence ...

```
to_corners (values, nv, sumnv)
```

Description   values is a one-dimensional array of floating point values defined on a set of polygons. nv is an integer array of the same size telling how many vertices each of the polygons has. sumnv is the sum of all the values in nv. This routine takes an array of floats describing cell-centered values and returns an array of node-centered values. It is very unsophisticated, merely creating an array of floats sumnv long, whose first nv [0] entries are all values [0], next nv [1] entries are all values [1], etc.

B.4   More slice3 details

The way slice3 works depends strongly on a standard ordering of the nodes, edges, and faces of mesh cells. In this section we shall delineate the standard ordering used. This ordering is encapsulated in various tables contained in slice3. py and arrayfnsmodule. c. The maintainer of this code must have an understanding of this order.

B.4.1   Standard ordering for the four types of mesh cells

Tetrahedra

On the illustration at the left, the vertices are numbered v0 through v4, and the edges, e0 through e5. The faces are numbered as shown in the following table:

Pyramids TABLE 3. tet face numbering face number edges on face

f0 e0, e1, e3 f1 e0, e2, e5 f2 e1, e2, e4 f3 e3, e4, e5

TABLE 4. pyr face numbering face number edges on face

f0 e0, e1, e4 f1 e1, e2, e5 f2 e2, e3, e6 f3 e0, e7, e3 f4 e4, e5, e6, e7

v0 v1 v2 v3 e0 e2 e1

e3 e4 e5

v0 v1 v4 v3

v2

e0 e1 e2 e3 e4 e5 e6 e7 prisms

hexahedra

B.4.2 Standard numbering of cells in a regular rectangular mesh

Suppose we have a regular rectangular mesh whose cell dimensions are ni-1 by nj-1 by nk-1 (and thus the vertex array is ni by nj by nk). The total number of cells is

```
ncells = (ni-1) * (nj-1) * (nk-1)
```

TABLE 5. pri face numbering face number edges

f0 e2, e7, e3, e6 f1 e0, e6, e1, e8 f2 e4, e8, e5, e7 f3 e0, e4, e2 f4 e1, e3, e5

TABLE 6. hex face numbering face number edges

f0 e0, e6, e2, e4 f1 e1, e5, e3, e7 f2 e0, e8, e1, e10 f3 e2, e11, e3, e9 f4 e4, e9, e5, e8 f5 e6, e10, e7, e11

v0 v1

v2 v3 v4

v5 e0

e1

e2 e3

e4 e5 e6 e7 e8

v0 v1 v2 v3

v4 v5 v6 v7 e0

e1 e2 e3 e4

e5

e6 e7 e8 e9

e10 e11

and the cells are numbered from 0 to ncells-1 according to the following scheme. Suppose that (i, j, k) are the maximum subscripts of the eight vertices of a cell numbered N in our scheme. Then the number of the cell with maximum vertex subscripts (i, j, k + 1) will be N + 1; the number of

the cell with maximum vertex subscripts(i, j + 1, k) will be N + nk; and the number of the cell with maximum vertex subscripts(i + 1, j, k) will be N + nj * nk. Thus each triple of subscripts (i, j, k), where none of the three is zero, uniquely determines a cell number, and cell numbers run consecutively as we increment the subscripts through their ranges (starting with 1) in row major order. Similarly, we can number the vertices from 0 through ni * nj * nk-1 by numbering them consecutively as we increment the subscripts through their ranges (starting with 0) in row major order.

This leads for the following scheme for computing the vertex numbers for all eight of the vertices of a cell, given the cell number. First, construct the scalar

```
N1 = N + N / (nk-1) + nk * (N / ( (nk-1) * (nj-1)))
Then, add this scalar to each element of the 2 x 2 x 2 array
array ( [ [ [0, 1], [nk, nk + 1]],
[ [njnk, njnk + 1], [nk + njnk, nk + njnk + 1]]])
```

The result is a 2 x 2 x 2 array of the vertex numbers of the vertices of the cell. Given that the arrays of vertex coordinates are stored in row major order, then if we ravel them (i. e., flatten them out), flatten the above array of vertex numbers, extract precisely those eight coordinates from each coordinate array, and then reshape them to 2 x 2 x 2, then we have the coordinates of the vertices of the cell under consideration.

The function `to_corners3` does this calculation for an arbitrary list of cell numbers (see "Find Corner Indices of List of Cells: to_corners3"). The function `slice3` calls `to_corners3` with a list of cells which are cut by a plane or isosurface in a rectangular mesh in order to determine the coordinates of their vertices, the final goal being to find the points at which the edges are cut by the plane or isosurface. These edge points are then connected in a systematic way using (among other things) the numbering schemes described previously, in order to yield the polygonal sections through cells made by the plane or isosurface.

B.4.3   How slice3 works

Recall the calling sequence of slice3 (see see Section "slice3: Plane and Isosurface Slices of a 3D mesh" on page 70):

```
[nverts, xyzverts, color] = \
   slice3 ( m3, fslice, nv, xyzv [, fcolor [, flg 1 ]]
   [, value = <val>] [, node = flg 2 ] )
```

The important arguments are m3, a mesh specification which was returned by an earlier call of mesh3 (see "Creating a mesh3 argument"); fslice (which specifies either the name of a slicing function, a slicing plane in plane3 format (see Section "Creating a Plane" on page 65), or the number of the function defined on the mesh with respect to which an isosurface is to be computed); fcolor, which (if None) specifies that the section is to be shaded, or (if a function) gives a set of values on the the cells specified to it when slice3 calls it; value, which in the case of an isosurface specifies the value of the function doing the slicing; and node, which if nonzero and color is calculated, says to return node-centered rather than cell-centered values.

One of the first things that slice3 does is to call iterator3 with m3 as argument, which in turn calls the appropriate iterator for the particular type of mesh. (Recall that m3 contains names of appropriate functions to call for this mesh.) The purpose of iterator3 is to "chunk" up the mesh into manageable pieces; the main loop in slice3 calls iterator3 repeatedly until it finally returns None, signalling that the entire mesh has been processed. The details of both types of iterator are straightforward and can be had by inspecting the source code. One thing to bear in mind is that in the case of an unstructured mesh, iterator3 is guaranteed to return a chunk which consists of only one type of cell.

Why "chunk" up the mesh? The creators of the Yorick version of slice3, Langer and Munro, did so in order to avoid the possibility of creating very large temporaries and thus, perhaps, having memory problems. It seemed to us judicious to do the same thing.

The first thing done inside the slice3 main loop is to call the appropriate slicing function. Two functions are supplied in slice3. py. Their calling sequences and descriptions are as follows:

```
_isosurface_slicer (m3, chunk, iso_index, _value)
```

an isosurface slicer brings back a list [vals, None] where vals is simply an array of the values of the `iso_index`-th mesh function on the vertices of the specified chunk, or (in the unstructured case) a triple, consisting of the array of values, an array of relative cell numbers in the chunk, and an offset to add to the preceding to get absolute cell numbers.

```
_plane_slicer (m3, chunk, normal, projection)
```

In the case of a plane slice, this returns a list [vals,_xyz3] (or [[vals, clist, cell_offset], _ in the irregular case) where _xyz3 is the array of vertices of the chunk. _xyz3 is ncells by 3 by something (in the irregular case), ncells by 3 by 2 by 2 by 2 in the regular case, and 3 by ni by nj by nk otherwise. vals will be the values of the projections of the corresponding vertex on the normal to the plane, positive if in front, and negative if in back.

In addition, the user may supply a slicing function; if so, its calling sequence must be of the form

```
fslice (m3, chunk)
```

and it must return something resembling the returned values above. If the m3 mesh is totally un-structured, the chunk should be arranged so that fslice returns an ncells-by-2-by-2-by-2 [hex case] (or ncells-by-3-by-2 [prism] or ncells-by-5 [pyramid] or ncells-by-4 [tet]) array of vertex values of the slicing function. Note that a chunk of an irregular mesh always consists of just one kind of cell. On the other hand, if the mesh vertices are arranged in a rectangular grid (or a few patches of rectangular grids), the chunk should be the far less redundant rectangular patch. Determination of the Critical Cells The critical cells are those cells (if any) which are cut by the slicing plane or isosurface. There are precisely those cells on the vertices of which the vals returned by the slicing function changes sign. For cells of one of the four types present in an unstructured mesh, one adds up the number of vertices on which vals is negative. If this is a positive number and also less than the number of vertices for that cell type, then the cell is critical. In the structured case, vals is ni by nj by nk. To the array which is 1 where vals is negative and 0 elsewhere, we apply the zcen_ (see page 103) function along each of its three dimensions. The result is an array ni-1 by nj-1 by nk-1 of values defined on each cell which can be one of the nine values 0., .125, .25, .375, .5, .625, .75, .875, 1.0. Those cells where the value is strictly between the two end values are critical. Thus we form clist, which is an array of absolute cell numbers of the critical cells.

If clist is not empty, then we extract the coordinates of the critical cells, the data values at these points, and (if appropriate) the colors of the cells. In the case of a structured mesh, we use the to_corners3 function discussed earlier (see page 94) to convert cell numbers to node numbers, in order to get the node coordinates and data. We append a list of this to our list of results (appending [None, None, None, None] if clist is empty) and then continue iterating.

Determination of the Cut Edges and the Intersection Points Once this loop completes, there is another for loop which loops through each type of cell (structured meshes are lumped under hex cells) present in the mesh, putting together the "chunks" of results if necessary. It then calls find_mask (page 107), which returns a mask array ncells * ne long (ncells is the total number of cells of this type, ne the number of edges on a cell) which contains 1's corresponding to edges which are cut by the plane or isosurface (in the standard ordering discussed earlier in this chapter; see page 108). It is now easy to get the coordinates of the endpoints of the cut edges using the standard numbering embodied in the tables; we then linearly interpolate along the cut edges, based on the values on their endpoints, to obtain a list of coordinates of the intersections of the plane or isosurface with the cells. This list of points now needs to be ordered so that the polygons of intersection can be drawn properly.

Ordering of the Intersection Points We associate with each critical cell a pattern number between 0 and 255 (non-inclusive) which denotes in one number the pattern of its vertices where the function value is negative. The pattern

number is arrived at by assigning the number 2 k to the k th vertex in the cell, then adding together for each cell the numbers assigned to its vertices that have negative values. We now create

a new pattern array which is ncells * ne long and in which the entry corresponding to each cut edge contains the same pattern number as its adjacent cell; i. e., if cell j has cut edge i and pattern number n, then pattern [i * ne + j] will contain n.

The `_poly_permutations` array. To each pattern, there corresponds a permutation of the edges so that they occur in the order in which the edges are to be connected. Let each such permutation be stored as a list of integers from 0 to ne-1 such that sorting the integers into increasing order rearranges the edges at the corresponding indices into the correct order. (The position of unsliced edges in the list is arbitrary as long as the sliced edges are in the proper order relative to each other.) Let these permutations be stored in a ne-by-254 array `_poly_permutations`.

`_poly_permutations` is computed (one time only) as follows. When slice3. py is imported, `_construct3` is called four times, once for each type of cell. `_construct3` first creates a

mask array below dimensioned (2 nv-2) by nv. The row below [k] has an entry for each vertex, marked 0 or 1 corresponding to the pattern number k + 1. `_construct3` now calls `find_mask` (see "Finding Edges Cut by Isosurfaces: find_mask") with parameters below and the `_node_edges` array for that particular type of cell. `find_mask` returns an array (called mask)

(2 nv-1) by ne in size; each set of ne consecutive entries is filled with an edge mask, i. e., the entry corresponding to an edge in the standard order is 1 or 0 according as the corresponding edge is cut or not. `_construct3` now calls construct3, a function in arrayfnsmodule (see section "Order Cut Edges of a cell: construct3"), with mask and the cell type as parameters.

The purpose of construct3 is to determine an order for the cut edges so that the polygons representing the plane or isosurface cut of the cell will be drawn properly. construct3 does this by calling an auxiliary function walk3 inside a loop, each call of walk3 being with the next ne entries of mask. walk3 not only decides the correct order of the points of intersection in order to draw the polygons, but also decides whether there are disjoint polygonal intersections with this cell. The walk3 algorithm begins with the lowest numbered cut edge (and marks that edge as having been used) and examines the lowest numbered face incident upon this edge. There must be at least one other cut edge on this face. If the face is triangular, it looks first at the next edge counterclockwise (in the outwards normal direction), then (if necessary) the next one clockwise. On a square face it looks first at the opposite edge, then at the next one clockwise, then counterclockwise. When it has selected an edge, it goes to the other face incident upon that edge and repeats the process. If at some point no unused edge can be found, then that means a closed polygon has been found. The next unused edge with the lowest number is chosen (if there is one) and the process repeats. In the latter case, there is more than one disjoint polygonal intersection with the cell, and the number ne * (no. of disjoint polygons so far) is added to the edge permutations.

Thus, for each cut cell in the mesh, `_poly_permutations` tells the order that the cutting points must be connected, and how many polygonal intersections there are with the cell. In the

function `slice3`, the following instructions compute subscripts into the array of points in the correct order for drawing:

```
pattern = take (ravel (transpose (_poly_permutations [i])),
_no_edges [i] * (pattern-1) + edges) \
+ 4 * _no_edges [i] * cells
order = argsort (pattern)
```

The order array is now used as a set of subscripts so that we can extract the coordinates of the cutting points in the proper order. Once this has been done, the array whose entries give the number of vertices in each polygon is calculated.

There remains only the question of splitting the points in a single cell into multiple disjoint polygons. To do this, recall that when computing `_poly_permutations`, we had added ne (the number of edges on this type of cell) to any second disjoint polygon's edge list, 2 * ne to any third one, etc. The following will now give an array whose entries corresponding to the edge orderings for each cell will be 0 for the first disjoint polygon, 1 for the second, 2 for the third, and 3 for the fourth (if there are that many).

```
pattern = pattern / _no_edges [i]
```

Now pattern jumps by 4 between cells, smaller jumps within cells get the list of places where a new value begins, and form a new pattern with values that increment by 1 between each plateau. Then the following relatively straightforward computation computes the nverts array. In order to fully appreciate how the algorithm works, we have indicated the results supposing that we began with the pattern [0,0,0,1,1,1,1,2,2,2,4,4,4,4,4,5,5,5,8,8,8].

```
pattern = dif_(pattern, 0)
#[ 0,0,1,0,0,0,1,0,0,1,0,0,0,0,1,0,0,1,0,0]
nz = nonzero (pattern)
#[ 2,6,9,14,17]
list = zeros (len (nz) + 1, Int)
#[ 0,0,0,0,0,0]
list [1:] = nz + 1
#[ 0,3,7,10,15,18]
newpat = zeros (len (pattern) + 1, Int)
#[ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
newpat [0] = 1
newpat [1:] = cumsum (not_equal (pattern, 0)) + 1
```

```
#[ 1,1,1,2,2,2,2,3,3,3,4,4,4,4,4,5,5,5,6,6,6]
pattern = newpat
nverts = histogram (pattern) [1:]
#[ 3,4,3,5,3,3]
```

C Change History for this Document

April 7, 2003 plh option to add labels below x-axis. Descriptions of `get_style` and `set_style` and how to change window style interactively. (Michiel de Hoon)

March 25, 2003 pldefault(cgmfilesize=20) to reset CGM filesize to 20 MB.

February 13, 2003 Add note about GISTPATH and description of plremove.

January 17, 2003 Add index entries for keywords.

January 16, 2003 Incorporate changes from Michiel de Hoon: Windows and Cygwin installation instructions and spelling corrections. Also, the environment variables PYTHONHOME and PYTHONPATH do not need to be set any longer.

January 6, 2003 Source for the original document (in Framemaker) no longer exists. This version was resurrected by applying a tool to convert PDF to plain text and adding the formatting required for LaTeX.