UCRL-JC-124228
PREPRINT

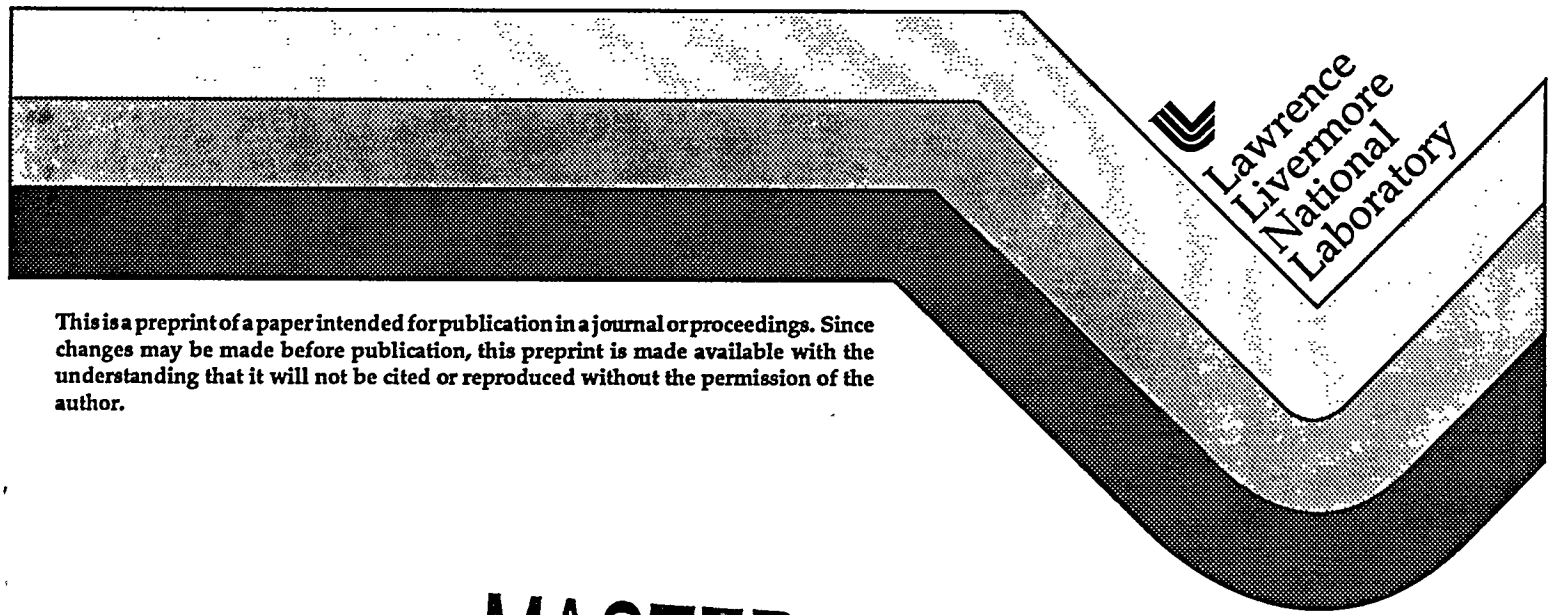
CONF-9606191--3

Gist: A Scientific Graphics Package for Python

L. E. Busby

This paper was prepared for submittal to the
4th International Python Workshop
Livermore, CA
June 3-6, 1996

May 8, 1996



Gist: A Scientific Graphics Package for Python

Lee E. Busby

Abstract

"Gist" is a scientific graphics library written by David H. Munro of Lawrence Livermore National Laboratory (LLNL). 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 versus Y plots with "good" tick marks and tick labels, 2-dimensional quadrilateral mesh plots with contours, vector fields, or pseudo color maps on such meshes, with 3-dimensional plots on the way.

The Python Gist module utilizes the new "Numeric" module due to J. Hugunin and others. It is therefore fast and able to handle large datasets. The Gist module includes an X Windows event dispatcher which can be dynamically added (e.g., via importing a dynamically loaded module) to the Python interpreter after a simple two-line modification to the Python core. This makes fast mouse-controlled zoom, pan, and other graphic operations available to the researcher while maintaining the usual Python command-line interface.

Munro's Gist library is already freely available. The Python Gist module is currently under review and is also expected to qualify for unlimited release.

Background

The Gist graphics library was written by David Munro of LLNL, a plasma physicist frustrated by the low quality of tools available in the UNIX workstation environment. In this respect, Gist satisfies a general rule of software design attributed to Eric Allman: Write programs that *you* want to use. (*A Quarter Century of UNIX*, Peter H. Salus, 1994, pp. 145.) Munro has invested several years in the design and implementation of Gist and *Yorick*, an extensible interpreter specialized for scientific applications. It is not an accident that Gist is well suited for incorporation into an interpretive programming environment, and my Python interface to Gist is, for now, virtually a clone of *Yorick's*. This decision certainly reduced the effort and cost of implementation, training, and documentation on my part, but it also recognizes that Munro's interface is good enough.

Gist attempts to satisfy four important requirements of scientific graphics: First, plots are often produced in great numbers and then scanned for patterns or trends. Gist provides an efficient format (ANSI/ISO CGM) for archival storage of graphics output, and a facility to easily create *families* of

output files. With such output and the CGM browser included with Gist, it is quite practical to produce, store, sift through, and select from files containing dozens or hundreds of plots.

Second, Gist is oriented towards portability. It is written in Standard C, and has demonstrated implementations on every major UNIX variant. It produces excellent quality color PostScript either directly or by conversion from the CGM format. Its X Window code is written directly to Xlib, so it is independent of all X Toolkits, window managers, or other support programs. Internally, the code drives a virtual output engine with a well-defined interface, so that new engines, say for *OpenGL*, can be added with relative ease.

Third, Gist provides a superior interface for detailed interactive analysis and manipulation of plots. It provides simple mouse-controlled zoom and pan, with continuous accurate feedback on the world coordinates of the mouse cursor. The interface is unadorned, simple to use, and highly responsive.

Fourth, Gist allows for iterative construction of important plots for display purposes. (There is a saying around LLNL, only half in jest, that our most important product is *viewgraphs*.) The user may edit plots element by element, change the position of axes, add or remove text, and so on, before committing the image to hardcopy.

Interface Overview

For the C programmer, Gist may be approached at several levels. At the lowest level, it may be treated as an implementation of the Graphical Kernel Standard (GKS), with bindings to C (and Fortran at level *ma*) and output drivers for PostScript, X Windows, and CGM. Above this level, Gist provides operators to control and image more complicated structures, such as *decorated polylines* and *quad meshes*. At a still higher level, Gist provides for display list creation and management, allowing the programmer to manipulate several *Drawings* at a time, to display them on a variety of output *Engines*, to edit and fine-tune their elements, and to encapsulate a variety of parameters into *style* and color *palette* files. Python's C interface to Gist is made at this level.

For the Python programmer and user, the Gist module provides a set of functions to produce and manipulate graphics in one or more new screen windows. The command

```
import gist; gist.plg([0, 1])
```

is enough to start out. You may create up to 8 simultaneous graphic windows. Each window may have its own associated hardcopy output file, or it may reference a default file. Windows or files may be opened or closed at any time, and windows can be directed to any X display. One window at a time is *current*, selected by the *current_window* function. Meanwhile, the familiar Python command line interface continues to operate in its original window without apparent change.

Each window contains one or more coordinate systems for graphic output. The current system and the (X,Y) value of the mouse cursor are continuously displayed in the window title bar whenever the cursor is within the window boundaries. Zoom and translation operations can be accomplished by clicking and dragging with the mouse, or by using the *limits* command.

Plots are built up by one or more calls from the command line to display graphic objects, add text, adjust the position and type of markers, and so forth. The current state of the plot may be saved to the hardcopy file at any time, or several times, using the *hcp* command. To begin a new plot, use the *fma* (frame advance) command.

The Gist library selects appropriate default values for most adjustable properties so as to minimize the tinkering required. Property default values can be set by several methods. The most comprehensive method is by *style* files, which preset things such as the number and location of coordinate systems in a frame, the location of axes, type, size, and labeling of tick marks, and so on. Gist ships with several style files to choose among, and most casual users will be quite satisfied with one of them.

Gist currently provides functions to plot two major types of objects. First, the classic X versus Y graph: You may plot any 1-dimensional Python sequence type, so for example,

```
plg ("Lee Busby")
```

produces a graph of the ASCII values of the successive characters in my name. More typically,

```
x = 2 * pi * arange (100)/99.0
plg(sin(x), x)
```

produces a plot of the sin function. The *pldj* function (plot disjoint line segments) provides a useful variation. It accepts two arrays as arguments, and draws a series of (disjoint) lines whose endpoints are the successive respective elements of each array.

The second area addressed by Gist concerns discrete functions over a 2-dimensional quadrilateral mesh. Such meshes are represented internally by a pair of 2-dimensional arrays X and Y, where the (i,j) element of each array gives the X or Y coordinate respectively of that point in the mesh. The library gives a variety of ways to display such functions, including contours, vector fields, and several color or gray scale cell plot forms. The library is also capable of producing cell image plots of any 2-dimensional array (without reference to an underlying mesh). In all, there are some 40 commands, from *animate* to *zoom-factor* in the Gist module.

One other part of the interface is novel enough to mention. All the commands in the Gist module have extensive on-line documentation. Recall that my Python interface to Gist is very similar to Munro's *Yorick* Gist interface. Documentation for that interface already existed as an ASCII text file of some 1,000 lines in length, which I was highly motivated to reuse.

The solution I provided is embarrassingly simple, but it is independent of Gist and written purely in Python, so I describe it here. My solution depends upon the ability of the UNIX *more* and *grep* commands to search for a regular expression in a file. I've written a small module *help.py* with a single external method *help*, which takes a string argument. It is used as in

```
from help import help
help ("plg")
```

This initiates a (grep) search along the components of *sys.path* for the pattern /^plg:$/ in all files having the suffix ".help". The first match found is passed to *more* using the same regular expression. The help files themselves are just arbitrary ASCII text, where the "plg" topic begins with a line of the form shown in the regular expression above. The lookup goes reasonably quickly on a good workstation, and it's easy to cache the most recently accessed help file to speed future searches. Searches can also be directed to a particular help file (e.g., module) if you know the module name: help ("gist.plg") will find the "plg" topic only in the file named "gist.help".

This is frankly a stopgap. I think the future of on-line help is clear now, and it is written in HTML. Ubiquity, the ability to display multiple data types including graphics and audio, and the ability to index essentially *everything* make the Web browser an unbeatable tool for documentation and on-line help. There may still be a separate role for class browsers and other tools which analyze and present views of a dynamic system, but I contend that they often should produce HTML as output.

Event Handling

The Gist library includes an event dispatching facility which multiplexes several concurrent input sources. Event handlers can be added or removed during execution, and there is provision (unused in the Python module) for calling a worker function at times when all input sources are idle. The facility is based upon the UNIX *select* or *poll* system call.

It is not immediately obvious how to retrofit event handling into a program structured like Python. Python executes an input program line by line in three phases: First, it builds a parse tree representing the current line; second, it compiles the tree into a list of instructions for a stack-based virtual machine; and third, it executes the stack. Input characters are read from *stdin* during the parse phase, whenever the tokenizer reaches the end of its buffer. Thus (like most compilers), the *program* (not the user) decides when to read input.

Event handlers often sit at or near the top of a program's call tree. In this model, *events* are the central elements which drive all responses of the program and which therefore motivate the program's logical structure. This model would be awkward to force onto Python. For (just) one thing, it would require radically re-structuring the logic that determines when to print an input prompt, and which one (primary or secondary) to print.

Some implementations graft event handling onto an interpreter by making it an external command called by the user. These fail in that, until the event handler returns, the command line ceases to be available for more input.

There is another way to obtain the benefits of a generalized event handler at minimal cost in terms of changes to the Python code. The approach I chose places the event dispatcher at the *lowest* point in the Python program call tree. Python normally reads input lines by calls to the *fgets* library routine, in the file *Parser/myreadline.c*. This call may be replaced by a call to a virtual function *Py_fgets*, declared and initialized in *myreadline.c* as:

```
char * (*Py_fgets)(char * buf, FILE * fp);
Py_fgets = &fgets;
```

This default initialization causes Python to operate without any change until the Gist module is imported. When that occurs, the line

```
Py_fgets = &gist_fgets;
```

is executed. *Gist_fgets* is the entry point to the event dispatcher. From that point on, Python's requests to read *stdin* are routed through the Gist event

handler. Where the old Python will block attempting to read *stdin*, Python Gist "blocks" in a *select* call. This means that the process is able to handle X Window events without any apparent change in its command-line interface.

There are some limitations to this approach, of course. Python doesn't get *all* of its input through the single *fgets* call in *myreadline.c*. There are some other places in the code where input is read, sometimes using other library routines or system calls. This problem could be overcome by a more thorough application of the original nostrum — that is, replace all calls for I/O by appropriate virtual function wrappers. (In fact, the single replacement in *myreadline.c* seems quite effective in practice — apparently, Python rarely goes elsewhere for input.)

A more fundamental limitation of my approach is that there is only one virtual function to go around. If a module other than Gist also wants to redirect *Py_fgets* to its own end, this would be difficult, to say the least. It would be better to provide X event handling in a separate standard Python module so that modules like Gist would simply register and remove their specific handlers as they came and went, instead of grabbing *Py_fgets*.

Finally, the current scheme is only as portable as the UNIX *select* system call. Gist has previously been ported to Macintoshes, although that port does *not* allow mouse event handling. Other ports are obviously interesting, but are beyond the scope of this paper. There may be other approaches entirely, such as co-operative multi-tasking, or threading, which can solve the problem of multiple input streams more generally and more portably.

Future Work

Future work on the Python Gist module depends on the requirements and wishes of its users. The current interface is unabashedly functional (as opposed to object-oriented), and we have already considered some ways to make the interface more OO. Presumably this would involve objects such as *curves* and *surfaces*, with attendant *methods* to plot them, change their color, size, orientation, and so forth. However, the natural *methods* and attributes of graphic objects may not map naturally to the currently available routines in the functional interface. For example, the current X-Y plotting function *plg* implicitly draws axes, bounding boxes, tick marks, and labels in addition to the curve of interest, and this may well interfere with the design of methods for a *curve* object. So there will clearly be some iteration involved in adapting a different style of interface to the module.

We anticipate that 3-dimensional graphic output will be needed soon. David Munro has done some preliminary work extending the Gist library in this direction. Along with 3-dimensional output, there will be the need to

handle non-quadrilateral meshes. Such *unstructured meshes* will generally be described using lists of edges, nodes, and cells, so we necessarily have to develop or adopt conventions for handling this information in a graphic context.

The users of Gist have often wished for simpler means to edit their plots for display purposes. (The *plq* and *pledit* functions currently provide a command line interface to *query* the identity of graphic objects in a plot, and then *edit* their attributes.) It's not clear how best to proceed in solving this problem. We hope that introducing Gist into Python may provide some new ideas, avenues, and tools for improving Gist itself, either by allowing us to more conveniently categorize and manipulate the objects in Gist, or by giving us access to other graphic tool kits in the same program.

Finally, I expect to consider the question of porting the Python Gist module to other platforms such as Macintosh and/or Windows. A Macintosh port of Gist already exists for Munro's *Yorick* program, although it is not fully capable in the area of mouse input. A Windows port will probably require a new *OpenGL* graphics engine for Gist (at least), but this work may have other benefits in the area of 3-dimensional primitives.

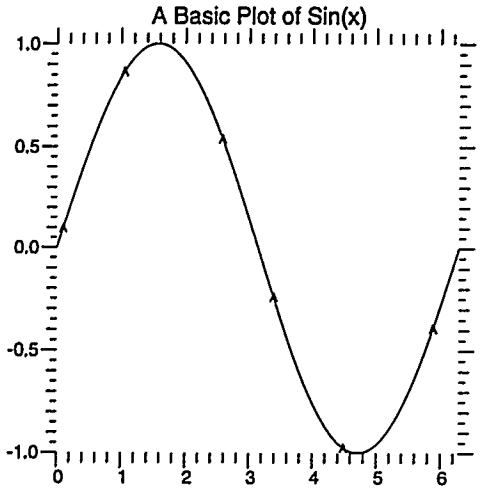
Availability

The Python Gist module, the Gist C extension module, and associated other files are available in gzipped tar format at

ftp://icf.llnl.gov:/pub/Python Gistmodule.tar.gz
(or at python.org.)

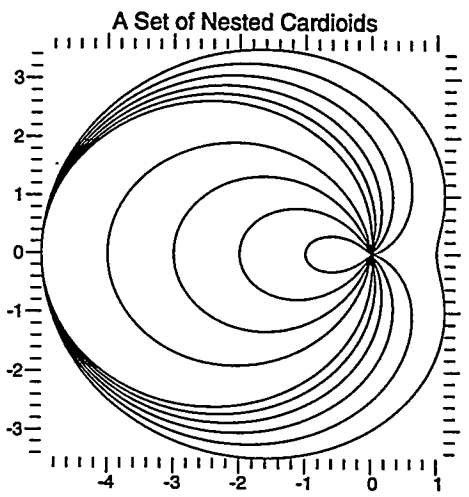
The Gist graphics library and Gist CGM browser are both included in David Munro's *Yorick* distribution, which may be obtained at ftp://icf.llnl.gov:/pub/Yorick. At the time of this writing, the current version of *Yorick* is 1.2. Please direct bug reports, problems, or questions to the author at <busby1@llnl.gov>.

Example output



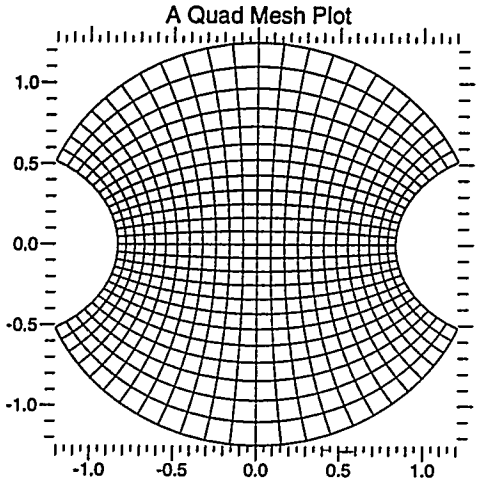
Produced by:

```
x = 2*pi*arange(200)/199.
plg(sin(x),x)
```



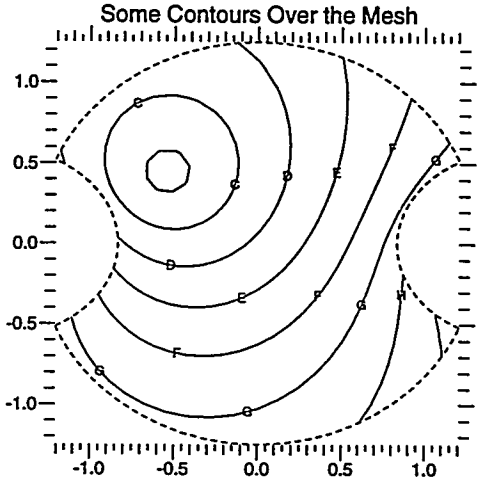
Produced by:

```
x = 2*pi*array(arange(200),'d')/199.0
for i in range(1,7):
  r = 0.5*i - (5 - 0.5*i) * cos(x)
  plg (r*sin(x), r*cos(x), marks=0)
```



Produced by:

```
x = ((arange(25*25)%25).reshape(25,25))/12. - 1
y = x.transpose()
z = x+1j*y; z = 5.*z/(5.+z*z)
xx = z.real; yy = z.imaginary
plm(yy,xx)
```



Produced by:

```
plm (boundary=1, type=2)
plc (mag (x+.5,y-.5), marks=1)
where x and y are as before, and mag is
  def mag(*args):
    r = 0
    for i in range(len(args)):
      r = r + args[i]*args[i]
    return sqrt(r)
```

Acknowledgments

Writing this module required little invention. Rather, I have stitched together the original work of several others. I am grateful to Paul Dubois for encouraging the project, to Zane Motteler for conversation about class structures for graphical objects and to Judy Harte for contributing test files and suggestions about test procedures.

I appreciate the assistance of Mark Hammond and Geoff Philbrick in helping me puzzle out how to handle keyword arguments, and I appreciate Guido van Rossum's answers to my numerous questions about Python internals.

This module would have been impractical without the contribution of J. Hugunin, K. Hinson, J. Fulton and others to the Python "Numeric" module. I am grateful to David Munro for patiently answering my many questions about Gist internals and X event handling.

Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under Contract W-7405-ENG-48.