UCRL-MA-128569, Manual 4

The Python Graphics Interface, Part I

EZPLOT User Manual

Written by

Zane C. Motteler
Lee Busby
Fred N. Fritsch

November 23, 1998

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

DISCLAIMER

Table of Contents

CHAPTER 1: The Python Graphics Interface

1.1 Overview of the Python Graphics Interface

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

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

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

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

tion (moving light sources and rotations). The Python Gist module `gist.py` and the associated Python extension `gistCmodule` provide a Python interface to this library (referred to as PyGist).

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

1.2 Using the Python Graphics Interface

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

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

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

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

1. Set your environment variable `PORT_SERVEUR`[2] to 0.

1. I am going to assume that you already have Narcisse installed on your system, and its directory path in your `PATH` variable.
2. We did tell you that Narcisse was French, didn't we?

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

1.3 About This Manual

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

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

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

PyGraph is available on Sun (both SunOS and Solaris), Hewlett-Packard, DEC, SGI workstations, and some other platforms. Currently at LLNL, Narcisse is installed only on the X Division HP and Solaris boxes, however, and Narcisse is not available for distribution outside this laboratory. Our French colleagues are going through the necessary procedures for public release, but these have not yet been crowned with success. Gist, however, is publicly available as part of the Yorick release, and may be obtained by anonymous ftp from `ftp-icf.llnl.gov`; look in the subdirectory `/ftp/pub/Yorick`.

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

- Lee Busby of LLNL, who wrote `gistCmodule`, and wrought the necessary changes in the Python kernel to allow it to work correctly;

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

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

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

CHAPTER 2: Introduction to EZPLOT

EZPLOT is a function-call-driven interface to PyGraph intended to resemble the Basis EZN Graphics Package, which is described in "EZN User Manual," UCRL-MA-118543 Pt 3. The primary difference is that calls to EZPLOT will look like function calls, rather than the command line format familiar to users of Basis and EZN.

Currently EZPLOT does only two-dimensional plots, and even with these, implements only a subset of what is available in the EZN package. Users wishing to do more elaborate two dimensional plots, or three dimensional plots, will have to use OOG or the extremely low level `narcissemodule` for Narcisse, or else OOG or the low level 3-D Gist plotting functions described in the Python Gist Graphics Manual.

It is possible that if the use of EZPLOT expands sufficiently, and enough users request additional features, then these features may be added.

2.1 Running EZPLOT

Assuming that you have set your `PYTHONPATH` environment variable to point to the subdirectories containing the Python modules which you intend to use, you should start up Python by typing

```
python
```

at the unix prompt; you will then receive the Python prompt ">>> " , at which you type the following two commands:

```
>>> from Numeric import *
>>> from ezplot import *
```

The first command puts the names of all the NumPy functions in your name space, and the second does the same with the EZPLOT functions.

If you prefer to keep name spaces separate, then you can do the following:

```
>>> import Numeric
>>> import ezplot
```

Then you can give these modules shorter names (for typing convenience), such as

```
>>> num = Numeric
>>> ez = ezplot
```

and then use the ‘‘dot’’ notation to refer to functions within the modules, e. g.

```
>>> ez.cgm ("close")
```

In what follows, for simplicity, we shall assume that the first form of the `import` statements was used.

2.2 The Additive Model

The basic model of this package is that of additive graphic functions to a single frame. That is, each graphic function call adds objects (curves, mesh plots, etc.) to a frame. The frame is not complete until a newframe “`nf ()`” function call is issued. The user controls whether or not to see each step in building a frame or just viewing the completed frame by setting the `ezcshow` status to `"true"` or `"false"`. In EZPLOT, this is done by invoking the function `ezcshow`, e. g.,

```
>>> ezcshow ("false")
```

`ezcshow` is fairly tolerant; it will accept any string beginning with “`t`”, “`T`”, “`y`”, or “`Y`” as “`true`”, and any string beginning with “`f`”, “`F`”, “`n`”, or “`N`” as “`false`”.

EZPLOT begins in interactive mode (the `ezcshow` status is `"true"`), so that each function call that changes the frame causes the whole frame to be redrawn. However, most programs using EZPLOT will probably want to set `ezcshow` to `"false"` when making plots, so that each frame is displayed only when finished. If you stop the program and want to view the plots as they are made, you must either reset `ezcshow` to `"true"` or use the showframe “`sf ()`” function.

Caution: When using multiple windows in interactive mode, be aware that “`nf ()`” (the new frame function) clears the display list, but only clears the currently open window. If you then change windows, you will have to issue another “`nf ()`” call to avoid overplotting any graph already on the window.

2.3 Controlling Layout

EZPLOT supports a subset of what EZN users might be accustomed to. The standard EZPLOT picture can be described as follows. There is a margin around the edges of the graph leaving room enough for titles at top, bottom, left, and right. In a contour plot, sufficient additional space is left at the right for a color bar which associates the contours with particular colors. More space around the edge of the plot is taken by the axes, unless the user suppresses the axes, in which case the area taken up by the plot may be somewhat larger.

Unlike EZN, EZPLOT does not allow you to change these values from their defaults. It is possible that with sufficient demand, these capabilities may be added at some time in the future. Users who can not afford to wait are encouraged to use the OOG, which has far more flexibility, or the low level interfaces `gistCmodule` and `narcissemodule`, which give access to the full machinery of the graphics engines.

2.4 Plot Function Summary

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

- Device and frame control functions (CHAPTER 3: “Devices”)

```
win (<cmd> [, n] [, <keylist>])
cgm (<cmd> [, n] [, <keylist>])
ps (<cmd> [, n] [, <keylist>])
tv (<cmd> [, n] [, <keylist>])
list_devices ()
frame ([xmin [, xmax [, ymin [, ymax]]]]
     [, window = val])
fr ([xmin [, xmax [, ymin [, ymax]]]] [, window = val])
nf ([new_frame = val1] [, window = val2])
sf ([window = val])
undo ([number])
```

- Attribute functions (CHAPTER 4: “Attributes”)

```
attr (keyword=value [, keyword=value ...])
              # set color, thickness, etc.
```

- General plot functions (CHAPTER 5: “General Plot Commands”)

```
plot (y, x [, <keylist>]) # curves, markers
plotz (fexpr [, xexpr [, yexpr]] [, <keylist>])
     # contours
ploti (<keylist>) # cell array plot
```

- Mesh-oriented functions (CHAPTER 6: “Mesh-Oriented Commands”)

```
set_mesh (<keylist>) # establish default mesh
clear_mesh ()        # erase default mesh
ezcpvar (val)        # set plotted variable for mesh
ezccindex (val)      # set color index for mesh
ezcx (val)           # set abscissa for mesh
ezcy (val)           # set ordinate for mesh
ezcireg (val)        # specify regions in mesh
ezcv (val)           # set x component of velocity
ezcu (val)           # set y component of velocity
plotm (<keylist>)    # plot mesh
plotb (<keylist>)    # plot region boundaries in mesh
plotc (<keylist>)    # plot contours of a mesh-based quantity
plotf (<keylist>)    # fillmesh plot
plotv (<keylist>)    # plot velocity field
```

- Text plotting and miscellaneous (CHAPTER 7: “Text Plotting and Miscellaneous”)

```
titles ("top"[, "bottom"[, "left"[, "right"]]])
text ("message", x, y, charsize [, <keylist>])
```

You can use attributes and the values of user-settable variables to control the detailed behavior of these functions. Attributes are explained in CHAPTER 4: “Attributes”, variables in CHAPTER 8: “Control Variables and Defaults”.

CHAPTER 3: Devices

EZPLOT has functions to control graphics devices. The devices supported by EZPLOT with PyGist graphics are CGM files, PostScript files, and Xwindows. The PyNarcisse graphics engine produces plots in an Xwindow, and optionally its own brand of files, which can not be sent directly to a printer, but which can be loaded into a PyNarcisse window and sent to a PostScript file or printer interactively.

A user can open multiple devices and direct the same or different graphics output to different devices. EZPLOT supports up to eight windows and/or files at a time. There can be at most one CGM file and one PostScript file open at a time, but there can be multiple windows. For example, a user can open several Xwindows, even at different workstations, and display different frames in different windows for comparison. When the user is satisfied with the result of a certain frame, say in window `n`, he/she can issue `cgm("send", window=n)` to record the frame into a CGM file.

Please note that there are major differences between PyNarcisse and PyGist, since you do not open PyNarcisse windows from within the graphics routines; instead, you must open one (or more) at the unix prompt prior to firing up the graphics routines. Thus the `win` function, which opens windows in PyGist, does not do so in PyNarcisse, but instead tries to find a PyNarcisse window to which to open a connection. Likewise, rather than specifying whether or not to write files via the graphics routines, you use the menus in the Narcisse GUI to do so. (Of course, as we shall see, one can simultaneously use PyNarcisse to draw plots in windows and PyGist to send the same plots to files.)

3.1 Device functions

The device functions are used to specify where the plot should go, the choices being PyNarcisse windows, PyGist windows, or PostScript or CGM files (PyGist only). If you issue a plot command before specifying at least one device, PyGist defaults to a single CGM file. In fact, PyGist will *always* write to a CGM file *unless* you issue a "`cgm("close")`". PyNarcisse will attempt to find a running Narcisse process, and if it finds one, will plot to that process's window.

The device functions are of the form:

file-type (file-command [, device-number] [, `new_frame` *= <str>])*

`win` *(win-command [, device-number] [,* `display` *= <str1>]*

[, `graphics` *= <str2>])*

The *file-type* function is only valid for PyGist graphics. *file-type* can be `cgm` or `ps`. *file-command* can be: `"on"` (or `"open"`), `"off"`, `"close"`, `"send"`, or `"plot"`. *device-number* can be a number from 0 to 7, and if not specified, defaults to the lowest available number in that range. `new_frame`, if specified, must be `"yes"` or `"no"`. See below for explanations.

win-command can be "on" (or "open") and "off" (or "close"). *device-number* can be a number from 0 to 7, and if not specified, defaults to the lowest available number in that range. For an explanation of the display and graphics keywords, see below.

The device cgm is a CGM file. The CGM file stores the frames of graphics output. Under PyGist, a standard CGM file is produced, with suffix .cgm. The filenames default to Aa00.cgm, Ab00.cgm, etc.

The device ps is a PostScript file, which has suffix .ps. The PS file stores the frames of graphics in the PostScript format. The filenames default to Aa00.ps, Ab00.ps, etc.

The device win (or tv) is an Xwindow on a certain display. The PyGist display is the network address of the device where the plot will be displayed, e. g. "icf.llnl.gov:0.0". If not specified, it will be set by the user's environment variable DISPLAY. The PyNarcisse display is more complicated and is described in section 3.2.2 "PyNarcisse". The keyword graphics is used to specify the type of graphics engine for this particular window; allowed values are "Gist" and "Nar". If none is specified, then the graphics will be as specified by the user's environment variable PYGRAPH, or "Gist" if that variable is unset.

The command "on" or "open" opens a device if the device has not been opened. Then "on" activates the device. It has no effect on the device if it is currently active.

The command "off" deactivates an opened file (but the linkage to the file for controlling still exists). The command "close" deactivates and then closes the file. Beware, however; "off" and "close" behave exactly the same for a graphics window, namely, they cause the window to close (i. e., go away forever).

The command "send" sends the current frame (see the next paragraph for the meaning of ''current frame") to the specified CGM or PS file; the send command turns on the device (i.e. CGM or PS file), sends a frame, and then turns the file off. The command "plot" also sends the current frame to the specified CGM or PS file; the difference is that the file is not turned off after the frame is displayed. The keyword new_frame is only meaningful with the send and plot commands. If "no", then any new graphical components will be added to the current frame and displayed. If "yes", then the new graph will not be displayed until a nf() is issued, which also has the effect of erasing the current display list.

And now, what is the current frame? The cgm or ps function has a keyword argument window which can be used to specify the number of an open window, or in the case of multiple windows, window = "min" will choose the window with the smallest number which has a nonempty display list, and "max" will choose the one with the largest number; this wndow's display list will then be sent to the CGM or PS file. Lacking this keyword, the command looks first to see if the CGM device itself has a display list, and if not, defaults to "min". If no display list can be found, an exception is raised.

For PyGist graphics, the window name appears in the title bar of the window. The window title will be ''PyGist n'', where n is the number of the window (an integer between 0 and 7, inclusive). For users with multiple windows, the function call list_devices() will give an informative printout listing the numbers of open devices, their status (e. g., in the case of CGM and PS files, active or closed), what type of device they are, their graphics, and their display.

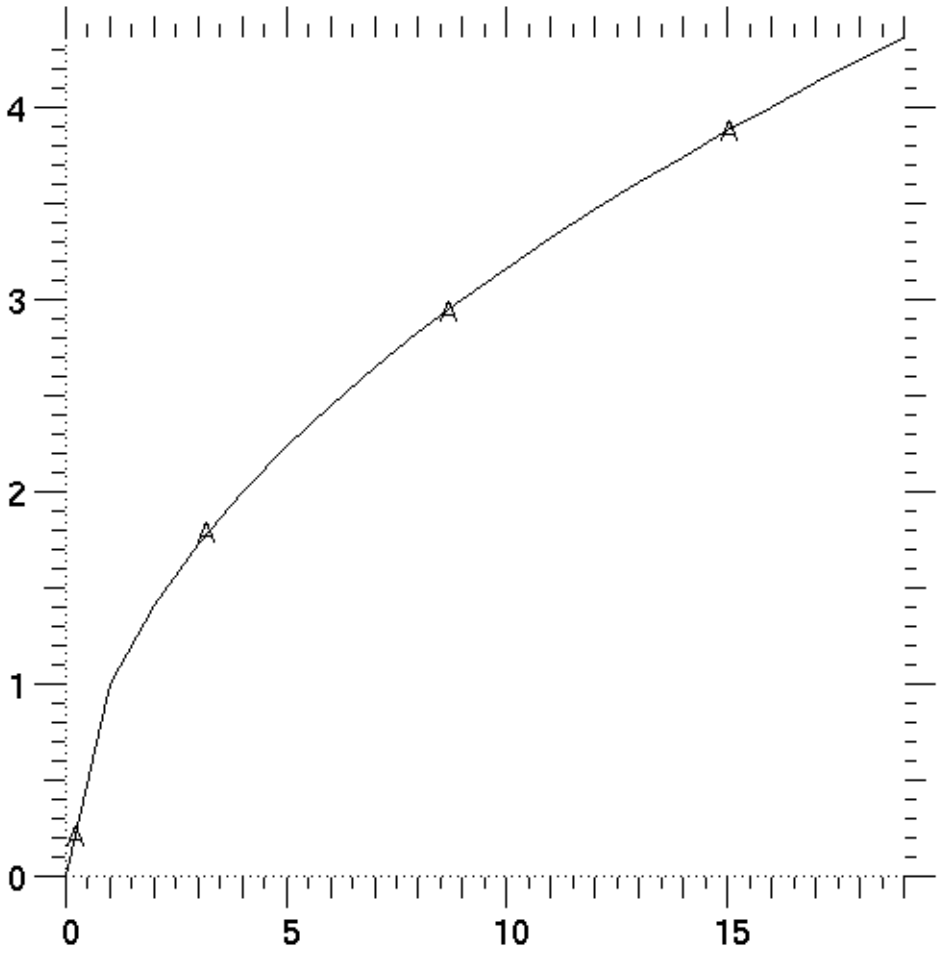
EZPLOT keeps track of the number of active devices. If a plot function is issued without any active

device, EZPLOT will open a CGM file as a default device to accept the plot function call.

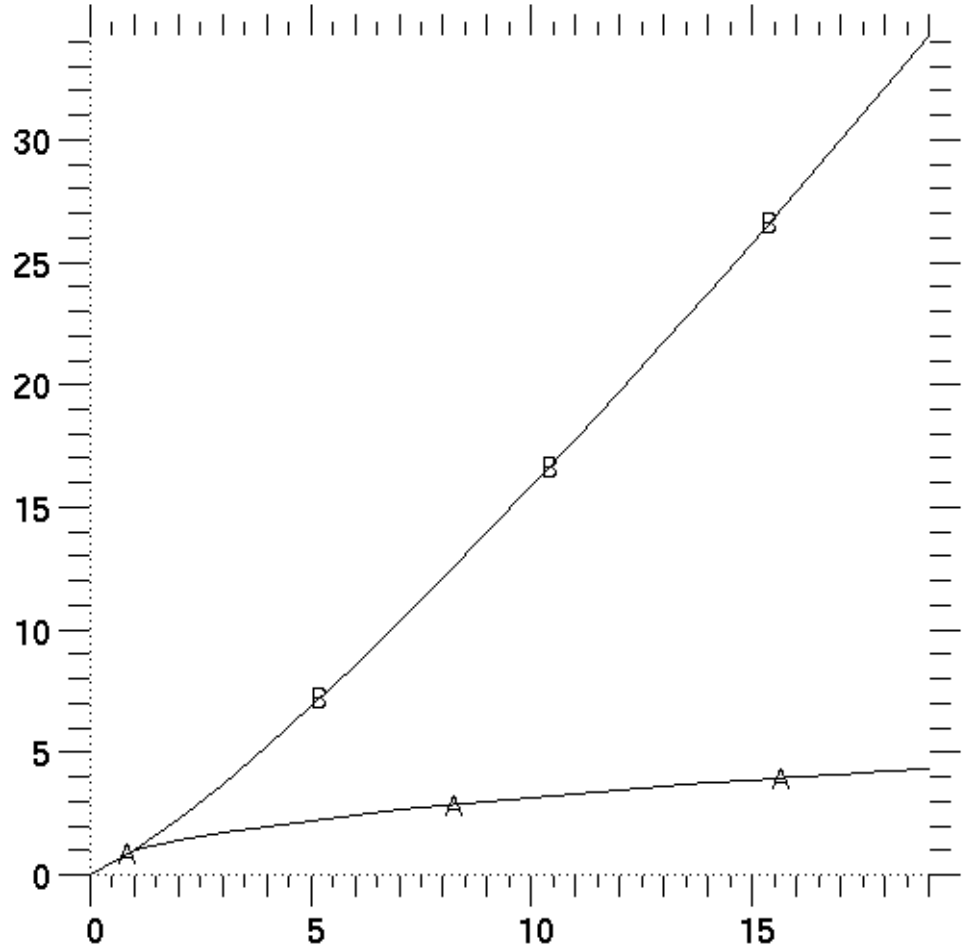
Example 1

This example illustrates the use of the "`open`" and "`send`" commands. (As with most examples in this document, we are assuming that the necessary graphics files and module `Numeric` have been imported.) We show the following set of EZPLOT function calls, first with comments explaining what happens with PyGist graphics; afterwards, we comment on how the PyNarcisse version will differ. We invite the reader to type these commands in and follow along.

```
# Assume PyGist graphics first (PYGRAPH set to Gist)
win ("on")
     # Open an Xwindow with name PyGist 0.
     # In PyGist, the window does not open until the
     # first plot is sent:
plot (arange (20, typecode = Float) ** 0.5)
     # Note: arange is the Python equivalent of iota.
```

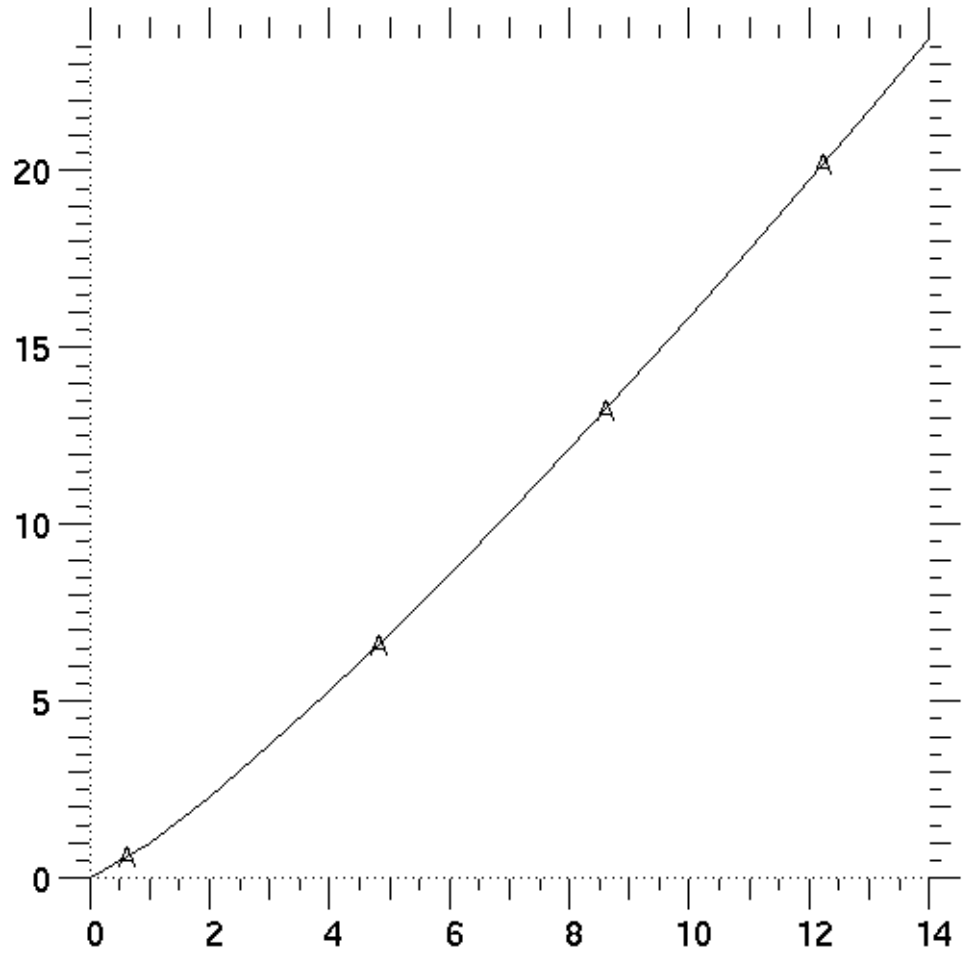


```
     # Add a second curve to the plot:
plot (arange (20, typecode = Float) ** 1.2)
```



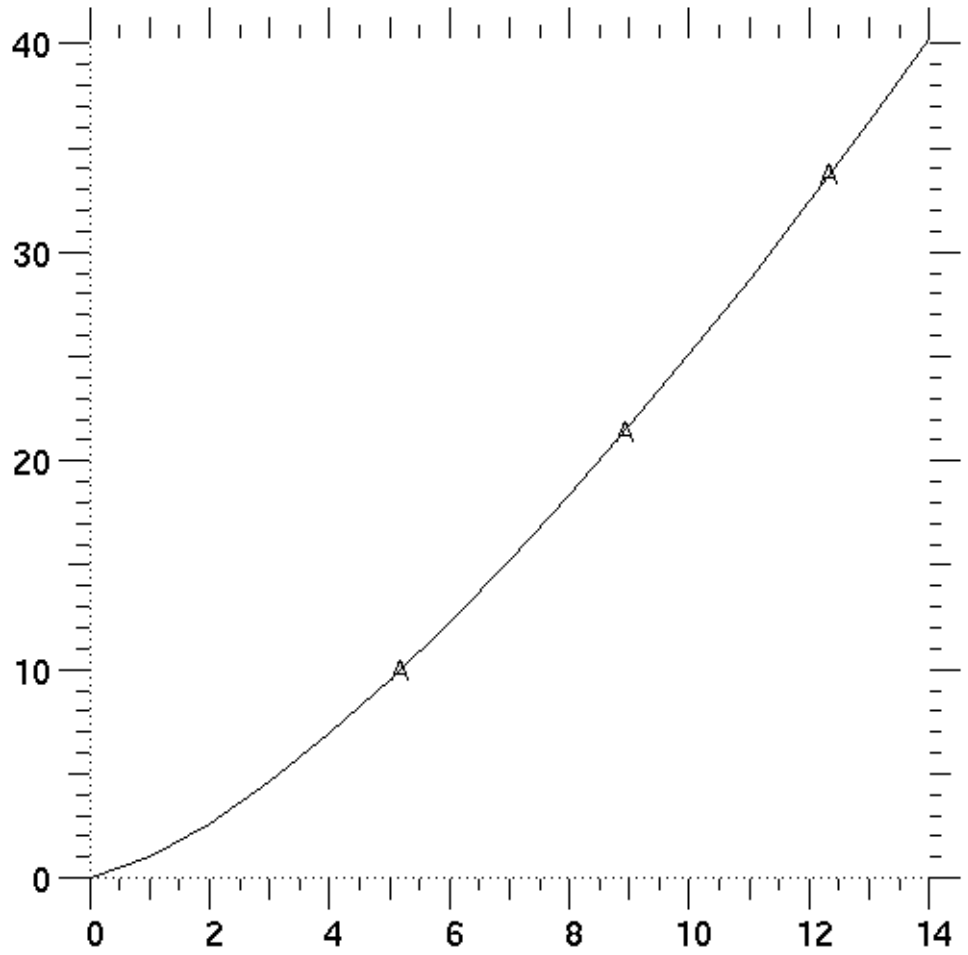

```
     # Open CGM file, send a frame to it with two curves,
     # then immediately set CGM file to off:
cgm ("send")
     # Prepare for next frame (plot will not be replaced
     # until next plot command is given):
nf ()
```

```
     # The plot appears on the window after we do:
plot (arange (15, typecode = Float) ** 1.2)
```




```
     # Activate the CGM file to accept a frame,
     # then deactivate the CGM file:
cgm ("send")
     # Open PS file, send a frame to it,
     # then deactivate the PS file:
ps ("send")
nf ()
```

```
     # The new plot appears on the window:
plot (arange (15, typecode = Float) ** 1.4)
```




```
     # Re-activate PS file, send a frame,
     # then deactivate the PS file:
ps ("send")
^D   # CTRL-D terminates Python.
     # Close all devices; close CGM file and PS file.
```

Note that PyGist curves, as they appear, are marked along their extent by the letter ‘‘A’’ for the first, ‘‘B’’ for the second, etc. It is possible to draw curves without such markers, or to specify your own; it is also possible to plot curves as dots, dashes, etc., as we shall see later. In the default plotting mode, as above, each successive plot will appear on the same frame as all of the previous; the function `nf ()` must be called to force PyGist to start a new frame.

Now try the same sequence of function calls with PyNarcisse graphics. You must have `PYGRAPH` set to `Nar`, and a PyNarcisse window must be open, with your `PORT_SERVEUR` variable agreeing with the port number in the window. (If you fail to do this, the PyNarcisse software will be unable to

find the window, and will go into a perpetual loop trying to make a connection.) After each plot command, the new curve will appear on the plot. Notice that PyNarcisse curves will be labeled by the letters ‘‘A’’, ‘‘B’’, etc. at the right ends. This is because PyNarcisse does not support in-curve labelling, and this was the closest approximation to the PyGist behavior that we could think of. If you use other options to label your curves, then your chosen labels will overrule these letters.

As we shall see later, it is possible to have both PyGist and PyNarcisse active at the same time, so a PyNarcisse plot can be sent to a PyGist CGM or PS file. Unfortunately, the PyGist and PyNarcisse graphs will usually not look exactly alike.

3.2 Working with Multiple Windows

3.2.1 PyGist

If multiple windows will be used, then again the situation differs considerably between PyNarcisse and PyGist. PyGist is fairly straightforward in that each opened window will automatically be named ‘‘`PyGist n`’’, where `n` is the window number. It is up to the user to keep track of which is which. It is possible, using the `window = n` keyword argument to `plot`, `sf`, and other frame control functions, to display different plots in different windows for comparison purposes. (Normally a plot or frame command will display the current plot in all open windows and CGM or PS files which have not explicitly been turned `off`.) It is also possible, using the `display` and `graphics` keyword arguments to the `win` function, to open a PyGist window on a remote machine.

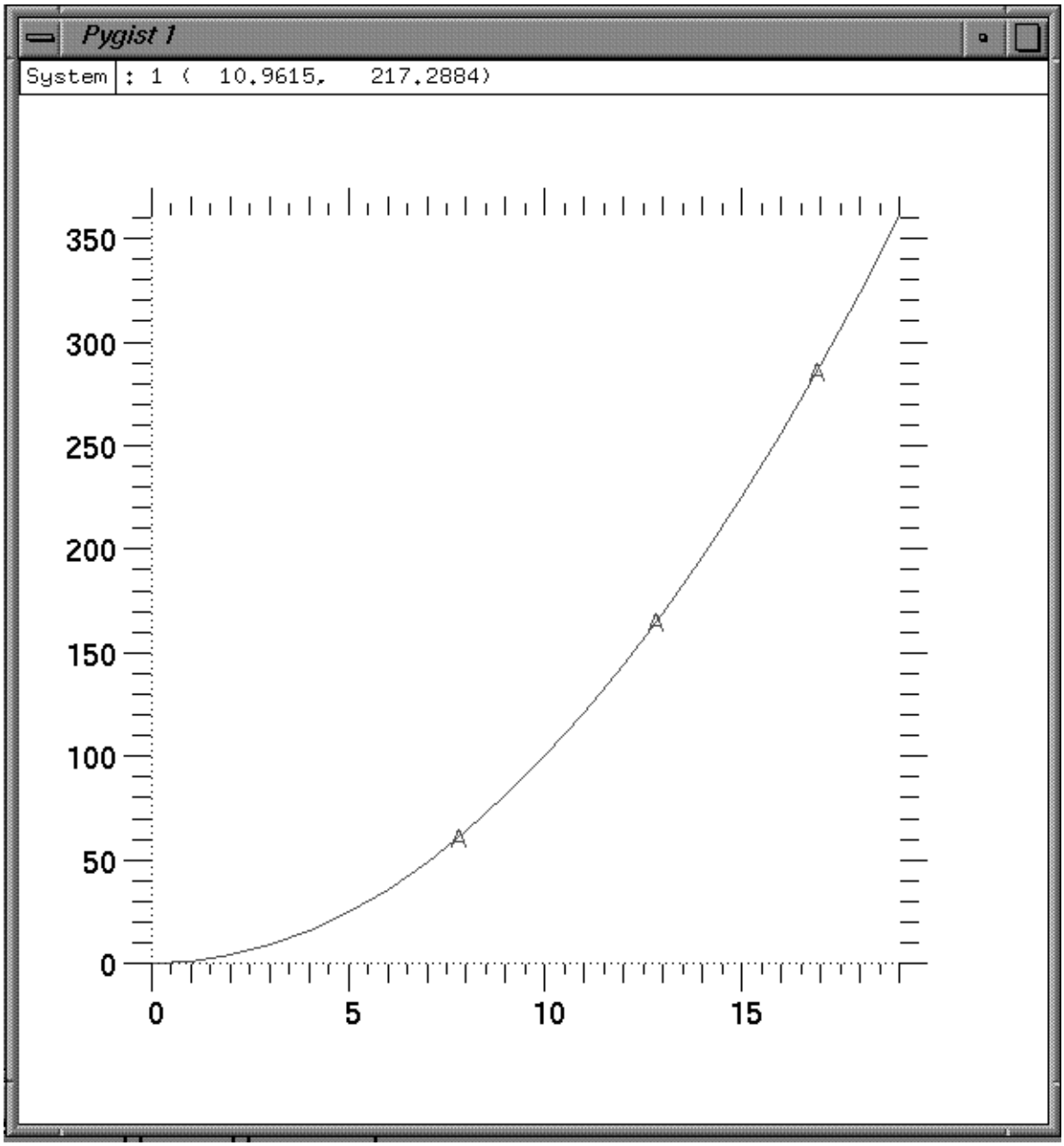
For making plots, changing frames, and closing windows or files, the `window` keyword argument can be assigned a single number between 0 and 7 (the number of any open window or file), or a python list of such numbers (between square brackets, comma separated), the string `"min"` (select the smallest open device number), the string `"max"` (select the largest), or the string `"all"`, which will select all open windows and files. PyGist keeps a separate display list for each opened window or file, so you can display quite different graphs on different devices. Upon opening a window or file, `window` defaults to the smallest unused device number. In most other commands, window defaults to `"all"`, so if you want different graphs in different windows, you must be specific.

Note that if a second window is opened in the same display, it will most likely appear on top of the first and will need to be repositioned to make both visible on the display. Study the example below for more information on working with multiple windows.

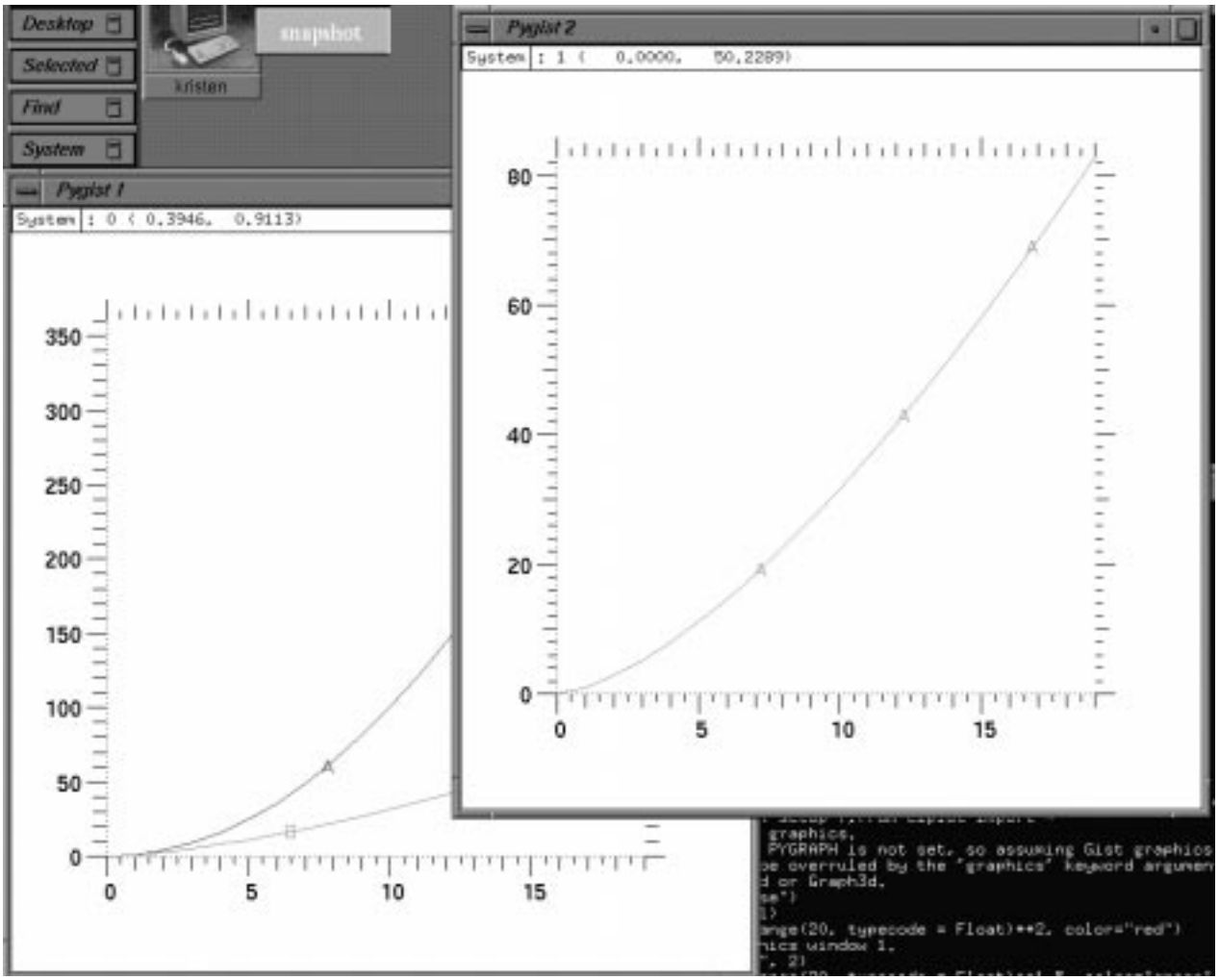
Example 2

Try out the following commands to gain experience working with multiple devices.

```
cgm ("close")  # repress cgm; always on otherwise
tv ("on", 1)   # (Same as win ("on", 1))
     # Create a window named "PyGist 1".
plot (arange(20, typecode = Float)**2, color="red")
# (plot on next page)
```

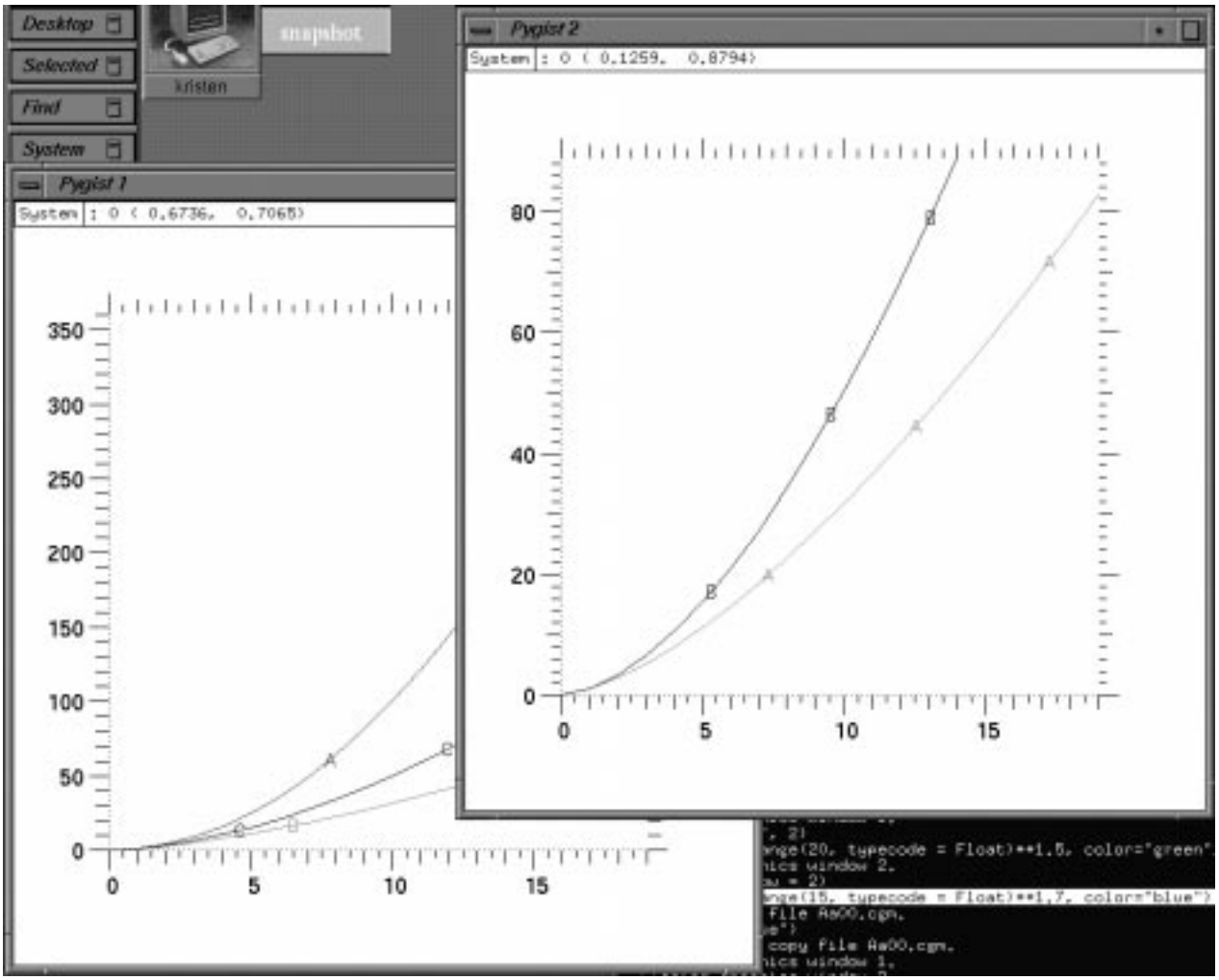

Pygist 1
System : 1 (10.9615, 217.2884)
350
300
250
200
150
100
50
0
0
5
10
15

```
win ("on", 2)
     # Create another window named "PyGist 2";
plot (arange(20, typecode = Float)**1.5, color="green",
     window = [1, 2])
     # The new plot will appear on PyGist 2. Note: both
     # curves appear on PyGist 1, since we did not do
     # nf (1).
```



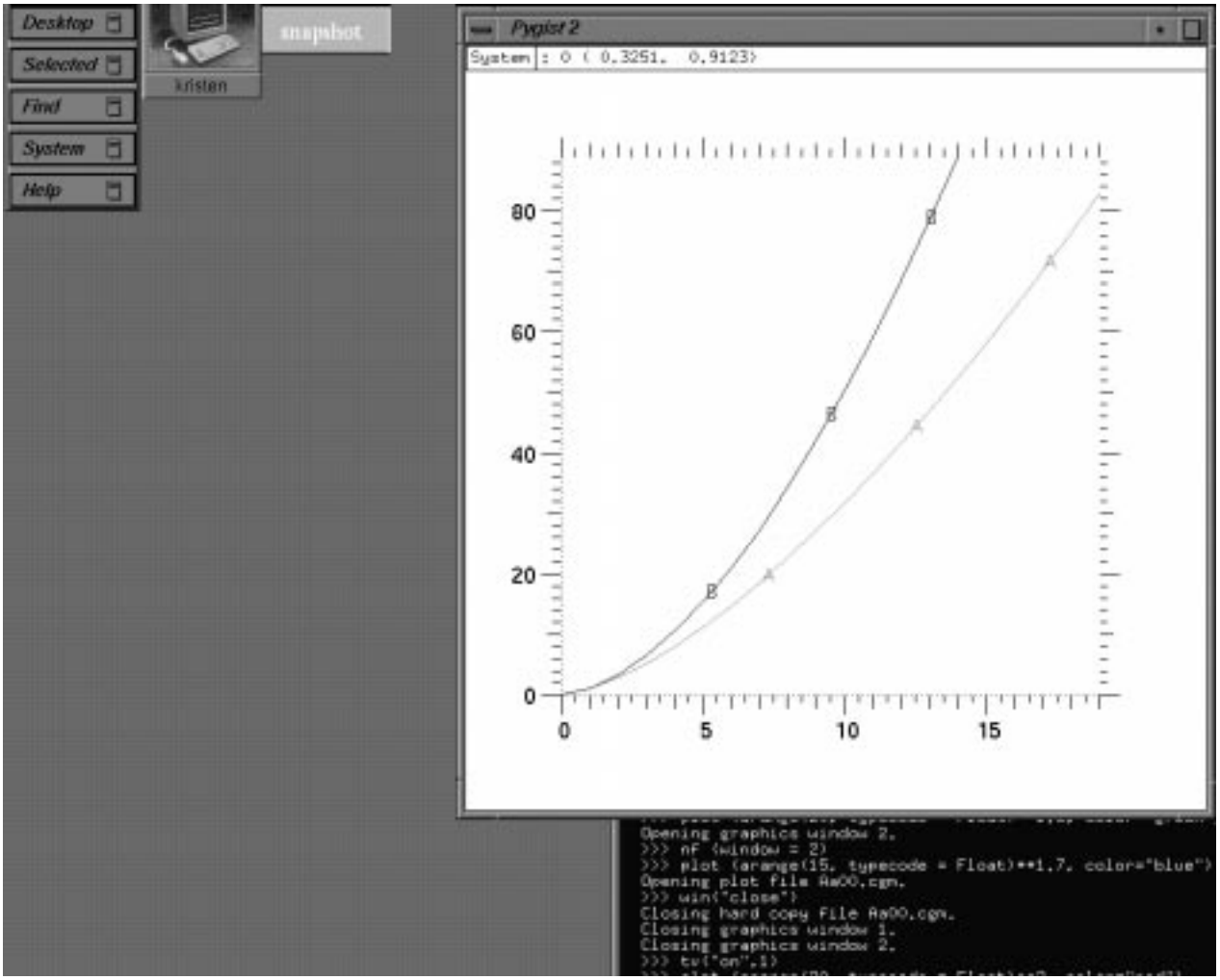

```
nf (window = 2) # Set a new frame (window PyGist 2 only)
plot (arange(15, typecode = Float)**1.7, color="blue")
     # Assuming ezcshow status "true", the plot will appear
     # on both windows.  On PyGist 2 it will be on a fresh
     # screen, but on PyGist 1, it will appear superimposed
     # on what was there already, because "nf" applied only
     # to window 2.  nf () (with no arguments) clears all
     # windows; nf (window = 1) would be necessary to clear
     # only window 1. (Plot is on next page)
```

. . .

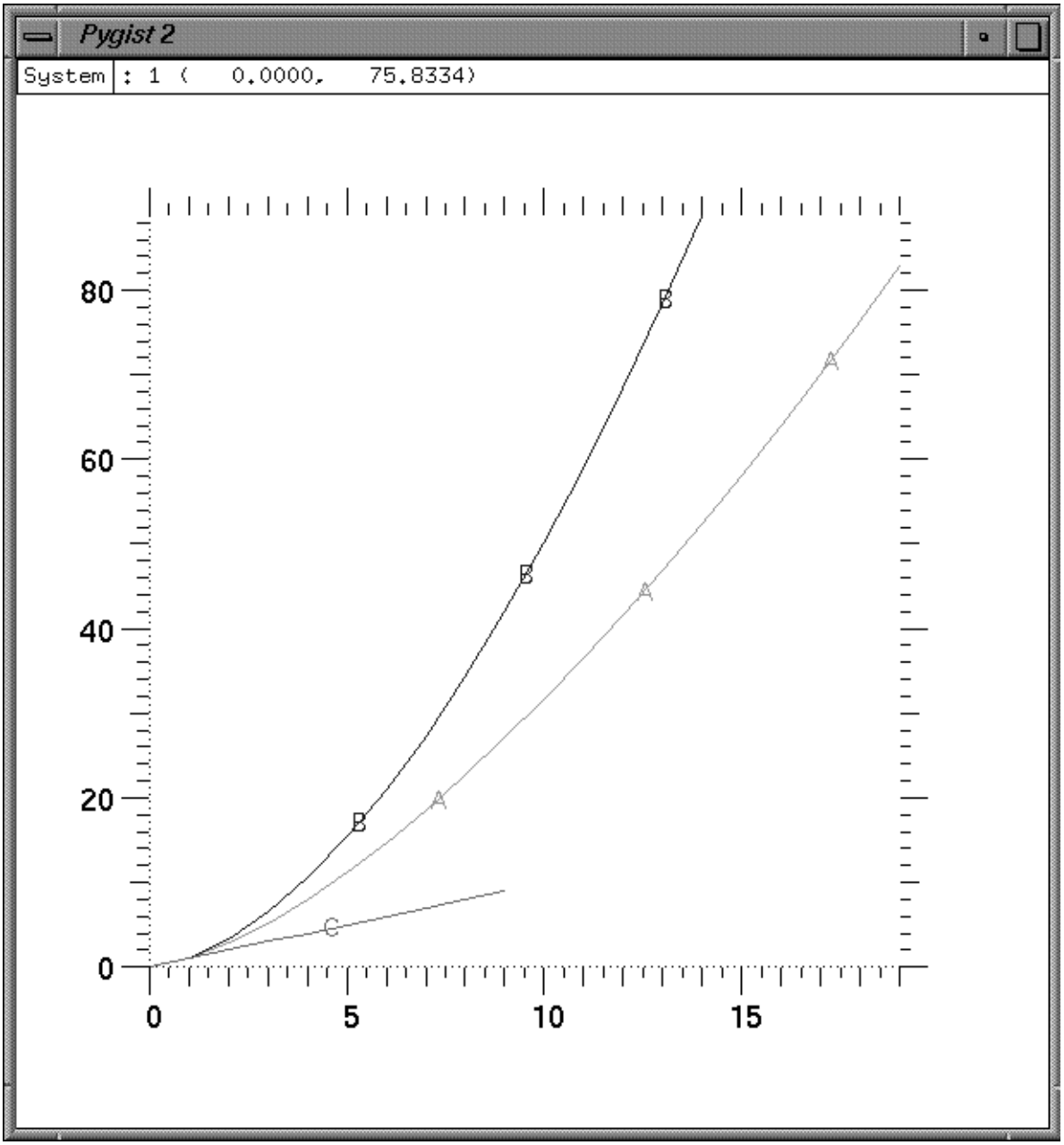


Note that curves within a window are lettered consecutively within that window. Thus the new curve is labeled B in window PyGist 2, and C in window PyGist 1.

```
win ("close", 1)
     # Close PyGist 1; now PyGist 2 is the only one active.
```



```
nf ()
plot (arange (10, typecode = Float), color="purple")
```




```
win ("close", 2)
     # Close PyGist 2; now no devices are active.
nf ()
...
cgm ("on")
     # Open a CGM file to accept the following plots.
plot (arange(20, typecode = Float), color="yellow")
```

```
     # A frame is send to the CGM file.
ps ("on")
     # Open a PostScript file.
plot (arange(20, typecode = Float)**1.2)
     # A frame has been send to both CGM and PS file.
     # The CGM file will contain both curves.
nf ()
win ("on", display = "greystoke.llnl.gov:0.0", graphics =
     "Gist")
     # Open a PyGist window on a remote machine.
list_devices ()
     # Just in case you forgot which device is which,
     # this function gives an informative list.
plot (arange(20, typecode = Float)**1.8)
     # A frame has been send to CGM, PS and remote machine.
cgm ("close")
     # Close CGM file.
ps ("close")
     # Close PS file.
^D
     # Implicitly close all active devices; quit Python.
```

3.2.2 PyNarcisse

Working with multiple PyNarcisse windows is quite different from PyGist, because as you already know, you can not open new PyNarcisse windows from within Python. You must do this by typing in the `Narcisse` command at the unix prompt; the `win ("on")` command from a Python script or interactive session simply causes EZPLOT to connect to an existing Narcisse process[1]. If, as also mentioned previously, no Narcisse process is to be found, then Narcisse goes into a loop trying to connect to what it can't find. This is not very logical behavior, but ours not to wonder why....

To open multiple Narcisse processes on the same Xterminal, the easiest way is to open them from separate windows, following the same procedure, namely first setting `PORT_SERVEUR` to 0, then typing in ‘‘`Narcisse &`’’, then changing `PORT_SERVEUR` to the port number given on the GUI. Likewise, if you want to converse with a Narcisse process on a remote machine, you need to have a colleague open the window there. Or, if you can `rlogin`, you can do it yourself, provided the `DISPLAY` variable is set to point to the remote machine. However, you'll still have to have the colleague let you know what the `PORT_SERVEUR` number is.

To open a connection to a particular Narcisse process, local or remote, use the `display` and `graphics` keyword arguments to the `win` function. The PyNarcisse `display` argument must be a character string in the form `"hostname+port_number++user@ie.32"`, where the

1. Note that Narcisse, when started, opens only its GUI. An actual graphics window will not open until (a) you are connected to the Narcisse process, and (b) you send a plot to this process.

port_number is the one displayed by the GUI of the Narcisse with which you wish to communicate, and the hostname is where the server is running. An example string might be:

```
"kristen.llnl.gov+44812++motteler@ie.32"
```

Suppose, for instance, you want to form a connection with the Narcisse process identified by the above string as window number 2. You would do this as follows:

```
win ("on", 2, display =
     "kristen.llnl.gov+44812++motteler@ie.32",
     graphics = "Nar")
```

We encourage the user to open a couple of Narcisse processes and go through the example of the previous section, the difference being using the win function to connect with Narcisse as we have just showed you.

3.2.3 Using PyGist and PyNarcisse together

The EZPLOT graphics model supports up to eight open ‘‘devices’’, and there is no reason, therefore, why a user cannot have a CGM file, a PS file, and several windows, both PyGist and PyNarcisse, all active at the same time; and some of the windows can even be remote. Use the second argument to the functions cgm, ps, and win to specify the window number (if you do not wish EZPLOT to give you default numbers). Use the display keyword argument and the graphics keyword argument to the win function to specify where the display is to appear (and remember, the form of display is different for PyGist and PyNarcisse). The graphics keyword should be used for safety, although if it is not supplied, EZPLOT will use the environment variable PYGRAPH, if set, or will default to "Gist", if PYGRAPH is not set.

Finally, use the window keyword argument to the various plot functions and to nf and sf to control what plot goes to which device.

3.3 Setting the Colormap

EZPLOT currently does not support allowing the user to change the colormap (or palette) used in plotting. You can do this using OOG or the low-level gistCmodule or narcissemodule functions. (EZPLOT commands can be successfully mixed with low-level gistCmodule calls.) If there is sufficient demand, we will eventually add this feature to EZPLOT. The current default colormap is a rainbow palette, i. e., a set of colors running through red, orange, yellow, green, blue, indigo, and violet, just like a real rainbow. On non-color devices, this will be displayed as a greyscale, which is unfortunate, because some shades will be essentially invisible.

3.4 Frame Control

There are three functions which control frame actions. The frame function sets the limits of the picture frame. The nf (New Frame) function is used to begin a new frame. The sf (Show Frame) function is used to display the current frame to all active devices. The undo function removes the most

recently plotted function from the display list of the specified device(s).

3.4.1 `frame`: Set Frame Limits

Calling Sequence

```
frame ([xmin [, xmax [, ymin [, ymax]]]] [, window = val])
fr ([xmin [, xmax [, ymin [, ymax]]]] [, window = val])
```

Description

The `frame` function sets the limits of the picture frame, which are *frame* type attributes. The `frame` command applies immediately to all plot commands in the frame. `fr` is an abbreviation for `nf` followed by `frame`.

You can supply zero to four positional arguments. If specified, *`xmin`* is the minimum value for the x scale, *`xmax`* is the maximum value for the x scale, *`ymin`* is the minimum value for the y scale, and *`ymax`* is the maximum value for the y scale. These positional arguments may be omitted from the right[1]; or, if you wish, you can specify any subset of the arguments using keywords, e. g.,

```
frame (xmax = 9.32, ymax = 1.05e11)
```

Those arguments omitted will be calculated from the data. The `window` keyword may be used to specify a particular window or file to which you wish to apply the limits. Allowed values are the number of an open window from 0 to 7, `"cgm"`, `"ps"`, or `"all"` (the default).

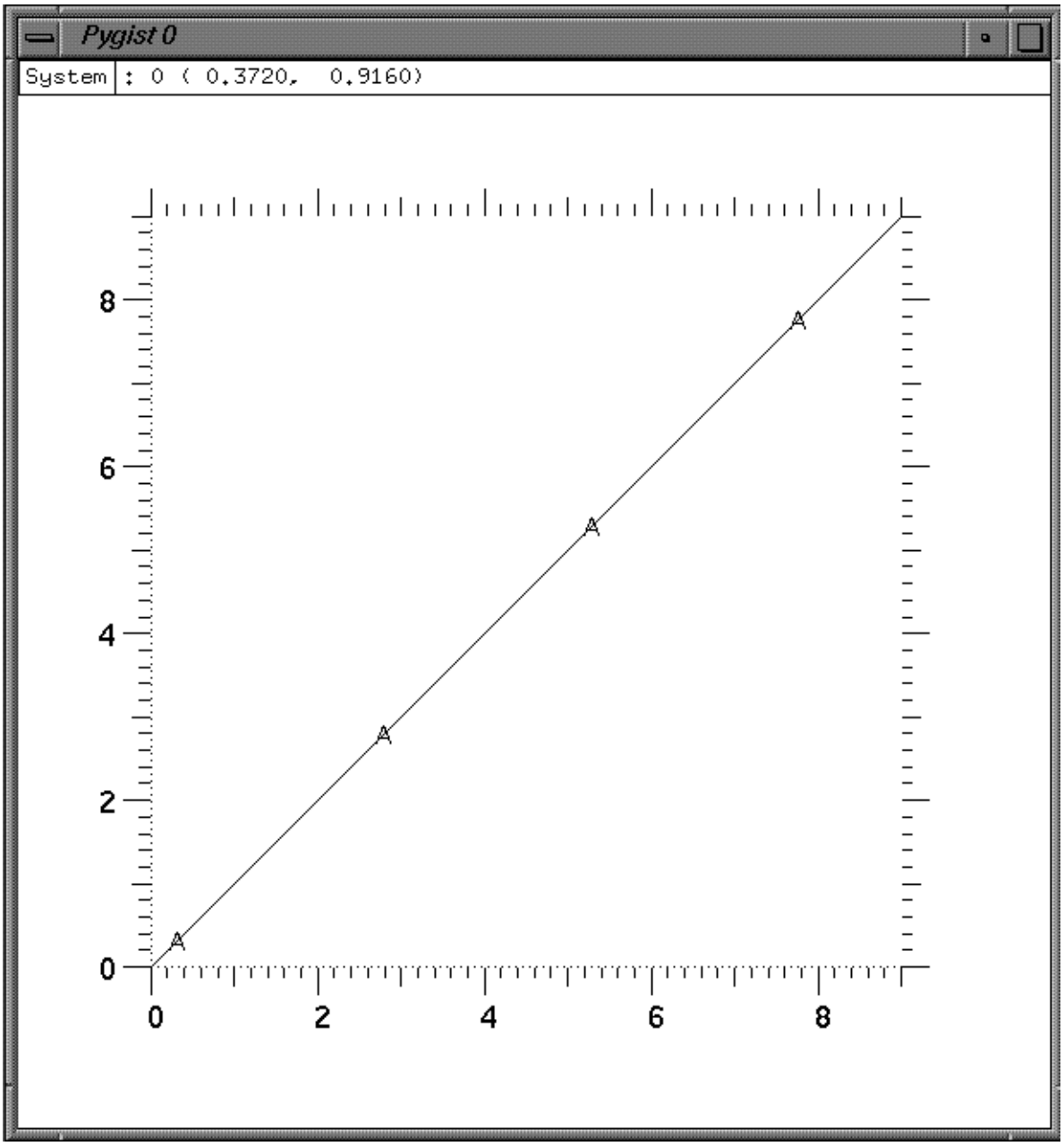
The frame limits will not be retained across frame advances. If a frame already contains objects it will be displayed with these frame limits. Currently the same frame limits apply to all open windows; if sufficient demand surfaces, we will implement separate frame limits for separate devices.

Example 3

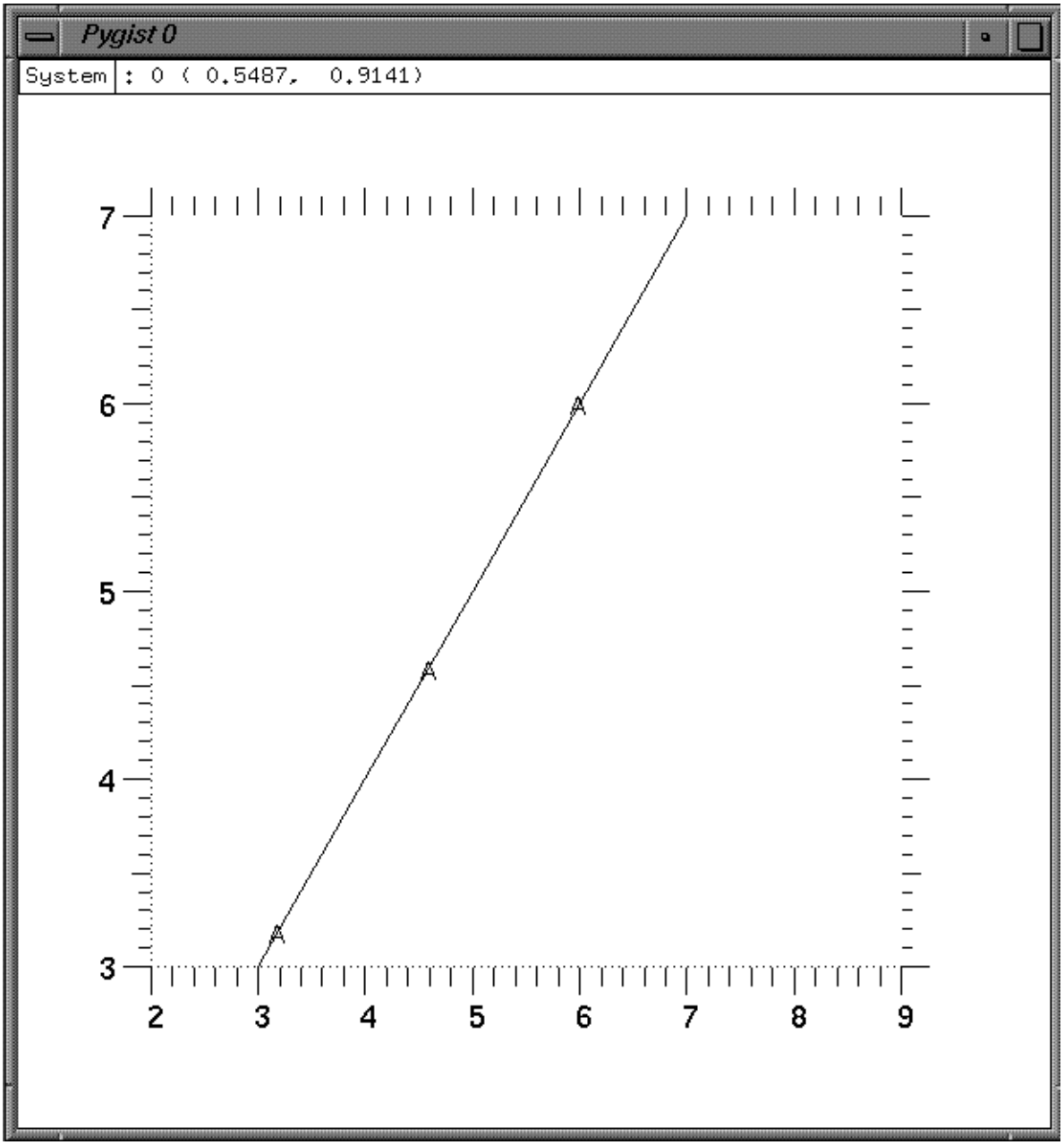
In the first example, the frame limits are set to the specified values. In the second call, the extreme values for *xmin* and *ymin* are used. Hence, the frame limits are 1,5,1,9.

```
ezcshow ("true")
plot (arange (10), arange(10))
# (plot on next page)
```

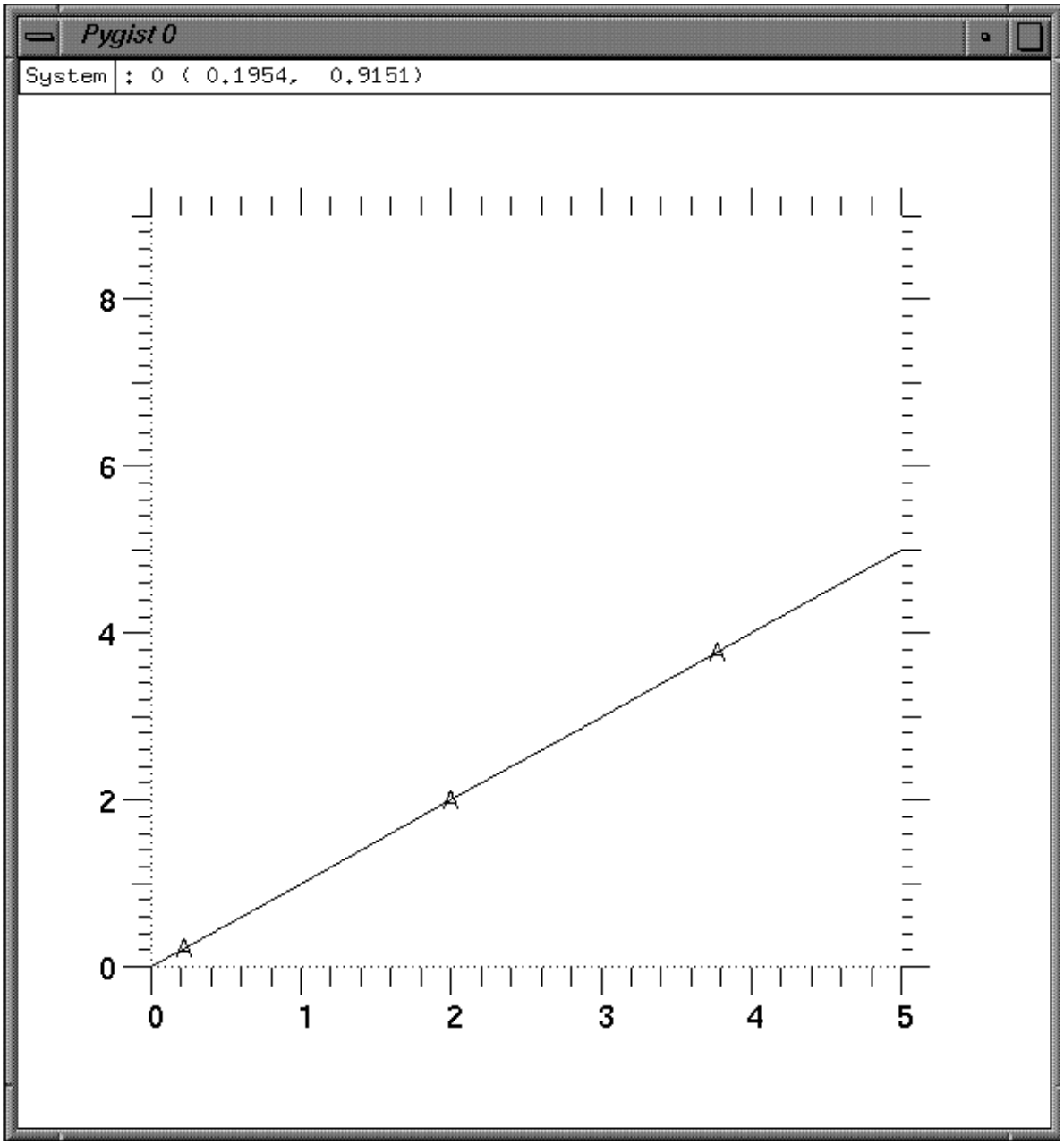
1. EZN (i. e., Basis) allows arbitrary arguments to be omitted, e. g.,
   `frame ,,ymin,ymax`
   but this form is not allowed in Python.


Pygist 0
System : 0 (0.3720, 0.9160)
0
2
4
6
8
0
2
4
6
8

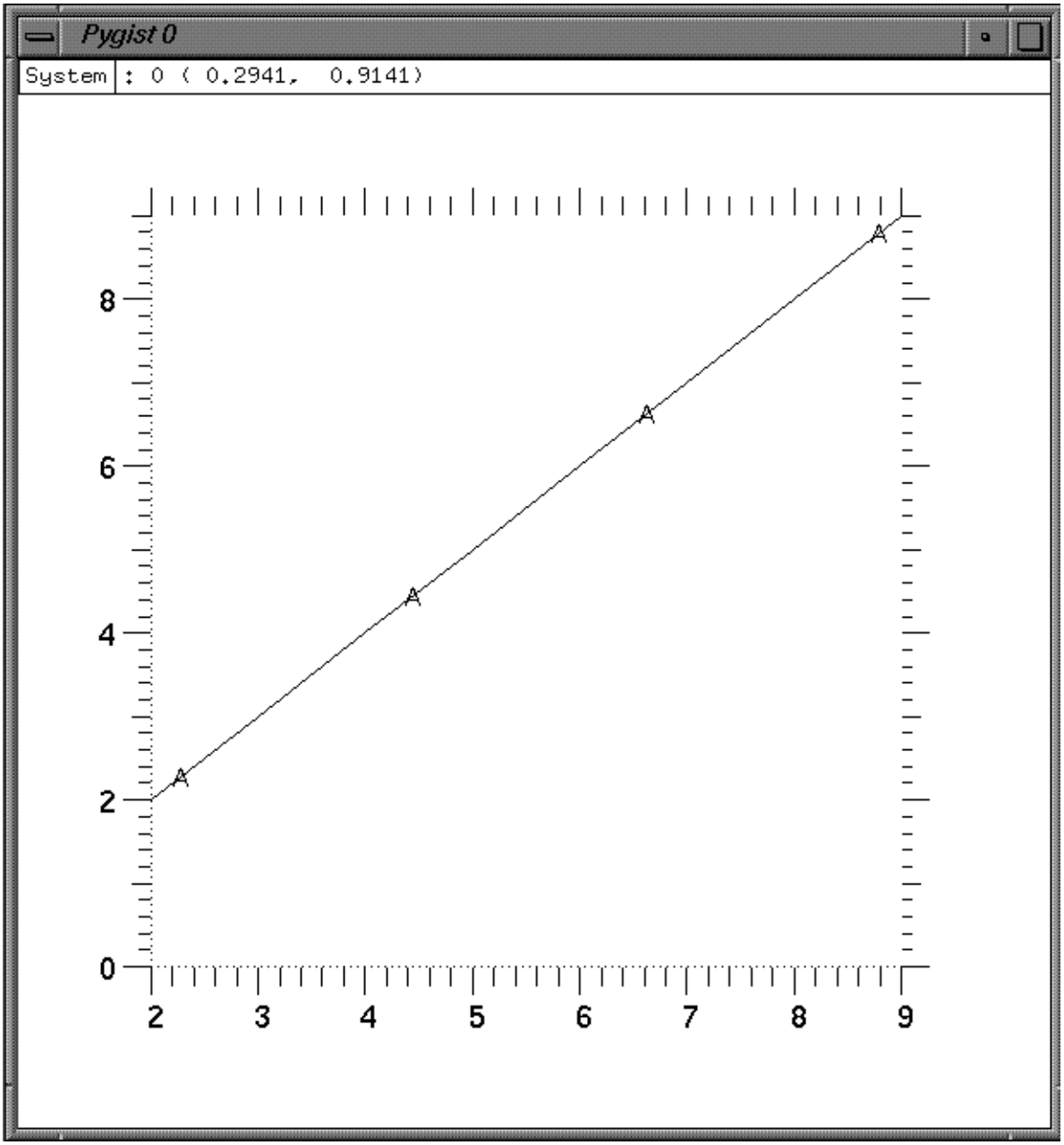
```
frame (2, 9, 3, 7)
```



```
frame (xmax = 5, ymax = 9)   # xmin,ymin defaulted
```



```
frame (2, 9)                    # ymin,ymax defaulted
```



Since `ezcshow` is `true`, four frames were displayed, as illustrated on the preceding pages. If `ezcshow` had been set `false`, only three frames would be displayed. The moral is: put the `frame` command first, normally, and use subsequent `frame` commands to plot a different view of the same set of objects

3.4.2 nf: New Frame

Calling Sequence

`nf ([new_frame = val1] [, window = val2])`

Description

The `nf` function signals that a new frame is to be started. By default, attributes set by the `attr` function (See “attr: Setting Attributes” on page 43.) are reset to their default values when a new frame is issued. If, however, the user has issued `ezcreset ("false")`, then the attributes set by the `attr` command will remain in effect across frame advances.

What `nf` really does is to close the currently displayed frame. What else happens depends on the value of the keyword parameter `new_frame` and also depends on the last call to `ezcshow`. If `new_frame` is `"yes"` (the default), then the current frame remains displayed, but EZPLOT clears that frame’s current display list in preparation for the next one. If `new_frame` is `"no"`, then the `nf` behaves exactly like `sf`, described in the next subsection; i. e., it redisplays the current display list but does not clear it. The most recent `ezcshow` call determines what happens the next time a plot function or an `sf` is invoked:"min"

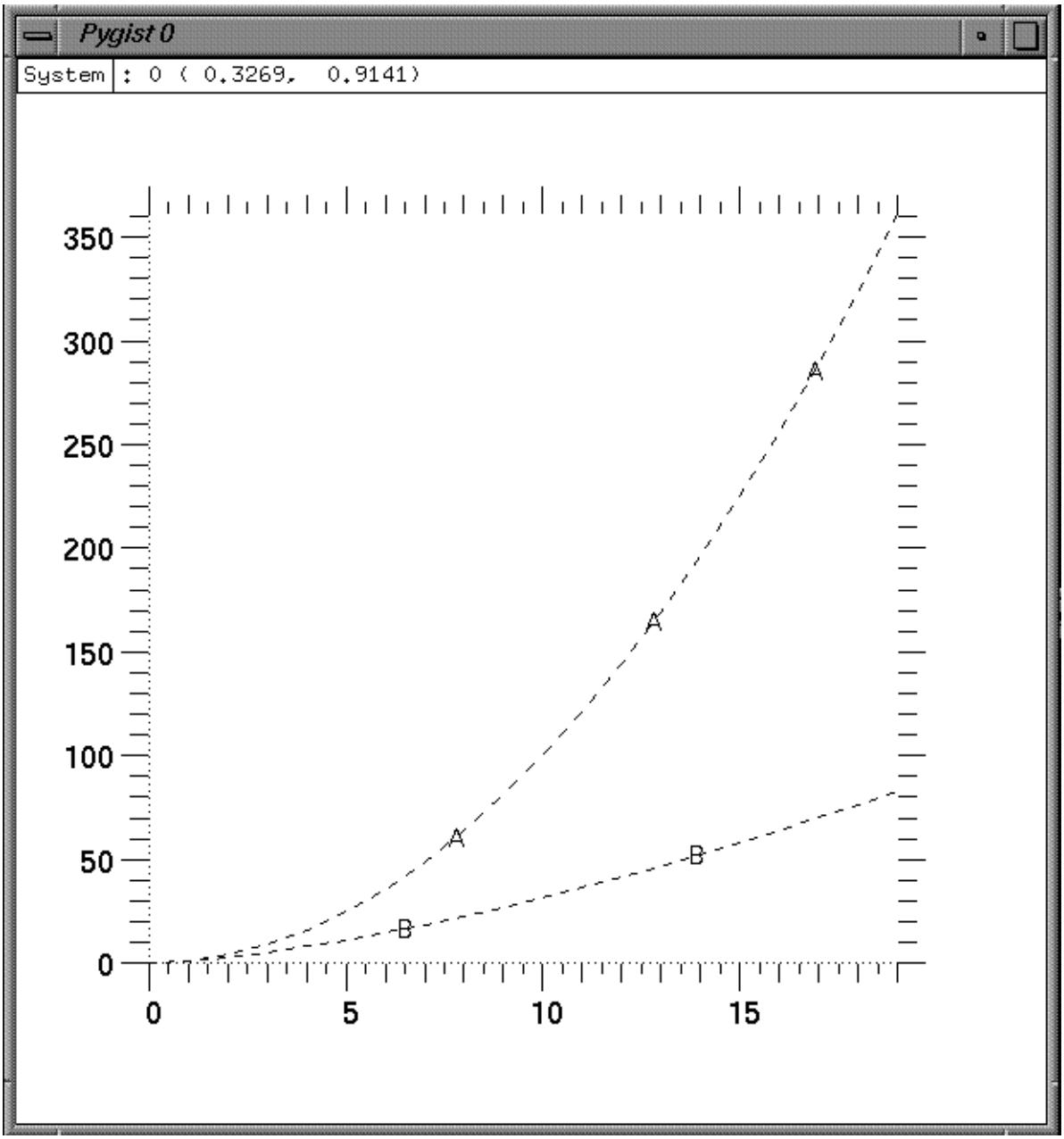
- If it was `ezcshow ("true")`, and a plot command is issued, then the specified curve will be added to the display list and the list will be displayed immediately. If an `sf ()` is issued, it has no effect unless `nf` was called with `new_frame` equal to `"yes"`, in which case it will tell you that there is nothing to graph.
- If it was `ezcshow ("false")`, and a plot command is issued, then the specified curve will be added to the display list, but the current display will not change until an `sf ()` is issued.

The `window` keyword argument can be used to specify a particular window or file; the allowed values for *`val2`* are `"all"` (the default), `"cgm"`, `"ps"`, `"min"`, `"max"`, an integer from 0 to 7, or a list of such integers.

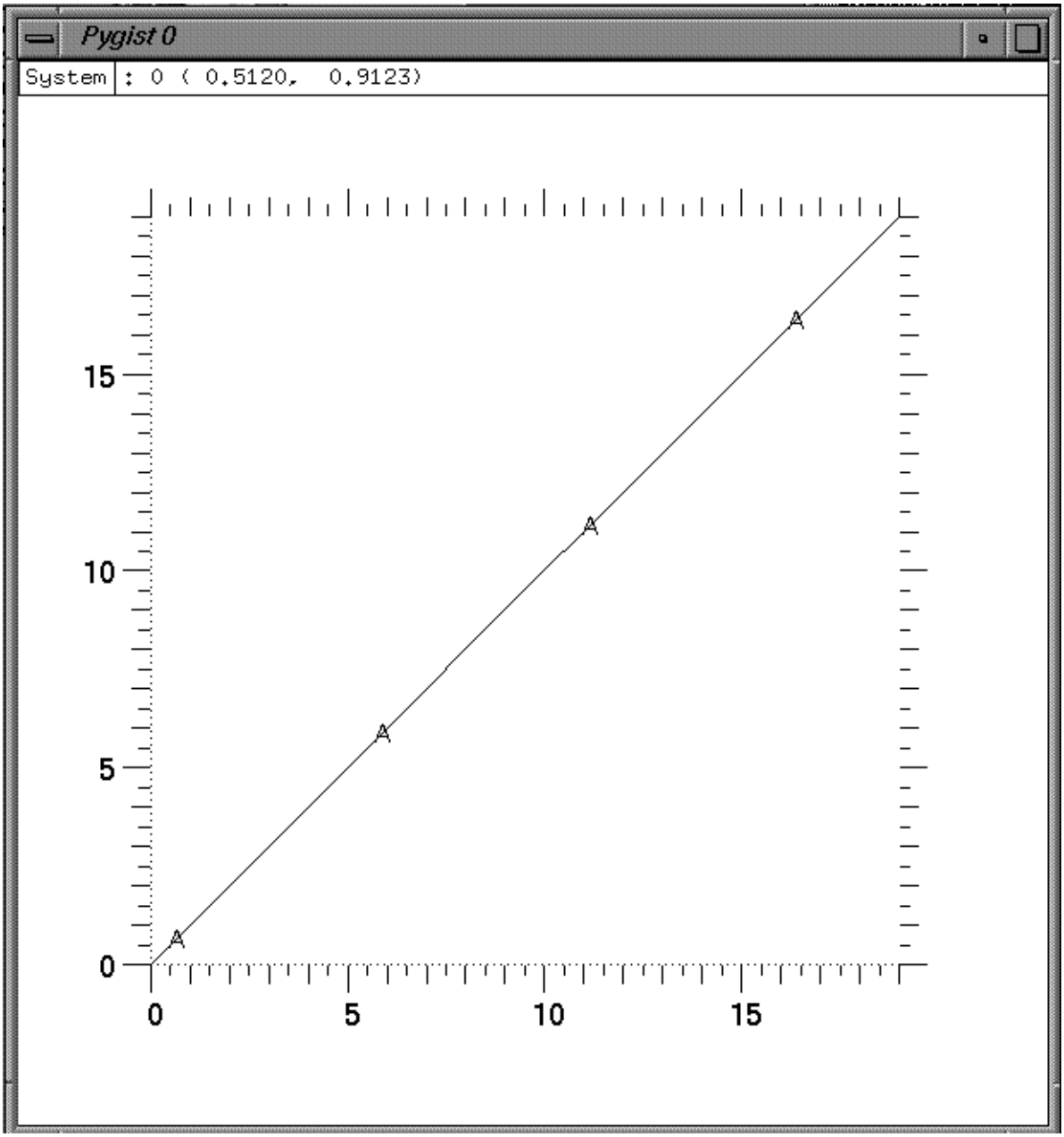
Example 4

In the default case, the line style is reset across frame advances. (Note, though, that PyNarcisse does not support dashed curves, so this example won't illustrate much of anything if used with PyNarcisse.)

```
ezcreset ("true")      # (default)
attr (style = "dashed")
plot (y, x)            # First plot dashed.
plot (y2, x2)          # Second plot dashed.
```

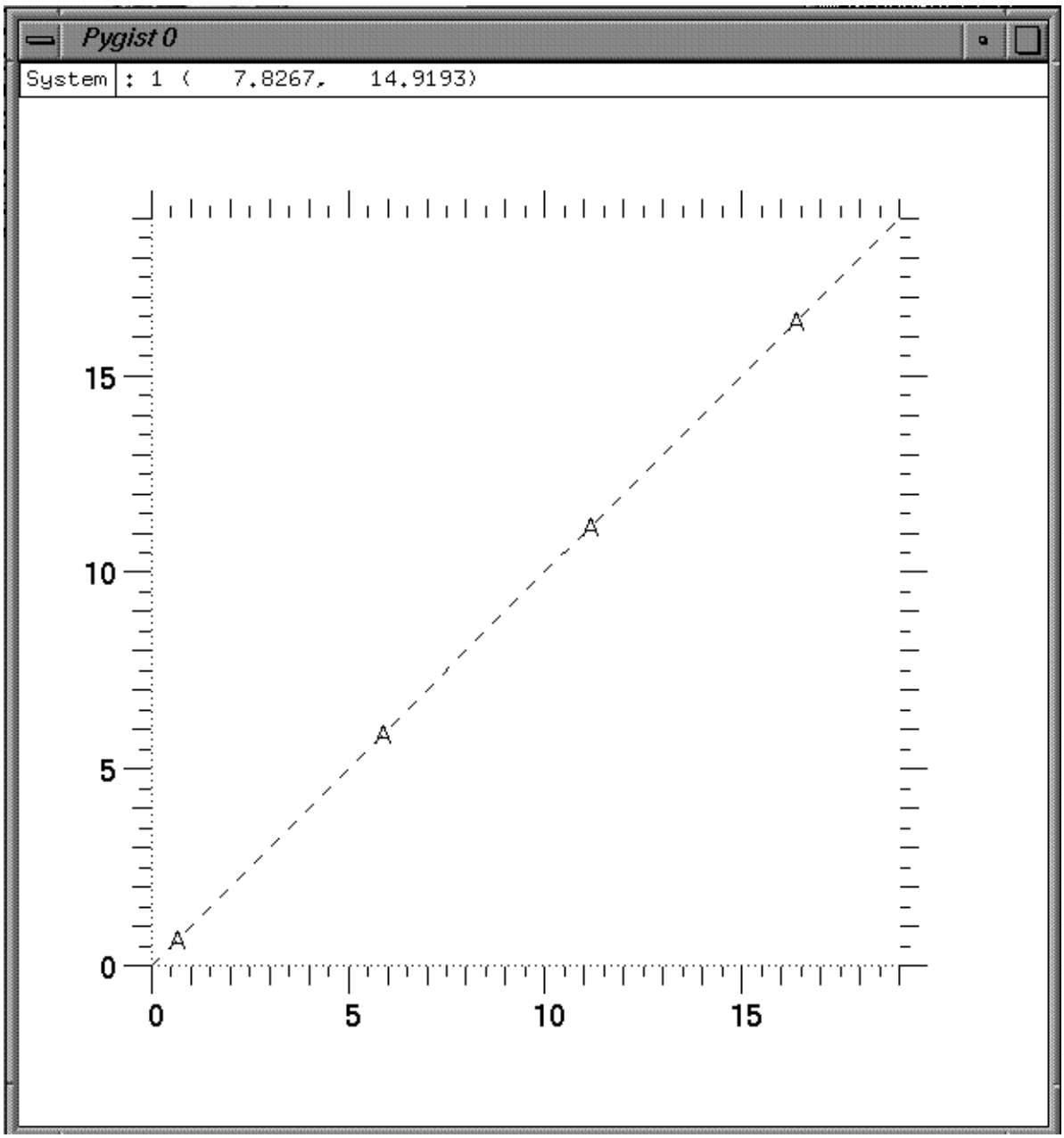


```
nf ()
plot (y3, x3)          # Style IS reset to solid (default).
```



In the example below, the line style remains dashed across frame advances.

```
ezcreset ("false")
attr (style="dashed")
nf ()
plot (y,x)            # First plot dashed
plot (y2,x2)          # Second plot dashed
# (same plot showing as two pages back)
nf ()
plot (y3,x3)          # Style NOT reset across frame advance
```



A better way to do this is usually to change the default variables, in this case `defstyle`. See Chapter 8 for details.

3.4.3 `sf`: Show Frame

Calling Sequence

`sf ([window = val])`

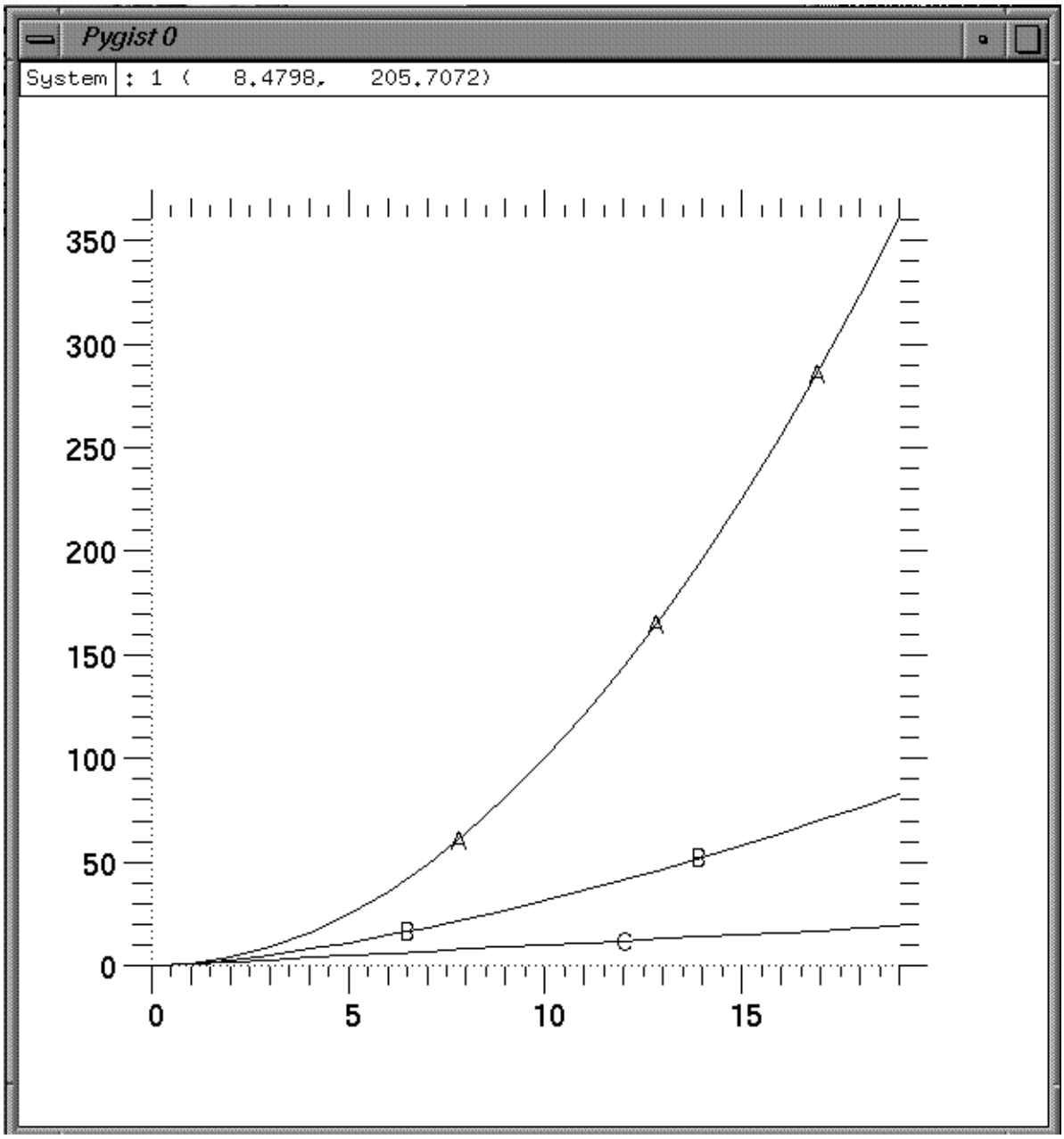
Description

The `sf` function displays the current frame to all active devices, or to particular window(s) or file(s), depending on *`val`* (`"all"` being the default). The frame is displayed regardless of the value of the status of `ezcshow`. This function is useful when a user wants to control the display of the frame at certain times; i.e., not every time a graphic object is added on a frame (default). Note that `sf` will complain if there is nothing on a selected display list, which will be the case any time an `nf ()` is issued with `new_frame` equal to `"yes"`, or missing altogether.

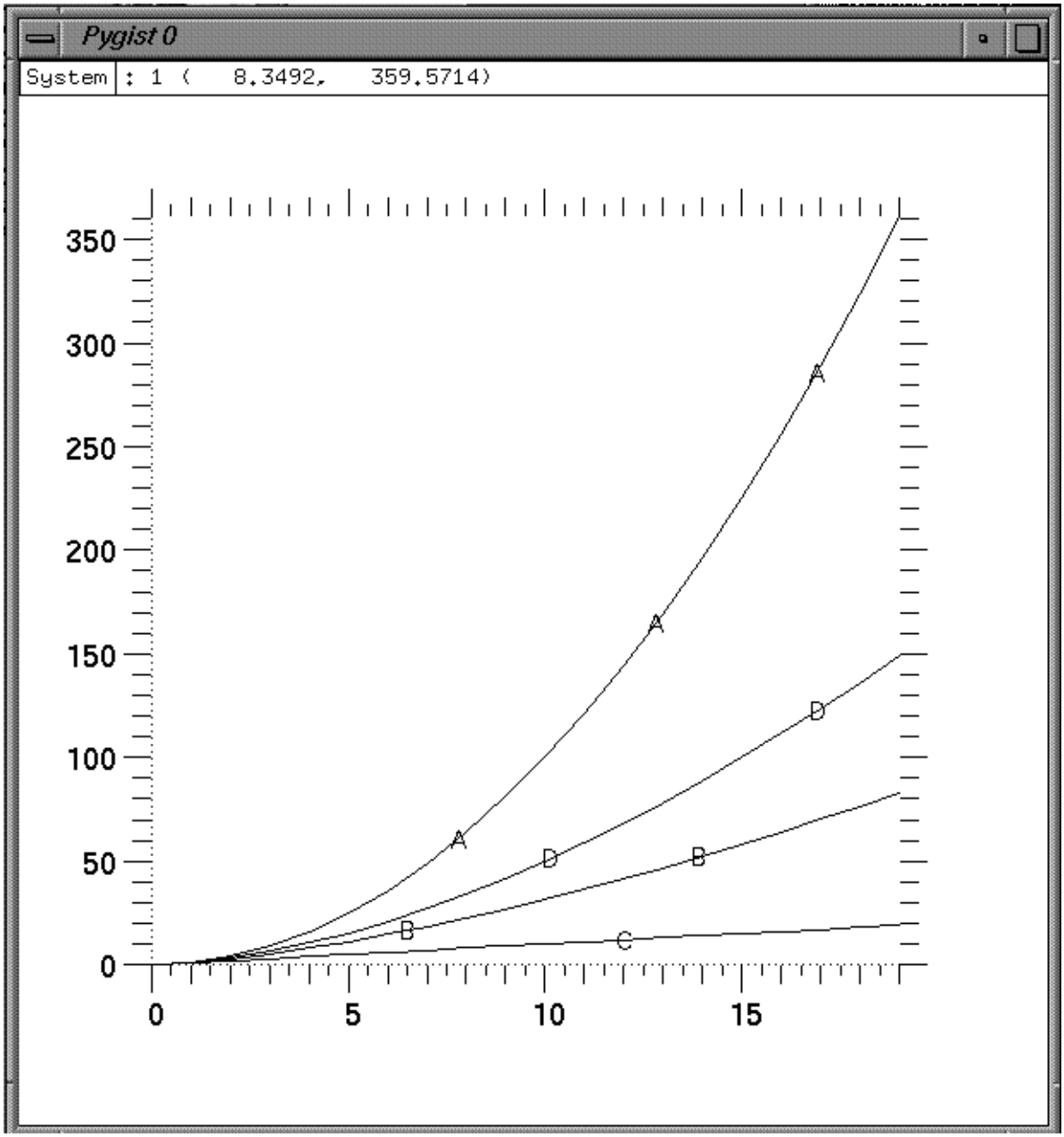
Example 5

In the example below, the `sf` function is used to display the frame after three curves have been added. Note that function `ezcshow` was called with argument `"false"`, so there will be no change in the display until the `sf`. A fourth curve can then be added; had `nf ()` been used instead of `sf ()`, the first three curves would no longer be in the picture.

```
ezcshow ("false")
plot (y1, x1)
plot (y2, x2)
plot (y3, x3)
sf ()          # force show of current frame (next page):
```


Pygist 0
System : 1 (8.4798, 205.7072)
350
300
250
200
150
100
50
0
0
5
10
15
A
A
A
B
B
C

```
plot (y4, x4)
sf ()
```


Pygist 0
System : 1 (8.3492, 359.5714)
350
300
250
200
150
100
50
0
0
5
10
15
A
A
D
D
B
B
C

3.5 undo: Undo a Plot Command

Calling Sequence

```
undo ([item = number] [, window = val])
```

Description

Remove the *number*th object in the EZPLOT display list for the device(s) specified by the `window` keyword.. (EZPLOT maintains a list of graphic objects created by successive plot calls. This list is cleared when an `nf` is issued.) If no `item` is given, `undo` the last graphic object. Some EZPLOT functions do not generate graphic objects in the display list (for example, the `frame` function), so *cannot be undone* in this way. It is the user's responsibility to figure out which number should be supplied for `undo`. (Numbering of objects begins at 1.)

The `window` keyword can be `"min"`, `"max"`, or a number between 0 and 7. It cannot be a list or `"all"`. The default value, if this argument is missing, is `"min"`. An exception will be raised if the item number referred to does not exist, or if the specified window does not have a nonempty diaplay list.

CHAPTER 4: Attributes

A set of “attributes” such as `color`, `thickness`, `scale`, `marks`, `labels`, etc., can be used to control the appearance of graphic objects or the layout of a frame.

4.1 Attribute Types

Some attributes affect the entire picture (such as scale, frame limits) while others affect the individual graphic objects in the picture (such as thickness, color).

If the attribute affects the entire picture, it will take effect immediately and we call it a *frame* attribute. If the attribute only affects the individual graphic object, we call it an *object* attribute. A special kind of object attribute (for mesh plots), which affects the current object and remains in effect until a frame advance or until another assignment is made to the attribute, is called "*sticky*". See “Attribute Table” on page 48. for a list of valid keywords, values and their attribute types.

The `grid` and `scale` attributes are examples of *frame* attributes. These attributes affect the entire picture. When these attributes are specified with the `attr` function or in an EZPLOT graphic function, a new picture is plotted with the grid and scale changed. (Note: This has the side effect of creating a new frame even if `ezcshow` (`"false"`) is in effect.)

The `color` and `style` attributes are examples of *object* attributes. If these attributes are specified in a graphics function call, the color and line style are changed only for the objects generated by this command. If these attributes are specified with the `attr` command, only those objects added to the frame following the `attr` command will have these specified attributes. Some special attributes for the mesh plots such as `region`, `krange`, `lrange` are ‘‘*sticky*’’: i.e., the specifications of `region`, `krange` and/or `lrange` will affect the following mesh plots until the end of the frame or the values have been redefined.

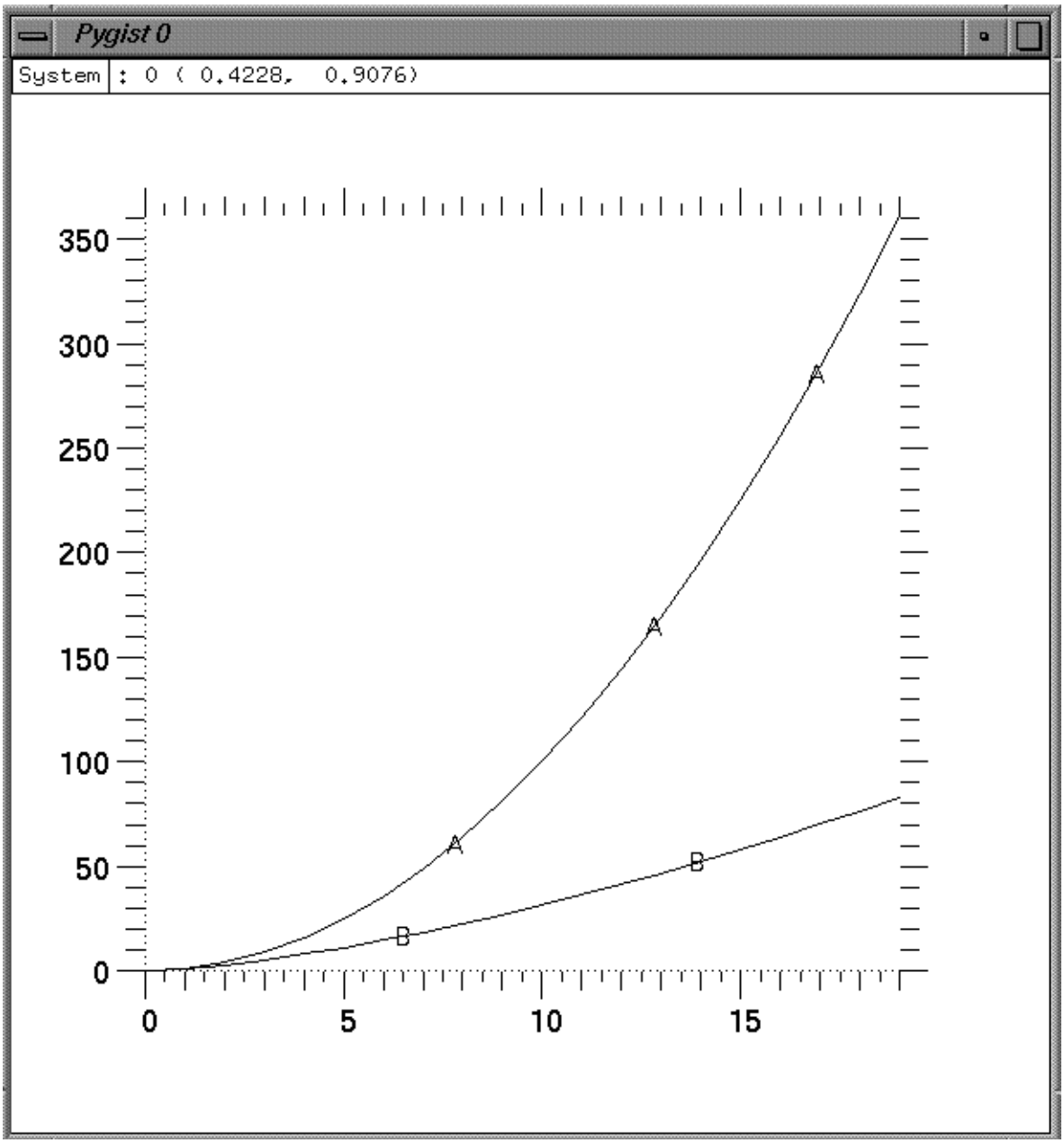
If no attribute value is set explicitly by the user, a default value will be used for the attribute. These default values in turn can be changed by setting certain control variables. User specified default values will be in effect until new default values are assigned. See CHAPTER 8: “Control Variables and Defaults” for details.

By specifying attributes and control variables, it is also possible to change many things about the layout of the picture, such as the size of the titles, and the minimum size of the text.

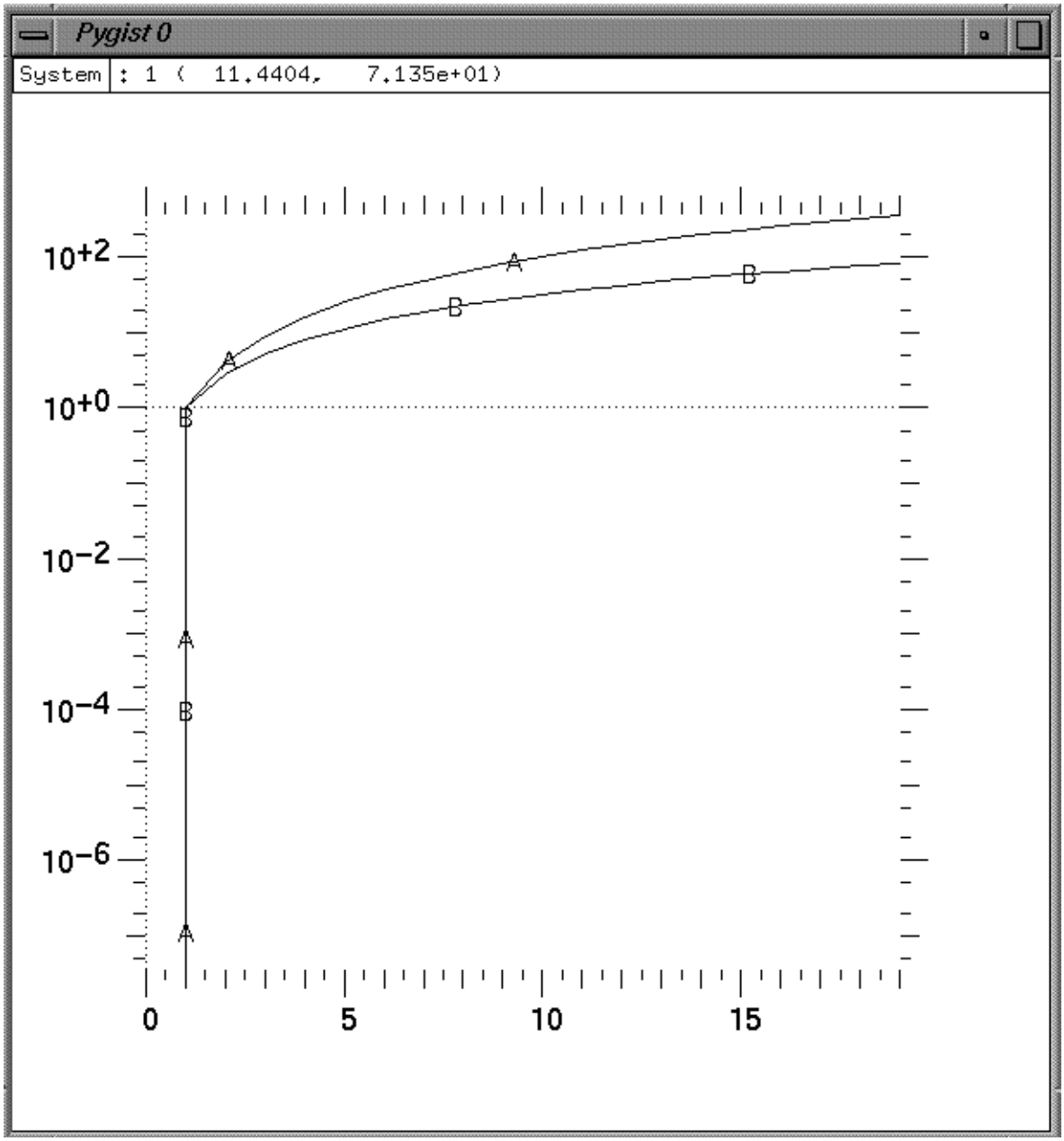
Usually all attributes will be re-initialized to their default values when a frame is advanced. However, calling the function `ezcreset` with argument `"false"` will cause the attribute settings to last across frames.

Examples

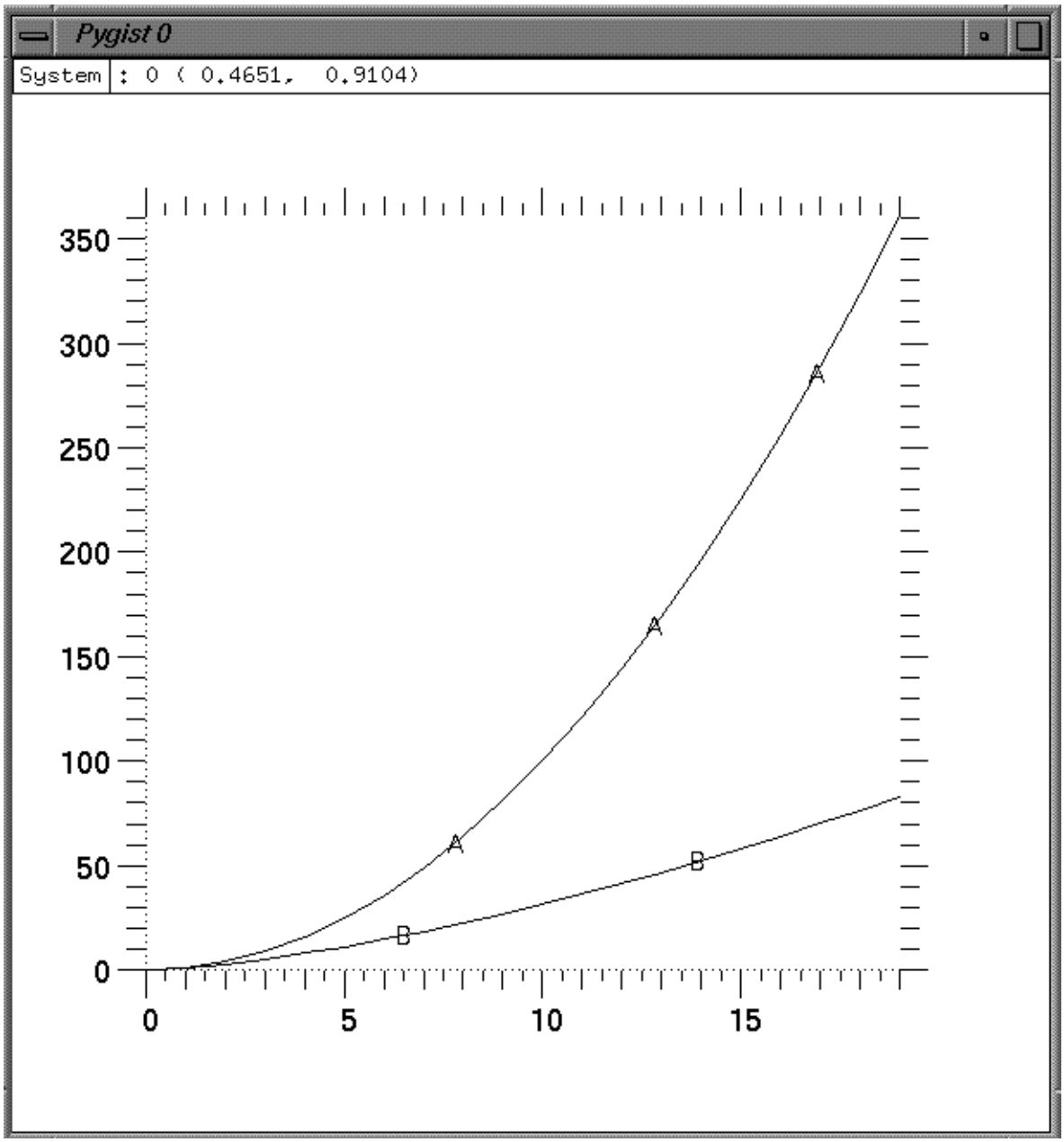
```
plot (y1, x1)
plot (y2, x2)
```



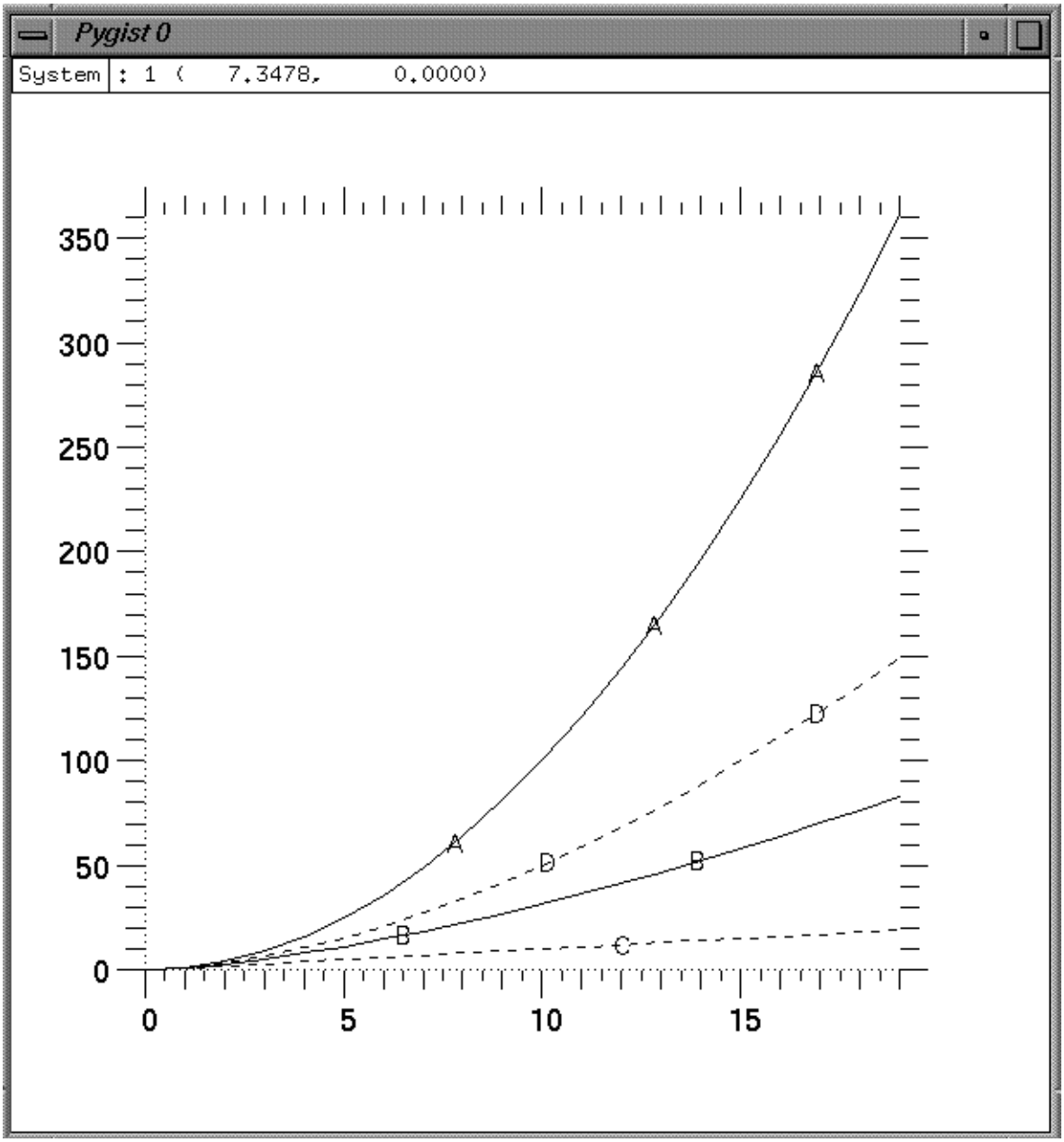
```
attr (scale = "linlog")  # Picture redisplayed.
```



```
nf ()
plot (y1, x1)
plot (y2, x2)
attr (style = "dashed")  # Only following curves affected;
                         #        no redisplay yet.
```

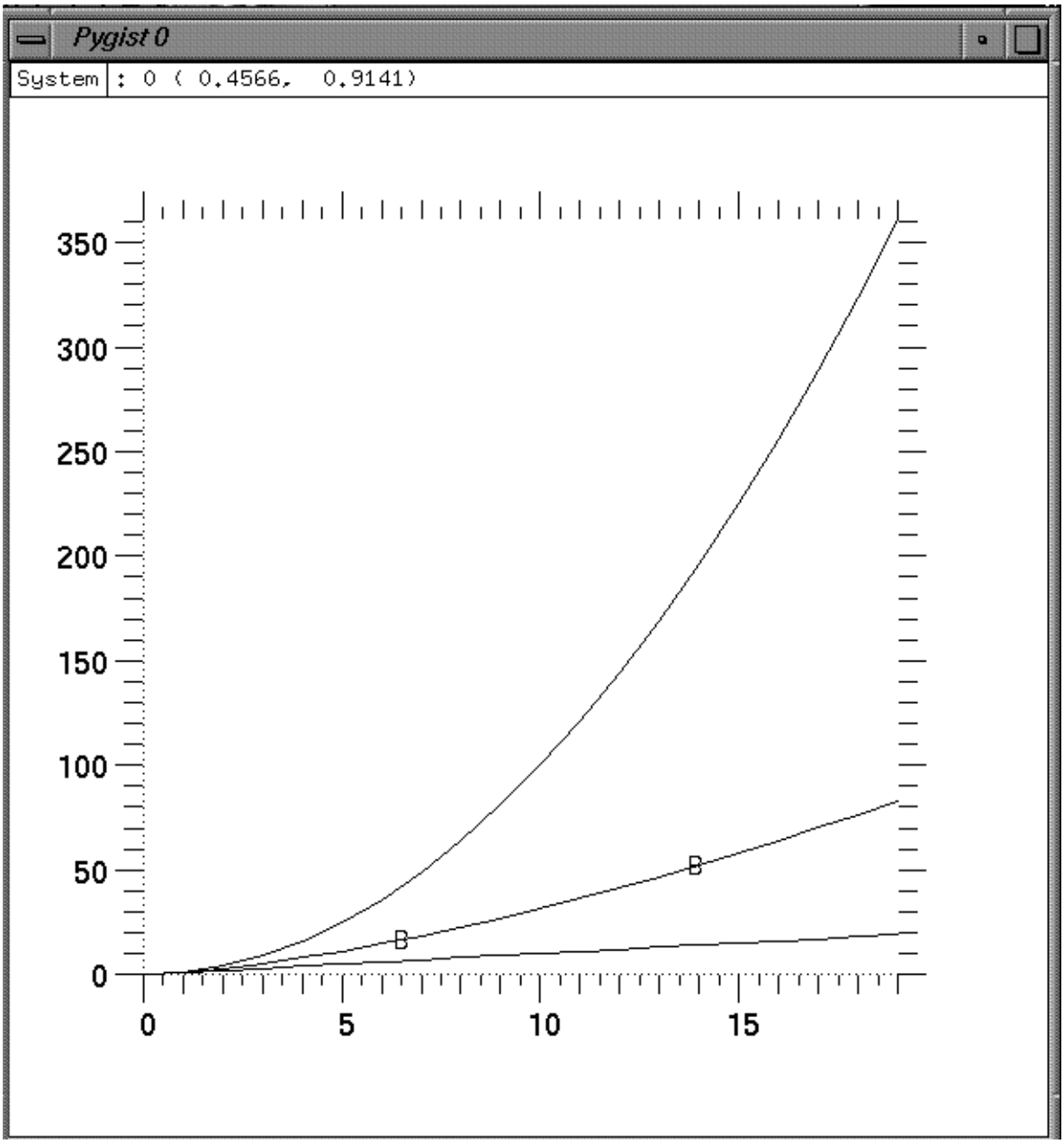


```
plot (y3, x3)
plot (y4, x4)
nf ()
```



The attributes `labels` and `lev` can be either *frame* or *object* attributes. For example, `attr` can set `labels` to `"yes"` or `"no"`, to indicate whether or not subsequent curves in the frame are to be labeled. As an *object* attribute, `labels` can also be set to the opposite value, as in:

```
attr (labels = "no")
plot (y1, x1)
plot (y2, x2, labels = "yes")
plot (y3, x3)
```



This results in the curves `y1` and `y3` being unlabeled, and `y2` being labeled (see above); i. e., subse-

quent curves in the frame will be unlabeled unless the object attribute `labels` is explicitly set to `"yes"`.

Labels for the curves (other than the default letters) can be specified with the `labels` keyword. Labels must be quoted strings, or variables or expressions (including arrays) whose values are quoted strings. The attribute `labels` is also used to turn labelling on and off (by setting it to `"yes"` or `"no"`). When the attribute `labels` is used in this sense, it is a *frame* attribute. i.e., all existing and subsequent curves on the frame will be either labelled or not.

The attribute `lev` can be used to assign the number of contour levels or a vector of contour level values as an object attribute. When `lev="log"`, it becomes a *frame* attribute, and it sets the contour levels based on a logarithmic scale.

4.2 `attr`: Setting Attributes

Calling Sequence

```
attr (keyword1=value1, keyword2=value2, ... ,
     keywordN=valueN)
```

Description

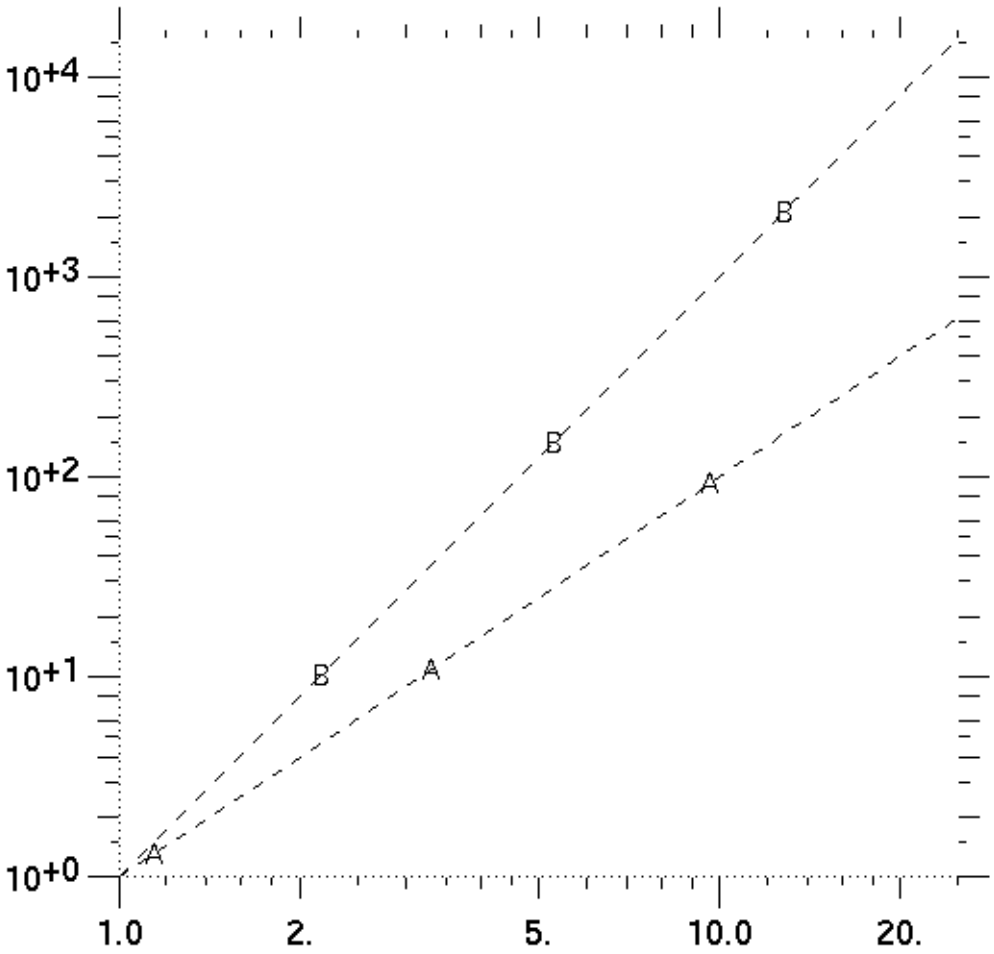
The `attr` function assigns values to attributes. The value assigned to an attribute remains in effect until a frame advance is issued, or until another assignment is made to the attribute via the `attr` command (within the same frame). An attribute's effect can be reversed for an object in the frame if it can also be used as an object attribute, as noted in the previous section. To make the values assigned to attributes remain in effect across frame advances, call function `ezcreset` with argument `"false"`.

To make a permanent change to a default, change the corresponding variable. See CHAPTER 8: "Control Variables and Defaults" for a list of these.

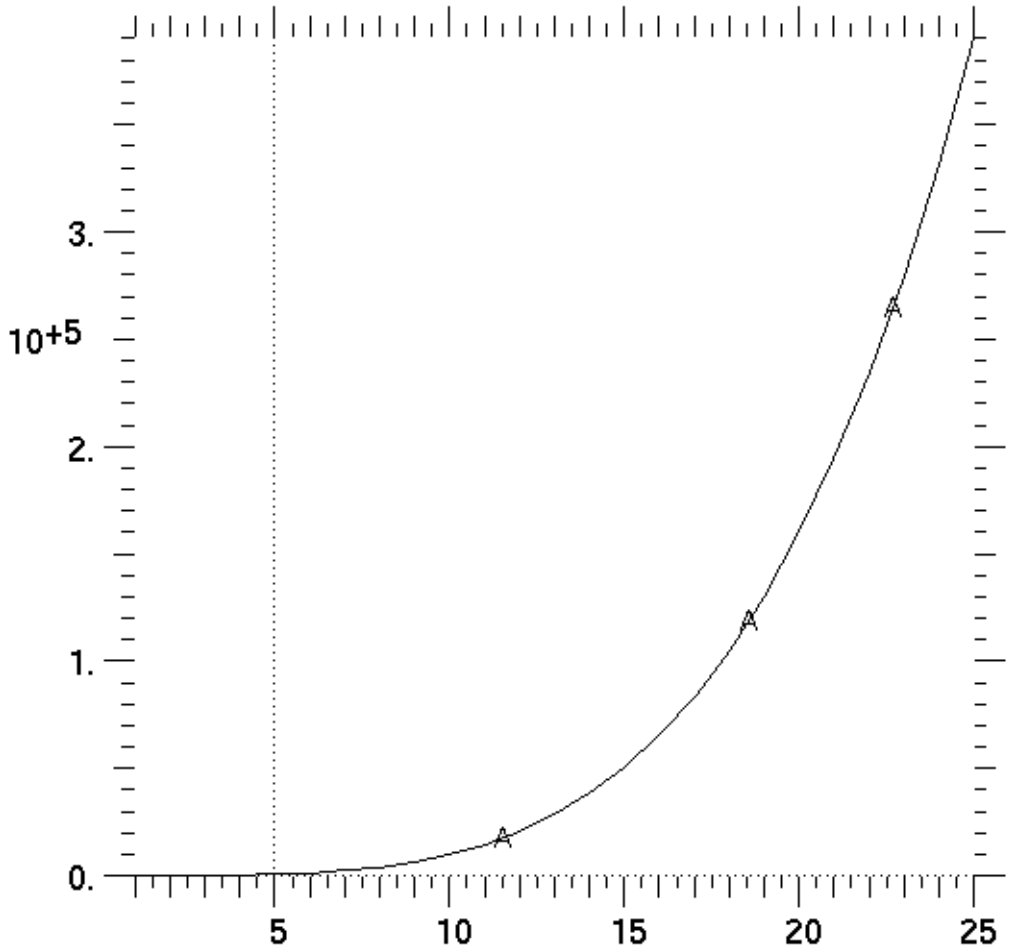
Examples

In the first example, the scale is set to `loglog`, the line style is set to `dashed`. Since the default value `"true"` was sent to `ezcreset`, the attributes set only remained in effect until the next frame advance. After that, the attributes were reset to their default values.

```
# assume ezcreset ("true") (default)
# Settings remain in effect only until next frame advance.
attr (scale = "loglog", style = "dashed")
plot (y1, x1)
plot (y2, x2)
```

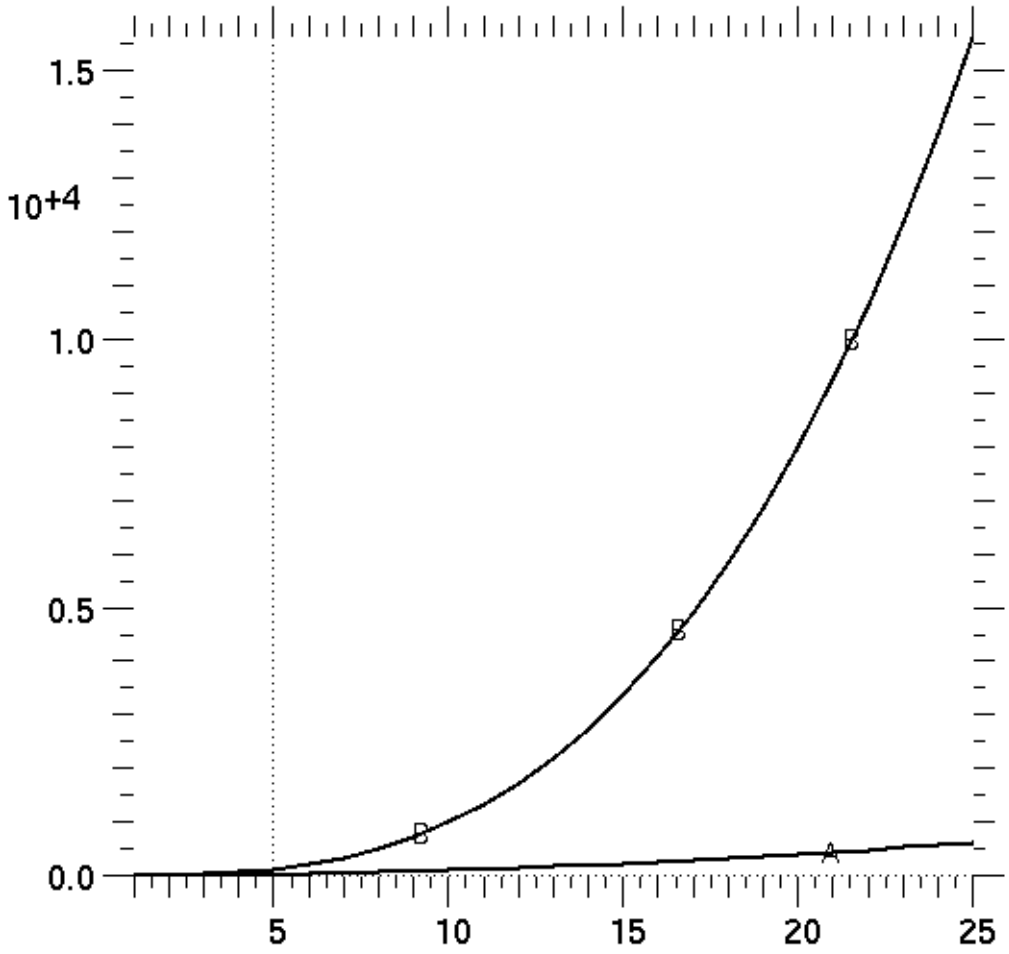


```
nf ()
plot (y3, x3)  # scale,style reset to defaults.
```

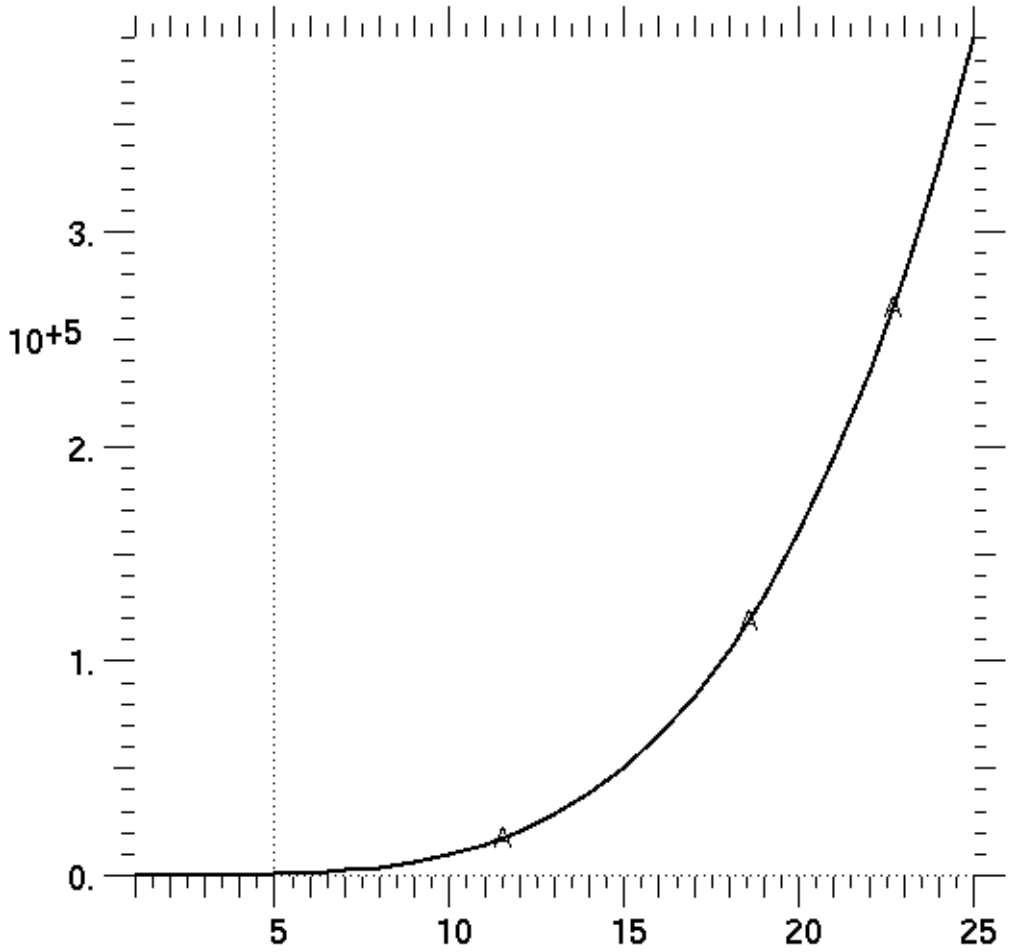


In the second example, `ezcreset` is called with argument `"false"`. This time the `attr` command remains in effect across frame advances. Hence, the line thickness remains set to 3.0 across frame advances.

```
ezcreset ("false")
# Settings remain in effect across frame advances.
attr (thick = 3.0)
plot (y1, x1)
plot (y2, x2)
```



```
nf ()
plot (y3, x3)  # Thickness still 3.0.
```



Or, we could accomplish the same thing more simply by making a permanent change to the default thickness (the plots will look identical to the previous two):

```
# ezcreset ("true") # (default)
defthick=3.0
plot (y1, x1)
plot (y2, x2)
nf ()
plot (y3, x3)  # Thickness still 3.0.
```

4.3 Attribute Table

The following is an alphabetical list of all allowable attribute keywords. Refer to individual plot commands for more specific information. Note that this is only a subset of what is available in the Basis EZN package. More attributes can be added to EZPLOT if there is sufficient interest.

TABLE 1. **`attr`: Attribute Table**

Keyword	Type	Value	Description
bnd	object	"no"	Plot full mesh (default).
		"yes"	Plot region boundaries only.
color	object	"bg" "fg"	The default background / fo ground color used by EZPL
		"*color*"	Use one of the following 16 colors (default="fg"): "red", "green", "blue", "cya "magenta", "yellow", "purpl "black", "white", "gold", "y lowgreen", "orangered", "re ange". (Not all colors are av on both Gist and Narcisse.)
		"filled"	Color fill the contour band, ing from blue to red.
		"fillnl"	"filled" without contour line
color_bar	frame	0 or 1 (default)	If 1, place a color bar at the side of a colored contour or contour plot
cscale	frame	"lin"	Linear color mapping with
		"log"	Logarithmic color mapping `plotf`
		"normal"	Normal color mapping with `plotf`
grid	frame	"no"	No reference grid
		"tickonly"	Tick marks only (default)
		"x"	x rulings
		" y"	y rulings
		"xy"	x and y rulings
krange	sticky	(*kmin,kmax,kinc)*	Range for k-lines in mesh p (default=(1,kmax,1))
kstyle	object	"none"	No lines in k direction.

TABLE 1. **`attr`: Attribute Table (Continued)**

Keyword	Type	Value	Description
		"*style"*	Use *style* for k-lines. (*See* st default="solid")
labels	frame	"yes"	Curves/marks are labelled i order added (default).
		"no"	No labels displayed, unless ruled by object label spec (b
label	frame	*"str"*	Label all subsequent curves frame with *str*, unless overru object label spec (below).
	object	*"str"*	Label next curve with *str*. *str* can be a vector for multi curves.
lev	object	*ival*	Number of contour levels. (default=8)
		[$rval_1$,$rval_2$,...]	Vector of contour levels.
	frame	"lin" or "linear"	Linear contour levels (defau
		"log"	Logarithmic contour levels.
lrange	sticky	*(lmin,lmax,linc)*	Range for l-lines in mesh pl (default=(1,lmax,1))
lstyle	object	"none"	No lines in l direction.
		"*style"*	Use *style* for l-lines. (*See* sty default=solid)
mark	object	"asterisk"	Asterisk marker
		"circle"	Circle marker
		"cross"	Cross marker
		"dot"	Dot marker
		"plus"	Plus marker
region	sticky	"all"	Display all regions in mesh (default).
		[$ival_1$,$ival_2$,...]	Vector of desired region nur
scale	frame	"linlin"	Both x and y axes linear (de
		"linlog"	x-axis linear, y-axis logarith
		"loglin"	x-axis logarithmic, y-axis li
		"loglog"	Both x and y axes logarithm
		"equal"	Both x and y axes linear, scales equalized

TABLE 1. **`attr`: Attribute Table (Continued)**

Keyword	Type	Value	Description
style	object	"solid"	Solid lines (default)
		"dashed"	Dashed lines
		"dotted"	Dotted lines
		"dotdash"	Dot-dashed lines
		"none"	Background color lines
thick	object	*rval*	Line thickness multiplier (default=1.)
vsc	object	*rval*	Vector scaling factor (default=0.05).

CHAPTER 5: General Plot Commands

This chapter describes the EZPLOT general-purpose plot commands.

5.1 `plot`: Plotting Curves and Markers

Calling Sequence

```
plot ([yexpr[, xexpr[ ,<keylist>]])
```

Description

The **`plot`** command plots line segments connecting points or discrete markers at the points. Markers are plotted at the data points, without connecting line segments, when the attribute `mark` is set to one of the valid marker types. The default scaling factor for markers is 1.0, the default line style is `"solid"`, and the default line thickness is 1.0. To override these values, set the attributes `mark-size`[1], `style`, or `thick`, respectively.

If neither *yexpr* nor *xexpr* is specified, then the current picture is redisplayed. Otherwise, *yexpr* is an array of y-axis values, *xexpr* is an array of x-axis values, and <*keylist*> is a list of optional attributes specified by pairs of keywords and values separated by equal signs. If *xexpr* is not specified, then *yexpr* is plotted against the index of *yexpr*. If *yexpr* differs in length by one from the length of *xexpr*, whether explicitly or implicitly specified, the longer of the two will be automatically averaged to shorten it. If the lengths of *xexpr* and *yexpr* differ by more than 1, then the command is an error, no object is added to the frame, and an exception will be raised.

If the arguments are two-dimensional arrays, `plot` plots the corresponding columns of *yexpr* and *xexpr* to produce multiple curves at once. Multi-dimensional arguments are reduced to two-dimensional by collapsing any higher dimensions. If *xexpr* is one-dimensional, then each column of *yexpr* is plotted against it.

Optional Attributes

The following optional attributes can be specified with this command. For object attributes, they are local to the command specified; i.e., they are not remembered across commands.

1. Not currently available in PyGraph.

window, grid, scale, style, thick, color, labels, label, mark

If optional attributes are given as keyword arguments to `plot`; they are specified in the usual form:

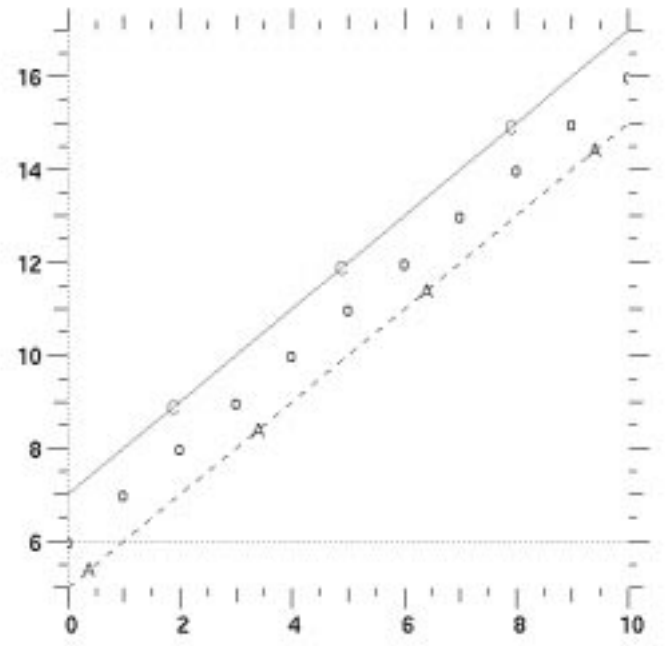
key1=value1,key2=value2,...,keyN=valueN

To set an object attribute across commands use the **attr** command. See “Attribute Table” on page 48. for descriptions of the values which can be assigned to these keywords.

Examples

In this example, three curves will be superimposed. The first `plot` function will plot a curve with dashed lines, the second `plot` function will mark circles, and the third `plot` command will plot the curve in red. Since the first plot command does not specify *xexpr*, y will be plotted against an array spanning from 0 to 10. In the second and third `plot` calls, the y values are also plotted against this same array. The curves are labelled "A" and "C" respectively. (“Marked” plots are not labelled.)

```
y = arange (5, 16, typecode = Float)
plot (y, style = "dashed")        # plot curve
plot (y + 1,mark = "circle")      # plot markers
plot (y + 2, color = "red")       # plot curve in red
nf ()
```

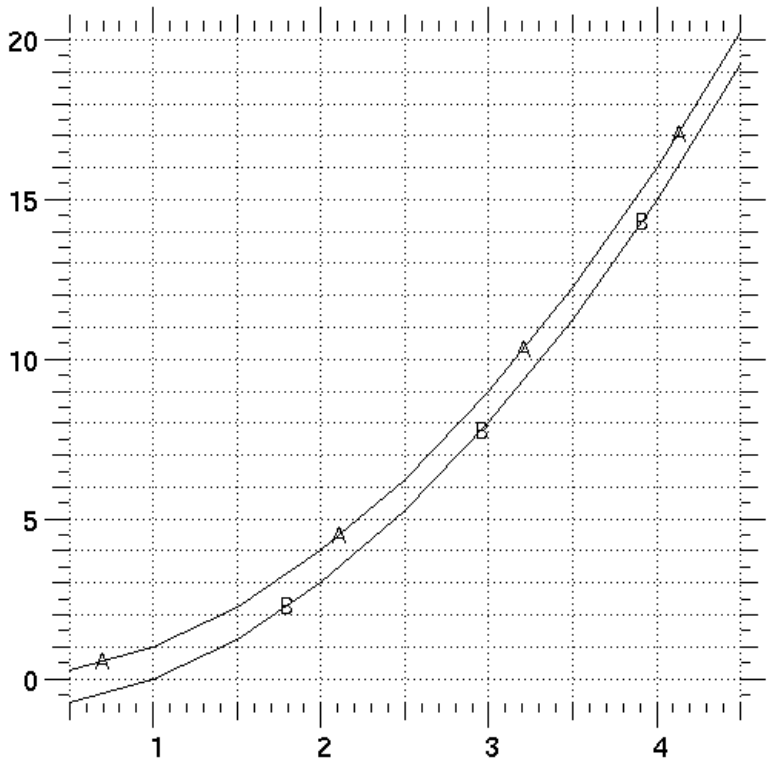


If you enter the above commands at the terminal, you will see three frames displayed in turn as the graphic objects are added. The `nf` call will clear the display list but not the screen. If you close the window (issue `win ("close")`), reopen it (`win ("open")`), and then you repeat the experiment but issue `ezcshow ("false")` first, you will not see any graphic objects at all, or indeed any win-

dow in the case of Gist, until you issue an `nf ("no")` or `sf ()` call, at which point the window with the completed frame will appear. Just issuing a naked `nf ()` will cause the window to open with nothing displayed, since the display list will have been erased.

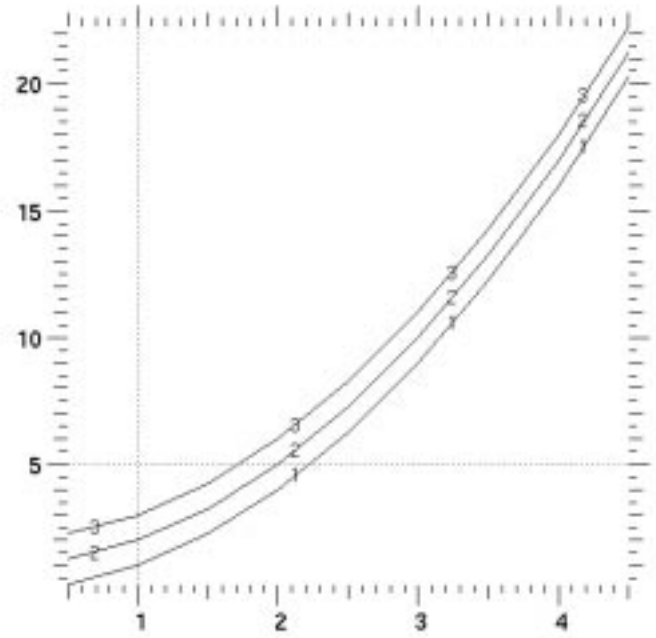
The next example replots two curves with an xy-grid added.

```
x = 0.5 * arange (1, 10, typecode = Float)
y=x**2
plot (y, x)
plot (y - 1, x)
plot (grid = "xy")
nf ()
```

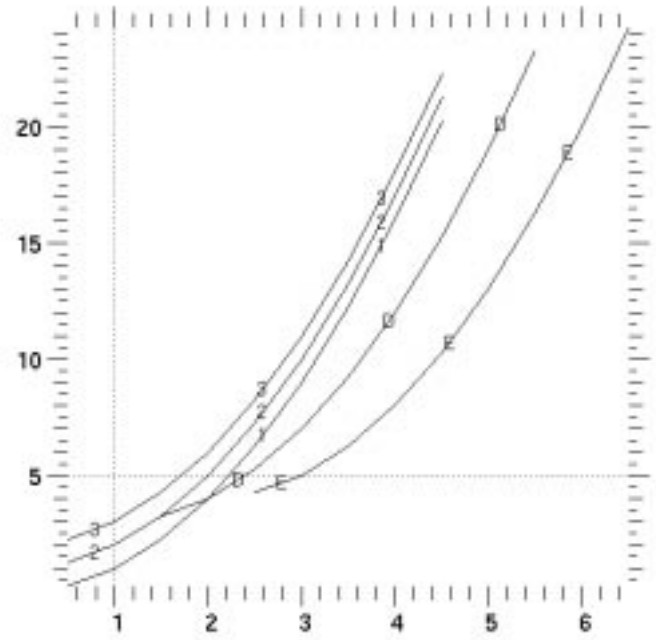


In the next example, the first `plot` call will plot three curves, `y`, `y + 1`, `y + 2` against the same x, labelled "`1`", "`2`", and "`3`", respectively. The second `plot` call will plot two curves, `y + 3` against `x + 1` and `y + 4` against `x + 2`, *perhaps* labelled ''`D`'' and ''`E`'', respectively. (Note that the default curve labels continue to increment even if the letter is not the curve label. Note also the caveat ''perhaps''. This is because the letters will be the next two available in this particular Gist session; they will be ''D'' and ''E'' only if this is the first plot you make.)

```
plot ( [y, y + 1, y + 2], x, label = ["1", "2", "3"])
```




```
plot ( [y + 3, y + 4], [x + 1, x + 2])
sf ()
```



Narcisse and Gist behave differently regarding labels. Gist will label a curve at several spots along

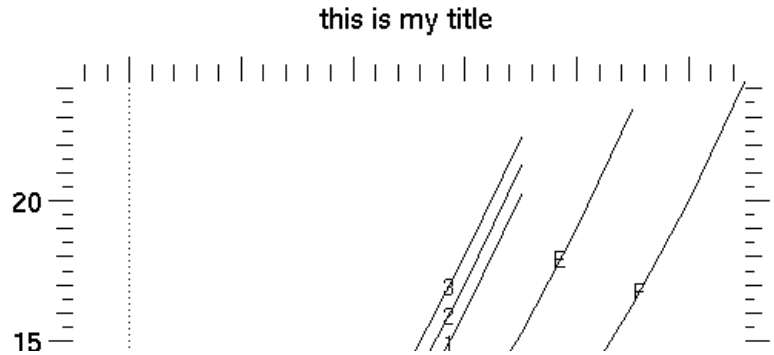
its length with a single character. If you specify a label of more than one character, it will simply use the first character. Otherwise it defaults to the capital letters, in order. Narcisse, on the other hand, allows multiple-character labels, but does not put the labels on the curves themselves, but rather at the right ends of the curves.

A difference between EZN and EZPLOT regarding labels is that EZN has only the keyword `labels`, which can be used to turn the labels attribute on and off, as well as to specify a label for a curve or a default label for the frame. This is confusing, especially because EZN allows multiple occurrences of the same keyword in a `plot` command. Python does not support multiple occurrences of a keyword, because keywords create Python dictionaruy entries, and there can only be one entry per keyword. We have solved this problem by using `labels` solely as the frame attribute which enables or disables labels for the entire frame, according as it is set to `"yes"` or `"no"` in an `attr` or `plot` call. The `label` keyword is then used in an `attr` call to set the default label for all curves in the frame (if labels are enabled for the frame), or in a `plot` call to specify the label(s) for one curve or a family of curves regardless of whether labels are enabled for the frame.

It is also important to realize the difference between marks and labels. Both graphics engines support plotting curves with marks at each point specified by the coordinate arrays. The `marks` attribute described in the previous chapter allows you to specify `"dot"`, `"circle"`, `"cross"`, `"plus"`, or `"asterisk"` to specify the mark to be used; marks, if used, forces the attribute `style` to be `"none"`. Narcisse supports only these five marks. Gist supports any single character mark. Narcisse plots only the points specified; Gist draws what is called a ‘‘polyline,’’ which is a curve densely populated with the specified character.

A major difference between EZPLOT and EZN is that EZPLOT does not support the keyword ‘‘`legend`’’. This is because EZPLOT does not put the text of the plotting commands on the graph the way EZN does. Instead, EZPLOT has a `titles` function, which allows the user to specify titles for the top, bottom, left, and right margins of the graph, and a `text` function (See Chapter 7), which allows text to be plotted at arbitrary places in the graph. To put a title at the top of the graph, do

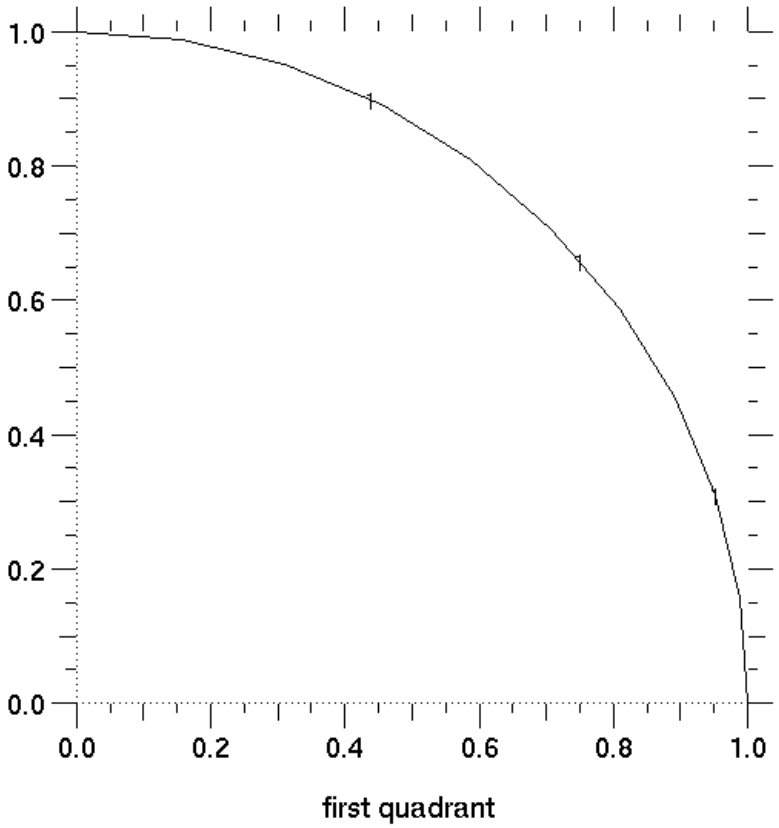
```
titles ("this is my title")
sf ()
```



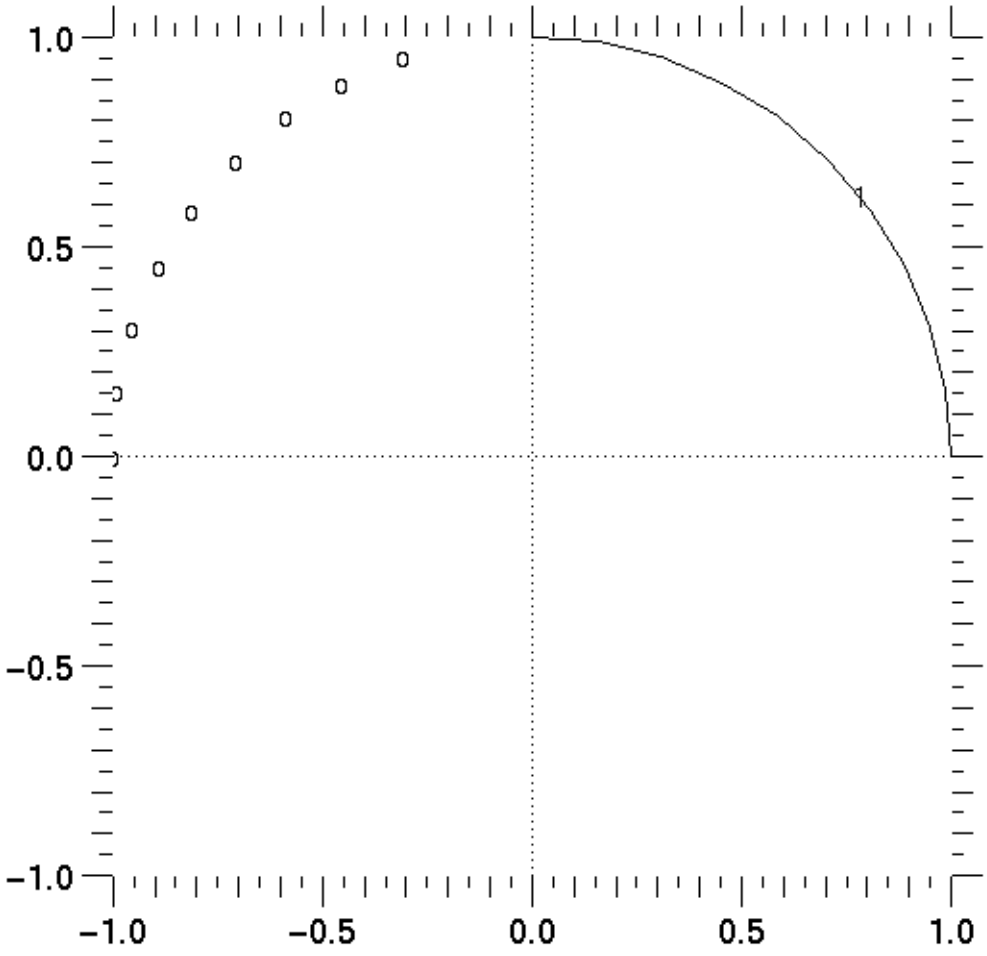
The arguments for `titles` must be given in the order top, bottom, left, right and may be omitted from the right. Omitted arguments default to blank. For convenience, EZPLOT also offers four functions, each of which sets just one of the titles (leaving the other three unchanged). These are `titlet`, `titleb`, `titler`, and `titlel`.

The fifth set of examples graphs the unit circle and x and y axes in a variety of styles and also illustrates how the `labels` attribute works. Comments in the code explain what happens on the frame.

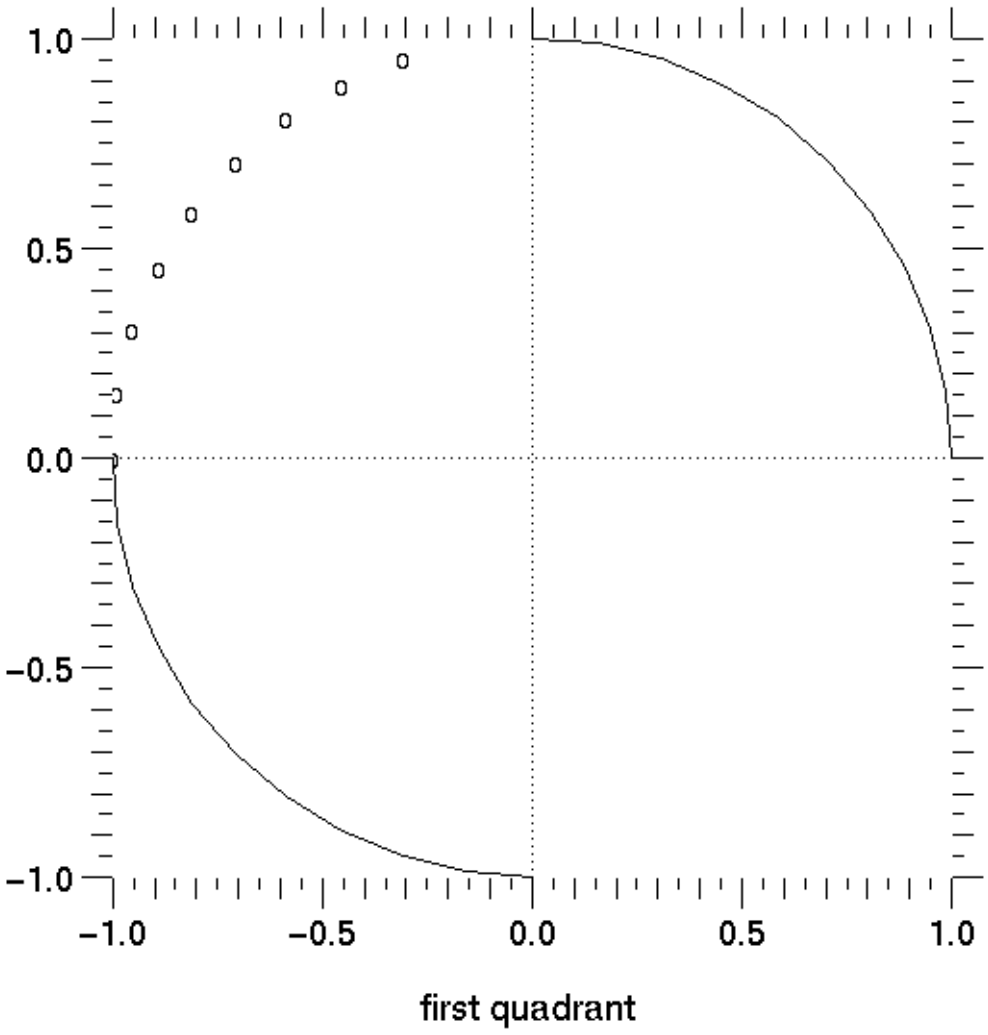
```
# Set x and y scales equal:
attr (scale = "equal")
a = (pi / 2.) * arange (11, typecode = Float) / 10.
# Curve in first quadrant labelled with 1:
titleb ("first quadrant")
plot (cos (a), sin (a), label = "1")
```



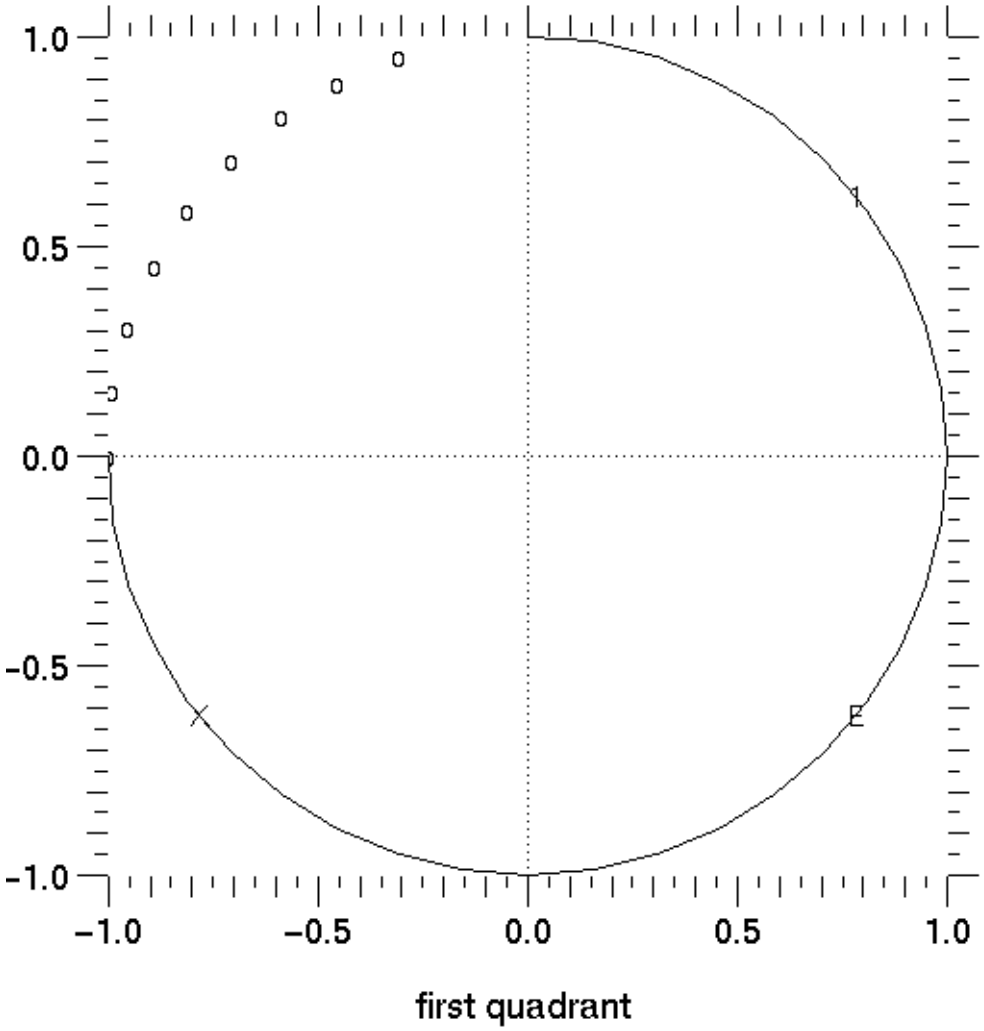
```
# Curve in second quadrant not labelled "Q2" since
#   drawn with a "mark":
plot (cos (a), -sin (a), label ="Q2", mark = "circle")
```



```
# Third quadrant drawn and all labels turned off, but
#   label "XXX" is still associated with quadrant 3:
plot (- cos (a), - sin(a), labels = "no", label = "XXX")
```

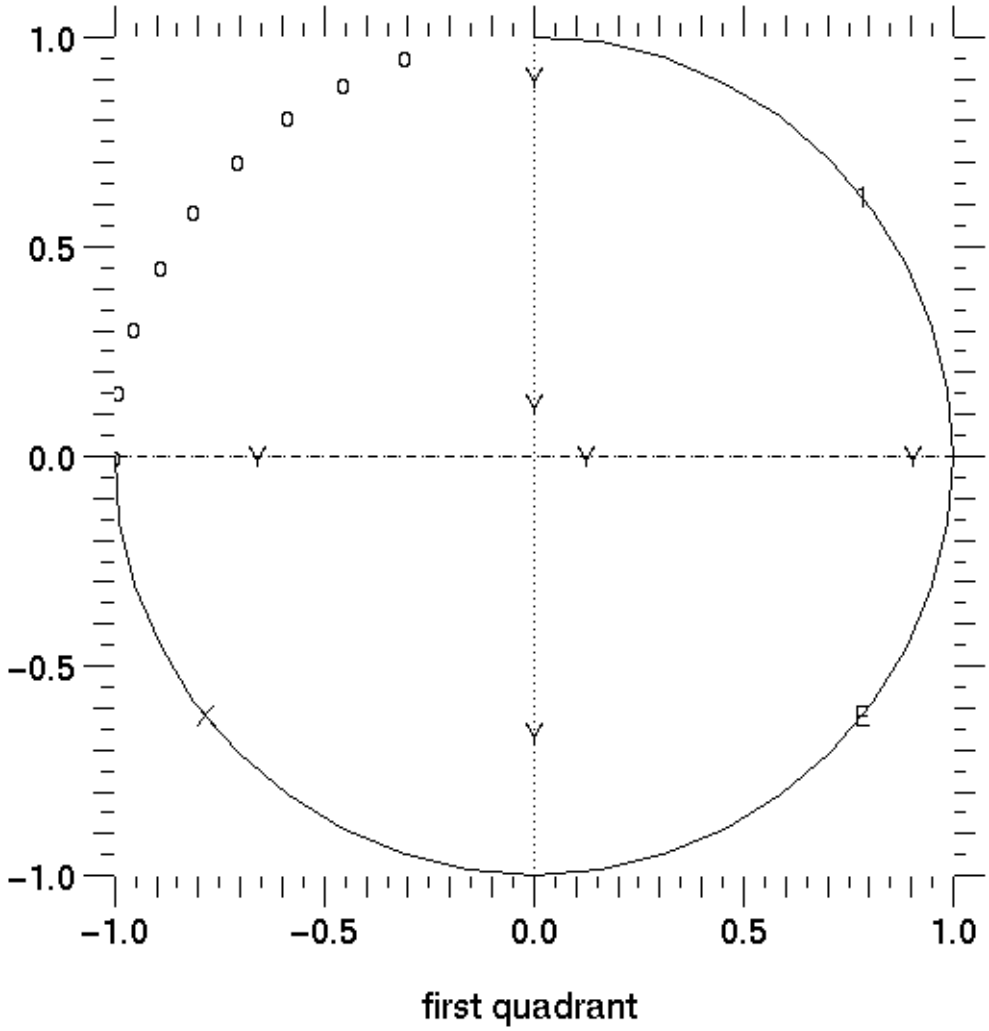


```
# All labels turned back on, including "XXX" in quadrant 3;
#   quadrant 4 labelled with D (maybe):
plot (- cos (a), sin (a), labels = "yes")
attr (label = "YYY")
```



```
# The following two curves will now be labelled with "YYY":
plot (zeros (11, Float),
     (5. - arange (0, 11, typecode = Float)) / 5.,
     style = "dashed")
plot ( (5. - arange (0, 11, typecode = Float)) / 5.,
     zeros (11, Float), style = "dotted")
```

See the following figure for the completed frame.



5.2 plotz: Plotting Contours

Note: **plotz** is not available in Narcisse, which does only three- and four-dimensional contour plots.

Calling Sequence

```
plotz (fexpr [, xexpr [, yexpr]] [, <keylist>])
```

Description

The **plotz** function plots contours of a surface defined by *fexpr* above the point set described by *xexpr* and *yexpr*. *<keylist>* is a list of optional keywords and values.

There are two allowed types of data for contour plots:

- Gridded data: *xexpr* and *yexpr* are one-dimensional arrays, say x and y, and *fexpr* is a two-dimensional array, say z, such that

  ```
  z(i, j) = f(x(i), y(j)), i in range(len(x)), j in range(len(y)).
  ```

  In order for *xexpr* and *yexpr* to form a valid rectangular grid, each array must contain either strictly increasing or strictly decreasing values.

- Mesh data: *fexpr*, *xexpr* and *yexpr* are all two-dimensional arrays of the same shape. In this case, *xexpr* and *yexpr* form a logically rectangular mesh and *fexpr*(i, j) is the value associated with point (*xexpr*(i, j), *yexpr*(i, j)). For mesh-based data, a plot of this type can also be generated by the plotc function; Section 6.5 on page 77.

EZN-style Scattered data plots are not supported by EZPLOT.

Note: *fexpr* can also be the name of a function which, when called with no arguments, returns an array of values of the appropriate shape.

Optional Attributes

The following optional attributes can be specified with this command. For object attributes, they are local to the command specified; i.e., they are not remembered across commands.

window, grid, scale, thick, style, font, mark, lev, color, color_bar

If optional attributes are given on the plot command line, they are specified in the usual form:

key1=value1,key2=value2,...,keyN=valueN

To set an object attribute across commands use the **attr** command. See “Attribute Table” on page 48. for descriptions of the values which can be assigned to these keywords.

The default line style is solid and the default line thickness is 1.0. The default color is the foreground color. To override these defaults, set attributes style, thick, color, respectively. The mark attribute will cause markers to be plotted at each of the mesh points, in the forground color.

5.2.1 Contour Levels

Contour levels are controlled by the `lev` attribute. The attribute `lev` can be used to specify the levels of contours, the scale of the contours (*linear* or *logarithmic*), or a list of specific values for the contour levels. The attribute `lev` can be set either on a plot command or with an attribute call such as "`attr (lev = foo)`". Like any such attribute, if set with `attr` it applies to all `plotz` commands on that frame, except those that override it with a "`lev` = " of their own. However, if a vector of values is specified for `lev`, it will be lost at the next frame advance. There is currently no way to specify such a list to be used on all frames.

In "`lev = foo`", `foo` can be:

- `"linear"` (or `"lin"`): at least `abs(deflev)` linear levels;
- `"log"`: `abs(deflev)` logarithmic levels;
- *n>0*: *n* linear levels;
- *n<0*: `abs(`*n*`)` logarithmic levels;
- a real or double precision list of values.

The default value of `lev` is in the variable `deflev`, whose value is 8; hence, the default is 8 linearly-spaced contour levels.

In Gist, every contour line is labeled with the default (consecutive capital letters). Contours colored according to value are currently not available in Gist.

5.2.2 Contour Color Fill

The `color` attribute, if given a color name, causes all of the contours to be plotted in that color. The `color` attribute for a contour plot can also be used to generate color filled contour bands. Each contour band is a closed polygon (coupled with frame boundaries if necessary) which can be filled with color. The user can set `color = "filled"` to fill the contour levels with colors ranging from *blue* to *red* with *increasing* altitude. When color fill is applied, the contour lines may become unnecessary. The user may specify `color = "fillnl"` to avoid the contour lines' being drawn. Details on filled contour plots may be found in the section "plotf: Fillmesh plot" on page 77

5.2.3 Contour Level Annotations (the Color Bar)

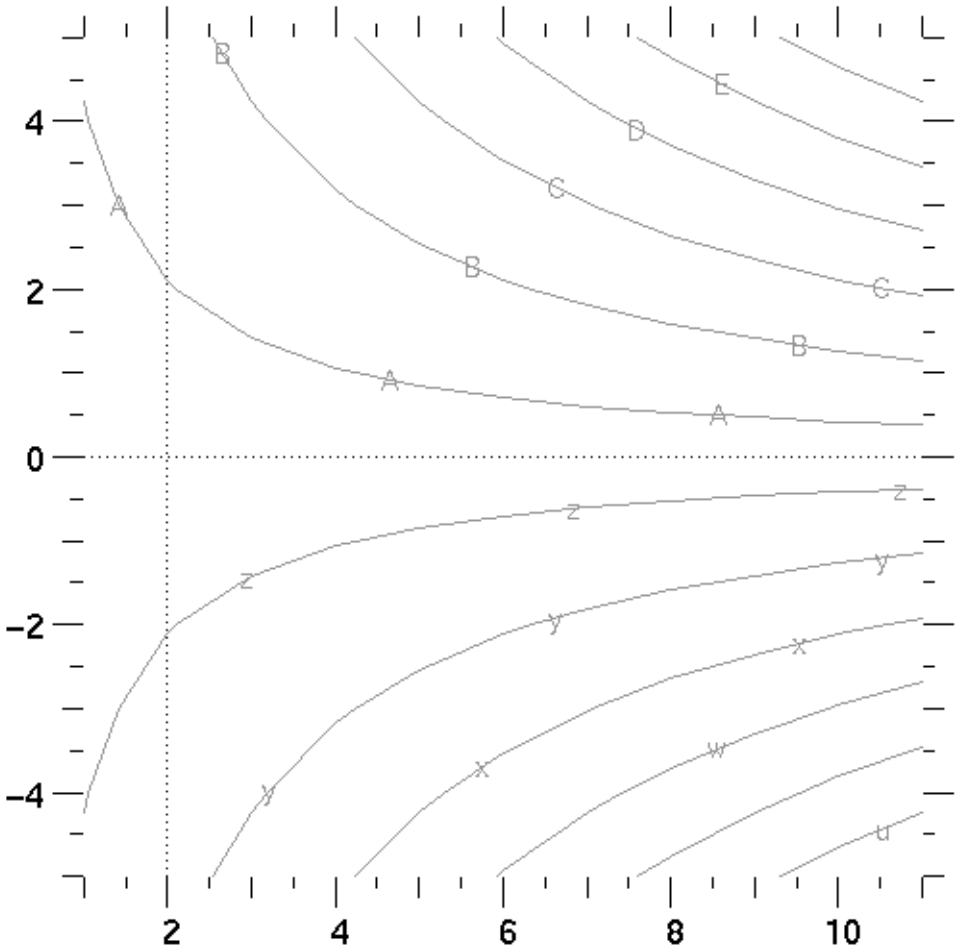
For the contour plots, the contour level annotations can be shown in the right margin of the frame under user's control. This is done by means of the **`color_bar`** keyword argument. If set to 1, then the grid will be slightly reduced in size to take into account the space needed for the color bar, and the color bar will be plotted. If 0, there will be no color bar. (The default is 0.)

The contour level annotation is *color coded* for easy association with the contour level colors. The color assigned is the color of the contour level.

Example

The following example plots a matrix z versus vectors x and y.

```
x = arange (-5, 6, typecode = Float)
y = x + 6
z = multiply.outer (x, y)
plotz (z, x, y, color = "green", lev=12)
sf ()
```



5.3 ploti: Cell Array Plots

Note: `ploti` is not available in Narcisse.

Calling Sequence

```
ploti ( [pvar] [, <keylist>])
ploti ( [cindex] [, <keylist>])
ploti ( <keylist>)
```

Description

The **`ploti`** command is used to plot cell arrays in EZPLOT. The argument *`cindex`* is a two-dimensional array of unsigned character (`typecode 'b'` in Python) whose equivalent integer values (0-255) are color cell indices (i. e., subscripts into the current palette). The argument *`pvar`* is a two-dimensional array of reals, in which case EZPLOT will convert the values to cell indices for you. *`<keylist>`* is a list of optional keywords and values.

For mesh-based data, a more realistic display may be obtained by using the `plotf` command instead; Section 6.5 on page 77.

Optional Attributes

The following optional attributes can be specified with this command. For object attributes, they are local to the command specified; i.e., they are not remembered across commands.

`window`, **`grid`**, **`scale`**, **`color_bar`**

If optional attributes are given in the `ploti` call, they are specified in the usual form:

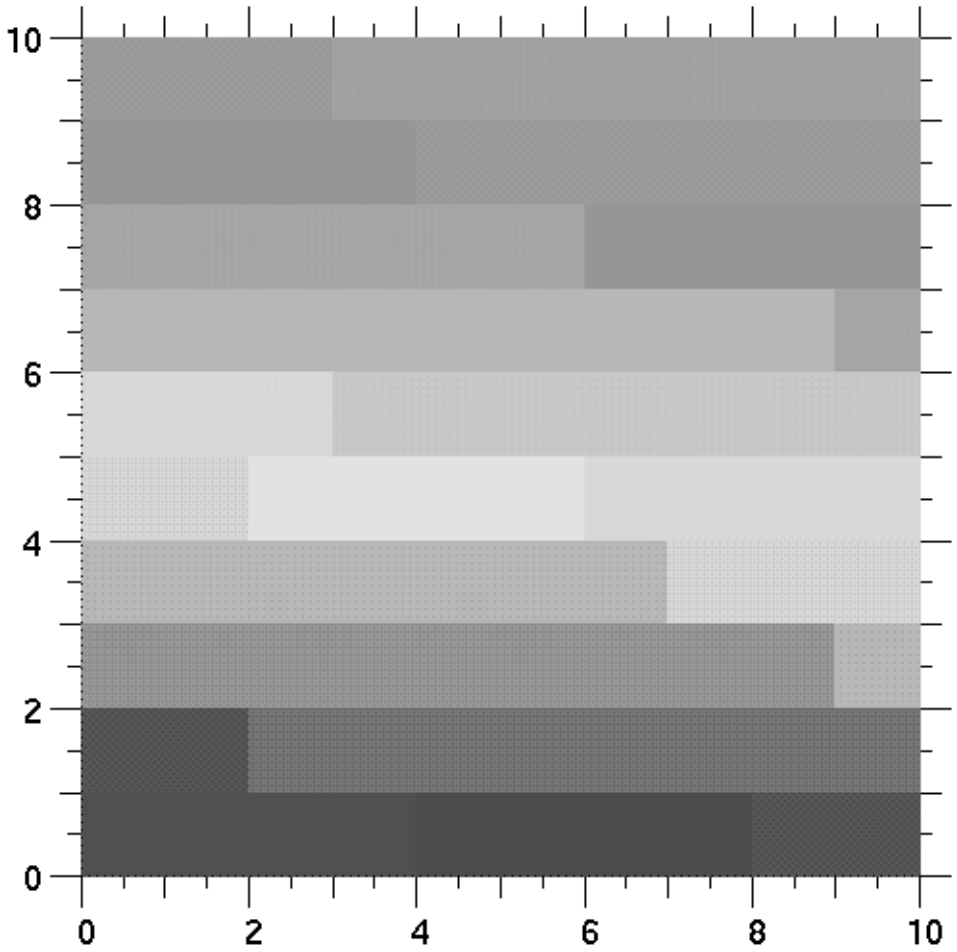
`key1=value1,key2=value2,...,keyN=valueN`

To set an object attribute across commands use the **`attr`** command. See “Attribute Table” on page 48. for descriptions of the values which can be assigned to these keywords.

Example

The following command will cause a 10 by 10 cell array to be plotted, with the first 100 colors of the current color palette, and a corresponding color bar.

```
nf ()
ploti (reshape(arange(100,typecode = 'b'),(10,10))
sf()
# (plot on next page)
```



5.3.1 Color-Mapping Functions

Setting the Color Map

Currently EZPLOT does not support letting you change the color map (or palette). This option can be added if sufficient demand arises.

Mapping Real Data to Color Indices

There is currently no way to do this implemented in EZPLOT, since EZPLOT will accept your data and do the conversions automatically.

CHAPTER 6: Mesh-Oriented Commands

Note: The mesh-oriented commands are not available in Narcisse graphics. Narcisse does three- and four-dimensional mesh plots only.

A mesh-oriented command assumes an underlying logically-rectangular two-dimensional mesh. The x-coordinate of the mesh, *xexpr*, and the y-coordinate, *yexpr*, are both two-dimensional real arrays dimensioned (kmax, lmax)[1]. By convention, *zone* (i, j) is the quadrilateral with upper-right corner (i,j); that is, with diagonally opposite corners (*xexpr* (i - 1, j - 1), *yexpr* (i - 1, j- 1)) and (*xexpr* (i, j), *yexpr* (i, j)).

A mesh-oriented command also requires a region map *ireg* as an argument. This is a two-dimensional integer array, also dimensioned (kmax, lmax), with *ireg* (i, j) the region number for zone (i,j). The values of *ireg* (0, :) and *ireg* (:, 0) are irrelevant. A value of 0 indicates a "void".

The three mesh-defining arrays *xexpr*, *yexpr*, *ireg*, if specified in the plot command, must appear before the first *key = value* pair. They may be dropped from the right, with missing values replaced by defaults. Thus, (x, color = "red") is equivalent to (x, rt, ireg, color = "red").

A mesh-oriented command accepts attribute specifications which specify a subset of the mesh to be plotted by defining values for krange, lrange, and region. The command will plot the subset of the mesh consisting of zones whose indices are in the ranges specified and with region numbers in the region list.

A range specification has the form *(start, stop, inc)*. Fields may be omitted from the right; unspecified fields in the range are set to default values. krange specifies a range for the first subscript, and lrange specifies a range for the second subscript. The defaults are krange = (0, kmax, 1) and lrange = (0, lmax, 1).

In the specification region = *region-list*, *region-list* can be a scalar or vector of integers containing a list of region numbers. The default is region = "all", meaning all regions.

The attributes krange, lrange and region are "*sticky*", which means that after a mesh-oriented plot specifies a value for an attribute, this attribute value will stay in effect for the following mesh-oriented commands until a new frame or until the attribute is reassigned another value.

1. Remember: unlike FORTRAN, Python arrays are subscripted beginning with zero, so the subscripts of a kmax by lmax array range from 0 to kmax - 1 and from 0 to lmax - 1.

For example,

```
plotm (region = [1,3,5])
     #Mesh plot for regions 1, 3 and 5.
plotc (te, color="filled")
     #The contour plot will be restricted to regions 1,3,5.
nf ("no")
```

6.1 `set_mesh` and `clear_mesh`: Specifying the Default Mesh

Calling Sequence

```
set_mesh ( <keylist>)
clear_mesh ()
```

`set_mesh` is used to set all or part of the default mesh for the next mesh plotting functions, until cleared or set to something else. `clear_mesh`, of course, removes the default mesh from existence. All of the mesh plotting functions require a mesh to be specified, either by a preceding `set_mesh` command, or by specifying the mesh to be plotted in the mesh plot function's own arguments. The allowed keywords in *<keylist>* are:

`rt`, `zt`, `ireg`, `pvar`, `cindex`, `ut`, `vt`

`rt` and `zt` are one-dimensional or two-dimensional arrays specifying the mesh. For plotting purposes, `zt` may be thought of as the abscissa (the x coordinate), and `rt` as the ordinate (the y coordinate). Suppose the mesh size is `kmax` by `lmax`. Then `zt` must be of dimension `kmax` (if one dimensional) or `kmax` by `lmax` (if two). Likewise, `rt` must be of dimension `lmax` or `kmax` by `lmax`. `ireg` is a `kmax` by `lmax` array of integers specifying the regions of the mesh. As mentioned earlier, the first row and column of `ireg` are meaningless and should be set to zeros. The keywords `pvar` and `cindex` are mutually exclusive: `pvar`, if present, is a `kmax` by `lmax` array of reals values used to make a contour plot on the mesh. `cindex`, if present, is a `kmax` by `lmax` array of indices into a color table specifying the colors for a filled mesh. Finally, `ut` and `vt` are `kmax` by `lmax` real arrays which specify a vector field defined at each mesh point, for use in making vector field plots.

Any part of the default mesh may be overruled by a specification in a plot function's arguments; any part of the default mesh that was not set by a call to `set_mesh` *must* be specified in a plot function call if that function needs it. There are no default or pre-set values.

6.2 `ezcpvar`, `ezccindex`, `ezcx`, `ezcy`, `ezcireg`, `ezcu`, `ezcv`: Convenience Functions

The functions enumerated above may be used (if desired) to set the global values of `pvar`, `cindex`, `rt`, `zt`, `ireg`, `ut`, and `vt`, respectively. They each accept a single, non-keyword argument.

6.3 `plotm`: Plotting Meshes, Boundaries, and Regions

Calling Sequence

```
plotm (<keylist>)
plotb (<keylist>)
```

Description

`plotm` is a mesh-oriented command. For general information, see the chapter introduction on page 67. In a departure from EZN, EZPLOT requires that all arguments be keywords.

The `plotm` function plots meshes. If the keyword `bnd` is set to `"yes"` (or `1`), only the boundaries of regions are plotted. If specified, `rt` is an array of y-axis values, `zt` is an array of x-axis values, `ireg` is a region map, and *`<keylist>`* is a list of other optional keywords and values.

If `plotm` arguments are omitted, they are supplied by using the values set by the nearest preceding call to `set_mesh`. If there was no such previous call, then the plot is not possible and an exception will be raised.

As a special case, `plotm (bnd = 1)` can be abbreviated `plotb`.

By convention, the curves connecting nodes are divided into two sets,

k-lines: (*xexpr*(k, :), *yexpr*(k, :)), k in range (kmax); and

l-lines: (*xexpr*(:, l), *yexpr*(:, l)), l in range (lmax).

The `krange` and `lrange` attributes can be given a stride `j` to cause only every j^{th} line in that direction to be plotted. The stride is ignored for boundary plots, and ignored in drawing the lines in the opposite direction (that is, the `l`-lines will have all their pieces even if `krange` has a stride `j`, while only every j^{th} `k`-line will be plotted).

Optional Attributes

The following optional attributes can be specified with this command. For object attributes, they are local to the command specified; i.e., they are not remembered across commands.

`grid`, `scale`, `kstyle`, `lstyle`, `thick`, `bnd`, `color`, `mark`, `labels`, `krange`, `lrange`, `region`, `window`

If optional attributes are given on the plot command line, they are specified in the usual form:

`key1=value1,key2=value2,...,keyN=valueN`

To set an object attribute across commands use the **`attr`** command. See “Attribute Table” on page 48. for descriptions of the values which can be assigned to these keywords.

The default line style is `"solid"` and the default line thickness is 1.0. The default color is the foreground color. To override these defaults, set attributes `style`, `thick`, and `color` respectively.

The attribute `mark` can be used to plot markers at the nodes instead of drawing mesh lines to con-

nect the nodes. This is similar to the function `plot` with the `mark` attribute.

Optional attributes `kstyle` and `lstyle` set the line style for the `k`-lines and `l`-lines, respectively. By default, both are set to `"solid"`. If a style is set to `"none"`, then no lines are plotted in that direction.

The color specified by the `color` attribute is used to specify the colors of `k`-lines and `l`-lines. EZPLOT does not currently support separate colors for `k`-lines and `l`-lines.

The `window` attribute is used to specify a device number from 0 to 7 for the plot, or `"cgm"`, or `"ps"`, or `"all"` if the plot is to appear in all active devices. `"all"` is the default.

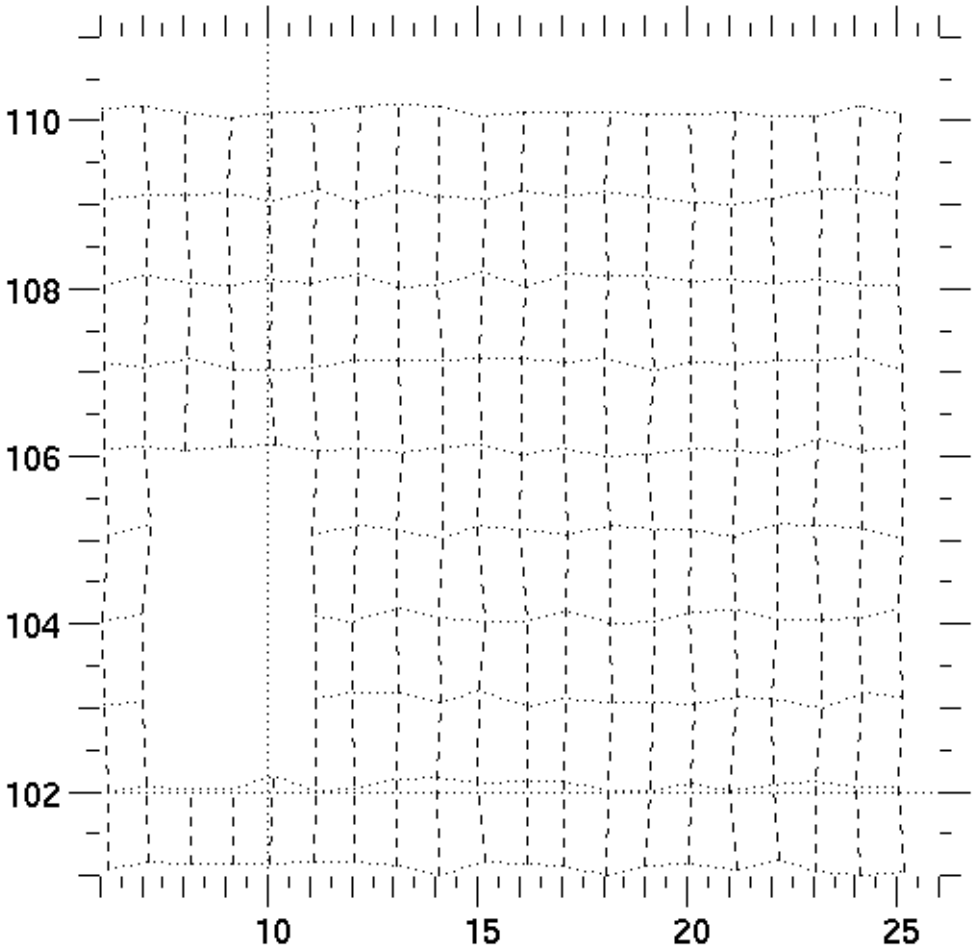
Examples

The following data are used for the examples here and in Section 6.5 “plotf: Fillmesh plot”.

```
#  Define mesh:
kmax = 25
lmax = 35 #Don’t make either smaller than 25.
xr = multiply.outer (arange (kmax, typecode = Float),
     ones(lmax, Float))
yr = multiply.outer (ones(kmax, Float),
     arange (lmax, typecode = Float))
from Ranf import *  # Used in following lines
zt = 5. + xr + .2 * random_sample (kmax, lmax)
rt = 100. + yr + .2 * random_sample (kmax, lmax)
#  Define region map:
ireg =  multiply.outer (ones (kmax), ones (lmax))
ireg [0:1, 0:lmax]=0
ireg [0:kmax, 0:1]=0
ireg [1:15, 7:12]=2
ireg [1:15, 12:lmax]=3
ireg [3:7, 3:7]=0    #Define an internal void.
k2 = 1
l2 = 7   #Index of a point in region 2.
#  Define data on the mesh:
s = 1000.
z = s * (rt + zt)
z [3:10, 3:12] = z [3:10, 3:12] * .9
z [5, 5] = z [5, 5] * .9
z [17:22, 15:18] = z [17:22, 15:18] * 1.2
z [16, 16] = z [16, 16] * 1.1
set_mesh (zt = zt, rt = rt, ireg = ireg, pvar = z)
```

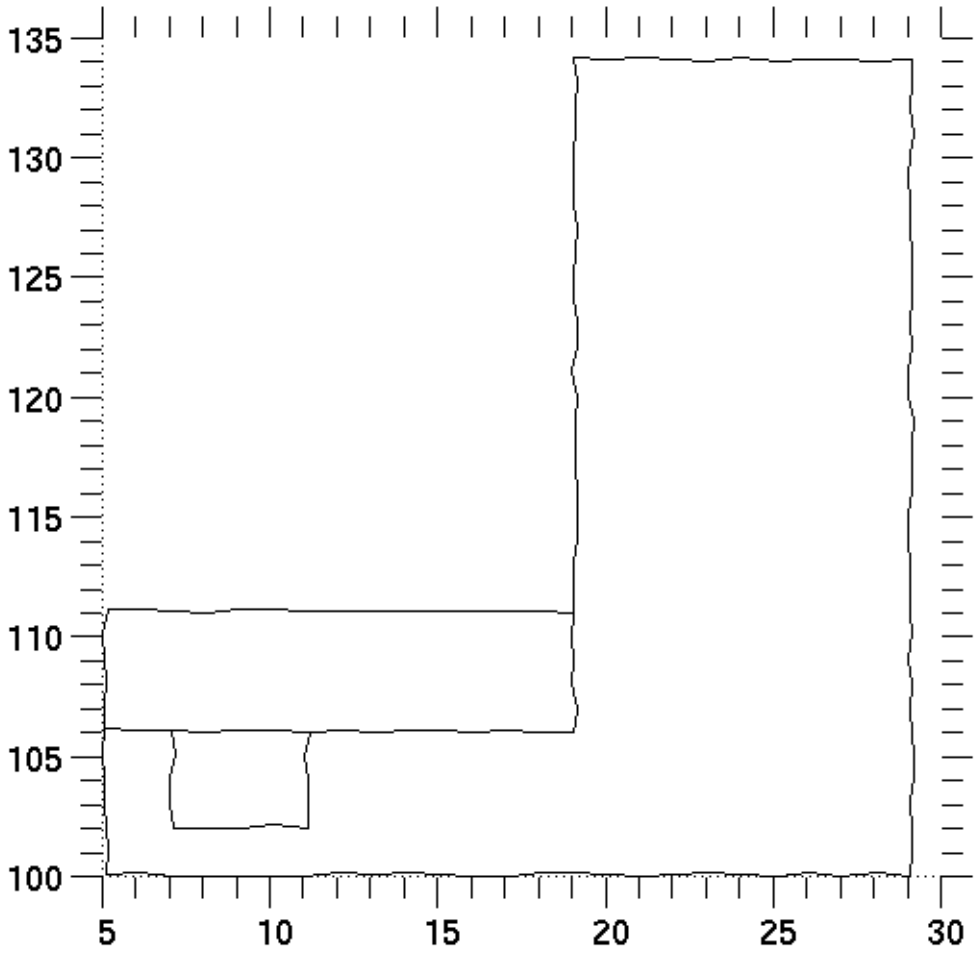
In the first example, a mesh is plotted with `k`-lines dashed and `l`-lines dotted. Here, the displayed mesh has been restricted to lines with k ranging from 1 to 20 and l from 1 to 10. Note that nothing is plotted where the interior void was defined.

```
nf ()
plotm (kstyle = “dashed”, lstyle = “dotted”,
   krange = (1, 20), lrange = (1, 10))
sf ()
```



Here we plot just two regions. Note that the full extent of the mesh is used.

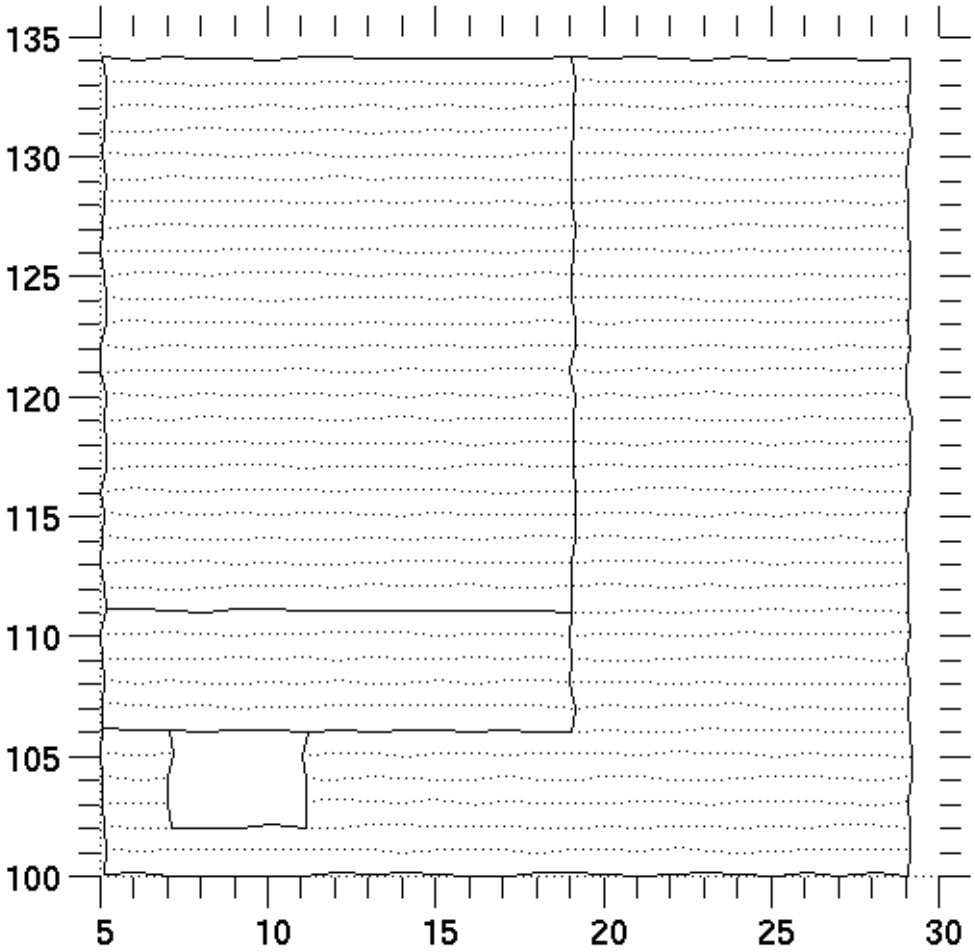
```
nf ()
plotm (bnd = 1, region = [1, 2])
# Plot boundaries of regions 1 and 2.
sf ()
```



And here we plot all region boundaries, and then just the l-lines:

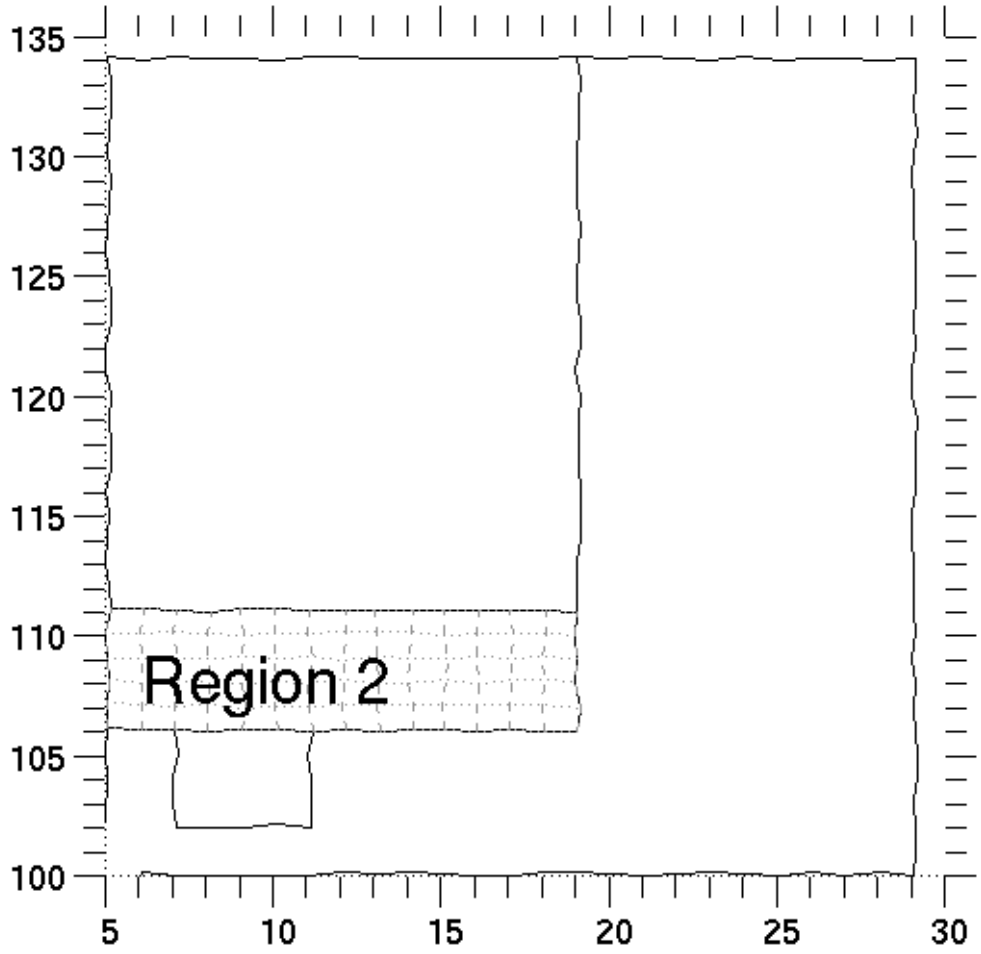
Figure 6.1: Example of Boundaries Plot

```
nf ()
plotb ()              # Plot boundaries.
plotm (kstyle = “none”, lstyle = “dotted”)
                    # Plot just the l-lines of the mesh.
sf ()
```



Finally, we plot all region boundaries, and mark region 2 with text in it. Note that this looks better on the screen, because the colored mesh lines make the text stand out.

```
plotb ()              # Plot boundaries
plotm (region = 2, kstyle = “dashed”, lstyle = “dotted”,
       color = “green”)
text (“Region 2”, zt[k2,l2], rt[k2,l2], 32, tosys = 1)
nf ()
```



6.4 plotc: Plotting Contours

Note: plotc is currently not available in Narcisse. Narcisse does contour plots only in three and four dimensions.

Calling Sequence

```
plotc (<keylist>)
```

Description

plotc is a mesh-oriented function. For general information, see the chapter introduction on page 67. Note that unlike EZN, all arguments of plotc must be keyword arguments.

The plotc function plots a contour map of *pvar* above the mesh described by *rt* and *zt*. *pvar* is a two-dimensional array of real values dimensioned the same as *rt* and *zt*, or, if the latter are one dimensional, then *pvar* will be len(rt) by len(zt). If specified, *rt* is an array of y-axis values, *zt* is an array of x-axis values, *ireg* is a region map, and *<keylist>* is a list of other optional keywords and values. Strides in krange or lrange are ignored by plotc.

If plotc mesh-defining arguments are omitted, then they are supplied by using the values set by the closest preceding call of set_mesh. If there has been no such call, or if they have not been set, then there is nothing to plot, and an exception will occur.

pvar can also be the name of a function which, when called with no arguments, returns a two-dimensional array of values of the appropriate shape.

Optional Attributes

The following optional attributes can be specified with this command. For object attributes, they are local to the command specified, i.e. they are not remembered across commands.

grid, scale, thick, style, mark, lev, color, krange, lrange, region, window

If optional attributes are given on the plot command line, they are specified in the usual form:

key1=value1,key2=value2,...,keyN=valueN

To set an object attribute across commands use the **attr** command. See “Attribute Table” on page 48. for descriptions of the values which can be assigned to these keywords.

The window attribute is used to specify a device number from 0 to 7 for the plot, or "cgm", or "ps", or "all" if the plot is to appear in all active devices. "all" is the default.

Contour Levels, Colors, etc.

The discussion of the command plotz (Section 5.2 on page 61) contains a detailed explanation of the way contour levels and colors are specified. The discussion there applies to plotc as well.

The primary difference between plotc and plotz is that the former is a mesh-oriented function. This means that only the plotz mesh data discussion applies to plotc. Furthermore, because of the underlying mesh and the associated region map, the plotc command has the possibility of controlling the subregion over which contours are displayed by use of attributes krange, lrange, region.

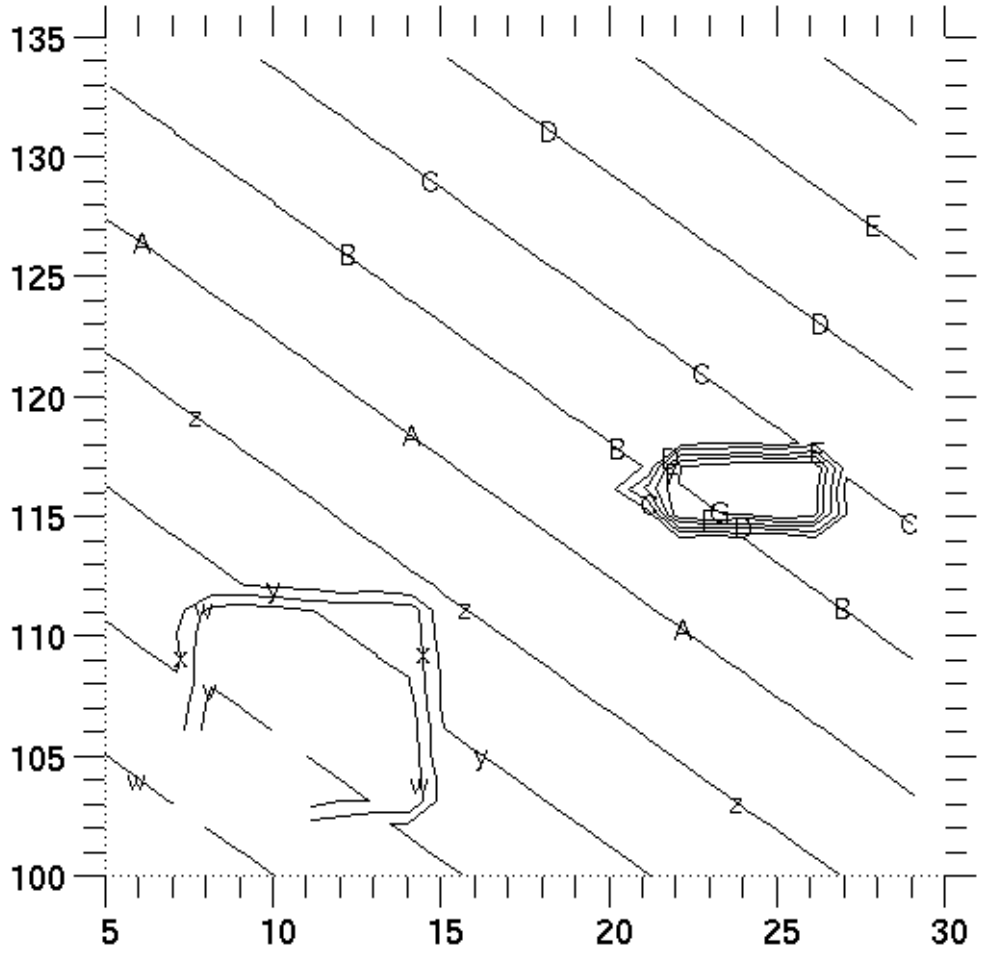
The plotc command assumes that the physics quantity *pval* is mesh-based, which means that

pval`(i, j)` is the value associated with mesh point `(i, j)`. Currently EZPLOT does not support zone-based quantities.

Example

The following is an example of using `plotc` with default arguments. The data are as defined before the `plotm` examples, page 70. Note the gap in the graph at the internal void.

```
nf ()
plotc (pvar = z, lev = 13)
sf ()
```



6.5 `plotf`: Fillmesh plot

Note: `plotf` is not available in Narcisse. Narcisse supports filled mesh plots only in three and four dimensions.

Calling Sequence

```
plotf ([pvar [,zt [,rt [,ireg]]]] [,<keylist>])
plotf ([cindex [,zt [,rt [,ireg]]]] [,<keylist>])
```

Description

`plotf` is a mesh-oriented command. For general information, see the chapter introduction on page 67.

Note that `plotf` allows the positional arguments *`pvar`* (or *`cindex`*), *`zt`*, *`rt`*, and *`ireg`*. For consistency with other mesh plotting commands, these variables can (and probably should) be entered as keyword arguments, or set by `set_mesh`.

The `plotf` function plots a color-filled mesh which displays the physics quantity *`pvar`* (or the colors indexed by *`cindex`*) in the zones of interest with colors. If specified, *`zt`* is an array of x-axis values, *`rt`* is an array of y-axis values, *`ireg`* is a region map, and *`<keylist>`* is a list of optional keywords and values.

If `plotf` arguments are omitted, they are supplied by using the values set by the most recent call of the function `set_mesh`. If the values have not been set, then there is nothing to plot, and an exception will be raised.

The colors assigned to the individual zones range from the beginning color in the colormap to the last color in the colormap. The color varies from low color index to high color index as *`pvar`* varies from its minimum to maximum values.

The mapping of colors can be *linearly*, *logarithmically*, or *normally distributed*. The user can use the attribute `cscale` to specify the mapping choice. For example, set `cscale = "log"` to set the color mapping to logarithmic values of the physics quantity. The default mapping is linear. The normal distribution color mapping (`cscale = "normal"`) will map *`pvar`* values which are 2 standard deviations below the mean to the lowest color index, and *`pvar`* values which are 2 standard deviations above the mean to the highest color index. The intermediate *`pvar`* values are mapped in the normal distribution fashion. A colored annotation on the right side of the frame displays the assignment of colors to the corresponding values of *`pvar`*.

The `plotf` command also accepts an integer array *`cindex`* to directly assign color indices to the zones in the mesh. The integer array must be of dimension `(kmax, lmax)` and with values between the lowest color index and the highest color index (usually between 1 and 192). When directly assigned color indices are used, no color annotation will be displayed, because EZPLOT has no knowledge how the color mapping is defined.

Optional Attributes

The following optional attributes can be specified with this command. As *object* attributes, they are local to the command specified, i.e. they are not remembered across commands.

`color`, `cscale`, `krange`, `lrange`, `region`, `window`, `color_bar`

If optional attributes are given on the plot command line, they are specified in the usual form:

`key1=value1,key2=value2,...,keyN=valueN`

To set an object attribute across commands use the **`attr`** command. See “Attribute Table” on page 48. for descriptions of the values which can be assigned to these keywords.

Due to the possibility of different color assignment schemes in different regions or with different physics quantities, the `krange`, `lrange`, `region` attributes are made “*non-sticky*”; i.e., the submesh specifications will not be remembered during subsequent fillmesh plots in the same frame. This differs from the effects of `krange`, `lrange`, `region` on the `plotm` command (Section 6.3 on page 69).

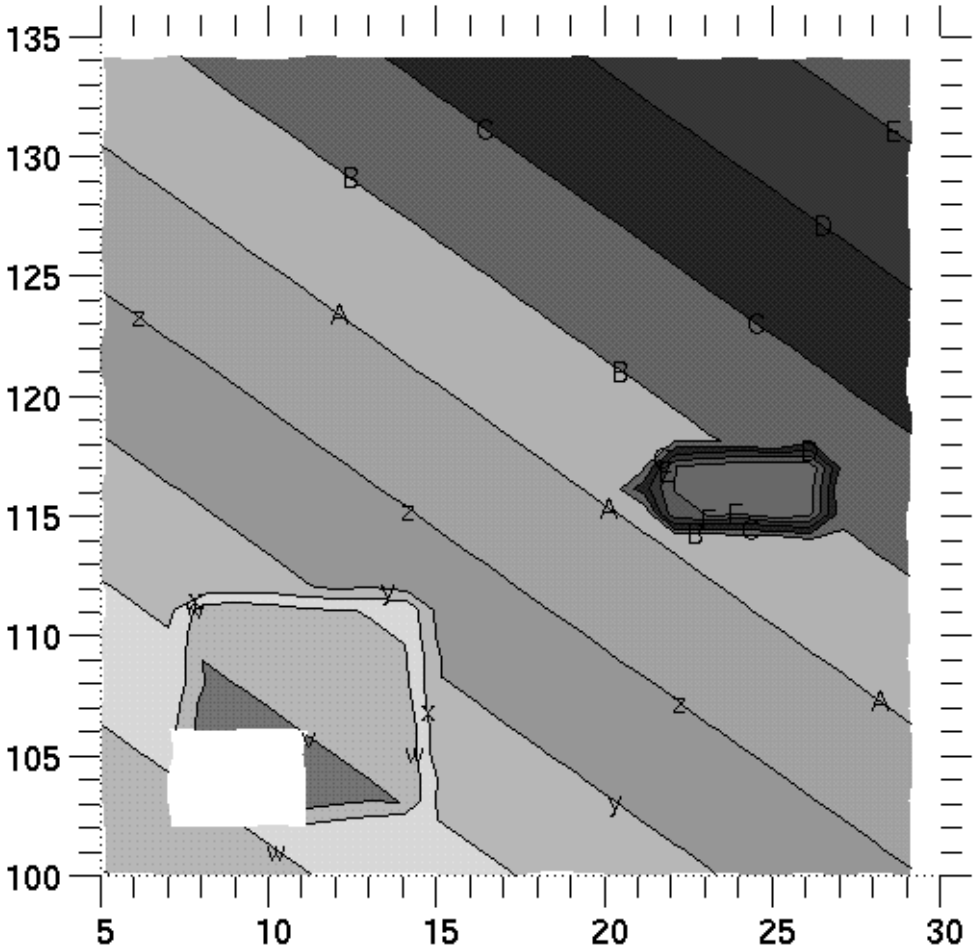
The `window` attribute is used to specify a device number from 0 to 7 for the plot, or `"cgm"`, or `"ps"`, or `"all"` if the plot is to appear in all active devices. `"all"` is the default.

Examples

For our first example, assume the same data as defined before the `plotm` examples, page 70. Note that nothing gets plotted in the void, so it has the background color. The `plotc` call in this example will superimpose contours on the filled mesh plot:

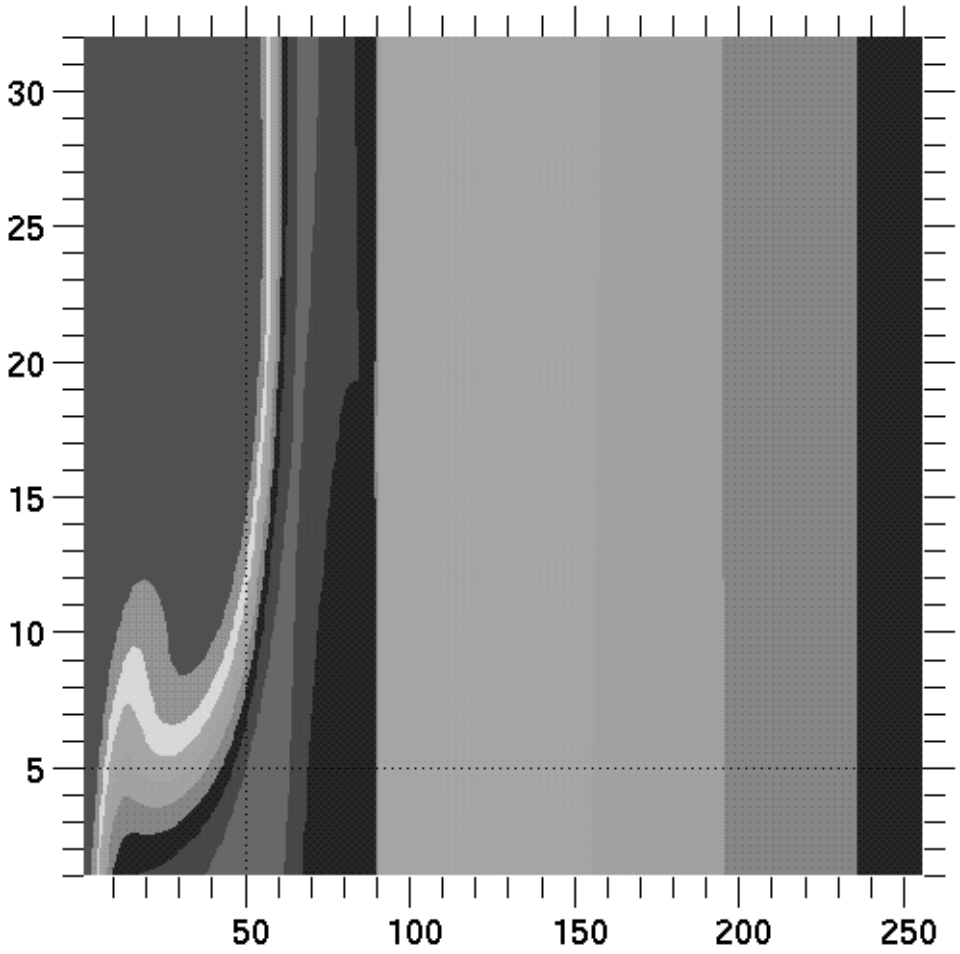
```
nf ()
plotc (pvar = z, color = "filled", lev = 12)
# Superimpose 12 contours
sf ()
```

The plot appears on the next page.


135
130
125
120
115
110
105
100
5
10
15
20
25
30
E
D
C
B
A
z
y
x
w

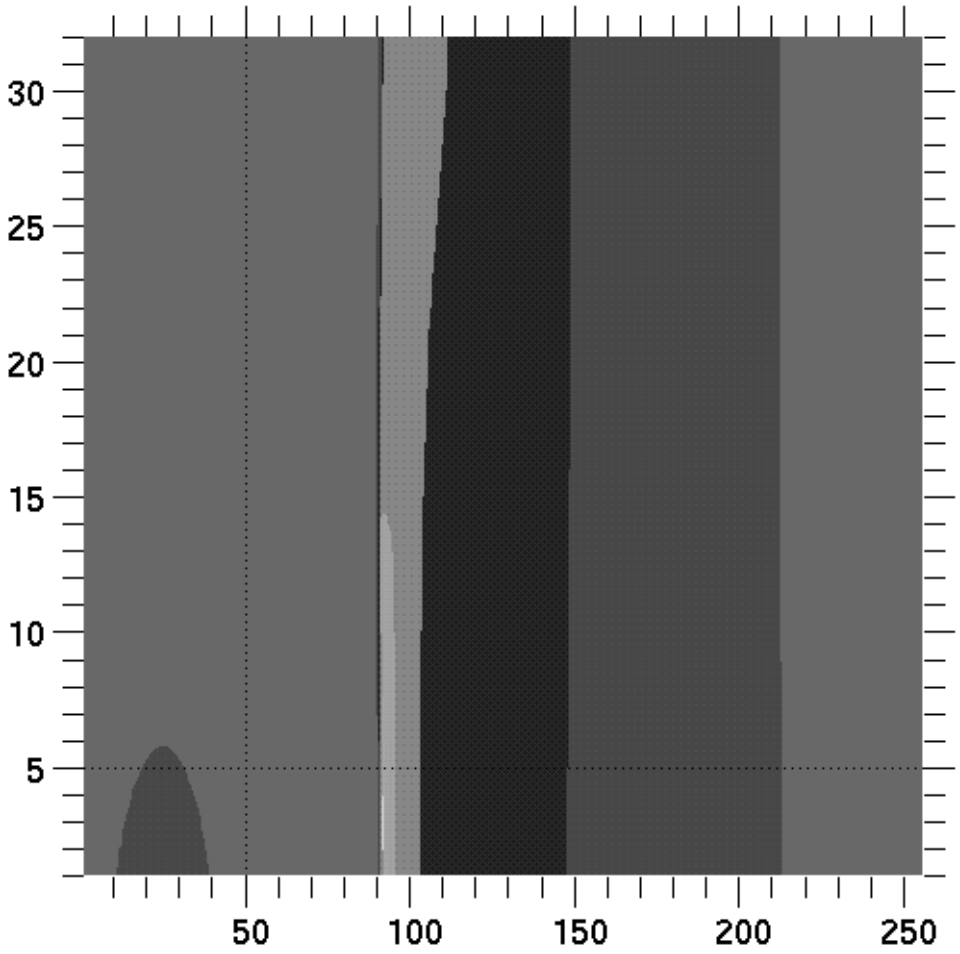
For our next set of examples, assume that a PFB dump file `test1z` has been created, and we want to examine some of its physics variables. First we do a linearly-scaled fillmesh plot of variable `rhoout`:

```
from PFB import *    # import the PDB read module
f = PR ('./test1z') # open the file
nf ()
plotf (pvar = f.rhoout)
```



Next we do a logarithmically-scaled fillmesh plot of variable `vxout`:

```
nf ()
plotf(pvar=f.vxout,cscale="log")
sf ()
```



6.6 plotv: Plotting Vectors

Note: `plotv` is not available in Narcisse.

Calling Sequence

```
plotv ([zt [,rt [,vt [,ut [,ireg]]]]] [,<keylist>])
plotv (<keylist>)
```

Description

plotv is a mesh-oriented command. For general information, see the chapter introduction on page 67.

Note that plotv allows the positional arguments zt, rt, vt, ut, and ireg. For consistency with other mesh plotting commands, these variables can (and probably should) be entered as keyword arguments, or by using set_mesh.

The plotv function plots velocity vectors on a mesh. If specified, *rt* is an array of y-axis values, *zt* is an array of x-axis values, *ut* is the displacement for *rt*, *vt* is the displacement for *zt*, *ireg* is a region map, and *<keylist>* is a list of optional keywords and values.

If plotv arguments are omitted, they are supplied by using the values entered in the most recent call to set_mesh. If any are undefined, there is nothing to plot, and an exception will occur.

A series of arrows from (*rt, zt*) to (*rt+ut*dx, zt+vt*dy*) is plotted. The values *dx* and *dy* are chosen so that the maximum extent of an arrow in the corresponding direction is the frame size in that direction multiplied by the vsc attribute. The default for vsc is .5; this default can be changed by assigning a new value to defvsc.

Optional Attributes

The following optional attributes can be specified with this command. For object attributes, they are local to the command specified, i.e. they are not remembered across commands.

grid, scale, style, thick, vsc, color, krange, lrange, region, window

If optional attributes are given on the plot command line, they are specified in the usual form:

key1=value1,key2=value2,...,keyN=valueN

To set an object attribute across commands use the **attr** command. See “Attribute Table” on page 48. for descriptions of the values which can be assigned to these keywords.

The default line style is solid and the default line thickness is 1.0. The default color is the foreground color. To override these defaults, set attributes style, thick, and color respectively.

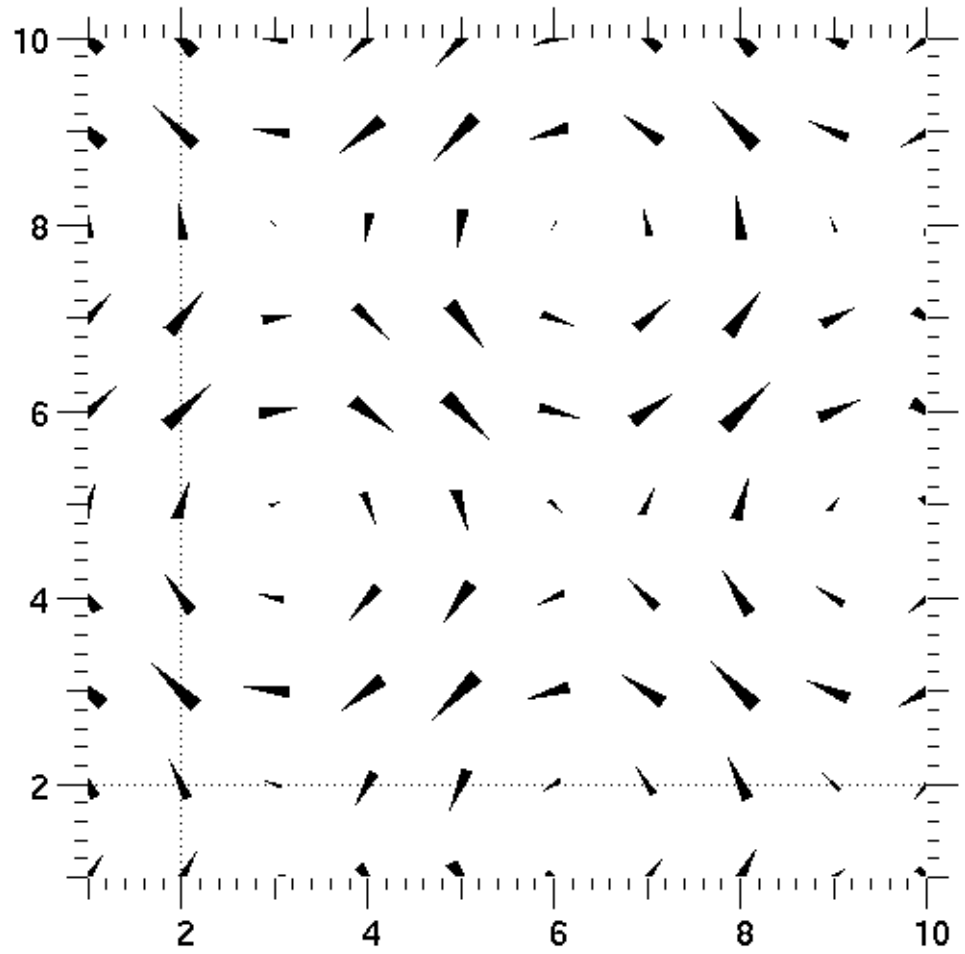
The window attribute is used to specify a device number from 0 to 7 for the plot, or "cgm", or "ps", or "all" if the plot is to appear in all active devices. "all" is the default.

Examples

In the first example, the input arrays are explicitly specified. The line thickness of vectors will be 0.1.

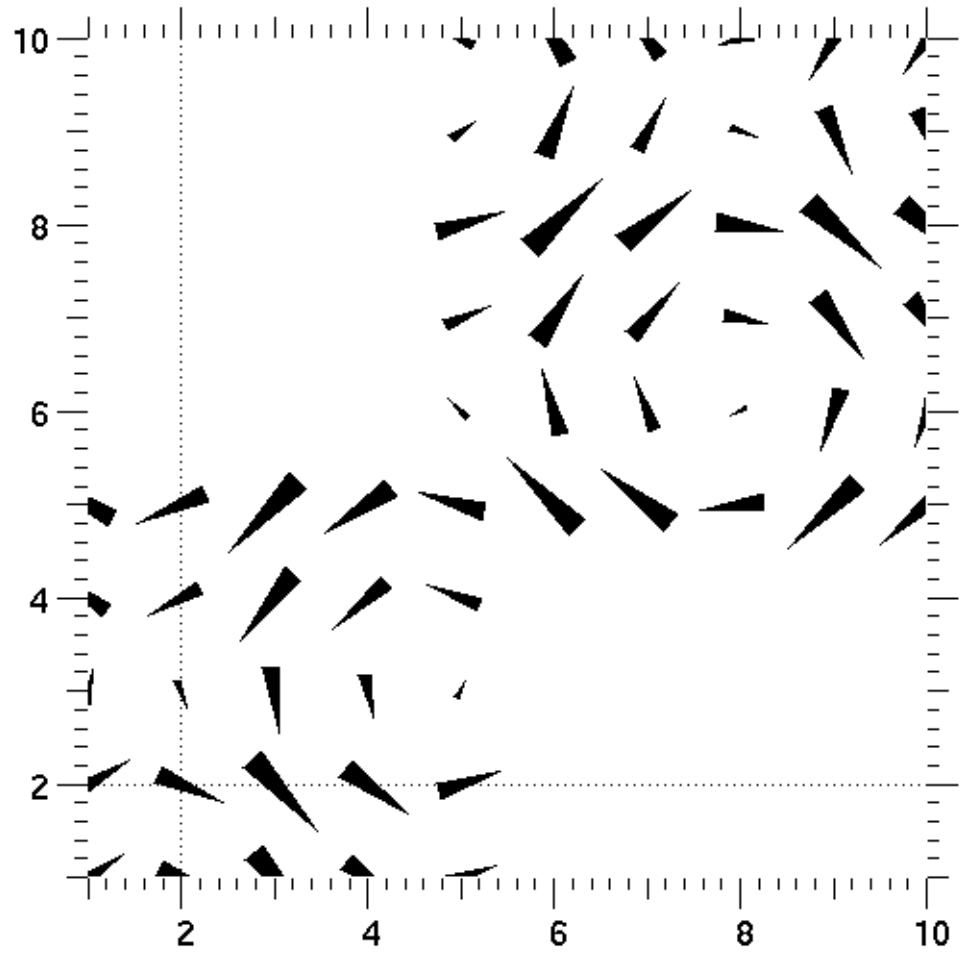
```
nf ()
ireg = zeros ( (10, 10), Int)
vals = arange (1, 11, typecode = Float)
muls = ones (10, typecode = Float)
x = multiply.outer (vals, muls)
y = multiply.outer (muls, vals)
```

```
vx = sin (y)
vy = cos (x)
# Define regions:
ireg [1:5, 1:5] = 1
ireg [1:5, 5:10] = 2
ireg [5:10, 1:5] = 3
ireg [5:10, 5:10] = 4
plotv (zt = y, rt = x, vt = vy, ut = vx, ireg = ireg, thick =
0.1) # Arguments explicitly specified.
```



In the second example, the default values are set by `set_mesh` and then are used. The displacement vectors are scaled to 0.8. Only vectors originating at nodes of zones in regions 1 and 4 are plotted. (Note that this is transposed relative to the previous example.)

```
# Continuation from the last example.
# Set up zt,rt,ut,vt:
set_mesh (zt = x, rt = y, vt = vx, ut = vy, ireg = ireg)
plotv (vsc = .8, region = [1, 4])
```



CHAPTER 7: Text Plotting and Miscellaneous

7.1 `titles`: Put titles on the plot

Calling Sequence

```
titles ( top [, bottom [, left [, right]]])
```

Description

Put up to four quoted strings at the top, bottom, left, and right of the picture, respectively. Each title can also be set individually by calling the appropriate function `titlet`, `titleb`, `titlel`, or `titler` with a quoted string as argument.

The default value of each title is a blank string.

7.2 `text`: Put text on the plot

Calling Sequence

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

Description

Write *str* on the plot beginning at coordinates *x*, *y*; *size* gives the size of the text in points.

Optional Attributes

The following optional keyword arguments can be specified with this function. For object attributes, they are local to the command specified, i.e., they are not remembered across commands.

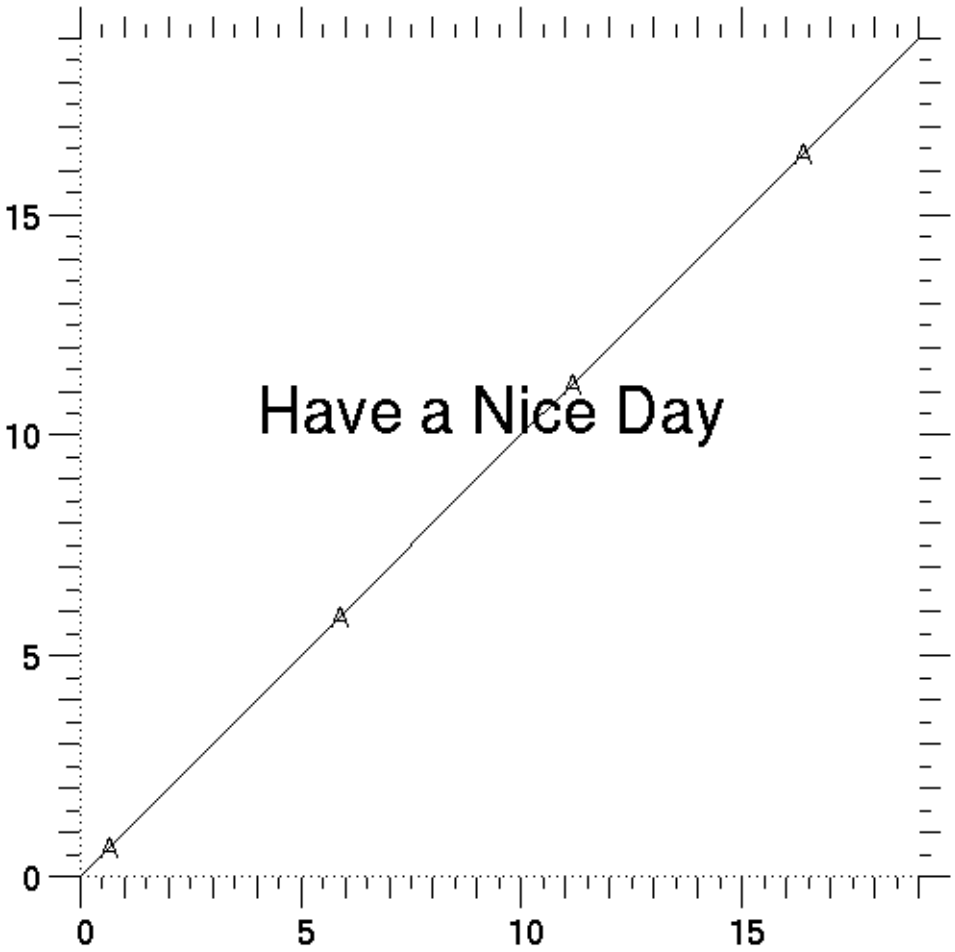
`tosys`, `window`, `color`

`tosys` specifies the coordinate system. If 0, it denotes window coordinates, which vary from [0., 0.] to [1., 1.]. If 1, it denotes the user coordinates (along the plotted x and y axes). The default is 1. `window` may be used to specify a particular device (0 through 7, `"cgm"`, or `"ps"`) or all active devices (`"all"`). The default is `"all"`. `color` may be used to specify the color of the text; it defaults to `"fg"` (foreground). Because of the `tosys` keyword, there is no EZPLOT equivalent of the EZN function `ftext`.

To set an object attribute across commands use the **attr** command; Section 4.2 on page 43 for descriptions of the values which can be assigned to these keywords.

Example

```
# Example of text command
nf ()
plot (arange (20))
text ("Have a Nice Day", 4, 10, 24, tosys = 1)
sf ()
```



CHAPTER 8: Control Variables and Defaults

EZPLOT differs from EZN in that its internal variables are not intended to be available to the user. They are set not by assignment, but by calling a function. Thus, for instance, in Basis one would say

```
ezcshow = true
```

whereas the correct Python function call to EZPLOT is

```
ezcshow ("true")
```

Internal values for attributes are set by calling `attr` (see “Attributes” on page 37). The *default* values of many internal attributes (i. e., the values to which they are set lacking calls to `attr`, which sets them) can be set by assignment, however; these variables are listed below (see “Default Attributes” on page 88).

8.1 Setting Control Variables

Here are some details on some of the functions in EZPLOT which set these internal variables.

`ezcshow` Determines if the current picture is displayed each time it is changed by an EZPLOT function call, or only when a *frame* attribute is changed or `nf` is called. `"true"` by default (display after each change).

`ezcreset` Determines if attributes set by the `attr` function are reset to the default values upon a frame advance. If `"false"`, attributes will remain set across frame advances. `"true"` by default.

`ezcvsc` Determines the size of the largest vector arrow relative to the frame size for the `plotv` command. See “Attribute Table” on page 48., `vsc`.

`titlet` takes string valued argument, sets the *top title* for a frame. Default: a blank string.

`titleb` takes string valued argument, sets the *bottom title* for a frame. Default: a blank string.

`titlel` takes string valued argument, sets the *left title* for a frame. Default: a blank string.

`titler` takes string valued argument, sets the *right title* for a frame. Default: a blank string.

`title` takes from one to four string valued arguments, which are from left to right the *top*, *bottom*, *left*, and *right titles*. Arguments may be omitted from the right only. Defaults: four blank strings.

8.2 Default Attributes

You can change default settings of internal attributes (i. e., what they would be set to lacking calls to `attr` or by use of keyword arguments in a plot function call) by assigning new values to the following variables in EZPLOT. Their original values are shown for each.

Defaults for attributes

```
defgridx ("off")
   #grid lines in x direction
defgridy ("off")
   #grid lines in y direction
defvsc (0.5)
   #size of largest vector, relative to the frame size
defthick (1.0)
   #thickness of lines
defmark (" ")
   #mark -- blank for curves
defstyle ("solid")
   #line style
deflabels (1)
   #show labels on curves?
deflabel (" ")
   #default curve label
defscale ("linlin")
   #scale: "linlin", "linlog", "loglin", "loglog", or
   # "equal"
defcolor ("fg")
   #normal color
deflev (8)
   #Minimum number of contour levels to choose
   #Negative means use logarithmic contours
deftop ("")
   #title for top
defbot ("")
   #title for bottom
defleft ("")
   #title for left
defright ("")
   #title for right
```

```
defcgm ("yes")
   #plot to a cgm file?
defps ("no")
   #plot to a postscript file?
```

8.3 Setting Default Mesh Variables

The following functions may be used to set default mesh variables for mesh plots. Default variables, once set, can be changed to different values by calling the same functions, cleared by calling `clear_mesh`, or overruled by the variables specified in an individual plot function call. In any of the following calls, the value specified for a variable may be `None`, in which case the default value is simply unspecified, and must be supplied by a plot command that needs it.

`set_mesh ([rt = <array1>] [,] [zt = <array2>] [,]`
`[ireg = <array3>] [,] [ut = <array3>] [,]`
`[vt = <array4>] [,] [pvar = <array5>] [,]`
`[cindex = <array6>])`

defines a two-dimensional mesh for plotting. `rt` and `zt` are either real, two-dimensional arrays of the same shape (say `kmax` by `lmax`) defining the mesh, or else `rt` is a vector `kmax` long and `zt` is a vector `lmax` long. (`rt` is the abscissa, `zt` the ordinate.) `ireg` is a two-dimensional (`kmax` by `lmax`) integer array defining which region of the mesh each quadrilateral in it belongs to. It should be the same shape as `rt` and `zt`, but the first row and first column are constrained to be 0. (`ireg` is a cell-centered quantity, and it is its value at the upper right corner of the cell that decides the region number of the cell.) `ut` and `vt` are velocity components used to plot vector fields. `pvar` and `cindex` are mutually exclusive. `pvar` is a real array used to color a filled mesh, while `cindex` is an unsigned character array (Python typecode `'b'`) the numerical value of whose components specify an index into a color table. All arrays must be the same shape (`kmax` by `lmax`) except, possibly, `zt` and `rt`, as mentioned earlier. Once set, these variables will define the mesh until the next `set_mesh` command. Any variable undefined by `set_mesh` must be supplied as a keyword argument to a plot function call (if that function needs the variable) or must have been previously defined by a `set_mesh` call (or a call to one of the functions below).

`ezcx (<val>)` set the abscissa `(zt)` to `<val>.`
`ezcy (<val>)` set the ordinate `(rt)` to `<val>.`
`ezcpvar (<val>)` set the array of function values `pvar` to `<val>.`
`ezccindex (<val>)` set the array of color indices `cindex` to `<val>.`
`ezcireg (<val>)` set the array `ireg` defining the mesh regions to `<val>.`
`ezcu (<val>)` set the array `ut` of `y` velocity components to `<val>.`
`ezcv (<val>)` set the array `vt` of `x` velocity components to `<val>.`

Index

D

E

F

G

I

K

S

T

U

V