Gri

A Program to Make Science Graphs
Version 2.12.18
2007

Dan E. Kelley (dankelley@users.sourceforge.net)

Gri

Gri is an extensible plotting language designed for scientists. It can draw x-y plots, contour plots, and image plots, and has rudimentary programming capabilities. It is not mouse driven, nor gui-based; rather, it is an interpreted scriping language. Users regard it as an analogue to the latex document formatting language: users gain considerable power, at the price of a moderate learning curve.

This manual describes Gri version 2.12.18 (c) 1991-2007, Dan Kelley at email address dankelley@users.sourceforge.net

Gri is released with the GNU Public License (Chapter 19 [License], page 245).

1 Introduction

Gri is a programming language for drawing science-style graphs. It is not mouse-driven, and it does not draw business-style graphs (e.g. pie charts, three-dimensional graphs). Gri has substantial power in advanced applications. It has been proven to be easy to learn; for simple applications, the learning curve is less than an hour. Many users regard Gri as the plotting equivalent of the LaTeX document preparation system.

Computers Gri works on: unix computers of many types, plus Microsoft Windows, and Macintosh OSX. You'll find Gri pre-packaged for various unixes, e.g. linux/debian, linux/redhat, and freeBSD.

Capabilities of Gri are those scientists commonly want, since Gri was written by a scientist. It is not so useful for business people – e.g., Gri draws xy graphs (Chapter 5 [X-y Plots], page 21), contour plots (Chapter 6 [Contour Plots], page 25), and image plots (Chapter 7 [Images], page 31), but it will not draw pie-charts unless you teach it how. The list of capabilities of Gri is similar to many packages, but unlike many of the other packages, Gri gives you control over line widths, fonts, grayscales, etc. (Chapter 4 [Getting More Control], page 13), and it is a programming language of moderate power.

The Gri drawing metaphor is that of pen on paper. The ink in the pen is opaque. An item drawn in white ink will erase a previously drawn underlying object drawn in black ink. For example, to draw a timeseries curve in which the region between positive data values and the y=0 axis is filled with black ink, you might use (`draw curve filled`) to draw the timeseries with black ink (the default color), blackening the area between the curve and the lower axis. Then you could load white ink into the pen (using the `set graylevel 1` or `set graylevel white` command) and white-out a box drawn between the zero line and the lower axis. Then you'd load black ink back into the pen (`set graylevel 0`) and draw the curve again, so that the negative part would appear again.

Input/output in Gri may be interactive or non-interactive. For interactive use, type `gri` at the system commandline prompt. For non-interactive use, with Gri commands in a command-file called '`cmd.gri`', type `gri cmd.gri`.

Gri output is in the PostScript page description language. The output is therefore of high quality, device-independent, capable of being inserted into popular text processors (e.g. LaTeX), and easily displayed.

Online help: the Gri command `help` makes Gri list the first words of all known commands, along with a hint for getting further help. To get more information, type `help` followed by a command-name (e.g. `help read`). There is also a tiny bit of information stored online and categorized by topic. Get this by typing for example `help - strings` (Section 10.2 [Online Help], page 146).

Data analysis in Gri is limited. It has rudimentary data analysis functions, such as regression, column manipulation, smoothing, etc, but it is not intended as an integrated analysis/graphics package.

System calls are an easy and important facet of Gri. It is easy to use operating system commands within Gri (Section 9.3.50 [System], page 139; Section 10.16 [Operating System], page 183; Section 9.3.13 [Get Env], page 97). This allows you to use familiar, powerful tools, and keeps Gri simple. Particularly useful is the ability to read files through operating system filters (Section 9.3.28 [Open], page 102).

Programming Gri is quite straightforward, and users familiar with other programming languages find it easy. If Gri lacks a drawing method, you can add it fairly easily, since Gri has programming elements such as `if` statements (Section 10.6 [If Statements], page 157), `while` loops (Section 9.3.52 [While], page 141), facilities for interacting with the user (Sec-

tion 9.3.31 [Query], page 106), and mechanisms for storing numbers in "variables" (Section 10.4 [Variables], page 147), and text strings in "synonyms" (Section 10.5 [Synonyms], page 151). The Gri syntax can be augmented easily (Section 10.11 [Adding New Commands], page 173), and these augmentations can be stored in a startup file (Section 10.17 [Resource File], page 185), creating personalized versions of Gri.

Manuals: Gri has an online texinfo manual, a PostScript manual, a WWW manual, a cookbook and several reference cards. It also has several discussion groups (Section 11.3 [Discussion Group], page 190).

Version Numbering Scheme

When you launch Gri interactively (without naming a commandfile, i.e. by just typing `gri` at the unix prompt), you'll see something like

```
gri - scientific graphic program (version 2.12.18)
GPL Copyright 2007 by Dan E. Kelley.

Type 'help' for an overview of Gri commands, or see the
full manual at
    /usr/share/doc/gri-2.12.18/html/index.html
and its text-only version in the 'gri' INFO node.

Visit http://gri.sourceforge.net for updates and resources.
gri:
```

The last line is a prompt, suggesting that you type in Gri commands. You may type `quit` to get out of gri.

The first line gives the version number. You can also get this by running Gri with the command `gri -v`. Version numbers have three numbers separated by periods. The first number increments for major changes, the second for smaller changes, the third for still smaller changes. The second number also indicates whether a copy is an experimental version or a more reliable release version. Experimental versions have the second digit being an odd integer, while release versions have this digit being even.

2 Simple Gri Program and How to Run it

This chapter introduces Gri with a common example: an x-y graph. The example is discussed in detail later (Chapter 5 [X-y Plots], page 21). The data files and command files here and throughout the manual should be available to you in a directory '`.../gri/examples`' on unix machines.

2.1 Gri Command file

Here is a Gri command file to plot a linegraph of a set of (x,y) data, stored as space-separated columns in a file called '`example1.dat`':

```
# Example 1 -- Linegraph of data in separate file
open example1.dat     # Open the data file
read columns x y      # Read (x,y) columns
draw curve            # Draw data curve
draw title "Example 1"# Title above plot
```

The first line is a comment, as are all things following hash symbols (`#`). (An exception to this rule is made within strings contained within the double-quote character `"`. This allows `sed` system commands to work as expected; (Section 9.3.50 [System], page 139).)

The other lines are Gri command lines; (Chapter 5 [X-y Plots], page 21) for more explanation.

2.2 Data File

The data file '`example1.dat`' looks like:

```
0.05 12.5  # first point
0.25 19    # second point
0.5  15    # third point
0.75 15    # ... you get the idea!
0.95 13
```

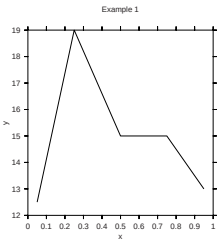
Note that spaces (or tabs) separate numbers. Any data line may have a comment on it, just as any command line may.

2.3 Running The Command File

Type '`gri example1.gri`' at the system prompt. Gri will create a PostScript file called '`example1.ps`'. For details on running Gri see Chapter 3 [Invoking Gri], page 7.

2.4 Output Graph

The output PostScript file is called '`example1.ps`'.



To view Gri output, use your favorite PostScript previewer.

Note that in the above example, Gri automatically chose reasonable scales for the axes, based on the range of the data. The next chapter illustrates that Gri also gives you control over such things.

3 Invoking Gri

3.1 Invoking Gri in a nutshell

First, the short story. In 90 percent of cases, Gri is run as

```
gri myscript
```

where the file '`myscript.gri`' holds a script (list of Gri commands), and Gri will create a PostScript file called '`myscript.ps`' with the output.

Some folks like to give the '`.gri`' suffix explicitly, so they would invoke Gri as

```
gri myscript.gri
```

instead.

If you'd rather not have '`myscript.ps`' as the PostScript output file name (let's say you prefer '`graph1.ps`') you'd do

```
gri -output graph1.ps myscript.gri
```

Few readers will need to know more than this. But, for the rest, the table in the next section gives full details on all the optional arguments that Gri can handle.

3.2 Using Gri to draw things

To draw things, invoke Gri as

```
gri [OPTIONS] [CmdFile [optional_arguments]]
```

where the square brackets indicate that the enclosed items are optional. The `OPTIONS` item may consist of one or more of the following (explained below):

```
[-batch]
[-b]
[-chatty N]
[-c      N]
[-debug]
[-d]
[-directory_default]
[-directory pathname]
[-help]
[-h]
[-no_bounding_box]
[-no_cmd_in_ps]
[-no_startup_message]
[-output PS_file_name|SVG_file_name]
[-private]
[-no_private]
[-publication]
[-p]
[-superuser N]
[-trace]
[-t]
[-yes]
[-y]
```

```
[-version]
[-v]
[-warn_offpage]
[-no_warn_offpage]
```

Here, the optional `optional_arguments` are a mechanism to customize the action of the given Gri script from the commandline. After Gri processes standard arguments (e.g. `-t` for tracing), it puts the remaining commandline arguments into a list. This behavior is borrowed from C and othe languages, so Gri borrows the name of the list as well: it's called the "arg" list, and its elements are available with the RPN operators named '`argc`' (Section 10.9.6 [Solitary Operators], page 168) and '`argv`' (Section 10.9.5 [Unary Operators], page 164).

For a note on usage within the Emacs gri-mode, see Section 12.7 [Filename arguments when running gri], page 203.

Details of command-line options

- `-batch` or `-b` Stops Gri from printing out prompts and hints.
- `-chatty N` or `-c N` Make Gri print out various informative messages. The numerical value gives a level of chattiness. A value of 1, the default if the `-chatty` code is not supplied, tells Gri to keep you informed of some important things, like the success in gridding data for contouring. Higher values make Gri tell you more:

  Information printed at various chatty levels:

  - **0** The bare minimum is printed. Thus invoking Gri as `gri -c 0...` will make it as quiet as can be.
  - **1 or higher** (the default) The full filenames of the commandfiles are displayed at startup time.

    `convert columns to grid` prints percentage of grid filled, as well as a suite of diagnostics, if you've let it calculate the region of influence automatically. It also prints a warning of the time it expects to take, before starting the calculation.

    `convert grid to image` prints characteristics of image created, including amount of image clipped.

    `read grid data` reports number of data values it could not read (since they were nonnumeric).

    `draw symbol` reports number of data points not drawn because they were missing or outside clip region (if one exists).
  - **2 or higher** `draw contour` prints value of contour being drawn.

    `open "...|"` prints the command to be passed to the operating system as well as the name of the temporary file being created; also notifies user when the temporary file is destroyed.

    `show image` reports histograms in intensity bands of 8 units, instead of the default 16 units.
  - **3 or higher** `show image` reports histograms in intensity bands of 4 units, instead of the default 16 units.
- `-debug` or `-d` Sets the built-in variable flag `..debug..` that you can use to isolate blocks of code.
- `-directory_default` Reports directory where '`gri.cmd`' is expected to be found, either in the default location or the one specified by `-directory` commandline option.

- `-directory pathname` Specifies the directory where Gri looks for the startup file '`gri.cmd`'. (This file teaches Gri the standard commands; Gri will report an error and die if it cannot find this file.) If this switch is not provided – and it is normally not – then Gri looks for '`gri.cmd`' in a standard system directory (sometimes, but not always, '`/usr/local/share/gri/2.12.18`') which was specified during the compilation of the Gri program itself. For more on how Gri looks for '`gri.cmd`', see the subsection below.
- `-no_bounding_box` Make the so-called "bounding box" in the PostScript file be the full page. The bounding box is used by some PostScript previewers to clip the image to just the drawn parts of the page, and is used by the `epsfbox` macro in `latex` to automatically determine the geometry of the graph for inclusion in text. Normally the bounding box is calculated automatically, to enclose all the items drawn on the page. But the box may also be set with the `set bounding box` command (Section 9.3.41.5 [Set Bounding Box], page 116).
- `-no_cmd_in_ps` Prevent Gri from inserting the lines of the commandfile into the PostScript file as comments. (These comments can be used by the `-creator` commandline option (see above), but they take up a little bit of space and users might sometimes want to get rid of them.)
- `-no_warn_offpage` Do not warn if items are offpage. (Contrast this with `-warn_offpage`.)
- `-output PS_file_name` Specify the PostScript filename. If this is not specified, the PostScript filename is derived from the name of the commandfile (e.g. '`mygraph.gri`' produces '`mygraph.ps`'), or, for interactive use, it will have a name like '`gri-00.ps`', or '`gri-01.ps`' if the former file exists, etc.
- `-output SVG_file_name` Specify the SVG filename. This is a pre-feature, as of version 2.12.x, meaning that SVG output is not working properly yet. If you specify an SVG file name, you will see a long list of warnings. These are debugging messages, and are not specific to your actual Gri script. For example, you will see warnings about centring strings, even if you are not centering any strings. This manual does not contain a list of working features (or broken features) for SVG output; the idea is that a discussion of such things be done using the bug-reporting system of the Gri website Section 15.2 [Reporting Bugs], page 227. In addition to bugs, the author is interested in users' opinions on the scheme of the SVG, especially the hiearchy of groupings of graphical elements. It is because such things are being altered that this is designated a pre-feature.
- `-no_startup_message` Stops Gri from printing the startup message.
- `-private` Prevents inserting any information about the user into the PostScript file (see `-no_private`, next). As of version 2.12.10, this privacy option is assumed by default.
- `-no_private` Instructs Gri to include comments in the PostScript file that identify the user, state the commandline arguments used in invoking Gri, and that list all the commands that were executed. This information can be recovered by calling Gri on the PostScript file, with the `-creator` commandline argument. Until version 2.12.10, the default was to include this information, but a change was made out of privacy concerns.
- `-publication` or `-p` Sets the built-in variable `..publication..` to 1. You may use this to distinguish between working plots and publication plots, as in this example:

```
if !..publication..
  draw time stamp
  draw title "working version of plot"
end if
```

- `-superuser` (This option is included here only for completeness. It should only be used by developers (who will alter the code to print debugging information if `-superuser` is set in addition to `-debug`). An optional value can be inserted (e.g. `-superuser 2`) to set the debugging level (retrievable by the function superuser()) to indicated integer value. Specifying the `-superuser` command-line option sets the built-in variable `..superuser..` to 1 or the specified value.)

  For flag meanings, see `superuser` command (Section 9.3.49 [Superuser], page 139). Using the question-mark symbol `?` instead of a flag number makes Gri print out the list of flags.
- `-trace` or `-t` Makes Gri print out command lines as they are executed; this has the same effect as the `set trace` command.
- `-version` or `-v` Display version information and exit successfully.
- `-warn_offpage` Causes warnings to be issued for all items drawn far off a 8.5x11 inch page. This is the default. (Contrast with `-no_warn_offpage`.)
- `-yes` or `-y` Bypasses all `query` commands, making Gri act as though the user typed a carriage-return (thus giving the default) for each `query`.
- `-help` or `-h` Prints explanation of options.
- `CommandFile` If a command file `CommandFile` is specified, then commands will be read from that file instead of the keyboard. If the `chatty` level is 1 or larger, Gri prints the names of the commandfiles at startup time. It is conventional but not necessary that the filename ends in `.gri`. If the filename does end in `.gri`, you may delete this suffix; Gri will assume it as implied.

Executable scripts. If you don't need to supply commandline options, you can put the following line as the first line in your Gri program

```
#!/usr/bin/gri
```

(or point to wherever Gri is located on your machine), and `chmod +x` the file. Then you can run Gri simply by naming the file. There is no particular advantage in this, except for saving the typing of a few characters, but some folks like this.

How Gri locates the '`gri.cmd`' **file**. In a normal installation, Gri finds the '`gri.cmd`' file all by itself. However, developers and some others may wish to control where Gri looks for this file. The rules below specify how Gri looks for '`gri.cmd`'.

Case 1 If `-directory` was given on the commandline used to invoke Gri (e.g. `gri -directory /some/place mycommand_file.gri`), then Gri will use the '`gri.cmd`' in the named directory. An error will result if '`gri.cmd`' is not found there.

Case 2 If `-directory` was not given on the commandline, then Gri looks for '`gri.cmd`' in a location that was specified during compilation. If '`gri.cmd`' is found there, then it is used. If it is not found, then Gri checks to see if an environment variable named `GRI_DIRECTORY_LIBRARY` is defined. If so, then Gri takes this to be the name of a directory that contains the '`gri.cmd`' file. If '`gri.cmd`' is not found there, an error results.

3.3 Extracting commandfile from a PostScript file

```
gri -creator PostScriptFile
```

See also `-no_cmd_in_ps`.

The `-creator` flag makes gri examine the indicate PostScript file, and produce a facsimile of the command file (or interactively-typed commands) that created this PostScript file. (This only works if the Gri command that created the PostScript file used the `-no_private` commandline argument.)

4 Controlling Axes, Fonts, Colors, etc

Gri provides a great many things that you can control, if you want to. An introduction to some of these things is presented in the sections below.

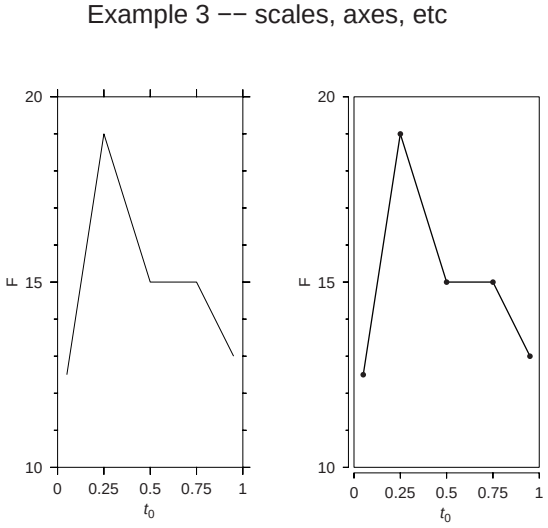
4.1 An example

Below is a followup to the previous example, which names the x and the y axes.

```
# Fancier version of Example 1
open example1.dat
read columns x y
set x name "Time, hours"
set y name "U, m/s"
draw curve
```

The difference is that the x and y axes are named with a `set` command. There are many `set` commands, and they are all pretty simple, e.g. `set x size 15` makes the x-axis be 15 centimeters long, instead of the default of 10 centimeters. Indeed, you can control anything you want in gri, e.g. graph size, line width, fonts, etc etc. Speaking of fonts, the `$\alpha$` type of latex formatting of Greek letters is supported in a limited way. Also, Gri handles ISO-Latin-1 encodings as well as the U.S. style.

The example below illustrates a few more `set` commands. This example is intentionally complicated, being about a good example of the level of complexity of many plots made by Gri. Read the comments to see what is being done, and consult the plot as you read the commandfile.




```
# Example 3 -- Controlling scales, etc
#
# Example of how to control axis scales, etc.  This example makes
# two panels, plotting the same data in different ways.
#
#
# ----- PANEL 1 ------------------------------------------------
#
# Set up the x axis.
#
# Make the x axis run from 0 to 1, with labelled tics each 0.25.
set x axis 0   1   .25
# Make the x-axis be 5 cm long; in other words, make the plot 5 cm wide.
set x size 5
# Put 2 cm of space between the left edge of the plot and the left
# edge of the paper.
set x margin 2
# Give the x-axis the name "t" with subscript 0.
set x name "$t_0$"
#
```

```
# Set up the y axis.
#
# Make the y axis run from 10 to 20, with labelled tics at intervals
# of 5 and smaller, unlabelled tics, at intervals of 1.  Other
# commands are similar to those for the x-axis.
set y axis 10 20 5 1
set y size 10
set y margin 2
set y name "F"
#
# Now, read our simple data set.
open example1.dat
read columns x y
close
#
# Draw a curve connecting these (x,y) data.  Note that the axes, as
# defined above, will be drawn automatically along with the curve.
draw curve


#
# ----- PANEL 2 -----------------------------------------------
#
# OK, now for a more complicated version.  We'll keep the same data, but
# redraw it in a new panel, to the right of the first graph.  So, the
# first step is to increase the x margin.  The {rpn} command simply
# creates a number which is the sum of the old x margin (stored in
# the variable ..xmargin..) and the old plot width (stored in
# the variable ..xsize..), plus an extra 1 cm
set x margin {rpn ..xsize.. ..xmargin.. + 1 +}
#
# Set the line thickness for the curve to 1 point (0.3 mm) and the
# axis line thickness to 0.2 points (0.1 mm).
set line width 1.0              # points
set line width axis 0.2         # points
# Set the tics to be 1.5 mm.
set tic size 0.15               # centimetres
# Draw axes and frame, with axes offset from frame.  Some
# people find this more attractive.
set axes style offset
draw axes 1
# Now draw the actual curve.
draw curve
# Superimpose dots (diameter 1.5 mm) at the data.
set symbol size 0.15
draw symbol bullet
#
# All done.
# Draw a title above the plot.
set font size 20
\label = "Example 3 -- scales, axes, etc"
```

```
draw label "\label" at          \
    {rpn 8.5 2.54 * "\label" width - 2 /} \
    {rpn ..ytop.. yusertocm 2 +}\
    cm
```

4.2 Axis scaling

Gri normally assumes that you are plotting scientific graphs, and therefore whenever it sees a command like `draw curve` or `draw symbol`, it draws an appropriate axis first. You can turn this feature off, by using `draw axes none` before the other `draw` command.

Furthermore, Gri picks axis scales by itself, by scanning the (`x`, `y`) columns. If you don't like the scales Gri picks, you can override them (Section 4.5 [Range], page 17).

Gri normally draws axes labelled at left and bottom, and with an axis frame with tics all around. If you don't like this default axis style you can specify other styles. For example, if the commands `draw x axis` and `draw y axis` are placed before the `draw curve` command, Gri will realize you've already specified axes, and just draw them on the left and bottom sides of the box, without completing the axis frame.

For your general use, Gri stores the minimum and maximum x and y values of the **axes** in the variables `..xleft..`, `..xright..`, `..ybottom..`, and `..ytop..`. It also stores the increments used in labelling these axes in the `..xinc..` and `..yinc..` variables.

To determine the minimum and maximum values of column data, you may use the built-in RPN functions `min`, `max`, and `mean` (Section 10.9.7 [Manipulation of Columns etc], page 169).

Gri stores the last (x,y) pair on a curve (whether data or axis) in the `..xlast..` and `..ylast..` variables

Gri stores the axis sizes in `..xsize..` and `..ysize..`. It stores the space to the left of the plot in `..xmargin..` and the space below the plot in `..ymargin..`.

4.3 Logarithmic and linear axes

Axes are linear by default; to make logarithmic axes, use commands `set x type log` and `set y type log`.

4.4 Axis Length

The axes are normally 10 centimetres long. To set the axis lengths (in centimetres), use commands like `set x size 5` and `set y size 7`. Some people like the ratio of axes to be in the so-called golden ratio `(root(5)-1)/2`; to get that, you could do this:

```
set x size 15
set y size {rpn ..xsize.. 5 0.5 power 1 - 2 / *}
```

For maps, you'll want the plot scaled so that shapes retain their aspect ratio. To do this, do `set x size .cm.` and then do `resize y for maps` (or vice versa).

4.5 Axis Range

To override axis ranges set by Gri, use `set x axis` and `set y axis`. With these commands, you specify the range of the axes; you may also set the interval for numbered tics, and an interval for unnumbered tics. The unnumbered tics must be at an interval that divides evenly into the numbered tic interval, but the numbered tic interval need not divide into the min/max range. Thus, `set x axis 0 1.1 0.5` will create an axis that will range from 0 to 1.1, with labelled tics at the values 0, 0.5 and 1.

4.6 Axis Name

To set the name of the x axis, use `set x name "string"`, and similarly for the y-axis. The default names are `x` and `y`.

4.7 Axis location

If you don't like the default position of axes (at left and bottom), you may get Gri to draw axes anywhere you like, using commands like `draw y axis at right` (so the y axis is at the right-hand end of the x range) or `draw x axis at top` (so the x axis is at the top of the plot); you may even specify an exact location, such as `draw x axis at 22.2`.

Normally, the x axis is placed at the bottom end of the y axis, and the y axis is placed at the left end of the x axis. Some people prefer a style in which the axes are positioned a small offset away from these locations. To get this effect, you may either position the axes yourself, or simply use the `set axes style offset` command (Section 9.3.41 [Set], page 115). If you want this axis style for all their plots, put the line `set axes style offset` in your '`~/.grirc`' startup file (Section 10.17 [Resource File], page 185).

4.8 Fonts

Fonts are selected with `set font to` (Section 9.3.41.20 [Set Font To], page 121) and font sizes are selected with `set font size` (Section 9.3.41.19 [Set Font Size], page 121).

Much more about text, including how to draw mathematical symbols, how to use subscripts and superscipts, how to write non-English (accented) European text, etc, is discussed (Section 10.10 [Text], page 170).

4.9 Colour of ink in pen

The darkness of the "pen" used in drawing commands (for either lines or for text) is set by `set graylevel .brightness.`. A brightness value of 0 corresponds to black ink, and a brightness value of 1 corresponds to white ink. Values outside this range are clipped to the nearer endpoint. Values inside this range choose a proportional graylevel in between; for example, `set graylevel 0.5` gives a 50 percent gray tone.

The graylevel applies to text as well as lines. Often you'll want to draw a gray line and a black label beside it, or you'll want to set a graylevel temporarily. Here's how to do it:

```
# Save old graylevl, set, then reset to old
.old_gray. = ..graylevel..
```

```
set graylevel 0.5
draw curve
set graylevel 0
draw label for last curve "TEST"
set graylevel .old_gray.
```

The color of the "pen" may be set to any value you can describe with an RGB (red, green, blue) or HSB (hue, saturation, brightness) specification, or a color name. This pen color applies to everything, even text.

The `set color \name` command

Set the pen color to the indicated name. There are two types of names: hexadecimal-triplet names and English names.

Hexadecimal-triplet names are of a form often used in web-pages. They consist of exactly 6 characters, which are divided by Gri into three sets of two characters, specifying the red component, the green component, and the blue component of the color, respectively. These components are in hexadecimal notation, i.e. ranging from 00 to FF, indicating values from 0 to 255. For example,

```
set color ACD4EF
```

sets a pastel blue color, almost the color of a robin's egg.

The English colors are written simply in the form

```
set color blue
```

where the color is from the following list. (Gri requires that you use the exact form shown, including the capitilization.) The color mixes are identical to those used in X11.

```
NAME                RED    GREEN  BLUE
"white"             1.000  1.000  1.000
"LightGray"         0.827  0.827  0.827
"darkslategray"     0.184  0.310  0.310
"black"             0.000  0.000  0.000
"red"               1.000  0.000  0.000
"brown"             0.647  0.165  0.165
"tan"               0.824  0.706  0.549
"orange"            1.000  0.647  0.000
"yellow"            1.000  1.000  0.000
"green"             0.000  1.000  0.000
"ForestGreen"       0.133  0.545  0.133
"cyan"              0.000  1.000  1.000
"blue"              0.000  0.000  1.000
"skyblue"           0.529  0.808  0.922
"magenta"           1.000  0.000  1.000
```

To get more colors than those provided in the above list, use the `read colornames` command.

You should do a test case for your printer to see which colors you find most to your liking. You'll want to pick colors that look different from each other. In some cases you might want to avoid dithered colors, since they look too broken on really thin lines. For example, on my printer I like the following colors: `black`, `red`, `yellow`, `green`, `cyan`, and `magenta`.

The `set color rgb .red. .green. .blue.` command

This command sets the color using the red-green-blue color model. If you are familiar with how colors add (e.g. red plus green yields yellow), then you might like this, but most people find it easier to use the `set color hsb ...` style described below.

Set the individual color components as follows. The numbers `.red.`, `.green.` and `.blue.` range from 0 (for no contribution of that color component to the final color) to 1 (for maximal contribution). Values less than 0 are clipped to 0; values greater than 1 are clipped to 1. EXAMPLES:

```
set color rgb 0   0   0  # black
set color rgb 1   1   1  # white
set color rgb 1   0   0  # bright red
set color rgb 0.5 0   0  # dark red
set color rgb 0   1   0  # pure green
set color rgb 1   1   0  # yellow: red + green
```

The `set color hsb .hue. .saturation. .brightness.` command

In this color model, the color (`"hue"`) is specified with a single parameter. Many people find this easier than using the corresponding `rgb` command.

Set the individual color components as follows. The numbers `.hue.`, `.saturation.` and `.brightness.` range from 0 to 1. The color, represented by `.hue.`, ranges from 0 for pure red, through 1/3 for pure green, and 2/3 for pure blue, and back to 1 again for pure red. (HINT: It is a good idea to limit the total range of hue you use to 2/3, instead of 1; otherwise you'll get confused by (nearly) repeated colors at the crossover. For example, limit the hue to range from 1/3 to 1, or 0 to 2/3.) The purity of the color, represented by `.saturation.`, ranges from 0 (none of the hue is visible) to 1 (the maximal amount is present). Less saturated colours are like those you would get from mixing black paint into colored paint. The brightness of the color, represented by `.brightness.`, ranges from 0 (black) to 1 (maximal brightness). Lowering brightness is like decreasing the intensity of the light you shine on a painting.

Hue, saturation, and brightness values are all clipped to the range 0 to 1. EXAMPLES:

```
set color hsb 0    1   1  # pure, bright red
set color hsb 0    1 0.5  # half black, half red
set color hsb .333 1   1  # pure, bright green
```

5 X-Y Plots

5.1 Linegraphs

The following Gri commands will draw a linegraph. For the output graph (Chapter 4 [Getting More Control], page 13).

This plots a simple linegraph:

```
# Example 1 -- Linegraph using data in a separate file

open example1.dat      # Open the data file
read columns x y       # Read (x,y) columns
draw curve             # Draw data curve
draw title "Example 1" # Title above plot
```

Here's what the command lines mean:

- The first line is a comment. Anything to the right of a hash-mark `#` is considered to be a comment. (This symbol is also called a "pound".)
- The second line is blank. Gri ignores blank lines between commands.
- `open example1.dat` tells Gri to open the indicated file (in the current directory) as an input data file. You can specify files outside the current directory by using conventional unix-shell pathnames (e.g., `open ~/data/TS/section1/T_S.dat` or `open ../data/file.dat`). You can even use "synonyms" (Section 10.5 [Synonyms], page 151.) in filenames, as in `open \BASENAME.dat`.
- `read columns x y` tells Gri to start reading columnar data, the first column being `x`, the second `y`. `x` and `y` are predefined names for whatever ends up on the horizontal and vertical axes.

  The number of data needn't be specified. Gri reads columns until a blank line or end-of-file is found. You can tell Gri how many lines to read with a command like `read columns 10 x y`. Multiple datasets can reside within one file; provided that they are separated by a single blank line, Gri can access them by multiple `read` commands.

  Like C, Gri expects numbers to be separated by one or more spaces or tabs. Commas are not allowed. If the columns were reversed, the command would be `read columns y x`. If there were an initial column of extraneous data, the command would be `read columns * x y`, or `read columns x=2 y=3` (Section 9.3.33.2 [Read Columns], page 107).
- `draw curve` tells Gri to draw a curve connecting the points in the `x` and `y` columns. A nice scale will be selected automatically. (You can change this or any other plot characteristics easily, as you'll see later.)
- `draw title` tells Gri to write the indicated string centered above the plot. The title **must** be enclosed in quotes.
- `quit` tells Gri to exit.

Gri will draw axes automatically, and pick its own scales.

If you wish to draw several curves which cross each other, you should try using `draw curve overlying` instead of `draw curve`. This will make it easier to distinguish the different curves.

5.2 Scattergraphs

This section contains two examples, the first being a fuller explanation of all the bells and whistles, the second being a simple explanation of how to get a very quick plot, given just a file containing a matrix of grid data.

To get a scattergraph with symbols at the data points, substitute `draw symbol` for `draw curve`. Both symbols and a curve result if both `draw curve` and `draw symbols` are used. See Chapter 4 [Getting More Control], page 13 for an example.

By default, the symbol used is an x. To get another symbol, use a command like `draw symbol 0` or `draw symbol plus`.

To change the symbol size from the default of 0.2 cm use commands like `set symbol size 0.1` to set to 1 mm (Section 9.3.41.41 [Set Symbol Size], page 129).

5.2.1 Coding data with symbols

To get different symbols for different data points, insert symbol codes from the above list as a column along with the x-y data, and substitute the command `read columns x y z`, and then draw them with `draw symbol`. Gri will interpret the rounded-integer values of the `z` columns as symbol codes. Note that even if you've read in a z column which you intend to represent symbols, it will be overridden if you designate a specific symbol in your `draw symbols` command; thus `draw symbol 0` puts a `+` at the data points whether or not you've read in a symbol column.

5.2.2 Drawing a symbol legend

The following example shows how you might write a symbol legend for a plot. The legend is drawn 1 cm to the right of the right-hand side of the axes, with the bottom of the legend one quarter of the way up the plot; Section 9.3.9.30 [Draw Symbol Legend], page 92. The lines in the legend are double-spaced vertically. To change the location of the legend, alter the `.legend_x. =` and `.legend_y. =` lines. To change the spacing, alter the `.legend_y. +=` line.

```
set x axis -1 5 1
set y axis -1 5 1
read columns x y z
0 0 0
1 1 1
2 2 2
3 3 3

draw symbol

# Legend
.leg_x. = {rpn ..xmargin.. ..xsize.. + 1 +}
.leg_y. = {rpn ..ymargin.. ..ysize.. 4 / +}
draw symbol legend 0 "Foo" at .leg_x. .leg_y. cm
.leg_y. += {rpn "M" ascent 2 *}
draw symbol legend 1 "Bar" at .leg_x. .leg_y. cm
.leg_y. += {rpn "M" ascent 2 *}
```

5.2.3 Coding data with symbol colors

To get different colors for different symbols, read a color code into the z column, and do for example `draw symbol bullet color hue z`. The numerical color code ranges from 0 (red) through to 1, passing through green at 1/3 and blue at 2/3.

5.3 Formula Plots

There are two methods for formula graphs.

1. **Use the system yourself.** Do as in this example:

```
open "awk 'BEGIN{for(i=0;i<3.141;i+=0.05)\
    {print(i,cos(i))}}' |"
read columns x y
close
draw curve
```

2. **Let Gri calculate things for you**

   The simplest is to let Gri calculate things for you with the `create columns from function` command (Section 9.3.5 [Create], page 80). The command assumes that you have defined the synonym called `\function` which defines `y` in terms of `x`.

   Gri uses the program `awk` to create the columns, and cannot work without it.

   Here is an example of using `create columns from function`:

```
show "First 2 terms of perturbation expansion"
set y axis name horizontal
set y name "sea-level"
set x name "$\omega$t"

\b = "0.4" # perturbation parameter b=dH/H
\xmin = "0"
\xmax = "6.28"
\xinc = "3.14 / 20"
\function = "cos(x)"
set x axis \xmin \xmax
create columns from function
draw curve
draw title "SOLID LINE  \function"

\function = "(cos(x)+\b/2*(1-cos(2*x)))"
create columns from function
set dash 1
draw curve
draw title "DASHED LINE \function"

draw title "b = \b"
```

   Here's another example, in which the curve `y = 1/(\int + \sl*x)` is drawn through some data. Note how `sprintf` is used to set `\xmin` and `\xmax` using the scales that Gri has determined in reading the data.

```
open file.data
read columns x y
```

```
close
draw symbol bullet
\int = "-0.1235"
\sl = "0.003685"
sprintf \xmin "%f" ..xleft..
sprintf \xmax "%f" ..xright..
\function = "1/(\int + x * \sl)"
create columns from function
draw curve
```

6 Contour Plots

Contour plots can be done with either pregridded data or randomly distributed (ie, ungridded) data.

6.1 Pre-gridded Data

This section presents two examples of contouring pre-gridded data, the first example illustrating a boilerplate program to contour data stored in a simple matrix form in a file, the second example illustrating a case with more control of the details (e.g., a nonuniform grid).

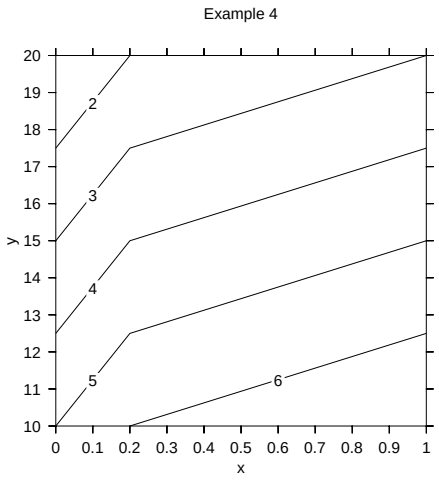
6.1.1 Simple example

This example was hardwired to know the size of the grid, etc. Here's an example which is more general, in that it determines the dimensions of the grid data from using unix system commands. Note that the grid is set to run from 0 to 1 in both x and y; you'll most likely want to change that after you see the initial plot, but this should get you started.

```
\file = "somefile.dat"
\rows = system wc \file      | awk '{print $1}'
\cols = system head -1 \file | awk '{print NF}'
set x grid 0 1 /\cols
set y grid 0 1 /\rows
open \file
read grid data \rows \cols
close
draw contour
```

6.1.2 Complicated example

To get a simple contour graph based on pre-gridded data, with full control of axes, etc, do something like this:




```
# Example 4 -- Simple contour graph

#
# Read x-grid; blank-line means stop reading.
read grid x
0
.2
1

# Note that the x-grid was irregular.  The y-grid
# in this example is regular, so we can just set
# it to range from 10 to 20, incrementing by 2.5.
```

```
set y grid 10 20 2.5
# Thus we now have a grid 3 wide and 5 high.  Let's
# read the actual data now.
read grid data
1 2 3
2 3 4
3 4 5
4 5 6
5 6 7

# Now draw contours (automatically set; we could
# have done 'draw contour 2' to draw contour for
# value 2 or 'draw contour 1 10 2' to draw contours
# ranging from 1 to 10 with an increment of 2.)
draw contour
draw title "Example 4"
```

Here several new things have been introduced.

First, you've got to define a grid in xy space. This example uses a non-uniform x-grid, and reads it in from the commandfile. In this form, the blank line is essential; it tells Gri that the end of data has been located; if you like, you can specify the number of lines to read, as in `read grid x 3`.

The y-grid for this example is uniform, however, so it may be specified with the `set y grid` command. It obtains values (10, 12.5, 15, 17.5, 20). The `set x|y grid` commands accept negative increments. Furthermore, it is possible to specify the number of steps, rather than the increment size, by putting `/` before the third number; thus `set x grid 0 1 /5` and `set x grid 0 1 0.2` are equivalent.

Having defined a grid, it is time to read in the gridded data. Here this is done with the `read grid data` command. Since Gri already knows the grid dimensions, it will read the data appropriately. You could also have told it (`read grid data 3 5`).

The first dataline is the top of the y-grid. In other words, the data appear in the file just as they would on the graph, assuming that the x-grid and y-grid both increase.

Sometimes you want to read in the transpose of a matrix. Gri lets you do that. If the `bycolumns` keyword is present at the end of the `read grid` command, the first dataline will contain the first **column**, of the data.

If you have an extraneous column of data to the left of your data matrix, do `read grid data * 2 3`

Now Gri has the grid in its head. We tell it to draw some contours with the `draw contour` command. As the comments in the example show, the contour values will be selected automatically, but you can alter that.

6.2 Ungridded data

When you have f=f(x,y) points at random x and y, you must cast them onto a grid to contour them. This is a difficult problem. There are many ways to grid data, and all have both good and bad features. You should try various methods, and various settings of the parameters of the methods. If you have a favorite gridding method that you prefer, you should probably pre-grid the data yourself. If not, Gri can do it for you. Gri has two methods for doing this, the "boxcar" method and the "objective analysis" method. Each

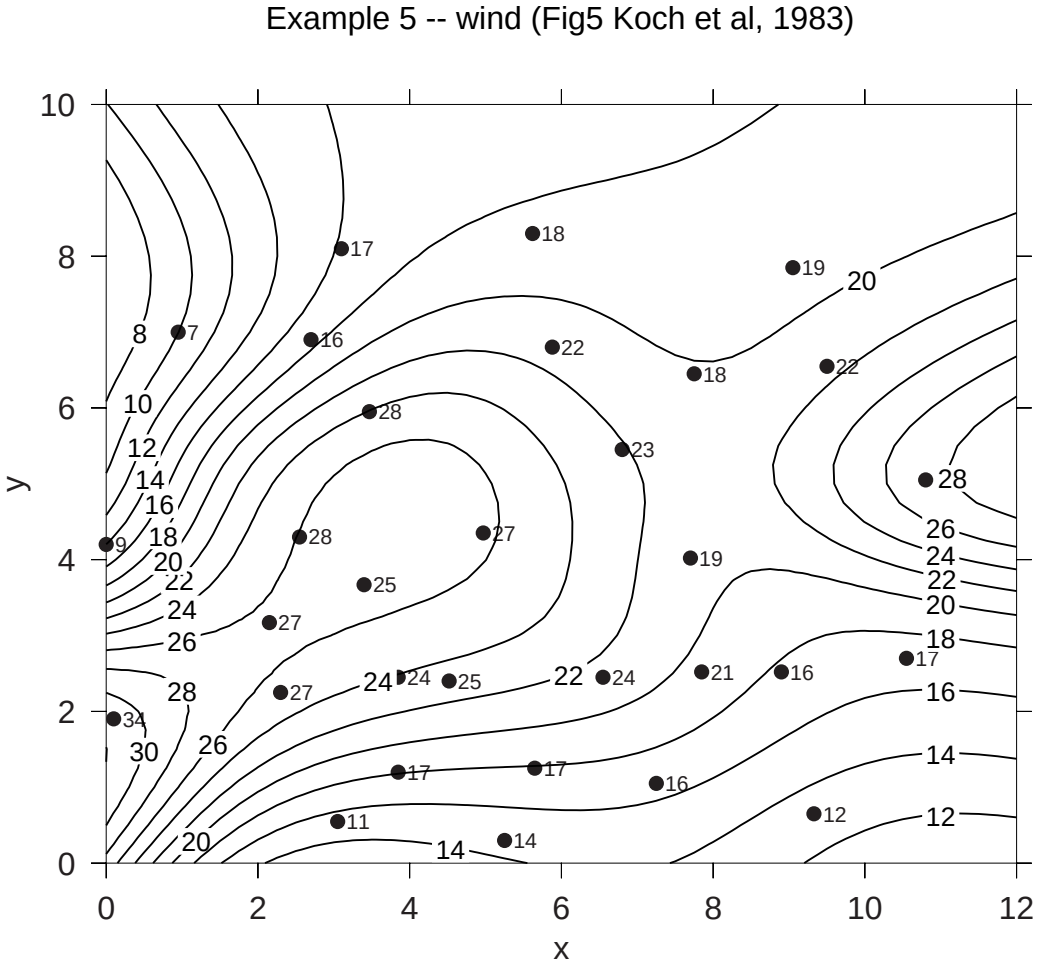
method puts holes in the grid wherever there are too few data to map to grid points, unless you specifically ask to fill in the whole grid.

The next two sections show first an example, then a discussion of the methods and how to use them.

6.2.1 Example

This example uses data taken from Figure 5 of S. E. Koch and M. DesJardins and P. J. Kocin, 1983. "An interactive Barnes objective map anlaysis scheme for use with satellite

and conventional data,", J. Climate Appl. Met., vol 22, p. 1487-1503. Readers should compare Figures 5 and 6 of that paper to the results shown here.



```
# Example 5 - Contouring ungridded data, from figure
# 5 of Koch et al., 1983, J. Climate Appl. Met.,
# volume 22, pages 1487-1503.
open example5.dat
read columns x y z
close
set x size 12
set x axis 0 12 2
set y size 10
set y axis 0 10 2
draw axes
set line width symbol 0.2
set symbol size 0.2
draw symbol bullet
set font size 8
draw values
set x grid 0 12 0.25
set y grid 0 10 0.25

# Use default method (Barnes)
convert columns to grid

set font size 10
draw contour 0 40 2
set font size 12
draw title "Example 5 -- wind (Fig5 Koch et al, 1983)"
```

6.2.2 Discussion of Methods

The various commands for converting columns to a grid are given in (Section 9.3.4.1 [Convert Columns To Grid], page 75). Generally, the Barnes method is best.

7 Image Plots

Gri can read in images stored in various formats. It can also create image data internally, by converting gridded data, which is quite handy in some contouring applications.

Note: if your diagram is to be reproduced by a journal, it is unlikely that the reproduction will be able to distinguish between any two graylevels which differ by less than 0.2. Also, graylevels less than 0.2 may appear as pure black, while those of 0.8 or more may appear as pure white. These guidelines are as specified by American Geophysical Union (publishers of J. Geophysical Res.), as of 1998.

7.1 Reading and Creating Image Data

Gri can do black and white image plots, such as satellite images. There are several ways to create image data in Gri

- Create images from gridded data using `convert grid to image`. For examples see Section 8.3 [Grayscale Images], page 43), Section 8.4 [Combination], page 46, and Section 8.2 [Contouring], page 39.
- Read raw ascii image data files. Use `read grid`.
- Read PGM (portable graymap) ascii files. (That is, a file with magic characters `P1` or `P3` at the start.) Use the `read image pgm` command, for a file opened in ascii mode with `open filename`.
- Read raw binary data, with or without headers. Use `read image`, after skipping any header bytes using the `skip` command, for a file opened in binary mode with `open filename binary`.
- Read a Sun "rasterfile" file (but only in uncompressed form). Use `read image rasterfile` for a file opened in binary mode with `open filename binary`.
- Read a PGM (portable graymap) binary file. (A file with magic characters `P2` or `P4` at the start.) Use the `read image pgm` for a file opened in binary mode with `open filename binary`.
- Aside: Images can be converted to grids (for contouring) using `convert image to grid` (Section 9.3.4 [Convert], page 74).

Once the image is created, its grayscale/colorscale may be manipulated with the commands `set image grayscale` and `set image colorscale`, which permit linear and histogram-equalized blendings over the grayscale or color range, or with `read image grayscale` and `read image colorscale`, which permit reading in the grayscale or color values individually, one for each of the 256 pixel values.

It is important to understand the structure of image data. Gri works only with 8-bit image data. This means that a given pixel of the image may have only one of 256 possible values. The example below uses a satellite image of surface temperature. The suppliers of the data dictate that pixel value 0 corresponds to a temperature of 5C, and a pixel value of 255 corresponds to 30.5C, so the resolution is 0.1C per pixel value. This resolution will be apparent if the output of the example below is previewed on a grayscale/color monitor — notice the quantization in the palette. This resolution issue is not very important with satellite images, since you have to use what you are given by the suppliers of the data. However, the issue is very important when you are converting grid data to images. When Gri converts grid data to image data, it neccessarily discards information, because the grid data have resolution to about 6 digits, whereas the image data have only 8-bit (2-3 digit) resolution. The `set image range` commands determines the range of this 8-bit resolution in

terms of user units. All other things being equal, it would be preferable to use the smallest range consistent with the range of your data. If your grid data ranged from 0 to 1, say, you might `set image range 0 1`. This would give a resolution in the image of 1/255 in the user units. But, when Gri converts the grid into an image, it will **clip** all data outside the indicated range. In this case, any data greater than 1 in the grid would translate to **exactly** 1 in the image. Naturally there is a tradeoff between having a range large enough to encompass any data in the grid, and a range small enough to yield adequate resolution. In most cases, 8-bit resolution will be adequate, but it is good to be aware of the limitations. One should always `draw image palette`, and check it on a color monitor for bandedness, which is a sign of resolution problems.

7.2 About The PostScript Output

Programmers Note: Gri inserts some special comments in the PostScript file, to help programmers extract the image data; to extract the information, you'll have to understand how PostScript handles images. Gri inserts a single comment line before a line ending in the token `im`:

```
%BEGIN_IMAGE
170.70 170.70 534.86 534.86 128 128 im
```

The first four numbers are the (x,y) locations of the lower-left and upper-right corners of the image, in units of points on the page (72 points = 1 inch). The fifth and sixth numbers are the width of the image and the height of the image. The keyword `im` is always present on this line. Gri inserts the following comment line at the end of the image data

```
%END_IMAGE
```

7.3 Example (Satellite image)

Here's an example that will plot different types of images, depending on your answers to `query` questions. The file called '`\filename`' is the data file, in binary format with one byte (`unsigned char` in C) for each pixel, stored with the northwest pixel first, and the pixel to the east of that next. The file called `\mask` is in the same format, and the numbers are 0 if the point is over the sea and 1 if over land. The mask file is used in computing the histograms, which is done if `\histo` is 1.

The file in this example covers 128 * 128 pixels over the Gulf of Maine. The numbers in `\filename` correspond to surface temperatures according to the equation

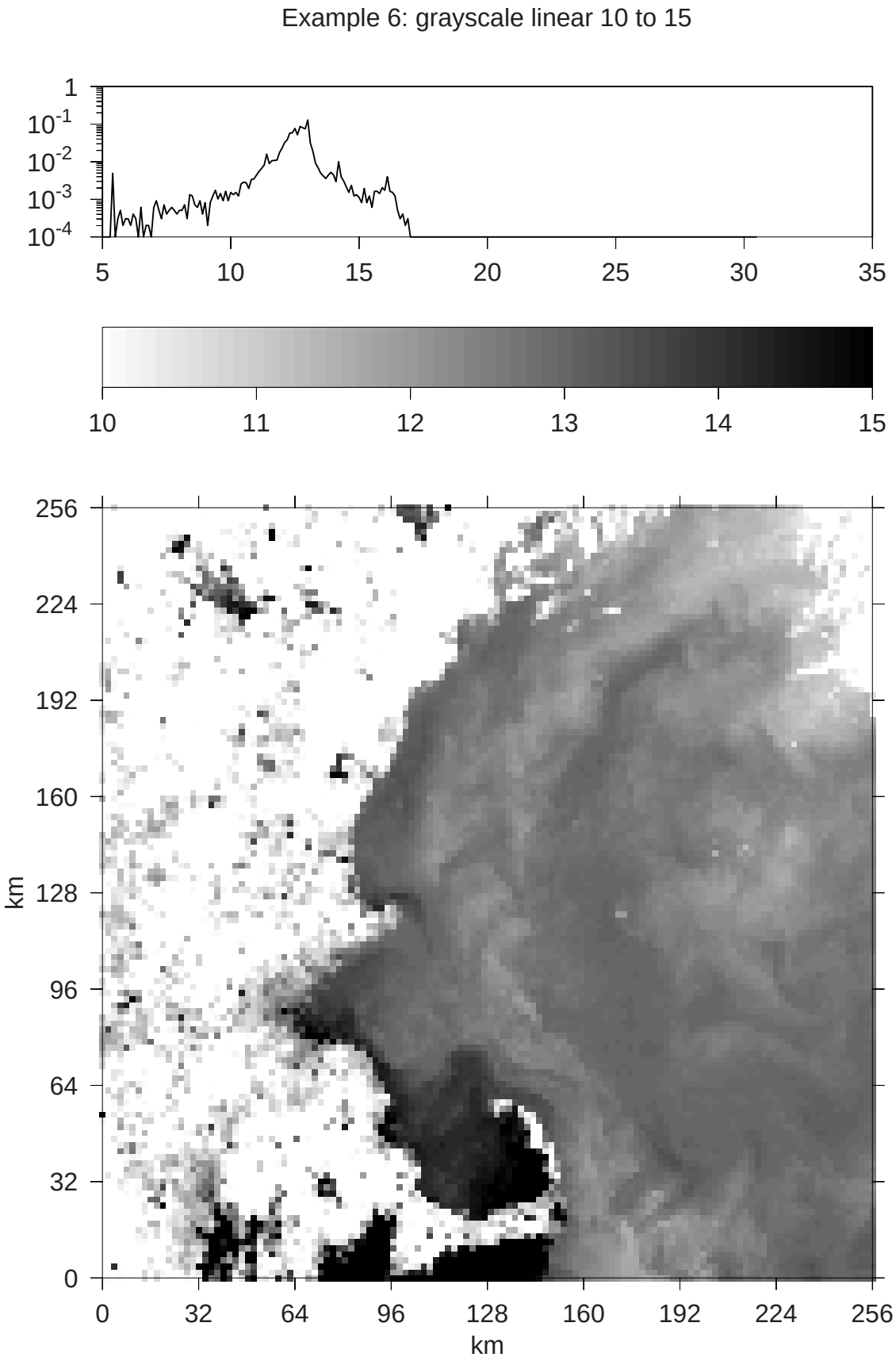
```
T = 5 + 0.1 * pixel_value
```

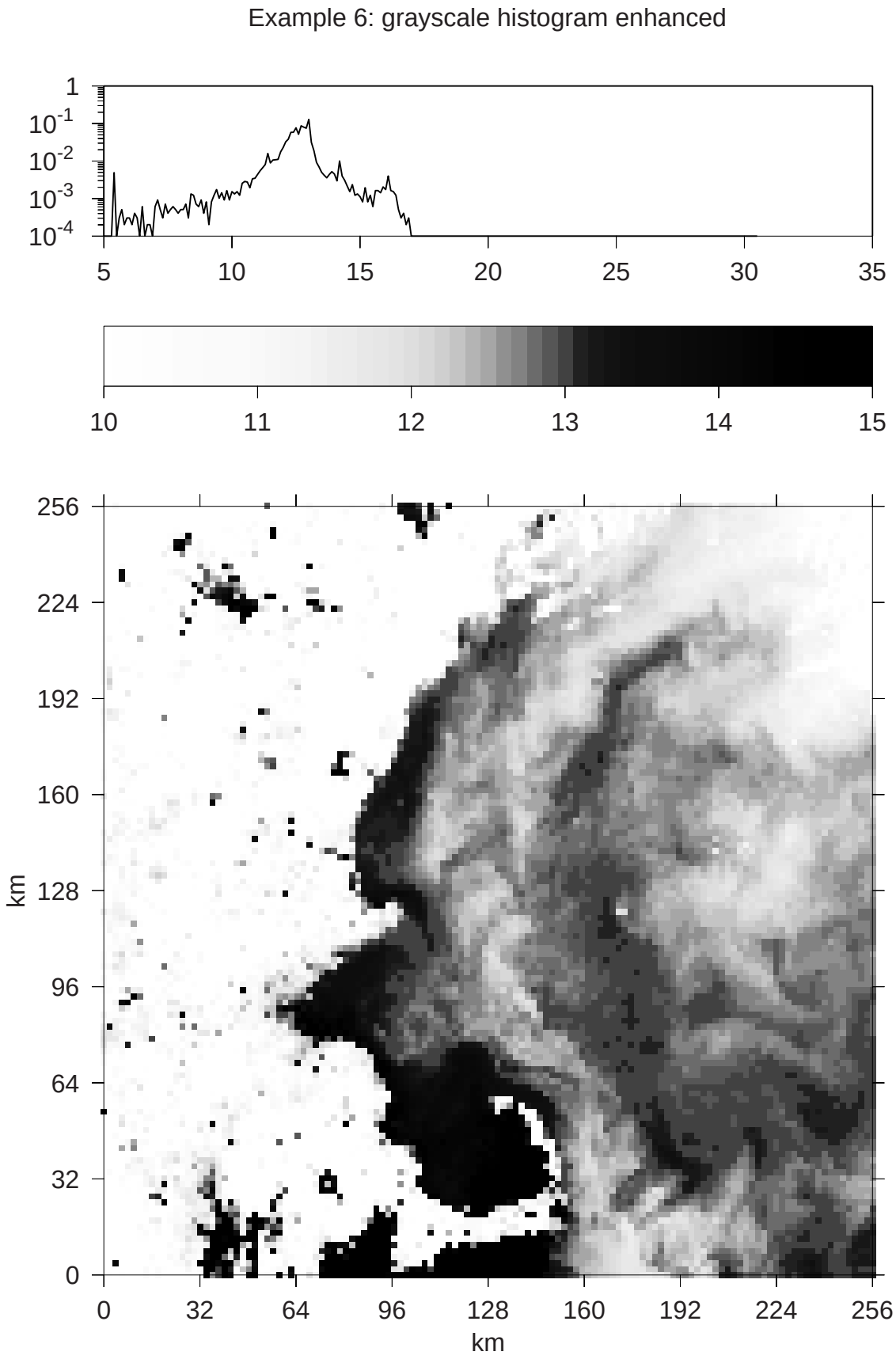
which explains the following lines in the command file:

```
\0val = "5"             # 0 in image
\255val = "30.5"        # 255 in image
```

Depending on `\histo`, the graymap will be linear or histogram-enhanced. The histogram method consists of dividing the cumulative histogram for the values in the image up into 256 levels, and assigning a graylevel to each. This has the effect of creating maximal contrast in all ranges of graylevel. It points up features really well, but it is a nonlinear mapping, so it is not good for telling you where gradients are strong or weak.

Examples are shown for linear mapping and histogram mapping.




Example 6: grayscale histogram enhanced
1
10-1
10-2
10-3
10-4
5
10
15
20
25
30
35
10
11
12
13
14
15
256
224
192
160
128
96
64
32
0
km
0
32
64
96
128
160
192
224
256
km

```
# Example 6 -- Plot IR image of Gulf of Maine
# Define characteristics of norda images
# Note that the pixel to temperature conversion formula is
#
#   Temperature = 5C + pixel_value / 10
#
# where pixel_value ranges from 0 to 255.  Thus, a pixel value of 0
# corresponds to a temperature of 5C, and 255 corresponds to 30.5C;
# this is why the limits \0val and \255val, for use by the ‘set image
# range’ command, take on these values.
\0val = "5"                     # 0 in image
\255val = "30.5"                # 255 in image
.rows. = 128
.cols. = 128
.pixel_width. = 2
.km. = {rpn .cols. .pixel_width. *}

# get filenames
query \filename "Name image file" ("example6image.dat")
query \maskname "Name mask  file" ("example6mask.dat")

# get data, then mask, each in 8-bit image format
open \filename 8bit
set image range \0val \255val
read image .rows. .cols. box 0 0 .km. .km.
close
open \maskname 8bit
read image mask .rows. .cols.
close

# find out what grayscale method to use
query \histo "Do histogram enhancement? (yes|no)" ("no")
query \minT  "T/deg for white on page?          " ("10")
query \maxT  "T/deg for black on page?          " ("15")
\incT = "1"

# set up scales.
set x size 12.8
set y size 12.8
set x name "km"
set y name "km"
set x axis 0 .km. 32
set y axis 0 .km. 32

# plot image, grayscale, and histogram
if {"\histo" == "yes"}
    set image grayscale using histogram black \maxT white \minT
else
    set image grayscale black \maxT white \minT
end if
draw image
```

```
draw image palette left \minT right \maxT increment \incT
draw image histogram
if {"\histo" == "yes"}
    draw title "Example 6: grayscale histogram enhanced"
else
    draw title "Example 6: grayscale linear \minT to \maxT"
end if
```

8 Real-world examples

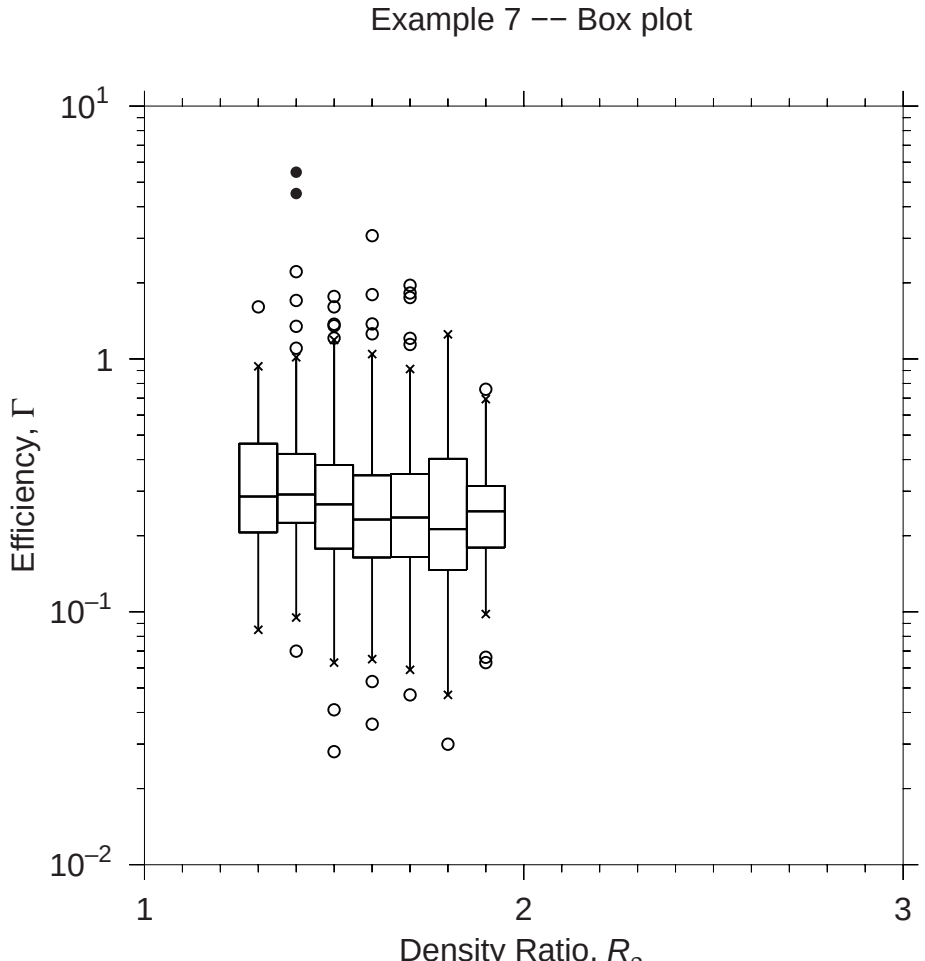
The example files in this manual should be available to you directly, having been installed with Gri; if not, ask your system manager to check the FTP site.

Additionally, I've collected a few real life examples here. Other sources are the Gri cookbook, available at `http://gri.sourceforge.net/gri-cookbook/index.html`.

8.1 Box plots

Box plots were invented by Tukey for eda (exploratory data analysis). They show nonparametric statistics. The centre of the box is the median. The box edges show the first quartile (q1) and the third quartile (q3). The distance from q3 to q1 is called the inter-quartile range. The whiskers (lines with crosses on them) extend to the furthest points still within 1.5 inter-quartile ranges of q1 and q3. Beyond the whiskers, all outliers are shown, in open circles up to a distance of 3 inter-quartile ranges beyond q1 and q3, and in closed circles

beyond that. Below is an example that uses a "new command" to define each box plot (Section 10.11 [Adding New Commands], page 173).



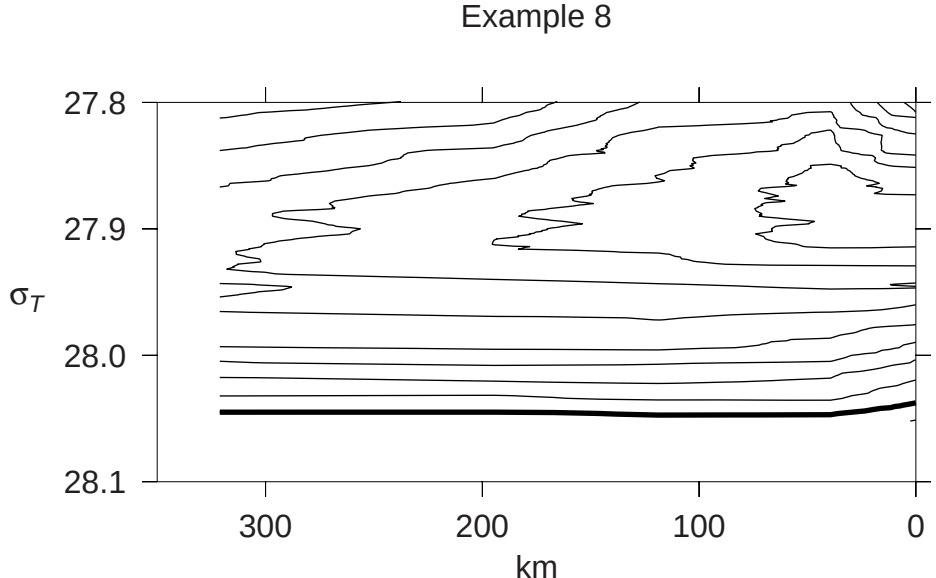
```
# Example 7 -- Box plots of mixing efficiency vs density ratio (meddy)

‘Draw y boxplot from \file at .x.’
Draw a y boxplot for data in given file, at given
value of x.
{
    open \.word4.
    read columns * y
    close
    draw y box plot at \.word6.
}
if !..publication..
    draw time stamp
end if
set x axis 1 3 1 0.1
set x name "Density Ratio, $R_\rho$"
set x margin 4
set y axis -2 1 1
#
# Must fool gri into not drawing the axes, because the y data
# are already in logspace.
draw axes none
Draw y boxplot from example7a.dat at 1.3
Draw y boxplot from example7b.dat at 1.4
Draw y boxplot from example7c.dat at 1.5
Draw y boxplot from example7d.dat at 1.6
Draw y boxplot from example7e.dat at 1.7
Draw y boxplot from example7f.dat at 1.8
Draw y boxplot from example7g.dat at 1.9
delete y scale
set y name "Efficiency, $\Gamma$"
set y type log
set y axis 0.01 10 1
draw axes
draw title "Example 7 -- Box plot"
```

8.2 Contouring

This example plots a section of dT/drho vs x and rho (actually, sigma-t, as the label indicates). The contours are unlabelled; I’m only interested in the zero crossings.

There are some other useful tricks in this example, such as calling `awk` and `wc` from the unix system.

(In the plot shown, all `query` questions were answered with carriage return, yielding the defaults; the `-p` flag was specified on execution, so the time stamp was not drawn.)



```
# Example 8 -- Plot T=T(x,rho) section of eubex data

‘Initialize Parameters’
{
    \FILE_DATA = "example8a.dat" # T vs rho
    \FILE_LOCN = "example8b.dat" # section distances
    set missing value -99.0
    #
    # Following values from ~/eubex/processing/to_rho_bins/do_rho_inter
    \RHO_MIN = "28.1"
    \RHO_MAX = "27.5"
    \RHO_INC = "-0.002"
    \NY = "301"
    \xmin = "350"
    \xmax = "0"
    \xinc = "-100"
    \ymin = "28.1"
    \ymax = "27.8"
    \yinc = "-0.1"
    \zmin = "0"
    \zmax = "2.5"
}
‘Initialize Axes’
/*
Set up axes
*/
{
    set x name "km"
    set x size 10
    set x axis \xmin \xmax \xinc
    set y name "$\sigma_T$"
    set y size 5
    set y axis name horizontal
    set y axis \ymin \ymax \yinc
    set y format "%.1f"
}
‘Initialize Files’
{
    query \data "Data file?   " ("\FILE_DATA")
    query \locn "Station locn?" ("\FILE_LOCN")
}
‘Read Data’
{
    # Read x-locations
    system awk ’{print $2}’ < \locn > TMP
    system wc TMP | awk ’{print $1}’ > NUM
    open NUM
    read .gridx_number.
    close
    system rm NUM
    open TMP
```

```
    read grid x .gridx_number.
    close
    system rm TMP
    # Create y-locations
    set y grid \RHO_MIN \RHO_MAX \RHO_INC
    #
    # Read data
    open \data
    read grid data \NY .gridx_number.
    close
}
‘Plot Contours’
{
    set graylevel .contour_graylevel.
    set clip on
    set line width 0.5
    draw contour -3 3 0.25 unlabelled
    #
    # wide line at 0 degrees
    set line width 2
    draw contour 0 unlabelled
}
‘Plot Image And Maybe Contours’
{
    \imagefile = "image"
    set image range \zmin \zmax
    convert grid to image box \xmin  \ymin \xmax \ymax
    query \dohisto "Do histogram scaling? (yes|no)" ("yes")
    \incs = "no"
    if {"\dohisto" == "yes"}
set image grayscale using histogram
    else
\zinc = "0.25"
query \incs "In linear scaling, band at an increment of \zinc?" ("yes")
if {"\incs" == "yes"}
    set image grayscale black \zmin white \zmax increment \zinc
else
    set image grayscale black \zmin white \zmax
end if
    end if
    write image rasterfile to \imagefile
    show "wrote image rasterfile ‘\imagefile ’"
    draw image
    draw image palette
    query \do_contours "Do contours as well (yes|no)" ("yes")
    if {"\do_contours" == "yes"}
Plot Contours
    end if
    draw title "Example 8 -- \data black=\zmin white=\zmax"
    if {"\dohisto" == "yes"}
draw title "Histogram enhanced grayscales"
```

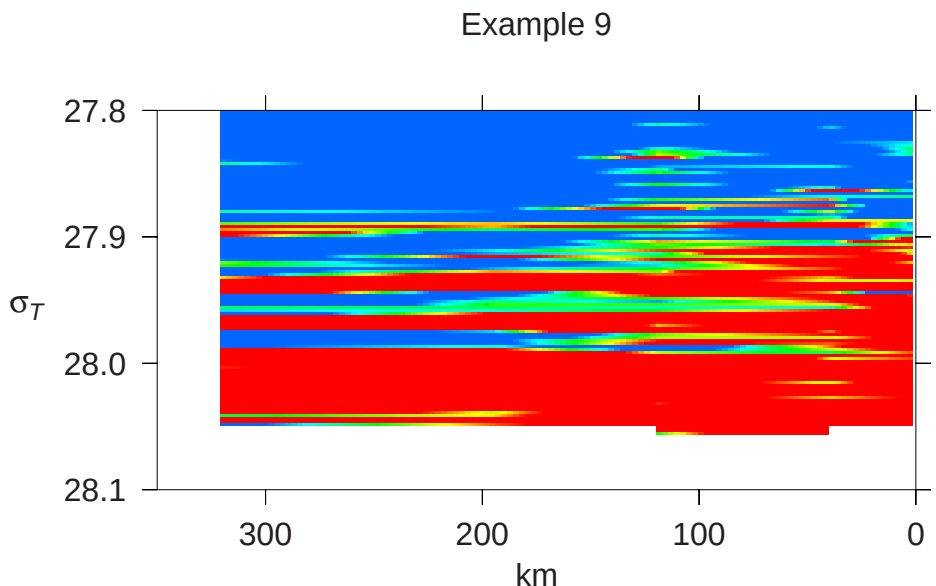
```
    else
if {"\incs" == "yes"}
    draw title "Grayscale banded at intervals of \zinc"
end if
    end if
}
Initialize Parameters
Initialize Axes
Initialize Files
Read Data
query \doimage "Draw image (yes|no)" ("no")
if {"\doimage" == "yes"}
    .contour_graylevel. = 1 # white contours
    Plot Image And Maybe Contours
else
    .contour_graylevel. = 0 # black contours
    Plot Contours
    draw title "Example 8"
end if
```

8.3 Image created from coarsely gridded data

This example reads gridded ascii station data (`Read Data`), creates an interpolated image (`convert grid ...`), and then plots the image.

There are some other useful tricks in this example, such as calling `awk` and `wc` from the unix system.

(In the plot shown, all `query` questions were answered with carriage return, yielding the defaults; the `-p` flag was specified on execution, so the time stamp was not drawn.)



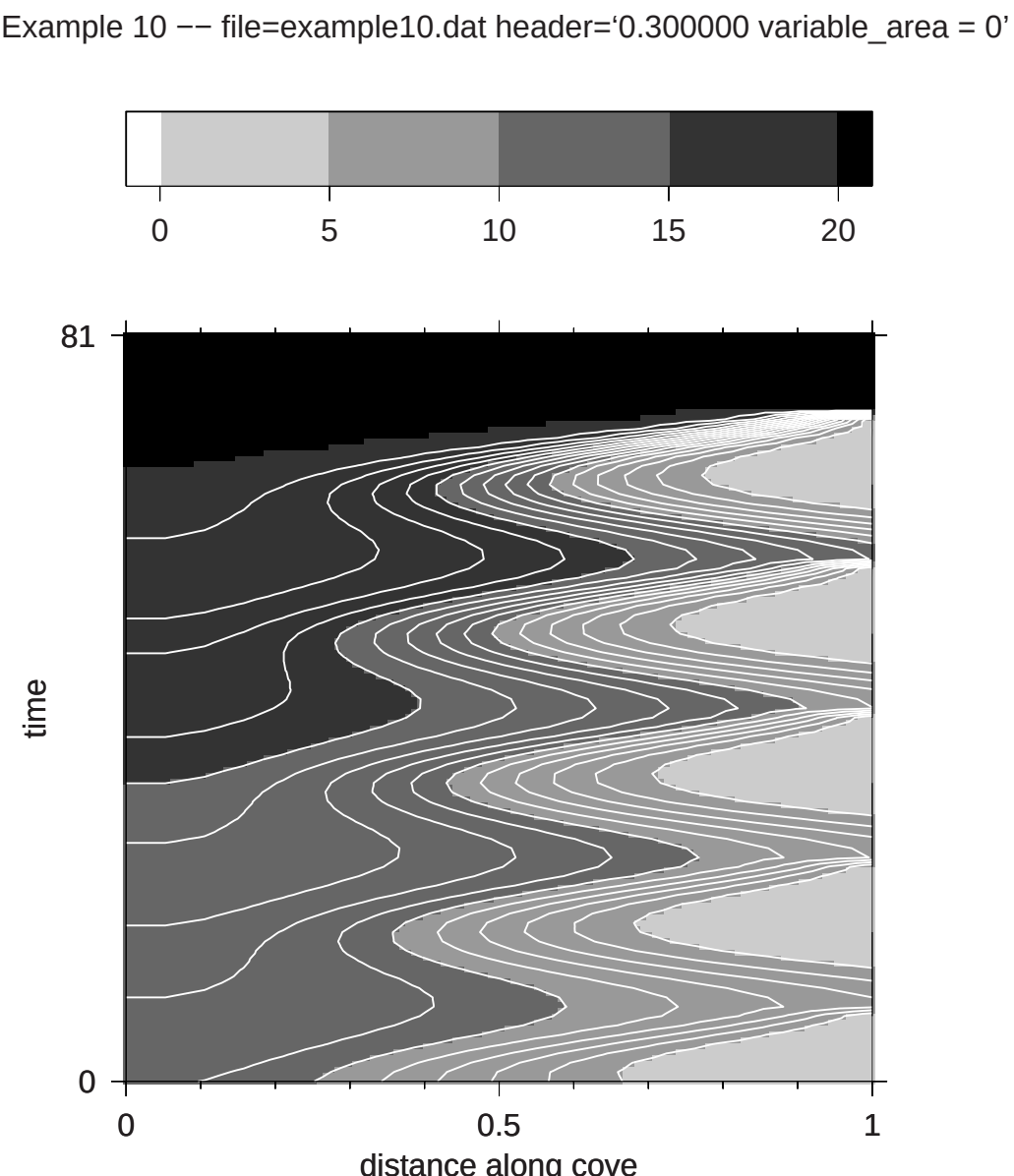
```
# Example 9 -- Plot dTdrho-rho section

'Initialize Parameters'
{
    \FILE_DATA = "example9a.dat" # T vs rho
    \FILE_LOCN = "example9b.dat" # section distances
    #
    # Following values from ~/eubex/processing/to_rho_bins/do_rho_inter
    \RHO_MIN = "28.1"
    \RHO_MAX = "27.5"
    \RHO_INC = "-0.002"
    \NY = "301"
    set missing value -99.0
    \xmin = "350"
    \xmax = "0"
    \xinc = "-100"
    \ymin = "28.1"
    \ymax = "27.8"
    \yinc = "-0.1"
    \zmin = "-10" # black
    \zmax = "0" # white
}
'Initialize Axes'
Set up axes.
{
    set x name "km"
    set x size 10
    set x axis \xmin \xmax \xinc
    set y size 5
    set y name "$\sigma_T$"
    set y axis name horizontal
    set y axis \ymin \ymax \yinc
    set y format %.1lf
    draw axes none
}
'Initialize Files'
{
    query \data "Data file?   " ("\FILE_DATA")
    query \locn "Station locn?" ("\FILE_LOCN")
}
'Read Data'
{
    # Read x-locations
    system awk '{print $2}' < \locn > TMP
    system wc TMP | awk '{print $1}' > NUM
    open NUM
    read .gridx_number.
    close
    system rm NUM
    open TMP
    read grid x .gridx_number.
```

```
    close
    system rm TMP
    # Create y-locations
    set y grid \RHO_MIN \RHO_MAX \RHO_INC
    #
    # Read data
    open \data
    read grid data \NY .gridx_number.
    close
}
Initialize Parameters
Initialize Axes
Initialize Files
Read Data
set image range \zmin \zmax
set image colorscale hsb 0 1 1 \zmin   hsb .6 1 1 \zmax
convert grid to image box \xmin  \ymin \xmax \ymax
#
# Draw the image, then draw the axes.  Note that the image has
# extends beyond the axes frame, so we will turn clipping
# on before drawing it, to make a clean picture.
set clip postscript on
draw image
set clip postscript off
draw axes

#
# All done.
draw title "Example 9"
if {"\dohisto" == "yes"}
    draw title "Histogram enhanced grayscales"
end if
```

8.4 Combination of image and contour

The following example reads gridded data and creates an image as in the previous example, but also superimposes unlabelled white contour lines.

(In the plot shown, all `query` questions were answered with carriage return, yielding the defaults; the `-p` flag was specified on execution, so the time stamp was not drawn.)

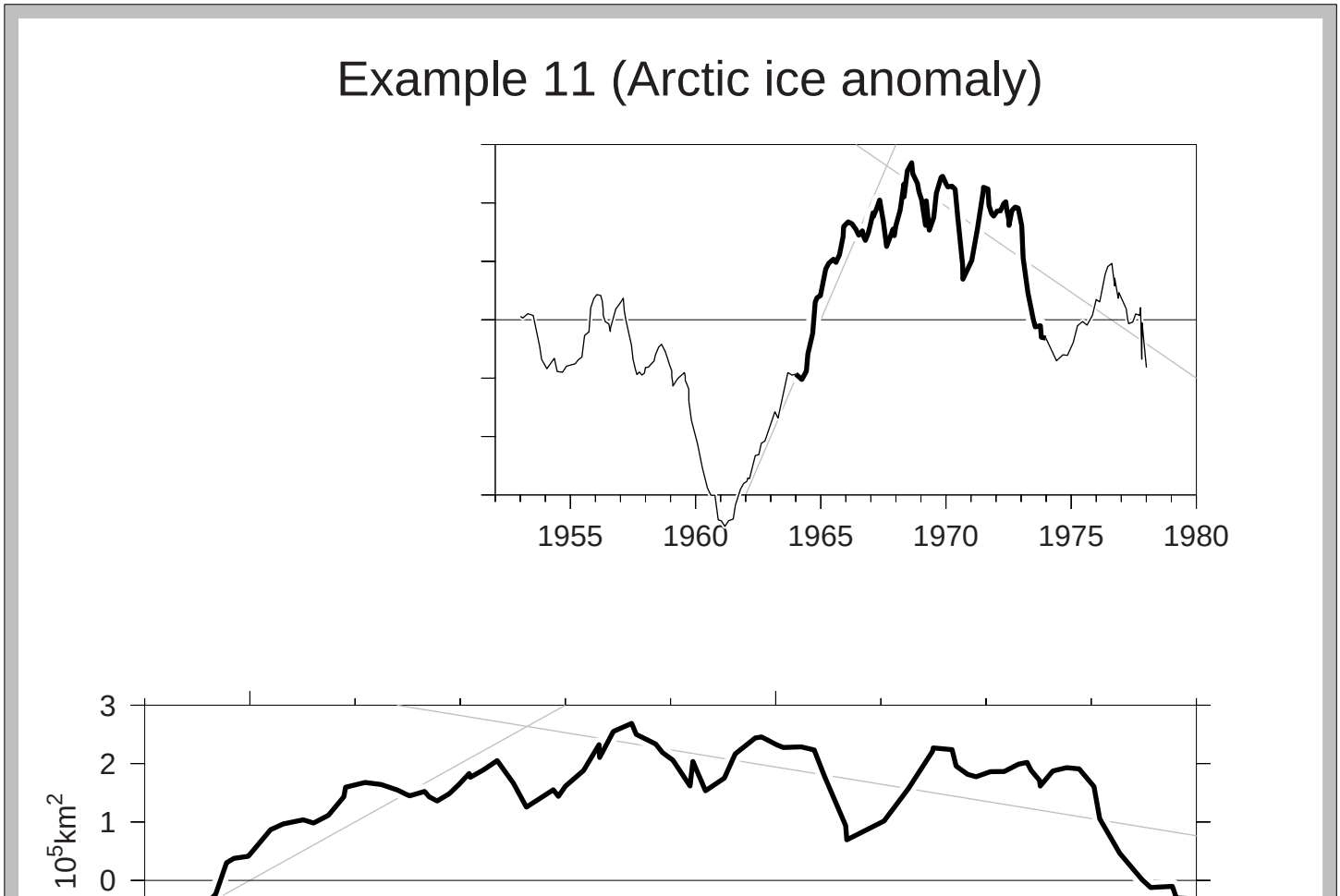


```
# Example 10 -- Draw image plot of flushing of dye out of cove
if !..publication..
    draw time stamp
end if
\file = "example10.dat"
query \contours "Superimpose contours? (yes|no)" ("yes")
query \file     "Input file name               " ("\file")
open \file
read line \header
read \D
read .nx.
read .ny.
set x name "distance along cove"
set y name "time"
set x grid 0 1 /.nx.
set x axis 0 1 0.5 0.1
set y grid 0 .ny. / .ny.
set y axis 0 .ny.
read grid data * * .ny. .nx.
set image range 0 20
set image grayscale black 20 white 0 increment 5
convert grid to image
draw image
if {"\contours" == "yes"}
    set graylevel 1.0
    draw contour 0 20 1 unlabelled
    set graylevel 0.0
end if
draw axes
draw image palette left -1 right 21 increment 5
draw title "Example 10 -- file=\file header=‘\header’"
# Example 10color -- Draw color image plot

# Test various colorscales.
# INSTRUCTIONS: Uncomment one of the following ’\scale = ’ statements

# CASE 1: From black at high values to white at low values
#\scale = "rgb 0 0 0 20.0 rgb 1 1 1  0.0 increment 5"

# CASE 2: From skyblue at 20 to tan for 0; traverse RGB space
#         See also case 5, which names the colors.
#\scale = "rgb 0.529 0.808 0.922 20.0 rgb 0.824 0.706 0.549 0.0 increment 5"

# CASE 3: From skyblue at 20 to tan for 0; traverse HSB space
#         Is it just me, or is this uglier than case 2?
#\scale = "hsb 0.548 0.426 0.922 20.0 hsb 0.095 0.334 0.824 0.0 increment 5"

# CASE 4: Use a spectrum; traverse HSB space
#\scale = "hsb 0 1 1 20.0 hsb 0.6666 1 1  0.0 increment 5"

# CASE 5: From skyblue to tan, traversing RGB space (by default)
```

```
#         (Compare case 2, which uses similar endpoints, with
#         colors specified with RGB values, and larger increment.)
#\scale = "skyblue 20.0 tan 0.0 increment 2"

# CASE 6: From skyblue to tan, traversing RGB space (by default)
#         Compare 2 and 5; note this has continuous increment
#\scale = "skyblue 20.0 tan 0.0"

# CASE 7: From blue to brown
\scale = "blue 20.0 brown 0.0 increment 2.5"

open example10.dat
read line \header
read \D
read .nx.
read .ny.
set x name "distance along cove"
set y name "time"
set x grid 0 1 /.nx.
set x axis 0 1 0.5 0.1
set y grid 0 .ny. / .ny.
set y axis 0 .ny.
read grid data * * .ny. .nx.
set image range 0 20
convert grid to image
set image colorscale \scale
draw image

# Draw contours in white ink
set graylevel 1.0
draw contour 0 20 1 unlabelled
set graylevel 0.0

draw axes                                  # redraw in case whited out
draw image palette left -1 right 21 increment 5
set font size 9

# Title tells what method used
draw title "Used ‘draw image colorscale \scale’"
```

8.5 Fancy x-y linegraph

The following code shows a fancy plot with lots of bells and whistles.



```
# Example 11 -- Fancy plot
# Pen sizes, etc.
#
.thin. = 0.5                    # for whole data set
.thick. = 2                     # for bravo time period
.gray_for_guiding_lines. = 0.75 # for guiding lines
.tmin. = 1964                   # time axis
.tmax. = 1974
.tinc. = 5
.tincinc. = 1
.missing_value. = -9
\file = "./example11.dat"
#
# Guiding lines to draw on both panels.
#
.1xl. = 1962
.1yb. = -3
.1xr. = 1968
.1yt. = 3
.1slope. = {rpn .1yt. .1yb. - .1xr. .1xl. - /}
.1intercept. = {rpn .1yb. .1slope. .1xl. * -}
.2xl. = 1966.4
.2yb. = 3
.2xr. = 1980
.2yt. = -1
.2slope. = {rpn .2yt. .2yb. - .2xr. .2xl. - /}
.2intercept. = {rpn .2yb. .2slope. .2xl. * -}
#
# PANEL 1: Bravo time period.
#
set x margin 3
set x size 15
set y margin 3
set y size 5
# Draw border big enough for this and next panel.
draw border box {rpn ..xmargin.. 2 -} \
    {rpn ..ymargin.. 2 -} \
    {rpn ..xmargin.. ..xsize.. + 2 +} \
    {rpn ..ymargin.. ..ysize.. 2 * 3 + + 2 +} \
    0.2 0.75
set missing value .missing_value.
set ignore error eof
set x name "Year"
set x axis .tmin. .tmax. .tinc. .tincinc.
set y name "Area / 10$^5$km$^2$"
set y axis -3 3 1
draw axes
#
# Draw index lines 1 and 2.
#
# Upward sloped line.
```

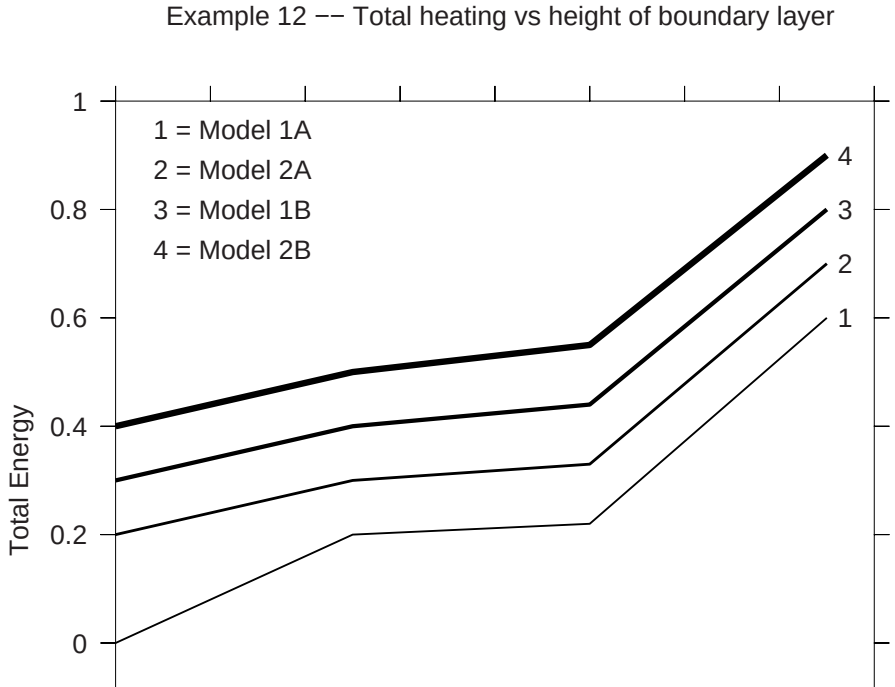
```
set line width .thin.
set graylevel .gray_for_guiding_lines.
if {rpn .1intercept. ..xright.. .1slope. * + ..ytop.. <}
    draw line from              \
        ..xleft..               \
        {rpn .1intercept. ..xleft.. .1slope. * +} \
        to                      \
        {rpn ..ytop.. .1intercept. - .1slope. /} \
        ..ytop..
else
    draw line from              \
        ..xleft..               \
        {rpn .1intercept. ..xleft.. .1slope. * +} \
        to                      \
        ..xright..              \
        {rpn .1intercept. ..xright.. .1slope. * +}
end if
set graylevel 0
#
# Downward sloped line.
set line width .thin.
set graylevel .gray_for_guiding_lines.
if {rpn .2intercept. ..xleft.. .2slope. * + ..ytop.. <}
    draw line from              \
        {rpn ..ytop.. .2intercept. - .2slope. /} \
        ..ytop..                \
        to                      \
        ..xright..              \
        {rpn .2intercept. ..xright.. .2slope. * +}
else
    draw line from              \
        ..xleft..               \
        {rpn .2intercept. ..xleft.. .2slope. * +} \
        to                      \
        ..xright..              \
        {rpn .2intercept. ..xright.. .2slope. * +}
end if
set graylevel 0
#
# Finally, draw the data curve on top, after first
# whiting out a background.
set input data window x .tmin. .tmax.
open \file
read columns x y
close
y /= 1e5
set line width ..linewidthaxis..
draw zero line
set line width {rpn .thick. 3 *}
set graylevel 1
draw curve
```

```
set graylevel 0
set line width .thick.
draw curve

#
# PANEL 2: Longer timescale.
#
delete x scale
set x margin bigger 5
set x size 10
set x name ""
set y name ""
set y margin bigger {rpn ..ysize.. 3 +}
#
# Draw long data set in thin pen.
set input data window x off
open \file
read columns x y
close
y /= 1e5
#
# Draw guiding lines, axes, etc.
set x axis 1952 1980 5 1
draw axes frame
set line width .thin.
set graylevel .gray_for_guiding_lines.
draw line from .1xl. .1yb. to .1xr. .1yt.
draw line from .2xl. .2yb. to .2xr. .2yt.
set graylevel 0
set line width ..linewidthaxis..
draw zero line


draw x axis at bottom
.old. = ..fontsize..
set font size 0
draw y axis at left
set font size .old.
delete .old.
#
# Draw full curve (first whiting out region around it).
set line width {rpn .thin. 4 *}
set graylevel 1
draw curve
set graylevel 0
set line width .thin.
draw curve
#
# Draw bravo time period (first whiting out region around it).
set input data window x .tmin. .tmax.
open \file
```

```
read columns x y
close
y /= 1e5
set line width {rpn .thick. 3 *}
set graylevel 1
draw curve
set graylevel 0
set line width .thick.
draw curve
#
# Done
set font size 20
\label = "Example 11 (Arctic ice anomaly)"
draw label "\label" at          \
    {rpn 8.5 2.54 * "\label" width - 2 /} \
    {rpn ..ytop.. yusertocm 0.7 +} \
    cm
if !..publication..
    draw time stamp
end if
```

8.6 Legends and annotated lines

The following example shows how to handle annotated curves and legends.



```
# Example 12 -- Linegraph with key inside plot
set font size 10                # points (1in = 72pt)
set x size 10                   # cm
set y size 10                   # cm
set x name "Height"
set y name "Total Energy"

# Following axis setups not necessary; will autoscale if you
# remove these.
set x margin 3
set x axis 800 960 20
set y margin 3
set y axis -0.4 1 0.2

# Read data.  Format is columns (x, y1, y2, y3, y4)
open example12.dat
read columns x y
draw curve
draw label for last curve "1"

rewind
set line width {rpn ..linewidth.. 1.5 *}
read columns x * y
draw curve
draw label for last curve "2"

rewind
set line width {rpn ..linewidth.. 1.5 *}
read columns x * * y
draw curve
draw label for last curve "3"

rewind
set line width {rpn ..linewidth.. 1.5 *}
read columns x * * * y
draw curve
draw label for last curve "4"

# Draw the key.
# NOTES:
# (1) This key is inside the plot; its location was chosen
#     after looking at the data.  To put the key in a different
#     location, alter the .key_topleft_x. and .key_topleft_y.
#     variables.  For example, you could put the key to the
#     right of the plot by changing the next line to:
#     '.key_topleft_x. = {rpn ..xsize.. 0.5 +}'
# (2) The variable .dy_inc. is the spacing between lines in
#     the key.  It should be OK even if you change the
#     font size above.
.key_topleft_x. = 0.5           # cm right of left axis
.key_topleft_y. = 0.5           # cm below top axis
```

```
.dy_inc. = {rpn ..fontsize.. pttocm 1.5 *}

draw label "1 = Model 1A" at    \
    {rpn ..xleft.. xusertocm .key_topleft_x. +} \
    {rpn ..ytop.. yusertocm .key_topleft_y. -} cm

.key_topleft_y. += .dy_inc.
draw label "2 = Model 2A" at    \
    {rpn ..xleft.. xusertocm .key_topleft_x. +} \
    {rpn ..ytop.. yusertocm .key_topleft_y. -} cm

.key_topleft_y. += .dy_inc.
draw label "3 = Model 1B" at    \
    {rpn ..xleft.. xusertocm .key_topleft_x. +} \
    {rpn ..ytop.. yusertocm .key_topleft_y. -} cm

.key_topleft_y. += .dy_inc.
draw label "4 = Model 2B" at    \
    {rpn ..xleft.. xusertocm .key_topleft_x. +} \
    {rpn ..ytop.. yusertocm .key_topleft_y. -} cm

draw title "Example 12 -- Total heating vs height of boundary layer"
```

8.7 Drawing gray polygons

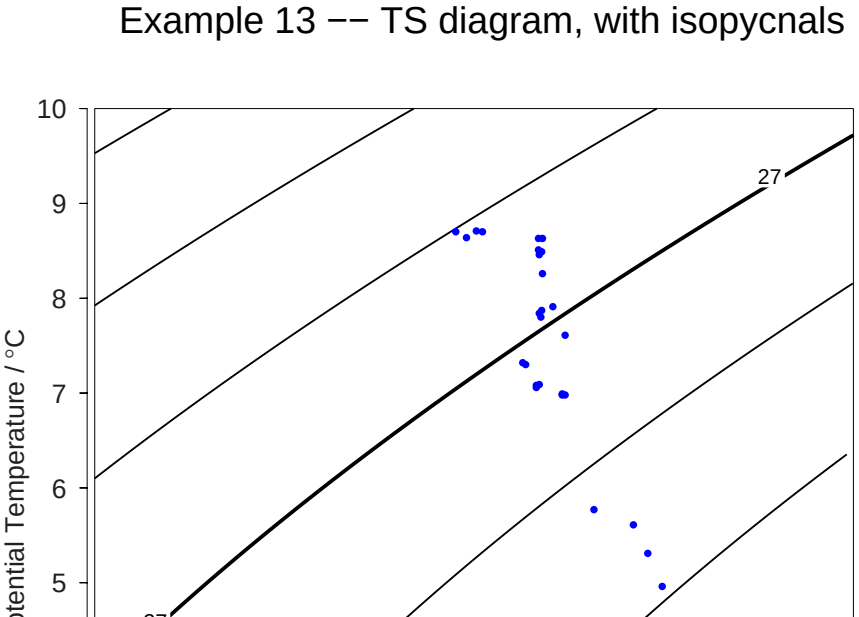
The following example shows how to draw polygons of a graylevel that is read in. It also draws a bullet (within the polygon). See the help lines at the start.

```
'Draw Polygon And Bullet'
Draw a polygon of a given graylevel, with a bullet
(black dot) at an indicated location.  The polygon
coordinates are read in by this command, as are
the graylevel and the location of the bullet.

Variables used:
  .black.
  .white.

Input data read:

line   1:
  A code (which is ignored) and a graylevel
  to draw the polygon with. The value for this
  graylevel ranges from .black. (which codes
  to black ink on the paper) to .white.
  (which codes to blank paper).

line   2:
  An (x,y) location for the bullet.

line   3:
```

```
  Skipped.

line 4-n:
  Locations of vertices of polygon, ended with a blank line.
{
  read .code. .graylevel. # Read line 1
  read .x. .y.            # Read line 2
  skip                    # Skip line 3
  read columns x y        # Read a polygon
  # Adjust .graylevel. to range between 0
  # (for black ink) and 1 (for white paper),
  # then set graylevel and draw polygon.
  .graylevel. = {rpn .graylevel. \
      .black. - .white. .black. - /}
  set graylevel .graylevel.
  draw curve filled
  # Draw black bullet
  set graylevel 0
  draw symbol bullet at .x. .y.
  # Clean up local storage.
  delete .code.
  delete .graylevel.
  delete .x.
  delete .y.
}
```

8.8 Temperature-Salinity Diagram

Here is how you might draw an oceanographic "TS" (temperature salinity) diagram:



```
# Example 13 -- TS diagram, with isopycnals
#
# Draw Axes
set line width axis 0.25
set line width 0.75
.tic_size. = 0.1                # cm
set symbol size 0.03
.isopycnal_fontsize. = 8        # for isopycnal labels
.axes_fontsize. = 10            # for axes
set font size .axes_fontsize.
set x margin 2
set x size 10
set y margin 2
set y size 10
.Smin. = 33.4
.Smax. = 35.0
.Sinc. = 0.5
.Sincinc. = 0.1
.thetamin. = -2.0
.thetamax. = 11.0
.thetainc. = 1.0
.thetaincinc. = 1.0
set tic size .tic_size.
set x name "Salinity / PSU"
set y name "Potential Temperature / $\circ$C"
set x axis .Smin. .Smax. .Sinc. .Sincinc.
set y axis .thetamin. .thetamax. .thetainc. .thetaincinc.
set axes style offset
draw axes 1
set clip on
.old. = ..fontsize..
set font size .isopycnal_fontsize.
draw isopycnal 26.00
draw isopycnal 26.50 unlabelled
draw isopycnal 27.00
draw isopycnal 27.50 unlabelled
draw isopycnal 28.00
draw isopycnal 28.50 unlabelled
draw isopycnal 29.00
set clip off
set font size .old.
#
# Draw the data.
open example13.dat
read columns x y
draw symbol bullet
set font size ..
draw title "Example 13 -- TS diagram, with isopycnals"
```

8.9 Probability Density Function Diagram

A common application is to draw the probability density function for (x,y) data. Gri has no builtin facility to do this, but the following example shows how to create the gridded PDF data using a call to the `perl` system command. The gridded data thus generated are contoured, creating a PDF diagram. As the comments in the program state, the first call to Perl is specific to a particular dataset, and can be ignored on first reading; it just creates the file '`tmp-xy.\.pid.`'.

```
# Draw prob-density-function TS diagram for Bravo data

# This first call to perl is specific to the
# particular (weird) dataset.  All that matters
# is that a file of (x,y) data is created, and
# stored in the file called 'tmp-xy.\.pid.'
system perl <<"EOF"
#
# Slurp in x[], y[] data
$dir = "/users/dek/kelley/Labrador/bravo/data";
$Sfile = "$dir/S_25db_1day";
$Tfile = "$dir/T_25db_1day";
open(S, "$Sfile") || die "Can't open input $Sfile";
open(T, "$Tfile") || die "Can't open input $Tfile";
open(ST, ">tmp-xy.\.pid.")
    || die "Can't open tmp-xy.\.pid.";
$day = 5;
$year = 1964;
while(<S>) {
    @S = split;
    $_ = <T>;
    @T = split;
    if (240 < $day && $day < 360) {
        for ($i = 0; $i < $#S; $i++) { #all depths
            print ST "$S[$i] $T[$i]\n";
        }
    }
    $day += 1;
    if ($day > 365) {
        $year++;
        $day = 0;
    }
    if ($year > 1967) { last; }
}
EOF

#
# Create 2D PDF for (x,y) data in file \infile
# storing the results in \outfile.  X ranges
# between the indicated limits, with the indicated
# binsize, as does y.  The synonyms defined
# on the next 4 lines are the only input this
```

```
# perlscript needs; the perlscript itself is
# quite general.  For details of what it does,
# particularly the scaling of the PDF, see
# the perl comments.
\xmin = "33.5"; \xmax = "35.5"; \xinc = "0.05";
\ymin = "-2.0"; \ymax = "11.0"; \yinc = "0.25";
\infile = "tmp-xy.\.pid."
\outfile = "tmp-grid.\.pid."
system perl <<"EOF"
#
# Prepare 2 dimensional probability-density-function
# dataset for contouring by Gri.  This reads (x,y)
# data from a file called $infile (defined below)
# and creates an output file called $outfile
# (also defined below) containing the
# x-grid, the y-grid, and then the grid data,
# suitable for reading/contouring by the Gri
# commands
#       open tmp-grid.\.pid.
#       read .number_of_data.
#       read grid x
#       read grid y
#       read grid data
#       draw contour
#
# The values in the output grid are defined so
# that they sum to the reciprocal of the
# product of the x binsize and y binsize (see
# definition of $factor below).
#
$xmin = \xmin; $xmax = \xmax; $xinc = \xinc;
$ymin = \ymin; $ymax = \ymax; $yinc = \yinc;
$infile = "\infile";
$outfile = "\outfile";
#
# Slurp in the x[], y[] data, storing
# the total number of data in $n.
open(IN,  "$infile")   || die "Can't open infile";
open(OUT, ">$outfile") || die "Can't open outfile";
$n = 0;
while(<IN>) {
    ($x[$n], $y[$n]) = split;
    $n++;
}
#
# Zero out matrix (stored in a linear array scanned
# as one reads a book).
$cols = int(1 + ($xmax - $xmin) / $xinc);
$rows = int(1 + ($ymax - $ymin) / $yinc);
for ($y = $ymax; $y > $ymin; $y -= $yinc) {
    for ($x = $xmin; $x < $xmax; $x += $xinc) {
```

```
        $l = int(($x - $xmin) / $xinc
            + $cols * int(($y - $ymin) / $yinc));
        $sum[$l] = 0;
    }
}
#
# Cumulate (x,y) data into the matrix
$inside = 0;
for ($i = 0; $i < $n; $i++) {
    if ($ymin <= $y[$i] && $y[$i] <= $ymax
        && $xmin <= $x[$i] && $x[$i] <= $xmax) {
        $l = int(($x[$i] - $xmin) / $xinc
            + $cols * int(($y[$i] - $ymin) / $yinc));
        $sum[$l]++;
        $inside++;
    } else {
        print STDERR "($y[$i], $x[$i]) clipped\n";
    }
}
#
# Print number of points (to allow renormalization
# if the user wishes)
print OUT "$inside\n";
#
# Print x grid, y grid, then grid itself
for ($x = $xmin; $x < $xmax; $x += $xinc) {
    printf OUT "%lg\n", $x;
}
print OUT "\n";
for ($y = $ymax; $y > $ymin; $y -= $yinc) {
    printf OUT "%lg\n", $y;
}
print OUT "\n";
#
# The $factor variable scales the PDF.
$factor = 1 / $xinc / $yinc / $inside;
for ($y = $ymax; $y > $ymin; $y -= $yinc) {
    for ($x = $xmin; $x < $xmax; $x += $xinc) {
        $l = int(($x - $xmin) / $xinc
            + $cols * int(($y - $ymin) / $yinc));
        printf OUT "%lg ", $factor * $sum[$l];
    }
    print OUT "\n";
}
EOF

# Axes
set x margin 3
set x margin 6
set x name "Salinity [PSU]"
set y name "Potential Temperature [$\circ$C]"
```

```
set x axis 34.5 34.8 0.1
set y axis 5 9 1
draw axes

# PDF
open tmp-grid.\.pid.
read .number_of_data.
read grid x
read grid y
read grid data
.smooth. = 0

# Contours.  Use clipping, since the axes are
# zooming in on part of the PDF.
set font size 8
set contour label position centered
set clip postscript on
set line width rapidograph 4x0
draw contour 0.2 2.2 0.4 unlabelled
set line width rapidograph 0
draw contour 0.4 2.4 0.4
set clip postscript off
end if
```

8.10 Running-Mean Skyline Diagram

Timeseries data are often cast into running means; e.g. a temperature record might be cast into monthly mean values. The following example shows how to use a perl script to accomplish this easily, producing a graph with both the raw data (bullets) and the running mean (a skyline plot).

```
'Bin with  x .min. .max. .inc. \in_file \out_file'

Creates \out_file from \in_file.  In each of these
files, column 1 represents x and column 2 represents
y.  The \out_file file contains the average values
of y in x bands of width .inc., centred at .min.,
(.min.+.inc.), up to .max, and with missing values
inserted in bands with no x-data in \in_file.
Each x-band is represented in \out_file by a
plateau in y, and adjacent bands with
non-missing data are connnected by vertical
lines; the effect is a skyline plot of the
banded means.  Sample application: plot
monthly means of a variable.
{
    if {rpn \.words. 8 !=}
        show "ERROR: '\.proper_usage.' called without"
        show " giving all parameters"
        quit 1
    end if
```

```
    system perl <<"EOF"
    $min = \.word3.;
    $max = \.word4.;
    $inc = \.word5.;
    open(IN,   "\.word6.")
        || die "‘\.proper_usage’: no \\in_file";
    open(OUT, ">\.word7.")
        || die "‘\.proper_usage’: no \\out_file";

    $n = ($max - $min) / $inc;
    #
    # Set up bins.
    for($i = 0; $i <= $n; $i++) {
       $xx[$i] = 0;
       $yy[$i] = 0;
       $nn[$i] = 0;
    }
    while(<IN>) {
        ($x, $y) = split(' ');
        $i = int(0.5 + ($x - $min) / $inc);
        $i =      0 if $i <      0;
        $i = $n - 1 if $i > $n - 1;
        $xx[$i] += $x;
        $yy[$i] += $y;
        $nn[$i]++;
    }
    for($i = 0; $i <= $n; $i++) {
        if ($nn[$i] > 0) {
            $xx[$i] /= $nn[$i];
            $yy[$i] /= $nn[$i];
            $xleft  = $min + $inc * ($i - 0.5);
            $xright = $min + $inc * ($i + 0.5);
            #
            # If datum to left non-missing,
            # just draw vertical line
            # down to the last yy value.
            if ($i > 0 && $nn[$i - 1] > 0) {
                print OUT "$xleft $yy[$i - 1]\n";
            } else {
                print OUT "$xleft \.missingvalue.\n"
            }
            print OUT "$xleft  $yy[$i]\n";
            print OUT "$xright $yy[$i]\n";
        }
    }
EOF
}

# Bin into months
Bin with x 1964 1974 {rpn 1 12 /} \
    timeseries.dat tmp.dat
```

```
open tmp.dat
read columns x y
close
draw curve                      # skyline of means
open timeseries.dat
read columns x y
close
draw symbol bullet              # data
system rm -f tmp.dat            # clean up
```

8.11 Finite Element Model mesh

Finite Element Models (used in fluid mechanics) employ non-rectangular meshes, and plotting these meshes requires a few intermediate steps. Consider the common case of triangular elements. Suppose two data files exist describing the mesh, the first, '`model.nodes`' say, consists of a description of the x-y coordinates of the nodes (vertices) of the triangles. The second, '`model.elements`' say, consists of a description of which triplet of nodes defines each triangle in the mesh. Here, from a sample application, is a node file called '`model.nodes`':

```
1 1 1
2 2 1
3 1 2
4 3 1.5
5 2 2
6 1.5 3
```

Here is the corresponding file of the elements, called '`model.elements`'

```
1 1 2 3
2 2 5 3
3 2 4 5
4 3 5 6
```

In each of these files, the first column is a reference number. Thus, '`model.elements`' indicates that the first triangle is defined by the nodes numbered `1`, `2` and `3` as defined in '`model.nodes`'. More specifically, the triangle is defined by vertices at (x,y) locations (1,1), (2,1), and (1,2).

A Gri program, named '`FEM.gri`', to draw the nodes is the following.

```
set missing value -99.99
# Create data using perl-script ...
system FEM.pl model.nodes model.elements > tmp
# ... then plot it ...
open tmp
read columns x y
close
draw curve
# ... and, finally, clean up the temporary file
system rm tmp
```

The work of interpreting the data files is done by the perlscript that follows, named '`FEM.pl`'

```
#!/usr/bin/perl -w
$missing = -99.99;              # missing value
$node_file = $ARGV[0];
```

```
$element_file = $ARGV[1];
open (NODE, $node_file)
    or die "Cannot open '$node_file' file";
open (ELEM, $element_file)
    or die "Cannot open '$element_file' file";

# Read in node information, creating arrays
# named $node_x[] and $node_y[]. Check that
# the first column (the index) makes sense.
$max_node = 1;
while(<NODE>) {
    ($index, $node_x[$max_node], $node_y[$max_node])
        = split;
    die "Node mismatch at index=$index"
        if ($index != $max_node);
    $max_node++;
}

# Read in triangle elements, into arrays
# $a[], $b[], and $c[].  Check that the
# first column (the index) makes sense.
$max_elem = 1;
while(<ELEM>) {
    ($index, $a[$max_elem], $b[$max_elem], $c[$max_elem])
        = split;
    die "Element mismatch at index=$index"
        if ($index != $max_elem);
    $max_elem++;
}

# Print out triangles suitable for plotting in gri.
for ($i = 1; $i < $max_elem; $i++) {
    print $node_x[$a[$i]], " ", $node_y[$a[$i]], "\n";
    print $node_x[$b[$i]], " ", $node_y[$b[$i]], "\n";
    print $node_x[$c[$i]], " ", $node_y[$c[$i]], "\n";
    # Repeat first, to close the triangle.
    print $node_x[$a[$i]], " ", $node_y[$a[$i]], "\n";
    print $missing, " ", $missing, "\n";
}
```

8.12 Handling Data

8.12.1 Handling headers

Case 1 – known number of header lines. This is easy. If you know that the file has, say, 10 header lines, you can just do this:

```
open file
skip 10
read columns x y
```

```
...
```

Case 2 – header itself indicates number of header lines. Quite often the first line of a file will indicate the number of header lines. For example, suppose the first line contains a single number, indicating the number of header lines to follow:

```
open file
read .skip.
skip .skip.
read columns x y
...
```

Case 3 – header lines marked by a textual key. Sometimes header lines are indicated by a textual key, for example, the characters `HEADER` at the start of the line in the file. The easy way to skip such a header is to use a system command. Depending on your familiarity with the operating system (here presumed to be Unix), you might choose to use Grep, Awk, or Perl. Here are examples:

```
open "grep -v '^HEADER' file |"
```

For more on the `|` mechanism, see Section 9.3.28 [Open], page 102. The Grep command prints lines which do not match the indicated string (because of the `-v` switch), and the `^` character stands for the start of the line (Section 11.2.2 [Grep], page 189). Thus all lines with the key word at the **start** of the line are skiped.

Case 4 – reading and using information in header. Consider a dataset in which the first line gives the time of observation, followed by a list of observations. This might be, for example, an indication of the data taken from a weather balloon released at a particular time from a fixed location, with the main data being air temperature as a function of elevation of the balloon. The time indication might be, for instance, the hour number. One might need to know the time to print a label on the diagram. You could do that by:

```
open file
read .time.
read columns x y
draw curve
sprintf \label "Time of observation is %f hour" .time.
draw title "\label"
```

where the `sprintf` command has been used to change the numerical time indication into a synonym that can be inserted into a quoted string for drawing the title of the diagram (Section 9.3.47 [Sprintf], page 138). Here the time has been assumed to be a decimal hour. You might also have three numbers on the line, perhaps a day, an hour and a minute. Then you could do something like

```
open file
read .d. .h. .m.
read columns x y
draw curve
sprintf \label "Obs. %.0f:%.0f, day %.0f" .h. .m. .d.
draw title "\label"
```

Here the `%.0f` code is used to ensure no numbers will be written after the decimal point. Naturally, you could convert this to a decimal day, by e.g.

```
...
.dday. = {rpn .day. .hour. 24 / .min. 24 / 60 /}
sprintf \label "Decimal day is %.4f" .dday.
...
```

(Some of you might know how many minutes in a day, but I'm silly so I kept the extra mathematical step – nothing is lost by being straightforward!)

8.12.2 Ignoring columns that are not of interest

Quite often a dataset will have many columns, of which only a couple are of interest to you. Consider for example oceanographic data which has columns storing, in order, these variables: (1) depth in water column, (2) "in situ" temperature, (3) "potential" temperature, (4) salinity, (5) conductivity, (6) density, (7) sigma-theta, (8) sound speed, and (9) oxygen concentration. But you might only be interested in plotting a graph of salinity on the x-axis and depth on the y-axis. Here are several ways to do this:

```
open file
read columns y * * x
draw curve
```

where the * is a place-keeper to indicate to skip that column. For a large number of columns, or as an aesthetic choice, you might prefer to write this a

```
open file
read columns y=1 x=4
draw curve
```

Many users would just as soon not bother with this syntax, preferring instead to use system tools with which they are more familiar. So a Gawk user might write

```
open "gawk '{print($1, $4)}' file |"
read columns y x
draw curve
```

For more on the Gawk command see Section 11.2.4 [Awk], page 189.

8.12.3 Algebra on column data

Suppose the file contains (x,y), but you wish to plot 2y times x. You could do the doubling of y within Gri, as

```
open file
read columns x y
y *= 2
draw curve
```

or you could use a system tool, e.g. gawk, as in this example (Section 11.2.4 [Awk], page 189).

```
open "gawk '{print($1,2*$2)}' file|"
read columns x y
draw curve
```

The latter is preferable in the sense that it is more powerful. The reason for this is that Gri allows you to manipulate the x and y columns, using so-called RPN mathematics (Section 10.9 [rpn Mathematics], page 161), but you cannot blend the columns. For example, you cannot easily form the ratio y/x in Gri. (Actually, you can, by looping through your data and doing the calculation index by index, but if you knew that already you wouldn't need to be reading this section!) Such blending is trivial in the operating system, though, as in the following Gawk example (Section 11.2.4 [Awk], page 189).

```
open "gawk 'print {($1, $2/$1)}' file |"
```

```
read columns x y
draw curve
```

8.12.4 Combining columns from different files

Suppose you want to plot a column (`y`, say) from one file versus a second column (`x`) from a second data file. The easy way is to use a system command to create a new file, for example the Unix command `paste` – but of course you don't want to clutter your filesystem with such files, so you should do this withing Gri:

```
open "paste file1 file2 |"
read columns x y
draw curve
```

8.12.5 Plotting several y-columns versus on x-column

Sometimes you'll have a datafile with the first column being x, and the other columns being various things to plot versus x. For example, you might have the data

```
1  8 11  9
2 22 21 20
3 11 10  9
4 20 15 10
```

in a file called `test.dat`. Let's say the x-column is time, and the y-columns are the readings from three temperature sensors. The following illustrates how you might plot these data. If you think the new-command which starts this script is useful, just insert it in your '`~/.grirc`' file and you can just use it without re-defining it each time. This will give Gri a command called `draw curves`.

```
'draw curves \xname \y1name ...'
Draw multiple y columns versus an x column.  Assumes
that the datafile is open, and that x is in the first
column, with the y values in one or more following
columns.

The number of columns is figured out from the options,
as is the name of the x-axis, and the labels to be
used on each of the y curves.
{
  # NB. the 3 below lets us skip the words 'draw'
  # and 'curves', and the name of the x-column.
  .num_of_y_columns. = {rpn wordc 3 -}
  if {rpn .num_of_y_columns. 1 >}
    show "ERROR: 'draw curves' needs at least 1 y column!"
    quit
  end if

  set x name {rpn 2 wordv}
  set y name ""

  # Loop through the columns.
  .col. = 0
```

```
  while {rpn .num_of_y_columns. .col. <}
    # The x-values will be in column 1, with y-values
    # in columns 2, 3, ..., of the file.
    .ycol. = {rpn .col. 2 +}
    rewind
    read columns x=1 y=.ycol.
    # At this point, you may want to change line thickness,
    # thickness, color, dash-type, etc.  For illustration,
    # let's set dash type to the column number.
    set dash .col.
    draw curve
    draw label for last curve {rpn .col. 3 + wordv}
    .col. += 1
  end while
}

open test.dat
draw curves time y1 y2 y3
```

9 List of Commands in the Gri Language

9.1 Overview of Gri Commands

The Gri commands may be divided roughly into a few categories, as indicated in the following list.

- **Working with files**: Commands are `open`, `close`, `skip`, `read`, and `rewind`.
- **Controlling parameters of the drawn material**: Various `set` commands control values of parameters, like size of plot, linewidth, font, etc.
- **Drawing things**: Various `draw` commands let you draw data, axes, etc.
- **Interacting with the user**: The `query` command gets instructions from the user. The `show` command displays messages to user.
- **Controlling program flow**: The `if` statement controls optional execution of commands (Section 10.6 [If Statements], page 157). The `while` statement allows loops.
- **Moving around in directories**: The `pwd`, `cd` and `ls` commands do the usual unix things.
- **Using the operating system** The `system` command passes instructions to the operating system; the output may be saved into a synonym by using `\syn = system ...`. The `get env` command determines the value of any unix environment variables the system has defined. For more discussion (Section 10.16 [Operating System], page 183).
- **Statistical operations**: Some very limited capabilities exist; for example, `regress` does linear regression.
- **Mathematical operations**: Simple mathematical manipulation of column, grid, and image data is provided. Also, wherever Gri expects a number, it will accept a reverse-polish expression; for example, `set x size 10` could also be written `set x size {rpn 20 2 /}`. For details (Section 10.8 [Mathematics], page 159).

9.2 Command syntax

The syntax description is enclosed within angled single quotes, optional items are enclosed in square brackets, multiword items are enclosed in curly braces, and vertical bars separate different legitimate choices. For example, the documentation item for the command for drawing contours

```
'draw contour \
    [.value. |  \
      {.min. .max. .inc. [.inc_unlabelled.]}] \
    [unlabelled]'
```

indicates that following are legal:

```
draw contour                    # gri selects levels
draw contour unlabelled         # " but unlabelled
draw contour 10                 # single contour line
draw contour 10 unlabelled      # " but unlabelled
draw contour 0 100 10           # contours at z=0,1,
draw contour 0 10 1 unlabelled # " but unlabelled
# contours at 0, 0.1, ... labelled at 0, 1
draw contour 0 10 1 0.1
```

Note that items enclosed in braces must appear in their entirety; for example,

```
draw contour 0 10               # WRONG; missing .inc.
```

which might look similar `draw contour .min. .max. .inc.` to you, looks like garbage to Gri. Gri will recognize it as an attempt at the `draw contour` command (because the first 2 words match the syntax) but it will then get confused, spit out an error message, and quit.

9.3 List of all Gri commands

Commands are listed below in the order in which they are defined in the '`gri.cmd`' file (Chapter 3 [Invoking Gri], page 7). What you see here is similar to, but not identical to, the text of the online help. Gri usually accepts both American and English spellings (As an example of spelling latitude, Gri accepts `grey` anywhere the manual says `gray`, and `colour` for `color`.)

9.3.1 `assert`

```
'assert .condition. ["message"]
```

The condition may be a variable, a synonym, or an RPN expression. If this condition is true (i.e. evaluates to a non-zero number), do nothing. If the condition is false, the program will terminate with an error condition (in unix, it will terminate with a non-zero exit code).

Before termination, a message will be printed, the form of which depends on the optional `"message"` string.

If no `"message"` string is given, the the printed message will indicate the name of the command-file and the line at which the assert command was encountered.

If a `"message"` string is given, and if it ends in a newline (`"\\n"`), then this string is printed.

If a `"message"` string is given, and if it does not end in `"\\n"`, then the string is printed along with an indication of the location in the command-file.

(Perl users will recognize this as being patterned on the `"die"` command.)

9.3.2 `cd`

```
'cd [\pathname]'
```

If a pathname specified, change to that directory. Normal unix filenames are used, as in the C-shell convention. For example, the commands `cd ~/src` and `cd $HOME/src` are equivalent. You may specify relative pathnames as in `cd ../sister_directory`.

If no \pathname directory path is specified, go to the home directory, exactly as `cd ~` and `cd $HOME` do.

9.3.3 `close`

```
'close [\filename]'
```

If no filename is specified, close the most recently opened data-file; otherwise close the indicated file.

9.3.4 The `convert` commands

9.3.4.1 `convert columns to grid`

Various forms exist:

```
'convert columns to grid OPTIONS'
```

where the `OPTIONS` may be omitted or selected from this list:

```
'neighbor'
'boxcar    [.xr. .yr. [.n. .e.]]'
'objective [.xr. .yr. [.n. .e.]]'
'barnes    [.xr. .yr. .gamma. .iter.]'
```

For more discussion on the methods see Section 6.2 [Ungridded Data], page 27.

All these commands "grid" columnar (x,y,z) data. That is, they fill up a grid based on some form of interpolation of the possibly randomly-spaced columnar data. There are many methods in existence for doing this, and Gri implements several of them as alternatives.

The grid will have been defined by commands such as `set x grid`, `set y grid`, `read grid x` and `read grid y`. As of version 2.1.9, Gri does not require a grid to have been pre-defined; it will create a regular 20 by 20 grid, spanning the range of x and y data, as a default. This is a good starting point in many cases.

'neighbor' method
: Very fast but very limited.

'boxcar' method
: Slower but a lot better. Still, this can produce noisy contours if the data are not densely and uniformly ditributed through domain.

'objective' method
: Somewhat slower than 'boxcar', but produces better fields since the averaging function is smooth.

'barnes' method
: Somewhat slower than 'objective', but only by a constant factor (that is, independent of number of data). This produces by far the best results, since the smoothing function has variable spatial scale. This is the default method if no method is supplied.

All except the `neighbor` method may take optional arguments to define the x and y scales of the smoothing function (called `.xr.` and `.yr.`). (The barnes method has two other optional arguments – see below.) If you do not supply these arguments, Gri will make a reasonable choice and inform you of its decision. Many users find that it is best to `convert columns to grid` with no additional parameters as a first step, to get advice on values to use for the optional parameters.

The default `.xr.` and `.yr.` are calculated by determining the span in x and in y directions, and dividing each by the square root of the number of data points. These numbers are then multiplied by the square root of 2. The method is as proposed by S. E. Koch and M. DesJardins and P. J. Kocin, 1983. "An interactive Barnes objective map anlaysis scheme for use with satellite and conventional data,", J. Climate Appl. Met., vol 22, p. 1487-1503.

If `.xr.` and `.yr.` were supplied but negative, then Gri interprets this as an instruction to modify the default values, described in last paragraph, by multiplying by the absolute values of the negative numbers given, instead of muliplying by square root of 2.

If the `chatty` option is turned on then Gri will print out the values of (dx,dy) that it has calculated; this gives you some guidance for supplying your own values of `(.xr.,.yr.)`

if you choose to supply them yourself. It is also a good idea to let these parameters be a guide for your grid spacing; for example, Koch et al., 1983, suggest using grid spacing of 0.3 to 0.5 times (dx,dy).

And now, the details . . .

- **"Neighbor" method** The `convert columns to grid neighbor` method is useful for (x,y,z) data which are already gridded (i.e., for which x and y take only values which lie on the grid), or nearly gridded. The (x,y,z) data are scanned from start to finish. For each data point, the nearest grid point is found. Nearness is measured as Cartesian distance, with scale factor given by the distance between the first and second grid points. In other words, distance is given by D=sqrt(dx*dx+dy*dy) where dx is ratio of distance from data point to nearest grid point, in x-units, divided by the difference between the first two elements of the x-grid, and dy is similarly defined. Once the grid point nearest the data point is determined, Gri adds the z-value to a list of possible values to store in the grid. Once the entire data set has been scanned, Gri then goes back to each grid point, and chooses the z-value of the data point that was nearest to the grid point – that is, it stores the z value of the (x,y,z) data triplet which has minimal D value. Note that this scheme is independent of the order of the data within the columns.

  The `neighbor` method is useful when the data are already pre-gridded, meaning that the (x,y,z) triplets have x and y values which are already aligned with the grid. **Computational cost:** For `P` data points, `X` x-grid points, and `Y` y-grid points, the method calculation cost is proportional to `P*[log2(X)+log2(Y)]` where `log2` is logarithm base 2. As discussed below, this is often several orders of magnitude lower than the other methods of gridding.

- **"Objective" method** In the `objective` method, a smoothing technique known as objective mapping is applied. It is essentially a variable-size smoothing filter of approximately Gaussian shape (it is method "two" of Levy and Brown [1986 J. Geophysical Res. vol 91, p 5153-5158]) The parameters `.xr.` and `.yr.` give the width of the filter.

  With the optional additional parameters `.n.` and `.e.` are specified, then grid values will be assigned the missing value if there are fewer than `.n.` (x,y,f) data in the neighborhood of the gridpoint, even after enlarging the neighborhood by widening and heightening by root(2) up to `.e.` times. (The enlargement is only done if fewer than `.n.` points are found.) If these parameters are not specified in the command, then values `.n.`=5 and `.e.`=1 are assumed. The special case where `.e.` is negative tells Gri to **always** fill in each grid point, by extending the neighborhood to enclose the entire dataset if necessary.

  Computational cost: For `P` data points, `X` x-grid points, and `Y` y-grid points, the method calculation cost is proportional to `P*X*Y`. Given that `X` and `Y` are determined by the requirement for smoothness of contours and the size of the graph, they are more or less fixed for all applications. They are often in the range of 20 or so – on 10 cm wide graph, this yields a contour footprint of 1/2 cm, which is often small enough to yield smooth contours. Therefore, the computational cost scales linearly with the number of data points. Compared to the "neighborhood" method, this is more costly by a factor of `X*Y/log_2(X)/log_2(Y)` which is normally in the range from 20 to 50.

- **"Boxcar" method** In the `boxcar` method, the grid points are derived from simple averages calculated in rectangles `.xr.` wide and `.yr.` tall, centred on the gridpoints. The `.n.` and `.e.` parameters have similar meanings as in the "objective" method.

  Computational cost: Roughly same as `objective` method described above.

- **"Barnes" method** This is the default scheme.

The Barnes algorithm is applied. If no parameters are specified, `.xr.` and `.yr.` are determined as above, with `.gamma.` set to 0.5, and `.iter.` set to 2 so that two iterations are done. On successive iterations, the smoothing lengthscales `.xr` and `.yr` are each reduced by multiplying by the square root of `.gamma.`. Smaller `.gamma.` values yield better resolution of small-scale features on successive iterations. Koch et al., 1983, recommend using a `.gamma.` value in the range 0.2 to 1, with two iterations.

Provided that all the grid points are close enough to at least some column data, the entire grid is filled. But if `.xr.` and `.yr.` are too small, the weighting function can fall to zero, since it is exponential in the sum of the squares of the x-distance/`.xr.` and the y-distance/`.yr.`; in that case missing values result at those grid points. On a 32 bit computer, the weighting function will fall to zero when x-distance/`.xr.` and y-distance/`.yr.` are less than about 15 to 20.

If weights have been read in (Section 9.3.33.2 [Read Columns], page 107), then these values are applied in addition to the distance-based weighting. (The normalization means that weights for two data points of e.g. 1 and 2 will yield the same result as if the weights had been given as 10 and 20.)

The computational cost at each iteration scales as `P*X*Y)`. This is comparable to that of the "objective" and "boxcar" methods. Since normally two iterations are done, "barnes" is about double the cost of these methods. (Note: versions prior to 2.1.8 were much slower for large datasets, being proportional to `P*P`.)

References: (1) Section 3.6 in Roger Daley, 1991, "Atmospheric data analysis," Cambridge Press, New York. (2) S. E. Koch and M. DesJardins and P. J. Kocin, 1983. "An interactive Barnes objective map anlaysis scheme for use with satellite and conventional data,", J. Climate Appl. Met., vol 22, p. 1487-1503.

The Barnes algorithm is as follows:

The gridded field is estimated iteratively. Successive iterations retain largescale features from previous iterations, while adding details at smaller scales.

The first estimate of the gridded field, here denoted $G_{ij}^{(0)}$ (the parenthetic superscript indicating the order of the iteration) is given by a weighted sum of the input data, with z_k denoting the k-th z value.

$$G_{ij}^{(0)} = \frac{\sum_1^{n_k} W_{ijk}^{(0)} z_k}{\sum_1^{n_k} W_{ijk}^{(0)}}$$

The weights $W_{ijk}^{(0)}$ are defined in terms of a Gaussian function decaying with distance from observation point to grid point:

$$W_{ijk}^{(0)} = \exp\left[-\frac{(x_k - X_i)^2}{L_x^2} - \frac{(y_k - Y_j)^2}{L_y^2}\right]$$

Here L_x and L_y are lengths which define the smallest (x, y) scales over which the gridded field will have significant variations (for details of the spectral response see Koch et al. 1983).

Note: if the user has supplied weights then these are multiplied into the normal distance-based weights; i.e. $w_i W_{ijk}$ is used instead of W_{ijk}.

The second iteration derives a grid $G_{ij}^{(1)}$ in terms of the first grid $G_{ij}^{(0)}$ and "analysis values" $f_k^{(0)}$ calculated at the (x_k, y_k) using a formula analogous to the above. (Interpolation

based on the first estimate of the grid $G_{ij}^{(0)}$ can also be used to calculate $f_k^{(0)}$, with equivalent results for a grid of sufficiently fine mesh.) In this iteration, however, the weighted average is based on the difference between the data and the gridded field, so that no further adjustment of the gridded field is done in regions where it is already close to through the observed values. The second estimate of the gridded field is given by

$$G_{ij}^{(1)} = G_{ij}^{(0)} + \frac{\sum_1^{n_k} W_{ijk}^{(1)}(f_k - f_k^{(0)})}{\sum_1^{n_k} W_{ijk}^{(1)}}$$

where the weights $w_{ik,1}$ are defined by analogy with $W_{ik}^{(0)}$ except that L_x and L_y are replaced by $\gamma^{1/2}L_x$ and $\gamma^{1/2}L_y$. The nondimensional parameter γ $(0 < \gamma < 1)$ controls the degree to which the focus is improved on the second iteration. Just as the weighting function forced the gridded field to be smooth over scales smaller than L_x and L_y on the first iteration, so it forces the second estimate of the gridded field to be smooth over the smaller scales $\gamma^{1/2}L_x$ and $\gamma^{1/2}L_y$.

The first iteration yields a gridded field which represents the observations over scales larger than (L_x, L_y), while successive iterations fill in details at smaller scales, without greatly modifying the larger scale field.

In principle, the iterative process may be continued an arbitrary number of times, each time reducing the scale of variation in the gridded field by the factor $\gamma^{1/2}$. Koch et al. 1983 suggest that there is little utility in performing more than two iterations, providing an appropriate choice of the focussing parameter γ has been made. Thus the gridding procedure defines a gridded field based on three tunable parameters: (L_x, L_y, γ).

9.3.4.2 `convert columns to spline`

```
‘convert columns to spline \
    [.gamma.] \
    [.xmin. .xmax. .xinc.]’
```

Fit a normal or taut interpolating spline, y=y(x), through the (x,y) data. Then subsample this spline to get a new set of (x,y) data. If the spline x-values, `.xmin.`, etc, are not specified, the spline ranges from the smallest x-value with legitimate data to the largest one, with 200 steps in between.

The parameter `.gamma.` determines the type of spline used. If `.gamma.` is not specified, or is given as zero, a standard interpolating spline is used. A knot appears at each x location, with cubic polynomials spanning the space between the knots. If `.gamma.` lies between 0 and 6, a taut spline is used; this tends to have fewer wiggles than a normal spline. If `.gamma.` lies in the range 0 to 3, a taut spline is used, with the possible insertion of knots between interior x pairs. The value 2.5 is used commonly. If `.gamma.` lies in the range 3 to 6, extra knots are permitted in the x pairs at the ends of the domain. A value of 5.5 is used commonly.

Reference Chapter 16 of Carl de Boar, 1987. “A Practical Guide to Splines” Springer-Verlag.

```
read columns x y  # function is y=x^2
0 0
1 1
2 4
3 9
4 16
```

```
set symbol size 0.2
draw symbol bullet
convert columns to spline
draw curve
```

9.3.4.3 convert grid to columns

```
'convert grid to columns'
```

Create column data from grid data. Each non-missing gridpoint is translated into a single (x,y,z) triplet. If column data already exist, then they are first erased. This command is useful in changing the grid configuration, perhaps from a non-uniform grid to a uniform grid. In the following example, a new grid with x=(0, 0.05, 0.1, ..., 0.1) and y=(10, 11, ..., 20) is created. The default gridding method (`convert columns to grid`) is used here, but of course one is free to adjust the method as usual.

```
# ... read/create grid
convert grid to columns
delete grid
set x grid  0  1 0.05
set y grid 10 20 1
convert columns to grid
```

9.3.4.4 convert grid to image

```
'convert grid to image [size .width. .height.] \
    [box .xleft. .ybottom. .xright. .ytop.]'
```

With no options specified, convert grid to a 128x128 image, using an image range as previously set by `set image range`.

Interpolation method: The interpolation scheme is the same used for contouring. Image points that lie outside the grid domain are considered missing. For points within the grid, the first step is to locate the patch of the grid upon which the pixel lies. Then the 4 neighboring grid points are examined, and one of the following cases holds.

1. If 3 or more of them are missing, the pixel is considered missing.
2. If just one of the neighboring grid points is missing, then the image pixel value is determined by bilinear interpolation on the remaining 3 non-missing grid points. (This amounts to fitting a plane to three measurements of height.)
3. If all 4 of the grid points are non-missing, then the rectangle defined by the grid patch is subdivided into four triangles. The triangles are defined by the two diagonal lines joining opposite corners of the rectangle. An "image point" is constructed at the center of the grid patch, with f(x,y) value taken to be the average of the values of the four neighbors. This value is taken to be the value at the common vertex of the four triangles, and then bilinear interpolation is used to calculate the image pixel value.

With the `size` options `.width.` and `.height.` specified, they set the number of rectanglular patches in the image.

With the `box` options specified, they set the bounding box for the image. If `box` is not given, the image spans the same bounding box as the grid as set by `set x grid` and `set y grid`.

Normally, missing values in the grid become white in the image, but this can be changed using the `set image missing value color to...` command.

9.3.4.5 `convert image to grid`

`'convert image to grid'`

Convert image to grid, using current graylevel/colorlevel mapping. For example, if one had a linear mapping of pixel values 0->255 into the user values 10->20, as in

```
set image range 10 20
set image grayscale black 10 white 20
```

then the output grid will be of value 10 where the pixel value is 0, etc. If the image is in color, the grid values will represent the result of mapping the colors to grayscale in the standard way (Foley and VanDam, 1984). [BUG: as of 1.063, the colorscale is ignored completely, and I'm not sure what happens.] The image data are interpolated onto the grid using a nearest-neighbor substitution. This command insists that the image x/y grids have already been defined.

9.3.5 The `create` commands

9.3.5.1 `create columns from function`

`'create columns from function'`

Plot a function of x which is defined in synonym \function.

```
ENVIRONMENT
\function = function to plot.
\xmin     = minimum x value
\xmax     = maximum x value
\xinc     = increment in x values

EXAMPLE
\function = "cos(x)"
\xmin     = "0"
\xmax     = "2 * 3.14"
\xinc     = "0.1"
create columns from function
draw curve
```

NOTE: This only works on machines which have the `awk` command available at the commandline. This means most unix machines and some vax machines.

9.3.5.2 `create image grayscale`

`'create image grayscale banded .band.'`

Make a banded grayscale with in units of .band. pixel values each. Thus, pixel values 0 to (.band. - 1) on the image will map to 0, while values from .band. to (2 * .band. - 1) will map to .band., etc. For example, .band. = 2 gives grayscale = (0 0 2 2 4 4 6 6 ... 252 252 254 254).

9.3.6 debug

```
'debug [.n.]|[clipped values in draw commands]|off'
```

With no optional parameters, sets the value of `..debug..` to 1. (Normally, `..debug..` is 0.) You may use `..debug..` in `if` statements, etc. Note that `..debug..` is also set to 1 when gri is invoked with the commandline switch `-d`.

With `.n.` specified, `..debug..` is set to `.n.`; a value of zero for `.n.` turns debugging off, while 1 turns it on. Higher values may be used for deeper debugging, if you choose:

```
if {rpn ..debug.. 2 <}
  # Code to do if ..debug.. is greater than 2.
end if
```

Note that you can assign to `..debug..` as you can to any other variable; `debug .n.` is equivalent to `..debug.. = .n.`.

With the `clipped` option, Gri prints any clipped data encountered during any `draw ...` commands, EXCEPT in the case of `postscript` clipping, where no check is possible. (Note that `..debug..` is not affected.)

All these forms of debugging are cancelled by `debug off`.

9.3.7 delete

```
'delete .variable.|\synonym [.variable.|\synonym [...]]'
'delete columns [where missing]'
'delete columns [randomly .fraction.]'
'delete grid'
'delete [x|y] scale'
```

Delete some item or characteristic.

- `delete .variable.` Delete definition of variable `.variable.`, making it undefined. Any number of variables or synonyms may be specified on one line.
- `delete \synonym` Delete definition of synonym `\synonym`, making it undefined. Any number of variables or synonyms may be specified on one line.
- `delete \@alias` Delete the item named by the alias (Section 10.5.4 [Alias Synonyms], page 155).
- `delete` with an `&` item Delete the item in the calling program.
- `delete columns` Delete column data.
- `delete columns where missing` Completely delete all column data for which any one of x, y, etc is missing.
- `delete columns randomly .fraction.` Randomly select fraction `.fraction.` of the non-missing column data, and designate them as being missing.
- `delete grid` Delete grid data.
- `delete scale` Delete scales for both x and y, so next `read columns` will set it.
- `delete x scale` Delete scales for x, so next `read columns` will set it.
- `delete y scale` Delete scales for y, so next `read columns` will set it.

9.3.8 differentiate

```
'differentiate {x|y wrt index|y|x} | {grid wrt x|y}'
```

Differentiate column data or grid data. Only the `x` and `y` columns may be differentiated. They may be differentiated either with respect to ("wrt") the index (forming a first difference) or with respect to the other column. The derivative is done with the backwards-difference algorithm. Grid data may differentiated with respect to `x` direction or `y` direction. Grid differentiation is done with a centred difference, with endpoints being assigned the derivative of the neighboring interior point (so that the second derivative is zero at the edges of the grid).

9.3.9 The `draw` commands

Draw commands do actual drawing on the page. You can draw axes, lineplots, symbols, contours, images, and text.

NOTE Gri likes drawings to have axes, so if a `draw` command is executed before any axes have been drawn, Gri will draw axes after it draws the item. (You can get drawings without axes by preceding any other `draw` commands with the command `draw axes none`.) Many users have been surprised by the results of this rule. For example, if you do `set graylevel 0.5` before `draw curve`, you'll find that the axes are drawn in gray also. To avoid this, make sure to do `draw axes` before you modify the graylevel.)

9.3.9.1 The `draw arc` command

`'draw arc [filled] .xc_cm. .yc_cm. .r_cm. .angle_1. .angle_2.'`

Draw an "arc", that is, a portion of a circle. The center of the circle is at the coordinate (`.xc_cm.`, `.yc_cm.`), and the circle radius is `.r_cm.`, all three quantities being in cm on the page, *not* in user-units. The arc starts at angle `.angle_1.`, measured in degrees counterclockwise from a horizontal line, and extends to angle `.angle_2.`, in the same units.

If the keyword `filled` is present, the arc is filled with the current color. Otherwise it is drawn with the current "curve" linewidth Section 9.3.41.33 [Set Line Width], page 126.

9.3.9.2 `draw arrow`

`draw arrow from .x0. .y0. to .x1. .y1. [cm]`

With no optional parameters, draw an arrow from (`.x0.`, `.y0.`) to (`.x1.`, `.y1.`), where coordinates are in user units. The arrow head will be at (`.x1.`, `.y1.`), and its size is as set by most recent call to `set arrow size`. With the `cm` keyword present, the coordinates are in centimetres on the page. NOTE: This will not cause auto-drawing of axes.

9.3.9.3 `draw arrows`

`'draw arrows'`

Draw a vector field consisting of arrows emanating from the coordinates stored in the (x, y) columns. The lengths and orientations of the arrows are stored in the (u, v) columns, and the scale for the (u,v) columns is set by `set u scale` and `set v scale`. **See also** (1) To set arrow size, use `set arrow size`. (2) To get a single arrow, use `draw arrow`.

9.3.9.4 `draw axes if needed`

`'draw axes if needed'`

Draw axes frame if required. Used within gri commands that auto-draw axes. NOTE: this should only be done by developers.

9.3.9.5 draw axes

```
'draw axes [.style.|frame|none]'
```

With no style (`.style.`) specified, draw x-y axes frame labelled at left and bottom. The value of `.style.` determines the style of axes:

- `.style. = 0` Draw x-y axes frame labelled at left and bottom. Since this is the default, it's best to leave it out altogether to make your code easier to understand.
- `.style. = 1` Draw axes without tics at top and right
- `.style. = 2` Draw axes frame with no tics or labels; same as `draw axes frame`

With the keyword `frame` specified, draw axes frame with no tics or labels (just like `.style.` = 2, but preferable because it makes for code that is easier to read and understand).

With the keyword `none` specified, prevent Gri from automatically drawing axes when drawing curves.

Note: `set axes style` can also be used to set axes properties, and then simply using `draw axes`, or letting axes be auto-drawn, will result in the desired effect (Section 9.3.41.1 [Set Axes Style], page 115). However, if the `draw axes` command explicitly asks for a particular style, then it over-rides the style set by `Set Axes Style`.

9.3.9.6 draw border box

```
'draw border box .xleft. .ybottom. .xright. .ytop. \
    .width_cm. .brightness.'
```

Draw gray box, as decoration or alignment key for pastup. The box, with outer lower left corner at (`.xleft.`, `.ybottom.`) and outer upper right corner at (`.xright.`, `.ytop.`) – both coordinates being in centimetres on the page – is drawn with thickness `.width_cm.` and with graylevel `.brightness.` (0 for black; 1 for white). The gray line is drawn inside the box. After drawing the gray line, a thin black line is drawn along the outside edge.

If the geometry is not specified with `.xleft.` and the other parameters, then a reasonable margin is used around the present axes area, and the defaults (`.border.` = 0.2, `.brightness.` = 0.75) are used.

NOTE: This command does not cause auto-drawing of axes.

9.3.9.7 draw box

```
'draw box filled .xleft. .ybottom. .xright. .ytop. [cm|pt]'
```

Draw filled box spanning indicated range, with lower-left corner at (`.xleft.`, `.ybottom.`) and upper-right corner at (`.xright.`, `.ytop.`). The corners are specified in user coordinates, unless the optional `cm` or `pt` keyword is present, in which case they are in centimetres or points on the page. An error will result if you specify user coordinates but they aren't defined yet.

No checking is done on the rectangle; for example, there is no requirement that `.xleft.` be to the left of `.xright.` in your coordinate system.

NOTE: if the box is specified in user units, this command will cause auto-drawing of axes, but not if the box is specified in `cm` or `pt` units

```
'draw box .xleft. .ybottom. .xright. .ytop. [cm|pt]'
```

Draw box spanning indicated range, with lower-left corner at (`.xleft.`, `.ybottom.`) and upper-right corner at (`.xright.`, `.ytop.`).

The corners are specified in user coordinates, unless the optional `cm` or `pt` keyword is present, in which case they are in centimetres or points on the page. An error will result if you specify user coordinates but they aren't defined yet.

No checking is done on the rectangle; for example, there is no requirement that `.xleft.` be to the left of `.xright.` in your coordinate system.

9.3.9.8 `draw circle`

```
draw circle with radius .r_cm. at .x_cm. .y_cm.
```

Draw circle of specified radius (in cm) at the specified location (in cm on the page).

9.3.9.9 `draw contour`

```
'draw contour [{.value. \
  [unlabelled | {labelled "\label"}]} \
  | {.min. .max. .inc. \
    [.inc_unlabelled.] [unlabelled]}]'
```

This command draws contours based on the "grid" data previously read in by a `read grid data` command or created by gridding column data with a `create grid from columns` command. If the grid data don't exist, or if the x and y locations of the grid points do not exist (see `set x grid`, `set y grid`, etc), Gri will complain.

With no optional parameters, draw labelled contours at an interval that is picked automatically based on the range of the data.

With a single numerical value (`.value.`), draw the indicated contour. With the addition of `labelled "\label"`, put the indicated label instead of a numeric label. This can be useful for using scientific notation instead of computer notation for exponents, e.g. `draw contour 1e-5 labelled "10$^{-5}$"`.

With (`.min.`, `.max.` and `.inc.`) given, draw contours for z(x,y) = `.min.`, z(x,y) = `.min. + .inc.`, z(x,y) = `.min. + 2*.inc.`, ..., z(x,y) = `.max.`

With the additional value `.inc_unlabelled.` specified, extra unlabelled contours are drawn at this finer interval.

With the optional parameter `unlabelled` at the end of any form of this command (except the `labelled "\label"` variation, of course), Gri will not label the contour(s).

Hint: It can be effective to draw contours at a certain interval with labels, and a thicker pen, e.g.

```
set line width rapidograph 3x0
draw contour -2 5 1 0.25
set line width rapidograph 1
draw contour -2 5 1
```

Interpolation method: The interpolation scheme is the same used for converting grid-values to image values (Section 9.3.4.4 [Convert Grid To Image], page 79).

See also `set contour labels`

9.3.9.10 draw curve

Several forms exist.

'draw curve'

Draws a curve connecting the points (x,y), which have been read in by a command like read columns x y. Line segments are drawn between all (x,y) points, except: (1) no line segments are drawn to any missing data (see set missing value), and (2) if clipping is turned on (see set clip on), no line segments are drawn outside the clipping region. **See also** draw curve overlying

'draw curve overlying'

Like draw curve, except that before drawing, the area underneath the curve (+/- one linewidth) is whited out. This clarifies graphs where curves overlie other curves or the axes. **See also** draw curve.

'draw curve filled [to {.y. y}|{.x. x}]'

The form draw curve filled ... draws filled curves. If the to .value. is not specified, fill the region defined by the x-y points using the current paint colour (see set graylevel). To complete the shape, an extra line is drawn between the first and last points.

The form draw curve filled to .y. y fills the region between y(x) and y = .y.; do not connect the first and last points as in the case where to .yvalue. is not specified.

The form draw curve filled to .x. x fills the region between x(y) and x = .x.

9.3.9.11 draw essay

'draw essay "text"|reset'

Draw indicated text on the page. Succeeding calls draw text further and further down the page, starting at the top.

The current font size is used; to alter this, use set font size before draw essay.

When reset is present instead of text, the drawing position is reset to the top of the page. Use this after a new page command to ensure that the next text lines will appear at the top of the page as expected. EXAMPLE:

```
set font size 2 cm
draw essay "Line 1, at top of page"
draw essay "Line 2, below top line"
```

9.3.9.12 draw gri logo

'draw gri logo .x_cm. .y_cm. .height_cm. .style. \fgcolor \bgcolor'

Draw a Gri logo at given location with given style and colors. The lower-left corner of the logo will be .x_cm. centimeters from the left-hand side of the page and .y_cm. centimeters from the bottom of the page. The logo will be .height_cm. centimeters tall. The textual parameters \fgcolor and \bgcolor give the foreground and background colors, respectively, and these are used in styles as noted in the table below

```
.style.    style
=======    ==================
   0       stroke curve
   1       fill with color \fgcolor, no background
   2       fill with color \fgcolor it in tight box of color \bgcolor
```

```
3       as 2 but in square box
4       draw in \fgcolor on top of shifted copy in \bgcolor
```

An example is given below

```
draw gri logo 1 1 3 4 blue green
```

9.3.9.13 draw grid

```
'draw grid'
```

Draw plus-signs at locations where grid data are non-missing.

9.3.9.14 draw image histogram

```
'draw image histogram \
  [box .llx_cm. .lly_cm. .urx_cm. .ury_cm.]'
```

With no optional parameters, draw histogram of all unmasked parts of the image, placing it above the current top of the plot.

When the `box` options are present, they specify the box (in centimetre coordinates on the page) in which the histogram plot is to be done.

9.3.9.15 draw image palette

```
'draw image palette
   [axisleft|axisright|axistop|axisbottom]
   [left .left. right .right. [increment .inc.]]
   [box .xleft_cm. .ybottom_cm. .xright_cm. .ytop_cm.]'
```

With no optional parameters, draw palette for image, placed above the current top showing values ranging from `.min_value.` to `.max_value.` as given in `set image range`.

Optional keywords (`axisleft`, etc) control the orientation of the palette, the default being `axisbottom`.

The optional parameters `.left.` and `.right.` may be used to specify the range to be drawn in the palette. If the additional optional parameter `.inc.` is present, it specifies the interval between tics on the scale; if not present, the tics are at increments of 2 * (`.right.` - `.left.`). (If `.inc.` has the wrong sign, it will be corrected without warning.)

When the optional `box` parameters are present, they prescribe the bounding box to contain the palette. The units are centimetres on the page. If these parameters are not present, the box will be drawn above the image plot.

Hint It is a good idea to make the palette range `.left.` to `.right.` extend a little beyond the range of full white and full black, since otherwise neither pure white nor pure black will appear in the colorbar. For example

```
set image grayscale black 0 white 1 increment 0.1
draw image palette left -0.1 right 1.1 increment 0.1
```

Hint Continuous-tone images with superimposed contours are often effective. To get the contour lines drawn on the image palette, do something like this

```
draw image
.left.   = 0
.right.  = 9
.inc.    = 1
```

```
.space.  = 3
.height. = 1
draw image palette left .left. \
  right .right. \
  increment .inc. \
  box \
    ..xmargin.. \
    {rpn ..ymargin.. ..ysize.. + .space. + } \
    {rpn ..xmargin.. ..xsize.. +} \
    {rpn ..ymargin.. ..ysize.. + .space. + .height. + }
draw contour .left. .right. .inc. unlabelled

.c. = .left.
while {rpn .right. .c. <= }
  .c_cm. = {rpn .c. .left. - \
   .right. .left. - / \
    ..xsize.. * ..xmargin.. +}
  draw line from \
    .c_cm. \
    {rpn ..ymargin.. ..ysize.. + .space. + }\
    to \
    .c_cm. \
    {rpn ..ymargin.. ..ysize.. + .space. + .height. +} \
    cm
  .c. += 1
end while
```

9.3.9.16 draw image

`'draw image'`

Draw black/white image made by `convert grid to image` or by `read image`.

9.3.9.17 draw isopycnal

```
'draw isopycnal \
   [unlabelled] .density. [.P_sigma. [.P_theta.]]'
```

Draw isopycnal curve for a temperature-salinity diagram. This curve is the locus of temperature and salinity values which yield seawater of the indicated density, at the indicated pressure. The UNESCO equation of state is used.

For the results to make sense, the x-axis should be salinity and the y-axis should be either in-situ temperature or potential temperature.

The `.density.` unit is kg/m^3. If the supplied value exceeds 100 then it will be taken to indicate the actual density; otherwise it will be taken to indicate density minus 1000 kg/m^3. (The deciding value of 100 kg/m^3 was chosen since water never has this density; the more intuitive value of 1000 kg/m^3 would be inappropriate since water can have that density at some temperatures.) Thus, 1020 and 20 each correspond to an actual density of 1020 kg/m^3.

The reference pressure for density, `.P_sigma.`, is in decibars (roughly corresponding to meters of water depth). If no value is supplied, a pressure of 0 dbar (i.e. atmospheric pressure) is used.

The reference pressure for theta, `.P_theta.`, is in decibars, and defaults to zero (i.e. atmospheric pressure) if not supplied. This option is used if the y-axis is potential temperature referenced to a pressure other than the surface. Normally the potential temperature is, however, referenced to the surface, so that specifying a value for `.P_theta.` is uncommon.

By default, labels will be drawn on the isopycnal curve; this may be prevented by supplying the keyword `unlabelled`. If labels are drawn, they will be of order 1000, or of order 10 to 30, according to the value of `.density.` supplied (see above). The label format defaults to `"%g"` in the C-language format notation, and may be controlled by `set contour format`. The label position may be controlled by `set contour label position` command (bug: only non-centered style works). Setting label position is useful if labels collide with data points. Labels are drawn in the whiteunder mode, so they can white-out data below. For this reason it is common to draw data points after drawing isopycnals.

If the y-axis is in-situ temperature, the command should be called without specifying `.P_sigma.`, or, equivalently, with `.P_sigma.` = 0. That is, the resultant curve will correspond to the (S,T) solution to the equation

```
.density. = RHO(S, T, 0)
```

where `RHO=RHO(S,T,p)` is the UNESCO equation of state for seawater. This is a curve of constant sigma_T.

If the y-axis is potential temperature referenced to the surface, `.P_theta.` should not be specified, or should be specified to be zero. The resultant curve corresponds to a constant value of potential density referenced to pressure `.P_sigma.`, i.e. the (S,theta) solution to the equation

```
.density. = RHO(S, theta, .P_sigma.)
```

For example, with `.P_sigma.=0` (the default), the result is a curve of constant sigma_theta.

If the y-axis is potential temperature referenced to some pressure other than that at the surface, `.P_theta.` should be supplied. The resultant curve will be the (S,theta) solution to the equation

```
.density. = RHO(S, T', .P_sigma.)
```

where

```
T'=THETA(S, theta, .P_theta., .P_sigma.)
```

where `THETA=THETA(S,T,P,Pref)` is the UNESCO formula for potential temperature of a water-parcel moved to a reference pressure of `Pref`. Note that `theta`, potential temperature referenced to pressure `.P_theta.`, is the variable assumed to exist on the y-axis.

9.3.9.18 draw isospice

```
'draw isospice .spice. [unlabelled]'
```

Draw an iso-spice line for a `"TS"` diagram, using (S, T) data stored in files in a subdirectory named `iso-spice0` in a directory named by the unix environment variable `GRI_EOS_DIR`. You must set this environment variable yourself, in the normal unix way. If `GRI_EOS_DIR` is not defined, Gri looks in the directory '`/data/po/ocean/EOS/iso0`'; of course, this will work only for people on the same machine as the author.

Only certain iso-spice lines are stored in these files, so only certain values of `.spice.` are allowed. They are 21.75, 22.00, 22.25, ..., 30.75. You must supply `.density.` in exactly

this format (with 2 decimal places), or else Gri will not find the appropriate TS file, and will give a "can't open file" error. NB: isopycnals ranging from about 23.00 to 26.00 cross a TS diagram spanning 34<S<36 and 0<T<10.

The line is labelled at the right with the density value, unless the `unlabelled` option is given.

Clipping should be on when drawing iso-spice lines. A warning will be given if the isospice line does not intersect the clipping region.

EXAMPLE

```
set clip on
draw isospice line 27.00
draw isospice line 27.50 unlabelled
```

9.3.9.19 draw label

```
'draw label boxed "string" at .xleft. .ybottom. [cm]'
```

Draw boxed label for plot, located with lower-left corner at indicated (x,y) position (specified in user units or in cm on the page). The current font size and pen color are used. The geometry derives from the current font size, with the label being centered within the box.

9.3.9.20 draw label whiteunder

```
'draw label whiteunder "\string" at .xleft. .ybottom. [cm]'
```

Draw label for plot, located with lower-left corner at indicated (x,y) position (specified in user units or in cm on the page). Whiteout is used to clean up the area under the label. BUGS: Cannot handle angled text; doesn't check for super/subscripts.

9.3.9.21 draw label for last curve

```
'draw label for last curve "label"'
```

Draw a label for the last curve drawn, using the `..xlast..` and `..ylast..` built-in variables.

9.3.9.22 draw label

```
'draw label "\string" [centered|rightjustified] \
    at .x. .y. [cm|pt] \
    [rotated .deg.]'
```

With no optional parameters, draw string at given location in USER units.

With the `cm` or `pt` keyword is present, the location is in centimetres or points on the page.

With the `rotated` keyword present, the angle in degrees from the horizontal, measured positive in the counterclockwise direction, is given.

With the keyword `centered` present, the text is centered at the given location; similarly the keyword `rightjustified` makes the text end at the given location.

9.3.9.23 draw line from ... to

```
'draw line from .x0. .y0. to .x1. .y1. [cm|pt]'
```

With no optional parameters, draw a line from (.x0., .y0.) to (.x1., .y1.), where coordinates are in user units. With the cm or pt keyword present, the coordinates are in centimetres or points on the page. NOTE: This will not cause auto-drawing of axes.

9.3.9.24 draw line legend

```
'draw line legend "label" at .x. .y. [cm] [length .cm.]'
```

Draw a legend identifying the current line type with the given label. A short horizontal line is drawn starting at the location ('.x.', .y.), which may be specified in centimetres or, the default, in user coordinates. The line length is normally 1 cm, but this length can be set by the last option. The indicated label string is drawn 0.25 cm to the right of the line.

See also draw symbol legend ....

EXAMPLE (of keeping track of the desired location for the legend)

```
.offset. = 1                    # cm to offset legends
# ... get salinity data
set line width 0.25
draw curve
draw line legend "Salinity" at .x. .y.
# ... get temperature data
set line width 1.0
set dash 0.45 0.05
draw curve
.y. += .offset.
draw line legend "Temperature" at .x. .y.
```

9.3.9.25 draw lines

```
'draw lines {vertically .left. .right. .inc.} | \
    {horizontally .bottom. .top. .inc.}'
```

Draw several lines, either vertically or horizontally. This can be useful in drawing gridlines for axes, etc. The following example shows how to draw thin gray lines extending from the labelled tics on the x axis (ie, at 0, 0.1, 0.2, ... 1):

```
set x axis 0 1 0.1 0.05
set y axis 10 20 10
draw axes
set graylevel 0.75
set line width 0.5
draw lines vertically 0 1 0.1
set graylevel 0
```

9.3.9.26 draw patches

```
'draw patches .width. .height. [cm]'
```

With the optional cm keyword not present, draw column data z(x,y) as gray patches according to the grayscale as set by most recent set image grayscale. The patches are aligned along the horizontal, and have the indicated size in user units.

With the optional keyword cm is present, the patch size is specified in centimetres.

9.3.9.27 draw polygon

`'draw polygon [filled] .x0. .y0. .x1. .y1. [other pairs] [user|cm|pt]'`

Draw a polygon connecting the indicated points, specified in user units. The last point is joined to the first by a line segment. At least two points must be specified. If the `filled` keyword is present, the polygon is filled with the current pen color. If no unit is given, user units are used.

9.3.9.28 draw regression line

`'draw regression line [clipped]'`

Fit and draw a regression line to column data, of the form `y = ..coeff0.. + ..coeff1.. * x`, exporting `..coeff0..`, `..coeff0_sig..`, `..coeff1..` and `..coeff1_sig..` as global variables (Section 9.3.34 [Regress], page 113).

Normally, the line is not clipped to the axes frame, but it will be if the keyword `clipped` is given.

HINT: to label the plot you might do the following:

```
sprintf \label "y = %f + %f * x. R$^2$=%f" \
  ..coeff0.. ..coeff1.. ..R2..
draw title "The linear fit is \label"
```

9.3.9.29 draw symbol ... at

`'draw symbol .code.|\name at .x. .y. [cm|pt]'`

Draw a symbol at given (single) location. The location is normally in user coordinates; it will be in centimetres on the page if the optional `cm` or `pt` keyword is given.

With the optional numerical/name code specified, then the symbol of that number or name is drawn at each (x,y) datum, whether or not a z-column exists. The numerical/name codes are:

```
 # name               description
-- ----               -----------
 0 plus               +
 1 times              x
 2 box                box
 3 circ               circle
 4 diamond            diamond
 5 triangleup         triangle with base at bottom
 6 triangleright      triangle with base at left
 7 triangledown       triangle with base at top
 8 triangleleft       triangle with base at right
 9 asterisk           *
10 star               star of David
11 filledbox          filled box
12 bullet             filled circle
13 filleddiamond      filled diamond
14 filledtriangleup   filled triangleup
15 filledtriangleright filled triangleright
16 filledtriangledown  filled triangledown
17 filledtriangleleft  filled triangleleft
```

9.3.9.30 draw symbol legend

```
‘draw symbol legend \symbol_name "label" \
    at .x. .y. [cm]’
```

Draw indicated symbol at indicated location, with the indicated label beside it. The label is drawn one M-space to the right of the symbol, vertically centered on the indicated `.y.` location.

9.3.9.31 draw symbol

```
‘draw symbol [[.code.|\name]    \
    | [graylevel z]             \
      [color [hue z|.h.]        \
             [brightness .b.]   \
             [saturation .s.]]]’
```

With no optional parameters, draw symbols at the (x,y) data. If a z-column has been read with `read columns`, then its value codes the symbol to draw, according to the table below. (The value of z is first rounded to the nearest integer.) If no z-column has been read, the symbol X is drawn at each datum.

With the optional numerical/name code specified, then the symbol of that number or name is drawn at each (x,y) datum, whether or not a z-column exists. The numerical/name codes are:

```
 # name               description
-- ----               -----------
 0 plus               +
 1 times              x
 2 box                box
 3 circ               circle
 4 diamond            diamond
 5 triangleup         triangle with base at bottom
 6 triangleright      triangle with base at left
 7 triangledown       triangle with base at top
 8 triangleleft       triangle with base at right
 9 asterisk           *
10 star               star of David
11 filledbox          filled box
12 bullet             filled circle
13 filleddiamond      filled diamond
14 filledtriangleup   filled triangleup
15 filledtriangleright filled triangleright
16 filledtriangledown  filled triangledown
17 filledtriangleleft  filled triangleleft
```

With the optional `graylevel z` fields specified, the graylevel is given by the `z` column (0=black, 1=white).

With the optional `color` field specified, the color is specified, either directly in the command (the `hue .h.` form) or in the z column. For more information on color, refer to the `set color hsb ...` command.

Examples: both `draw symbol bullet color` and `draw symbol bullet color hue z` draw bullets whose hue is given by the value in the z column. The hue (or the color, in

other words) blends smoothly across the spectrum as the numerical value ranges from 0 to 1. The value 0yields red, 1/3 yields green, 2/3 yields blue, etc. If the `brightness` and the `saturation` are not specified, they both default to the value 1, which yields pure, bright colors.

Example: draw all in green dots `draw symbol bullet color hue 0.333 brightness 1 saturation 1`

Example: display spectrum of dots

```
set symbol size 0.3
open "awk 'END{ \
    for(c=0;c<1;c+=1/40) \
        print(c,c,c)}' | "
read columns x y z
close
draw symbol bullet color hue z
```

9.3.9.32 draw time stamp

```
‘draw time stamp \
    [fontsize .points. \
        [at .x_cm. .y_cm. cm \
            [with angle .deg.]]]’
```

Draw the command-file name, PostScript file name, and time, at the top of graph. Normally, the timestamp is drawn at the top of the page, in a fontsize of 10 points. But the user can specify the fontsize, and additionally the location (in cm) and additionally the angle measured in degrees anticlockwise from the horizontal.

NOTE: If you want to have the plot drawn in landscape mode, ensure that `set page landscape` precedes `draw time stamp`.

9.3.9.33 draw title

```
‘draw title "\string"’
```

Draw the indicated string above the plot.

9.3.9.34 draw values

```
draw values                 \
    [.dx. .dy.]             \
    [\format]               \
    [separation .xcm. .ycm.]
```

Draw values of `z` column, at corresponding (`x`, `y`) locations. If the `separation` keyword is present, the distance between successive points is checked, and points are skipped unless the x and y separations exceed the indicated distances.

- `draw values` Draw the values of `z(x,y)`, positioned 1/2 M-space to the right of `(x,y)` and vertically centred on `y`. The values are written in a good general format known as `%lg`, in C terminology.
- `draw values %.2f` Draw values of `z(x,y)` positioned as described above, but using the indicated format string. This format string specifies that 2 numbers be used after the

decimal place, and that floating point should be used. See any C manual for format codes.

- `draw values .dx. .dy.` Print values of `z(x,y)` at indicated offset vector (`.dx.`,`.dy.`), measured in centimeters, from the values of `(x,y)` at which the data are defined.
- `draw values .dx. .dy. %.3f` Print values of `z(x,y)` at indicated distance from `(x,y)`, indicated format.

9.3.9.35 draw x axis

```
draw x axis [at bottom|top|{.y. [cm]} [lower|upper]]
```

Draw an x axis, optionally at a specified location and of a specified style.

- `draw x axis` Draw a lower x axis (ie, one with the numbers below the line) at the bottom of the box defined by `set y axis`.
- `draw x axis at bottom` Draw a lower x axis (ie, one with the numbers below the line) at the bottom of the box defined by `set y axis`.
- `draw x axis at top` Draw an upper x axis (ie, one with the numbers above the line) at the top of the box defined by `set y axis` (or above any existing stacked x axes there)
- `draw x axis at .y.` Draw a lower x axis at indicated value of `.y.`.
- `draw x axis at .y. upper` Draw an upper x axis at indicated value of .y.

9.3.9.36 draw x box plot

```
draw x box plot at .y. [size .cm.]
```

Draw Tukey box plots (which give a summary of histogram properties). Box plots were invented by Tukey for eda (exploratory data analysis). The centre of the box is the median. The box edges show the first quartile (q1) and the third quartile (q3). The distance from q3 to q1 is called the inter-quartile range. The whiskers (i.e., the lines with crosses at the end) extend from q1 and q3 to the furthest data points which are still within a distance of 1.5 inter-quartile ranges from q1 and q3. Beyond the whiskers, all outliers are shown: open circles are used for data within a distance of 3 inter-quartile ranges beyond q1 and q3, and in closed circles beyond that.

As a side effect, this command stores q1, q2, and q3 into variables `..q1..`, `..q2..`, and `..q3..`.

- `draw x box plot at .y.` Draw Tukey's box plot, spreading in the x direction, centered at y=`.y.` and of default width 0.5 cm.
- `draw x box plot at .y. size .cm.` Draw Tukey's box plot, spreading in the x direction, centered at y=`.y.` and of width `.cm.` centimetres.

9.3.9.37 draw y axis

```
draw y axis [at left|right|{.x. cm} [left|right]]
```

Draw a y axis, optionally at a specified location and of a specified style.

- `draw y axis` Draw a left-hand-side y axis (ie, one with the numbers to the left of the line) at left of box defined by 'set x axis'
- `draw y axis at left` Draw a left-hand-side y axis (ie, one with the numbers to the left of the line) at left of box defined by `set x axis`.

- `draw y axis at right` Draw a right-hand-side y axis (ie, one with the numbers to the right of the line) at right of box defined by `set x axis`.
- `draw y axis at .x.` Draw a left-hand-side y axis (ie, one with the numbers to the left of the line) at indicated value of `.x.`
- `draw y axis at .x. right` Draw a right-hand-side y axis (ie, one with the numbers to the right of the line) at indicated value of `.x.`

9.3.9.38 draw y box plot

```
draw y box plot at .x. [size .cm]
```

Draw Tukey box plots (which give summary of histogram properties).

- `draw y box plot at .x.` Draw Tukey's box plot, spreading in the y direction, centered at x=`.x.` and of default width 0.5 cm.
- `draw y box plot at .x. size .cm.` Draw Tukey's box plot, spreading in the y direction, centered at x=`.x.` and of width `.cm.` centimetres.

As a side effect, this command stores q1, q2, and q3 into variables `..q1..`, `..q2..`, and `..q3..`.

9.3.9.39 draw zero line

```
draw zero line [horizontally|vertically]
```

Draw lines corresponding to x=0 or y=0.

- `draw zero line` Draw line y=0 if it is within axes.
- `draw zero line horizontally` Draw line y=0 if it is within axes.
- `draw zero line vertically` Draw line x=0 if it is within axes.

9.3.9.40 end grouip

```
end group
```

Command not implemented yet. Syntax may change.

9.3.10 expecting

```
expecting version a.b
expecting version a.b.c
```

Show a list of incompatibilites that have been introduced since the named version. This command can make your commandfiles more reliable against changes to Gri.

There are two forms of the version number. Modern versions use the triplet form, e.g. `expecting version 2.8.7`, but prior to October 1996 the version numbers were written in decimal form, so that you would write `expecting version 1.069` for example.

9.3.11 filter

- `filter column x|y|z|u|v|weight recursively a[0] a[1] ... b[0] b[1] ...` Filter indicated column, using a two-pass recursive filter. The first pass runs from the start to the end, while the second pass runs from the end to the start; in this way, the

phase shift inherent in this type of filter is removed entirely. The coefficients are used in the following formula (demonstrated on the `x` column):

```
x_new[i] = b[0] * x[i] \
  + b[1] * x[i-1] \
  + b[2] * x[i-2] \
  + ... \
  - a[1] * x_new[i-1] \
  - a[2] * x_new[i-2] \
  - ...
```

Thus, for example, setting `a[i]` = 0 results in a simple backwards-looking moving-average filter applied in two passes. The real power of this type of filter, however, comes when non-zero `a[i]` coefficients are given, thus adding recursion (i.e., `x_new[i]` depends on `x_new[i-...]`). See any standard reference on digital filters for an explanation. You might find that the Matlab command `butter` an easy way to design filter coefficients. Here are some examples:

```
# Filter x column with simple 2-point moving
# average.  (This slurs into a 3-point moving
# average, in effect, since the filter is run
# forwards and then backwards.)
filter column x recursively 0 0 0.5 0.5

# Use filter designed with the Matlab
# command butter(2,0.1), which creates a
# 2nd order lowpass butterworth filter
# with a cutoff frequency of 0.1
# (in units which have a frequency
# of 1 corresponding to one-half the
# sampling rate).
filter column x recursively \
    1     -1.561  0.6414 \
    0.0201 0.0402 0.0201
```

- `filter grid rows|columns recursively a[0] a[1] ... b[0] b[1] ...` Apply recursive filter (see `filter column ... recursively` for meaning of this filter operation) to the individual rows or columns of the grid data. For example, the command `filter grid columns recursively 0 0 0.5 0.5` applies a 2-point moving average filter across the columns, smoothing the grid in the x-direction.
- `filter image highpass` Remove low-wavenumber components from image (ie, sharpen edges). Do this by subtracting a Laplacian smoothed version of the image.
- `filter image lowpass` Remove high-wavenumber components from image (ie, smooth shapes). Do this by Laplacian smoothing.

See also Section 9.3.45 [Smooth], page 137.

9.3.12 `flip`

`'flip grid|image x|y'`

Flip grid or image by relecting it about a horizontal or vertical centerline.

- `flip grid x` Flip grid so right-hand side becomes left-hand side.
- `flip grid y` Flip grid so bottom side becomes top side.

- `flip image x` Flip image so right-hand side becomes left-hand side.
- `flip image y` Flip image so bottom side becomes top side.

9.3.13 get env

`'get env \result \environment_variable'`

Get the value of an "environment variable" from the unix operating system, and store the result in the indicated synonym. This makes most sense on unix systems (hence the name, patterned after the unix command `getenv`). This command can be useful in making gri programs resistant to changes in data-file locations. Suppose, for example, there is a file called '`data`', normally in a local directory called `Bravo`. The line `open Bravo/data` will fail if the Bravo directory is moved. But if the name of the datafile is stored in an unix environment variable, `DIR_BRAVO` say, then the gri program will work no matter where the Bravo data are moved, so long as an appropriate environment variable is modified when the data are moved. Example:

```
get env \dir DIR_BRAVO
if {rpn "\dir" "" ==}
    show "Cannot determine location of the Bravo data,"
    show "which should be stored in the environment"
    show "variable DIR_BRAVO.  You should"
    show "do something like"
    show "export DIR_BRAVO='/data/Bravo/'"
    show "in your ~/.environment file"
    quit
end if
open \dir/data
...
```

9.3.14 group

`'group ["\name"]'`

Command not implemented yet. Syntax may change.

9.3.15 heal

`heal columns|{columns along x|y}`

The `heal` command heals over gaps in either columnar or gridded data. This is done by linear interpolation across the missing-value gaps.

- `heal columns` Fill in missing values in x, y, z, ... columns, by linear interpolation to neighboring valid data. All gaps in the data will get replaced by a linear function of index which matches the data at the indices just before and just after the gap. For example, if the y data were like

```
111
3
-9
-9
-9
7
333
```

where `-9` is the missing-value code, then they would get replace by

```
111
3
4
5
6
7
333
```

Notes: (1) This is done **independently** for all existing columns. (2) Gaps at the start and end of the columns are not filled in.

- `heal grid along x` Scan in the x direction, filling in missing values by linear interpolation. Since this uses the the x-grid, you must first have done `read grid x` or `set x grid`.
- `heal grid along y` Scan in the y direction, filling in missing values by linear interpolation. Since this uses the the y-grid, you must first have done `read grid y` or `set y grid`.

9.3.16 help

```
'help [*|command_name|{- topic}]'
```

Give help on a command or topic.

- `help` Print a general help message.
- `help *` Prints complete help info.
- `help command_name` Prints help on the command whose name begins with the string `command_name`. The string may be several words long; e.g. `help set` or `help set x axis`.
- `help - topic_name` The minus sign tells Gri that the string to follow it is a topic, not a command. Topics Gri knows about are listed by the one-word `help` request.

9.3.17 if

(**See also** Section 10.6 [If Statements], page 157.)

```
'if {[!] .flag.}|\flag|{{"string1" == "string2"}}'
```

Control program flow. The `if` block is ended with a line containing `end if`. Optional `else` and `else if` blocks are allowed. Note that rpn expressions are allowed, and a special form of string comparison is allowed, as in the examples below.

```
if .flag.
  # List of Gri commands to be done if .flag. is 1.
  # This list may extend across any number of lines.
end if
```

If the variable `.flag.` is not equal to 0, do the code between the `if` line and the `end if` line.

```
if .flag.
  # Commands done if .flag. is 1
else
  # Commands done if .flag. is 0
end if
```

If the variable `.flag.` is not equal to 0, do the code between the `if` line and the `else` line. If `.flag.` is equal to 0, do the code between the `else` line and the `end if` line.

```
if ! .flag.
  # Commands done if .flag. is 0
end if
```

If the variable `.flag.` is equal to 0, do the code between the `if` line and the `end if` line.

```
if {rpn .flag. 10 <}
  # Commands done if 10 is less than .flag.
end if
```

If the variable `.flag.` is greater than 10, do the code between the `if` line and the `end if` line.

```
if \smooth
  # Commands done if \smooth is 1
else
  # Commands done if \smooth is 0
end if
```

If the number stored in the synonym `\smooth` is not equal to 0, do the code between the `if` line and the `else` line. If the synonym stores a representation of a number not equal to zero, do the `else` part. If the synonym contains text that does not decode to a number, generate error message.

```
if {"\item" == "Temperature"}
  # Commands done if the synonym \item is equal to the
  # indicated text string.
end if
```

If the synonym `\item` has the value `Temperature` stored in it, do the indicated code.

```
if {rpn "\item" "Temperature" ==}
  # Commands done if the synonym \item
  # equals indicated text string.
end if
```

As above, but using the `rpn` calculator (Section 10.9 [rpn Mathematics], page 161).

```
if {rpn "\item" "Temperature" !=}
  # ...
end if
```

As above, but do the indicated code if `\item` is **not** equal to `Temperature`.

9.3.18 ignore

```
'ignore last .n.'
```

Ignores last `.n.` lines read by `read columns`.

9.3.19 input

```
'input \ps_filename \
  [.xcm. .ycm. \
    [.xmag. .ymag. \
      [.rot_deg.]]]'
```

Input the named PostScript file directly into the Gri output PostScript file. (If the filename has punctuation, insert it in double quotes, e.g. `input "../thefile"`.) If no options are specified, the file is input at normal scale, with normal margins. (Aside to PostScript programmers: the named file is sandwiched between `gsave` and `grestore` commands.) If `.xcm.` and `.ycm.` are specified, then the origin is moved to the named location first. If, in addition, `.xmag.` and `.ymag.` are specified, then these are used as scale factors after translation. Finally, if `.rot_deg.` is specified in addition, then the indicated counterclockwise rotation is applied after translation and scaling. Hint: if the results look wrong, the first thing to do is to think carefully about the order of the (translation, scaling, rotation) operations.

9.3.20 `insert`

```
'insert \filename'
```

Perform the commands in the indicated file.

If the file name is absolute (i.e. starts with `.`, or with `/` or with `~`) then an error results if the file is not present (or cannot be read by this user).

However, if the file name starts with a normal letter, Gri will try harder to locate the file. If it is not in the local directory, and if a `set path to "PATH" for commands` has been done, then Gri will search the colon-separated directories for the file (Section 9.3.41.39 [Set Path To], page 129).

If you don't want path-searching done, use the `source` command instead (Section 9.3.46 [Source], page 138).

9.3.21 `interpolate`

```
interpolate x grid to .left. .right. .inc.|{/.cols.}'
interpolate y grid to .bottom. .top. .inc.|{/.rows.}'
```

Transform grid by interpolating between existing grid data, according to a new `x` or `y` grid specified in the manner of `set x grid` and `set y grid`. Note that the new grid is neccessarily regular, while the first grid needn't have been. The data of the new grid are constructed by interpolation, using the same interpolation algorithm as the `convert grid to image` command.

9.3.22 `list`

```
'list \command-syntax'
```

List the source of a gri command. Often this is just the name of a C function internal to gri (try `list list` for an example), but when the command is written in the gri programming language the source will be more understandable (try `list set panel`).

9.3.23 `ls`

```
'ls [\file_specification]'
```

List files in current directory. (The current directory can be printed by the gri command `pwd` and can be set by the gri command `cd`.) `ls \file_specification` lists files in current directory which match the file specification. Normal unix file specification options are understood.

9.3.24 mask

```
'mask the image [to {uservalue .u.}|{imagevalue .i.}]'
```

Examine both the image and the mask pixel by pixel. For any pixels which have a mask value of 1 (which indicates an invalid region of the image), change the image value. If no `to` phrase is present, change the image value to 0 in pixel units. If the `to uservalue .u.` phrase is present, change the pixel to hold the imagevalue that corresponds to this uservalue (see `set image range` command for a discussion of this correspondance). If the `to imagevalue .i.` phrase is present, change the pixel to hold that imagevalue (in range 0 to 255 inclusive for 8-bit images).

9.3.25 new

```
new .variable_name. | \synonym_name \
   [.variable_name. | \synonym_name \
   [...]]
```

`new` sets aside storage for new version of the named variable(s) and/or synonym(s). Any number of variables and synonyms may be specified. If a given variable/synonym already exists, this will create a new version of it, and future assignments will be stored in this new version **without** affecting the pre-existing version. If the variable/synonym is `delete`ed, the new version is deleted, making the old, unaltered, version accessible again.

This command is used mostly for temporary use, to prevent clashing with existing values. Suppose you want to change the font size inside a new command or an if block. Then you might do the following, where the variable `.tmp.` is used to store the old font size. Note that the use of the `new/delete` statements prevents the assignment to the local version of the variable `.tmp.` from affecting the value known outside the `if` block, if in fact `.tmp.` happened to exist outside the block.

```
set font size 10
draw label "This is in fontsize 10" at 10 2 cm
if .want_title.
  new .tmp.
  .tmp. = ..fontsize..
  set font size 22
  draw label "This is 22 font" at 10 5 cm
  set font size .tmp.
  delete .tmp.
end if
draw label "This is 10 font" at 10 8 cm
```

Special case: for local synonyms (e.g. `\.word1.`, etc.), the `new` operator checks to see whether the synonym is standing for an "ampersand" argument, signalling a changeable argument that is a variable or a synonym. In such a case, `new` creates a new instance of the item in the calling context. The test suite has examples (Chapter 16 [Test Suite], page 233).

9.3.26 new page

```
'new page'
```

Finish the present page, and start a new page. All settings (of linewidth, axes, landscape/portrait, etc) are retained on the new page. Among these settings is the flag that tells gri whether you need axes plotted along with your data.

9.3.27 `new postscript file`

`'new postscript file "name"'`

Finish the present Postscript file, and start a new page with the given name. All settings (of linewidth, axes, landscape/portrait, etc.) and data are retained on the new file.

9.3.28 `open`

There are two styles of `open` command. In the first style, a simple file is to be opened. In the second style a unix-like "pipe" is opened, i.e. Gri will read the output of a system command instead of a file.

9.3.28.1 Opening simple files

Ascii Files Most applications involve ascii files, and these are very easy to handle in Gri. For example given a data file named '`foo.dat`', just use the command

```
open foo.dat
```

and then you can read the data using various commands. Thus a complete program might be

```
open foo.dat
read columns x y
draw curve
```

If a filename contains blanks or punctuation symbols, you must put it in double quotes ("), e.g.

```
open "foo bar.dat"
```

Indeed, Gri accepts double-quotes on any `open` command and some folks use it on all commands, as a matter of habit.

Gri can handle compressed files appropriately, e.g.

```
open foo.data.gz
```

so that there is no need to uncompress data for use with Gri.

Gri is quite persistant in looking for your file, and if a given file is not found, it will then check to see if a compressed version is available, and use that instead. Thus

```
open foo.dat
```

will look for a file named '`foo.dat.gz`' if '`foo.dat`' is not available. (Only files compressed with the GNU `gzip` utility are handled.)

If the `open` command was successful in opening the file, it will set the value of the synonym `\.return_value.` to the **full** pathname of the file. Thus, if `open a.dat` is done in directory `/home/gri`, then `\.return_value.` will equal the string `/home/gri/a.dat`.

Binary Files Like most computer programs, Gri has some trouble with binary files. One big issue is the so-called "endian" character of the computer. Some computers store multi-byte values with the most significant bytes first, while others store them with the most significant bytes last. The problem is that nothing is stored in data files to indicate which convention was employed. For this reason, a version of Gri compiled on a so-called "big-endian" computer will misinterpret multi-byte values that were created on a so-called "little-endian" computer. Many folks in the scientific community have converted to using the

NetCDF format (see next section) for precisely this reason, since this format is independent of the endian character of the computer.

Presuming an appropriate endian character, however, reading is straightforward. A command of the form

```
open foo.dat binary
```

tells Gri that the data are stored in a binary format. With the above syntax, Gri expects images to be in `unsigned char` (8 bits), while other data, such as columns and grids, are expected to be in 32-bit format (suitable for reading into a so-called "float" variable in the C programming language).

You may also specify the format directly, as in the following examples; Gri then interprets all data as being in the indicated format and then converts to the internal format before using the data.

```
open \filename binary uchar
open \filename binary 8bit
open \filename binary int
open \filename binary float
open \filename binary double
open \filename binary 16bit
```

As with ascii files, Gri will automatically uncompress any files that are compressed, and if it fails to find a given filename, it will try to open a compressed version of it (i.e. one with a '`.gz`' suffix).

NetCDF Files The NetCDF format provides the best of both worlds. It is binary, so that data are relatively compact, and may be read very quickly. (Reading ascii data is time-consuming in C++, the language in which Gri is written.) But it does not suffer the endian problem problem of normal binary files (see previous section), since information about the endian character is stored in the file itself, and Gri uses this information to decode the data without difficulty, regardless of the endian characteristics of the computer on which Gri is running and of the computer that created the data.

For more information on netCDF format, see

`http://www.unidata.ucar.edu/packages/netcdf/index.html`

The syntax of opening NetCDF files is as below

```
open foo.nc netCDF
```

and the syntax for reading such files is described in sections on the various `read` commands (see e.g. Section 9.3.33.2 [Read Columns], page 107).

9.3.28.2 Opening pipes

Sometimes it makes sense to get Gri to work with the results of another command in the OS. Gri handles this by creating a so-called "pipe", thus reading the output from the other command. (Readers familiar with the unix OS will know what pipes are all about, and especially why they are a good thing. Other readers might wish to skip this section.)

Suppose we wish to plot an x-y plot using just the first few lines of a datafile named '`foo.dat`'. Unix users will know that a good way to see the first few lines of such a file would be to type the command `head foo.dat`. They also know that these lines could be provided to a second unix command, named '`do_foo`' say, by the command `head foo.dat | do_foo`. This uses a so-called "pipe", designated by the vertical line (called a pipe symbol below).

Gri can read the output from system commands by using a syntax in which the (quoted) system command ends in a pipe symbol, e.g.

```
open "head foo.dat |"
```

as in the example above.

Aside: When pipe-open commands are used, Gri creates a temporary file (often located in '`/usr/tmp`', but that varies with machine). This is automatically cleaned up when Gri completes executation, but if Gri dies (or is interrupted) before it finishes, you'll be left with an extra file in this temporary-storage directory. It's up to you to clean that directory up from time to time.

Some common examples of pipe-open commands are given below.

1. **Comma-separated values** are common in files created by, or intended for, spreadsheets. Since Gri expects data elements to be separated by blanks (or tabs), you'll have to convert the commas into blanks. There are many ways to do that using pipes, e.g. '`sed`' system utility, e.g.

   ```
   open "sed -e 's/,/ /g' foo.dat |"
   ```

   Other unix facilities, such as `tr` will also work, of course. If the file has headers, you'll want to remove them also. This can be done with the `skip` command (Section 9.3.43 [Skip], page 136) but you could also do it at the open stage, e.g. to remove the first two lines, use

   ```
   open "sed -e 's/,/ /g' foo.dat | tail +2 |"
   ```

2. **Manipulating column data** is done by e.g.

   ```
   open "cat foo.dat | awk '{$1, $2 * 22}' |"
   ```

   where '`awk`' has been used to multiply the second column in the file named '`foo.dat`' by 22.

3. **Time-based and geographical data** are sometimes encountered. For an example, suppose that longitude/latitude (i.e. x/y) data are stored in Hour.minutesecond format, e.g. 12.2133 means hour 12, minute 21, second 33. Gri doesn't read HMS format, but gawk can be told to:

   ```
   open "cat datafile.HMS |        \
       awk '{                      \
       split($1, hms, \".\");      \
       h = hms[1];                 \
       m = int(hms[2] / 100);      \
       s = hms[2] - 100 * m;       \
       x = h + m / 60 + s / 3600;  \
       split($2, hms, \".\");      \
       h = hms[1];                 \
       m = int(hms[2] / 100);      \
       s = hms[2] - 100 * m;       \
       y = h + m / 60 + s / 3600;  \
       print(x,y)                  \
       }' | "
       read columns x y
   ```

4. **Timeseries data** are often stored in formats that blend letters and numbers. For one thing, using letters (e.g. `aug`) removes an ambiguity in numerically-based data. (Example: 02/03/2000 means one thing to an American and another thing in the rest of the world. However, everybody agrees on what 2000-Feb-03 means.) Suppose, for example, that we have data in a format such as

```
Tue_Jul_25_11:07:51 0.62
Tue_Jul_25_11:22:51 0.59
Tue_Jul_25_11:37:51 0.56
```

(stored in a file called 'foo.dat' say) and we want a graph of the y-variable (0.62, 0.59, 0.56) versus x-variable, time expressed say as seconds in the day. Then here is how that could be done:

```
open "cat foo.dat |\
    sed -e 's/_/ /g' -e 's/:/ /g' |\
    awk '{print ($4*3600+$5*60+$6, $7)}' |"
read columns x y
draw curve
```

Note that the actual day information is skipped in this example; seasoned `awk` users could easily fill in the code to handle datasets spanning several days.

9.3.28.3 Opening URLs

Gri can open a URL, *if* you have the `wget` program on your machine. (`wget` is available from the GNU website `http://www.gnu.org/software/wget/`.)

The URL must be enclosed in quotes (since otherwise, Gri will interpret the `//` sequence as indicating an old way of denoting comments). For example,

```
open "http://gri.sourceforge.net/gridoc/examples/example1.dat"
read columns x y
show columns
```

If you don't have `wget` installed on your machine, the above won't work, but you can always use another fetching program, with a system call, as in the following:

```
\url = "http://gri.sourceforge.net/gridoc/html/examples/example1.dat"
open "lynx -dump \url |"
read columns x y
draw curve
```

9.3.29 postscript

```
'postscript \string'
```

Write the indicated string to the PostScript output file, after substitution of synonyms if there are any. Example:

```
\a = "45"  # angle
\w = "8.5" # page width
postscript gsave \w 72 mul 0 translate \a rotate
# ... other code to do stuff
postscript grestore
```

Here is how to draw an image palette vertically instead of horizontally:

```
\X = "3" # cm
\Y = "10" # cm
\a = "90"       # degrees counterclockwise
postscript gsave \X 28.35 mul \Y 28.35 mul translate \a rotate
# Palette is at user's origin
draw image palette box 0 0 10 1
postscript grestore
```

NOTE: the `postscript` command is **very** dangerous, and should normally only be used by developers. Most of the code concerning this is in the file '`doline.cc`'; look for the string `postscriptCmd` to find the relevant code.

9.3.30 `pwd`

```
'pwd'
```

Print current directory (which can be set by `cd`).

9.3.31 `query`

```
'query \synonym|.variable. \
  ["\prompt" ["\default"|.default.]]'
```

Ask the user for the value of a variable (number) or synonym (text string). Gri recognizes the type of the item being asked for, either a variable or synonym, by the presence of a dot or backslash in the second word of the command line. If a prompt string is given (in quotes), then this string is shown to the user. If a default is given (in parentheses), then it will be displayed also, and if the user types carriage-return, then that item will be assigned to the variable or synonym. If the default has more than one item, then Gri considers this a restrictive list of possibilities, and will demand that the answer be in that list, going into an infinite query loop until an item from the list (or carriage-return, meaning take first item) is found. The items in the list are to be separated by spaces, not commas or any other non-whitespace characters.

NOTE: The `-y` command-line option bypasses all query commands, fooling Gri into thinking that the user typed a carriage-return to all questions. Thus the defaults, if they exist, are selected.

9.3.32 `quit`

```
'quit [.exit_status.]'
```

Exits the gri program. If an exit status (`.exit_status.`) is specified, then Gri returns this value, rounded to the nearest integer, as the "exit status" (a concept meaningful mostly in the unix environment, where it designates an error).

9.3.33 The `read` commands

There are several varieties of `read` command. Those commands used for reading numerical information (e.g. `read columns`) are able to decode variables and synonyms as well as simple numbers.

9.3.33.1 `read colornames`

```
'read colornames from RGB "\filename"'
```

Read colornames from named file, which is in the X11 format. This format has 4 or more columns, the first three giving the red, green and blue values in the range 0 to 255, and the last columns giving the colorname (which may have more than one word). You can create colors yourself or read an X11 color file. In many cases you will want to `read colornames from RGB "/usr/lib/X11/rgb.txt"`. Full filenames must be used; the '~' syntax is not

permitted. Once you have read in a colorname table, the named colors may be used as builtin colors (Section 9.3.41.7 [Set Color], page 117). To view the colors available on your particular system, use the Unix command `xcolors` or `excolors`; to see the RGB values of all colors on your X11 system, use the `showrgb` unix command.. To view the names and RGB values of the colors Gri knows, including builtin ones and ones from `read colornames`, use `show colornames`.

This command is akin to `set colorname` (Section 9.3.41.8 [Set Colorname], page 118), except that the latter uses the Gri notation of color constituents being in the range from 0 to 1, whereas for `read colornames` uses an X11 database, so that the color constitutents range from 0 to 255.

9.3.33.2 `read columns`

`'read columns ...'`

Read numbers into columns. These columns have predefined meanings and names. For example, `read columns x y` instructs Gri to read data into columns called `x` and `y`; it is these data that Gri will use if you tell it to `draw curve`. Other columns are: `z`, used for contouring a function `z=z(x,y)`; `weight`, used for weighting data points; `u` and `v`, used for arrow (vector) plots.

If the keyword `appending` is given as the last word on the `read columns` line, then the new data will be appended to any existing columnar data; otherwise they will overwrite any existing data.

As a special case, if the `x` column is not indicated (e.g. `read columns y`) then Gri creates x-values automatically, in the sequence 0, 1, 2, etc.

- `read columns x y` Read `x` in column 1, `y` in column 2 until blank-line found. Only the first two numbers on each line will be read; any extra numbers (or words) on the line will be ignored.
- `read columns * y * * x` Read `x` in column 5, `y` in column 2. The `*` character is a spacer. It instructs Gri to skip the first, third, and fourth words on the data line. These words need not be numbers. This example illustrates a general mechanism of using the `*` character to skip over unwanted items in the data file. Note that there is no need to supply `*` characters for trailing extraneous words; Gri will skip them anywary. Finally, note that any order of `x` and `y` (and the other columns; see below) is allowed.
- `read columns y=2 x=5` or `read columns x=5 y=2` As above; read `x` in column 5 and `y` in column 2. The column number may be specified in this manner for all the named column variables. No spaces are allowed before or after the `=` sign. The first column is called column 1. Whether this format is used or the `*` format is a matter of choice, except that numbered format also permits using a given number to fill several variables (for example `read columns x=1 y=2 u=1 v=2`).
- `read columns x="netCDF_name" ...` If the file is a `netCDF` file, opened by e.g. `open myfile.nc netCDF`, then the `netCDF` variables for the columns, e.g.

  ```
  open latlon.nc netCDF
  read columns x="longitude" y="latitude"
  ```

  Note: the data **must** be stored as the `netCDF` "float" type.

  For more information on netCDF format, see

  `http://www.unidata.ucar.edu/packages/netcdf/index.html`
- `read columns * y z * x` Read `x` in column 5, `y` in column 2, and `z` in column 3. The `z` column is used for contouring.

- `read columns x y u v` Read `x` and `y` in first two columns, and the "arrow" data `u` and `v` as third and fourth columns.
- `read columns .rows. x y` Read `.rows.` rows of column data.

Sometimes you'll have `x` in one file and `y` in another. In that case, use the operating system or an editor to put the columns in one file. In unix, the easy way is

```
open "paste file_with_x file_with_y |"
read columns x y
```

NOTE FOR BINARY FILES: For ascii files, Gri will proceed to a new line after it has read the items requested; it skips any words appearing on the data line after the last object of interest. Thus `read columns x y` will read the first two columns and ignore any other columns that might be present. But for binary files, Gri has no way of knowing how to "skip" to the next line (see `skip` command), so you will have to flesh out the `read columns` command with as many spacers as are present in your data. For example, if you have four numbers in each data record and want to interpret the first two as `x` and `y`, you would use `read columns x y * *` to read the data.

RETURN VALUE: Sets `\.return_value` to `N rows N non-missing N inside-clip-region`

9.3.33.3 `read grid`

`read grid` commands read grid characteristics. (The "grid" is the object that is contoured.)

For normal ascii or binary files, the commands to read the grid's x-locations, y-locations and data are:

```
'read grid x [.rows.]'
'read grid y [.rows.]'
'read grid data [spacers] \
  [.rows. .cols.] [spacers] [bycolumns]'
```

For `netCDF` files, the commands are as follows (note that it is not possible to specify the number of data to read, nor to read the grid by columns).

```
'read grid x = "variable_name"'
'read grid y = "variable_name"'
'read grid data = "variable_name"'
```

The ordering of the y-grid data is the same as if they were read from a normal file: the first number is considered to be at the top of the plot.

For more information on netCDF format, see

`http://www.unidata.ucar.edu/packages/netcdf/index.html`

Details of the non-netCDF commands:

- `read grid x [.cols.]` Read the `x` locations of the grid points, one number per line. If `.cols.` is supplied, then that many values will be read; otherwise, reading will stop at end-of-file or blank-line.
- `read grid y [.rows.]` As above, but for y grid; `.rows.` is the number of rows. The first number to be read corresponds to the location of the **top** edge of the grid. Thus, if you were to view the column of numbers with a text editor, they would be oriented the same way as the corresponding elements will appear on the page.
- `read grid data [.rows. .cols.]` Read data for a grid having `.rows.` and `.cols.` columns. (If `.rows.` and `.cols.` are not supplied, but the grid already exists, then those pre-existing values are used. If they are specified here, then they are checked

for consistency with the pre-existing values if they exist.) Gri will read `.rows.` lines, each containing `.cols.` numbers. (Extra information in the file can be skipped; see discussion of the `*` keyword below.) Gri will interpret the first line it reads as the grid data corresponding to a value of y equal to `y[.rows.]`. Thus, file should be arranged like this:

```
f(x[1], y[.rows.])  ...  f(x[.cols.], y[.rows.])
        .
        .
        .
f(x[1], y[3])       ...  f(x[.cols], y[3])
f(x[1], y[2])       ...  f(x[.cols], y[2])
f(x[1], y[1])       ...  f(x[.cols], y[1])
```

- `read grid data [.rows. .cols.] bycolumns` As above, but the `bycolumns` keyword tells Gri to read the data one column at a time, instead of one row at a time. Each line is expected to contain `.rows.` numbers (as opposed to `.cols.` numbers, as in the format where the `bycolumns` keyword is not present). (Extra information in the file can be skipped; see discussion of the `*` keyword below). The first line of the data file contains the first column of the gridded data, corresponding to x equal to `x[1]`). The file should look like this:

```
f(x[1], y[1])     ...     f(x[1], y[.cols.])
f(x[2], y[1])     ...     f(x[2], y[.cols.])
f(x[3], y[1])     ...     f(x[3], y[.cols.])
        .
        .
        .
f(x[.rows.],y[1]) ...  f(x[.rows.], y[.cols.])
```

- `read grid data * * [.rows. .cols.]` As `read grid data .rows. .cols.` except that the first two words on each line are skipped. As usual, trailing extraneous numbers are also skipped.

See also `set x grid`, `set y grid`

RETURN VALUE:

`read grid x` sets `\.return_value` to `N cols`

`read grid y` sets `\.return_value` to `N rows`

`read grid data` sets `\.return_value` to `N rows N cols`

9.3.33.4 `read image colorscale`

`'read image colorscale [rgb|hsb]'`

Read colorscale for image, from 256 lines each containing values for Red, Green, and Blue (or Hue, Saturation and Brightness), separated by whitespace. The values are expected to be in the range 0 to 1, and are clipped to these limits if not.

For hints on how to create such an input file, Section 9.3.33.5 [Read Image Grayscale], page 110. If the example given there has the following code instead,

```
open "awk 'BEGIN {                \
     for(i=0;i<256;i++) {         \
       print((i - 50)/50, 1, 1)   \
     }                            \
   }' |"
```

```
read image colorscale hsb
```

then a linear full-color spectrum running from red at 10C to magenta at 15C is achieved.

9.3.33.5 `read image grayscale`

‘`read image grayscale`’

Read grayscale for an image, from 256 lines each containing a single value. The values are expected to be in the range 0 to 1, and are clipped to these limits if not. For 8-bit images, Gri multiplies these values by 255, and uses this list for the grayscale mapping. Such a list is created by `write image grayscale`.

As an example, consider the code fragment (Chapter 7 [Images], page 31).

```
set image range 5 30.5
set image grayscale black 10 white 15
```

is equivalent to

```
set image range 5 30.5
open "awk 'BEGIN {\
  for(i=0;i<256;i++) {\
    print(1-(i-50)/50)\
  } \
}' |"
read image grayscale
close
```

because the image formula is

Temperature = 5C + 0.1C * pixelvalue

where the pixelvalue ranges from 0 to 255. Therefore, a temperature of 10C is a pixelvalue of 50, and 15C is 100. To get a grayscale ranging between these values, therefore, we create a linear function which maps the 50th pixelvalue into grayvalue 1, and the 100th pixelvalue into grayvalue 0. That is what the awk line does; to see the actual numbers, you could insert the line `write image grayscale to TMP` and look at the file ‘`TMP`’ (bear in mind that Gri will clip the values to the range 0 to 1).

Sometimes you will have a file, say named ‘`map.dat`’, with RGB numbers in the range 0-255, rather than 0-1 as Gri requires. To read them, use the operating system to convert the numbers for you (Section 9.3.28 [Open], page 102).

```
open "cat map.dat \
  | awk '{print(($1+$2+$3)/3/255)}' |"
read image grayscale
close
```

9.3.33.6 `read image mask`

‘`read image mask rasterfile|{.rows. .cols.}`’

Read image mask. The mask is associated with the image read in by the `read image` command in the following way. When computing image histograms, Gri ignores any pixels in the image for which the corresponding pixel in the mask is set to `1`.

- `read image mask rasterfile` The image size is specified in the rasterfile file itself, so it is not specified.
- `read image mask .rows. .cols.` The file must contain `.rows.*.cols.` binary data.

Pixel order is the same as for images.

9.3.33.7 `read image`

There are several types of `read image` commands, depending on the file format. If the file is "raw", with no embedded information about things like the width and height, then we need to specify everything, as in the first format given below. The other formats make use of the header information in, e.g. PGM files.

Headerless images

```
'read image .rows. .cols. \
  [box .xleft. .ybottom. .xright. .ytop.] \
  [bycolumns]'
```

With no options specified (`read image .rows. .cols.`), read binary data defining an `image`. The image range must have previously have been set by `set image range`. The data are as written by "unsigned char" format in C.

When the `box` option is specified, the geometry of the image, in user coordinates, is specified in terms of the cartesian coordinates of the lower-left corner (`.xleft.`, `.ybottom.`) and upper-right corner (`.xright.`, `.ytop.`). If the `box` option is not specified, this geometry can be specified with either `read x grid` or `set x grid`, plus either `read y grid` or `set y grid`.

With the `bycolumns` keyword present, the image is read sweeping from top-to-bottom, then left-to-right, instead of the usual order.

Sun rasterfile images

```
'read image rasterfile [box .xleft. .ybottom. .xright. .ytop.]'
```

Read image in Sun rasterfile format. Image geometry is inferred from the header, so `.rows.` and `.cols` parameters are not given.

PGM images

```
'read image pgm [box .xleft. .ybottom. .xright. .ytop.]'
```

Read image in PGM (Portable Gray Map) format. Image geometry is inferred from the header, so `.rows.` and `.cols` parameters are not given. Both ascii and binary PGM formats are supported (that is, files with magic characters of P2 and P5).

NOTE that there are many image formats and Gri doesn't try to deal with them all. The idea is to use another program to convert images to a file format that Gri understands. In the future Gri may support PNG and other popular formats, especially in the Linux versions, for which libraries exist to ease the input.

9.3.33.8 `read from \filename`

```
'read from \filename'
```

Cause future `read` commands to read from the indicated file. If that file is not open, an error message will result. Use `read from \filename` to shuffle reading among several open files.

Gri can look up filenames for `read from` in terms of their **full** pathnames or their **local** pathnames. Thus, a local file called `a.dat` in the directory `/home/gri` can be referred to by `read from a.dat` or by `read from /home/gri/a.dat`, which comes in handy if you need to work with two files of the same name, in other directories. However, since Gri has the ability to search for files in a "path" (Section 9.3.41.39 [Set Path To], page 129), you may

not have specified an exact path name; this is why the `open` command provides a return value which names the full pathname (Section 9.3.28.1 [Opening Simple Files], page 102).

9.3.33.9 `read` synonym/variable

```
'read [* [*...]] \synonym|.variable. [\synonym|.variable. [...]]}'
```

Read one or more items from the next line of the input file. These items may be synonyms or variables. The token `*` indicates that the word in the datafile should be skipped. As usual, the datafile may be embedded in the commandfile, providing the last data line is blank.

Normally one would use synonyms for words, and variables for numbers. The items are separated by one or more "whitespace" characters (e.g. space or TAB). Thus, if a file contained the line

```
Temperature 10.3
```

then the line

```
read \var_name .var_value.
```

would have the same result as

```
\var_name = "Temperature"
.var_value. = 10.3
```

This command ignores any trailing items on the line. That is, the next `read` command will start on the next line of the file. In a sense then, you get just one shot at analysing the input line in your datafile. If you need flexibility, you may wish to read the **whole** contents of the line into a synonym, which may be done using the `read line` command instead, to read it in as a string. (Section 9.3.33.10 [Read Line], page 113).

If the input file is in the netCDF format, the indicated item will be read. For example, `read \time:_MissingValue` reads the missing value for the variable called `time`. This conveniently allows your data file to dictate axes names, units, missing values, etc. Example:

```
# Plot profile of TU81N (age-corrected tritium)
open profile.nc netCDF
read columns x="TU81N" y="z"
read \{z:_FillValue} # assume same for all
read \{z:long_name}
read \{z:units}
read \{TU81N:long_name}
read \{TU81N:units}
close
set missing value \{z:_FillValue}
set x name "\{TU81N:long_name} (\{TU81N:units)}"
set y name "\{z:long_name} (\{z:units)}"
set y axis decreasing
draw curve
```

For more information on netCDF format, see `http://www.unidata.ucar.edu/packages/netcdf/index.ht`

9.3.33.10 `read line`

```
'read line \synonym'
```

Read the next line of the datafile (or commandfile), trim off a trailing comment if there is one, and then store the next line of datafile into the named synonym.

9.3.34 regress

```
'regress {y vs x [linear]}|{x vs y [linear]}'
```

Perform linear regression of `y` as a function of `x` or `x` as a function of `y`.

- `regress y vs x` Linear regression of y vs x. Several quantities are reported and also saved into builtin variables. The intercept is defined as `..coeff0..`, its 95 percent confidence limit is defined as `..coeff0_sig..` Thus the confidence range is `..coeff0..-..coeff0_sig..` to `..coeff0..+..coeff0_sig..` Similarly the slope and confidence limit are stored in `..coeff1..` and `..coeff1_sig..`

  The squared correlation coefficient is stored in `..R2..`

  Historical note prior to version 2.1.15, a different meaning was attached to `..coeff0_sig..` and `..coeff1_sig..`; they used be defined as standard errors, without having been multiplied by the appropriate student-t coefficient.
- `regress x vs y` Linear regression of x vs y; for notation see above.
- `regress y vs x linear` Linear regression of y vs x; for notation see above.
- `regress x vs y linear` Linear regression of x vs y; for notation see above.

9.3.35 reorder

```
'reorder columns randomly \
  |{ascending in x|y|z} \
  |{descending in x|y|z}'
```

Reorder the columns in various ways.

In the `randomly` style, the column data are shuffled randomly by index, retaining the correspondance between a given x and y. This is useful with `draw symbol` using colored dots – it prevents the overpainting of one dot on another from biasing the color field to values that happened to occur near the end of the column data. If you prefer the overpainting to be done in random order, use this command to reorder the columns randomly. The random number is selected using the system `rand` call, with the seed being provided by the PID (process ID) of the job.

The `ascending` and `descending` styles do what you'd expect.

9.3.36 rescale

```
'rescale'
```

Re-determine the scales for the x and y axes. Typically used after a column math operation, when you want the new data to be auto-scaled. (Note: normally, if the axes have already been drawn, Gri won't rescale automatically just because you modify the column data. This is designed so that you can add offsets to curves, etc, while staying in saved axes. But if the axes have not been drawn yet when you modify the column data, then Gri will automatically rescale the axes it is planning to draw.)

9.3.37 resize

```
'resize {x for maps}|{y for maps}'
```

Resize the axes frame region in such a way that geographical objects appear in correct proportions. This assumes that y is degrees latitude and x is degrees longitude.

- `resize x for maps` Resize the plot width for maps, assuming that x represents longitude and y represents latitude. Before using this, you must have defined scales for both x and y, and a size for y (ie, you must have done `set x axis ...`, `set y axis ...` and `set y size`); this command sets the x size, thus eliminating `set x size`. The result is that, at the central latitude (y), a centimetre on the page will correspond to an equal distance on the earth, in both the north-south and east-west directions.
- `resize y for maps` Resize the plot height for maps, assuming that x represents longitude and y represents latitude. Before using this, you must have defined scales for both x and y, and a size for x (ie, you must have done `set x axis ...`, `set y axis ...` and `set x size`); this command sets the y size, thus eliminating `set y size`. The result is that, at the central latitude (y), a centimetre on the page will correspond to an equal distance on the earth, in both the north-south and east-west directions.

  See also `resize x for maps`.

9.3.38 return

`'return'`

Return early from a user-defined function or an `insert` file. Or, in the main gri program, do the same thing as `quit`.

9.3.39 rewind

`'rewind \filename'`

Rewind a data-file to the beginning. If no filename is given, this is done for the currently active file; otherwise the named file is rewound.

9.3.40 rpnfunction

`'rpnfunction name replacement-words'`

Create a new keyword for use in rpn expressions. Inside any RPN expression which follows this line, the word `name` will be substituted with the indicated replacement words.

For example, the following shows the definition and use of a function which computes the sine of twice an angle, by multiplying whatever is on the stack by `2`, and then taking the sine of the result.

```
rpnfunction sin2 2 * sin
show "expect the number 1 to follow: " {rpn 45 sin2}
```

The replacement words will have any synonyms in them translated first, unless they start with an underscore followed by a double backslash. Similarly, variables are substituted unless they start with an underscore. These exceptions are to allow the use of the `defined` operator (Section 10.9 [rpn Mathematics], page 161).

Note: The mathematical constants `e` and `pi` are stored using `rpnfunctions`. Also, two `rpnfunctions` are provided for finding the slope and intercept of a line joining two points, e.g.

```
# Expect answers 1 and 10 ...
show "slope=" {rpn 0 1 1 11 linear_slope}
show "inter=" {rpn 0 1 1 11 linear_intercept}
```

These are useful in `set grid missing above .intercept. .slope.` and `set z missing above .intercept. .slope.`.

9.3.41 The `set` commands

Set various flags and parameters which Gri will use in later commands.

9.3.41.1 `set axes style`

```
set axes style .style.     \
  | {offset [.dist_cm.]} \
  | rectangular|none|default
```

Tell Gri how you want the axes to look, when they are drawn later.

- `set axes style 0` Set axes to be rectangular, with an x-y axis frame labelled at the left and bottom and tic marks on all axes.
- `set axes style 1` As style `0`, but only put tics on the lower and left axes.
- `set axes style 2` As style `0` but without labels or tics on any axis, i.e. just an axis frame.
- `set axes style offset [.dist_cm.]` Set axes so that the actual x and y axes will be drawn with a space separating them from the data area. The space, if not set by the `.distance_cm.` option, will be equal to the current tic size (see `set tic size`). This command can be used together with any other `set axes style` command. It applies to both the `draw axes` command and with any `draw x|y axis` command in which the axis location is not explicitly given.
- `set axes style rectangular` Set axes to be rectangular, with an x-y axes frame labelled at the left and bottom.
- `set axes style none` Tell gri not to bother drawing axes before drawing curves, etc.
- `set axes style default` Same as `set axes style 0`, and with `offset` turned off.

9.3.41.2 `set arrow size`

```
'set arrow size .size.           \
  | {as .num. percent of length} \
  | default'
```

Set the arrowsize (which is stored in the builtin variable `..arrowsize..`).

- `set arrow size .size.` Set the arrow size (ie, half-width of the arrowhead) to `.size.` centimetres.
- `set arrow size as .num. percent of length` Set the arrow size to be the indicated percentage of arrow length, as in `"HWP"` in the singles ads. (As a flag to this, `..arrowsize..` is set to the negative of the fractional size measured in percent.)
- `set arrow size default` Set the arrow size to the default of 0.2 cm.

9.3.41.3 `set arrow type`

```
set arrow type .which.
```

Set type of arrow, according to the value of `.which.`, rounded to the nearest integer. A rounded `.which.` value of 0 yields the default arrows, drawn with three line strokes. Value 1 yields outlined arrows, sometimes used on definition sketches. Value 2 yields filled, swept-back arrow heads.

This command uses the "line join" parameters that are presently active (Section 9.3.41.32 [Set Line Join], page 125). So, by default, the arrow ends are rounded (because the default line-join parameter is 1). To get pointy ends, first set the line join parameter to 0.

9.3.41.4 `set beep`

```
'set beep on|off'
```

The command `set beep on` makes gri beep on errors and `query`. `set beep off` (the default) turns this beeping off.

9.3.41.5 `set bounding box`

```
'set bounding box .xleft. .ybottom. .xright. .ytop. [pt|cm]'
```

Set the PostScript bounding box for the graph to the indicated values. The bounding box is used by some programs to determine the region of the page on which marks have been made. For example, LaTeX uses the bounding box to decide how to position figures in documents.

Normally, the bounding box is computed automatically unless the `-no_bounding_box` commandline option has been specified; (Chapter 3 [Invoking Gri], page 7). But if `set bounding box` is done, the automatically computed value is ignored and the given box is used instead. Use this if Gri makes mistakes in its automatic selection of bounding box.

The coordinates of the bounding box may be specified in (1) user coordinates, as defined **at the moment** the command is executed, or (2) in points on the page, measured from an origin at the lower-left (72 point per inch), or (3) in centimeters on the page. Which coordinate system is used depends on the last keyword – use `pt` for points, `cm` for centimeters, and nothing at all for user-units.

The most common use is in points, since that is how many other application packages, e.g. LaTeX and dvips, specify the bounding box.

If the box is specified in the user units, the user units in effect **at the moment** of executing the `set bounding box` command are used. This must be born in mind if the coordinate system is changing during the execution of the program, e.g. if margins are changing or the x and y axes are changing. For this reason it often makes sense to put this command at the end of the commandfile.

9.3.41.6 `set clip`

```
'set clip [postscript] \
  {on [.xleft. .xright. .ybottom. .ytop.]} \
  {to curve} \
  | off'
```

Control clipping of following drawing commands. Note that the commands have two styles, one of which includes the keyword `postscript`. PostScript clipping does a cleaner job, but it results in larger file sizes. *Important* if you are using `postscript` clipping, then you be *sure* to turn it off using `set clip postscript off`, instead of `set clip off`; otherwise Gri will get mixed up.

- `set clip on` Don't plot data outside axes.
- `set clip postscript on` As above, but Postscript clipping.

- `set clip on .xleft. .xright. .ybottom. .ytop.` Don't plot data outside indicated box.
- `set clip postscript on .xleft. .xright. .ybottom. .ytop.` As above, but Postscript clipping.
- `set clip off` Plot all data, whether inside the axes or not.
- `set clip postscript off` As above, but Postscript clipping.
- `set clip to curve` set clip to the curve, as would be drawn by a `draw curve filled` command, i.e. to the polygon constructed by running along the xy points, in order, followed by a final segment from the last point back to the first point. This is a "postscript" clip, as explained in the next item.
- `set clip postscript to curve` As above, but Postscript clipping.

```
draw axes
set clip postscript on 10 20 0 1
draw curve
set clip postscript off
```

- `set clip postscript off` Turn PostScript clipping off. **See also** `set input data window`.

9.3.41.7 set color

```
'set color \name                | \
    {rgb .red. .green. .blue.} | \
{hsb .hue. .saturation. .brightness.}'
```

Set the color of the "pen" used for drawing lines and text. Normally lines and text are drawn in the same color, but the text color can be specified independently if desired (Section 9.3.41.17 [Set Font Color], page 120). This might be useful to get contour lines of one color and labels of another. The spelling `colour` is also accepted.

In the `set color \name` style, set the drawing color to the indicated name, either from the builtin list (`white`, `LightGray`, `darkslategray`, `black`, `red`, `brown`, `tan`, `orange`, `yellow`, `green`, `ForestGreen`, `cyan`, `blue`, `skyblue`, `magenta`), or from a list created by `read colornames`. In the latter case, if the colorname has more than one word in it, use quotes, e.g. `set color "ghost white"`.

In the `set color rgb ...` style, set the individual color components as indicated. The numbers `.red.`, `.green.` and `.blue.` range from 0 (for no contribution of that color component to the final color) to 1 (for maximal contribution). Values less than 0 are clipped to 0; values greater than 1 are clipped to 1. EXAMPLES:

```
set color rgb 0   0   0 # black
set color rgb 1   1   1 # white
set color rgb 1   0   0 # bright red
set color rgb 0.5 0   0 # dark red (only 50 percent)
set color rgb 0   1   0 # pure green
set color rgb 1   1   0 # yellow: red + green
```

In the `set color hsb ...` style, set the individual color components as indicated. The numbers `.hue.`, `.saturation.` and `.brightness.` range from 0 to 1. The color, represented by .hue., ranges from 0 for pure red, through 1/3 for pure green, and 2/3 for pure blue, and back to 1 again for pure red. The purity of the color, represented by .saturation., ranges from 0 (none of the hue is visible) to 1 (the maximal amount is present). The brightness of

the color, represented by `.brightness.`, ranges from 0 (black) to 1 (maximal brigntness). Values less than 0 are clipped to 0; values greater than 1 are clipped to 1. EXAMPLES:

```
set color hsb 0    1   1 # pure, bright red
set color hsb 0    1 0.5 # half black, half red
set color hsb .333 1   1 # pure, bright green
```

9.3.41.8 `set colorname`

```
'set colorname \name {rgb .red. .green. .blue.} \
                   | {hsb .hue. .saturation. .brightness.}'
```

Create a colorname with the indicated color. The color components range from 0 to 1, and will be clipped to these values if they are outside this range. EXAMPLE (borrowing a color from '`/usr/lib/X11/rgb.txt`'):

```
set colorname peachpuff rgb 1 {rpn 218 255 / }  {rpn 185 255 / }
draw box filled 2 2 3 3 cm
```

This command is akin to `read colornames` (Section 9.3.33.1 [Read Colornames], page 106), except that the latter uses an X11 database, so the color constituents range from 0 to 255, whereas for `set colorname` they range from 0 to 1.

9.3.41.9 `set contour format`

```
'set contour format \style|default'
```

Normally, Gri draws the numeric labels of contours using a format code called `%g` in the "C" language. You may specify any other "long" format using this command. For example, `set contour format %.1f` tells Gri to use one decimal place in the numbers, and also to prefer the "float" notation to the exponential notation. `set contour format default` resets to the default `%f` format. You may use quotes around the format if you need to, to make the item be a single word (e.g. `set contour format "%.1f m/s"`).

9.3.41.10 `set contour label for`

```
'set contour label for lines exceeding .x. cm'
```

Make it so contour lines shorter than `.x.` centimeters will not be labelled.

9.3.41.11 `set contour label position`

```
'set contour label position \
  {.start_cm. .between_cm.} \
  | centered \
  | default'
```

By default, contour labels are drawn at the location 1 cm into the contour curve, measured from the startpoint of the contour (e.g., for contours crossing the axes frames, the label will be 1 cm from the frame), and then at a uniform distance along the contour. By default, this uniform distance is the average dimension of the plotting area inside the axes. If `.start_cm.` and `.between_cm.` are specified, the first label is drawn at a distance `.start_cm.` from the start of the contour, and thereafter at a separation of `.between_cm.`.

If the `centered` option is used, then the contour labels are centered along the length of the line.

9.3.41.12 set contour labels

```
set contour labels rotated \
  | horizontal \
  | whiteunder \
  | nowhiteunder
```

The first two options control whether contour labels are rotated to line up with the contour lines, or whether they are horizontal (the default).

The second two options control whether the region under contour labels is whited out before drawing the label. The default is `whiteunder`, which has the visual effect of the label having been drawn on a piece of paper and then pasted on. This can look jarring when the material under the contour is an image. When `nowhiteunder` is specified, the contour line is broken to make space for the text, but no whiting out is done.

9.3.41.13 set dash

```
'set dash [.n.|{.dash_cm. .blank_cm. ...}|off]'
```

Control dash-style for following `draw curve` and `draw line` commands.

- `set dash` Set to dashed line (0.4cm dashes, 0.1cm blanks).
- `set dash .n.` Set to indicated pre-defined dashed line, according to table:

```
.n. dash/cm blank/cm
 0        -        -   ... (Solid line)
 1      0.2      0.1
 2      0.4      0.1
 3      0.6      0.1
 4      0.8      0.1
 5      1.0      0.1
10      w        w
11      w        2w
12      w        3w
13      w        4w
14      w        5w
15      w        6w
```

  Where `w` is written, it indicates the current linewidth. Thus, types 10 through 15 give square-dotted lines.

- `set dash .dash_cm. .blank_cm. .dash_cm. .blank_cm. ...` Set to indicated dashed line. The series of lengths `.dash_cm.` and `.blank_cm.` give the lengths of dash and blank portions (measured in centimeters). Any number of dash/blank lengths may be given. For example, `set dash 0.5 0.1 0.1 0.1` looks good.
- `set dash off` Turn dashing off, setting to a solid line.

9.3.41.14 set environment

```
'set environment'
```

Set environment (graylevel, axis length, etc) so that following plotting commands will make use of anything set by either a `set` command or by direct manipulation of builtin variables like `..xsize..`, etc. NOTE: this should **only** be done by developers.

9.3.41.15 set error

```
'set error action to core dump'
```

Make Gri dump core when any error is found, to facilitate debugging.

9.3.41.16 set flag

```
'set flag \name [off]'
```

Set the indicated flag to YES. The name of the flag is contained in a single word, e.g. `set flag dan_28sep_test`. The action of the flags may change with time and is undocumented. This command is provided to enable selected users (e.g., the developer himself) to use test features of Gri before they are frozen into a fixed syntax and action. The keyword `off` turns the indicated flag off. NOTE: this should **only** be done by developers.

```
FLAG          DATE      ACTION

jinyu1        29sep94   'convert columns to grid'
                         outputs (x,y,z,z_predicted)

emulate_gre    9jun97   'E' format on axes yields
                        scientific notation

kelley1       17jun97   for kelley only - quit contour
                        trace if hit nonlegit

kelley2       17jun97   for kelley only: print
                        info while contour tracing
```

9.3.41.17 set font color

```
set font color \name            | \
    {rgb .red. .green. .blue.} | \
{hsb .hue. .saturation. .brightness.}
```

The syntax is the same as `set color`, except that this applies to text only. By default, text is drawn in the same color as lines, so text color is changed as line color is changed (e.g. by using the `set color` or `set graylevel` commands)). However, once `set font color` is used in a Gri program, the font thereafter maintains a separate color from the lines.

9.3.41.18 set font encoding

```
set font encoding PostscriptStandard | isolatin1
```

Permits one to control the so-called "font encoding" used in text. The default font encoding is ISO-Latin-1, which is best for English and other European languages. To learn how to enter accents in a text editor, and for a brief overview of font encodings, (Section 10.10.3 [Non-English Text], page 172).

If the so-called "Postscript Standard" font encoding is required, this command permits changing the encoding.

Note: few users will ever need this command. If you don't even know what "font encoding" is about, ignore this command!

9.3.41.19 `set font size`

```
'set font size {.size. [cm]}|default'
```

Set the size of the font for drawing of text.

- `set font size .size.` Set font size to `.size.` points. (A point is 1/72 of an inch, or 1/28 of a centimetre.)
- `set font size .size. cm` Set font size to `.size.` centimetres.
- `set font size default` Set font size to default = 12 pts.

If your figure is to be reproduced by a journal, you should check with them about the range of font size they need. This will, of course, depend on whether your figure is reduced or enlarged during reproduction. For example, American Geophysical Union (publishers of J. Geophysical Res.) recommends that one use fonts no smaller than 8 points. They also recommend that the range in fontsize in a given figure not exceed 2.

9.3.41.20 `set font to`

```
'set font to \fontname'
```

Set font to named style. Note that the backslash is **not** to be written, but here merely means that this word has several alternatives. For example, one might say `set font to Courier`. The allowed fontnames are: `Courier`, a typewriter font (and the only monospaced font in Gri); `Helvetica` (the default), a sans-serif font commonly used in drafting scientific graphs; `HelveticaBold`, a bold version of Helvetica; `Times` (also called `TimesRoman`), a font used in most newspapers; `TimesBold`, a bold version of Times; `Palatino` (also called `PalatinoRoman`), similar to Times; `Symbol`, included for completeness, is a mathematical font in which `"a"` becomes α of the math mode, etc. For reference on these fonts see any book on PostScript. The default font is `Helvetica`.

9.3.41.21 `set graylevel`

```
'set graylevel .brightness.|white|black'
```

Set graylevel for lines to indicated numerical value between 0=black and 1=white, or to the named color.

Note: if your diagram is to be reproduced by a journal, it is unlikely that the reproduction will be able to distinguish between any two graylevels which differ by less than 0.2. Also, graylevels less than 0.2 may appear as pure black, while those of 0.8 or more may appear as pure white. These guidelines are as specified by American Geophysical Union (publishers of J. Geophysical Res.), as of 1998.

9.3.41.22 `set grid missing`

General format is

```
'set grid missing \
  {above|below .intercept. .slope} \
  | {inside curve}'
```

The style

```
'set grid missing above|below .intercept. .slope'
```

sets grid to missing value for all points above/below the line defined by y = .intercept. + .slope.x

The style

```
'set grid missing inside curve'
```

sets the grid to the missing value throughout an area described by the curve last read in with `read columns`. This is useful for e.g. excluding land areas while contouring ocean properties. The curve may contain several "islands," each tracing (clockwise) a region inside of which the grid is to considered missing. If the first point in an island doesn't match the last, then an imaginary line is assumed which connects them. Multiple islands may be separated by missing-value codes.

See also `Set Z Missing`.

9.3.41.23 `set ignore initial newline`

```
'set ignore initial newline [off]'
```

Make Gri ignore a newline if it occurs as the first character of the next data file. This is used for files made by FORTRAN programs on VAX/VMS computers.

9.3.41.24 `set ignore error eof`

```
'set ignore error eof'
```

Stop Gri from considering that to encounter an end of file in future `read` commands consitutes an error; Gri will simply warn about future EOFs.

9.3.41.25 `set image colorscale`

```
set image colorscale hsb .h. .s. .b. .im_value. \
    hsb .h. .s. .b. .im_value. [increment .im_value.]
set image colorscale rgb .r. .g. .b. .im_value. \
    rgb .r. .g. .b. .im_value. [increment .im_value.]
set image colorscale \
  \name .im_value. \
  \name .im_value. \
  [increment .im_value.]
```

Set colorscale mapping for image, using HSB (hue, saturation, brightness) specification, RGB (red, green, blue) color specification, or named colors. The image range must previously have been set by `set image range`, so that the `.im_value.` values will have meaning. Two pairs of (color, image-value) are given, and possibly an increment. Normally the colors are smoothly blended between the endpoints, but if an increment is supplied, the colors are quantized. The HSB method allows creation of spectral palettes, while the other two methods create simple blending between the two endpoints.

EG: To get a spectrum ranging between pure red (H=0) for image value of -10.0, and pure blue (H=2/3) for image value of 10.0, do this:

```
set image colorscale hsb 0 1 1 -10 hsb .666 1 1 10
```

EG: To get a scale running from pure red (at image-value 10.0) into pure blue (at image-value 25.1), but with the colors blending intuitively in between (i.e., blending as paint might), use `rgb` color specification, as follows:

```
set image colorscale rgb 1 0 0 10 rgb  0 0 1 25.1
```

EG: To get a quantized blend between the X11 colors `skyblue` at image value of 0 and `tan` at image value of 20, and with steps at image values incrementing by 5, do this:

```
set image colorscale skyblue 0 tan 20 increment 5
```

Note that the traversal is through RGB space, so it is intuitive, not spectral. See `set color` for a list of X11 colors known to Gri.

See also `read image colorscale` (Section 9.3.33.4 [Read Image Colorscale], page 109).

9.3.41.26 `set image grayscale`

```
'set image grayscale using histogram \
  [black .bl. white .wh.]'

'set image grayscale \
  [black .bl. white .wh. [increment .inc.]]'
```

Set up a grayscale mapping for images. The image range must have previously have been set by `set image range`.

```
'set image grayscale using histogram [black .bl. white .wh.]'
```

Create a grayscale mapping using linearized cumulative histogram enhancement. The image range must have previously have been set by `set image range`.

This creates maximal contrast in each range of graylevels, and is useful for tracing subtle features between different images (for example, it makes it easier to trace fronts between successive satellite images). The entire histogram is expanded, from the smallest value in the image to the largest.

With no options specified, the histogram is done from 0 in the image to 255 in the image. If the black/white options are specified, the histogram is done between these values.

```
'set image grayscale \
  [black .bl. white .wh. [increment .inc.]]'
```

With no optional parameters, create a grayscale mapping for the current image, scaling it from black for the mininum value in the image to white for the maximum value. The image range must have previously have been set by `set image range`.

The optional parameters `.wh.` and `.bl.` specify the values to be drawn in white and black in the image, with smooth linear blending in between.

Normally the blending from white to black is smooth (linear), but if the additional optional parameter `.inc.` is specified, the blending is quantized, jumping to darker values at (`.wh. + .inc.`), (`.wh. + 2* .inc.`), etc. (The sign of `.inc.` will be altered, if necessary, to ensure that (`.wh. + .inc.`) is between `.wh.` and `.inc.`.) The colour switches to pure white at the value `.wh.`, and remains pure white everywhere on the "white" side of this value. Similarly, the transition to pure black occurs at the value `.bl.`. In other words, neither pure white nor pure black is present inside the interval from `.wh.` to `.bl.`. Therefore, when using the `draw image palette` command, you might want to extend the range by one increment so as to get an example of both pure white and pure black.

```
.w. = 0
.b. = 1
.i. = 0.2
set image grayscale white .w. black .b. increment .i.
draw image palette left \
  {rpn .w. .i. -} \
  right {rpn .b. .i. +} \
```

```
increment .i.
```

9.3.41.27 set image missing value color

```
set image missing value color to white|black|\
    {graylevel .brightness.}|{rgb .r. .g. .b.}
```

Set the color of "missing" pixels (white by default). The image range must have previously have been set by `set image range`. Pixels with missing values can result from creating images from grids which have missing values; see the `convert grid to image` command. The `.brightness.` parameter in the `graylevel` style ranges from 0 for black to 1 for white. The `rgb` parameters allow setting to full color.

9.3.41.28 set image range

```
set image range .0value. .255value.
```

Specify maximum possible range of values that images can hold, in user units. Gri needs to know this because it stores images in a limited format capable of holding only 256 distinct values. Unlike some other programs, Gri encourages (forces) the user to specify things in terms of user-units, not image-units. This has the advantage of working regardless of the number of bits per pixel. Thus, for example, `set image grayscale`, `set image colorscale`, `draw image grayscale`, etc, all use **user** units.

When an image is created by `convert grid to image`, values outside the range spanned by `.0value.` and `.255value.` are clipped. (There is no need, however, for `.0value.` to be less than `.255value.`.) This clipping discards information, so make sure the range you give is larger than the range of data in the grid.

EXAMPLE: consider a satellite image in which an internal value of 0 is meant to correspond to 0 degrees Celsius, and an internal value of 255 corresponds to 25.5 degrees. (This is a common scale.) Then the Gri command `set image range 0 25.5` would establish the required range. If this range were combined with a linear grayscale mapping (see `set image grayscale`), the resultant granularity in the internal representation of the user values would be (25.5-0)/255 or 0.1 degrees Celsius; temperature variations from pixel to pixel which were less than 0.1 degrees would be lost.

All other image commands **require** that the range has been set. Thus, all these commands fail unless `set image range` has been done: `draw image`, `draw image palette`, `read image`, `convert grid to image`, `set image grayscale`, and `set image colorscale`.

NOTE: If a range already exists when `set image range` is used, then the settings are altered. Thoughtless usage can therefore lead to confusing results. (The situation is like setting an axis scale, plotting a curve with no axes, then altering the scale and plotting the new axes. It is legal but not necessarily smart.)

9.3.41.29 set input data window

```
'set input data window x|y {.min. .max.}|off'
```

Create a data window for following `read` statements.

- `set input data window x .min. .max.` For future reading commands, ignore all data with x less than `.min.` or x greater than `.max.` The data not in the interval will not be read in at all. This will hold until `set data window x off` is done, in which case all data will be read in.

- `set input data window x off` Return to normal conditions, in which all data are read in.
- `set input data window y .min. .max.` Analgous to command for x.
- `set input data window y off` Analagous to command for x.

EXAMPLE: To set the input data window as the current x axis plus a border of 5 centimetres to left and right, do the following:

```
set input data window x \
  {rpn ..xleft..  xusertocm 5 - xcmtouser} \
  {rpn ..xright.. xusertocm 5 + xcmtouser}
```

See also `set clip`

9.3.41.30 `set input data separator`

`'set input data separator TAB|default'`

Set the separator between data items. The `default` method is to assume that data items are separated by one or more spaces or tabs, and also to ignore any spaces or tabs at the start of a data line.

In the `TAB` method the data are assumed to be separated by a SINGLE tab character. (Multiple tabs will result in null values being assigned to items – almost certainly not what you want!) Also, initial spaces and tabs on lines are NOT skipped.

Use the `TAB` method only after thinking carefully about the above, since the assignment of null values is problematic.

9.3.41.31 `set line cap`

`'set line cap .type.'`

Set the type of ends (caps) of lines. Use `.type.` of value 0 for square ends, cut off precisely at the end of line, or 1 for round ends which overhang half the line width, or 2 for square ends which overhang half the line width. The selected style is used for the ends of line segments, as well as at corners. In PostScript parlance, therefore, this command sets both the linecap and the linejoin parameters.

This command only applies to lines drawn with `draw curve`, `draw line` and `draw polygon`. Axes are always drawn with a line cap of 0.

9.3.41.32 `set line join`

`'set line join .type.'`

Set the type of intersection of lines. Use `.type.` of value 0 for mitered joins (pointy ends that may extend beyond the data points), a value of 1 for rounded ends (the default), or a value of 2 for bevelled (squared-off) ends. See the `setlinejoin` command in any text on the PostScript language for more information.

This command only applies to lines drawn with `draw curve`, `draw line` and `draw polygon`. Axes are always drawn with a line join of 0.

9.3.41.33 `set line width`

`'set line width \`

```
[axis|symbol|all] \
.width_pt. \
| {rapidograph \name} \
| default'
```

Set the width of lines used to draw curves (default), axes, symbols, or all of the above. The width may be set to a value specified in points (conversion: 72 pt = 1 inch), to a named rapidograph width, or to the default value. The initial default values are: 0.709pt (or rapidograph 3x0) for curves; 0.369pt (or rapidograph 6x0) for axes; 0.369pt (or rapidograph 6x0) for symbols. (To learn more about standard pen widths, see the ISO 9175-1 documents.)

If your figure is to be reproduced by a journal, you should check with them about the range of line thicknesses they recommend. This will, of course, depend on whether your figure is reduced or enlarged during reproduction. For example, American Geophysical Union (publishers of J. Geophysical Res.) recommends that one use line thicknesses no less than 0.5 points and no more than 4 points.

The rapidograph settings match the standard set of widths used in technical fountain pens. The table below gives width names along with the width in points and centimetres, as given in the specifications supplied with Rapidograph technical fountain pens. Names marked by the symbol * are in sequence increasing by the factor root(2). Texts on technical drawing often suggest using linewidths in the ratio of 2 or root(2). On many printers, the variation in width from root(2) increase is too subtle to see, so the factor-of-2 rule may be preferable. To get sizes in a sequence doubling in width, pick from the list (`6x0`, `3x0`, `1`, `3.5` `7`). To get a sequence increasing in width by root(2), pick from the list (`6x0`, `4x0`, `3x0`, `0`, `1`, `2.5`, `3.5`, `6`, `7`). The eye can distinguish curves with linewidths differing by a factor of root(2) if the image is of high quality, but a factor of 2 is usually better. Similarly, for overhead projections and projected slides, one would do well to use linewidths differing by a factor of 4.

This is the list of `rapidograph` linewidths:

	Name	pt	cm
*	6x0	0.369	0.013
*	4x0	0.510	0.018
*	3x0	0.709	0.025
	00	0.850	0.03
*	0	0.992	0.035
*	1	1.417	0.05
	2	1.701	0.06
*	2.5	1.984	0.07
	3	2.268	0.08
*	3.5	2.835	0.1
	4	3.402	0.12
*	6	3.969	0.14
*	7	5.669	0.2

9.3.41.34 `set missing value`

```
'set missing value .value.|none'
```

If a numerical value is given, set the missing-value code to that value, and also store this value in the builtin variable `..missingvalue..` and the builtin synonym `\.missingvalue..`

also. After this command, Gri will ignore any data that are within 0.1 percent of this value. (This feature is commonly used in geophysical data.)

If `none` is given, turn off this feature.

The default is that the feature is turned off.

9.3.41.35 `set page size`

```
'set page size letter|legal|folio|tabloid|A0|A1|A2|A3|A4|A5'
```

Set the page dimension, as indicated below.

letter	American "letter" size: 8.5 x 11 inches
legal	American "legal" size: 8.5 x 14 inches
folio	American "folio" size: 8.5 x 13 inches
tabloid	American "tabloid" size: 11 x 17 inches
A5	Metric size: 14.8 x 21.0 cm
A4	Metric size: 21.0 x 29.7 cm
A3	Metric size: 29.7 x 42.0 cm
A2	Metric size: 42.0 x 59.4 cm
A1	Metric size: 59.4 x 84.1 cm
A0	Metric size: 84.1 x 118.9 cm

The effect is to possibly alter the PostScript "bounding box." If all the drawn material fits within the indicated page, then the bounding box is not altered. (In other words, Gri will still keep the bounding box tight on the drawn items.)

However, if any drawn item extends beyond the indicated size, it will be clipped to the boundary.

9.3.41.36 `set page`

```
'set page portrait \
  | landscape \
  | {factor .mag.} \
  | {translate .xcm. .ycm.}'
```

Control orientation or scaling of what is drawn on the paper.

- `set page portrait` Print graph normally (default).
- `set page landscape` Print graph sideways.
- `set page factor .mag.` Scale everything to be drawn on the paper by the indicated magnification factor. This **must** be called before any drawing commands.
- `set page translate .xcm. .ycm.` Translate everything to be drawn on the paper by the indicated x/y distances. This **must** be called before any drawing commands.

Note: The order of the factor/translate commands matters, so you may need to experiment. For example,

```
set page translate 2 1
set page factor 0.5
```

moves anything that would have been drawn at the lower-left corner of the paper onto the point 2cm from the left side and 1cm from the bottom side of the paper, and then applies the multiplication factor. Reversing the order gives quite different results. PostScript gurus should note that the following two commands are inserted into the PostScript file:

```
56.900000 28.450000 translate
0.500000 0.500000 scale
```

9.3.41.37 set panel

```
set panel .row. .col.
```

Establish the geometry for the panel in the indicated row and column; that is, select which defined panel to draw into. The bottom row has `.row.` = 1, and the leftmost column has `.col.` = 1. This must be used only after defining the panel layout using `Panels .row. .col. .dx_cm. .dy_cm.`.

9.3.41.38 set panels

```
set panels .rows. .cols. [.dx_cm. .dy_cm.]
```

Set up for multipanel plots, with spacing `.dx_cm.` between the columns and `.dy_cm.` between the rows. If the spacings are not supplied, 2cm is used. The panels fill the rectangle which would otherwise contain the single axis frame, as set by `set x size` and `set x margin`, etc.

The global variables `.panel_dx.`, `.panel_dy.`, `.panel_xmargin.`, `.panel_ymargin.`, `.panel_xsize.`, and `.panel_ysize` are created, to be used by later calls to `set panel`.

EXAMPLE

```
# Draw 2 panels across, 3 up the page.

# The Panel interiors will be in region cornered
# by (2,2), (12,22) cm
set x margin 2
set y margin 2
set x size 10
set y size 20
set panels 2 3

# Create dummy scale
set x axis 0 1
set y axis 0 1

# Draw blank axes
et panel 1 1
raw axes
set panel 1 2
draw axes
set panel 1 3
draw axes
set panel 2 1
draw axes
set panel 2 2
```

```
draw axes
set panel 2 3
draw axes
```

See also `set panel .row. .col.`

9.3.41.39 `set path`

```
set path to "\path"|default for data|commands
```

Set the directory path that `open` will search for data files, or that `insert` will search for command files. This search will *not* be done if the filename starts with a `/`, `~`, or `.` character.

The path is formatted in a colon-separated manner, following the normal Unix convention, and searching is from left to right. For example, the path `".:/usr/lib/gri"` tells Gri to search for the file first in the local directory (named '`.`'), and if it is not found there, to look next in the directory named '`/usr/lib/gri`'.

The indicated path is stored in either `\.path_data.` or `\.path_commands.`, as appropriate. At startup time, each of these paths is set to '`"."`', the current directory, and this value is reset if the `default` keyword is provided to this command.

If you need to know where the file was eventually found, save the `\.return_value.` just after the `open` command was executed. For example, the following defines the synonym `\uk`, which is the full pathname of the file containing some sort of data about Great Britain.

```
set path to "/atlases/world:/atlases/northern_hemisphere" for data
open britain.dat        # we don't know where file is ...
\gb = "\.return_value." # ... until now!

# Can later do command such as
#    read from \gb
# or
#    rewind \gb
# to work with this particular file, even if
# there is another file open that also is
# named "britain.dat".
```

9.3.41.40 `set postscript filename`

```
'set postscript filename "\name"'
```

Set name of PostScript file, over-riding the present name.

9.3.41.41 `set symbol size`

```
'set symbol size .diameter_cm.|default'
```

Control the diameter of symbols drawn by `draw symbol` command.

- `set symbol size .diameter_cm.` Make symbol size be `.diameter_cm.` centimeters in diameter.
- `set symbol size default` Set to default diameter of 0.1 cm.

9.3.41.42 `set tic size`

```
'set tic size .size.|default'
```

Control size of tics on axes.

- `set tic size .size.` Set tic size to `.size.` centimetres.
- `set tic size default` Set tic size to default of 0.2 cm.
- `set tics in|out` Make axis tics point inward or outward. The default is outward.

9.3.41.43 `set trace`

```
'set trace [on|off]'
```

Control printing of command lines as they are processed.

- `set trace` Make Gri print command lines as they are processed.
- `set trace on` Same as `set trace`.
- `set trace off` Prevent printing command lines (default).

9.3.41.44 `set transparency`

```
'set transparency .transparency.'
```

Set the transparency of drawn items, 0 for opaque and 1 for invisibly faint. *This command is provisional, as of summer 2004, and this part of the documentation needs to be fleshed out so users can build intuition on transparency. For example, a quick quiz: what color do you think comes from drawing red on top of yellow, or on top of blue?*

9.3.41.45 `set u scale`

```
'set u scale .cm_per_unit.|{as x}'
```

Set scale for x-component of arrows.

- `set u scale .cm_per_unit.` Set scale for `u` component of arrows.
- `set u scale as x` Set scale for u component of arrows to be the same as the x-scale. Equivalent to `set u scale as {rpn ..xsize.. ..xright.. ..xleft.. - /}`.

NOTE: this only works if the x-scale is LINEAR (see `set x type`).

9.3.41.46 `set v scale`

```
'set v scale .cm_per_unit.|{as y}'
```

Set scale for y-component of arrows.

- `set v scale .cm_per_unit.` Set scale for `v` component of arrows.
- `set v scale as y` Set scale for v component of arrows to be the same as the y-scale. Equivalent to `set v scale as {rpn ..ysize.. ..ytop.. ..ybottom.. - /}`.

NOTE: this only works if the y-scale is LINEAR (see `set y type`).

9.3.41.47 `set x axis`

```
'set x axis top'
'set x axis bottom'
'set x axis increasing'
'set x axis decreasing'
```

```
'set x axis unknown'
'set x axis .left. .right. [.incBig. [.incSml.]]'
'set x axis labels [add] .position_1. "label_1" [.position_2. "label_2" [...]]'
'set x axis labels automatic'
```

Control various things about the x axis.

- `set x axis top` Make next x-axis to be drawn have labels above the axis.
- `set x axis bottom` Make next x-axis to be drawn have labels below the axis.
- `set x axis increasing` Make next x-axis to be drawn have numeric labels increasing to the right. This applies only if autoscaling is done; otherwise, the supplied values (`.left. .right. [.incBig. [.incSml.]]`) are used.
- `set x axis decreasing` Make next x-axis to be drawn have numeric labels decreasing to the right. This applies only if autoscaling is done; otherwise, the supplied values (`.left. .right. [.incBig. [.incSml.]]`) are used.
- `set x axis unknown` Make Gri forget any existing scale for the x axis, whether set by another `set x axis` command or automatically, during reading of data. This is essentially a synonym for `delete x scale`.
- `set x axis .left. .right.` Make x-axis range from `.left.` to `.right.`
- `set x axis .left. .right. .incBig.` Make x-axis range from `.left.` to `.right.`, with labelled increments at `.incBig.` Note: In the case of log axes, and provided that `set x type log` has been called previously, the `.incBig.` parameter has a different meaning: it is the interval, in decades, between numbered labels; the default is 1.
- `set x axis .left .right. .incBig. .incSml.` Make x-axis range from `.left.` to `.right.`, with labelled increments at `.incBig.`, and small tics at `.incSml.` NOTE: if the axis is logarithmic, the value of `.incSml.` takes on a special meaning: if it is positive then small tics are put at values 2, 3, 4, etc. between the decades, but if `.incSml.` is negative then no such small tics are used.
- `set x axis labels .position. "label" [.position. "label" [...]]` Override the automatic labelling at axis tics, and instead put the indicated labels at the indicated x values. For example, a day-of-week axis can be created by the code:

  ```
  set x axis 0 7 1
  set x axis labels 0.5 "Mon" 1.5 "Tue" 2.5 "Wed" \
                    3.5 "Thu" 4.5 "Fri" 5.5 "Sat" \
                    6.5 "Sun"
  ```

  The command replaces any existing labels, unless the 'add' keyword is present, in which case the new label information is appended to any existing information.
- `set x axis labels automatic` Return to automatically-generated axis labels, undoing the command of the previous item.

9.3.41.48 `set x format`

```
'set x format \format|default|off
```

Set format for numbers on x axis. The format is specified in the manner of the "C" programming language. The C formats (i.e., `%f`, `%e` and `%g`) are permitted. For example, `set x format %.1f` tells Gri to use 1 decimal place, and to prefer the "float" notation to the exponential notation. The form `set x format off` tells Gri not to write numbers on the axis. To get spaces in your format, enclose the format string in double-quotes, e.g., you might use `set x format "%.0f$\circ$ W"` for a map, or `set x format "%f "` to make the numbers appear to the left of their normal location.

The default format is `%lg`.

9.3.41.49 `set x grid`

`'set x grid .left. .right. .inc.|{/.cols.}'`

Create x-grid for contour or image. If a grid already exists, an error will be declared; the way to interpolate from an existing grid to a new one is with the `interpolate x grid` command.

- `set x grid .left. .right. .inc.` Create x-grid ranging from the value `.left.` at the left to `.right.` at the right, stepping by an increment of `.inc.`.
- `set x grid .left. .right. /.cols.` Create x-grid with `.cols.` points, ranging from the value `.left.` at the left to `.right.` at the right.

9.3.41.50 `set x margin`

`'set x margin {[bigger|smaller] .size.}|default'`

Control x margin, that is, the space between the left-hand side of the page and the left-hand side of the plotting area. (Note that axis labels are drawn inside the margin; the margin extends to the axis line, not to the labels.)

- `set x margin .size.` Set left margin to `.size.` cm. It is permissible to have negative margins, with the expected effect.
- `set x margin bigger .size.` Increases left margin by `.size.` cm.
- `set x margin smaller .size.` Decreases left margin by `.size.` cm.
- `set x margin default` Set left margin to default = 6 cm.

9.3.41.51 `set x name`

`'set x name "\name"|default'`

Set name of x-axis to the indicated string. An empty string (`set x name ""`) causes the x axis to be unlabelled. The `default` is `"x"`.

9.3.41.52 `set x size`

`'set x size .width_cm.|default'`

Set the width of the plotting area. This does not include axis labels, only the interior part of the plot.

- `set x size .width_cm.` Set width of x-axis in centimeters.
- `set x size default` Set width of x-axis to default = 10 cm.

9.3.41.53 `set x type`

`set x type linear|log|{map E|W|N|S}`

Control transformation function mapping user units to centimetres on the page.

- `set x type linear` Set to linear axis.
- `set x type log` Set to log axis. To avoid clashes in the linear to log transform, this command should precede the creation of an axis scale, either explicitly through the `set`

`x axis .left. .right. ...` command or implicitly through the `read columns` command.

- `set x type map E|W|N|S` Set to be a map. This means that whole numbers on the axis will have a degree sign written after them (and then the letter `E`, etc) and that numbers which are multiples of 1/60 will be written in degree-minute format, and that similarly numbers which are divisible by 1/3600 will be in degree-minute-second format. If none of these things apply, the axis labels will be written in decimal degrees. Note that this command overrides any format set by `set x format`.

  BUG: this only has an effect if the axis is not already of type `log`.

9.3.41.54 `set y axis`

```
'set y axis left'
'set y axis right'
'set y axis increasing'
'set y axis decreasing'
'set y axis .bottom. .top. [.incBig. [.incSml.]]'
'set y axis labels [add] .position_1. "label_1" [.position_2. "label_2" [...]]'
'set y axis labels automatic'
```

Control various things about the y axis.

- `set y axis name horizontal` Make y-axis name be horizontal.
- `set y axis name vertical` Make y-axis name be vertical (default).
- `set y axis left` Make next y-axis to be drawn have labels to the left of the axis.
- `set y axis right` Make next y-axis to be drawn have labels to the right of the axis.
- `set y axis increasing` Make next y-axis to be drawn have numeric labels increasing up the page. This applies only if autoscaling is done; otherwise, the supplied values (`.left. .right. [.incBig. [.incSml.]]`) are used.
- `set y axis decreasing` Make next y-axis to be drawn have numeric labels decreasing up the page. This applies only if autoscaling is done; otherwise, the supplied values (`.left. .right. [.incBig. [.incSml.]]`) are used.
- `set y axis unknown` Make Gri forget any existing scale for the y axis, whether set by another `set y axis` command or automatically, during reading of data. This is essentially a synonym for `delete y scale`.
- `set y axis .bottom. .top.` Make y-axis range from `.bottom.` to `.top.`
- `set y axis .bottom. .top. .incBig.` Make y-axis range from `.bottom.` to `.top.`, with labelled increments at `.incBig.`
- `set y axis .bottom. .top. .incBig. .incSml.` Make y-axis range from `.bottom.` to `.top.`, with labelled increments at `.incBig.`, and small tics at `.incSml.` NOTE: if the axis is logarithmic, the value of `.incSml.` takes on a special meaning: if it is positive then small tics are put at values 2, 3, 4, etc. between the decades, but if `.incSml.` is negative then no such small tics are used.
- `set y axis labels .position. "label" [.position. "label" [...]]` Override the automatic labelling at axis tics, and instead put the indicated labels at the indicated y values. For example, a physical-condition axis can be created by the code:

  ```
  set y axis 0 1 0.5
  set y axis labels 0.25 "Weak" 0.75 "Strong"
  ```

  The command replaces any existing labels, unless the 'add' keyword is present, in which case the new label information is appended to any existing information.

- `set y axis labels automatic` Return to automatically-generated axis labels, undoing the command of the previous item.
- `set y axis name vertical` Cause future y axes to be drawn with the name aligned vertically (the default).
- `set y axis name horizontal` Cause future y axes to be drawn with the name aligned horizontally.

9.3.41.55 `set y format`

`'set y format \format|default|off'`

Set format for numbers on y axis. The format is specified in the manner of the "C" programming language. The C formats (i.e., `%f`, `%e` and `%lg`) are permitted. For example, `set y format %.1f` tells Gri to use 1 decimal place, and to prefer the "float" notation to the exponential notation. The form `set y format off` tells Gri not to write numbers on the axis. To get spaces in your format, enclose the format string in double-quotes, e.g., you might use `set y format "%.0f$\circ$ N"` for a map, or `set y format "%f"` to make the numbers appear to the right of their normal location.

The default format is `%lg`.

9.3.41.56 `set y grid`

`'set y grid .bottom. .top. .inc.|{/.rows.}'`

Create y-grid for contour or image. If a grid already exists, an error will be declared; the way to interpolate from an existing grid to a new one is with the `interpolate x grid` command.

- `set y grid .bottom. .top. .inc.` Create y-grid ranging from the value `.bottom.` at the bottom to `.top.` at the top, stepping by an increment of `.inc.`.
- `set y grid .bottom. .top. /.rows.` Create y-grid with `.rows.` points, ranging from the value `.bottom.` at the bottom to `.top.` at the top.

9.3.41.57 `set y margin`

`'set y margin {[bigger|smaller] .size.}|default'`

Control y margin, that is, the space between the bottom side of the page and the bottom of the plotting area. (Note that axis labels are drawn inside the margin; the margin extends to the axis line, not to the labels.)

- `set y margin .size.` Set bottom margin to `.size.` centimeters. It is permissible to have negative margins, with the expected effect.
- `set y margin bigger .size.` Increases bottom margin by `.size.` centimeters.
- `set y margin smaller .size.` Decreases bottom margin by `.size.` centimeters.
- `set y margin default` Set bottom margin to default = 6 cm.

9.3.41.58 `set y name`

`'set y name "\name"|default'`

Set name of y-axis to the indicated string. An empty string (`set y name ""`) causes the x axis to be unlabelled. The `default` is `"y"`.

9.3.41.59 `set y size`

`'set y size .height_cm.|default'`

Set the width of the plotting area. This does not include axis labels, only the interior part of the plot.

- `set y size .height_cm.` Set height of y-axis in centimeters.
- `set y size default` Set width of y-axis to default = 10 cm.

9.3.41.60 `set y type`

`set y type linear|log|{map N|S|E|W}`

Control transformation function mapping user units to centimetres on the page.

- `set y type linear` Set to linear axis.
- `set y type log` Set to log axis. To avoid clashes in the linear to log transform, this command should precede the creation of an axis scale, either explicitly through the `set y axis .left. .right. ...` command or implicitly through the `read columns` command.
- `set y type map N|S|E|W` Set to be a map. This means that whole numbers on the axis will have a degree sign written after them (and then the letter `N`, etc), and that numbers which are multiples of 1/60 will be written in degree-minute format, and that similarly numbers which are divisible by 1/3600 will be in degree-minute-second format. If none of these things apply, the axis labels will be written in decimal degrees. Note that this command overrides any format set by `set y format`.

  BUG: this only has an effect if the axis is not already of type `log`.

9.3.41.61 `set z missing`

`set z missing above|below .intercept. .slope.`

Set `z` column to be missing whenever the associated `y` and `x` columns are above/below the line defined by $y =$ .intercept. + .slope.x

9.3.42 `show`

`'show ...'`

Show some information by printing it to standard output.

- `show all` Show lots of information about plot.
- `show axes` Show information about axes.
- `show color` Show the current pen color used for lines and text. This is not to be confused with image color, which is independent.
- `show colornames` Show all colors known by name, as defined by `read colornames` command and also the builtin colors defined automatically (e.g. `white`, `black`, `red`, etc), (Section 9.3.33.1 [Read Colornames], page 106).
- `show columns` Show x, y, z, u, v column data.
- `show columns statistics` Show means, std devs, etc for columns.
- `show flags` Show values of all flags. (Developers only.)
- `show grid` Show an indication of the grid data (used for contouring).

- `show grid mask` Show 1 if grid data are valid or 0 if contours will not extend into this region.
- `show image` Show information about image, such as a histogram of values, and, if the image is small enough, the actual data.
- `show license` Show the license for Gri, which outlines how users are allowed to share it freely.
- `show misc` Show miscellaneous information about the plot, the data, etc.
- `show next line` Show next line of data-file.
- `show synonyms` Show values of all synonyms, whether built-in or user-defined.
- `show stopwatch` Show elapsed time since first call to this command in the given Gri program.
- `show time` Show the current time.
- `show traceback` Show traceback (i.e., the tree of commands being done at this instant).
- `show variables` Show values of all variables, whether built-in or user-defined.
- `show .value.` Show value of indicated variable.
- `show {rpn ...}` Show result of computing indicated expression.
- `show "some text"` Print the indicated string. You may use a double-slash to prevent Gri from substituting synonym values; thus it is common to do e.g.

  ```
  \var = "Temperature"
  show "Plotting \\var = 'var'"
  ```

  which will produce the output line

  ```
  Plotting \\var = 'Temperature'
  ```

- `show "time=" .time. "; depth=" .depth.` Print strings and values as indicated. If the last item is ellipses (three dots with no spaces between them), then no newline is printed; this makes the next `show` statement print on the same line.

  To get a newline in a printed string, use the three-character glyph `\<<`, and to get a horizontal tab, use `\>>`, as in the examples below

  ```
  \a = "HI"
  show "\\a=\a"
  system echo -e "a\\nb"
  show "first line\<<second line"
  show "first line\<<\>>(tabbed) second line"
  show "first line\<<\>>(tabbed) second line"
  ```

9.3.43 skip

```
'skip [forward|backward] [.n.]'
```

- `skip` For ascii files, skip next line in the data file. For binary files, skip forward 1 byte.
- `skip backward` For ascii files, skip backward 1 line in the data file. For binary files, skip backward 1 byte.
- `skip .n.` or `forward .n.` For ascii files, skip forward `.n.` lines in the data file. For binary files, skip forward `.n.` bytes.
- `skip backward .n.` For ascii files, skip backward `.n.` lines in the data file. For binary files, skip backward `.n.` bytes.

9.3.44 sleep

```
'sleep .sec.'
```

Cause Gri to sleep for the indicated number of seconds, which should be a positive integer. This command is ignored if `.sec.` is zero or negative, and the value of `.sec.` is first rounded to the nearest integer.

Normally, this command is used only by the developer, as a way to slow down Gri execution, to allow easier monitoring for debugging purposes. Beware: it is tricky to kill a sleeping job!

9.3.45 smooth

All these smoothing commands ignore the **location** of the data. For equispaced data these algorithms have the standard interpretation in terms of digital filters. For non-equispaced data, the interpretation is up to the user.

```
'smooth {x [.n.]} \
  | {y [.n.]} \
  | {grid data [.f.|{along x|y}]}'
```

The `smooth x` command does smoothing by the following formula

```
x[i-1]   x[i]   x[i+1]
------ + ---- + ------
  4       2       4
```

The `smooth x .n.` command does boxcar smoothing with centred boxcars `.n.` points wide. The `smooth y` command does the same as `smooth x`, but on the `y` column.

There are several methods of smoothing grid data. Note that isolated missing values are filled in by each method. (Let the author know if you'd like that 'feature' to be an option.)

The `smooth grid data` command smooths gridded data, by weighted average in a plus-shaped window about each gridpoint. The smoothing algorithm replaces each interior grid-point value `z[i][j]` by

```
z[i][j]   z[i-1][j] + z[i+1][j] + z[i][j-1] + z[i][j+1]
------- + -----------------------------------------
   2                          8
```

Points along the edges are smoothed by the same formula, after inventing image points outside the domain by planar extrapolation.

The `smooth grid data .f.` command performs partial smoothing. A temporary fully-smoothed grid `zSMOOTH[i][h]` is constructed as above, and a linear combination of this grid and the original grid is used as the replacement grid:

```
z[i][j] = (1-f) * z[i][j] + f * zSMOOTH[i][j]
```

where `f` is the value indicated on the command line. Thus, `smooth grid data 0` performs no smoothing at all, while `smooth grid data 1` is equivalent to `smooth grid data`.

The `smooth grid data along x` command smooths the grid data across `x` (i.e., horizontally), by replacing each value `z[i][j]` with the value

```
z[i][j]   z[i-1][j] + z[i+1][j]
------- + ---------------------
   2               4
```

Points along the edges are smoothed by the same formula, after inventing image points outside the domain by linear extrapolation.

The `smooth grid data along y` command does the same thing as `smooth grid data along x`, but the smoothing is along `y`.

See also Section 9.3.11 [Filter], page 95, a generalization of `smooth x|y` which allows for more sophisticated filters.

9.3.46 source

`'source \filename'`

Perform the commands in the indicated file.

If the file cannot be found, an error results. Contrast this with the `insert` command (Section 9.3.20 [Insert], page 100), which has the ability to search for the file through a user-specified path (Section 9.3.41.39 [Set Path To], page 129).

9.3.47 sprintf

`'sprintf \synonym "format" .variable. [.variable. [...]]'`

Write numbers into a synonym (text string). This is useful for labelling plots.

`sprintf \out "a = %lf b = %.2f" .a. .b.` - Create a synonym called `\out`, and print the values of the variables `.a.` and `.b.` into it. If `.a.` = 1 and `.b.` = 0.112, then `\out` will be `"a = 1 b = 0.11"`

Formatting codes are as in the C programming language, eg:

```
%.2f  -- Use floating point with 2 decimal places.
%9.2f -- As above, but number takes 9 characters.
%e    -- Use exponential notation.
```

CAUTION: Variables are stored in the **floating point** in Gri, so you must use a format like `"%f"`, **not** an integer code like `"%d"`. If you want an integer, use `"%.0f"`.

9.3.48 state

`'state save|restore|display'`

The `save` operation pushes a record of the graphics state (pen and font characteristics, margins, axis lengths, min/max/inc values on axes, etc) onto a stack. The `restore` operation replaces the present state with whatever is on top of the stack, and then pops the stack. Use `display` to see some of the state properties.

The `state` command is useful for temporary changes of axis properties, etc.

BUG: only line characteristics (width, color) and font characteristics (font, size, color) are saved so far. In fact, the full list of what should be saved has not yet been finalized by the author.

9.3.49 superuser

`'superuser [value]'`

Allow extra debugging information and commands. Normally, this command and the corresponding commandline flag `-superuser` are only used by programmers altering the Gri source.

These are the flags and their meanings:

- **1** Print cmdline before/after substituting synonyms.

- **2** Print cmdline before/after substituting rpn expressions.
- **4** Print all new commands as they are being defined
- **8** Print the system commands that are used with `open "...|"` and in other instances.
- **128** Changeable; only author should use this.
- **256** Changeable; only author should use this.

Note that all flags are equal to 2 raised to an integer power. Since the flag values are detected by a bitwise OR, you can combine flags by adding; thus specifying a flag of 5 yields flags 1 and 4 together; specifying 15 yields flags 1, 2, 4 and 8.

9.3.50 `system`

```
'system \system-command'
```

Tell the operating system to perform the indicated action. Whatever string follows the word `system` is passed directly to the operating system, **after** substitution of synonyms if any exist.

If your system command contains double-slashes, you must protect them from Gri (which will interpret them as comments) by enclosing in double-quotes, e.g. `system cat A | sed -e "s/foo//g" | cat > B`. (In the particular case of the `sed` command you could also do `system cat A | sed -e "s:foo::g" | cat > B`.

Note that `rpn` expressions are not evaluated, and variable values are not substituted before passing the string to the operating system. The exit status is stored in the builtin variable `..exit_status..`.

There are two ways to use the system:

- **Assign output to synonym**: The form `\synonym = system ...` does the system command and then inserts the output from that command into the indicated synonym.)
- **Just run a command**: The command `system ls` will list the files in the current directory.

For long commands, there are two approaches, the second preferred:

- **Use continuation lines**: String a lot of information onto one effective system line, using the `\` line-continuation character at the ends of lines. The problem is that it is very easy to lose one of these backslashes. The next method is better.
- **Here-is syntax** The here-is syntax of many unix shells is also provided. If the system command contains the characters `<<` followed by a word (with no space between!) then Gri will issue a system command which includes not only this line but also all succeeding lines, until a line is found which matches the indicated word precisely (with no leading space allowed). The `<< "WORD"` syntax is also supported, meaning that the operating system is being told not to mess with the dollar-signs – needed in perl.

  Be careful using this inside a new-command. Gri Recognizes the end of the body of a new-command by a line with `}` in the **first column**, and no non-white characters thereafter. If you system command happens to use a line with a curly brace (as in a loop in perl, for example), you must put whitespace before the brace. This won't affect the system command, but it will let Gri correctly realize that this is **not** the end of the new-command. For more information on new-commands (Section 10.11.2 [Parsing], page 174).

  Caution: Before sending the string to the system, Gri first translates any synonyms present. Be careful with this, since system commands calling awk, etc, very often use backslashes for the newline character `\n` within strings. If you have a synonym whose name starts with `\n`, you can get a conflict. For example, the awk command `print`

`"foo\nbar";` should print a line with `foo` on it, followed by a line with `bar` on it, but it will instead print a single line with `fooMISTAKE`, if you had previously defined a synonym as `\nbar = "MISTAKE"`. One way to avoid this mistake is to make sure any `\n` characters appear at the end of strings, and then simply avoid having a synonym named `\n`.

Here is a Perl example.

```
\m = "Foo bar"
system perl <<"EOF"
$a = 100;
print "foo bar is \m, and a is $a\n";
print "BETTER: foo bar is \m, and a is $a\n";
print "Q: was a 100?\n";
EOF
```

Some more examples:

- To get the first 15 lines of a file called '`foo.dat`' inserted into another file called '`bar.dat`', you might do the following. Only the first method works; the second will fail because `.n.` will not be translated before passing to the operating system.

```
\num = "-15"
system head \num foo.dat > bar.dat
# Following will not work correctly because .num.
# will not be translated
.num. = -15
system head .num. foo.dat > bar.dat
```

- Issue a unix command to get a listing of files in the current working directory, and pipe them into the `more` system command.

```
system ls -l *c | more
```

- Store the date and time into a synonym, and use it in a title:

```
\time = system date
...
draw title "Plotted at time \time"
```

- Use `awk` to prepare a two-column table of x, ranging from 0 to 1 in steps of 0.1, and sin(x). The table is stored in a file whose suffix is the process ID of the Gri job. This file is then opened, and the data plotted. Finally, a system command is issued to remove the temporary file.

```
system awk 'BEGIN { \
    for (x=0; x<1; x+=0.1) { \
      printf("%f %f\n", x, sin(x)) \
    } \
  }'  > tmp.\.pid.
open tmp.\.pid.
read columns x y
close
system rm tmp.\.pid.
draw curve
```

Note: in unix, this command calls the Bourne shell, not the C-shell that is often used interactively. For many simple uses, the only difference from normal interactive use will be that `~` is not expanded to the home directory. For example, you should write

```
system awk -f $HOME/foo/bar/cmd.gawk
```

instead of the

```
system awk -f ~/foo/bar/cmd.gawk
```

that you might expect from interactive C-shell use. RETURN VALUE: Sets `\.return_value` to system status `N status`

9.3.51 unlink

`'unlink \filename'`

Delete a filename and possibly the file to which it refers. On non-unix machines, this simply means to delete the file. On unix machines, the action is more subtle. The unix OS permits several processes to use a given file at once. Therefore, `unlink` doesn't immediately remove the file, but instead waits until other processes are done with it. Most users will never realize the difference, however, and it is safe to think of `unlink` as simply removing the file. To learn more, type `man unlink` in a unix shell.

A common use of `unlink` is to remove files that were created with the `tmpname` facility (Section 10.16.1 [Using OS Inside Gri], page 183), e.g.

```
\tmp = tmpname
# do some system commands to put data into this file
open \tmp
read columns x y
draw curve
unlink \tmp
```

9.3.52 while

`'while .test.|{rpn ...}'`

Perform statements in loop while the value of `.test.` or the RPN expression is nonzero. The end of the loop designated by a line containing the words `end while`. The value `.test.` may be an rpn expression. To leave the loop prematurely, use a `break` statement. Upon encountering a `break` statement, Gri jumps to the line immediately following the loop. If the `-chatty` option is nonzero, a notification is printed every 1000 passes through the loop, as a debugging measure to catch unintended infinite loops.

Examples:

- Loop forever, printing a message over and over.

```
while 1
  show "This loops forever. Need to 'break'"
end while
```

- Read number pairs from a file, plotting bullets at the indicated locations. Note the use of an infinite loop, with a break condition following an end-of-file test. (Do not be tempted to write such loops as `while !..eof..` because that would not catch the end of file until the next time through the loop. The result would be to draw the last bullet twice, since the `read` will not update the variables when the end of file is encountered.)

```
while 1
  read .x. .y.
  if ..eof..
    break
  end if
  draw symbol bullet at .x. .y.
```

```
end while
```

- Loop 10 times, printing the values of `.i.` as they range 0, 1, . . . , 9. After exiting from the loop, `.i.` will equal 10. Be **careful** to use the correct rpn greater-than test to avoid an infinite loop.

```
.i. = 0
while {rpn .i. 10 >}
  show .i.
  .i. += 1
end while
```

9.3.53 The `write` commands

The `write` commands write various things.

If the filename is '`stdout`', the information is written to the standard output device (ie, the screen); if it is '`stderr`', the information is written to the standard error device (ie, the screen).

IMPORTANT NOTE: The `write` commands **append** to the output file, as opposed to overwriting the contents of the file. Therefore if you've run the Gri script before, and want fresh output, make sure to do something like the following

```
system rm -f the_grid.dat
write grid to grid.dat
```

9.3.53.1 `write columns`

```
'write columns to \filename'
```

Append data columns to the end of the indicated file.

9.3.53.2 `write contour`

```
'write contour .value. to \filename'
```

Append to the named file the (x,y) pairs defining the contour of the indicated value.

The first line of output is a header line, containing two numbers: the contour value and the missing value. Then the (x,y) pairs are written a line at a time, with missing values being used to indicate ends of segments. A blank line is written after the last data pair. For example, if the contour contained two closed regions, Gri would output a pair of missing values as one of the xy pairs, to denote the separation of the two curves. You could read and plot the output as in this example

```
write contour 10 to contour.out
open contour.out
read .contour_value. .missing.
set missing value .missing.
read columns x y
draw curve
```

9.3.53.3 `write grid`

```
'write grid to \filename [bycolumns]'
```

Append grid to the end of the named file. Storage is in `%f` format, and is in normal image order. If the keyword `bycolumns` is present, then the grid is transposed first, in such a way that `read grid data bycolumns` performed on that file will read back the original grid data.

9.3.53.4 `write image`

```
'write image ... to \filename'
```

The variants of this command write various things about the image to the named file, as illustrated in the following table.

- `write image to image.dat` Append image to the end of the named file. Storage is by unsigned-char, and is in normal image order. There is no header.
- `write image rasterfile to image.dat` Append image to the end of the named file, in Sun Rasterfile format.
- `write image pgm to mask.dat` Append image mask to the end of the named file, in PGM 'rawbits' format.
- `write image mask to mask.dat` Append image mask to the end of the named file. Storage is by unsigned-char, and is in normal image order.
- `write image mask rasterfile to mask.dat` Append image mask to the end of the named file, in Sun Rasterfile format.
- `write image mask pgm to mask.dat` Append image mask to the end of the named file, in PGM 'rawbits' format.
- `write image colorscale to colorscale.dat` Append image colorscale transform to the end of the named file. Storage is a series of 256 lines, each containing 3 numbers (for Red, Green and Blue) in the range 0 to 1. The file is suitable for reading with the `read image colorscale` command.
- `write image grayscale to grayscale.dat` Append image grayscale transform to the end of the named file. Storage is a series of 256 lines, each containing a number in the range 0 to 1. The file is suitable for reading with the `read image grayscale` command.

10 Programming in the Gri Language

The Gri programming language has `if` statements to control program flow, and a `while` statement to repeat commands. There are two data types in Gri: "variables" (to store numbers) and "synonyms" (to store character strings). Gri recognizes commands by matching statements against its list of known commands. This list is extensible; it is easy to add new commands as extensions to Gri.

10.1 Defaults

At startup time, Gri sets the values of some things, like font size. Since Gri is still under development, some of these defaults might change, so you should not rely on them remaining the same. Presently, the defaults are equivalent to:

```
set arrow size 0.2         # (cm)
set axes style 0
set beep off
set clip off
set clip postscript off
set contour format %lg
set contour label position ? ?
set contour labels horizontal
set contour labels whiteunder
set dash off
set font size 12           # (pt)
set font to helvetica
set graylevel 0            # Black ink
set ignore initial newline off
set input data window x off
set input data window y off
# Following set (curve, axes, symbol) widths to width
# of rapidograph pens called (6x0, 3x0,  6x0)
set line width 0.709        # (pt) for curves
set line width axis 0.369   # (pt) for axes
set line width symbol 0.369 # (pt) for symbols
set missing value 1.0e22
set page portrait
set page factor 1
set symbol size 0.1        # (cm)
set tic size 0.2           # (cm)
set tics out
set trace off
set x format %lg
set x margin 6.0           # (cm)
set x name "x"
set x size 10              # (cm)
set x type linear
set y axis name vertical
set y format %lg
set y margin 6.0           # (cm)
set y name "y"
```

```
set y size 10              # (cm)
set y type linear
```

(NOTE: Programmers may alter the gri source file '`defaults.h`' and then recompile Gri, if they feel the need to change these things. Also, see the file '`startup.c`' and the function `gr_begin()` in '`gr.c`'.)

10.2 Online Help

Type `help` to get a list of available commands and other topics of interest. Here's how Gri responds

```
Type 'help' followed by a command-name:
 assert      cd            close         convert
 create      debug         delete        differentiate
 draw        expecting     filter        flip
 get         help          if            ignore
 input       insert        interpolate   list
 ls          mask          move          new
 open        pwd           query         quit
 read        regress       reorder       rescale
 resize      return        rewind        set
 show        skip          sleep         smooth
 source      sprintf       state         superuser
 system      write
Or type 'help -' followed by a topic from this list:
 example     extending     files         math
 strings     synonyms      variables     manual
```

Some commands have more words than shown. You can type these additional words to narrow the help down; otherwise Gri will give you help on all commands that begin with the indicated words. For example, try `help set` and `help set x`. When you ask for help on a multi-word command, Gri tells you about all commands which begin with the words you've typed. Thus,

```
help
help draw
help draw zero
help draw zero line
```

narrow in on the command `draw zero line`. The response to the most complete request is

```
'draw zero line [horizontally|vertically]'
  draw zero line
    Draw line y=0 if it is within axes
  draw zero line horizontally
    Draw line y=0 if it is within axes
  draw zero line vertically
    Draw line x=0 if it is within axes
```

The part enclosed in angled quotes is the syntactical description of the command. (NOTE: The square brackets indicate an optional word (in this case) or words. The vertical bar indicates that either the item on the left or the item on the right may appear; it is a logical OR operator. The only other special characters in syntax descriptions are the braces `{}`, which are used to enclose multiple words which act as one unit; they are used to clarify the choices presented to the OR operator.) Following the syntactical description

are examples. Each example is indented 2 spaces, and a description of it (which always starts with an upper-case character and ends with a period, to indicate that it is an English description) follows that, indented by an additional 2 spaces.

10.3 Long Command Lines

To extend a command across several lines, use a backslash \ at the **very** end of all lines but the last:

```
draw line from \
  10 20 \
  to \
  10 30
```

10.4 Variables

10.4.1 About variables

Variables store numbers. As it reads your program, Gri substitutes variable values any place a variable appears where a number normally would. For example, in the code below `.number.` is a variable storing the value 10, so the two `read` statements have the same effect:

```
.number. = 10
read columns .number. x y
read columns 10 x y
```

Variable names begin and end with a single period (example: `.num.`). (Gri uses this odd notation to distinguish variable names from "normal" words, which is necessary because Gri does not have a limited list of keywords as other languages do. Thus, the C programming language is happy to let you use a variable name like `latitude`, since it is not a keyword, but Gri is not, since it might like to use that word itself in a new command.)

You should not use names beginning and ending with double periods, because Gri uses names like that to store built-in variables for its own use (e.g., `..xsize..` saves the width of the plot).

To store a number into a variable, use a command like

```
.time. = 10
```

or

```
.time. = {rpn 10 sin}
```

Storage is automatically set aside when you assign into a nonexistent variable; no "declaration" statements are required as in the C language.

The Gri command, `new` (Section 9.3.25 [New], page 101), allows you to have several "versions" of a variable. This is useful for local storage in new commands, inside `if` statements, etc, since it lets you use temporary variables without worrying about overwriting values outside the local block of code. The syntax is `new .variable. = value` (where, as usual, `value` may be an rpn expression (Section 10.9 [rpn Mathematics], page 161). Here is an example:

```
'foo bar'
{
```

```
  new .a.          # Get storage
  .a. = 10         # Store a local value
  show "Locally, .a.=" .a. " (expect 10)"
  delete .a.       # Delete this local one
}
.a. = 1
show "Global version has .a.=" .a. " (expect 1)"
foo bar
```

To see if a given named variable (or synonym) exists, use the RPN operator `defined` (Section 10.9 [rpn Mathematics], page 161).

10.4.2 User variables

You can get Gri to read values for variables from your file. Here's how to read a number from a header line and then read that many lines of columnar data:

```
open file.dat
read .num.
read columns .num. x y
```

You can define variables within the Gri program:

```
.num. = 10
read columns .num. x y
```

You can get variables interactively from the user, using the `query` command. (If the user types carriage-return, or if the command-line flag `-y` was specified when invoking Gri, the value 100 will be assigned to `.num.`). For example,

```
query .num. "Number of rows to read?" (100)
read columns .num. x y
```

Gri allows you to use a previous value of the variable in the default string, as in this example:

```
.start. = 8                              # default
.stop. = 2                               # default
query .start. "Start time? " (.start.)
query .stop.  "Stop time?  " (.stop.)
```

Variables can be manipulated using reverse polish notation (RPN) mathematical operations (Section 10.9 [rpn Mathematics], page 161).

Variables are often useful in `if` statements. Here are some examples:

```
read .num_pts.
if .num_pts.
  show "There are some data"
  read columns .num_pts. x y
else
  show "There are no data"
end if
# ...
read .latitude.
if {rpn .latitude. 10 <}
  read .num.
  read .num. x y
  draw curve
```

```
else
  show "Skipping data North of 10deg N"
  read .num.
  skip .num.
end if
```

10.4.3 Built-in variables

Built-in variables ([Index of Builtins], page 259) have names which begin and end with **two** periods. For example, `..xsize..` is the width of the x-axis in centimetres. You may use these variables as you wish (example: `..xsize.. = 4` is an alternative to `set x size 4`), but you must be aware that these are not "free" variables for you to use for arbitrary purposes. You can find out what the built-in variables are by the command `show variables`.

There are two types of variables

- **Startup** variables, which are created by Gri at startup time. These variables can be relied upon to exist (barring changes in Gri itself), unless you `delete` them.
- **Spontaneous** variables (which are created by certain Gri commands, and only exist if these commands have been executed). For example, the `regress` command defines `..coeff0..` (the intercept of the fitted line), `..coeff1..` (the slope of the fitted line), `..R2..` (the correlation coefficient).

To see the values of the built-in variables (along with the user variables), use `show variables`. Here are some useful builtin variables:

- `..arrowsize..` Stores either a positive number representing the halfwidth of arrowheads measured in centimetres, or a negative number giving the negative of the ratio of arrowhead halfwidth to arrow length (Section 9.3.41.2 [Set Arrow Size], page 115).
- `..batch..` Flag used for batch mode.
- `..debug..` Equal to 1 if the `-debug` command-line flag was set. Flag used for debugging (Chapter 3 [Invoking Gri], page 7). The `..debug..` built-in variable is useful in isolating code to use only in test runs. For example, you might use

  ```
  if ..debug..
    show "Following are the column data"
    show columns
  end if
  ```

  When you run the program with command-line `gri -debug file.gri` the code in the `if` block will print out the columnar data, but when you run it with `gri file.gri` these lines are not printed.
- `..eof..` Flag indicating whether an end-of-file was encountered on the last `read columns`.
- `..words_in_dataline..` Number of words on last dataline. This is useful in constructs like

  ```
  open tmp.dat
  .num. = 0
  while 1
    read .a. .b.
    if !..words_in_dataline..
      show "Got empty line or EOF, so break loop"
      break
    end if
  ```

```
  show "a=" .a. "b=" .b.
  show "; words in line=" ..words_in_dataline..
  .num. += 1
end while
show "Got " .num. "data lines."
```

- `..fontsize..` Size of letters, measured in points; there are 72.27 points in an inch and 28.45 points in a centimetre. The mathematical operators `pttocm` and `cmtopt`, which do conversion between points and centimetres, are often useful in labelling data curves (Section 10.9 [rpn Mathematics], page 161).
- `..graylevel..` Graylevel to use in drawing lines, text, etc. Black ink is 0; white paper is 1. **See also** `..red..` etc.
- `..image_height..` Height of image, or 0 if no image defined yet.
- `..image_width..` Width of image, or 0 if no image defined yet.
- `..length_dash..` Length in cm of dashes in dashed lines.
- `..length_blank..` Length in cm of blanks in dashed lines.
- `..linewidth..` Width of lines for data curves (Section 9.3.41.33 [Set Line Width], page 126).
- `..linewidthaxis..` Width of lines on axes (Section 9.3.41.33 [Set Line Width], page 126).
- `..linewidthsymbol..` Width of lines in symbols (Section 9.3.41.33 [Set Line Width], page 126).
- `..missingvalue..` Missing value code, also stored in the synonym `\.missingvalue.`; (Section 9.3.41.34 [Set Missing Value], page 127).
- `..num_col_data..` Number of column data that exist. You might want to use this after `read columns` to see if a data file actually had any data in it, or use it in accessing individual elements of columns (Section 10.9 [rpn Mathematics], page 161).
- `..publication..` Flag for final copy of plot. The command-line option `-p` sets the value of `..publication..` to 1. A typical, and highly recommended, code fragment is

```
if !..publication..
  draw time stamp
end if
```

- `..red..`, `..green..`, `..blue..` Description of present color. The values are between 0 and 1, with (0,0,0) being black and (1,1,1) being white. If color is gray, all these will be equal. You may assign to these, but it will **not** change the color.
- `..symbolsize..` Size of symbols in centimetres.
- `..superuser..` Equal to 0 if the flag was not set, or equal to the flag if it was.
- `..tic_direction..` Direction of axis tics, 1 for inside or 0 for outside.
- `..tic_size..` Size of axis tics in centimetres.
- `..trace..` Equal to 1 if the `-trace` command-line flag was set. Used for tracing program execution.
- `..xinc..` x increment on axes.
- `..xleft..` x value at left of plot.
- `..xmargin..` Left margin, in centimetres.
- `..xright..` x value at right of plot.
- `..xsize..` x-axis length in centimetres.
- `..ybottom..` y value at bottom of plot.

- `..yinc..` y increment on axes.
- `..ymargin..` Bottom margin in centimetres.
- `..ysize..` y-axis length in centimetres.
- `..ytop..` y value at top of plot
- `..exit_status..` The exit status from the most recent `system` call (or 0 if no system calls have been done yet).

You may use any of these built-in variables anywhere. For example, here's how to stack 3 graphs vertically on the page:

```
.offset. = {rpn ..ysize.. 3 + }
open file1
read columns x y
close
draw axes
draw curve

..ymargin.. += .offset.
open file2
read columns x y
draw axes
draw curve
close

..ymargin.. += .offset.
open file3
read columns x y
draw axes
draw curve
close
```

The first line needs a bit of explanation. It is a reverse-polish expression. The format is `{` followed by `rpn` followed by an expression followed by `}`. Within the expression, spaces must separate operands. This makes `.offset.` equal to the height of y-axis plus 3 cm, so plots are separated by 3 cm. To learn more about `{rpn ... }` (Section 10.9 [rpn Mathematics], page 161).

Another possibly unfamiliar thing is the code `+=`. It means take the thing on the left hand side, and add to it the thing on the right hand side. (In this case, it is used to increase the y margin by the value of `.offset.`.)

10.5 Synonyms

Synonyms are used by Gri to store character strings. Gri denotes synonyms with words beginning with backslash (e.g., `\syn`), following the TEX convention.

10.5.1 Naming convention for synonyms

Synonym names begin with a backslash (e.g., `\filename`). After the backslash, Gri expects a letter (upper or lower case) or one or more periods. Following this is an arbitrary string of letters, numbers, or underscores. If there are periods at the start, then the same number of periods must be used at the end. The following are some examples

```
\simple = "Howdie"
\.longer_example. = "Dots and underscores are ok too"
\a2 = "OK for number at end ..."
\a3bb = "... and inside"
```

Gri defines several synonyms for its own use, so that if you modify these, you may get strange results. Each of these starts and ends with a single period.

There is an exception to the above rule, one which mostly comes up when using netCDF files which may have variable names that may contain punctuation. Gri permits synonym names to have punctuation characters (but not blanks or tabs) in synonym names, provided that the second character in the name is an opening brace and that the last character is a closing brace, e.g.

```
\{foo.bar} = "Foo bar"
```

This is used particularly for files in the netCDF format, for reading variable attributes, which by netCDF convention use a colon (`:`) to separate variable name and attribute name (Section 9.3.33.9 [Read Synonym or Variable], page 112). For more information on netCDF format, see

```
http://www.unidata.ucar.edu/packages/netcdf/index.html
```

Synonyms may be freely embedded in strings (a common example is `draw title "Data from file '\datafile'"`. They may also appear anywhere in commands (e.g., `open \filename`). The exception to this rule is that Gri ignores your synonyms within math mode, in order to prevent clashes (e.g. you might define `\alpha` as a synonym storing the value `"foo bar"`, but Gri will ignore this within math-mode, so that `$\alpha$` will still mean the Greek letter alpha).

To get a backslash in a string without Gri thinking it is part of a synonym, use two backslashes (e.g., `show "The backslash character \\ is used for synonyms."`). This may sometimes be required in `system` commands (Section 9.3.50 [System], page 139), to prevent Gri from converting substrings like `\n` (which many system commands use to represent the newline character). For example, the command `system perl -e 'print "foo\nbar";'` will be mangled if Gri has already been told that `\nbar` is a synonym. (There will be no problem if `\nbar` is not an existing synonym, since Gri will then just leave it in place.) To be sure that no mangling can occur, replace each backslash with two backslashes. This tells Gri not to try to substitute a synonym at that location. In the example below, the first system call prints `fooled you!` on one line line, because Gri substituted for what it thought was a synonym called `\nbar`; the second (correctly) prints `foo` on one line and `bar` on the next.

```
\nbar = "led you!"
system perl -e 'print "foo\nbar\n";'
system perl -e 'print "foo\\nbar\\n";'
```

Similarly, if your system command is expecting to see `\t` (for a tab character), then you must write `\\t` to prevent Gri from trying to substitute a synonym named `\t`.

The `show` command has a special syntax for permitting newlines and tabs in strings (Section 9.3.42 [Show], page 135).

10.5.2 Some uses for synonyms

Synonyms store strings and are useful for anything strings are useful for, e.g. filenames, plot labels, names of variables, etc.

10.5.2.1 Using synonyms to generalize code

Synonyms are often used to store filenames, since then only a single line of a file may need to be altered, in order to work with another file, e.g.

```
\filename = "columns.dat"
open \filename
# a lot more code using the file name
```

10.5.2.2 Using synonyms to store OS output

Synonyms provided a convenient way to store information from the OS.

```
# Show the date.
\date = system date
show "Time is \date"

# Show the command file name, then use the system
# to construct a filename with the same beginning
# but ".dat" as the ending instead of ".gri".
show "The commandfile name is \.command_file."
\fn = system echo `basename \.command_file. .gri`.dat
show "A filename constructed from this is \fn"
```

This example uses the Unix system commands `echo` and `basename` to construct a filename ending in '`.dat`', from the command file name (stored in the builtin string `\.command_file.`), assuming that the command file name ends in '`.gri`'.

NOTE: As usual, if the system command contains the Gri comment designator (the string `#`), protect it with double-quotes (Section 9.3.50 [System], page 139).

10.5.2.3 Storing user responses via `query`

You can ask the user for the contents of strings:

```
query \filename "What's the data file?" ("file.dat")
```

The prompt `What's the name of the data file?` is typed to the terminal, and whatever string the user types is inserted into the synonym `\filename`. If the user types nothing, but simply presses carriage return, the (optional) default string (which must be enclosed in parentheses as shown) is put into `\filename`. Note that the default is ignored if it is not written properly: it must be enclosed in double quotes enclosed in parentheses, with no intervening spaces.

10.5.2.4 Storing File Contents

You can read the contents of synonyms from a file:

```
open \directory_file
read \file_name
close
open \file_name
read columns x y
```

The first (space-separated) word is read into the the first synonym after the `read` command, the second word into the second synonym, and so on. If the word you want is not near the front of the line, you can use the command `read line` to get the whole line, then use the method described above to extract the word you want. Indexing begins with 0, remember.

10.5.2.5 Working with words within strings

Sometimes a synonym will contain several words that you need to work with indidually (e.g. it might contain a list of files that should be processed). There are two ways to do this.

The `word of` *syntax.*

```
\sentence = "This sentence has five words"
\first_word = word 0 of "\sentence"
\last_word = word 4 of "This sentence has five words"
```

The `[]` *syntax*

Individual words of synonyms may be accessed by prefixing the synonym name with the index number of the word (starting at 0) enclosed in square brackets.

The number in the square brackets may be a constant, a variable, or a synonym, but not a more complicated expression. If the index value is a floating-point number, it is first rounded to the nearest integer. If the index value is negative or exceeds the number of words minus 1, then an empty string is retrieved.

If **no number** appears in the square brackets, the result is the number of words in a synonym.

```
\syn = "This has 4 words in it"
show "\[0]syn   ... gives 'This'"
show "\[1]syn   ... gives 'has'"
.i. = 3
show \[.i.]syn  ... gives 'words'"
\i = "3"
show \[\i]syn   ... gives 'words'"
show "\[]syn    ... gives '6', i.e. number of words"
```

10.5.3 Some important builtin synonyms

Within mathematics mode (portions of strings enclosed within dollar-signs), Gri stores the definitions of many Greek letters and mathematical symbols as math-mode synonyms (Section 10.10.2 [Mathematical Text], page 171).

Global synonyms are shared among commands. To see the built-in global synonyms ([Index of Builtins], page 259) use `show synonyms`, which produces output that looks something like the following.

```
Synonyms...
    \.missingvalue.          = "10000000000000000000.000000"
    \.return_value.          = ""
    \.version.               = "2.7.0"
    \.pid.                   = "3043"
    \.wd.                    = "/home/kelley"
    \.time.                  = "Sun May 20 13:18:32 2001"
    \.user.                  = "kelley"
    \.host.                  = "Intrusion.phys.ocean.dal.ca"
```

```
\.system.                = "unix"
\.home.                  = "/home/kelley"
\.lib_dir.               = "/usr/share/gri"
\.command_file.          = "stdin"
\.readfrom_file.         = "stdin"
\.ps_file.               = "gri-00.ps"
\.path_data.             = "."
\.path_commands.         = "."
```

These things will be obvious to unix users; for example `\.pid.` is the process ID of the job (often used in names for temporary files), and `\.wd.` is the working directory (often used in `draw title` commands to indicate in which directory the gri job was run.

Some commands set `\.return_value.` to non-blank; the meaning of the return value varies from command to command.

10.5.4 Alias synonyms: the `\@alias` syntax

Sometimes you need to work with a variable or a synonym whose name can only be determined at run-time, perhaps through interaction with the user, examination of a datafile, or examination of the command provided to the OS when invoking Gri.

Gri handles this by so-called "alias" synonyms, which store the names of other items.

The syntax is simple. Suppose that a synonym, called `\pointer` say, contains the **name of** another synonym, or a variable. Then you may use `\@pointer` anyplace you would normally use the item named.

Illustrations of using the value of a named item

The following prints an approximation to Pi followed by the name of movie star.

```
.pi. = 3.14
\pi_pointer = ".pi."
show \@pi_pointer  # just like 'show .pi.'

\hero = "Gregory Peck"
\our_hero = "\\hero"
show "\@our_hero"  # just like 'show "\hero"'
```

Illustrations of assigning to a named item

The following prints an approximation to 2*Pi and yet another star; the point is that the alias appears to the left of an assignment operator.

```
# Print approximation to 2*Pi
.pi. = 3.14
\pi_pointer = ".pi."
\@pi_pointer *= 2
show .pi.

# Stars don't shine alone
\hero = "Gregory Peck"
\our_hero = "\\hero"
\@our_hero = "Harrison Ford"
show "\hero"
```

10.5.5 Local synonyms

Local synonyms are created by Gri upon entry to a Gri command. You use them to parse the command line that was used in calling the new command, to look for options, gather filenames, etc. Local synonyms are known only from within the local Gri command. They are not listed by `show synonyms`, but they can be used freely in commands like `show "Number of words is \.words."`.

- Within any new Gri command, the number of words in the line that called the command is available in `\.words.`. The RPN operator `wordc` also yields the same value (Section 10.9.6 [Solitary Operators], page 168).
- The first word in the calling line is `\.word0.`, the second `\.word1.`, etc. (Note that this is the C convention, **not** the FORTRAN convention. If `\.words.` is 2, then `\.word0.` and `\.word1.` are defined, but `\.word2.`, which FORTRAN programmers expect, will not be defined.) If you don't know the place of the synonym in advance (i.e. 0 versus 1, for `\.word0.` versus `\.word1.`), then use the RPN operator `wordv` instead (Section 10.9.5 [Unary Operators], page 164).
- Within any new Gri command, the proper calling usage is available in `\.proper_usage.`. This is useful in tests of syntax (Section 10.11 [Adding New Commands], page 173). For example:

```
'draw depths from \file'
Draw depth data stored in indicated file.  If the
filename contains periods or slashes, you'll
have to enclose it in double quotes, as
in the second example:
  draw depths from file upper_cove
  draw depths from file ../old_data/upper_cove
{
  if {rpn \.words. 4 !=}
    show "FATAL ERROR in '\.proper_usage.':"
    show "  Need 4 words; got \.words. words."
    quit
  end if
  # Right number of words, so continue onward...
}
```

These synonyms help you scan for optional words in commands. Suppose you have defined a new command `New Thing [option]`. If you call it with `New Thing`, then (within `New Thing`) `\.words.` will be `"2"`, `\.word0.` will be `"New"` and `\.word1.` will be `"Thing"`. On the other hand, if you call it with `New Thing 22.3` then `\.words.` will be `3`, `\.word0.` will be `"New"`, `\.word1.` will be `"Thing"` as before, and `\.word2.` will be `"22.3"`.

EXAMPLE Here is a new command to label lines drawn by `draw curve`:

```
'Draw Label For Last Curve "label"'
Draw a label for the last curve drawn, using
..xlast.. and ..ylast.. built-in variables.
{
  new .draw_label_for_last_curve_graylevel.
  .draw_label_for_last_curve_graylevel. = ..graylevel..
  set graylevel 0
  draw label "\.word5." at \
      {rpn ..xlast.. xusertocm 0.1 + xcmtouser} \
```

```
        {rpn ..ylast.. yusertocm \
            ..fontsize.. pttocm 2 / -
            ycmtouser}
    set graylevel .draw_label_for_last_curve_graylevel.
    delete .draw_label_for_last_curve_graylevel.
  }
  open file.dat
  read columns x y
  draw curve
  \label = "Illustration"
  Draw Label For Last Curve "\label"
```

(Note that Gri has a built-in command `draw label for last curve "\label"` written much as above, so there is no need for you to enter this new command into your '`.grirc`' file. But you might want to check '`gri.cmd`' to see how a full command does checking of the calling syntax (Chapter 3 [Invoking Gri], page 7).

10.6 If Statements

Gri has `if` statements to make your programs more flexible. Here's an example:

```
query \thick "Use thick lines? (0 or 1)" ("0")
if \thick
  set line width 2
else
  set line width 0.5
end if
```

If you answer 1 to the question, the line thickness will be set at 2 points. If you answer 0 then a thin line will be used. If you press carriage return a thin line will be used.

The item following the `if` can be

- a number (1 means true; anything else means false)
- a variable (1 means true; anything else means false). Example:

  ```
  if .plot_contours.
    draw contour
  end if
  ```

- a synonym which expands to a number (1 means true; anything else means false). Example:

  ```
  \plot_contours = "1"
  if \plot_contours
    draw contour
  end if
  ```

  (Don't worry about the fact that synonyms are strings; Gri expands the string value before interpreting the `if` statement.)

- an expression of the form `{string1 == string2 }`. The symbol `==` is an operator which tests for string equality. This expands to `1` if the strings are equal, or `0` otherwise. The strings may be either synonyms or string constants. If the string constant contains only one word, then it is not necessary to enclose it in quotes, but it is clearer to do so. Examples:

  ```
  if {"\variable" == "Salinity"}
    set x name "Salinity"
  ```

```
else
  set x name "Unknown"
end if
```

- a rpn (reverse polish notation) expression (Section 10.9 [rpn Mathematics], page 161):

```
if {rpn .time. 100 <}
  # ie, (100 < time), not (time < 100)
  show "Time > 100"
else if {rpn .time. 100 >}
  show "Time < 100"
else if {rpn "\item" "later" ==}
  show "Time ... later babe"
else
  show "Time is equal to 100"
end if
if {rpn .time. 10 * 100 ==}
  show "Time is equal to 10"
else
  show "Time is not equal to 10"
end if
```

There is no need to put the else part in if you don't need it. You can do

```
set line width 0.5
if \use_thick_lines
  set line width 2
end if
```

if you wish.

If you want just the else part, you can do

```
if ! \use_thick_lines
  set line width 0.5
end if
```

(The exclamation point denotes logical negation: `! true` equals `false`.)

If statements may be nested many levels deep. You may also have `else if` blocks, as in:

```
if {"\variable" == "S"}
  set x name "Salinity"
  set x axis 32 33 0.5 .1
else if {"\variable" == "T"}
  set x name "Temperature"
  set x axis 15 20 1 0.5
else
  set x name "Unknown"
end if
```

10.7 Loops

Gri provides only one type of loop, the `while` loop, described elsewhere (Section 9.3.52 [While], page 141).

10.8 Mathematics

Gri lets you do some simple mathematical manipulations on your column and grid data.

10.8.1 Column data

The column operators are `=`, `+=`, `-=`, `*=`, `/=`, `^=` (exponentiation) and `_=` (logarithm). There must be spaces before and after the operators, but no space between the 2 letters of the operators. The operations may be applied not only to `x` and `y` as shown, but also to `z` (used to hold data to be contoured or written as symbols), and `u` and `v` (used to store vector fields).

The axis scales are **not** changed by mathematical operations on the columns, regardless of whether the scales were set manually or by Gri command (Section 4.2 [Axis Scaling], page 16).

Elements of columns are available by the `@` reverse polish operator (Section 10.9 [rpn Mathematics], page 161).

Examples:

- To multiply all the x data by 10, use `x *= 10`; to add 5 to each y-value, use `y += 5`.
- To set all the y data to 10, do `y = 10`. (This will only work if you've already read column data.)

See also Section 10.9.3 [Tertiary Operators], page 162 for a method of assigning or altering column data using the RPN operator.

10.8.2 Grid data

Various commands let you alter grid data as used in contouring (Chapter 6 [Contour Plots], page 25). Possible commands are as follows.

```
grid data = number
grid data += number
grid data -= number
grid data *= number
grid data /= number
grid data ^= number # take data to power 'number'
grid data _= number # take log base 'number'
grid x = number
grid x += number
#... others as in 'grid data'
grid y = number
grid y += number
#... others as in 'grid data'
```

10.8.2.1 Image data

Various commands let you alter image data (Chapter 7 [Images], page 31.). Possible commands are as follows.

```
image += number
image -= number
```

```
image *= number
image /= number
image ^= number # power
image _= number # logarithm
```

10.8.2.2 Image grayscale/colorscale

Various commands let you alter image data (Chapter 7 [Images], page 31). Possible commands are as follows.

```
image grayscale += number
image grayscale  -= number
image grayscale  *= number
image grayscale  /= number
image grayscale  ^= number # power
image grayscale  _= number # logarithm
image colorscale += number
image colorscale  -= number
image colorscale  *= number
image colorscale  /= number
image colorscale  ^= number # power
image colorscale  _= number # logarithm
```

10.8.2.3 Variables

Possible commands are:

```
.variable. = number
.variable. += number
.variable. -= number
.variable. *= number
.variable. /= number
.variable. ^= number # power
.variable. _= number # logarithm
```

10.9 Rpn (reverse-polish notation) Calculator

Gri can do simple mathematics on numbers. The syntax is reverse-polish notation (`rpn`), which is used in some calculators. Most users can learn rpn in a few minutes, so don't worry if you don't know RPN yet.

Syntax: rpn expressions can be used anywhere Gri expects a number. RPN expressions start with a opening curly brace (`{`) which is immediately followed by the word `rpn`. rpn expressions end with a closing curly brace (`}`). Instead of `set x size 10` you could write `set x size {rpn 20 2 /}`, where the expression `{rpn 20 2 /}` tells Gri to insert the number 20 onto a stack, then insert the number 2 above it on the stack, and then divide the top two items on the stack. The following are equivalent:

```
set x size {rpn 20 2 /}           # 10 = 20/2
set x size {rpn 30 2 / 5 -}       # 10 = (30/2-5)
set x size {rpn pi 3.1415 / 10 *} # 10 = 10*pi/pi
```

If an rpn expression contains a variable whose value is "missing", then the value of the result of the expression will also be missing (unless the value of the missing variable is thrown away with a "pop" operator). However, if a missing value just happens to occur as the result of an intermediate calculation, then the result is not considered to be missing.

RPN operations can be divided roughly into the following groups.

10.9.1 Stack Operators

Stack operators manipulate or display the stack.

`pop` removes the top item from the stack (Section 10.9.5 [Unary Operators], page 164).

`dup` duplicates the top item on the stack (Section 10.9.5 [Unary Operators], page 164).

`exch` reorders the top two items on the stack (Section 10.9.4 [Binary Operators], page 162).

`pstack` prints the items on the stack (without changing the stack).

`roll_right` rolls the items to the right.

`roll_left` rolls the items to the left.

For example, the following shows how you might use `exch` or `roll_right` to change the sense of a subtraction.

```
show {rpn 1 2            -} " ... yields -1"
show {rpn 1 2 exch       -} " ... yields  1"
show {rpn 1 2 roll_right -} " ... yields  1"
```

10.9.2 Rpn function Operators

`rpnfunction` operators are user-defined operators. The parser replaces any such operator with the user-defined rpn expression. The `rpnfunction` operators are both general and powerful. An `rpnfunction` may be composed of any legal primitive rpn constructs or even other legal `rpnfunction` constructs. For details, (Section 9.3.40 [Rpnfunction], page 114).

10.9.3 Tertiary Rpn Operators

`{rpn 0 4 "hello" substr }`
: Extract 4 characters from the indicated string, starting at character number 0 (i.e. the start of the string). In other words, replace the three items on the top of the stack with the single item `\"hell\"`.

10.9.4 Binary Operators

Binary operators act on the **top two** items on the stack. Most binary operators replace two items on the stack with one item, e.g. `{rpn 1 2 /}` yields 0.5. However, a few binary operators replace one pair of items with a new pair of items, e.g. the `xyusertocm` operator replaces an (x,y) pair in user coordinates with an (xcm,ycm) pair in coordinates of centimeters on the page.

The binary operators are illustrated below, in rough alphabetical order.

`{rpn 3 2 +}`
: Add 2 to 3.

`{rpn 3 2 -}`
: Subtract 2 from 3.

`{rpn 3 2 *}`
: Multiply 3 by 2.

`{rpn 3 2 /}`
: Divide 3 by 2.

`{rpn 3 2 <}`
: Test whether 2 is less than 3, yielding 1. Note: this convention may be confusing to users who are familiar with HP calculators from decades past. Present-day calculators use this convention, but possibly older calculators used the reverse convention, using `>` where Gri uses `<`.

`{rpn 3 2 <=}`
: Test whether 2 is less than or equal to 3.

`{rpn 3 2 >}`
: Test whether 2 is greater than 3, yielding 0.

`{rpn 3 2 >=}`
: Test whether 2 is greater than or equal to 3, yielding 0.

`{rpn 3 2 ==}`
: Test whether 2 and 3 are equal, yielding 0. (Do not confuse this with the asignment operator `=`, described next.)

`{rpn 10 ".ten." =}`
: Assign the value `10` to the variable named `.ten.`. The variable name must be put in quotes, or else Gri will insert the value of the variable (if it exists) into the expression, instead of trying to assign to it.

  After the assignment is done, the stack is cleared of both the value and the variable name. For example, in the following code

```
{rpn 3.1415 ".pi." =}
show .pi.
```

the first line evaluates to a blank line, and the second prints the value of Pi.
NOTE: Do not confuse this with the == equality operator described above.

`{rpn "hello" "\\greeting" =}`
Assign the value `"hello"` to the synonym `\greeting`. See notes at the above item.

`{rpn 3.14159 0 "x" =}`
Assign the value Pi to the first element (at index `0`) of the `x` column. All columns may be assigned to in this way, e.g. the following is a technique for plotting a quadratic function:

```
.i. = 0
.n. = 10
while {rpn .i. .n. >}
    {rpn .i. .n. 1 - /     .i. "x" =}
    {rpn x .i. @ 2 power  .i. "y" =}
    .i. += 1
end while
draw curve
```

`{rpn 0 1 &}`
Test whether 0 and 1 are both true, yielding 0.

`{rpn 0 1 and}`
Test whether 0 and 1 are both true, yielding 0.

`{rpn y x area}`
Calculate the area under the curve y=y(x). For details (Section 10.9.7 [Manipulation of Columns etc], page 169).

`{rpn 0 1 |}`
Test whether either 0 or 1 is true, yielding 1.

`{rpn 0 1 or}`
Test whether either 0 or 1 is true, yielding 1.

`{rpn 2 3 exch}`
Exchange 2 and 3 on the stack, yielding `3 2` on the stack. (See also `pop` and `dup`.)

`{rpn x 0 @}`
Yields the value of the first number in the x column. A similar form also works for `y`, etc. (Section 10.9.7 [Manipulation of Columns etc], page 169).

`{rpn 2 3 inf}`
Pick the smaller of two values, yielding 3. (Opposite to `sup`.)

`{rpn 2 3 power}`
Take 2 to the 3rd power, yielding 8. Note: This convention may be confusing to users who are familiar with HP calculators from decades past. Present-day calculators use this convention, which they write as `y^x`, but older calculators used the reverse convention, labelling the key `x^y`.

`{rpn 2 3 remainder}`
Calculate the remainder after dividing 2 by 3, yielding 2. The return value for `{rpn A B remainder}` is `B - n * A`, where `n` is the quotient of `A/B`, rounded towards zero to an integer. In this case, `2/3` rounds to an `n` value of zero, yielding 2 as the resulting remainder.

`{rpn "heLLo" "s/L/l/g" sed}`
: Switch all instances of `L` into `l`, yielding the string `"hello"` on the stack. This can be helpful for working with filenames, etc. The work is preformed with a system call to the `sed` utility (present on unix systems), and therefore this command will fail if `sed` is not installed, or if the OS cannot be contacted.

`{rpn "file" ".dat" strcat}`
: Concatenate the two strings, yielding the string `"file.dat"`.

`{rpn 2 3 sup}`
: Pick the larger of two values, yielding 3. (Opposite to `inf`.)

10.9.5 Unary Operators

Unary operators replace the last item on the stack with another item. For example, the `sin` operator takes the sine of the number on the top of the stack; e.g., `{rpn 45 sin}` yields the sine of 45 degrees.

The unary operators are illustrated below, in rough alphabetical order.

`{rpn 0 !}` Replace 0 (false) with its logical negation 1 (true).

`{rpn 0 not}`
: Replace 0 (false) with its logical negation 1 (true).

`{rpn -3 abs}`
: Calculate the absolute value of `-3`.

`{rpn 0.5 acos}`
: Calculate the inverse cosine of 0.5, yielding 60 (degrees).

`{rpn 2 acosh}`
: Calculate the inverse hyperbolic cosine of 2, yielding 1.317. (Note: argument must equal or exceed 1, or an error results.)

`{rpn "filename" age}`
: Calculate the "age" of the indicated file, in seconds. An age of zero indicates that the file was created, or last modified, within 1 second of the execution of the `age` operator.

  On unix (and similar) machines, the calculation is done on unix machines with the `stat()` subroutine. On other machines, the `age` operator may cause an error.

  The age of a non-existent file is reported as the number of seconds since the system clock's reference time, i.e. since 1970-jan-1 on unix machines. This convention is so that scripts like that in the example below will work as intended.

  A typical use of this command is the creation of data-files from shell scripts, as illustrated below. The idea is to update (or create) the file '`file.dat`' using the system-executable script '`creator.pl`', but only to do so if '`creator.pl`' is younger than '`file.dat`'.

```
if {rpn "file.dat" age "creator.pl" age <}
   system "./creator.pl > file.dat"
end if
open file.dat
```

  For the convenience in such usage, a non-existent file is assigned the maximum possible file age on the given OS, e.g. on a unix machine, the age is reported

as though the non-existent file had been created on January 1, 1970 on a unix machine.

`{rpn 0.5 asin}`
: Calculate the inverse sine of 0.5, yielding 30 (degrees).

`{rpn 1 atan}`
: Calculate the inverse tangent of 1, yielding 45 (degrees).

`{rpn 0.5 atanh}`
: Calculate the inverse hyperbolic tangent of 0.5, yielding 0.549306 (radians).

`{rpn 0 argv}`
: Returns the name of the Gri command-file, which is considered as the first "optional" argument. (It may seem odd that the name of the command-file is considered an option, but Gri does this for consistency with C and other languages. It is useful.) Other arguments provided when Gri was invoked are provided as `rpn 1 argv`, etc.

  A string consisting of a single blank character results if one tries to access beyond the list of arguments that were actually supplied. See also the `argc` solitary operator (Section 10.9.6 [Solitary Operators], page 168), which returns the number of optional arguments.

  For example, if Gri is invoked as

  ```
  gri myscript.gri file1.dat file2.dat
  ```

  and if '`myscript.gri`' contained

  ```
  .n. = {rpn argc}
  .i. = 0
  while {rpn .n. .i. <}
      show "argument " .i. " is " {rpn .i. argv}
      .i. += 1
  end while
  ```

  then the output would be

  ```
  argument 0 is myscript.gri
  argument 1 is file1.dat
  argument 2 is file2.dat
  ```

  For usage within the Emacs gri-mode, see Section 12.7 [Filename arguments when running gri], page 203.

`{rpn "hi" ascent}`
: Determine ascent of this string (in cm), in the present font and fontsize. (See also `descent` and `width`.)

`{rpn "3.1" atof}`
: Calculate the numerical value contained in indicated string.

`{rpn 1.5 ceil}`
: Calculate the next higher integer, yielding 2. (Opposite of `floor`.)

`{rpn 45 cos}`
: Calculate the cosine of 45 degrees, yielding 0.707.

`{rpn 1 cosh}`
: Calculate the hyperbolic cosine of 1 (radian), yielding 1.543.

`{rpn 1 cmtopt}`
: Convert from 1 centimeter to so-called "point" units, yielding 28.45. (Opposite of `pttocm`.)

`{rpn 170 dec2hex}`
: Convert a number into a string which is its hexadecimal representation. Before the conversion, the number is rounded to the nearest integer, and if the result is negative, an error results. The string is double-quoted, with letters (if there are any) being in upper case.

  For example `\hex = {rpn 63 dec2hex}` is equivalent to `\hex = "3F"`.

  Compare with `hex2dec`, the inverse.

`{rpn "\\syn" defined}`
: Test whether the synonym is defined at the moment, returning 1 if so and 0 if not. (Note the double-backslash in the synonym name, which is required.)

`{rpn ".var." defined}`
: Test whether the variable is defined at the moment, returning 1 if so and 0 if not.

`{rpn "\\@alias" defined}`
: Test whether the variable/synonym item that is named by the alias (Section 10.5.4 [Alias Synonyms], page 155) is defined at the moment, returning 1 if so and 0 if not.

`{rpn "hi" descent}`
: Calculate the descent (below the baseline in cm) for the given string, in the present font and fontsize. (See also `ascent` and `width`.)

`{rpn "/home/me/data/timeseries" directory_exists}`
: Determine whether indicate directory exists, yielding `1` if it does and `0` otherwise. (See also `file_exists`.)

`{rpn 2 dup}`
: Duplicate the top item on stack, yielding `2 2` on the stack. (See also `exch` and `pop`.)

`{rpn 1 exp}`
: Calculate the value of `e` raised to the indicated power, yielding 2.71828.

`{rpn 2 exp10}`
: Calculate the value of `10` raised to the indicated power, yielding 100.

`{rpn "foo.dat" file_exists}`
: Determine whether the indicate file exists, yielding `1` if it does and `0` otherwise. (See also `directory_exists`.)

`{rpn 1.5 floor}`
: Calculate the nearest smaller integer, yielding 1. (Opposite of `ceil`.)

`{rpn "AA" hex2dec}`
: Convert a string, representing a hexadecimal value, into an integer. The string must be double-quoted, and it may contain either lower- or upper-case letters; this is in contrast to the inverse function, `dec2hex`, which returns upper-case.

  This operator is most often used in working with colours, since Gri handles colour components in decimal terms, whereas many other applications refer to the components in hexadecimal notation.

  Compare with `dec2hex`, the inverse.

`{rpn 3 ismissing}`
: Yields 1 if the indicated value is a "missing value" or 0 otherwise.

`{rpn 100 log}`
: Calculate the base-10 logarithm of 100, yielding 2.

`{rpn 10 ln}`
: Calculate the natural logarithm of 10, yielding 2.30259.

`{rpn x mean}`
: Yields the mean value of the (non-missing) numbers in the x column. A similar form also works for `y`, etc. (Section 10.9.7 [Manipulation of Columns etc], page 169).

`{rpn x max}`
: Yields the largest value of the (non-missing) numbers in the x column. A similar form also works for `y`, etc. (Section 10.9.7 [Manipulation of Columns etc], page 169).

`{rpn x min}`
: Yields the smallest value of the (non-missing) numbers in the x column. A similar form also works for `y`, etc. (Section 10.9.7 [Manipulation of Columns etc], page 169).

`{rpn 28.45 pttocm}`
: Calculate the number of centimeters in 28.45 printers points, yielding 1. (Opposite of `cmtopt`.)

`{rpn 1 2 pop}`
: Remove the top item from the stack, yielding `1` on the stack. Generates an error if the stack is empty. (See also `exch` and `dup`.)

`{rpn 4 sqrt}`
: Calculate the square root of 4, yielding 2. (Negative arguments yield errors.)

`{rpn 45 sin}`
: Calculate the sine of 45 (degrees), yielding 0.707107.

`{rpn 2 sinh}`
: Calculate the hyperbolic sine of 2, yielding 3.62686.

`{rpn "hello" strlen}`
: Determine the number of characters in string, e.g. 6 here.

`{rpn "date" system}`
: Call the indicated system function and insert its ouput on the stack, yielding the date as a character string.

`{rpn 45 tan}`
: Calculate the tangent of 45 (degrees), yielding 1.

`{rpn tanh}`
: Calculate the hyperbolic tangent of 2, yielding 0.964028.

`{rpn "hi" width}`
: Determine width of this string (in cm), in the present font and fontsize. (See also `ascent` and `descent`.)

`{rpn 0 wordv}`
: Returns the first word used in invoking the present command. Similar to the `\.word0.` synonym (Section 10.5.5 [Local Synonyms], page 156). Example:

```
'let us test .it.'
{
  .w. = 0
  while {rpn .w. wordc >}
    show "The " .w. "-th word is '" {rpn .w. wordv} "'."
    .w. += 1
  end while
}
let us test "this thing"
let us test "this" "thing"
let us test "Pi is" {rpn 3.14}
```

If you are using this to parse options given to the command, it is up to you to skip the non-optional words in the command. In this case, for example, we skipped the first three words (`let`, `us`, and `test`).

`{rpn 1 xusertocm}`
: Calculate the x coordinate, in centimeters measured from left-hand side of page, corresponding to a user-value of x=1. (Opposite of `xcmtouser`.)

`{rpn 1 xcmtouser}`
: Calculate the x value, in user units, for a point that is 1 centimeter from the left-hand edge of the paper. (Opposite of `xusertocm`.)

`{rpn 1 yusertocm}`
: Calculate the y coordinate, in centimeters measured from bottom side of page, corresponding to a user-value of x=1. (Opposite of `ycmtouser`.)

`{rpn 1 ycmtouser}`
: Calculate the y value, in user units, for a point that is 1 centimeter from the bottom edge of the paper. (Opposite of `yusertocm`.)

10.9.6 Solitary Operators

Solitary operators do not act on items on the stack; rather, they generate items themselves and insert them on the stack.

The solitary operators are illustrated below, in alphabetical order.

`{rpn argc}`
: Yields number of command-line arguments given by the user when Gri was invoked. Thus, invoking Gri as

  ```
  gri myfile.gri file1.dat file2.dat
  ```

  yields 3, for arguments '`myfile.gri`', `file1.dat`, and `file2.dat`. These arguments are accessible through the `argv` unary operator (Section 10.9.5 [Unary Operators], page 164).

`{rpn e}` Yields the base of natural logarithms, i.e. `2.718`...

`{rpn pi}` Yields Pi, i.e. `3.141`...

`{rpn rand}`
: Generate a random number in the range 0 to 1, using the C subroutine `drand48()` if it is available, otherwise the less well-distributed `rand()` subroutine.

`{rpn wordc}`

Returns number of words used in invoking the present command. Similar to the `\.words.` synonym (Section 10.5.5 [Local Synonyms], page 156). Example:

```
'let us test .it.'
{
  show "This command has " {rpn wordc} " words"
}
let us test 10
let us test {rpn 3 1 +}
let us test "this"
let us test "this thing"
```

The operator `wordv` may be used to extract the words of the command (Section 10.9.5 [Unary Operators], page 164).

10.9.7 Manipulation of Columns etc

10.9.7.1 Columns

Individual data in the `x`, `y`, `z`, `u`, `v` and `weight` columns can be accessed with the `@` operator. The first point has index 0. Examples:

```
show "first x is " {rpn x 0 @ }
show "last  x is " {rpn x ..num_col_data.. 1 - @ }
show "and here are all the data:"
.i. = 0
while {rpn .i. ..num_col_data.. >}
    show {rpn x .i. @ }
    .i. += 1
end while
```

The mean value is available from the `mean` operator (e.g., `.xmean. = {rpn x mean }`, while the standard deviation is given by `stddev`, the skewness is given by `skewness`, and the kurtosis is given by `kurtosis` (using the definition that yields 3 for a gaussian distribution).

The minimal and maximal values are given by `min` and `max`.

The area under the curve y=y(x) is found by `{rpn y x area }`, defined by $\frac{1}{2\left(\sum_1^{n-1}(y_i+y_{i-1})(x_i-x_{i-1})\right)}$ for data with i ranging 0–*..num_col_data..* − 1

10.9.7.2 Grid

Grid data can be accessed with e.g. `{rpn grid min }` , `{rpn grid max }` , and `{rpn grid mean }` .

The value of the grid at a given `(.x.,.y.)` coordinate may be found by by e.g. `{rpn grid .x. .y. interpolate}`. The interpolation scheme is the same as that used in converting grids to images.

10.9.8 rpn Examples

Here are some reverse-polish expressions and the corresponding algebraic interpretations:

- `{rpn 1 2 + 10 / }` = (1+2)/10
- `{rpn .a. .b. + .c. + .d. / }` = (.a.+.b.+.c.)/.d.
- `{rpn e 2 / }` = e/2 (Gri knows values of "e" and "pi")
- `{rpn 23 sin 100 * 12 cos + }` = cos(12) + 100sin(23)
- `{rpn 5 2 power }` = 25
- `{rpn 2 log exp }` = exp(log 2)
- `{rpn 2 ln exp10 }` = 10^ln2
- `{rpn 1.7 floor }` = 1 (rounds down to nearest integer. Note that the floor of -1.7 is -2)
- `{rpn 10.1 2 remainder }` = 0.1 (remainder of 10.1 after division by 2; see C function `remainder(x,y)`)
- `{rpn -10.1 2 remainder }` = -0.1
- `{rpn -10.1 -2 remainder }` = -0.1
- `{rpn .num. 10 > }` = 1 if 10 exceeds .num., or 0 otherwise

NOTES:

- The units of `sin`, `cos`, etc, are degrees, not radians.
- The scales of the plot are accessible to `rpn`. For example, with the command

  ```
  draw label "hi" at 10 20
  ```

  you draw the indicated string at the indicated location in user coordinates. To put it 0.15 centimetres to the right of this location and 0.1 centimetres lower, you could do as follows:

  ```
  draw label "\label" at \
      {rpn .x. xusertocm 0.15 + xcmtouser} \
      {rpn .y. yusertocm 0.10 - ycmtouser}
  ```

  (Note that the x and y scales have individual translations from "user" to "cm" coordinates.)
- Some conversion factors are built into `rpn`; `cmtopt` converts from centimetres to points (by dividing by 28.45; the conversion factor to inches is 72.27) while `pttocm` converts from points to centimetres. For example, here is how to label a data curve with a label placed near the last y-value of the data set:

  ```
  draw curve
  .y. = {rpn ..ylast.. yusertocm 0.5 - ycmtouser}
  draw label "Smoothed" at ..xlast.. .y.
  ```

10.10 Text Strings

Any text can be drawn in any size; Gri does not limit font size to a list, e.g. 10 point, 12 points, etc. Several fonts are available in Gri, e.g. Times, Helvetica, etc.; these are all standard PostScript fonts. Support for some non-English languages (e.g. French) is also provided. And, finally, Gri supports inclusion of simple mathematical expressions (Greek letters, superscripts, etc.) in text, using a LaTeX-style syntax.

10.10.1 Embedding synonyms in quoted text strings

Outside math strings, you can embed your synonyms at will. For example, you can include the name of a data file in the title of your plot as follows

```
query \filename "File to read from?" ("data.file")
open \filename
read columns x y
draw curve
draw title "data from \filename"
```

Within math strings (ie, between matched dollar-signs), these synonyms are disabled, and only the mathematical symbols and Greek letters work.

10.10.2 Mathematical text

10.10.2.1 Subscripts

As in TEX and LaTEX, you must be in math-mode to use subscripts; in other words, you must enclose the string or substring in dollar-signs. For single-character subscripts, insert an underline prior to the character to be subscripted:

```
draw title "$a_2$"
```

For multiple-character subscripts, insert braces before and after the item to be subscripted:

```
draw title "$a_{22}$"
```

10.10.2.2 Superscripts

As in TEX and LaTEX, you must be in math-mode to use superscripts; in other words, you must enclose the string or substring in dollar-signs. For single-character superscripts, insert a carat prior to the character to be superscripted:

```
draw title "$a^2$"
```

For multiple-character superscripts, insert braces before and after the item to be superscripted:

```
draw title "$a^{22}$"
```

10.10.2.3 Mathematical symbols

As in TEX and LaTeX, you indicate mathematical symbols and Greek letters with backslash sequences. The following LaTeX symbols are defined in math mode in Gri (cf tables in Lamport's section 3):

(\Delta, Δ), (\Downarrow, $\Downarrow$), (\Gamma, Γ), (\Im, $\Im$), (\Lambda, Λ), (\Leftarrow, $\Leftarrow$), (\Leftrightarrow, $\Leftrightarrow$), (\Omega, Ω), (\Phi, Φ), (\Pi, Π), (\Psi, Ψ), (\Re, $\Re$), (\Rightarrow, $\Rightarrow$), (\Sigma, Σ), (\Theta, Θ), (\Uparrow, $\Uparrow$), (\Upsilon, Υ), (\Xi, Ξ), (\alpha, α), (\approx, $\approx$), (\ast, $*$), (\beta, β), (\bullet, $\bullet$), (\chi, χ), (\circ, $\circ$), (\cong, $\cong$), (\delta, δ), (\div, $\div$), (\downarrow, $\downarrow$), (\epsilon, ϵ), (\equiv, $\equiv$), (\eta, η), (\exists, $\exists$), (\forall, $\forall$), (\gamma, γ), (\geq, $\geq$), (\gg, $\gg$), (\in, $\in$), (\int, $\int$), (\infty, ∞), (\iota, ι), (\kappa, κ), (\lambda, λ), (\langle, $\langle$), (\leftarrow, $\leftarrow$), (\leftrightarrow, $\leftrightarrow$), (\leq, $\leq$), (\ll, $\ll$), (\mu, μ), (\nabla, ∇), (\neq, $\neq$), (\nu, ν), (\omega, ω), (\partial, ∂), (\phi, ϕ), (\pi, π), (\pm, $\pm$), (\prod, $\prod$), (\propto, $\propto$), (\psi, ψ), (\rangle, $\rangle$), (\rho, ρ), (\rightarrow, $\rightarrow$), (\sigma, σ), (\sim, $\sim$), (\subset, $\subset$), (\subseteq, $\subseteq$), (\sum, $\sum$), (\supset, $\supset$), (\supseteq, $\supseteq$), (\surd, $\surd$), (\sqrt, $\surd$), (\tau, τ), (\theta, θ), (\times, $\times$), (\uparrow, $\uparrow$), (\upsilon, υ), (\varpi, ϖ), (\wedge, $\wedge$), (\xi,

ξ), (\zeta, ζ), (\vartheta, ϑ), (\varsigma, ς), (\varphi, φ), (\aleph, $\aleph$), (\oplus, $\oplus$), (\otimes, $\otimes$), (\wp, $\wp$), (\prime, $\prime$), (\emptyset, $\emptyset$), (\angle, $\angle$), (\neg, $\neg$), (\clubsuit, $\clubsuit$), (\diamondsuit, $\diamondsuit$), (\spadesuit, $\spadesuit$) (\cdot, $\cdot$) (\lfloor, $\lfloor$) (\lceil, $\lceil$) (\rceil, $\rceil$) (\rfloor, $\rfloor$)

For example, you might use these as follows:

```
draw title "$\alpha$ = thermal expansion coefficient"
```

Sometimes you'll want a mathematical symbol to be adjacent to a normal text string, with no space between. You can do this by enclosing in braces, as in LaTeX.

TEX and LaTEX handle combinations of superscripts and subscripts very cleanly, putting one above the other. Presently, Gri does not do this; for example `set x name "$A_1^2$"` will have the 2 appearing to the right of the 1 instead of above it. Proper positioning will be added to a later version of Gri, but in the meantime you can achieve the desired effect with the TEX "negative thinspace" psuedo-character in math-mode. Using this feature will not hurt you when the new Gri becomes available. The symbol for a negative thinspace is `\!` in math-mode. It has no meaning in nonmath mode. A thinspace is 1/6 of an "em-space" (a TEX term, normally equal to the width of the character "M" in the current font). In most fonts, numbers are half the width of the letter "M", so that 3 negative thinspaces will move leftward over a single number. Thus, if the example above becomes `set x name "$A_1\!\!\!^2"`, the 2 will be positioned above the 1. (Equivalently, you could write `set x name "$A^2\!\!_1$"`.) Depending on the actual characters you have in the super/subscripts, you might need more or less thinspaces; some experimentation might be required. Also, note that the symbol `\,` in math mode is a positive thinspace (which moves the next character a little bit to the right). Thus, you can add a little extra spaces between characters by doing something like `set x name "A$\,$B"`.

To get a hat over a single character, do something like the following (which draws a hat over the character "h"):

```
draw label "h${\!\!\!^{^\wedge}}$" at 10 12 cm
```

To get an overbar on a rho, do this:

```
draw label "$\rho\!\!\!\!^-$" at 3 3 cm
```

10.10.3 Non-English characters

Gri relies on the "standard" PostScript fonts, however, and it suffers all limitations of these fonts.

Gri supports both English and some other European-derived languages, permitting text with accents on letters. (It does not support Oriental or other languages at this time.) The accents are supported by using the so-called ISO-Latin-1 font-encoding scheme (also called the ISO-8859-1 scheme), and so, from what the author can gather from his reading, Gri should support various languages from western European, e.g. English, French, Spanish, Catalan, Basque, Portuguese, Italian, Albanian, Rhaeto-Romanic, Dutch, German, Danish, Swedish, Norwegian, Finnish, Faroese, Icelandic, Irish, Scottish, and as well as Afrikaans and Swahili.

Gri uses the ISO-Latin-1 font encodings by default, although the so-called 'standard' font-encoding may also be selected with the `Set Font Encoding` command (Section 9.3.41.18 [Set Font Encoding], page 120). For more on font encodings see any book on PostScript fonts ... although the bottom line is that if you are using accented characters in your work, then you probably already know about encodings, and if you don't use accents then you needn't learn about this topic except for the pleasure of learning about other languages.

The method of handling accented characters is very simple. If you can type it, Gri can draw it! It is up to you to determine how to enter the accents. Most text editors permit this. Since many users will prefer the Emacs editor, a few words about that are in order.

For complete information about entering iso-latin-1 characters in Emacs, consult your Emacs manual in the section which describes the available methods suitable for the Emacs version you are using. A few examples are nevertheless provided below.

Consider the task of inserting French text, with the Emacs text-editor. There are several ways of doing this (and you may wish to consult your emacs info manual). A method that works in emacs-19 up to current emacs-20 versions uses the emacs '`iso-transl.el`' package by putting the following in your '`~/.emacs`' file:

```
(require 'iso-transl)
(iso-transl-set-language "French")
(standard-display-european t)
```

Loading the iso-transl package defines three ways of entering the non-ASCII printable characters with codes above 127: the prefix *C-x 8*, or the ⟨Alt⟩ key, or a dead accent key. For example, you can enter uppercase A-umlaut as *C-x 8 " A* or *Alt-" A* (if you have an Alt key) or *umlaut A* (if you have an umlaut/diaeresis key).

A more recently introduced method is to enter the mode which allows quick insertion of iso-latin-1 characters. Do the Emacs command `M-x iso-accents-mode` (either manually, or in a hook that's done automatically). Now, suppose the x-axis is to represent temperature. All you'd have to do is type in the command

```
set x name "Temp'erature"
```

As you type, the quote mark will dissappear, and reappear as an accent on the `e`. And then, Gri will recognize this accented , and it will draw the accent on the axis label.

Perhaps the future default way of accomplishing this task is to use MULE support directly. First, customize MULE using `M-x customize-group RET mule` setting the `current language environment` (e.g. latin-1) and the `default input method` (e.g. latin-1-prefix). Then, invoking `M-x toggle-input-method` (e.g. ⟨C-\⟩) toggles into a mode similar to the `iso-accents-mode` minor-mode described above.

10.10.4 Adjustment Of Character Position

Micro-positioning is available within math-mode, via the symbols `\!` (which means go left one thin-space) and `\,` (which means go right one thin-space). (A thin-space is 1/6 the width of the letter "M").

10.11 Adding new commands to Gri

Gri provides so-called "newcommands" as a sort of subroutine syntax on steroids.

10.11.1 Purpose of newcommands

Gri can be extended easily. Primitive commands (e.g. `set x name`) can be supplemented with so-called "new commands." New commands are a little like subroutines other programming languages. For example, you might find that you often draw filled curves with a particular graylevel (say 0.5), and then return the graylevel to the previous value. This requires you to do the following each time:

```
new .old_graylevel.
.old_graylevel. = ..graylevel..
set graylevel 0.5
draw curve filled to 0 y
set graylevel .old_graylevel.
delete .old_graylevel.
```

This gets a bit tedious, and it would obviously be nicer to just say something like

```
Draw my kinda curve
```

To make this shortcut, you'd tell Gri about the existence of a new command called `Draw my kinda curve`, and tell it that the new command can be accomplished by the longer code fragment written above.

Once you've learned how to make new commands, you are likely to use them a lot. The following explains how you add new commands. For advice on programming style, etc., (Section 10.17 [Resource File], page 185).

10.11.2 How Gri parses commands

Whenever Gri reads a command line, it compares it with its list of commands. This list is searched in this order: (1) the universal '`gri.cmd`' file (Chapter 3 [Invoking Gri], page 7), (2) your resource file (Section 10.17 [Resource File], page 185), if it exists, and then (3) your command file itself. Gri stops searching when it finds a Gri command that matches the command line. "Matching" means that the command line is identical in all words in a Gri command, scanning from the left, until it encounters a word containing

- A quote (e.g. `"string"`)
- A synonym name (e.g. `\file`)
- A variable name (e.g. `.number.`)
- An opening square bracket (e.g. `[option]`)
- An opening brace (e.g. `{a|b}`)
- A choice between two items (e.g. `first|second`)
- A variable-name with a `&` character immediately to the left (e.g. `&.var.`). This is a signal that the variable may be changed inside the newcommand (Section 10.11.5.1 [The Ampersand Syntax], page 177).
- A synonym-name with a `&` character immediately to the left (e.g. `&\syn`). This is a signal that the synonym may be changed inside the newcommand (Section 10.11.5.1 [The Ampersand Syntax], page 177).

When Gri finds a command that matches your command line, it assumes that this is the intended command, and searches no further. This means that you must be careful not to have your command hidden by other commands. For example, if your resource file contained these lines, Gri would **never** execute the second new command, because calls to it match the first command. To avoid this, you may either reverse the order of the definitions, so that Gri will find the proper routine, or rename one of the routines.

```
'Draw foo'
Draw a foo.
{
  show "drawing a foo"
}
'Draw foo bar'
```

```
Draw a foo bar.
{
  show "drawing a foo bar"
}
```

Gri searches the '`gri.cmd`' file first, so any new command that you create that clashes with built-in commands will be ignored by Gri (Chapter 3 [Invoking Gri], page 7). Gri will warn you of this, and proceed, ignoring your newer definition. To get around this, you can use capital letters to begin the words of your new command. By convention, Gri never uses capital letters in this way, so a clash is impossible (except with any similar command you might have defined previously, such as in your '`~/.grirc`' file).

10.11.3 Simple example of a new command

To make a new command called `Show The Time` insert the following into your '`~/.grirc`' resource file or into your command-file somewhere near the top, or at least before you use the command.

```
'Show The Time'
New command to show the time of day.
{
  show "\.time."
}
```

EXPLANATION:

- The name of the new command is enclosed in angled single-quote marks. The words of the new command should begin with upper-case letters to prevent a name clash with a present or future built-in Gri command. **Formatting convention:** Make sure that the entire definition string, from the opening angled quote to the ending angled quote, appears on one line. Otherwise Gri will give an error like

  ```
  ERROR: Can't extract syntax for new command
  ```

- Following the name line, you may optionally insert any number of lines which will become the `help` information for the new command. See the file '`gri.cmd`' for the recommended stylistic conventions in writing help information (Chapter 3 [Invoking Gri], page 7). If your `help` text includes an example that uses an opening brace (`{`) you must escape the brace with a backslash, e.g. (`\{`).

- Following the help lines, if they exist, the body of the new command is given, sandwiched between a starting line containing an opening brace (`{`) as the **only** nonwhite character, and an ending line containing a closing brace (`}`) as the only nonwhite character. Any valid Gri commands may be used in the body of the new command. It is acceptable to use other new commands in the body. Recursion is also allowed – a new command is allowed to call itself (although there is a limit on nesting, of perhaps a thousand or so; this varies with the version of Gri). **Formatting convention:** It is usual, but not necessary, to use an indentation level of 2 spaces in the body of the new command.

The new command is invoked by `Show The Time`. Help for the command is found by `help Show The Time` or `help Show`.

10.11.4 Complicated example of a new command

The following example from the global 'gri.cmd' file illustrates how to parse/check the commandline (Section 10.5.5 [Local Synonyms], page 156), which is a good practice in any code you expect to re-use. The first `if` statement checks that the word `at` is in the right place (this would not have been checked by the syntax matcher, the word having followed a string). The presence of the keyword `cm` is checked for, and user units or cm units are used accordingly. Local variables are created (`new`) and then destroyed (`delete`) so that this new command cannot affect outside code.

```
'draw label whiteunder "\string" at .xleft. .ybottom. [cm]'
Draw label for plot, located with lower-left corner
at indicated (x,y) position (specified in user
units or in cm on the page).  Whiteout is used
to clean up the area under the label.  BUGS:
Cannot handle angled text; doesn't check for
super/subscripts.
{
    if {rpn "\.word4." "at" !=}
      show "ERROR: 5th word must be 'at', not '\.word4.'"
      show traceback
      quit
    end if
    new .x. .y. .oldgray. .space.
    if {rpn \.words. 7 ==}
      .x. = {rpn \.word5. xusertocm}
      .y. = {rpn \.word6. yusertocm}
    else if {rpn \.words. 8 ==}
      if {rpn "\.word7." "cm" !=}
        show "ERROR: Require 8th word to be 'cm'"
        show traceback
        quit
      end if
      .x. = \.word5.
      .y. = \.word6.
    else
      show "ERROR: Require 7 or 8 words, not \.words."
      show traceback
      quit
    end if
    # Coordinates now in cm.  Next, white out a box
    # under the text (and .space. centimetres
    # beyond text), then draw label.
    .space. = 0.1               # Space of 1mm
    .oldgray. = ..graylevel..
    set graylevel white
    draw box filled                          \
      {rpn .x. .space. -}                    \
      {rpn .y. .space. -}                    \
      {rpn .x. "\.word3." width + .space. +} \
      {rpn .y. "M" ascent + .space. + } cm
```

```
    set graylevel .oldgray.
    draw label "\.word3." at .x. .y. cm
    delete .x. .y. .oldgray. .space.
}
```

10.11.5 Altering command arguments – the & syntax

The Gri language permits a newcommand to change variables and synonyms passed as arguments, using a syntax that is quite similar to that employed by the C++ language.

10.11.5.1 Overview of the & syntax

Normally the arguments to a newcommand are parsed into either numerical values or strings, before execution is passed into the newcommand. This is a akin to the scheme called "call by value" in some programming languages. Gri also provides a syntax, borrowed from C++, that permits a newcommand to alter the contents of variable or synonym arguments.

The technique is simple. To permit a newcommand to modify an argument that is a variable or a synonym, just put a `&` to the left of the item on the calling line. Then, within the newcommand, the corresponding local synonym (i.e. `\.word1.`, etc.) will behave as though it were the instance of the original variable or synonym.

The `&` is placed to the left of the variable-name or synonym-name without intervening space. For example `foo &.var. &\syn` tells the parser that the newcommand named `foo` may possibly alter the values of the variable `.var.` and the synonym `\syn`, as they exist in the calling context.

It is important to note that Gri pays very little attention to the `&` in a syntax-declaration line. All it does is to note that the item to the right of the `&` is not a fixed word in the newcommand being defined; this follows the usual rules for parsing newcommand syntax (Section 10.11.2 [Parsing], page 174).

10.11.5.2 Example: doubling a variable

Consider the task of adding a fixed amount to a variable. If the variable we wish to double is `.x.`, we might write

```
'double_a_particular_variable'
{
    .x. = {rpn .x. 2 *}
}
.x. = 10
double_a_particular_variable
```

Code such as that presented above occurs in many applications. (Turn the multiplication into an addition, and change `.x.` to `..ymargin..`, and you'll start to see the core of an application that draws multiple graph panels, one above another.) However, the code is too specific to be of much general use!

What if we want to double some other variable instead? The code below shows how to do that.

```
'double &.value.'
{
    \.word1. = {rpn \.word1. 2 *}      # line 3
```

```
}
.x. = 10                                    # line 5
double &.x.
.y. = 3.14
double &.y.
```

At line 3 Gri interprets the `\.word1.` to the **left** of the equals sign as a reference to the variable that is set to the value 10 in line 5. Similarly, the `\.word1.` to the **right** of the equals sign evaluates to 10, the value in the calling program.

Gri automatically determines whether an item is a variable or a synonym, and does the correct thing. Thus, for example, if line 3 above were written as

```
\.word1. = "hello"                    # ERROR
```

an error would be reported, since `double` was called with a variable as an argument, and variables cannot hold strings.

10.11.5.3 Example: manipulating a synonym

Synonyms are treated in the same way, as is illustrated in the following example.

Q: what does the following print?

```
'add_a_dat &\filename'
{
    \.word1. = {rpn "\.word1." ".dat" strcat}
}
\filename = "test"
add_a_dat &\filename
show "\filename"
```

A: it prints `test.dat`.

10.11.5.4 Nesting

One newcommand may call another, letting a deeply-nested newcommand alter values of synonyms and variables that may otherwise be hidden behind `new` items of the same name. This is done by using the `&` notation at each step. The following provides an example of passing a variable through two levels of newcommands.

Q: what does the following print?

```
'food critic &food'
{
    \.word2. = "\.word2.s"
    yummy &\.word2.
}
'yummy &foods'
{
    \.word1. = "\.word1. are tasty"
}
\a = "apple"
food critic &\a
show "\a"
```

A: it prints `apples are tasty`.

10.11.5.5 About `new` and `delete`

If `new` and `delete` are executed on local synonyms inside newcommands (e.g. `new \.word1.`) they create and destroy variables and synonyms in the context of the **calling program**.

For example, consider the following.

Q: what does the following print?

```
'poetry &\s'
{
    new \s                              # line 3
    \s = "rose"
    \.word1. = "\.word1. is a \s"       # line 5
    delete \s                           # line 6
}
\s = "A rose "                          # line 8
poetry &\s
show "\s"
```

A: it prints `A rose is a rose`.

The key point here is that the instance of the synonym named `\s` in the calling program, set in line 8, is modified by the `poetry` newcommand, in line 6. This modification involves the use of a synonym, **also named** `\s`, that `"lives"` wholly within the newcommand, being created in line 3 and destroyed in line 6.

10.11.5.6 Determining calling information

Newcommands may determine the name and the nesting level of changeable calling arguments. To get the name, put a single ampersand after the backslash of the local synonym of interest. To get the nesting level (0 for main program, etc.) put two ampersands after the backslash.

```
'NC &.var.'
{
    show "\&.word1. (expect '.a.')"
    show "\&&.word1. (expect 0)"
}
.a. = 1
NC &.a.
```

Note: neither of these items may be used an lvalue. That is, they may not be used to the left of an equals sign. But you can always get around that by clever use of alias synonyms (Section 10.5.4 [Alias Synonyms], page 155).

10.11.5.7 How Gri implements the `&` syntax

When the parser encounters an unquoted `&` followed immediately by the name of a variable or a synonym, it converts the whole token (`&` plus name) into a specially-encoded string that can be recognized inside newcommands. (This is **only** done if the `&` and the variable name are **not** enclosed in double quotes.)

This specially-encoded string contains not just the name of the variable or synonym, but also the current level of nesting of newcommands. In this way a newcommand can have

its own private versions of variables, created by `new`, that won't be misinterpreted for the identically-named variables in the calling program.

The format of these specially-encoded strings is `#\bn\ba\bm\be\b:\bN \b_ \bl\be\bv\be\bl\b:\bL#\b`, where `N` stands for the name of the variable/synonym, `L` stands for the current level, and `\b` is the backspace character (hexadecimal 08 in the ascii table). This string is designed to be strange enough that users are unlikely to use it themselves. The coding scheme is not entirely arbitrary, however; note that if the backspace characters are ignored the result has the form `name:N_level:L`. It's also worth noting that if this string were printed on a terminal that erased characters when typing backspaces the result would be of the form `N_L`.

Examples: `.a.` in the main program (i.e. at level 0) encodes to `#\bn\ba\bm\be\b:\b.a. \b_ \bl\be\bv\be\bl\b:\b0#\b` and `&\my_syn` inside a newcommand called by the main program (i.e. at level 1) encodes to `#\bn\ba\bm\be\b:\b\my_syn \b_ \bl\be\bv\be\bl\b:\b1#\b`.

Inside a newcommand, Gri checks to see if builtin synonyms (e.g. `\.word1.`) hold such specially-encoded strings. If so, then the appropriate versions of the variables are used in preference to any variables that may have been newly created by `new` inside the newcommand.

10.12 Hints for Gri Programming

Here are some hints for good Gri programs:

- Whenever working with grids (for contouring) or images, make use of the `show grid` or `show image` commands. They will give you useful information about the statistics (min/max/histogram) of the items.
- Use the operating system, not Gri, to manipulate your data. For example, if you have a file whose first column is x times 100, and third is the arcsin of y, you could do:

  ```
  open "gawk '{print $1/100, sin($3)}' |"
  read columns x y
  ```

  If you have x and y in a non-decimal geographical format (e.g. hour.minute-second format), use the operating system to convert for you (Section 9.3.28 [Open], page 102).
- Use the `pstack` operator liberally in your rpn expressions to see what is going on (Section 10.9 [rpn Mathematics], page 161).
- While developing programs, put a `show columns statistics` command after every `read column` command, to check that the data have been read correctly.
- Development time can be minimized by limiting the number of data being processed. For example, in a multi-panel plot, it is often necessary to try various alternatives before aesthetic scales and page layout is achieved. The process can be speeded up by limiting the number of data being processed, as shown below. (If Gri finds fewer data in the file than specified, it will simply use the data that it found; so when the program works, just change `.n.` into something large.)

  ```
  .n. = 100 # 10000 for later
  ...
  # Panel 1
  read columns .n. x y
  ...
  # Panel 2
  read columns .n. x y
  ```

```
...
```

- Create new commands to do repetitive work.
- Use `draw time stamp` on all plots except for publication versions:

```
if !..publication..
  draw time stamp
end if
```

- For multiple panels on one page, do `delete x scale` or `delete y scale` before each new panel, so you will start afresh. Clearly identify code for particular panels with comments. This reduces errors you might get if you shuffle things later.
- Use the `..num_col_data..` built-in variable to see how many data have passed the `set input data window` data window in the last `read columns` command. The following example shows how to avoid drawing an unwanted curve:

```
open \f
read columns x y
close
if ..num_col_data..
  draw curve
  draw label "\f" at \
      {rpn ..xlast.. xusertocm 0.5} \
      {rpn ..ylast.. yusertocm 0.2} cm
end if
```

- Use synonyms and `query` for filenames. This makes programs much more flexible. Note that you can string synonyms together:

```
\dir = "~/EOS/iso0/"
query \file "Give file in directory \dir" ("1.dat")
open \dir/\file
```

  It is also a good idea to give a restrictive list of possibilities in your `query` command, to avoid complicated `if` commands later (Section 9.3.31 [Query], page 106).
- Use multiple `draw title` commands:

```
draw title "Atlantic water entering Arctic Ocean"
draw title "\.command_file. \.time."
```

- Use the `query` command to interact with the user (Section 9.3.31 [Query], page 106). The results can be stored in variables and synonyms.

10.13 Debugging Gri Programs

Here are some hints for debugging Gri programs:

- If no data appear on an xy plot, insert `show columns statistics` or `show columns` after the `read columns` command. It may be that you have fixed your axes, and that the axes frame does not include the data.
- If you get an error message, rerun your Gri program using the `-trace` command-line option, to see which line is causing the problem. This often reveals logic errors (e.g. in (`if` statements). You may also turn tracing on or off at any point in your Gri program by setting the built-in variable `..trace..` to 1 or 0. Many Gri users have the Gri command aliased to be in trace mode by default.
- If Gri complains of a syntax error, consult the printed manual, one of the online manuals, or the online help facility (Section 10.2 [Online Help], page 146).

- Check the version number (printed at startup) to see if a new version of Gri has been installed, and check the manual for known incompatabilities.
- Sprinkle `show` commands throughout your program, to see what's happening. Even when you are sure your program works, it is a good idea to embed `show` statements so are they executed if the `-debug` flag is set:

```
if ..debug..
  show "X=" .x. "and label is `\label'"
end if
```

- If your `draw` commands don't draw anything, check to see whether you've fooled yourself by enforcing an improper scaling; remove explict scaling (`set x axis ...`), clipping (`set clip`), data selection windows (`set input data window x|y`) and missing values (`set missing value`). Another trick is to read only a portion of the data set (`read columns 10 x y`) and then print out all the values (`show columns`).

If you determine that the bug is in Gri, not in your program, please report the bug, so that other users will not have the same hassle; (Section 15.2 [Reporting Bugs], page 227).

10.14 Error Messages

Gri error messages are in three types:

1. Operating system error messages, such as `segmentation fault`. These should never appear, and indicate a bug in Gri. Please report these to the author (Section 15.2 [Reporting Bugs], page 227).
2. Internal Gri error messages. The message starts with the words `FATAL error`, and quotes a file number and a line number, e.g.

   ```
   FATAL error: startup.c:199: ...
   ```

   Such errors indicate either a deficiency in your computer (e.g. insufficient storage space) or an internal bug in Gri. If the message does not indicate running out of storage, please report the error to the author (Section 15.2 [Reporting Bugs], page 227).

   For fatal error messages on a unix system, Gri dumps core, unless you have turned that feature off, with the `ulimit -c 0` unix command in a startup file. This creates a file called '`core`', which can help you in diagnosing the Gri bug. If you have the `gdb` debugger, just type `gdb gri core` and then type `where` to get a traceback stack. Please email this with your other information about the Gri bug.
3. An indication that your commandfile is flawed, either in syntax or in meaning. These messages end with a line indicating the offending line in your commandfile, e.g. the command `set x axis 0 1 -1` yields:

   ```
   ERROR: `set x axis .left. .right. .incBig.'
          has .incBig. of wrong sign
    Bad command:  `set x axis 0 1 -1 '
   ```

   Normally, such error messages do not indicate a flaw in Gri, but rather in your reasoning, so report them to the author only if you are very sure that a Gri bug must underly them.

10.15 Missing data

Most Gri commands will ignore data points equal to a "missing value." For example, `draw curve` connects only points which are not equal (to within 0.01 percent) of the missing value.

The curve has holes at missing data. Initially the missing-value is set to 1.0e22. You may alter this value with `set missing value .value.`. The built-in variable `..missingvalue..` stores the current value of the missing-value.

Additionally, Gri will ignore anything it reads that is equal to the string `NaN` or `Inf`. These are produced by matlab, C, and other programs when dividing by zero, etc.

Gri also ignores mathematical operations on data items which are equal to the missing value. Thus, for example, if your missing value is -99 then the command `x += 1` will not change the values equal to -99 to -98, since this would have the side-effect of making the datum no longer be considered missing.

10.16 Interaction Between Gri and Operating System

10.16.1 Using the OS from within Gri

Gri uses the operating system internally for things like paging through help information.

The operating system may be called **within** Gri commands, using a syntax borrowed from the `Bash` unix shell. After substituting synonyms in the commandline, Gri scans for dollar-parenthesis blocks (e.g. `\$(system-command)`, replacing them with the textual result of sending the indicated system-command to the OS. The replacements are done from left to right in the commandline, starting at the deepest nesting level.

Often the dollar-parentheis syntax is used in title commands, to indicate the full pathname of the Gri commandfile, e.g.

```
draw title "\$(pwd)/\.command_file."
```

In assignment to synonyms, expansion of dollar-parenthesis is not done. Thus the operating system is called twice on the second line below, and not at all on the first line; to see this, run it as `gri -s8 -t`.

```
\dir = "\$(echo $MY_DIR)"
show "\$(head -1 \dir/MY_FILE)"
```

Syntax Note Dollar-parenthesis blocks must be prefixed with backslash to avoid confusion with math expressions within strings, for example to avoid breaking `draw label "$(x,y)$" at 3 3 cm`. This is an example of how TEX notation and unix shell notation collide.

Example It is a good idea to employ unix environment variables to name directories containing data, so that Gri scripts will work unchanged even if the data are moved (so long as the environment variables are altered), e.g.

```
# Figure how many lines in a file
\dir ="$(echo DIRECTORY_WHERE_I_STORE_SOLAR_DATA)"
open "\dir/solar_data_file_name‘
...
open "\$(echo DIR_ANOTHER)/another_data_set"
```

Another method is to pass instructions to the operating system with the `system` command. This discards output. Whatever follows the word `system` is first scanned for synonyms (but not rpn expressions or variables); after replacement of any existing synonyms, the line is passed directly to the operating system. Any results are printed on the terminal.

Frequently used system commands are `awk`, `head`, `grep` and `sed`. Examples:

- Here's how to paste several files together to form a temporary file for plotting. (Notice that a temporary file incorporating the PID of the job is created and later removed.)

```
system paste -d" " 1.dat 2.dat 3.dat > tmp.\.pid.
open tmp.\.pid.
read columns x y
close
system rm tmp.\.pid.
```

- Here's how to plot each line in a file called '`inp`' which has the string `;` at the start of the line.

```
system cat inp | grep -v "^;" | cat > tmp.\.pid.
open tmp.\.pid.
read columns x y
system rm tmp.\.pid.
```

- Here's how to use the `awk` system command to create a tabulated function for plotting.

```
system awk                                \
    'BEGIN {                              \
       for (x=0; x<1; x+=0.1) {           \
         printf ("%f %f\n", x, sin(x)) \
       }                                  \
     }'                                   \
    > tmp.\.pid.
open tmp.\.pid.
read columns x y
close
system rm tmp.\.pid.
draw curve
```

  This example is more cleanly written using the piping facility of the `open` command (which automatically creates a temporary file, and destroys it when `close` is done)

```
open "awk 'BEGIN {       \
  for(x=0;x<1;x+=0.1) {  \
    print(x,sin(x))      \
  }                      \
}'|"
read columns x y
close
draw curve
```

- Sometimes you need just a single output item from the operating system. In this case, you can store the results from the operating system in a synonym by using the `\synonym = system ...` assignment command.
- subroutine A related command is `\synonym = tmpname`, which stores in the synonym the name of a temporary file created by the system. The file is created with the `tmpname` system call, so it is guaranteed not to clash with any existing files. Typically, the filename is something like '`/usr/tmp/griAAAa1233`'. In many cases you'll want to remove the file within Gri, once you're done, and that can be done with `unlink` (Section 9.3.51 [Unlink], page 141) or with a `system rm -f` command. A useful bit of code is as follows

```
\file = tmpname
system ... SOME_SYSTEM_COMMANDS ... > \file
... use this new file for something ...
unlink \file
```

10.16.2 Using Gri from within the OS

This section only applies to unix systems.

Save the following into a file called 'p' and then make it executable using `chmod`. It runs Gri on a named file, with the `-yes` flag set so that any `query` commands are automatically answered in the affirmative, and then displays the results in a Ghostscript window. (USAGE: `p cmdfile.gri`)

```
#!/usr/bin/sh
# Run Gri, then plot in gs window
case $# in
1)
        base=‘basename $1 .gri | sed -e s/.*,#‘
        gri -yes $base.gri && ghostview $base.ps
        ;;
*)
        echo "Proper usage: $0 cmdfile.gri"
        ;;
esac
```

10.17 Sample Resource File

The following shows a sample '`~/.grirc`' resource file. Much of the code here is adding new commands (Section 10.11 [Adding New Commands], page 173.)

```
# Some folks like tics to point inwards ...
set tics in

# Hey, I'm bored with Helvetica font, and
# Palatino is a bit prettier than Times
set font to PalatinoRoman

# Now for something a bit less trivial.
# This lets me draw a set of y-values
# against a single x-value, by e.g.
#    open xyyy.dat
#    draw curves time T1 T2 T3
# where the first column of the file is
# time, and the others are temperatures
# at various sensors.
‘draw curves \xname \y1name ...’
Draw multiple y columns versus an x column.  Assumes
that the datafile is open, and that x is in the first
column, with the y values in one or more following
columns.

The number of columns is figured out from the options,
as is the name of the x-axis, and the labels to be
used on each of the y curves.
{
  # NB. the 3 below lets us skip the words 'draw'
```

```
  # and 'curves', and the name of the x-column.
  .num_of_y_columns. = {rpn wordc 3 -}
  if {rpn .num_of_y_columns. 1 >}
    show "ERROR: 'draw curves' needs at least 1 y column!"
    quit
  end if

  set x name {rpn 2 wordv}
  set y name ""

  # Loop through the columns.
  .col. = 0
  while {rpn .num_of_y_columns. .col. <}
    # The x-values will be in column 1, with y-values
    # in columns 2, 3, ..., of the file.
    .ycol. = {rpn .col. 2 +}
    rewind
    read columns x=1 y=.ycol.
    # At this point, you may want to change line thickness,
    # thickness, color, dash-type, etc.  For illustration,
    # let's set dash type to the column number.
    set dash .col.
    draw curve
    draw label for last curve {rpn .col. 3 + wordv}
    .col. += 1
  end while
}
```

11 Environment

Gri comes with a few ancillary programs that may be useful. Gri also provides a simple scheme to use other system tools (e.g. `awk`). These are discussed here. And speaking of discussing, this chapter ends with a note about a mail-list Gri discussion group.

11.1 Extra things provided with Gri

11.1.1 `gri_merge` – combine PostScript files

Merge separate PostScript files, created by Gri, into one page. To learn how it works, type `gri_merge -h` at the system prompt, to see:

```
PURPOSE: Strings Gri PostScript files together.

BUGS:    With old versions, of gri, make sure to match each ‘set clip
 postscript on’ with a ‘set clip postscript off’.

USAGE (style 1):
    gri_merge [OPTIONS] CxR a.ps b.ps ... > merged_file.ps
 Merges the files onto one page, in ’C’ columns and ’R’ rows.  The C*R
 files are given in the order of words on a page.  The page is
 presumed to be 8.5x11 in size, as are all the input files, and the
 input files are sized to fit, and kept in natural scale.

USAGE (style 2):
    gri_merge [OPTIONS] xcm ycm enlarge a.ps [b.ps ...] > merged_file.ps
 Where ‘enlarge’ is a scale factor applied after offsetting ‘xcm’ to
 the right and ‘ycm’ upward.

EXAMPLE (style 2):
 The following
   gri_merge  2 12 .5 a.ps \
             12 12 .5 b.ps \
              2  2 .5 c.ps \
             12  2 .5 d.ps > all.ps
 produces 4 panels from gri plots done using margins and sizes
 as specified in the following lines in a gri commandfile
   set x margin 2
   set x size 15
   set y margin 2
   set y size 15
 The OPTIONS, available if your ’perl’ has ’getopts’ library, are:
   -u graylevel -- set graylevel for underlay beneath panels, by default 0.75.
                   Values range from 0 (black) to 1 (white), although a value
                   of precisely 1 means do NOT draw underlay.
   -b graylevel -- Set value for background under individual panels, again 0
                   for black to 1 for white, with 1 meaning no drawing.
   -h           -- Print this help message and quit.
```

11.1.2 `gri_unpage` – split pages into files

Given a multipage PostScript file, `gri_unpage` creates several new PostScript files, one for each page. To learn how it works, type `gri_unpage -h` at the system prompt, to see:

```
PURPOSE: Chop multipage Gri PostScript file into one file per page.
USAGE:   gri_unpage name.ps -- create files name-1.ps, name-2.ps, etc, one
                               for each page of name.ps.
         gri_unpage -h      -- print this help message

BUGS:    1. Bounding box is always 8.5x11 inches.
         2. Assumes that all Gri fonts are used even if they aren't.
```

11.2 Using System Tools With Gri

Using system tools to manipulate your data makes sense for several reasons. First, you may be familiar with those tools already. Second, learning these tools will help you in all your work.

11.2.1 Introduction

Each of the programs listed in the sections below is available for Unix. Some (e.g. Perl and the Awk variant called Gawk) are available on other operating systems as well. Each of these tools is fully documented in Unix manpages, so here I'll just give an indication of a couple of useful techniques you might want to use.

Bear in mind that these tools can do very similar jobs. For example, Awk can do much of what Sed and Grep can do, but also a whole lot more. If you don't know Sed or Grep, I suggest you learn Awk instead. Then again, Perl can also do anything Gawk can do, and a whole lot more! (For one thing, it is easier to work with multiple files in Perl.) In fact, Perl is the most powerful of this list. If you know none of these tools, you might want to learn Perl instead of the others. But Perl is more complicated for simple work than Awk is, so the most reasonable path might be to learn both Awk and Perl.

For simple applications, you will probably want to use these tools in a piped open command, e.g.

```
open "awk '{print $1, $3/$2}' MyFile |"
```

which creates a temporary file (automatically erased when Gri finishes) which contains the output from running the system command that preceeds the pipe symbol (`|`) (Section 9.3.28 [Open], page 102).

(Here and in all the examples of this chapter, it is assumed that the user's input file is named '`MyFile`'.)

For more complicated appplications, you may use the Gri `system` command as follows.

```
system perl   >tmp  <<"EOF"
open(IN, "MyFile");
while(<IN>) {
    ($x, $y) = split;
    print "$x", " ", cos($y), "\n";
}
EOF
open tmp
```

```
read columns x y
draw curve
system rm -f tmp
```

Here a temporary file, named 'tmp', has been used to store the results of the calculation. Note that this file was specifically cleaned up by the second `system` command. (Many folks, including the author, would prefer to take the perl script out of the above and make it a standalone executable script, calling it from Gri with the one-line form. But it is just a matter of style.)

11.2.2 Grep

The most common application of Grep is to select lines matching a pattern, or not matching a pattern. Here is how to skip all lines with the word "HEADER" in them:

```
open "grep -v 'HEADER' MyFile |"
...
```

Note that Gawk and Perl do this just as easily.

11.2.3 Sed

Sed is normally used for simple changes to files. For example, if you have columnar data which are separated with comma characters instead of whitespace, you could make it Gri compatible by

```
open "sed -e 's/,/ /g' MyFile |"
```

Where the `-e` flag indicates that the next item is a command to Sed, in this case a command to switch ("s") the comma character with the blank character, globally ("g") across each line of the file. See also the overview of Perl.

11.2.4 Awk

Awk is great for one-liners. If your system lacks Awk, you can procure the GNU version, called Gawk, from the web for free. For better or worse, Gawk is not fully compatible with Awk. The good thing is that Gawk is pretty much the same on all operating systems, whereas the installed Awk may not be. I use Gawk instead of Awk, for this reason and because it is normally faster.

The main concept in Awk is of "patterns" and "actions." In the Awk syntax, actions are written in braces following patterns written with no braces. (This will become clear presently.)

Whenever a line in the data file matches the pattern, the action is done.

The default pattern is to match to every line in the file. This is done if no pattern is supplied.

The default action is to print the line, and this is done if no action is supplied.

Here are a few examples. To skip first 10 lines of a file:

```
open "awk 'NR>10' MyFile |"
read ...
```

Here the pattern was that `NR` (a special Awk variable for the number of the record, starting with 1 for the first line of the file) exceeded 10. And the action was taken by default since nothing was supplied between braces, to print this line.

To plot the cosine of the second column against the first column:

```
open "awk '{print ($1, cos($2))}' MyFile |"
read columns x y
draw curve
```

Here no pattern was supplied, so the action was done for every line.

Combining these two forms, then, and supplying both a pattern and an action, here is how one might print the first and eighth columns of the file '`MyFile`', but only for the first 10 lines of the file:

```
open "awk NR<=10 {print ($1, $8)} MyFile |"
```

11.2.5 Perl

Perl can do almost **anything** with your data, since it is a full programming language designed to also emulate several Unix utilities.

In perl, as in other commands, the commandline switch `-e` indicates that a Perl command is given in the next word of the command line. The commandline switch `-p` indicates to print the line, after any indicated Perl actions have been done on it. Here, for example, is how one would emulate a Sed replacement of comma by blank:

```
open "perl -pe 's/,/ /g' MyFile |"
```

Perl also has a commandline switch `-a` indicating that lines should be "autosplit" into an array called `$F`. The first element of this array is `$F[0]`. Splitting is normally done on white-space character(s), although this may be changed if desired. Here, for example, is how to take the cosine of the second column of a file, and print this after the first column:

```
open "perl -pea 'print($F[0], cos($F[1]))' MyFile |"
```

11.3 Gri Discussion Group

Before emailing to the Gri newsgroup, it is a good idea to consult the list of answers to Frequently Asked Questions.

As of summer 2000, Gri traffic has been moved to the web-based discussion groups at the SourceForge website `http://gri.sourceforge.net` about which little need be said here, because the website is very easy to use.

12 Editing Gri Files in GNU Emacs

If you use the GNU Emacs (or XEmacs) text editor, writing Gri command files is made easier with the Gri editing mode (gri-mode.el), written by Peter S. Galbraith <psg@debian.org>.

12.1 About Gri Mode

Gri mode has all the wonderful things you've come to expect from Emacs modes. Here's a brief overview of the features:

- It can **complete** partially typed commands, builtin variables and synonyms (`gri-complete`, ⟨M-Tab⟩) and help you edit the syntax that was thus inserted for you (`gri-option-select`, *C-c C-o*; `gri-option-kill`, *C-c C-k*).
- It can provide a short help synopsis concerning the command on the current line (`gri-help-this-command`, *C-c C-h*), or load the info manual for that command (`gri-info-this-command`, *C-c C-i*). It knows the list of all Gri commands, and can provide help or info regarding any of them (`gri-help`, *C-c M-h*; `gri-info`, *C-c M-i*) using command name completion at the prompt (⟨Tab⟩).
- All Gri commands are listed in a pull-down menu from the menubar, which you can use to either enter the text of the selected command, or obtain help or info about it.
- It can help you find an unknown command by listing all containing a given word (`gri-apropos`, *C-c C-a*).
- It fontifies your Gri code using colour coding.
- It indents if statements, loops, and so on (`gri-indent-line`, ⟨Tab⟩).
- It can let you run Gri and view its output without leaving the editor (`gri-run`, *C-c C-r*). If an error is encountered, Emacs will rearrange the buffer so the cursor is on the bad line of the Gri command-file.
- If you've already run Gri, and therefore have a PostScript output file, the mode will let you view that file (`gri-view`, *C-c C-v*) even if that file is compressed.

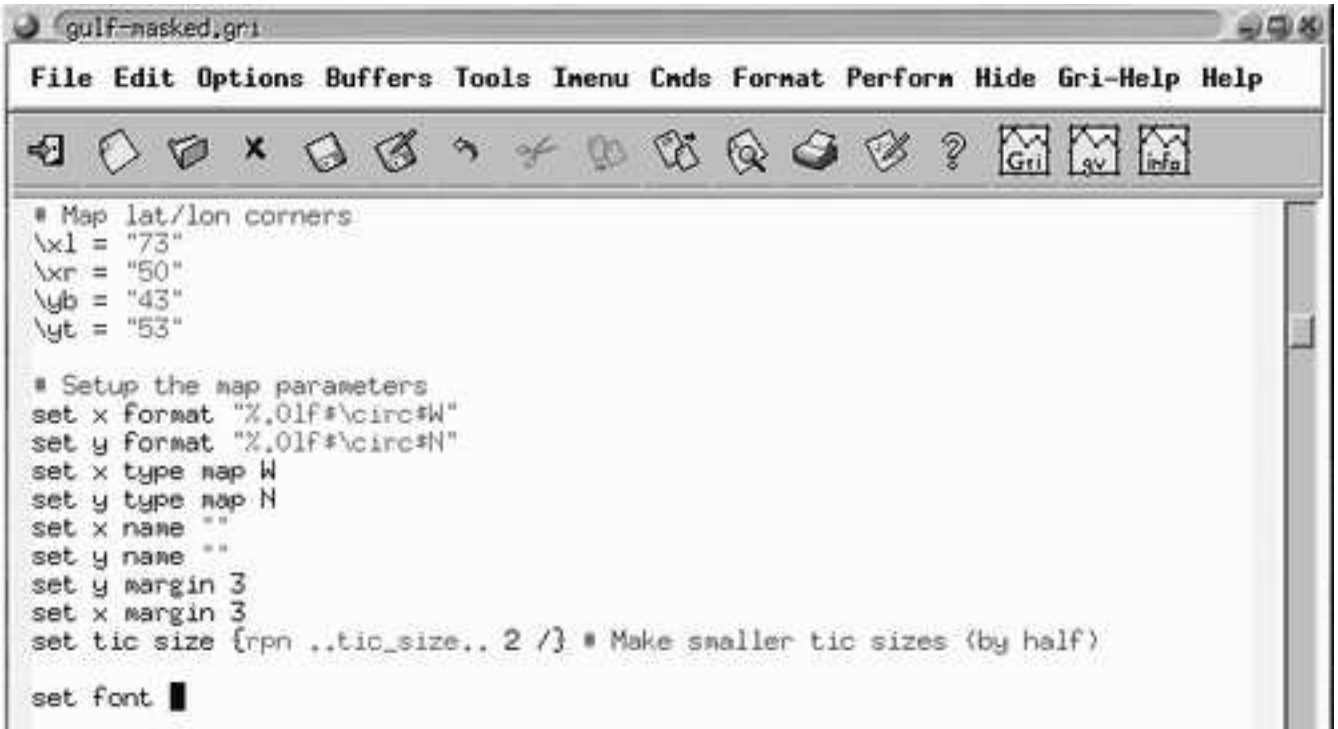
Thus one never has to leave Emacs; type *C-c C-r* to run Gri, and if there is no error, the graph comes up automatically. If there was an error, gri-mode will move editing point to the line with the error and display the error message. Given that the mode can complete partially typed commands, this means a substantial saving in development time.

Inside gri-mode, type *C-h m* for help on the mode, including a list of all commands and key definitions.

12.2 Gri-mode screenshots, what it looks like.

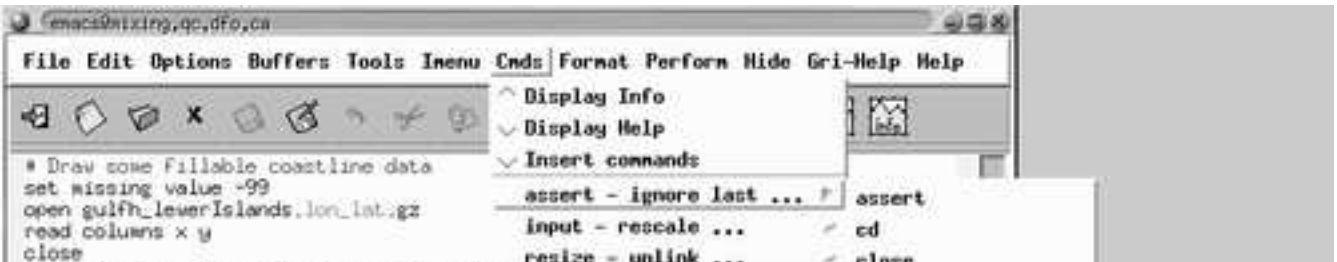
In the first screenshot, the user has entered the text `set font` and has pressed ⟨M-Tab⟩ twice to see the list of possible completions. The '`*Completions*`' buffer is displayed in a separate

frame because the user is using the '`framepop.el`' add-on package. A similar effect can be obtained using `special display` buffers in Emacs.

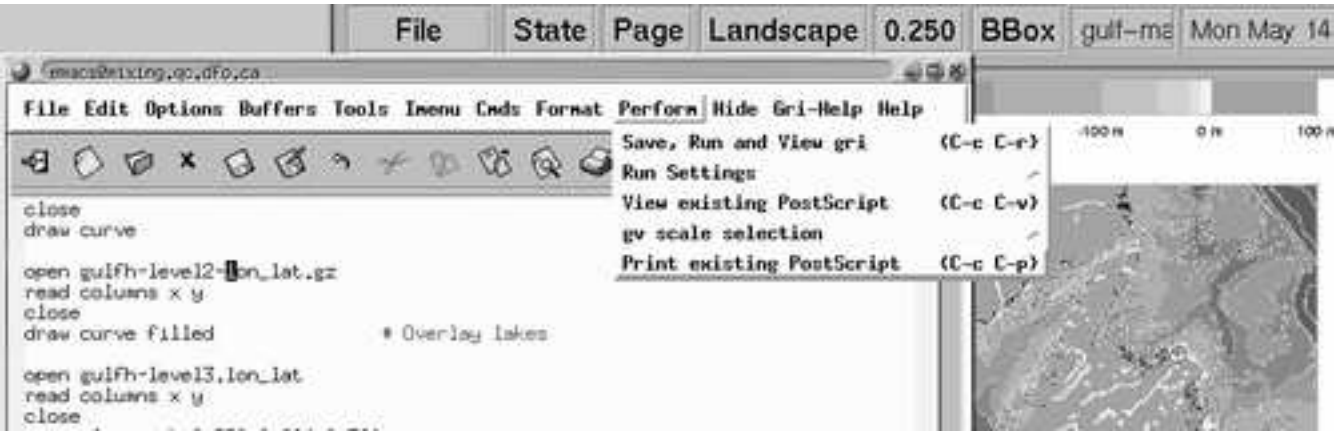


Here, the user has selected the command `set font to` and, forgetting what valid font name could be used, then invoked *C-c C-h* to get help about that command. The user found the needed information and finished typing in 'Helvetica'.

Here we see the user cascading through the Gri commands pull-down menu until `draw axes` is reached.



With the output previewed in a `gv` window after pressing `C-c C-v`, the user is ready to print the file using a pull-down menu.



12.2.1 Screenshot 1

In the first screenshot, the user has entered the text `set font` and has pressed ⟨M-Tab⟩ twice to see the list of possible completions. The '`*Completions*`' buffer is displayed in a separate frame because the user is using the '`framepop.el`' add-on package. A similar effect can be obtained using `special buffers` in Emacs.

12.2.2 Screenshot 2

Here, the user has selected the command `set font to` and, forgetting what valid font name could be used, then invoked *C-c C-h* to get help about that command. The user found the needed information and finished typing in 'Courier'. Meanwhile, an idle timer displays the default setting for this command in the minibuffer after a few seconds of inactivity.

12.2.3 Screenshot 3

Here we see the user cascading through the Gri commands pull-down menu until `draw axes` is reached. This menu can be used to display Info (or help) about a given command, or to insert the selection. The default is Info. This menu is a good way to browse for a command when you don't exactly remember its name.

12.2.4 Screenshot 4

With the output previewed in a `gv` window after pressing *C-c C-v* (or pressing the `gv` icon in the toolbar, or selecting it from the `Perform` pull-down menu), the user is ready to print the file, here using the `Perform` pull-down menu.

12.3 Installing gri-mode.el, the nuts and bolts.

The Emacs '`gri-mode.el`' file is now bundled with gri, so odds are you already have it. If not, you will find it at gri's web site http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/gri/gri/gri-mode.el

The following installation steps appear a **bit** complicated. That's only because gri has changed how it gets installed a few times, and gri-mode.el works with all of these various methods. If you use gri from a Linux package (Debian or Red Hat) or if you compiled it yourself using the default configuration, you won't need to do much.

To install '`gri-mode.el`', follow these 4 steps. If gri-mode is already installed, you can skip the first two steps and move on to the last two, in which you tell Emacs that you'd like to use Gri mode when you edit files that end in '`.gri`', the Gri suffix. (Actually, if you're using Debian linux, you can skip all of these steps since the system will assume that you want gri-mode if you're editing a Gri file.)

12.3.1 Placing gri-mode.el where Emacs can find it.

(Those using gri from gri's **RPM** package, a **Debian** package or a **Red Hat** package users can skip this, as it is done for them)

Extra '`.el`' files like '`gri-mode.el`' that are not part of Emacs should be stored in a directory where Emacs will find them when you ask it to load them. The files should therefore be found in Emacs' **load-path**. To see the directory list currently in the load-path, do this in Emacs:

```
C-h v load-path
```

If you have access to system directories, put gri-mode.el in a **site-lisp** directory, such as '`/usr/local/share/emacs/site-lisp/`' That way all users will have access to the files.

If you don't have access to a site-lisp directory (e.g. you have only a user account), then create a directory where your extra '`.el`' files will be stored and add it to Emacs' load-path. For example, say you created the directory '`~/emacs`' and stored gri-mode.el there, you would then put this near the top of your '`~/.emacs`' file:

```
(setq load-path (cons "~/emacs" load-path))
```

12.3.2 Telling gri-mode where gri resides

(Those using gri from gri's **RPM** package, a **Debian** package or a **Red Hat** package users can skip this, as it is done for them)

You may skip this section if gri is installed on your system as '`/usr/local/bin/gri-2.12.18`' and '`/usr/local/share/gri/2.12.18/gri.cmd`' (the default when compiling gri yourself). If not, then you may need to set the Emacs variable `gri*directory-tree` in a startup file such as in your '`~/.emacs`' file.

The Emacs variable `gri*directory-tree` is used to configure gri-mode to tell it where Gri is installed on your system. For the default gri installation paths used in this gri release, gri-mode expects to find the gri executable and the file gri.cmd as: `gri*directory-tree/2.12.18/gri.cmd` and `/usr/local/bin/gri-2.12.18` where `gri*directory-tree` is by default set to '`/usr/local/share/gri/`'.

If you have only one version of gri installed on your system, gri-mode will also look to find '`gri.cmd`' and the gri executable like so:

1. Gri executable in the PATH, with startup file '`gri*directory-tree/gri.cmd`'.
2. Gri executable in the PATH, with startup file '`gri*directory-tree/lib/gri.cmd`'.
3. Gri executable `gri*directory-tree/bin/gri`, with startup file `gri*directory-tree/lib/gri.cmd`.

However, gri-mode was designed to support, and ease the use of, **multiple installed versions** of gri. To use this feature, you must use the gri version number as a directory name under the `gri*directory-tree` path, like this:

```
gri*directory-tree/VERSION/bin/gri
gri*directory-tree/VERSION/lib/gri.cmd
```

(e.g. '`/opt/gri/2.040/bin/gri`' and '`/opt/gri/2.040/lib/gri.cmd`' with `gri*directory-tree` set to `"/opt/gri"`)

or without the **lib** and **bin** subdirectories if the executable is found in the PATH named like '`gri-VERSION`' (This is the way Debian packages are set up): the file '`gri*directory-tree/VERSION/gri.cmd`' and the '`gri-VERSION`' executable in the PATH (e.g. '`/usr/share/gri/2.1.18/gri.cmd`' and '`/usr/bin/gri-2.1.18`' with `gri*directory-tree` set to `"/usr/share/gri"`)

Important note: You may have more than one tree and make a list of them:

```
(setq gri*directory-tree '("/opt/gri/" "/usr/share/gri/"))
```

Examples:

1. If you use a RedHat package installed like:

   ```
   /usr/bin/gri
   /usr/share/gri/gri.cmd
   ```

   then you'd also use:

   ```
   (setq gri*directory-tree "/usr/share/gri/")
   ```

   but gri-mode would know of only one installed version of gri.
2. If you use a Debian GNU/Linux installation like:

   ```
   /usr/bin/gri -> /usr/bin/gri-2.1.18
   /usr/share/gri/2.1.18/gri.cmd
   ```

   then you'd use:

   ```
   (setq gri*directory-tree "/usr/share/gri/")
   ```

   Note that all gri binaries must exist in the path with version number suffixes (e.g. '`gri-2.1.18`') since there is no '`/usr/share/gri/2.1.18/bin/`' directory (using a similar structure to '`opt/gri`' below) where gri-mode can find the binary corresponding to a given version number.
3. If you had multiple versions of Gri installed like so (this reflects the installation paths used in older gri releases):

   ```
   /opt/gri/2.040/bin/gri
   /opt/gri/2.040/lib/gri.cmd
   /opt/gri/2.041/bin/gri
   /opt/gri/2.041/lib/gri.cmd
   ```

   then you'd use:

   ```
   (setq gri*directory-tree "/opt/gri/")
   ```

12.3.3 Telling emacs to load gri-mode

(Those using a **Debian** package can skip this, as it is done for them)

To tell emacs to use this mode with '`.gri`' files, you can load gri-mode whenever a new emacs session is starting by adding the following line to your '`~/.emacs`' file:

```
(require 'gri-mode)
```

This is a good method when you only start emacs once a week and use it for every file you edit (as you should).

If you startup a fresh emacs every time you edit then you probably only want to load gri-mode into emacs when you need it. In that case, instead of the `require` statement above, add the following lines to your '`~/.emacs`' file:

```
(autoload 'gri-mode "gri-mode" "Enter Gri-mode." t)
(setq auto-mode-alist (cons '("\\.gri$" . gri-mode) auto-mode-alist))
```

The first line tells Emacs that it will find out what it needs to know about running the command `M-x gri-mode` by loading the file '`gri-mode.el`'. The second line tells it to run the command `M-x gri-mode` when a file with extension '`.gri`' is visited (thus using gri-mode with all those files).

12.3.4 Extra user configuration of gri-mode

All users should do this at some time.

At this point, gri-mode should start up when you edit a gri file. You may optionally customize gri-mode by:

1. using the Custom interface (see the Help or Gri-Help menu or run the command `M-x gri-customize`), or
2. manually setting variables in your '`~/.emacs`' file. These are briefly described by typing *`C-h m`* while in gri-mode. Then, for further infomation, use emacs' `describe-variable` command, bound to *`C-h v`*. For example, for more information about the `gri*WWW-program` variable, you'd type `C-h v gri*WWW-program` (note that emacs does ⟨Tab⟩ completion, so pressing the ⟨Tab⟩ key after typing-in gri will display all gri related variables.)

12.4 Major Gri-mode commands.

This section describes the major gri-mode commands, briefly showing how they can increase your productivity.

12.4.1 How gri-mode names Gri commands

A major feature of Gri mode is the completion of partially typed commands. Let's examine how gri-mode decides to name gri commands. A name is determined by removing bracketed options from the syntax line of a given command. This is different from what the gri parser does. In this way, the gri command

```
draw label "\string" [centered|rightjustified] at .x. .y. [cm] [rotated .deg.]
```

is named by gri-mode simply `draw label at`. Note how the 'at' stays in the name because it is not optional. So when you see `draw label at` in gri-mode's menus or prompts, you are more likely to associate the name with what the command actually does.

12.4.2 Possible completions of gri command names

When you press ⟨M-tab⟩ to complete a command name (or a variable or synonym name as described below), gri-mode will expand it as much as it can and do nothing further. If you type in nothing more and insist by using `gri-complete` (⟨M-tab⟩) again, gri-mode will respond by showing all possible completions in the '`*completions*`' buffer. In this way you can use `gri-complete` word-by-word to abbreviate commands without ever displaying completions, like you would for file completion in emacs or bash.

If a completion is ambiguous, but could be exact, invoke gri-complete a second time to complete it. e.g.

`sh`⟨M-tab⟩

expands to

`show`

and informs you that 12 possible completions exists; then

`show`⟨M-tab⟩

will display these completions in the completions buffer; then typing ⟨M-tab⟩ again forces completion to a complete but not unique possibility:

```
show .value.|{rpn ...}|"\text" [.value.|{rpn ...}|text [...]]
```

Completions are shown immediately (without invoking gri-complete again) if the completions window is already displayed or if there are 3 possibilities or less. In this case they are displayed in the minibuffer.

Note: The '`*completions*`' window is deleted after a command is fully completed. `gri-complete` uses its own '`*completions*`' buffer, which is not displayed in the buffer-list to avoid clutter.

12.4.3 Command name abbreviation

gri-mode uses command name abbreviations in a non-Unix way in that all words that compose the command name may be abbreviated (and not only the word preceding the cursor). For example, one can type in

```
dr x a⟨M-tab⟩
```

and it will expand1 all the words to

```
draw x axis [at bottom|top|{.y. [cm]} [lower|upper]]
```

This is reminiscent of VMS, which the author was used to when gri-mode.el was initially created. It's likely that a gri-mode.el rewritten from scratch wouldn't have this feature since it's not part of the Unix mindset.

12.4.4 Variable (and synonym) completion

It's also possible to do ⟨M-tab⟩ completion on gri builtin variable and synonym names while editing a command. Simply type in the leading `.` (dot) or `\` (slash) character and invoke completion with ⟨M-tab⟩.

For example,

```
..x⟨M-tab⟩
```

displays a completions buffer listing the possible completions:

```
..xmargin..                    ..xsize..
..xleft..                      ..xright..
..xlast..
```

Typing in an extra `m` will complete to `..xmargin..` and the following help information will be displayed in the minibuffer:

```
margin to left of axis area. Default is 6 cm.
```

12.4.5 Editing the syntax output by gri-complete

You might wonder what to do with all the bracketed code left behind by `gri-complete`. It certainly won't go through the gri parser without error, so you have to edit it out.

The tools provided in gri-mode are `gri-option-select` (*C-C C-o*) and `gri-kill-option` (*C-C C-k*) to narrow in on a particular gri command, given a syntax description left on the line by `gri-complete`. The cursor location is used to decide which gri command(s) to narrow to.

For example, if `gri-complete` is used on the line 'dr x a', the result will be a line like

```
draw x axis [at bottom|top|{.y. [cm]} [lower|upper]]
```

This is the gri way of describing many commands at once. The above syntax description is a shortcut formulation for all of:

```
draw x axis
draw x axis at bottom
draw x axis at bottom top
draw x axis at bottom bottom
draw x axis at top
draw x axis at top top
draw x axis at top bottom
draw x axis at .y. cm
draw x axis at .y. cm lower
draw x axis at .y. cm upper
```

The `gri-option-select` (*C-C C-o*) command provides easy navigation to select one of these commands. The narrowing process is governed by the cursor position. For example, to get the command narrowed down to

```
draw x axis at bottom [lower|upper]
```

place the cursor somewhere in the word 'bottom' and invoke `gri-option-select`. To complete the narrowing process, selecting

```
draw x axis at bottom lower
```

move the cursor to some place in the word 'lower' and invoke `gri-option-select again`. On the other hand, to get

```
draw x axis at bottom
```

you would have put the cursor over either the word 'lower' or 'upper', and invoke `gri-kill-option` (*C-C C-k*) instead.

You might want to practice using this example to learn how to do it. If you make a mistake, note that the normal Emacs undo works.

12.4.6 User commands with gri-mode

User gri commands defined in '`~/.grirc`' (Section 10.17 [Resource File], page 185) or at the beginning of a gri file can also be used with `gri-complete`. Note that user commands are added from the current buffer whenever 'gri-mode' is invoked. They may override previous user commands, but not gri system commands.

12.4.7 Inserting gri code fragments in Emacs

Since gri version 1.063, gri has special commands that begin with a question mark '?'. These special commands have no options, and are composed only of standard gri commands. Their purpose is to provide a short-cut for entering many lines of gri at once (e.g. bits of sample code about contouring grids, or your own preamble which you use at the time to set fonts and line widths).

`gri-complete` acts in a special way with these commands, by replacing the abbreviated name which you are completing by all the lines contained within the gri command.

The user is allowed to define new fragments in '`~/.grirc`', and also to override the gri system fragments. You can therefore fine-tune gri's fragments to your taste. To see the names of all gri fragments, type in a question mark at the beginning of a line in a gri buffer and press ⟨M-Tab⟩ twice to `gri-complete` it and display possible choices. The gri commands used to replace them are found in the '`*gri-syntax*`' buffer.

12.4.8 Info interface for help on current command

The Info system is built into Emacs (*C-h i*), and provides an easy method of navigation on-line help (including this manual). We suggest that you install the gri info files on your system (they are already installed with packaged versions of gri in Debian or Red Hat formats).

Not only will Info (*C-h i*) be able to navigate through all of gri's inline manual, but gri-mode's function `gri-info-this-command` (*C-c C-i*) will display the correct info page about the gri command being edited on the current line.

Additionally, gri-mode has the command `gri-info` (*C-c M-i*) which will prompt you (using ⟨tab⟩ completion) for any command to list Info about. There's also a menu-bar pull-down menu which lists all gri commands, and you can get to Info from that too.

12.5 Other features of gri-mode

- `IMenu support`: IMenu is a GNU/Emacs package used to source code files. It can be setup for a particular mode to list a menu of function definitions and variable declarations. In gri-mode, the IMenu is available from the menubar and provides links to `new command declarations` as well as value settings for `variables` and `synomyns`.
- `Toolbar support`: XEmacs21 and Emacs-21 have support for a toolbar (a set of iconic buttons that are shortcuts for commands), however the two editors don't not have compatible toolbar setups. Currently gri-mode only defines a single icon for XEmacs (to run gri on the current buffer) and defines three for GNU/Emacs (to run gri, to view the postscript file, and to lookup Info about the command on the current line).
- `Idle Help`: When GNU/Emacs sits idle with the cursor on a command line, it looks up any defaults the command may have and displays the information in the minibuffer. For example, sitting on a line that says `set font size 10`, it will display `set font size: default is 12 point in variable ..fontsize...`

12.6 Dealing with many Gri versions, gri-mode handles it.

Earlier we explained that you might have many versions of gri installed on your system (Section 12.3 [Installing gri-mode.el], page 196). You may use gri-mode to access these various versions with the following gri-mode commands:

- `gri-set-version`: Tells gri-mode to use a given version as the default for all your gri files. Tab completion is available listing all available versions (if they are not all listed, that means that the Emacs variable `gri*directory-tree` is not correctly set (Section 12.3 [Installing gri-mode.el], page 196)). Your selection is stored in the file '`~/.gri-using-version`' and is remembered across editing sessions. If you select **default** as your answer, gri-mode reverts to using the default version of gri as installed on your system (which may change after you update it) and deletes the '`~/.gri-using-version`' file.
- `gri-set-local-version`: Tells gri-mode to use a given version of gri **this** gri file (the one currently being edited when you invoke the command). Tab completion is again available to list all available versions. Your selection overrides the default version selected by `gri-set-version` for this file, and is stored as an Emacs variable at the bottom of the gri file. It is remembered across editing sessions. If you select **default**

as your answer, gri-mode reverts to using the default version of gri as selected by `gri-set-version` and deletes the Emacs variable setting at the bottom of the gri file.

12.7 Filename arguments when running gri

Usually, gri is run specifying only a gri command file to process, which lends itself well to the gri-mode command `gri-run`. But Gri can be also invoked from the command line using optional arguments, usually filenames but not necessarily, e.g.

```
$ gri somefile.gri datafile.dat datafile2.dat ...
```

or

```
$ gri somefile.gri *.dat
```

The arguments are accessed with RPN operators '`argc`' and '`argv`' (Section 10.9.5 [Unary Operators], page 164).

gri-mode provides a method to set the arguments (usually filenames) to use when `gri-run` is called in a specific gri script. Use the gri-mode command `gri-set-command-postarguments` to setup a string that gri-mode will use, and the gri-mode command `gri-unset-command-postarguments` to clear it. For ease-of-use, these commands are made available from the menubar under the `Perform -> Run Settings` entries. The specified string will be stored in the locally-defined (aka buffer-local) variable `gri-command-postarguments` and will be written within a gri comment at the bottom of the file such that Emacs remembers it across editing sessions.

13 History of Gri Versions

Gri, like many modern software projects, has two "streams" of development running side by side. One is called "stable," the other "unstable." *Most users should use the stable stream*, which has been well tested. (The unstable version is for users who wish to try out new features that are under development.)

Stable versions always have an even number as the middle number in their version designation, e.g. `2.6.0`, while unstable versions always have an odd number there, e.g. `2.7.0`. This middle number indicates a sort of family of related releases. Within a stable stream, all versions sharing a given middle number have the same features; new releases are made only to fix bugs. Thus, version `2.6.1` fixed bugs in version `2.6.0`.

To learn more about the bugs, and their resolution, visit the bugs page at gri.sourceforge.net and adjust the "status" menu to read "closed".

13.1 Stable Stream

In any given stable stream, only the version with a `0` as the last number in the triplet has new features; the later versions only correct flaws. For example, version `2.6.0` offers new features compared to any of the `2.4.x` versions, but `2.6.1` differs only in the fixing of bugs.

13.1.1 Version 2.12

13.1.1.1 Version 2.12.18 [2008 Sep 8 International Literacy Day]

Bug Fixes

- Improve security of temporary-file handling.
- Fix SourceForge bug #1985862 . . . SVG output had axis linewidth equal to curve line width.

13.1.1.2 Version 2.12.17 [2008 May 29 Oak Apple Day (England)]

New Features

- Add GNU readline support so that interactive mode will have history, command editing, etc.

Bug Fixes

- Fix SourceForge bug #1913577 . . . superscripts did not end correctly, if preceeded by an inline `{}` block.
- Fix SourceForge bug #1761562 . . . y axis name printed upside down, for log axes in which user specified a high values at the bottom end of the axis

13.1.1.3 Version 2.12.16 [2007 Jul 20 anniversary of the first moon landing]

Bug Fixes

- Fix Debian bug #130802 . . . postscript problem in landscape mode, refreshed in gv viewer
- Fix Debian bug #434010 . . . `set page landscape` requires `set page size` first, but it should really default to something reasonable instead.

13.1.1.4 Version 2.12.15 [2007 Apr 16 celebration of birthday of Muhammad]

Bug Fixes

- Fix SourceForge bug #1700978 . . . html concept index mostly broken
- Fix SourceForge bug #1698924 . . . box plots show missing data
- Fix Debian bug #417217 . . . will not compile in GCC 4.3
- Fix SourceForge bug #1698116 . . . poorly-positioned name of RHS y-axis

13.1.1.5 Version 2.12.14 [2007 Jan 08: Coming-of-Age Day (Japan)]

Bug Fixes

- Fix SourceForge bug #1630768 . . . Fix to segfault in clipped images (a bug that may have developed after version 2.13.3)

13.1.1.6 Version 2.12.13 [2006 Nov 06: Constitution Day (Tajikistan)]

Bug Fixes

- Fix SourceForge bug #1591475 . . . Fix to compile in Solaris CC
- Fix SourceForge bug #1591062 . . . Fix to compile in OpenBSD

13.1.1.7 Version 2.12.12 [2006 July 16: Yellow Pigs Day]

Bug Fixes

- Fix SourceForge bug #1523033 . . . Malloc error (freeing something already freed?)
- Fix SourceForge bug #1523032 . . . `create columns from function` bug, if there is an existing directory called `tmp`.
- Fix SourceForge bug #1491105 . . . `set x axis labels` had no affect for log axes (same for y)

13.1.1.8 Version 2.12.11 [2006 Mar 30: Hindu New Year]

Bug Fixes

- Fix SourceForge bug #1449546 . . . x axis limits not correctly inferred from `set x grid` (same for y).

13.1.1.9 Version 2.12.10 [2006 Jan 26: Australia Day]

Bug Fixes

- Fix SourceForge bug #1408259 . . . PostScript file contained private information. This was fixed by adding new commandline arguments `-private` and `-no_private`, the former of which (the new default) means to not include the user's name, the invocation arguments, or the command-file contents (Chapter 3 [Invoking Gri], page 7).
- Fix SourceForge bug #1285180 . . . NaN was mishandled. (The bug may have arisen in version 2.12.7 or thereabouts.)
- Port to the FreeBSD operating system, with help from Christopher Illies and Roman Neuhauser.
- Fix SourceForge bug #1217273 . . . missing some version numbers within docs
- Fix SourceForge bug #1196613 . . . user-supplied x-axis labels can run offscale (fix for y-axis later...)
- Fix SourceForge bug #1198341 . . . x-axis labels incorrectly rotated (sometimes)
- Fix SourceForge bug #1199280 . . . warning about `malloc` for RPN assignments
- Fix SourceForge bug #1196115 . . . `gri_unpage` and `gri_merge` mis-installed
- Fix SourceForge bug #1153209 . . . Emacs mode incompatible with new version of `gv` PostScript viewer

  Fix SourceForge bug #1101172 . . . `gri -help` incorrectly stated meaning of last argument(s)
- Fix SourceForge bug #835711 . . . `draw gri logo` fails.
- Fix SourceForge bug #1098269 . . . problem compiling on AMD64 machine. (Solution provided by Andreas Jochens, a Debian user.)
- Fix SourceForge bug #867515 . . . problem with junk appearing in images.
- Fix SourceForge bug #875881 . . . problem compiling with gcc 2.95.3 compiler.

13.1.1.10 Version 2.12.9 [2005 Jan 6: Feast of Epiphany]

Bug Fixes

- Fix SourceForge bug #1094087 . . . `set path to` incorrectly parsed colon-separated paths
- Fix SourceForge bug #1085788 . . . `image *=`, `image /=`, `image ^=`, and `image _=` all gave incorrect results
- Fix SourceForge bug #1084123 . . . does not compile in fink
- Fix SourceForge bug #676767 . . . on fink systems, `help` does not work

13.1.1.11 Version 2.12.8 [2004]

Bug Fixes

- Fix SourceForge bug #1019141 . . . `draw arc` ignores the present pen color
- Fix SourceForge bug #997741 . . . PostScript broken on images with y-axis decreasing, and enclosed by PostScript clipping
- Fix SourceForge bug #978822 . . . documentation wrong on `set path to`
- Fix SourceForge bug #932203 . . . misplaced labels caused by `set x axis labels`
- Fix SourceForge bug #928277 . . . `draw polygon` should take `cm` and `pt` units
- Fix SourceForge bug #930259 . . . fix `draw arc`'s drawing of an extra line (thanks for the fix, Wolfgang Voegeli)

- Fix SourceForge bug #923719 ... `draw curve overlying` ignored the effect of `set dash`
- Fix SourceForge bug #914125 ... offpage points in axes were reported as having been drawn by `draw curve`.
- Fix SourceForge bug #877613 ... `help` (and other commands using temporary files) does not work in OSX/Fink version.
- Fix SourceForge bug #874483 ... `state save` doesn't keep track of `dash` settings.
- Fix SourceForge bug #873245 ... inaccurate times are given in the warnings about slow operations on OSX platform (days are reported instead of seconds)
- Fix SourceForge bug #871477 ... the 'missing value' feature should not be the default. The solution involved adding a new command `set missing value none`, which is now the default.

13.1.1.12 Version 2.12.7 [2003 Sep 4]

Bug Fixes

- Fix SourceForge bug #800022 AKA Debian bug #208589, ... did not build on some Debian platforms because it was based on an old version of `automake`.

13.1.1.13 Version 2.12.6 [2003 Sep 1: Labour Day]

New Features

- Add `age` RPN function, for testing file ages ([age-rpn-operator], page 164).

Bug Fixes

- Fix SourceForge bug #773850 ... bounding-box is increased by `draw symbol` even if (rectangular) postscript clipping is active.
- Fix SourceForge bug #760130 ... Solaris cannot compile with ⟨C-l⟩ in Makefile.
- Fix SourceForge bug #743134 ... bounding box not limited by `set clip postscript`
- Fix SourceForge bug #750561 ... during compilation, `make` rebuilds HTML docs even if up-to-date

13.1.1.14 Version 2.12.5 [2003 May 19: Victoria Day]

New Features

- Apply a patch provided Kawamura Masao, relating to (a) errors in the documentation of file locations and (b) a programming error hidden behind an unset precompiler flag.
- Add hexadecimal colornames ([pen-color-hexadecimal], page 18).

Bug Fixes

- Fix SourceForge bug #739761 ... `draw time stamp` named the command-file incorrectly
- Fix SourceForge bug #720607 ... the emacs mode looked for html documentation files in an incorrect location on linux/redhat systems

13.1.1.15 Version 2.12.4 [2003 Apr 06]

Bug Fixes

- Fix SourceForge bug #696073 ... did not handle `\$()` syntax correctly.
- Fix SourceForge bug #715884 ... mixup on protected-quotes within double-quoted strings
- Fix Debian bug #183912 ... would not compile on m86k architecture.
- Fix SourceForge bug #711354 ... `Creator:` comment in PostScript file named the command-file incorrectly, if there were options on the gri invocation command. Since this naming of the command-file was not especially useful, and since nobody has complained of this bug but the author, the feature has simply been deleted. Just complain if you want it back!
- Fix SourceForge bug #706202 ... A hint about the page orientation was not given in the Postscript comments. The lack of this hint has no affect on printing, etc., only viewing in some viewers.

13.1.1.16 Version 2.12.3 [2003 Mar 1]

Bug Fixes

- Fix SourceForge bug #685919 ... `-output` commandline argument failed on '`.eps`' file extension.

13.1.1.17 Version 2.12.2 [2003 Feb 7: Munro Day at Dalhousie University]

Bug Fixes

- Fix SourceForge bug #675304 ... Segmentation fault can occur on `read image pgm` if an image already exists, e.g. created by `convert grid to image`.
- Fix SourceForge bug #647234 ... Source file '`draw.cc`' will not compile on MAC OS X 10.1.5 at optimization level 2, so drop the level to no optimization, as a temporary measure.
- Fix SourceForge bug #671022 ... `flip image x|y` did not flip images correctly, except in special circumstances.
- Fix SourceForge bug #669603 ... `skip backward .n.` erroneously skipped forward
- Fix SourceForge bug #667754 ... `read image pgm` segfaults on memory if an image has already been created by `convert grid to image`
- Fix SourceForge bug #664388 ... `read image pgm` fails if an image already exists
- Fix SourceForge bug #654129 ... ignoring '`~/.grirc`' file
- Fix SourceForge bug #654127 ... configure scripts are broken
- Fix SourceForge bug #649132 ... removed unused LDFLAGS phrase from Makefile
- Fix SourceForge bug #649134 ... tweak gcc optimization
- Fix SourceForge bug #649136 ... examples 8 and 9 broken
- Fix SourceForge bug #641406 ... RPN too aggressive on missing values

13.1.1.18 Version 2.12.1 [2002 Sep 25]

Bug Fixes

- Fix SourceForge bug #613075 ... RPN operators `sin`, `cos`, and `tan` were limited in range (new bug in last release)

13.1.1.19 Version 2.12.0 [2002 Sep 15: Terry Fox Day, Canada.]

New Features

- Add `sed` RPN operator, to work on strings Section 10.9.4 [Binary Operators], page 162.
- Add `skewness` and `kurtosis` RPN operators, to work on columns Section 10.9.7 [Manipulation of Columns etc], page 169.

Removed Features

- n/a

Bug Fixes

- Fix SourceForge bug #606303 . . . web-pages were not valid html-4.0.
- Fix SourceForge bug #593958 . . . missing-values should be ignored if they occur in intermediate results of RPN calculations.
- Fix SourceForge bug #600395 . . . won't compile with recently released version (3.2) of GCC compiler.
- Fix SourceForge bug #600233 . . . segfaults if some RPN operators are used on stacks that do not contain enough entries (e.g. `show {rpn cosh}`).

13.1.2 Version 2.10

13.1.2.1 Version 2.10.1 [2002 Jun 1]

Bug Fixes

- Fix SourceForge bug #562911 . . . won't build with '`gcc-3.0`' compiler. This was reported by the Debian hppa builder as Debian bug #148572.
- Fix SourceForge bug #562558 . . . `draw title` terminates program if have non-positive data and had initially specified log axes.
- Fix SourceForge bug #562014 . . . won't build if `popt` library is unavailable. This was reported by the Debian ia64 builder as Debian bug #148493.
- Fix SourceForge bug #558463 . . . in HTML docs, the "press" margin tag was misdirected.
- Fix SourceForge bug #562017 . . . parser fails with DOS end-of-line.
- Fix SourceForge bug #562113 . . . `new page` postscript error in '`gv`' viewer.

13.1.2.2 Version 2.10.0 [2002 May 20: Victoria Day]

New Features

- In the documentation, change the names of some variables to be clearer: `ll_x` is now written `xleft`, etc.
- Add RPN binary operators `and`, `or` for logical operations Section 10.9.4 [Binary Operators], page 162, along with negation operator `not` Section 10.9.5 [Unary Operators], page 164.
- Add `draw arc` command (Section 9.3.9.1 [Draw Arc], page 82).
- Add `set x axis labels` (Section 9.3.41.47 [Set X Axis], page 130) and `set y axis labels` (Section 9.3.41.54 [Set Y Axis], page 133) commands.

- Permit specification of `pt` units for `draw label` (Section 9.3.9.22 [Draw Label], page 89), `draw box` (Section 9.3.9.7 [Draw Box], page 83), `draw symbol at` (Section 9.3.9.29 [Draw Symbol At], page 91), and `draw line from` (Section 9.3.9.23 [Draw Line From], page 89).
- Add `set clip to curve` (Section 9.3.41.6 [Set Clip], page 116) command. *Caution:* this needs extension, and may have a bug if called twice in succession [but is this with an intervening `set clip off`]
- Add `group` and `end group` commands, in preparation for SVG output. So far these commands do nothing, and are basically just a signal that users should not create commands with these names since Gri will need them soon.
- Add `..xinc..` and `..yinc..` builtin variables (Section 4.2 [Axis Scaling], page 16).
- Make the `open` command accept URLs as filenames (Section 9.3.28 [Open], page 102).

Removed Features

- Remove `gri -repair old.ps new.ps` since (a) it was only half functional, (b) it was hard to code in the new scheme of argument-processing and (c) the author was not aware of anybody every having used it.
- Make the `-chatty` commandline option require a value (Chapter 3 [Invoking Gri], page 7).

Bug Fixes

- Fix SourceForge bug #552009 (`rpn` can segfault if `!=` operator is written as `=!`)
- Fix SourceForge bug #546109 (bounding box wrong if postscript clipping used)
- Fix SourceForge bug #514495 in which `set clip to curve` failed to handle missing values in the curve.
- Fix SourceForge bug #450465 (`create columns from function` was broken).

13.1.3 Version 2.8

13.1.3.1 Version 2.8.7 [2002 Apr 03]

- Fix SourceForge bug #482120 (`regress` ignores data weights)
- Fix SourceForge bug #508657 (missing backslash in drawing undefined synonyms)
- Fix SourceForge bug #523450 (log axes detect non-positive values too late)
- Fix SourceForge bug #521045 (installation/compilation with Solaris Workshop V6 Update 2 compiler)

13.1.3.2 Version 2.8.6 [2002 Feb 14]

- Fix SourceForge bug #513002 (minor documentation error in `set clip`)
- Fix SourceForge bug #509592 (HTML docs had midordered indices)
- Fix SourceForge bug #506534 (map axes give wrong minutes in negative regions). Thanks, Brian, for the patch on this!
- Fix SourceForge bug #508088 (grimode: gv should update, not be relaunched)
- Fix SourceForge bug #506490 (`-v` commandline option returned wrong version number)

13.1.3.3 Version 2.8.5 [2001 Dec 13]

- Fix SourceForge bug #492472 (`inf` rpn operator caused segfault)

13.1.4 Version 2.8.4 [2001 Oct 4]

- Fix SourceForge bug #467973 (`gri -version` gave wrong version number, breaking the Emacs Gri mode.)
- Fix SourceForge bug #468014 (`draw grid` disobeyed pencolor.)

13.1.4.1 Version 2.8.3 [2001 Oct 1]

- Fix SourceForge bug #462243 (endian problem in Rasterfile images + reading problem in PGM images).

13.1.4.2 Version 2.8.2 [2001 Sep 19]

- Fix SourceForge bug #461444 (wouldn't compile with the `mingw32` compiler, for windows 32 machines).
- Fix SourceForge bug #454557 (wouldn't compile with the pre-release version 3.0.1 of the GNU c++ compiler, in the case in which netCDF was already installed)

13.1.4.3 Version 2.8.1 [2001 Sep 7]

- Fix SourceForge bug #450465 (`create columns from function` was broken).
- Fix SourceForge bug #454557 (wouldn't compile with the pre-release version 3.0.1 of the GNU c++ compiler)

13.1.5 Version 2.8.0 [2001 Jul 30]

New command syntax

- Add `unlink` command as a unix-familiar way to delete files (Section 9.3.51 [Unlink], page 141).
- Add `set page size` command to clip to a given page size (Section 9.3.41.36 [Set Page], page 127).
- Add `substr` RPN operator to permit extraction of sub-strings (Section 10.9.3 [Tertiary Operators], page 162).
- Add `default` possibility for the `set x name` and the `set y name` commands.
- Add Perl-like ability to put underscores in numerical constants, to guide your eye. These underscores are ignored by Gri, so that for example `.v. = 1_000` and `.v. = 1000` are equivalent as far as Gri is concerned.

Emacs mode (Chapter 12 [Emacs Mode], page 191)

- Give ⟨M-Tab⟩ the ability to complete builtin variables and synonyms as well as commands
- Add "idle-timer minibuffer help" to display defaults for builtin variables under the cursor.
- Add fontification of builtin variables *only* if they are spelled correctly.

Developer changes

- Add `make source-arch-indep` target in sources. This will build a source tar file in which all the architecture-independent material (documentation in HTML, postscript and Info formats) is pre-made. This makes it easier to install gri on a host that doesn't have TeX and ImageMagick installed.
- Complete the port to IBM's compiler for their AIX operating system.

13.1.6 Version 2.6

13.1.7 Version 2.6.4 [2001 Jul 9: Dan's birthday]

- Fix SourceForge bug #435688 (remnants of `polar` column type, no longer supported, remained in documentation)
- Fix SourceForge bug #435603 (`set dash` produced broken PostScript)
- Fix SourceForge bug #431114 (`-0` could appear on axes)

13.1.8 Version 2.6.3 [2001 Jun 22]

- Fix SourceForge bug #433250 (`draw symbol` ignored dashing state sometimes)
- Fix SourceForge bug #432549 (contours sometimes unlabelled)
- Tweak internal coding for compilation on AIX compilers.

13.1.8.1 Version 2.6.2 [2001 May 19]

- Fix SourceForge bug #425174 (synonym interpolation broken on e.g. `show "[\syn]"`
- Fix SourceForge bug #425175 (`while !..eof..` acted ignored end of file)

13.1.8.2 Version 2.6.1 [2001 May 10]

- Fix SourceForge bug #420499 (gri-mode.el compatibility issues with emacs-21; Mostly bad old code.)
- Fix SourceForge bug #421076 (byte-compiled gri-mode.el has broken IMenu support; Affects Debian package.)
- Fix SourceForge bug #419599 (wouldn't compile under GNU g++ 3.x compiler)
- Fix SourceForge bug #418065 (documentation mentions back-tic notation, which is not available)
- Fix SourceForge bug #417333 (vague error message `RPN string operator`)
- Fix SourceForge bug #415277 (make fails on MSDOS)
- Fix SourceForge bug #415149 (`file.cc` parse error on MSDOS)
- Fix SourceForge bug #414520 (`draw symbol ... at` should automatically produces axes unless the location is in `cm` coordinates)
- Fix SourceForge bug #414010 (items in the html concept index were in an odd order)
- Fix SourceForge bug #413986 (`~username` was broken in `open`)
- Fix SourceForge bug #411904 (`/` was ugly in math mode)

13.1.8.3 Version 2.6.0 [2001 April 1]

- Permit `rewind` to take a filename.
- Make `open` set `\.return_value.` to the full pathname of the file that was opened.
- Add `set path` command (Section 9.3.41.39 [Set Path To], page 129).
- Remove functioning of `GRIINPUTS` environment variable, since this is more cleanly handled with the newly added `set path to` command (Section 9.3.41.39 [Set Path To], page 129).
- Remove `\.awk.` synonym, which was deemed to be unhelpful. Users with many scripts that use this variable might wish to put a line like

  ```
  \.awk. = "gawk"
  ```

  in their '`~/.grirc`' file.
- Change the format of images in the PostScript output file, as a workaround for a bug in the `ps2pdf` program.
- Add "ampersand" (`\&` and `\&&`) syntax to permit newcommands to look up the name and nesting level of changeable arguments (Section 10.11.5 [Changeable Command Arguments], page 177).
- Add "at-sign" (`@`) syntax for aliases (Section 10.5.4 [Alias Synonyms], page 155).
- Add ability to embed newlines in `show` commands with the `\<<` sequence (Section 9.3.42 [Show], page 135).
- Add ability to embed TAB characters in `show` commands with the `\>>` sequence (Section 9.3.42 [Show], page 135).
- Make various `read` commands (Section 9.3.33 [Read], page 106) able to decode synonyms as well as variables and simple numbers.
- Add `strlen` RPN operator (Section 10.9.5 [Unary Operators], page 164).
- Add `default` option to `set x format` and `set y format` commands.
- Add `new postscript file` (Section 9.3.27 [New Postscript File], page 102) command.
- Switch email list from `majordomo` to GNU `mailman`.
- No longer remove comments from data lines that are read. This served little function, interfered with recent user code, and could be accomplished by reading through a `sed` pipe in any case.
- Make the first element of the `argv` array be the name of the command-file. (This makes Gri consistent with languages such as C.)
- Add chapter on test suite (Chapter 16 [Test Suite], page 233).
- Let newcommands have changeable arguments (Section 10.11.5 [Changeable Command Arguments], page 177).
- Remove `-s` as an abbreviation for the commandline option `-superuser`.
- Remove the `:` syntax from the commandline options.
- Add commandline option `-output PS_file_name`.
- Add `assert` command (Section 9.3.1 [Assert], page 74).
- Add test suite (mainly for developers).
- Add `sleep` command (Section 9.3.44 [Sleep], page 137).
- Remove command `show hint of the day`, since I no longer permission to use cgi-bin to provide the hints via web-server.
- Permit indexing of synonym words with variables, in addition to constants.

- Fix bug in interpolating synonyms in the test-expression of `while` loops.
- Put source on the Source Forge open-source development website, at `http://gri.sourceforge.net`. Gri users can benefit from this site in may ways. First, it is a great way to keep track of new versions. With a mouse-click you can cause SourceForge to email you whenever a new version of Gri is released. Second, SourceForge has newsgroups (`http://sourceforge.net/forum/?group_id=5511`) and email lists (`http://sourceforge.net/mail/?group_id=5511`) devoted to Gri. These are archived, so that you can stop monitoring them for a while and then go back and see what you've missed. Third, Source Forge has a powerful bug-tracking facility. (`http://sourceforge.net/bugs/?group_id=5511`) With this you may check for existing bugs reported by users, submit new bug reports, and also track the process of bug removal (e.g. optionally receiving emails when the bug is removed).
- Add `set colorname` command (Section 9.3.41.8 [Set Colorname], page 118).
- Add `source` command (Section 9.3.46 [Source], page 138).
- Add RPN operators `wordc` (Section 10.9.6 [Solitary Operators], page 168) and `wordv` (Section 10.9.5 [Unary Operators], page 164) for accessing the words in the present Gri command.
- Add RPN operators `argc` (Section 10.9.6 [Solitary Operators], page 168) and `argv` (Section 10.9.5 [Unary Operators], page 164) for accessing the command-line arguments used when Gri was invoked from the operating system.
- Add automatic support for compressed data files. So far, this only works with gzipped files (Section 9.3.28 [Open], page 102).
- Add two new RPN operators, `file_exists` and `directory_exists` (Section 10.9.5 [Unary Operators], page 164).
- Reorganize parts of manual (e.g. changing the section about the Emacs '`gri-mode.el`' into a chapter, with screen snapshots).
- Improve the HTML form of the manual (e.g. color-code the Gri syntax in examples, provide access to all the indices, use Jpeg format, et.).
- Add several new features to `gri-mode.el`:

  1) Add a menubar pull-down menu listing all Gri commands, allowing the user to get `Help` or `Info` on any command or `Insert` it in the current Gri file.

  2) Add a menubar pull-down entry to `Perform` to set `gri-run` settings such as flag options passed to gri.

  3) Gri info file can have **.info** extension now.

  4) Made `gri-apropos` an alias for `gri-help-apropos`.

  5) gri-mode now uses the customize interface (See `gri-customize`).

  6) The `~/.gri-syntax` file has changed format. As a side-effect, spaces are used instead of hyphens to display Gri commands, and one can now select a command with the mouse in the **Completions** buffer.

13.1.9 Version 2.4

13.1.9.1 Version 2.4.4 [2000 May 7]

Change so that only a warning is printed if mathematical operations are requested on empty arrays (e.g. `y += 10`). Previously, an error was reported and Gri exited with an error condition, which is bothersome in scripts that rely on successful completion of the Gri job.

Port to the BeOS operating system.

13.1.9.2 Version 2.4.3 [2000 Apr 1]

Change location of all files, to be consistent with Redhat convention. Previously, things were in `/opt/gri`; now they are in more standard locations.

13.1.9.3 Version 2.4.2 [2000 Mar 25]

Remove bug in which `convert grid to image` produced incorrect images, visible as a patchy appearance with coarse grids.

13.1.9.4 Version 2.4.1 [2000Jan 31]

Remove bug in which `convert image to grid` failed to take note of the gri minimum and maximum, so that contouring of the grid was not possible for grids created from images.

13.1.9.5 Version 2.4.0 [2000 Jan 05]

Add `set input data separator`. Make `read columns` work with various input separators. Make `read .x.`, etc, work with various input separators.

13.1.10 Version 2.2

13.1.10.1 Version 2.2.6 [1999 Nov 25]

Make web-based manual easier to read by putting a light-grey background under sample code.

13.1.10.2 Version 2.2.5 [1999 Nov 10]

Fix bug in RPN calculations that prevented using a negative exponent.

13.1.10.3 Version 2.2.4 [1999 Nov 7]

Add `set font encoding`, and also change the encoding to ISO-Latin-1. (This doesn't hurt old code since Gri didn't make any claims to handle characters outside the normal printing-set before anyway.)

Fix bug in which there were 4 dead links in the HTML version of manual.

Clean up some problems with Debian distribution (thanks, Peter Galbraith!).

13.1.10.4 Version 2.2.3 [1999 Jun 30]

Fix bug in which word-of-synonym (e.g. `\[0]mysyn`) was not detected correctly (thanks to bug report from Kazuhiko Nakayama in Japan)

13.1.10.5 Version 2.2.2 [-]

Clean a few spelling and cross-reference errors in documentation.

13.1.10.6 Version 2.2.1 [1999 Mar 31]

For debian, properly locate the `netcdf` library, if it is installed.

Remove remnants of old commands for polar axes.

Correct error in which the right-most and upper-most pixel of images created by `convert grid to image` may be blank (or not, depending on roundoff error) under certain conditions of exact matchup between grid spacing and image spacing.

Don't create PostScript file if the commandfile is non-existent, or if there were errors on the commandline.

13.1.10.7 Version 2.2.0 [1999 Mar 25]

First debian release. Versions exist for intel, alpha, 68K and powerpc.

13.2 Unstable Stream

13.2.1 Development Version

The list below shows changes that have been made in the development version of Gri, i.e. the version on CVS at `http://gri.sourceforge.net`. Some or all of these features may enter the next stable stream of Gri, but it is important to note that *anything* that's listed here may be changed without notice! The main exception is bug fixes, which will enter the stable stream unless they are found to create new bugs.

13.2.1.1 New Features

- n/a

13.2.1.2 Removed Features

- n/a

13.2.1.3 Bug Fixes

- Fix SourceForge bug #618041 ... solaris compile error, relating to the subroutine `gethostname()` in the '`startup.cc`' source code file.

13.2.2 Plans

Some plans for future versions are given below. A more extensive list, together with target timescales and notes on the progress towards completion, is available in the to-do list at

the development site (`http://gri.sourceforge.net`), which is also the place to go if you have suggestions for other changes to Gri.

13.2.2.1 High Priority

1. Add ability to generate SVG (scalable vector graphics) output. This file type can be edited with GUI-based tools, making it easy for users to put the final touches on graphs "manually." This feature will require a great deal of work, mainly to do with grouping of graphical elements, and so I'll need to rely on advice from users.
2. Switch to `popt` library for the processing of command-line options. (Docs on this are at `http://cvs.gnome.org/lxr/source/popt`.)
3. Change `set axes style` from number-based selection to word-based selection, and print a deprecation warning if the number-based syntax is used.
4. Switch from `tmpname` to `mkstemp` subroutine, for temporary filenames.
5. Add readline support, to handle history, command-line editing, etc, for interactive usage (Docs on this are at `http://cnswww.cns.cwru.edu/~chet/readline/readline.html`.)

13.2.2.2 Medium priority

1. Add optimal interpolation as another method for gridding data (Section 9.3.4.1 [Convert Columns To Grid], page 75).
2. Add ability to read PNG images (Section 9.3.33.7 [Read Image], page 111) (Docs on this are at `http://www.libpng.org/pub/png/`.)

13.2.2.3 Low priority

1. Add non-RPN mathematics.
2. Add graphical output to the screen as commands are executed.
3. Add support for both little-endian and big-endian binary data.

13.3 Deprecated Commands

- *version 2.9.0*
  - Replace `set y axis label horizontal` with `set y axis name horizontal`.
  - Replace `set y axis label vertical` with `set y axis name vertical`.

14 Installing Gri

14.1 Unix Installation

14.2 Archiving Old Versions

Gri, like other complex programs, sometimes changes in such a way as to break old scripts. This is less so of changes since about 1998 or so, since the syntax became pretty firm about that time.

Still, disk costs are so cheap that you would be well-advised to keep a backup version of Gri, whenever you update. This is pretty simple; you need to keep a copy of the executable and the library file, and you need to write a tiny shellscript that calls this particular executable with this particular library file. For example, you might do the following

```
mkdir -m 755 -p           /usr/local/share/gri/2.12.18
cp /usr/bin/gri           /usr/local/share/gri/2.12.18
cp /usr/share/gri/gri.cmd /usr/local/share/gri/2.12.18
```

and then create a shellscript called '`/usr/local/bin/gri-2.12.18`' containing

```
#!/usr/bin/sh
/usr/local/share/gri/2.12.18/gri \
    -directory /usr/local/share/gri/2.12.18 \
    "$@"
```

to invoke this version of Gri.

To be able to access this old version of gri from within the Emacs gri-mode, you would reset the Emacs variable `gri*directory-tree` like so in the file '`~/.emacs`' (Section 12.3.2 [Step 2], page 197 in Section 12.3 [Installing gri-mode.el], page 196)

```
(setq gri*directory-tree
    '("/usr/local/share/gri/" "/usr/share/gri/"))
```

14.2.1 Installation on Linux computers

Installation is easy on RedHat and Debian flavors of linux. Versions are available from the official distributions and also at '`http://sourceforge.net/project/showfiles.php?group_id=5511`'

The RedHat version supports intel platforms only, while the Debian version supports intel, alpha, 68K and powerpc. Installation follows the normal method for these distributions ... if you have the distribution, you surely know what to do. If you don't, you may want to switch when you see that you can install Gri by clicking an icon in a window, or by typing, e.g. in RedHat linux:

```
rpm -i gri-2.12.18
```

Debian-linux installation may be done with any of several GUI-based tools, or with the following system command:

```
dkpg -i gri_2.12.18-1_i386.deb
```

or via a properly configured `apt` interface that will fetch the latest version from the net:

```
apt-get install gri
```

14.2.2 Pre-compiled unix versions

Pre-compiled versions of Gri exist for various computers, e.g. SPARC (both sunOS and solaris), IBM-RS, Sequent, and Linux. The following instructions, which assume a version number 2.12.18, and a SPARC solaris machine, show how to install these.

The compressed tar file, with a name like '`gri-binary-SunOS5-2.12.18.tar.gz`' (in this example a Sun OS 5 binary is assumed, with gri version number 2.12.18) contains a directory which holds the executable Gri file, '`gri`', the library file '`gri.cmd`', an instructions file, '`README`', and a Makefile to install Gri.

Here's how to install Gri, starting with this tar file:

1. First, uncompress (unzip) the file, and un-tar it, by doing

   ```
   gunzip -c gri-binary-SunOS5-2.12.18.tar.gz | tar xvf -
   ```

   or

   ```
   zcat gri-binary-SunOS5-2.12.18.tar.gz | tar xvf -
   ```

   This will yield a new directory, with a name like '`gri-binary-SunOS5-2.12.18`', which contains the indicated files.
2. Follow the instructions in the '`README`' file to install Gri.

14.3 Compilation on Unix computers

The following steps indicate how to compile Gri on unix computers. The procedure is quite standard.

Requirements

You'll need a C++ compiler that is modern enough to handle 'templates' (i.e. almost any compiler from 1998 onward). Don't worry – if your compiler isn't new enough, you'll see that in a minute or two!

You'll need TeX and texinfo to make the info files, and optionally the netCDF library (if you wish gri to be able to read netCDF binary data files). To make the HTML manual, you'll need imagemagick, info, gs and its fonts.

On Debian GNU/Linux systems, the required packages are listed as Build-Depends in the '`control`' file found in the '`debian`' directory, to which you must add the package '`build-essential`'.

Unpack the source

Type

```
gunzip gri-2.12.18.tgz
tar xvf gri-2.12.18.tar
```

(or similar commands) to uncompress and untar the contents. This will yield a new directory named '`gri-2.12.18`' which contains many files.

Move to the Gri directory

```
cd gri-2.12.18
```

Configure your compiler

Next you must "configure" the Gri source files. During this step, a series of tests will be made about your operating system and your compiler. Most of these tests need no interaction from you, but there is one overall choice that you may wish to make: the place on your filesystem where Gri (and many associated

library and documentation files) will reside. To get the default installation, with files residing within the directory '`/usr/local`', type

```
./configure
```

at this time. If you'd rather the files go into another location, run the `configure` script differently, e.g. to get the Gri files to reside within the '`/opt`' directory, type:

```
./configure --prefix=/opt
```

In response, you'll see the results of several tests of the properties of your operating system, your C++ compiler, etc. Normally you can ignore these results.

As an example, typing `./configure` without a `--prefix` option yields the directory tree (`...` indicates several files not displayed in this list, for brevity). In this example, the most up-to-date version is 2.12.18, but a previous version 2.12.17 has also been retained. (Note that only one copy of the documentation is retained; this is all that's needed, since old versions are documented there as well as new versions.)

```
/usr/local
|-- bin
|   |-- gri -> gri-2.12.18
|   |-- gri-2.12.18
|   |-- gri-2.12.17
|   |-- gri_merge
|   `-- gri_unpage
|-- info
|   |-- ...
`-- share
    `-- gri
        |-- 2.6.0
        |   |-- gri.cmd
        |   |-- license.txt
        |   |-- logo.dat
        |   `-- startup.msg
        |-- 2.4.0
        |   |-- gri.cmd
        |   |-- license.txt
        |   |-- logo.dat
        |   `-- startup.msg
        `-- doc
            |-- examples
            |   |-- ...
            `-- html
                |-- ...
                |-- resources
                |   |-- ...
                `-- screenshots
                    |-- ...
```

Trouble-shooting 1: If the permission of the '`configure`' file is wrong, you'll get an error like `Permission denied`; if so, try typing `sh ./configure` to run it in the Bourne shell. If that fails, you are going to have to do some old-fashioned work! Start by copying the generic Makefile called '`Makefile.generic`' into

'`Makefile`', and try the following steps, perhaps editing the '`Makefile`' if you run into errors.

Trouble-shooting 2: Gri uses a C++ feature called 'templates'. Unfortunately, templates are handled in different ways by different compilers. At least as of Spring 1997, the GNU compiler, vsn 2.7.x (used by many Gri folks) has problems with templates. Therefore the configure script will check to see if you are using the GNU c++ compiler, and if you are it will check whether the ("template repository") compiler flag `-frepo` is known on your machine. If it is not, an alternative method of templates will be used. But if it is, you'll be asked, for confirmation, whether you wish to use the `-frepo` flag. On many machines (e.g. Solaris) you should answer `n` to this question. The prompt will explain. Also, note that you can avoid the prompt by running configure as either of the two below:

```
./configure --enable-frepo
./configure --disable-frepo
```

(Such switches will be ignored unless you're using the GNU compiler.)

Trouble-shooting 3: If optional system libraries like the netCDF library, if it exists, are installed in nonstandard places, you might have to change the unix environment variable `LD_LIBRARY_PATH`. For example, on my machine the `netcdf` library is not installed in '`/usr/lib`', as the `configure` script assumes, but rather in '`/usr/local/share/netcdf/lib`'. Therefore I have the following line in one of my startup files:

```
export LD_LIBRARY_PATH=/usr/lib:/usr/local/share/netcdf/lib
```

Compiling Gri

Now compile Gri by typing

```
make
```

Testing Gri

Type

```
make test
```

to do some tests on the version of Gri that you just compiled. If no errors are reported, you may go to the next step.

Installing Gri

Assuming compilation succeeds, install '`gri`' and the ancillary file '`gri.cmd`', by typing

```
make install
```

If you wish to see where files where be installed, first try a dry run typing

```
make -n install
```

Cleaning up

Once these things are done, you may type

```
make clean
cd doc ; make clean
```

to clean up some files. Of course, you could just erase the whole source, but the source is probably worth a penny of hard-drive space, isn't it?

14.4 Compilation on x86 (PC-style) Computers

Versions exist for MSDOS, windows, and Linux operating systems. (Actually, the windows version is just the MSDOS version, which can be run inside an msdos window within windows-95, windows-NT, etc.)

14.4.0.1 MSDOS Operating System

To compile and install Gri under MSDOS, do this:

1. To begin, install `DJGPP V2`, if it is not on your system already. It can be found at `http://www.delorie.com/djgpp`
2. Uncompress and extract from gri source package (Section 14.3 [Uncompiled Unix], page 222).
3. Type `make -f Makefile.dj2` to compile. If it fails, you might have to edit the file '`Makefile.dj2`' to match the characteristics of your system. Please inform the author, Dan Kelley at Dan.Kelley@Dal.CA, if you think your modifications might be useful to others.
4. Type `make -f Makefile.dj2 install` to install it (normally on the `C:` drive).

If you encounter problems, read the first few lines of the Makefile (i.e. the file '`Makefile.dj2`') for hints on things to try. For example, in the present version of '`Makefile.dj2`' these hints are given:

1. There is a good chance that this Makefile will work as is, so try that first.
2. If you have the '`netcdf`' library (used for certain types of atmospheric and oceanographic datasets), then un-comment and possibly edit the appropriate `NETCDF_...` lines below, as instructed by the comments preceding these lines.
3. If you don't want Gri inserted in the directory '`c:/gri`', edit the `instdir = ...` line below.
4. If you get error messages about the '`stdcxx`' library, edit the `LIBS` line below, rewriting '`-lstdcxx`' as '`-lstdcx`'.
5. If you get compilation errors relating to `time` or to `ftime`, try putting the token `-DHAVE_FTIME=1` in the list of similar token in the `DEFS = ...` line. For consistency (basically, so the author can help you if you do this), put it right after the `-D_GRI_=1` token.

To view the output, use a PostScript viewer such as GSview. (`http://www.cs.wisc.edu/~ghost/index.htr`

14.4.0.2 LINUX Operating System

Linux is a good emulation of unix, and it is free. Gri for linux is compiled and installed according to the normal unix instructions. The compiled version is with a name like `gri-binary-solaris-2.1.10.tar.gz`; treat it as in other unix systems (Section 14.3 [Uncompiled Unix], page 222).

14.5 Compilation under OS/2

Gri compiles, using the gcc compiler under OS/2, provided that the included `Makefile.os2` is used. Be sure to edit the first few lines to change filenames as required, especially taking care to account for whether the netCDF library is installed, etc.

14.6 Compilation in Macintosh OS X

The OS X system provides a BSD unix that suites Gri very well. With the (free) developer package, it also provides a very up-to-date version of the `gcc` compiler. Thus, installing Gri on Macintosh can be done using the normal Unix instructions.

But there are also easier ways. Gri is compatible with Fink and Darwinports, the two popular packaging systems on OS X. If you use OS X and do not have Fink or Darwinports installed, then you should probably install one, or both. Each distribution has strengths, and each has weaknesses, and it is difficult to provide a firm recommendation between the two.

Caveat. As of mid-2007, neither distribution appears to handle package dependencies as well as is done by popular linux distributions. For example, in working through the steps listed below, the author found that his Darwinports system had a problem with a system library that handles internationalization. The "update" operation of the system was insufficient to solve the problem, and so it was necessary to do some web searching to find a patch. The patch failed, but another search revealed a second (hand-edit) patch that got it working. In excess 4 CPU hours were required to rebuild the packages that were broken.

If you'd like to build a local Darwinports version of Gri, to get the latest version instead of whatever version is provided by Darwinports, follow these steps:

- Download the source from CVS at SourceForge. (If you don't know what the previous sentence means, you will quite likely have difficulties with the other steps.)
- Visit the `darwinports` directory of the newly-created directory tree, and type

  ```
  sudo port -d -v build
  ```

  to build it. This will take several minutes, during which you may find it helpful to do a search on "darwinport build". For example, the O'Reilly page (`http://www.oreillynet.com/pub/a/mac/2004/04/09/darwinports.html?page=3`, last checked in July 2007) is very good. Note that you will be building from the source that is stored on SourceForge, not from the Darwinports source. That's the trick of issuing the `build version` of the `port` command.
- Do a "destroot" operation:

  ```
  sudo port -d -v destroot
  ```
- Install it:

  ```
  sudo port -d -v install
  ```

  Note: if you already have an older version of Gri in the Darwinports system, you must first issue the command

  ```
  sudo port deactivate gri
  ```

14.7 Compilation under BeOS

The BeOS system compiles Gri cleanly without modification; just follow the instructions for unix.

15 Bugs

15.1 Known bugs

To get a list of known bugs, in the present or past versions, visit `http://sourceforge.net/tracker/?atid=105
id=5511&func=browse`.

15.2 Reporting Bugs

Bug reports help make Gri more reliable and useful for all users, so please report any bugs you find. The scheme will be familiar to you, if you've participated in open-source work before.

1. Visit the bug-tracking part of the Source Forge Gri development site, at `http://sourceforge.net/bugs/?group_id=5511` to see whether your bug has already been reported. You may find that it has been reported and fixed already in a new version, in which case you should archive your existing Gri version and upgrade. If the bug has been reported and **not** fixed, then you may still wish to add a supplemental bug report, so the author knows that this bug is of concern to you as well as the others. (Plus, reporting it this way ensures that you'll receive an email when the bug is fixed.)
2. Debugging is made easier if the problem is reduced in scope. Therefore, please reduce your application and your data to the simplest case that produces the error.
3. If you know how, please try to find the bug yourself. After all, Gri is open-source for a reason! The procedure is described in the next section (Section 15.3 [Killing Bugs], page 227).

15.3 Killing Bugs

15.3.1 Software that you'll need

This section is intended to help you find and kill bugs yourself, by indicating how the author does this work. Since Gri is open-source, all users are invited to try to kill bugs themselves!

If you know nothing of C or C++, you may as well not read further, since there is little chance of your making progress. On the other hand, experienced programmers won't need any of the advice I give below.

You'll need the Gri source and a C++ compiler. It also helps if you have `gdb`, the GNU debugger, installed; the instructions below assume that's the case. Also, the instructions assume that you're using the Emacs editor, and running `gdb` from within Emacs. Otherwise, you'll want to glance at the documentation on `gdb` to see how to use it in standalone mode.

15.3.2 Debugging at a glance

The list below is a sketch of what you might try, and in what order.

- Check the bug list to see if other users have found your bug, and also to see if there is a workaround.

- Try a more recent version of gri. If it works, you might wish to archive the version you have at the moment and upgrade.
- If you suspect your bug has something to do with system calls, as in the `system` command (Section 9.3.50 [System], page 139) or as in piped input files (Section 9.3.28 [Open], page 102), you should re-run the script with `gri -superuser8` instead of with `gri`. This will cause Gri to print out all commands that it is handing over to the operating system; you may see the error that way. (Hint: it may help to interactively cut/paste the commands into your OS shell to see what the action of the command is.)
- If your bug results in early termination, you should run Gri inside a debugger (e.g. GDB, assumed henceforth). When the program terminates, type `where` to see where termination occured. Often this will give a clue. In many cases, early termination results from faults in memory allocation. To check memory allocation, you'll need to recompile Gri, linking it against a debugging memory allocator. Many such tools exist; see comments in the `Makefile` for a hint at how to use a popular one, called "Electric Fence."
- If your bug does not result in early termination, you may find the best scheme is to trim your example down as much as possible, and then run Gri inside the GDB (or other debugger) so that you can monitor program execution. The next section explains this in detail.

15.3.3 A debugging Example

Let's take a recent bug as an example. Peter Galbraith found that the gri script

```
set color hsb 0.999 1 1
draw box filled 2 2 3 3 cm
set color hsb 1.000 1 1
draw box filled 4 2 5 3 cm
```

produced odd results in a previous version of Gri; the color patches should have been of nearly the same color, but the first one was red, as expected, and the second was magenta.

The list below shows how I found Peter's bug. Experienced C or C++ programmers will find all of this very familiar, and will really only need to read item 6 of the list below, since that's the only action that is really specific to Gri. (Note: for display purposes, I've broken some of the lines in the files into two lines in this list.)

1. Copy the above script (called '`test.gri`') into the Gri source directory, and the script into an Emacs buffer. **Note:** all the following steps are done within Emacs, and the items in parentheses are the Emacs keys to get the indicated actions.
2. If you're working from a pre-compiled version, you'll need to get the source first and do a compile yourself (Section 14.3 [Uncompiled Unix], page 222). Then do a `make tags` command (type this to the unix shell) to create a so-called "tag" table.
3. Run Gri in this emacs buffer (*C-cC-r*) noting from the postscript window that pops up that the colors are, indeed, mixed up.
4. Load up the `gdb` debugger by typing *M-x gdb gri*. This will open a new Emacs buffer in which you may type commands. We'll be switching back and forth between this buffer and various source files.
5. Reasoning that the error probably occurs at `set color` or `draw box`, try replacing the latter by a command such as `draw label "hi" at 3 3 cm`. The color is still wrong, indicating that it is the `set color` command that has the problem.

6. Next, we must find where the C++ code corresponding to the `set color` command resides. As it turns out, all `set` commands are defined in the source file `set.cc`, and this command is defined in a subroutine called `set_colorCmd()`. But the author knows this – how would you? The answer is to look in the '`gri.cmd`' file, for the `set color` command. (Search for the string '`set color` .) Then read down to see the body of the command, enclosed in braces; you'll see

```
{
    extern "C" bool set_colorCmd(void);
}
```

   which indicates that the subroutine name is `set_colorCmd()`.

7. Next we need to edit this subroutine to see what it is doing. There are several ways to find it (e.g. `grep` through the source files), but the easiest is to use the "tags" feature of Emacs, by typing *`M-. set_colorCmd`*. This will bring you to the indicated subroutine.

8. Have a look through this subroutine to see what it is doing. It looks very much like many other Gri subroutines. A check is done on the number of words provided to the command, in the

```
switch (_nword) {
```

   line. (That's line '`set.cc:484`' at the moment – but it may be different by the time you read this file, if I've changed it!) We are calling it with 6 words (`set color hsb 1.000 1 1`), so move down to the line

```
case 6:
```

   and you'll see that there is an `if` statement seeing whether this word is `rgb` or `hsv`. These statements are checking `_word[2]`, which is the third word of the command. (In Gri, as in C, words start at zero. Thus, for this command, `_word[0]` is `set`, `_word[1]` is `color`, and `_word[2]` is expected to be either `rgb` or `hsb`. We are having problems with the `hsb` style, so we'll move down to that code. The code that's being executed is as follows.

```
} else if (!strcmp(_word[2], "hsb")) {
        // 'set color hsb .hue. .saturation. .brightness.'
        double          hue, saturation, brightness;
        Require(getdnum(_word[3], &hue),
                READ_WORD_ERROR(".hue."));
        Require(getdnum(_word[4], &saturation),
                READ_WORD_ERROR(".saturation."));
        Require(getdnum(_word[5], &brightness),
                READ_WORD_ERROR(".brightness."));
        // Clip if necessary
        CHECK_HSB_RANGE(hue);
        CHECK_HSB_RANGE(saturation);
        CHECK_HSB_RANGE(brightness);
        gr_hsv2rgb(hue, saturation, brightness,
                &red, &green, &blue);
        PUT_VAR("..red..", red);
        PUT_VAR("..green..", green);
        PUT_VAR("..blue..", blue);
        c.setHSV(hue, saturation, brightness);
        _griState.set_color_line(c);
        if (_griState.separate_text_color() == false)
                _griState.set_color_text(c);
```

```
return true;
```

The `Require` lines are ensuring that we could decode the values of the variables `hue`, etc, from the commandline. Then we clip the range of these values. Then we convert from `hsb` color format to `rgb` color format, save the values of the colors in Gri variables with `PUT_VAR`, and then set the color with `c.setHSV`. Finally we save this color in the Gri "state" with `_griState.set_color_line(c)`.

As it turns out, Gri outputs all colors to the PostScript file in RGB format, so the we may well suspect that the problem is in the `gr_hsv2rgb()` line.

9. We have an idea where to look now, so let's go to the line just after it, in the editor, and insert a "breakpoint" there by typing *C-x SPC*. Then move to the `gdb` buffer and re-run Gri by typing

   ```
   run -directory . test.gri
   ```

   to run Gri on our script. Then, magic happens! Gri stops at the indicated breakpoint, and Emacs will display both the `gdb` buffer and the `set.cc` buffer. The latter has a margin indication telling what line were are on. You may now type `gdb` commands in the `gdb` buffer. In particular, type

   ```
   p hue
   ```

   to print the hue. Then type

   ```
   p red
   ```

   to see the red value. Then type

   ```
   c
   ```

   to continue running Gri. It will pause again. Check the hue and red values again, as above. If you like, play around with hue value in the Gri script '`test.gri`' and run gri again (type `r` in `gdb`). This seems to indicate that the conversion is working strangely.

10. To see how the conversion is done, clear the breakpoints by typing *`delete`* in the `gdb` buffer, then insert a breakpoint **before** `gr_hsv2rgb` is called. Then, run Gri again (*`r`* in the `gdb` buffer). When it stops just before this subroutine, type *`s`* to "step into" the subroutine. Then you'll see a conversion code from the (wonderful) textbook of Foley and Van Dam. You'll see

    ```
    void
    gr_hsv2rgb(double h, double s, double v,
               double *r, double *g, double *b)
    {
    h = 6.0 * pin0_1(h);
    s = pin0_1(s);
    v = pin0_1(v);
    int i = (int) floor(h);
    if (i > 5)
    i = 5; // Prevent problem if hue is exactly 1
    double f = h - i;
    double p = v * (1.0 - s);
            ...
    ```

    in the present version of Gri, but in the previous (buggy) version, the `if` statement was missing. Without this `if` statement, Gri produced wrong colors. With the statement, the colors are correct.

And so ends the example. You may wish to read the Foley and Van Dam textbook to see just what I'm doing in `gr_hsv2rgb`, but suffice it to say that the problem in the (older)

version of Gri was that `i` could take the value 6 if the hue was exactly equal to 1, and that was erroneous.

In reading the code, you may notice that it is formatted in a uniform way: the Kernighan and Ritchie scheme (from their classic C textbook), with 8-character indents. I get this by putting the following lines in the '`~/.emacs`' file, which is used to customize the Emacs editor:

```
(defun my-c-mode-common-hook ()
  (c-set-style "K&R")
  (setq c-basic-offset 8)
  )
(add-hook 'c-mode-common-hook 'my-c-mode-common-hook)
```

If you submit patches, please use the same format as I've done, so that I can more easily see the changes you've made.

16 Test Suite

The following test files are invoked by typing `make test`, after compiling Gri. They are provided here because they are examples of scripts that are **known to work** for the version of Gri described in this manual.

Heavy use is made of the `assert` command (Section 9.3.1 [Assert], page 74) in these test files. Thus the **code itself** demonstrates the features, instead of comments in the code. This is advantageous since comments tend to be incorrect!

Test file '`tst_IO.gri`':

```
show "doc/tst_suite/tst_IO.gri      ..." ...

# Test some I/O features.  NOTE: this will make _no_ sense
# to you unless you have a look at the test files!

# Read columns from file with newline at the end ...
open tst_IO_1.dat
read columns x y
close
assert {rpn ..num_col_data.. 2 ==}      " failed test 1-a"
assert {rpn x 0 @  1 ==}                " failed test 1-b"
assert {rpn x 1 @  2 ==}                " failed test 1-c"
assert {rpn y 0 @ 11 ==}                " failed test 1-d"
assert {rpn y 1 @ 22 ==}                " failed test 1-e"

# ... as above, but from a file without newline at the end.
open tst_IO_2.dat
read columns x y
close
assert {rpn ..num_col_data.. 2 ==}      " failed test 2-a"
assert {rpn x 0 @  1 ==}                " failed test 2-b"
assert {rpn x 1 @  2 ==}                " failed test 2-c"
assert {rpn y 0 @ 11 ==}                " failed test 2-d"
assert {rpn y 1 @ 22 ==}                " failed test 2-e"

# Read a line into a synonym.
open tst_IO_1.dat
read line \s
assert {rpn "\s" "1 11" ==}             " failed test 3-a"
read line \s
assert {rpn "\s" "2 22" ==}             " failed test 3-b"
close
open tst_IO_2.dat
read line \s
assert {rpn "\s" "1 11" ==}             " failed test 3-c"
read line \s
assert {rpn "\s" "2 22" ==}             " failed test 3-d"
close

# Read variable/synonym, in various orders.
open tst_IO_1.dat
read .a. .b.
assert {rpn .a.  1 ==}                  " failed test 4-a"
assert {rpn .b. 11 ==}                  " failed test 4-b"
read \a \b
assert {rpn "\a"  "2" ==}               " failed test 4-c"
assert {rpn "\b" "22" ==}               " failed test 4-d"
rewind
read .a. \b
assert {rpn .a.  1 ==}                  " failed test 4-e"
assert {rpn "\b" "11" ==}               " failed test 4-f"
read \a .b.
assert {rpn "\a" "2" ==}                " failed test 4-g"
assert {rpn .b.  22  ==}                " failed test 4-h"
close
open tst_IO_2.dat
read .a. .b.
assert {rpn .a.  1 ==}                  " failed test 4-i"
assert {rpn .b. 11 ==}                  " failed test 4-j"
```

Test file ‘`tst_control.gri`’:

```
show "doc/tst_suite/tst_control.gri ..." ...

# If statements
if 0
    assert 0 " failed test 1"
end if
if 1
else
    assert 0 " failed test 2"
end if

# Loops.

# Loop with if inside
.i. = 0
while {rpn .i. 3 >}
    .i. += 1
    if 1
    else
assert 0 " failed test 3"
    end if
end while
assert {rpn .i. 3 ==} " failed test 4"

# Loop inside if
if 1
    .i. = 0
    while {rpn .i. 3 >}
.i. += 1
    end while
    assert {rpn .i. 3 ==} " failed test 5"
else
    assert 0 " failed test 6"
    .i. = 0
    while {rpn .i. 3 >}
.i. += 1
    end while
    assert 0 " failed test 7"
end if

# Nested loops
.i. = 0
while {rpn .i. 2 >}
    .j. = 0
    while {rpn .j. 4 >}
.j. += 1
    end while
    .i. += 1
end while
assert {rpn .i. 2 ==} " failed test 8"
assert {rpn .j. 4 ==} " failed test 9"

show " passed"
```

Test file ‘`tst_rpn.gri`’:

```
show "doc/tst_suite/tst_rpn.gri     ..." ...

rpnfunction same - abs 1e-5 > #  Are numbers virtually same?

# Simple arithmetic
.a. = 0
assert {rpn .a. !}   " failed test 1.1"
.a. += 1
assert .a.    " failed test 1.2"
.a. += 1
assert {rpn .a. 2 same}   " failed test 1.3"
.a. *= 2
assert {rpn .a. 4 same}   " failed test 1.4"
.a. /= 4
assert {rpn .a. 1 same}   " failed test 1.5"
assert {rpn 2 1 - 1 same}                " failed test 1.6"
assert {rpn 2 1 + 3 same}                " failed test 1.7"
assert {rpn 3 2 * 6 same}                " failed test 1.8"
assert {rpn 4 2 / 2 same}                " failed test 1.9"


# Conversions (lower-case ok on input, but output is upper-case)
assert {rpn "aa" hex2dec  170 ==}        " failed test 2.1"
assert {rpn "AB" hex2dec  171 ==}        " failed test 2.2"
assert {rpn  63  dec2hex "3F" ==}        " failed test 2.3"
assert {rpn 193  dec2hex "C1" ==}        " failed test 2.4"

# Logic
assert {rpn 1 0 or}                      " failed test 3.1"
assert {rpn 0 1 or}                      " failed test 3.2"
assert {rpn 1 0 |}                       " failed test 3.3"
assert {rpn 0 1 |}                       " failed test 3.4"
assert {rpn 1 0 and not}                 " failed test 3.5"
assert {rpn 1 0 & !}                     " failed test 3.6"

# Logs, powers
.a. _= 10
assert {rpn .a. 0 same}   " failed test 4.1"
.a. = 2
.a. ^= 8
assert {rpn .a. 256 same}   " failed test 4.2"
assert {rpn -2 4 power 16 ==}            " failed test 4.3"
assert {rpn -2 3 power -8 ==}            " failed test 4.4"
assert {rpn -2 2 power  4 ==}            " failed test 4.5"
assert {rpn  2 3 power  8 ==}            " failed test 4.6"


# String operations
\a = {rpn "file" ".dat" strcat}
assert {rpn "\a" "file.dat" ==}   " failed test 5.1"
\sentence = "This sentence has five words"
\w1 = word 0 of "\sentence "
assert {rpn "\w1" "This" ==}   " failed test 5.2"
\w2 = word 1 of "\sentence "
assert {rpn "\w2" "sentence" ==}   " failed test 5.3"
assert {rpn 0 4 "hello" substr "hell" ==} " failed test 5.4"


# Q: will the below work on all OS?????
```

Test file ‘`tst_var_syn.gri`’:

```
show "doc/tst_suite/tst_var_syn.gri ..." ...
rpnfunction same - abs 1e-10 >          #  Are numbers virtually same?

# New and delete (variables)
 .v. = 1
new .v.
.v. = 2
assert {rpn .v. 2 same}                 " failed test 1.1"
assert {rpn ".v." defined}              " failed test 1.2"
delete .v.
assert {rpn .v. 1 same}                 " failed test 1.3"
assert {rpn ".v." defined}              " failed test 1.4"
delete .v.
assert {rpn ".v." defined !}            " failed test 1.4"

# New and delete (synonyms)
\v = "hi"
new \v
\v = "hello"
assert {rpn "\v" "hello" ==}            " failed test 2.1"
assert {rpn "\\v" defined}              " failed test 2.2"
delete \v
assert {rpn "\v" "hi" ==}               " failed test 2.3"
assert {rpn "\\v" defined}              " failed test 2.4"
delete \v
assert {rpn "\\v" defined !}            " failed test 2.5"

# Multi-word synonyms
\h = "Hi there buddy"
assert {rpn  \[]h    3      ==}         " failed test 3.1"
.i. = \[]h
assert {rpn  .i.     3      ==}         " failed test 3.2"
assert {rpn "\[0]h" "Hi"    ==}         " failed test 3.3"
assert {rpn "\[1]h" "there" ==}         " failed test 3.4"
assert {rpn "\[2]h" "buddy" ==}         " failed test 3.5"
\h = "Hi means \"hello\""
assert {rpn "\[0]h" "Hi"        ==}     " failed test 3.6"
assert {rpn "\[1]h" "means"     ==}     " failed test 3.7"
assert {rpn "\[2]h" "\"hello\"" ==}     " failed test 3.8"
.i. = 1
assert {rpn "\[.i.]h" "means"   ==}     " failed test 3.9"
\i = "1"
assert {rpn "\[\i]h"  "means"   ==}     " failed test 3.10"

# Setting by quoted name
set ".var." to 10
assert {rpn .var. 10 ==}                " failed test 4.1"
set "\\syn" to "hi"
assert {rpn "\syn" "hi" ==}             " failed test 4.2"

# Setting by quoted name, in a new command
‘hi pi "\\Greeting" ".Pi."’
{
    set "\.word2." to "hi"
    set "\.word3." to 3.14
}
hi pi "\\g" ".p."
assert {rpn "\g" "hi" ==}                       " failed test 5.1"
assert {rpn .p.  3.14 ==}                       " failed test 5.2"
```

17 Gri in the Press

A gentle introduction to Gri is provided in a Linux Journal article, available at `http://www2.linuxjournal.com/lj-issues/issue75/3743.html` on the web.

In March 2000, I emailed some Gri users to ask for the names of scientific journals in which they have published Gri-produced graphs. The list, which mainly arrived on Sunday in response to an email I sent Saturday (don't Gri users take a break?) is presented below, to indicate the range of Gri use in the press.

Journals in Oceanography, Atmospheric Science, Earth Science, etc.

- **Bulletin on Coastal Oceanography** (in Japanese with English abstract)
- **Fisheries Oceanography**
- **Deep-Sea Research**
- **EOS, Transactions of the American Geophysical Union**
- **Estuarine, Coastal and Shelf Science**
- **Geophysical Research Letters**
- **Journal of Atmospheric and Oceanic Technology**
- **Journal of Geophysical Research**
- **Journal of Marine Research**
- **Journal of Plankton Research**
- **Journal of Physical Oceanography**
- **Journal of the Japan Scoiety for Marine Surveys and Technology** (in Japanese with English abstract)
- **Limnology and Oceanography**
- **Marine Biology**
- **Marine Ecology Progress Series**
- **Monthly Weather Review**
- **Oceanography**
- **Paleoceanography**
- **Progress in Oceanography**
- **Quaternary Research**
- **Scientia Marina**
- **South African Journal of Marine Science**
- **Umi no Kenkyu** (Japanese version for Journal of Oceanography, in Japanese with English abstract)

Journals in Physics, Chemistry, etc.

- **Applied Physics A**
- **Journal of Fluid Mechanics**
- **Journal of Physical Chemistry B**
- **Physica D**
- **Physics of Fluids**
- **Physics A**
- **Physics Letters B**
- **Physical Review C**

- **Physical Review Letters**
- **Research on Chemical Intermediates**
- **Theoretical Computational Fluid Dynamics**

Other Journals

- **Nature**

Books, Monographs, etc.

- **The Atlas of Hawai'i** (University of Hawai'i Press)
- **Differential Equations with Applications to Biology** (American Mathematical Society)
- **Ocean Drilling Program** publications

18 Acknowledgments

Over the years, many Gri users have been kind enough to help in its development. Some have done so by sending in bug reports, others by requesting features, others by submitting patches. A few of their names are: Ivo Alxneit, Karin Bryan, Sara Bennett, Luke Blaikie, Dave Brickman, Steve Cayford, Clyde Clements, Andrew Collier, Pierre Flament, Peter Galbraith, Dave Hebert, David A. Holland, Christopher Illies, Cody Kirkpatrick, Thomas Larsen, Alejandro Lpez-Valencia, Kawamura Masao, Steve Matheson, Brian May, Ed Nather, Roman Neuhauser, Carl Osterwisch, Richard Andrew Miles Outerbridge, Tim Powers, Jinyu Sheng, Toru Suzuki, Keith Thompson, David Trueman, Wolfgang Voegeli, Jeff Whitaker, and George White.

Prime among these is Peter Galbraith, who has been a close collaborator in Gri development and in my scientific work, for well over a decade.

To all, thanks.

19 License

Gri is distributed under the GPL public license, an Open Source license that provides certain rights and freedom to users, notably the access to source code and the right to modify it and/or redistribute it. The full text of the GPL license is given below.

```
        GNU GENERAL PUBLIC LICENSE
           Version 2, June 1991

 Copyright (C) 1989, 1991 Free Software
     Foundation, Inc.  59 Temple Place, Suite
     330, Boston, MA 02111-1307 USA

 Everyone is permitted to copy and distribute
 verbatim copies of this license document, but
 changing it is not allowed.

    Preamble

  The licenses for most software are designed to
take away your freedom to share and change it.
By contrast, the GNU General Public License is
intended to guarantee your freedom to share and
change free software--to make sure the software
is free for all its users.  This General Public
License applies to most of the Free Software
Foundation's software and to any other program
whose authors commit to using it.  (Some other
Free Software Foundation software is covered by
the GNU Library General Public License instead.)
You can apply it to your programs, too.

  When we speak of free software, we are
referring to freedom, not price.  Our General
Public Licenses are designed to make sure that
you have the freedom to distribute copies of free
software (and charge for this service if you
wish), that you receive source code or can get it
if you want it, that you can change the software
or use pieces of it in new free programs; and
that you know you can do these things.

  To protect your rights, we need to make
restrictions that forbid anyone to deny you these
rights or to ask you to surrender the rights.
These restrictions translate to certain
responsibilities for you if you distribute copies
of the software, or if you modify it.

  For example, if you distribute copies of such a
program, whether gratis or for a fee, you must
give the recipients all the rights that you have.
```

You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and

modification are not covered by this License;
they are outside its scope. The act of running
the Program is not restricted, and the output
from the Program is covered only if its contents
constitute a work based on the Program
(independent of having been made by running the
Program). Whether that is true depends on what
the Program does.

1. You may copy and distribute verbatim copies
of the Program's source code as you receive it,
in any medium, provided that you conspicuously
and appropriately publish on each copy an
appropriate copyright notice and disclaimer of
warranty; keep intact all the notices that refer
to this License and to the absence of any
warranty; and give any other recipients of the
Program a copy of this License along with the
Program.

You may charge a fee for the physical act of
transferring a copy, and you may at your option
offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the
Program or any portion of it, thus forming a work
based on the Program, and copy and distribute
such modifications or work under the terms of
Section 1 above, provided that you also meet all
of these conditions:

a) You must cause the modified files to carry
prominent notices stating that you changed
the files and the date of any change.

b) You must cause any work that you
distribute or publish, that in whole or in
part contains or is derived from the Program
or any part thereof, to be licensed as a
whole at no charge to all third parties under
the terms of this License.

c) If the modified program normally reads
commands interactively when run, you must
cause it, when started running for such
interactive use in the most ordinary way, to
print or display an announcement including an
appropriate copyright notice and a notice
that there is no warranty (or else, saying
that you provide a warranty) and that users
may redistribute the program under these

conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a

```
    complete machine-readable copy of the
    corresponding source code, to be distributed
    under the terms of Sections 1 and 2 above on
    a medium customarily used for software
    interchange; or,

    c) Accompany it with the information you
    received as to the offer to distribute
    corresponding source code.  (This alternative
    is allowed only for noncommercial
    distribution and only if you received the
    program in object code or executable form
    with such an offer, in accord with Subsection
    b above.)

The source code for a work means the preferred
form of the work for making modifications to it.
For an executable work, complete source code
means all the source code for all modules it
contains, plus any associated interface
definition files, plus the scripts used to
control compilation and installation of the
executable.  However, as a special exception, the
source code distributed need not include anything
that is normally distributed (in either source or
binary form) with the major components (compiler,
kernel, and so on) of the operating system on
which the executable runs, unless that component
itself accompanies the executable.

If distribution of executable or object code is
made by offering access to copy from a designated
place, then offering equivalent access to copy
the source code from the same place counts as
distribution of the source code, even though
third parties are not compelled to copy the
source along with the object code.

  4. You may not copy, modify, sublicense, or
distribute the Program except as expressly
provided under this License.  Any attempt
otherwise to copy, modify, sublicense or
distribute the Program is void, and will
automatically terminate your rights under this
License.  However, parties who have received
copies, or rights, from you under this License
will not have their licenses terminated so long
as such parties remain in full compliance.

  5. You are not required to accept this License,
since you have not signed it.  However, nothing
```

else grants you permission to modify or
distribute the Program or its derivative works.
These actions are prohibited by law if you do not
accept this License. Therefore, by modifying or
distributing the Program (or any work based on
the Program), you indicate your acceptance of
this License to do so, and all its terms and
conditions for copying, distributing or modifying
the Program or works based on it.

6. Each time you redistribute the Program (or
any work based on the Program), the recipient
automatically receives a license from the
original licensor to copy, distribute or modify
the Program subject to these terms and
conditions. You may not impose any further
restrictions on the recipients' exercise of the
rights granted herein. You are not responsible
for enforcing compliance by third parties to this
License.

7. If, as a consequence of a court judgment or
allegation of patent infringement or for any
other reason (not limited to patent issues),
conditions are imposed on you (whether by court
order, agreement or otherwise) that contradict
the conditions of this License, they do not
excuse you from the conditions of this License.
If you cannot distribute so as to satisfy
simultaneously your obligations under this
License and any other pertinent obligations, then
as a consequence you may not distribute the
Program at all. For example, if a patent license
would not permit royalty-free redistribution of
the Program by all those who receive copies
directly or indirectly through you, then the only
way you could satisfy both it and this License
would be to refrain entirely from distribution of
the Program.

If any portion of this section is held invalid or
unenforceable under any particular circumstance,
the balance of the section is intended to apply
and the section as a whole is intended to apply
in other circumstances.

It is not the purpose of this section to induce
you to infringe any patents or other property
right claims or to contest validity of any such
claims; this section has the sole purpose of
protecting the integrity of the free software

distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for

this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a

pointer to where the full notice is found.

> <one line to give the program's name and a
> brief idea of what it does.> Copyright (C)
> yyyy <name of author>
>
> This program is free software; you can
> redistribute it and/or modify it under the
> terms of the GNU General Public License as
> published by the Free Software Foundation;
> either version 2 of the License, or (at your
> option) any later version.
>
> This program is distributed in the hope that
> it will be useful, but WITHOUT ANY WARRANTY;
> without even the implied warranty of
> MERCHANTABILITY or FITNESS FOR A PARTICULAR
> PURPOSE. See the GNU General Public License
> for more details.
>
> You should have received a copy of the GNU
> General Public License along with this
> program; if not, write to the Free Software
> Foundation, Inc., 59 Temple Place, Suite 330,
> Boston, MA 02111-1307 USA

Also add information on how to contact you by
electronic and paper mail.

If the program is interactive, make it output a
short notice like this when it starts in an
interactive mode:

> Gnomovision version 69, Copyright (C) yyyy
> name of author Gnomovision comes with
> ABSOLUTELY NO WARRANTY; for details type
> 'show w'. This is free software, and you are
> welcome to redistribute it under certain
> conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c'
should show the appropriate parts of the General
Public License. Of course, the commands you use
may be called something other than 'show w' and
'show c'; they could even be mouse-clicks or menu
items--whatever suits your program.

You should also get your employer (if you work as
a programmer) or your school, if any, to sign a
"copyright disclaimer" for the program, if

necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright
> interest in the program ‘Gnomovision’ (which
> makes passes at compilers) written by James
> Hacker.
>
> <signature of Ty Coon>, 1 April 1989
> Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Index, general

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Index of Commands

P

Q

R

S

U

W

Index of Built-in Variables and Synonyms

Short Contents

Table of Contents